AutoCorres2

Matthew Brecknell David Greenaway Johannes Hölzl
Fabian Immler Gerwin Klein Rafal Kolanski
Japheth Lim Michael Norrish Norbert Schirmer
Salomon Sickert-Zehnter Thomas Sewell Harvey Tuch
Simon Wimmer

March 17, 2025

Abstract

AutoCorres2 is a tool to facilitate the verification of C programs within Isabelle [5]. It is a fork of AutoCorres.

Contents

Chapter 1

Introduction

AutoCorres2 is a tool to facilitate the verification of C programs within Isabelle [5]. It is a fork of AutoCorres: https://trustworthy.systems/projects/OLD/autocorres/. Here some quick links into the document:

- Quickstart guide for users (chapter 25): `doc/quickstart/Chapter1_MinMax.thy`
- Background information, internals and some history of AutoCorres (chapter 26): `doc/AutoCorresInfrastructure.thy`
- C-Parser
  - Some internals (chapter 27): `c-parser/CTranslationInfrastructure.thy`
  - Original documentation (outdated) (chapter 28): The supported subset of C is extended. Moreover, the C-parser is integrated into Isabelle/ML and no standalone C-parser is supplied. The description of the design principles is still valid: `c-parser/doc/ctranslation_body.tex`

Part I

Library

Chapter 2

Misc. Definitions and Lemmas

theory *More-Lib*
imports
Introduction-AutoCorres2
HOL−Library.Prefix-Order
Word-Lib.Word-Lib-Sumo
HOL−Eisbach.Eisbach-Tools
begin

abbreviation (*input*)
split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
where
split == *case-prod*

lemma *hd-map-simp*:
$b \neq [] \Longrightarrow hd\ (map\ a\ b) = a\ (hd\ b)$
by (*rule hd-map*)

lemma *tl-map-simp*:
tl (*map a b*) = *map a* (*tl b*)
by (*induct b*,*auto*)

lemma *Collect-eq*:
$\{x.\ P\ x\} = \{x.\ Q\ x\} \longleftrightarrow (\forall x.\ P\ x = Q\ x)$
by (*rule iffI*) *auto*

lemma *iff-impI*: $[\![P \Longrightarrow Q = R]\!] \Longrightarrow (P \longrightarrow Q) = (P \longrightarrow R)$ **by** *blast*

definition
fun-app :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixr** ‹$› *10*) **where**
$f\ \$\ x \equiv f\ x$

declare *fun-app-def* [*iff*]

lemma *fun-app-cong*[*fundef-cong*]:
$[\![f\ x = f'\ x']\!] \Longrightarrow (f\ \$\ x) = (f'\ \$\ x')$
by *simp*

lemma *fun-app-apply-cong*[*fundef-cong*]:
$f\ x\ y = f'\ x'\ y' \Longrightarrow (f\ \$\ x)\ y = (f'\ \$\ x')\ y'$
by *simp*

lemma *if-apply-cong*[*fundef-cong*]:
$[\![P = P'; x = x'; P' \Longrightarrow f\ x' = f'\ x'; \neg\ P' \Longrightarrow g\ x' = g'\ x']\!]$
$\Longrightarrow (\mathit{if}\ P\ \mathit{then}\ f\ \mathit{else}\ g)\ x = (\mathit{if}\ P'\ \mathit{then}\ f'\ \mathit{else}\ g')\ x'$
by *simp*

lemma *case-prod-apply-cong*[*fundef-cong*]:
$[\![f\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)\ s = f'\ (\mathit{fst}\ p')\ (\mathit{snd}\ p')\ s']\!] \Longrightarrow \mathit{case\text{-}prod}\ f\ p\ s = \mathit{case\text{-}prod}\ f'\ p'\ s'$
by (*simp add*: *split-def*)

lemma *prod-injects*:
$(x,y) = p \Longrightarrow x = \mathit{fst}\ p \wedge y = \mathit{snd}\ p$
$p = (x,y) \Longrightarrow x = \mathit{fst}\ p \wedge y = \mathit{snd}\ p$
by *auto*

definition
pred-imp :: $('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
where
$\mathit{pred\text{-}imp}\ P\ Q \equiv \forall x.\ P\ x \longrightarrow Q\ x$

lemma *pred-impI*: $(\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \mathit{pred\text{-}imp}\ P\ Q$
by (*simp add*: *pred-imp-def*)

definition
pred-conj :: $('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool})$ (**infixl** ‹*and*› *35*)
where
$\mathit{pred\text{-}conj}\ P\ Q \equiv \lambda x.\ P\ x \wedge Q\ x$

definition
pred-disj :: $('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool})$ (**infixl** ‹*or*› *30*)
where
$\mathit{pred\text{-}disj}\ P\ Q \equiv \lambda x.\ P\ x \vee Q\ x$

definition
pred-neg :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ *bool*) (‹(‹*open-block notation*=‹*prefix pred-neg*››*not* -)› [*40*] *40*)
where
pred-neg P ≡ λ*x*. ¬ *P x*

lemma *pred-neg-simp*[*simp*]:
(*not P*) *s* ⟷ ¬ (*P s*)
by (*simp add*: *pred-neg-def*)

definition *K* ≡ λ*x y*. *x*

definition
zipWith :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list* **where**
zipWith f xs ys ≡ *map* (*case-prod f*) (*zip xs ys*)

primrec
delete :: '*a* ⇒ '*a list* ⇒ '*a list*
where
delete y [] = []
| *delete y* (*x*#*xs*) = (*if y*=*x then xs else x* # *delete y xs*)

definition
swp f ≡ λ*x y*. *f y x*

lemma *swp-apply*[*simp*]: *swp f y x* = *f x y*
by (*simp add*: *swp-def*)

primrec (*nonexhaustive*)
theRight :: '*a* + '*b* ⇒ '*b* **where**
theRight (*Inr x*) = *x*

primrec (*nonexhaustive*)
theLeft :: '*a* + '*b* ⇒ '*a* **where**
theLeft (*Inl x*) = *x*

definition
isLeft x ≡ (∃ *y*. *x* = *Inl y*)

definition
isRight x ≡ (∃ *y*. *x* = *Inr y*)

definition
const x ≡ λ*y*. *x*

primrec
opt-rel :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a option* ⇒ '*b option* ⇒ *bool*
where
opt-rel f None y = (*y* = *None*)

| *opt-rel f* (*Some x*) *y* = (∃ *y'*. *y* = *Some y'* ∧ *f x y'*)

lemma *opt-rel-None-rhs*[*simp*]:
opt-rel f x None = (*x* = *None*)
by (*cases x*, *simp-all*)

lemma *opt-rel-Some-rhs*[*simp*]:
opt-rel f x (*Some y*) = (∃ *x'*. *x* = *Some x'* ∧ *f x' y*)
by (*cases x*, *simp-all*)

lemma *tranclD2*:
(*x*, *y*) ∈ R^+ ⟹ ∃ *z*. (*x*, *z*) ∈ R^* ∧ (*z*, *y*) ∈ *R*
by (*erule tranclE*) *auto*

lemma *linorder-min-same1* [*simp*]:
(*min y x* = *y*) = (*y* ≤ (*x*::'*a*::*linorder*))
by (*auto simp*: *min-def linorder-not-less*)

lemma *linorder-min-same2* [*simp*]:
(*min x y* = *y*) = (*y* ≤ (*x*::'*a*::*linorder*))
by (*auto simp*: *min-def linorder-not-le*)

A combinator for pairing up well-formed relations. The divisor function splits the population in halves, with the True half greater than the False half, and the supplied relations control the order within the halves.

definition
wf-sum :: ('*a* ⇒ *bool*) ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set*
where
wf-sum divisor r r' ≡
(
{(*x*, *y*). ¬ *divisor x* ∧ ¬ *divisor y*} ∩ *r'*)
∪ {(*x*, *y*). ¬ *divisor x* ∧ *divisor y*}
∪ ({(*x*, *y*). *divisor x* ∧ *divisor y*} ∩ *r*)

lemma *wf-sum-wf*:
⟦ *wf r*; *wf r'* ⟧ ⟹ *wf* (*wf-sum divisor r r'*)
apply (*simp add*: *wf-sum-def*)
apply (*rule wf-Un*)+
apply (*erule wf-Int2*)
apply (*rule wf-subset*
[**where** *r*=*measure* (λ*x*. *If* (*divisor x*) *1 0*)])
apply *simp*
apply *clarsimp*
apply *blast*
apply (*erule wf-Int2*)
apply *blast*
done

abbreviation(*input*)

option-map == *map-option*

lemmas *option-map-def* = *map-option-case*

lemma *False-implies-equals* [*simp*]:
(($False \Longrightarrow P$) $\Longrightarrow$ *PROP Q*) $\equiv$ *PROP Q*
apply (*rule equal-intr-rule*)
apply (*erule meta-mp*)
apply *simp*
apply *simp*
done

lemma *split-paired-Ball*:
$(\forall x \in A.\ P\ x) = (\forall x\ y.\ (x,y) \in A \longrightarrow P\ (x,y))$
by *auto*

lemma *split-paired-Bex*:
$(\exists x \in A.\ P\ x) = (\exists x\ y.\ (x,y) \in A \wedge P\ (x,y))$
by *auto*

lemma *bexI-minus*:
$[\![\ P\ x;\ x \in A;\ x \notin B\ ]\!] \Longrightarrow \exists x \in A - B.\ P\ x$
unfolding *Bex-def* **by** *blast*

lemma *delete-remove1*:
delete x xs = *remove1 x xs*
by (*induct xs*, *auto*)

lemma *ignore-if*:
(*y and z*) *s* $\Longrightarrow$ (*if x then y else z*) *s*
by (*clarsimp simp*: *pred-conj-def*)

lemma *zipWith-Nil2* :
zipWith f xs [] = []
unfolding *zipWith-def* **by** *simp*

lemma *isRight-right-map*:
isRight (*case-sum Inl* (*Inr o f*) *v*) = *isRight v*
by (*simp add*: *isRight-def split*: *sum.split*)

lemma *zipWith-nth*:
$[\![\ n <$ *min* (*length xs*) (*length ys*) $]\!] \Longrightarrow$ *zipWith f xs ys* ! *n* = *f* (*xs* ! *n*) (*ys* ! *n*)
unfolding *zipWith-def* **by** *simp*

lemma *length-zipWith* [*simp*]:
length (*zipWith f xs ys*) = *min* (*length xs*) (*length ys*)
unfolding *zipWith-def* **by** *simp*

lemma *first-in-uptoD*:
$a \leq b \Longrightarrow (a::'a::order) \in \{a..b\}$
by *simp*

lemma *construct-singleton*:
$[\![S \neq \{\}; \forall s{\in}S.\ \forall s'.\ s \neq s' \longrightarrow s' \notin S]\!] \Longrightarrow \exists x.\ S = \{x\}$
by *blast*

lemmas *insort-com* = *insort-left-comm*

lemma *bleeding-obvious*:
$(P \Longrightarrow True) \equiv (Trueprop\ True)$
by *auto*

lemma *Some-helper*:
$x = Some\ y \Longrightarrow x \neq None$
by *simp*

lemma *in-empty-interE*:
$[\![A \cap B = \{\}; x \in A; x \in B]\!] \Longrightarrow False$
by *blast*

lemma *None-upd-eq*:
$g\ x = None \Longrightarrow g(x := None) = g$
by (*rule ext*) *simp*

lemma *exx* [*iff*]: $\exists x.\ x$ **by** *blast*
lemma *ExNot* [*iff*]: *Ex Not* **by** *blast*

lemma *cases-simp2* [*simp*]:
$((\neg P \longrightarrow Q) \wedge (P \longrightarrow Q)) = Q$
by *blast*

lemma *a-imp-b-imp-b*:
$((a \longrightarrow b) \longrightarrow b) = (a \vee b)$
by *blast*

lemma *length-neq*:
length as $\neq$ *length bs* $\Longrightarrow$ *as* $\neq$ *bs* **by** *auto*

lemma *take-neq-length*:
$[\![x \neq y; x \leq length\ as; y \leq length\ bs]\!] \Longrightarrow take\ x\ as \neq take\ y\ bs$
by (*rule length-neq*, *simp*)

lemma *eq-concat-lenD*:
$xs = ys\ @\ zs \Longrightarrow length\ xs = length\ ys + length\ zs$
by *simp*

lemma *map-upt-reindex'*: $map\ f\ [a\ ..<\ b] = map\ (\lambda n.\ f\ (n + a - x))\ [x\ ..<\ x +$

```
b − a]
  by (rule nth-equalityI; clarsimp simp: add.commute)

lemma map-upt-reindex: map f [a ..< b] = map (λn. f (n + a)) [0 ..< b − a]
  by (subst map-upt-reindex' [where x=0]) clarsimp

lemma notemptyI:
  x ∈ S ⟹ S ≠ {}
  by clarsimp

lemma setcomp-Max-has-prop:
  assumes a: P x
  shows P (Max {(x::'a::{finite,linorder}). P x})
proof −
  from a have Max {x. P x} ∈ {x. P x}
    by − (rule Max-in, auto intro: notemptyI)
  thus ?thesis by auto
qed

lemma cons-set-intro:
  lst = x # xs ⟹ x ∈ set lst
  by fastforce

lemma list-all2-conj-nth:
  assumes lall: list-all2 P as cs
  and       rl: ⋀n. ⟦P (as ! n) (cs ! n); n < length as⟧ ⟹ Q (as ! n) (cs ! n)
  shows   list-all2 (λa b. P a b ∧ Q a b) as cs
proof (rule list-all2-all-nthI)
  from lall show length as = length cs ..
next
  fix n
  assume n < length as

  show P (as ! n) (cs ! n) ∧ Q (as ! n) (cs ! n)
  proof
    from lall show P (as ! n) (cs ! n) by (rule list-all2-nthD) fact
    thus Q (as ! n) (cs ! n) by (rule rl) fact
  qed
qed

lemma list-all2-conj:
  assumes lall1: list-all2 P as cs
  and      lall2: list-all2 Q as cs
  shows   list-all2 (λa b. P a b ∧ Q a b) as cs
proof (rule list-all2-all-nthI)
  from lall1 show length as = length cs ..
next
  fix n
  assume n < length as
```

```
  show P (as ! n) (cs ! n) ∧ Q (as ! n) (cs ! n)
  proof
    from lall1 show P (as ! n) (cs ! n) by (rule list-all2-nthD) fact
    from lall2 show Q (as ! n) (cs ! n) by (rule list-all2-nthD) fact
  qed
qed

lemma all-set-into-list-all2:
  assumes lall: ∀ x ∈ set ls. P x
  and           length ls = length ls'
  shows  list-all2 (λa b. P a) ls ls'
proof (rule list-all2-all-nthI)
  fix n
  assume n < length ls
  from lall show P (ls ! n)
    by (rule bspec [OF - nth-mem]) fact
qed fact

lemma GREATEST-lessE:
  fixes x :: 'a :: order
  assumes gts: (GREATEST x. P x) < X
  and       px: P x
  and    gtst: ∃ max. P max ∧ (∀ z. P z ⟶ (z ≤ max))
  shows    x < X
proof −
  from gtst obtain max where pm: P max and g': ⋀z. P z ⟹ z ≤ max
    by auto

  hence (GREATEST x. P x) = max
    by (auto intro: Greatest-equality)

  moreover have x ≤ max using px by (rule g')

  ultimately show ?thesis using gts by simp
qed

lemma set-has-max:
  fixes ls :: ('a :: linorder) list
  assumes ls: ls ≠ []
  shows ∃ max ∈ set ls. ∀ z ∈ set ls. z ≤ max
  using ls
proof (induct ls)
  case Nil thus ?case by simp
next
  case (Cons l ls)

  show ?case
  proof (cases ls = [])
```

case *True*
thus *?thesis* **by** *simp*
next
case *False*
then obtain *max* **where** *mv*: $max \in set\ ls$ **and** *mm*: $\forall z \in set\ ls.\ z \leq max$
using *Cons.hyps*
by *auto*
show *?thesis*
proof (*cases* $max \leq l$)
case *True*
have $l \in set\ (l \# ls)$ **by** *simp*
thus *?thesis*
proof
from *mm* **show** $\forall z \in set\ (l \# ls).\ z \leq l$ **using** *True* **by** *auto*
qed
next
case *False*
from *mv* **have** $max \in set\ (l \# ls)$ **by** *simp*
thus *?thesis*
proof
from *mm* **show** $\forall z \in set\ (l \# ls).\ z \leq max$ **using** *False* **by** *auto*
qed
qed
qed
qed

lemma *True-notin-set-replicate-conv*:
$True \notin set\ ls = (ls = replicate\ (length\ ls)\ False)$
by (*induct ls*) *simp+*

lemma *Collect-singleton-eqI*:
$(\bigwedge x.\ P\ x = (x = v)) \Longrightarrow \{x.\ P\ x\} = \{v\}$
by *auto*

lemma *exEI*:
$[\![\ \exists y.\ P\ y;\ \bigwedge x.\ P\ x \Longrightarrow Q\ x\ ]\!] \Longrightarrow \exists z.\ Q\ z$
by (*rule ex-forward*)

lemma *allEI*:
assumes $\forall x.\ P\ x$
assumes $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\forall x.\ Q\ x$
using *assms* **by** (*rule all-forward*)

General lemmas that should be in the library

lemma *dom-ran*:
$x \in dom\ f \Longrightarrow the\ (f\ x) \in ran\ f$
by (*simp add*: *domD ranI*)

lemma *orthD1*:
$[\![S \cap S' = \{\};\ x \in S]\!] \Longrightarrow x \notin S'$ **by** *auto*

lemma *orthD2*:
$[\![S \cap S' = \{\};\ x \in S']\!] \Longrightarrow x \notin S$ **by** *auto*

lemma *distinct-element*:
$[\![b \cap d = \{\};\ a \in b;\ c \in d]\!] \Longrightarrow a \neq c$
by *auto*

lemma *ball-reorder*:
$(\forall x \in A.\ \forall y \in B.\ P\ x\ y) = (\forall y \in B.\ \forall x \in A.\ \ P\ x\ y)$
by *auto*

lemma *hd-map*: $ls \neq [] \Longrightarrow hd\ (map\ f\ ls) = f\ (hd\ ls)$
by (*cases ls*) *auto*

lemma *tl-map*: $tl\ (map\ f\ ls) = map\ f\ (tl\ ls)$
by (*cases ls*) *auto*

lemma *not-NilE*:
$[\![xs \neq [];\ \bigwedge x\ xs'.\ xs = x \# xs' \Longrightarrow R]\!] \Longrightarrow R$
by (*cases xs*) *auto*

lemma *length-SucE*:
$[\![length\ xs = Suc\ n;\ \bigwedge x\ xs'.\ xs = x \# xs' \Longrightarrow R]\!] \Longrightarrow R$
by (*cases xs*) *auto*

lemma *map-upt-unfold*:
assumes *ab*: $a < b$
shows $map\ f\ [a\ ..<\ b] = f\ a \#\ map\ f\ [Suc\ a\ ..<\ b]$
using *assms upt-conv-Cons* **by** *auto*

lemma *tl-nat-list-simp*:
$tl\ [a..<b] = [a + 1\ ..<b]$
by (*induct b*, *auto*)

lemma *image-Collect2*:
$case\text{-}prod\ f\ `\ \{x.\ P\ (fst\ x)\ (snd\ x)\} = \{f\ x\ y\ |x\ y.\ P\ x\ y\}$
by (*subst image-Collect*) *simp*

lemma *image-id'*:
$id\ `\ Y = Y$
by *clarsimp*

lemma *image-invert*:
assumes *r*: $f \circ g = id$
and *g*: $B = g\ `\ A$
shows $A = f\ `\ B$

by (*simp add*: *g image-comp r*)

lemma *Collect-image-fun-cong*:
assumes *rl*: $\bigwedge a.\ P\ a \Longrightarrow f\ a = g\ a$
shows $\{f\ x \mid x.\ P\ x\} = \{g\ x \mid x.\ P\ x\}$
using *rl* **by** *force*

lemma *inj-on-take*:
shows *inj-on* (*take n*) $\{x.\ drop\ n\ x = k\}$
proof (*rule inj-onI*)
fix *x y*
assume *xv*: $x \in \{x.\ drop\ n\ x = k\}$
and *yv*: $y \in \{x.\ drop\ n\ x = k\}$
and *tk*: *take n x* = *take n y*

from *xv* **have** *take n x* @ *k* = *x*
using *append-take-drop-id mem-Collect-eq* **by** *auto*
moreover from *yv tk*
have *take n x* @ *k* = *y*
using *append-take-drop-id mem-Collect-eq* **by** *auto*
ultimately show *x* = *y* **by** *simp*
qed

lemma *foldr-upd-dom*:
dom (*foldr* ($\lambda p\ ps.\ ps\ (p \mapsto f\ p)$) *as g*) = *dom g* ∪ *set as*
proof (*induct as*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as*)
show *?case*
proof (*cases a* ∈ *set as* ∨ *a* ∈ *dom g*)
case *True*
hence *ain*: *a* ∈ *dom g* ∪ *set as* **by** *auto*
hence *dom g* ∪ *set* (*a* # *as*) = *dom g* ∪ *set as* **by** *auto*
thus *?thesis* **using** *Cons* **by** *fastforce*
next
case *False*
hence *a* ∉ (*dom g* ∪ *set as*) **by** *simp*
hence *dom g* ∪ *set* (*a* # *as*) = *insert a* (*dom g* ∪ *set as*) **by** *simp*
thus *?thesis* **using** *Cons* **by** *fastforce*
qed
qed

lemma *foldr-upd-app*:
assumes *xin*: *x* ∈ *set as*
shows (*foldr* ($\lambda p\ ps.\ ps\ (p \mapsto f\ p)$) *as g*) *x* = *Some* (*f x*)
(**is** (*?f as g*) *x* = *Some* (*f x*))
using *xin*
proof (*induct as arbitrary*: *x*)

case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as*)
from *Cons.prems* **show** *?case* **by** (*subst foldr.simps*) (*auto intro*: *Cons.hyps*)
qed

lemma *foldr-upd-app-other*:
assumes *xin*: $x \notin set\ as$
shows (*foldr* ($\lambda p\ ps.\ ps\ (p \mapsto f\ p)$) *as g*) $x = g\ x$
(**is** (*?f as g*) $x = g\ x$)
using *xin*
proof (*induct as arbitrary*: *x*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as*)
from *Cons.prems* **show** *?case*
by (*subst foldr.simps*) (*auto intro*: *Cons.hyps*)
qed

lemma *foldr-upd-app-if*:
foldr ($\lambda p\ ps.\ ps(p \mapsto f\ p)$) *as g* = ($\lambda x.$ *if* $x \in set\ as$ *then Some* (*f x*) *else g x*)
by (*auto simp*: *foldr-upd-app foldr-upd-app-other*)

lemma *foldl-fun-upd-value*:
$\bigwedge Y.$ *foldl* ($\lambda f\ p.\ f(p := X\ p)$) *Y e p* = (*if* $p \in set\ e$ *then X p else Y p*)
by (*induct e*) *simp-all*

lemma *foldr-fun-upd-value*:
$\bigwedge Y.$ *foldr* ($\lambda p\ f.\ f(p := X\ p)$) *e Y p* = (*if* $p \in set\ e$ *then X p else Y p*)
by (*induct e*) *simp-all*

lemma *foldl-fun-upd-eq-foldr*:
!!*m*. *foldl* ($\lambda f\ p.\ f(p := g\ p)$) *m xs* = *foldr* ($\lambda p\ f.\ f(p := g\ p)$) *xs m*
by (*rule ext*) (*simp add*: *foldl-fun-upd-value foldr-fun-upd-value*)

lemma *Cons-eq-neq*:
$[\![\ y = x;\ x \#\ xs \neq y \#\ ys\ ]\!] \Longrightarrow xs \neq ys$
by *simp*

lemma *map-upt-append*:
assumes *lt*: $x \leq y$
and *lt2*: $a \leq x$
shows *map f* [*a* ..< *y*] = *map f* [*a* ..< *x*] @ *map f* [*x* ..< *y*]
proof (*subst map-append* [*symmetric*], *rule arg-cong* [**where** f = *map f*])
from *lt* **obtain** *k* **where** *ky*: $x + k = y$
by (*auto simp*: *le-iff-add*)

thus [*a* ..< *y*] = [*a* ..< *x*] @ [*x* ..< *y*]
using *lt2*

by (*auto intro*: *upt-add-eq-append*)
qed

lemma *Min-image-distrib*:
assumes *minf*: $\bigwedge x\ y.\ [\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow min\ (f\ x)\ (f\ y) = f\ (min\ x\ y)$
and *fa*: *finite A*
and *ane*: $A \neq \{\}$
shows $Min\ (f\ `\ A) = f\ (Min\ A)$
proof −
have *rl*: $\bigwedge F.\ [\![\ F \subseteq A;\ F \neq \{\}\ ]\!] \Longrightarrow Min\ (f\ `\ F) = f\ (Min\ F)$
proof −
fix *F*
assume *fa*: $F \subseteq A$ **and** *fne*: $F \neq \{\}$
have *finite F* **by** (*rule finite-subset*) *fact+*

thus *?thesis F*
unfolding *min-def* **using** *fa fne fa*
proof (*induct rule*: *finite-subset-induct*)
case *empty*
thus *?case* **by** *simp*
next
case (*insert x F*)
thus *?case*
by (*cases* $F = \{\}$) (*auto dest*: *Min-in intro*: *minf*)
qed
qed

show *?thesis* **by** (*rule rl* [*OF order-refl*]) *fact+*
qed

lemma *min-of-mono′*:
assumes $(f\ a \leq f\ c) = (a \leq c)$
shows $min\ (f\ a)\ (f\ c) = f\ (min\ a\ c)$
unfolding *min-def*
by (*subst if-distrib* [**where** $f = f$, *symmetric*], *rule arg-cong* [**where** $f = f$], *rule if-cong* [*OF - refl refl*]) *fact+*

lemma *nat-diff-less*:
fixes $x :: nat$
shows $[\![\ x < y + z;\ z \leq x]\!] \Longrightarrow x - z < y$
using *less-diff-conv2* **by** *blast*

lemma *take-map-Not*:
$(take\ n\ (map\ Not\ xs) = take\ n\ xs) = (n = 0 \vee xs = [])$
by (*cases n*; *simp*) (*cases xs*; *simp*)

lemma *union-trans*:
assumes *SR*: $\bigwedge x\ y\ z.\ [\![\ (x,y) \in S;\ (y,z) \in R\ ]\!] \Longrightarrow (x,z) \in S\hat{}*$

shows $(R \cup S)\hat{}* = R\hat{}* \cup R\hat{}* \; O \; S\hat{}*$
apply (*rule set-eqI*)
apply *clarsimp*
apply (*rule iffI*)
apply (*erule rtrancl-induct*; *simp*)
apply (*erule disjE*)
apply (*erule disjE*)
apply (*drule* (*1*) *rtrancl-into-rtrancl*)
apply *blast*
apply *clarsimp*
apply (*drule rtranclD* [**where** *R*=*S*])
apply (*erule disjE*)
apply *simp*
apply (*erule conjE*)
apply (*drule tranclD2*)
apply (*elim exE conjE*)
apply (*drule* (*1*) *SR*)
apply (*drule* (*1*) *rtrancl-trans*)
apply *blast*
apply (*rule disjI2*)
apply (*erule disjE*)
apply (*blast intro*: *in-rtrancl-UnI*)
apply *clarsimp*
apply (*drule* (*1*) *rtrancl-into-rtrancl*)
apply (*erule* (*1*) *relcompI*)
apply (*erule disjE*)
apply (*blast intro*: *in-rtrancl-UnI*)
apply *clarsimp*
apply (*blast intro*: *in-rtrancl-UnI rtrancl-trans*)
done

lemma *trancl-trancl*:
$(R^+)^+ = R^+$
by *auto*

Some rules for showing that the reflexive transitive closure of a relation/predicate doesn't add much if it was already transitively closed.

lemma *rtrancl-eq-reflc-trans*:
assumes *trans*: *trans X*
shows *rtrancl X* = *X* ∪ *Id*
by (*simp only*: *rtrancl-trancl-reflcl trancl-id*[*OF trans*])

lemma *rtrancl-id*:
assumes *refl*: *Id* ⊆ *X*
assumes *trans*: *trans X*
shows *rtrancl X* = *X*
using *refl rtrancl-eq-reflc-trans*[*OF trans*]
by *blast*

lemma *rtranclp-eq-reflcp-transp*:
assumes *trans*: *transp X*
shows *rtranclp X* = (λ*x y*. *X x y* ∨ *x* = *y*)
by (*simp add*: *Enum.rtranclp-rtrancl-eq fun-eq-iff*
rtrancl-eq-reflc-trans trans[*unfolded transp-trans*])

lemma *rtranclp-id*:
shows *reflp X* ⟹ *transp X* ⟹ *rtranclp X* = *X*
apply (*simp add*: *rtranclp-eq-reflcp-transp*)
apply (*auto simp*: *fun-eq-iff elim*: *reflpD*)
done

lemmas *rtranclp-id2* = *rtranclp-id*[*unfolded reflp-def transp-relcompp le-fun-def*]

lemma *if-1-0-0*:
((*if P then 1 else 0*) = (*0* :: (′*a* :: *zero-neq-one*))) = (¬ *P*)
by (*simp split*: *if-split*)

lemma *neq-Nil-lengthI*:
Suc 0 ≤ *length xs* ⟹ *xs* ≠ []
by (*cases xs*, *auto*)

lemmas *ex-with-length* = *Ex-list-of-length*

lemma *in-singleton*:
S = {*x*} ⟹ *x* ∈ *S*
by *simp*

lemma *singleton-set*:
x ∈ *set* [*a*] ⟹ *x* = *a*
by *auto*

lemma *take-drop-eqI*:
assumes *t*: *take n xs* = *take n ys*
assumes *d*: *drop n xs* = *drop n ys*
shows *xs* = *ys*
proof −
have *xs* = *take n xs* @ *drop n xs* **by** *simp*
with *t d*
have *xs* = *take n ys* @ *drop n ys* **by** *simp*
moreover
have *ys* = *take n ys* @ *drop n ys* **by** *simp*
ultimately
show *?thesis* **by** *simp*
qed

lemma *append-len2*:
zs = *xs* @ *ys* ⟹ *length xs* = *length zs* − *length ys*
by *auto*

lemma *if-flip*:
(*if* ¬*P then T else F*) = (*if P then F else T*)
by *simp*

lemma *not-in-domIff*:*f x* = *None* = (*x* ∉ *dom f*)
by *blast*

lemma *not-in-domD*:
x ∉ *dom f* ⟹ *f x* = *None*
by (*simp add*:*not-in-domIff*)

definition
graph-of f ≡ {(*x*,*y*). *f x* = *Some y*}

lemma *graph-of-None-update*:
graph-of (*f* (*p* := *None*)) = *graph-of f* − {*p*} × *UNIV*
by (*auto simp*: *graph-of-def split*: *if-split-asm*)

lemma *graph-of-Some-update*:
graph-of (*f* (*p* ↦ *v*)) = (*graph-of f* − {*p*} × *UNIV*) ∪ {(*p*,*v*)}
by (*auto simp*: *graph-of-def split*: *if-split-asm*)

lemma *graph-of-restrict-map*:
graph-of (*m* |‘ *S*) ⊆ *graph-of m*
by (*simp add*: *graph-of-def restrict-map-def subset-iff*)

lemma *graph-ofD*:
(*x*,*y*) ∈ *graph-of f* ⟹ *f x* = *Some y*
by (*simp add*: *graph-of-def*)

lemma *graph-ofI*:
m x = *Some y* ⟹ (*x*, *y*) ∈ *graph-of m*
by (*simp add*: *graph-of-def*)

lemma *graph-of-empty* :
graph-of Map.empty = {}
by (*simp add*: *graph-of-def*)

lemma *graph-of-in-ranD*: ∀ *y* ∈ *ran f*. *P y* ⟹ (*x*,*y*) ∈ *graph-of f* ⟹ *P y*
by (*auto simp*: *graph-of-def ran-def*)

lemma *graph-of-SomeD*:
⟦ *graph-of f* ⊆ *graph-of g*; *f x* = *Some y* ⟧ ⟹ *g x* = *Some y*
unfolding *graph-of-def*
by *auto*

lemma *graph-of-comp*:
⟦ *g x* = *y*; *f y* = *Some z* ⟧ ⟹ (*x*,*z*) ∈ *graph-of* (*f* ∘ *g*)

by (*auto simp*: *graph-of-def*)

lemma *in-set-zip-refl* :
(*x*,*y*) ∈ *set* (*zip xs xs*) = (*y* = *x* ∧ *x* ∈ *set xs*)
by (*induct xs*) *auto*

lemma *map-conv-upd*:
m v = *None* ⟹ *m o* (*f* (*x* := *v*)) = (*m o f*) (*x* := *None*)
by (*rule ext*) (*clarsimp simp*: *o-def*)

lemma *sum-all-ex* [*simp*]:
(∀ *a*. *x* ≠ *Inl a*) = (∃ *a*. *x* = *Inr a*)
(∀ *a*. *x* ≠ *Inr a*) = (∃ *a*. *x* = *Inl a*)
by (*metis Inr-not-Inl sum.exhaust*)+

lemma *split-distrib*: *case-prod* (λ*a b*. *T* (*f a b*)) = (λ*x*. *T* (*case-prod* (λ*a b*. *f a b*) *x*))
by (*clarsimp simp*: *split-def*)

lemma *case-sum-triv* [*simp*]:
(*case x of Inl x* ⇒ *Inl x* | *Inr x* ⇒ *Inr x*) = *x*
by (*clarsimp split*: *sum.splits*)

lemma *set-eq-UNIV*: ({*a*. *P a*} = *UNIV*) = (∀ *a*. *P a*)
by *force*

lemma *allE2*:
⟦∀ *x y*. *P x y*; *P x y* ⟹ *R*⟧ ⟹ *R*
by *blast*

lemma *allE3*: ⟦ ∀ *x y z*. *P x y z*; *P x y z* ⟹ *R* ⟧ ⟹ *R*
by *auto*

lemma *my-BallE*: ⟦ ∀ *x* ∈ *A*. *P x*; *y* ∈ *A*; *P y* ⟹ *Q* ⟧ ⟹ *Q*
by (*simp add*: *Ball-def*)

lemma *unit-Inl-or-Inr* [*simp*]:
(*a* ≠ *Inl* ()) = (*a* = *Inr* ())
(*a* ≠ *Inr* ()) = (*a* = *Inl* ())
by (*cases a*; *clarsimp*)+

lemma *disjE-L*: ⟦ *a* ∨ *b*; *a* ⟹ *R*; ⟦ ¬ *a*; *b* ⟧ ⟹ *R* ⟧ ⟹ *R*
by *blast*

lemma *disjE-R*: ⟦ *a* ∨ *b*; ⟦ ¬ *b*; *a* ⟧ ⟹ *R*; ⟦ *b* ⟧ ⟹ *R* ⟧ ⟹ *R*
by *blast*

lemma *int-max-thms*:
(*a* :: *int*) ≤ *max a b*

$(b :: int) \leq max\ a\ b$
by (*auto simp*: *max-def*)

lemma *sgn-negation* [*simp*]:
$sgn\ (-(x::int)) = -\ sgn\ x$
by (*clarsimp simp*: *sgn-if*)

lemma *sgn-sgn-nonneg* [*simp*]:
$sgn\ (a :: int) * sgn\ a \neq -1$
by (*clarsimp simp*: *sgn-if*)

lemma *inj-inj-on*:
$inj\ f \Longrightarrow inj\text{-}on\ f\ A$
by (*metis injD inj-onI*)

lemma *ex-eqI*:
$[\![\bigwedge x.\ f\ x = g\ x]\!] \Longrightarrow (\exists\, x.\ f\ x) = (\exists\, x.\ g\ x)$
by *simp*

lemma *pre-post-ex*:
$[\![\exists\, x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q\ x]\!] \Longrightarrow \exists\, x.\ Q\ x$
by *auto*

lemma *ex-conj-increase*:
$((\exists\, x.\ P\ x) \wedge Q) = (\exists\, x.\ P\ x \wedge Q)$
$(R \wedge (\exists\, x.\ S\ x)) = (\exists\, x.\ R \wedge S\ x)$
by *simp+*

lemma *all-conj-increase*:
$((\ \forall\, x.\ P\ x) \wedge Q) = (\forall\, x.\ P\ x \wedge Q)$
$(R \wedge (\forall\, x.\ S\ x)) = (\forall\, x.\ R \wedge S\ x)$
by *simp+*

lemma *Ball-conj-increase*:
$xs \neq \{\} \Longrightarrow ((\ \forall\, x \in xs.\ P\ x) \wedge Q) = (\forall\, x \in xs.\ P\ x \wedge Q)$
$xs \neq \{\} \Longrightarrow (R \wedge (\forall\, x \in xs.\ S\ x)) = (\forall\, x \in xs.\ R \wedge S\ x)$
by *auto*

lemma *disjoint-subset*:
assumes $A' \subseteq A$ **and** $A \cap B = \{\}$
shows $A' \cap B = \{\}$
using *assms* **by** *auto*

lemma *disjoint-subset2*:
assumes $B' \subseteq B$ **and** $A \cap B = \{\}$
shows $A \cap B' = \{\}$

using *assms* **by** *auto*

lemma *UN-nth-mem*:
$i < length\ xs \Longrightarrow f\ (xs\ !\ i) \subseteq (\bigcup x{\in}set\ xs.\ f\ x)$
by (*metis UN-upper nth-mem*)

lemma *Union-equal*:
$f\ `\ A = f\ `\ B \Longrightarrow (\bigcup x \in A.\ f\ x) = (\bigcup x \in B.\ f\ x)$
by *blast*

lemma *UN-Diff-disjoint*:
$i < length\ xs \Longrightarrow (A - (\bigcup x{\in}set\ xs.\ f\ x)) \cap f\ (xs\ !\ i) = \{\}$
by (*metis Diff-disjoint Int-commute UN-nth-mem disjoint-subset*)

lemma *image-list-update*:
$f\ a = f\ (xs\ !\ i)$
$\Longrightarrow f\ `\ set\ (xs\ [i := a]) = f\ `\ set\ xs$
by (*metis list-update-id map-update set-map*)

lemma *Union-list-update-id*:
$f\ a = f\ (xs\ !\ i) \Longrightarrow (\bigcup x{\in}set\ (xs\ [i := a]).\ f\ x) = (\bigcup x{\in}set\ xs.\ f\ x)$
by (*rule Union-equal*) (*erule image-list-update*)

lemma *Union-list-update-id'*:
$[\![i < length\ xs;\ \bigwedge x.\ g\ (f\ x) = g\ x]\!]$
$\Longrightarrow (\bigcup x{\in}set\ (xs\ [i := f\ (xs\ !\ i)]).\ g\ x) = (\bigcup x{\in}set\ xs.\ g\ x)$
by (*metis Union-list-update-id*)

lemma *Union-subset*:
$[\![\bigwedge x.\ x \in A \Longrightarrow (f\ x) \subseteq (g\ x)]\!] \Longrightarrow (\bigcup x \in A.\ f\ x) \subseteq (\bigcup x \in A.\ g\ x)$
by (*metis UN-mono order-refl*)

lemma *UN-sub-empty*:
$[\![list\text{-}all\ P\ xs;\ \bigwedge x.\ P\ x \Longrightarrow f\ x = g\ x]\!] \Longrightarrow (\bigcup x{\in}set\ xs.\ f\ x) - (\bigcup x{\in}set\ xs.\ g\ x)$
$= \{\}$
by (*simp add*: *Ball-set-list-all*[*symmetric*] *Union-subset*)

lemma *bij-betw-fun-updI*:
$[\![x \notin A;\ y \notin B;\ bij\text{-}betw\ f\ A\ B]\!] \Longrightarrow bij\text{-}betw\ (f(x := y))\ (insert\ x\ A)\ (insert\ y\ B)$
by (*clarsimp simp*: *bij-betw-def fun-upd-image inj-on-fun-updI split*: *if-split-asm*; *blast*)

definition
$bij\text{-}betw\text{-}map\ f\ A\ B \equiv bij\text{-}betw\ f\ A\ (Some\ `\ B)$

lemma *bij-betw-map-fun-updI*:
$[\![x \notin A;\ y \notin B;\ bij\text{-}betw\text{-}map\ f\ A\ B]\!]$

$\Longrightarrow$ *bij-betw-map* $(f(x \mapsto y))$ $(insert\ x\ A)$ $(insert\ y\ B)$
unfolding *bij-betw-map-def* **by** *clarsimp* (*erule bij-betw-fun-updI*; *clarsimp*)

lemma *bij-betw-map-imp-inj-on*:
bij-betw-map f A B $\Longrightarrow$ *inj-on f A*
by (*simp add*: *bij-betw-map-def bij-betw-imp-inj-on*)

lemma *bij-betw-empty-dom-exists*:
$r = \{\} \Longrightarrow \exists\, t.$ *bij-betw t* $\{\}$ *r*
by (*clarsimp simp*: *bij-betw-def*)

lemma *bij-betw-map-empty-dom-exists*:
$r = \{\} \Longrightarrow \exists\, t.$ *bij-betw-map t* $\{\}$ *r*
by (*clarsimp simp*: *bij-betw-map-def bij-betw-empty-dom-exists*)

lemma *funpow-add* [*simp*]:
fixes $f :: 'a \Rightarrow 'a$
shows $(f \hat{\ }\hat{\ } a)\ ((f \hat{\ }\hat{\ } b)\ s) = (f \hat{\ }\hat{\ } (a + b))\ s$
by (*metis comp-apply funpow-add*)

lemma *funpow-unfold*:
fixes $f :: 'a \Rightarrow 'a$
assumes $n > 0$
shows $f \hat{\ }\hat{\ } n = (f \hat{\ }\hat{\ } (n - 1)) \circ f$
by (*metis Suc-diff-1 assms funpow-Suc-right*)

lemma *relpow-unfold*: $n > 0 \Longrightarrow S \hat{\ }\hat{\ } n = (S \hat{\ }\hat{\ } (n - 1))\ O\ S$
by (*cases n*, *auto*)

definition
equiv-of :: $('s \Rightarrow 't) \Rightarrow ('s \times 's)$ *set*
where
equiv-of proj $\equiv \{(a, b).$ *proj a* $=$ *proj b*$\}$

lemma *equiv-of-is-equiv-relation* [*simp*]:
equiv UNIV (*equiv-of proj*)
by (*auto simp*: *equiv-of-def intro*!: *equivI refl-onI symI transI*)

lemma *in-equiv-of* [*simp*]:
$((a, b) \in$ *equiv-of f*$) \longleftrightarrow (f\ a = f\ b)$
by (*clarsimp simp*: *equiv-of-def*)

lemma *equiv-relation-to-projection*:
fixes $R :: ('a \times 'a)\ set$
assumes *equiv*: *equiv UNIV R*
shows $\exists f :: 'a \Rightarrow 'a\ set.\ \forall x\ y.\ f\ x = f\ y \longleftrightarrow (x,\ y) \in R$
apply (*rule exI* [*of* - $\lambda x.\ \{y.\ (x,\ y) \in R\}$])
apply *clarsimp*
subgoal for *x y*
apply (*cases* $(x,\ y) \in R$)
apply *clarsimp*
apply (*rule set-eqI*)
apply *clarsimp*
apply (*metis equivE sym-def trans-def equiv*)
apply (*clarsimp*)
apply (*metis UNIV-I equiv equivE mem-Collect-eq refl-on-def*)
done
done

lemma *range-constant* [*simp*]:
range (λ-. *k*) = {*k*}
by (*clarsimp simp*: *image-def*)

lemma *dom-unpack*:
dom (*map-of* (*map* ($\lambda x.$ (*f x*, *g x*)) *xs*)) = *set* (*map* ($\lambda x.$ *f x*) *xs*)
by (*simp add*: *dom-map-of-conv-image-fst image-image*)

lemma *fold-to-disj*:
fold (++) *ms a x* = *Some y* $\Longrightarrow$ ($\exists\, b \in$ *set ms*. *b x* = *Some y*) $\vee$ *a x* = *Some y*
by (*induct ms arbitrary*:*a x y*; *clarsimp*) *blast*

lemma *fold-ignore1*:
a x = *Some y* $\Longrightarrow$ *fold* (++) *ms a x* = *Some y*
by (*induct ms arbitrary*:*a x y*; *clarsimp*)

lemma *fold-ignore2*:
fold (++) *ms a x* = *None* $\Longrightarrow$ *a x* = *None*
by (*metis fold-ignore1 option.collapse*)

lemma *fold-ignore3*:
fold (++) *ms a x* = *None* $\Longrightarrow$ ($\forall\, b \in$ *set ms*. *b x* = *None*)
by (*induct ms arbitrary*:*a x*; *clarsimp*) (*meson fold-ignore2 map-add-None*)

lemma *fold-ignore4*:
b $\in$ *set ms* $\Longrightarrow$ *b x* = *Some y* $\Longrightarrow$ $\exists\, y.$ *fold* (++) *ms a x* = *Some y*
using *fold-ignore3* **by** *fastforce*

lemma *dom-unpack2*:
dom (*fold* (++) *ms Map.empty*) = $\bigcup$(*set* (*map dom ms*))
apply (*induct ms*; *clarsimp simp*:*dom-def*)
apply (*rule equalityI*; *clarsimp*)

```
 apply (drule fold-to-disj)
 apply (erule disjE)
  apply clarsimp
  apply (rename-tac b)
  apply (erule-tac x=b in ballE; clarsimp)
 apply clarsimp
apply (rule conjI)
 apply clarsimp
subgoal for - - - y
  apply (rule exI[where x = y])
  apply (erule fold-ignore1)
  done
apply clarsimp
apply (rename-tac y)
apply (erule-tac y=y in fold-ignore4; clarsimp)
done

lemma fold-ignore5:fold (++) ms a x = Some y ⟹ a x = Some y ∨ (∃ b ∈ set
ms. b x = Some y)
  by (induct ms arbitrary:a x y; clarsimp) blast

lemma dom-inter-nothing:dom f ∩ dom g = {} ⟹ ∀ x. f x = None ∨ g x = None
  by auto

lemma fold-ignore6:
  f x = None ⟹ fold (++) ms f x = fold (++) ms Map.empty x
  apply (induct ms arbitrary:f x; clarsimp simp:map-add-def)
  by (metis (no-types, lifting) fold-ignore1 option.collapse option.simps(4))

lemma fold-ignore7:
  m x = m' x ⟹ fold (++) ms m x = fold (++) ms m' x
  apply (cases m x)
   apply (frule-tac ms=ms in fold-ignore6)
   apply (cut-tac f=m' and ms=ms and x=x in fold-ignore6)
    apply clarsimp+
  apply (rename-tac a)
  apply (cut-tac ms=ms and a=m and x=x and y=a in fold-ignore1, clarsimp)
  apply (cut-tac ms=ms and a=m' and x=x and y=a in fold-ignore1; clarsimp)
  done

lemma fold-ignore8:
  fold (++) ms [x ↦ y] = (fold (++) ms Map.empty)(x ↦ y)
  apply (rule ext)
  subgoal for xa
    apply (cases xa = x)
     apply clarsimp
     apply (rule fold-ignore1)
     apply clarsimp
    apply (subst fold-ignore6; clarsimp)
```

```
    done
  done

lemma fold-ignore9:
  ⟦fold (++) ms [x ↦ y] x' = Some z; x = x'⟧ ⟹ y = z
  by (subst (asm) fold-ignore8) clarsimp

lemma fold-to-map-of:
  fold (++) (map (λx. [f x ↦ g x]) xs) Map.empty = map-of (map (λx. (f x, g x)) xs)
  apply (rule ext)
  subgoal for x
    apply (cases fold (++) (map (λx. [f x ↦ g x]) xs) Map.empty x)
     apply clarsimp
     apply (drule fold-ignore3)
     apply (clarsimp split:if-split-asm)
     apply (rule sym)
     apply (subst map-of-eq-None-iff)
     apply clarsimp
     apply (rename-tac xa)
     apply (erule-tac x=xa in ballE; clarsimp)
    apply clarsimp
    apply (frule fold-ignore5; clarsimp split:if-split-asm)
    apply (subst map-add-map-of-foldr[where m=Map.empty, simplified])
    apply (induct xs arbitrary:f g; clarsimp split:if-split)
    apply (rule conjI; clarsimp)
     apply (drule fold-ignore9; clarsimp)
    apply (cut-tac ms=map (λx. [f x ↦ g x]) xs and f=[f a ↦ g a] and x=f b in
fold-ignore6, clarsimp)
    apply auto
    done
  done

lemma if-n-0-0:
  ((if P then n else 0) ≠ 0) = (P ∧ n ≠ 0)
  by (simp split: if-split)

lemma insert-dom:
  assumes fx: f x = Some y
  shows   insert x (dom f) = dom f
  unfolding dom-def using fx by auto

lemma map-comp-subset-dom:
  dom (prj ∘ₘ f) ⊆ dom f
  unfolding dom-def
  by (auto simp: map-comp-Some-iff)

lemmas map-comp-subset-domD = subsetD [OF map-comp-subset-dom]
```

lemma *dom-map-comp*:
  $x \in dom\ (prj \circ_m f) = (\exists\, y\ z.\ f\ x = Some\ y \wedge prj\ y = Some\ z)$
  by (*fastforce simp*: *dom-def map-comp-Some-iff*)

lemma *map-option-Some-eq2*:
  $(Some\ y = map\text{-}option\ f\ x) = (\exists\, z.\ x = Some\ z \wedge f\ z = y)$
  by (*metis map-option-eq-Some*)

lemma *map-option-eq-dom-eq*:
  assumes *ome*: $map\text{-}option\ f \circ g = map\text{-}option\ f \circ g'$
  shows $dom\ g = dom\ g'$
proof (*rule set-eqI*)
  fix x
  {
    assume $x \in dom\ g$
    hence $Some\ (f\ (the\ (g\ x))) = (map\text{-}option\ f \circ g)\ x$
      by (*auto simp*: *map-option-case split*: *option.splits*)
    also have $\ldots = (map\text{-}option\ f \circ g')\ x$ **by** (*simp add*: *ome*)
    finally have $x \in dom\ g'$
      by (*auto simp*: *map-option-case split*: *option.splits*)
  } moreover
  {
    assume $x \in dom\ g'$
    hence $Some\ (f\ (the\ (g'\ x))) = (map\text{-}option\ f \circ g')\ x$
      by (*auto simp*: *map-option-case split*: *option.splits*)
    also have $\ldots = (map\text{-}option\ f \circ g)\ x$ **by** (*simp add*: *ome*)
    finally have $x \in dom\ g$
      by (*auto simp*: *map-option-case split*: *option.splits*)
  } ultimately show $(x \in dom\ g) = (x \in dom\ g')$ **by** *auto*
qed

lemma *cart-singleton-image*:
  $S \times \{s\} = (\lambda v.\ (v,\ s))\ `\ S$
  by *auto*

lemma *singleton-eq-o2s*:
  $(\{x\} = set\text{-}option\ v) = (v = Some\ x)$
  by (*cases v*, *auto*)

lemma *option-set-singleton-eq*:
  $(set\text{-}option\ opt = \{v\}) = (opt = Some\ v)$
  by (*cases opt*, *simp-all*)

lemmas *option-set-singleton-eqs*
    = *option-set-singleton-eq*
      trans[*OF eq-commute option-set-singleton-eq*]

lemma *map-option-comp2*:

map-option (*f o g*) = *map-option f o map-option g*
by (*simp add*: *option.map-comp fun-eq-iff*)

lemma *compD*:
⟦*f* ∘ *g* = *f* ∘ *g*′; *g x* = *v* ⟧ ⟹ *f* (*g*′ *x*) = *f v*
by (*metis comp-apply*)

lemma *map-option-comp-eqE*:
assumes *om*: *map-option f* ∘ *mp* = *map-option f* ∘ *mp*′
and *p1*: ⟦ *mp x* = *None*; *mp*′ *x* = *None* ⟧ ⟹ *P*
and *p2*: ⋀*v v*′. ⟦ *mp x* = *Some v*; *mp*′ *x* = *Some v*′; *f v* = *f v*′ ⟧ ⟹ *P*
shows *P*
proof (*cases mp x*)
case *None*
hence *x* ∉ *dom mp* **by** (*simp add*: *domIff*)
hence *mp*′ *x* = *None* **by** (*simp add*: *map-option-eq-dom-eq* [*OF om*] *domIff*)
with *None* **show** *?thesis* **by** (*rule p1*)
next
case (*Some v*)
hence *x* ∈ *dom mp* **by** *clarsimp*
then obtain *v*′ **where** *Some*′: *mp*′ *x* = *Some v*′ **by** (*clarsimp simp add*: *map-option-eq-dom-eq* [*OF om*])
with *Some* **show** *?thesis*
proof (*rule p2*)
show *f v* = *f v*′ **using** *Some*′ *compD* [*OF om*, *OF Some*] **by** *simp*
qed
qed

lemma *Some-the*:
x ∈ *dom f* ⟹ *f x* = *Some* (*the* (*f x*))
by *clarsimp*

lemma *map-comp-update*:
f ∘$_m$ (*g*(*x* ↦ *v*)) = (*f* ∘$_m$ *g*)(*x* := *f v*)
apply (*intro ext*)
subgoal for *y* **by** (*cases g y*; *simp*)
done

lemma *restrict-map-eqI*:
assumes *req*: *A* |‘ *S* = *B* |‘ *S*
and *mem*: *x* ∈ *S*
shows *A x* = *B x*
proof −
from *mem* **have** *A x* = (*A* |‘ *S*) *x* **by** *simp*
also have ... = (*B* |‘ *S*) *x* **using** *req* **by** *simp*
also have ... = *B x* **using** *mem* **by** *simp*
finally show *?thesis* **.**
qed

lemma *map-comp-eqI*:
assumes *dm*: *dom g* = *dom g'*
and *fg*: $\bigwedge x.\ x \in dom\ g' \Longrightarrow f\ (the\ (g'\ x)) = f\ (the\ (g\ x))$
shows $f \circ_m g = f \circ_m g'$
apply (*rule ext*)
subgoal for *x*
apply (*cases x* ∈ *dom g*)
apply (*frule subst* [*OF dm*])
apply (*clarsimp split*: *option.splits*)
apply (*frule domI* [**where** *m* = *g'*])
apply (*drule fg*)
apply *simp*
apply (*frule subst* [*OF dm*])
apply *clarsimp*
apply (*drule not-sym*)
apply (*clarsimp simp*: *map-comp-Some-iff*)
done
done

definition
modify-map m p f ≡ *m* (*p* := *map-option f* (*m p*))

lemma *modify-map-id*:
modify-map m p id = *m*
by (*auto simp add*: *modify-map-def map-option-case split*: *option.splits*)

lemma *modify-map-addr-com*:
assumes *com*: *x* ≠ *y*
shows *modify-map* (*modify-map m x g*) *y f* = *modify-map* (*modify-map m y f*) *x g*
by (*rule ext*) (*simp add*: *modify-map-def map-option-case com split*: *option.splits*)

lemma *modify-map-dom* :
dom (*modify-map m p f*) = *dom m*
unfolding *modify-map-def* **by** (*auto simp*: *dom-def*)

lemma *modify-map-None*:
m x = *None* ⟹ *modify-map m x f* = *m*
by (*rule ext*) (*simp add*: *modify-map-def*)

lemma *modify-map-ndom* :
x ∉ *dom m* ⟹ *modify-map m x f* = *m*
by (*rule modify-map-None*) *clarsimp*

lemma *modify-map-app*:
(*modify-map m p f*) *q* = (*if p* = *q then map-option f* (*m p*) *else m q*)
unfolding *modify-map-def* **by** *simp*

lemma *modify-map-apply*:
m p = Some x ⟹ modify-map m p f = m (p ↦ f x)
by (*simp add*: *modify-map-def*)

lemma *modify-map-com*:
assumes *com*: *⋀x. f (g x) = g (f x)*
shows *modify-map (modify-map m x g) y f = modify-map (modify-map m y f) x g*
using *assms* **by** (*auto simp*: *modify-map-def map-option-case split*: *option.splits*)

lemma *modify-map-comp*:
modify-map m x (f o g) = modify-map (modify-map m x g) x f
by (*rule ext*) (*simp add*: *modify-map-def option.map-comp*)

lemma *modify-map-exists-eq*:
(∃ cte. modify-map m p' f p= Some cte) = (∃ cte. m p = Some cte)
by (*auto simp*: *modify-map-def split*: *if-splits*)

lemma *modify-map-other*:
p ≠ q ⟹ (modify-map m p f) q = (m q)
by (*simp add*: *modify-map-app*)

lemma *modify-map-same*:
modify-map m p f p = map-option f (m p)
by (*simp add*: *modify-map-app*)

lemma *next-update-is-modify*:
⟦ m p = Some cte'; cte = f cte' ⟧ ⟹ (m(p ↦ cte)) = modify-map m p f
unfolding *modify-map-def* **by** *simp*

lemma *nat-power-minus-less*:
a < 2 ^ (x − n) ⟹ (a :: nat) < 2 ^ x
by (*erule order-less-le-trans*) *simp*

lemma *neg-rtranclI*:
*⟦ x ≠ y; (x, y) ∉ R⁺ ⟧ ⟹ (x, y) ∉ R**
by (*meson rtranclD*)

lemma *neg-rtrancl-into-trancl*:
¬ (x, y) ∈ R ⟹ ¬ (x, y) ∈ R⁺*
by (*erule contrapos-nn*, *erule trancl-into-rtrancl*)

lemma *set-neqI*:
⟦ x ∈ S; x ∉ S' ⟧ ⟹ S ≠ S'
by *clarsimp*

lemma *set-pair-UN*:
{x. P x} = ⋃ ((λxa. {xa} × {xb. P (xa, xb)}) ‘ {xa. ∃ xb. P (xa, xb)})
by *fastforce*

lemma *singleton-elemD*: $S = \{x\} \Longrightarrow x \in S$
by *simp*

lemma *singleton-eqD*: $A = \{x\} \Longrightarrow x \in A$
by *blast*

lemma *ball-ran-fun-updI*:
$[\![\forall v \in ran\ m.\ P\ v; \forall v.\ y = Some\ v \longrightarrow P\ v]\!] \Longrightarrow \forall v \in ran\ (m\ (x := y)).\ P\ v$
by (*auto simp add*: *ran-def*)

lemma *ball-ran-eq*:
$(\forall y \in ran\ m.\ P\ y) = (\forall x\ y.\ m\ x = Some\ y \longrightarrow P\ y)$
by (*auto simp add*: *ran-def*)

lemma *cart-helper*:
$(\{\} = \{x\} \times S) = (S = \{\})$
by *blast*

lemmas *converse-trancl-induct'* = *converse-trancl-induct* [*consumes 1*, *case-names base step*]

lemma *disjCI2*: $(\neg P \Longrightarrow Q) \Longrightarrow P \vee Q$ **by** *blast*

lemma *insert-UNIV* :
insert x UNIV = *UNIV*
by *blast*

lemma *not-singletonE*:
$[\![\forall p.\ S \neq \{p\}; S \neq \{\}; \bigwedge p\ p'.\ [\![p \neq p'; p \in S; p' \in S]\!] \Longrightarrow R]\!] \Longrightarrow R$
by *blast*

lemma *not-singleton-oneE*:
$[\![\forall p.\ S \neq \{p\}; p \in S; \bigwedge p'.\ [\![p \neq p'; p' \in S]\!] \Longrightarrow R]\!] \Longrightarrow R$
using *not-singletonE* **by** *fastforce*

lemma *ball-ran-modify-map-eq*:
$[\![\forall v.\ m\ x = Some\ v \longrightarrow P\ (f\ v) = P\ v]\!]$
$\Longrightarrow (\forall v \in ran\ (modify\text{-}map\ m\ x\ f).\ P\ v) = (\forall v \in ran\ m.\ P\ v)$
by (*auto simp*: *modify-map-def ball-ran-eq*)

lemma *eq-singleton-redux*:
$[\![S = \{x\}]\!] \Longrightarrow x \in S$
by *simp*

lemma *if-eq-elem-helperE*:
$[\![x \in (if\ P\ then\ S\ else\ S'); [\![P; x \in S]\!] \Longrightarrow a = b; [\![\neg P; x \in S']\!] \Longrightarrow a = c]\!]$
$\Longrightarrow a = (if\ P\ then\ b\ else\ c)$

by *fastforce*

lemma *if-option-Some*:
(*if P then None else Some x*) = *Some y*) = (¬*P* ∧ *x* = *y*)
by *simp*

lemma *insert-minus-eq*:
x ∉ *A* ⟹ *A* − *S* = (*A* − (*S* − {*x*}))
by *auto*

lemma *modify-map-K-D*:
modify-map m p (λ*x*. *y*) *p'* = *Some v* ⟹ (*m* (*p* ↦ *y*)) *p'* = *Some v*
by (*simp add*: *modify-map-def split*: *if-split-asm*)

lemma *tranclE2*:
assumes *trancl*: (*a*, *b*) ∈ *r*$^+$
and *base*: (*a*, *b*) ∈ *r* ⟹ *P*
and *step*: ⋀*c*. ⟦(*a*, *c*) ∈ *r*; (*c*, *b*) ∈ *r*$^+$⟧ ⟹ *P*
shows *P*
using *trancl base step*
proof −
note *rl* = *converse-trancl-induct* [**where** *P* = λ*x*. *x* = *a* ⟶ *P*]
from *trancl* **have** *a* = *a* ⟶ *P*
by (*rule rl*, (*iprover intro*: *base step*)+)
thus *?thesis* **by** *simp*
qed

lemmas *tranclE2'* = *tranclE2* [*consumes 1*, *case-names base trancl*]

lemma *weak-imp-cong*:
⟦ *P* = *R*; *Q* = *S* ⟧ ⟹ (*P* ⟶ *Q*) = (*R* ⟶ *S*)
by *simp*

lemma *Collect-Diff-restrict-simp*:
T − {*x* ∈ *T*. *Q x*} = *T* − {*x*. *Q x*}
by (*auto intro*: *Collect-cong*)

lemma *Collect-Int-pred-eq*:
{*x* ∈ *S*. *P x*} ∩ {*x* ∈ *T*. *P x*} = {*x* ∈ (*S* ∩ *T*). *P x*}
by (*simp add*: *Collect-conj-eq* [*symmetric*] *conj-comms*)

lemma *Collect-restrict-predR*:
{*x*. *P x*} ∩ *T* = {} ⟹ {*x*. *P x*} ∩ {*x* ∈ *T*. *Q x*} = {}
by (*fastforce simp*: *disjoint-iff-not-equal*)

lemma *Diff-Un2*:
assumes *emptyad*: *A* ∩ *D* = {}
and *emptybc*: *B* ∩ *C* = {}
shows (*A* ∪ *B*) − (*C* ∪ *D*) = (*A* − *C*) ∪ (*B* − *D*)

proof −
have $(A \cup B) - (C \cup D) = (A \cup B - C) \cap (A \cup B - D)$
by (*rule Diff-Un*)
also have $\ldots = ((A - C) \cup B) \cap (A \cup (B - D))$ **using** *emptyad emptybc*
by (*simp add*: *Un-Diff Diff-triv*)
also have $\ldots = (A - C) \cup (B - D)$
proof −
have $(A - C) \cap (A \cup (B - D)) = A - C$ **using** *emptyad emptybc*
by (*metis Diff-Int2 Diff-Int-distrib2 inf-sup-absorb*)
moreover
have $B \cap (A \cup (B - D)) = B - D$ **using** *emptyad emptybc*
by (*metis Int-Diff Un-Diff Un-Diff-Int Un-commute Un-empty-left inf-sup-absorb*)
ultimately show *?thesis*
by (*simp add*: *Int-Un-distrib2*)
qed
finally show *?thesis* **.**
qed

lemma *ballEI*:
$[\![\forall x \in S.\ Q\ x;\ \bigwedge x.\ [\![x \in S;\ Q\ x]\!] \Longrightarrow P\ x]\!] \Longrightarrow \forall x \in S.\ P\ x$
by *auto*

lemma *dom-if-None*:
$dom\ (\lambda x.\ if\ P\ x\ then\ None\ else\ f\ x) = dom\ f - \{x.\ P\ x\}$
by (*simp add*: *dom-def*) *fastforce*

lemma *restrict-map-Some-iff*:
$((m\ |`\ S)\ x = Some\ y) = (m\ x = Some\ y \wedge x \in S)$
by (*cases* $x \in S$, *simp-all*)

lemma *context-case-bools*:
$[\![\bigwedge v.\ P\ v \Longrightarrow R\ v;\ [\![\neg\ P\ v;\ \bigwedge v.\ P\ v \Longrightarrow R\ v]\!] \Longrightarrow R\ v]\!] \Longrightarrow R\ v$
by (*cases P v*, *simp-all*)

lemma *inj-on-fun-upd-strongerI*:
$[\![inj\text{-}on\ f\ A;\ y \notin f\ `\ (A - \{x\})]\!] \Longrightarrow inj\text{-}on\ (f(x := y))\ A$
by (*fastforce simp*: *inj-on-def*)

lemma *less-handy-casesE*:
$[\![m < n;\ m = 0 \Longrightarrow R;\ \bigwedge m'\ n'.\ [\![n = Suc\ n';\ m = Suc\ m';\ m < n]\!] \Longrightarrow R]\!]$
$\Longrightarrow R$
by (*cases n*; *simp*) (*cases m*; *simp*)

lemma *subset-drop-Diff-strg*:
$(A \subseteq C) \longrightarrow (A - B \subseteq C)$
by *blast*

lemma *inj-case-bool*:
$inj\ (case\text{-}bool\ a\ b) = (a \neq b)$

by (*auto dest*: *inj-onD*[**where** *x*=*True* **and** *y*=*False*] *intro*: *inj-onI split*: *bool.split-asm*)

lemma *foldl-fun-upd*:
foldl (λ*s r*. *s* (*r* := *g r*)) *f rs* = (λ*x*. *if x* ∈ *set rs then g x else f x*)
by (*induct rs arbitrary*: *f*) (*auto simp*: *fun-eq-iff*)

lemma *all-rv-choice-fn-eq-pred*:
⟦ ⋀*rv*. *P rv* ⟹ ∃*fn*. *f rv* = *g fn* ⟧ ⟹ ∃*fn*. ∀*rv*. *P rv* ⟶ *f rv* = *g* (*fn rv*)
apply (*rule exI*[**where** *x*=λ*rv*. *SOME h*. *f rv* = *g h*])
apply (*clarsimp split*: *if-split*)
by (*meson someI-ex*)

lemma *ex-const-function*:
∃*f*. ∀*s*. *f* (*f*′ *s*) = *v*
by *force*

lemma *if-Const-helper*:
If P (*Con x*) (*Con y*) = *Con* (*If P x y*)
by (*simp split*: *if-split*)

lemmas *if-Some-helper* = *if-Const-helper*[**where** *Con*=*Some*]

lemma *expand-restrict-map-eq*:
(*m* |‘ *S* = *m*′ |‘ *S*) = (∀*x*. *x* ∈ *S* ⟶ *m x* = *m*′ *x*)
by (*simp add*: *fun-eq-iff restrict-map-def split*: *if-split*)

lemma *disj-imp-rhs*:
(*P* ⟹ *Q*) ⟹ (*P* ∨ *Q*) = *Q*
by *blast*

lemma *remove1-filter*:
distinct xs ⟹ *remove1 x xs* = *filter* (λ*y*. *x* ≠ *y*) *xs*
by (*induct xs*) (*auto intro*!: *filter-True* [*symmetric*])

lemma *Int-Union-empty*:
(⋀*x*. *x* ∈ *S* ⟹ *A* ∩ *P x* = {}) ⟹ *A* ∩ (⋃*x* ∈ *S*. *P x*) = {}
by *auto*

lemma *UN-Int-empty*:
(⋀*x*. *x* ∈ *S* ⟹ *P x* ∩ *T* = {}) ⟹ (⋃*x* ∈ *S*. *P x*) ∩ *T* = {}
by *auto*

lemma *disjointI*:
⟦⋀*x y*. ⟦ *x* ∈ *A*; *y* ∈ *B* ⟧ ⟹ *x* ≠ *y* ⟧ ⟹ *A* ∩ *B* = {}
by *auto*

lemma *UN-disjointI*:
assumes *rl*: ⋀*x y*. ⟦ *x* ∈ *A*; *y* ∈ *B* ⟧ ⟹ *P x* ∩ *Q y* = {}
shows (⋃*x* ∈ *A*. *P x*) ∩ (⋃*x* ∈ *B*. *Q x*) = {}

by (*auto dest*: *rl*)

lemma *UN-set-member*:
assumes *sub*: $A \subseteq (\bigcup x \in S.\ P\ x)$
and *nz*: $A \neq \{\}$
shows $\exists x \in S.\ P\ x \cap A \neq \{\}$
proof −
from *nz* **obtain** *z* **where** *zA*: $z \in A$ **by** *fastforce*
with *sub* **obtain** *x* **where** $x \in S$ **and** $z \in P\ x$ **by** *auto*
hence $P\ x \cap A \neq \{\}$ **using** *zA* **by** *auto*
thus *?thesis* **using** *sub nz* **by** *auto*
qed

lemma *append-Cons-cases* [*consumes 1*, *case-names pre mid post*]:
$[\![(x,\ y) \in set\ (as\ @\ b\ \#\ bs);$
$(x,\ y) \in set\ as \Longrightarrow R;$
$[\![(x,\ y) \notin set\ as;\ (x,\ y) \notin set\ bs;\ (x,\ y) = b]\!] \Longrightarrow R;$
$(x,\ y) \in set\ bs \Longrightarrow R]\!] \Longrightarrow R$
by *auto*

lemma *cart-singletons*:
$\{a\} \times \{b\} = \{(a,\ b)\}$
by *blast*

lemma *disjoint-subset-neg1*:
$[\![\ B \cap C = \{\};\ A \subseteq B;\ A \neq \{\}\ ]\!] \Longrightarrow \neg\ A \subseteq C$
by *auto*

lemma *disjoint-subset-neg2*:
$[\![\ B \cap C = \{\};\ A \subseteq C;\ A \neq \{\}\ ]\!] \Longrightarrow \neg\ A \subseteq B$
by *auto*

lemma *iffE2*:
$[\![\ P = Q;\ [\![\ P;\ Q\ ]\!] \Longrightarrow R;\ [\![\ \neg\ P;\ \neg\ Q\ ]\!] \Longrightarrow R\ ]\!] \Longrightarrow R$
by *blast*

lemma *list-case-If*:
$(case\ xs\ of\ []\ \Rightarrow P\ |\ -\ \Rightarrow Q) = (if\ xs = []\ then\ P\ else\ Q)$
by (*rule list.case-eq-if*)

lemma *remove1-Nil-in-set*:
$[\![\ remove1\ x\ xs = []; xs \neq []\ ]\!] \Longrightarrow x \in set\ xs$
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *remove1-empty*:
$(remove1\ v\ xs = []) = (xs = [v] \vee xs = [])$
by (*cases xs*; *simp*)

lemma *set-remove1*:

$x \in set\ (remove1\ y\ xs) \Longrightarrow x \in set\ xs$
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *If-rearrage*:
(*if P then if Q then x else y else z*) = (*if P ∧ Q then x else if P then y else z*)
by *simp*

lemma *disjI2-strg*:
$Q \longrightarrow (P \vee Q)$
by *simp*

lemma *eq-imp-strg*:
$P\ t \longrightarrow (t = s \longrightarrow P\ s)$
by *clarsimp*

lemma *if-both-strengthen*:
$P \wedge Q \longrightarrow (if\ G\ then\ P\ else\ Q)$
by *simp*

lemma *if-both-strengthen2*:
$P\ s \wedge Q\ s \longrightarrow (if\ G\ then\ P\ else\ Q)\ s$
by *simp*

lemma *if-swap*:
(*if P then Q else R*) = (*if ¬P then R else Q*) **by** *simp*

lemma *imp-consequent*:
$P \longrightarrow Q \longrightarrow P$ **by** *simp*

lemma *list-case-helper*:
$xs \neq [] \Longrightarrow case\text{-}list\ f\ g\ xs = g\ (hd\ xs)\ (tl\ xs)$
by (*cases xs*, *simp-all*)

lemma *list-cons-rewrite*:
$(\forall x\ xs.\ L = x \#\ xs \longrightarrow P\ x\ xs) = (L \neq [] \longrightarrow P\ (hd\ L)\ (tl\ L))$
by (*auto simp*: *neq-Nil-conv*)

lemma *list-not-Nil-manip*:
$[\![\ xs = y \#\ ys;\ case\ xs\ of\ [] \Rightarrow False \mid (y \#\ ys) \Rightarrow P\ y\ ys\ ]\!] \Longrightarrow P\ y\ ys$
by *simp*

lemma *ran-ball-triv*:
$\bigwedge P\ m\ S.\ [\![\ \forall x \in (ran\ S).\ P\ x\ ;\ m \in (ran\ S)\ ]\!] \Longrightarrow P\ m$
by *blast*

lemma *singleton-tuple-cartesian*:
$(\{(a,\ b)\} = S \times T) = (\{a\} = S \wedge \{b\} = T)$
$(S \times T = \{(a,\ b)\}) = (\{a\} = S \wedge \{b\} = T)$
by *blast+*

lemma *strengthen-ignore-if*:
A s ∧ *B s* ⟶ (*if P then A else B*) *s*
by *clarsimp*

lemma *case-sum-True* :
(*case r of Inl a* ⇒ *True* | *Inr b* ⇒ *f b*) = (∀ *b*. *r* = *Inr b* ⟶ *f b*)
by (*cases r*) *auto*

lemma *sym-ex-elim*:
F x = *y* ⟹ ∃ *x*. *y* = *F x*
by *auto*

lemma *tl-drop-1* :
tl xs = *drop 1 xs*
by (*simp add*: *drop-Suc*)

lemma *upt-lhs-sub-map*:
[*x* ..< *y*] = *map* ((+) *x*) [*0* ..< *y* − *x*]
by (*induct y*) (*auto simp*: *Suc-diff-le*)

lemma *upto-0-to-4* :
[*0*..<*4*] = *0* # [*1*..<*4*]
by (*subst upt-rec*) *simp*

lemma *disjEI*:
⟦ *P* ∨ *Q*; *P* ⟹ *R*; *Q* ⟹ *S* ⟧
⟹ *R* ∨ *S*
by *fastforce*

lemma *dom-fun-upd2*:
s x = *Some z* ⟹ *dom* (*s* (*x* ↦ *y*)) = *dom s*
by (*simp add*: *insert-absorb domI*)

lemma *foldl-True* :
foldl (∨) *True bs*
by (*induct bs*) *auto*

lemma *image-set-comp*:
f ‘ {*g x* | *x*. *Q x*} = (*f* ∘ *g*) ‘ {*x*. *Q x*}
by *fastforce*

lemma *mutual-exE*:
⟦ ∃ *x*. *P x*; ⋀*x*. *P x* ⟹ *Q x* ⟧ ⟹ ∃ *x*. *Q x*
by *blast*

lemma *nat-diff-eq*:
fixes *x* :: *nat*
shows ⟦ *x* − *y* = *x* − *z*; *y* < *x*⟧ ⟹ *y* = *z*

by *arith*

lemma *comp-upd-simp*:
$(f \circ (g\ (x := y))) = ((f \circ g)\ (x := f\ y))$
by (*rule fun-upd-comp*)

lemma *dom-option-map*:
$dom\ (map\text{-}option\ f\ o\ m) = dom\ m$
by (*rule dom-map-option-comp*)

lemma *drop-imp*:
$P \Longrightarrow (A \longrightarrow P) \wedge (B \longrightarrow P)$ **by** *blast*

lemma *inj-on-fun-updI2*:
$[\![\ inj\text{-}on\ f\ A;\ y \notin f\ `\ (A - \{x\})\ ]\!] \Longrightarrow inj\text{-}on\ (f(x := y))\ A$
by (*rule inj-on-fun-upd-strongerI*)

lemma *inj-on-fun-upd-elsewhere*:
$x \notin S \Longrightarrow inj\text{-}on\ (f\ (x := y))\ S = inj\text{-}on\ f\ S$
by (*simp add*: *inj-on-def*) *blast*

lemma *not-Some-eq-tuple*:
$(\forall\, y\ z.\ x \neq Some\ (y,\ z)) = (x = None)$
by (*cases x*, *simp-all*)

lemma *ran-option-map*:
$ran\ (map\text{-}option\ f\ o\ m) = f\ `\ ran\ m$
by (*auto simp add*: *ran-def*)

lemma *All-less-Ball*:
$(\forall\, x < n.\ P\ x) = (\forall\, x{\in}\{..<\ n\}.\ P\ x)$
by *fastforce*

lemma *Int-image-empty*:
$[\![\ \bigwedge x\ y.\ f\ x \neq g\ y\ ]\!]$
$\Longrightarrow f\ `\ S \cap g\ `\ T = \{\}$
by *auto*

lemma *Max-prop*:
$[\![\ Max\ S \in S \Longrightarrow P\ (Max\ S);\ (S :: ('a :: \{finite,\ linorder\})\ set) \neq \{\}\ ]\!] \Longrightarrow P\ (Max\ S)$
by *auto*

lemma *Min-prop*:
$[\![\ Min\ S \in S \Longrightarrow P\ (Min\ S);\ (S :: ('a :: \{finite,\ linorder\})\ set) \neq \{\}\ ]\!] \Longrightarrow P\ (Min\ S)$
by *auto*

lemma *findSomeD*:

find P xs = Some x ⟹ P x ∧ x ∈ set xs
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *findNoneD*:
find P xs = None ⟹ ∀ x ∈ set xs. ¬P x
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *dom-upd*:
dom (λ*x*. *if x = y then None else f x*) = *dom f* − {*y*}
by (*rule set-eqI*) (*auto split*: *if-split-asm*)

definition
is-inv :: ('*a* ⇀ '*b*) ⇒ ('*b* ⇀ '*a*) ⇒ *bool* **where**
is-inv f g ≡ *ran f* = *dom g* ∧ (∀ *x y*. *f x* = *Some y* ⟶ *g y* = *Some x*)

lemma *is-inv-NoneD*:
assumes *g x* = *None*
assumes *is-inv f g*
shows *x* ∉ *ran f*
proof −
from *assms*
have *x* ∉ *dom g* **by** (*auto simp*: *ran-def*)
moreover
from *assms*
have *ran f* = *dom g*
by (*simp add*: *is-inv-def*)
ultimately
show *?thesis* **by** *simp*
qed

lemma *is-inv-SomeD*:
⟦ *f x* = *Some y*; *is-inv f g* ⟧ ⟹ *g y* = *Some x*
by (*simp add*: *is-inv-def*)

lemma *is-inv-com*:
is-inv f g ⟹ *is-inv g f*
apply (*unfold is-inv-def*)
apply *safe*
apply (*clarsimp simp*: *ran-def dom-def set-eq-iff*)
apply (*rename-tac a*)
apply (*erule-tac x*=*a* **in** *allE*)
apply *clarsimp*
apply (*clarsimp simp*: *ran-def dom-def set-eq-iff*)
apply *blast*
apply (*clarsimp simp*: *ran-def dom-def set-eq-iff*)
apply (*rename-tac x y*)
apply (*erule-tac x*=*x* **in** *allE*)
apply *clarsimp*

done

lemma *is-inv-inj*:
is-inv f g ⟹ *inj-on f* (*dom f*)
apply (*frule is-inv-com*)
apply (*clarsimp simp*: *inj-on-def*)
apply (*drule* (*1*) *is-inv-SomeD*)
apply (*auto dest*: *is-inv-SomeD*)
done

lemma *ran-upd'*:
⟦*inj-on f* (*dom f*); *f y* = *Some z*⟧ ⟹ *ran* (*f* (*y* := *None*)) = *ran f* − {*z*}
by (*force simp*: *ran-def inj-on-def dom-def intro*!: *set-eqI*)

lemma *is-inv-None-upd*:
⟦ *is-inv f g*; *g x* = *Some y*⟧ ⟹ *is-inv* (*f*(*y* := *None*)) (*g*(*x* := *None*))
apply (*subst is-inv-def*)
apply (*clarsimp simp*: *dom-upd*)
apply (*drule is-inv-SomeD*, *erule is-inv-com*)
apply (*frule is-inv-inj*)
apply (*auto simp*: *ran-upd' is-inv-def dest*: *is-inv-SomeD is-inv-inj*)
done

lemma *is-inv-inj2*:
is-inv f g ⟹ *inj-on g* (*dom g*)
using *is-inv-com is-inv-inj* **by** *blast*

Map inversion (implicitly assuming injectivity).

definition
the-inv-map m = (λ*s*. *if s*∈*ran m then Some* (*THE x*. *m x* = *Some s*) *else None*)

Map inversion can be expressed by function inversion.

lemma *the-inv-map-def2*:
the-inv-map m = (*Some* ∘ *the-inv-into* (*dom m*) (*the* ∘ *m*)) |‘ (*ran m*)
apply (*rule ext*)
apply (*clarsimp simp*: *the-inv-map-def the-inv-into-def dom-def*)
apply (*rule arg-cong*[**where** *f*=*The*])
apply (*rule ext*)
apply *auto*
done

The domain of a function composition with Some is the universal set.

lemma *dom-comp-Some*[*simp*]: *dom* (*comp Some f*) = *UNIV* **by** (*simp add*: *dom-def*)

Assuming injectivity, map inversion produces an inversive map.

lemma *is-inv-the-inv-map*:
inj-on m (*dom m*) ⟹ *is-inv m* (*the-inv-map m*)
apply (*simp add*: *is-inv-def*)
apply (*intro conjI allI impI*)

apply (*simp add*: *the-inv-map-def2*)
apply (*auto simp add*: *the-inv-map-def inj-on-def dom-def intro*: *ranI*)
done

lemma *the-the-inv-mapI*:
inj-on m (*dom m*) $\Longrightarrow$ *m x* = *Some y* $\Longrightarrow$ *the* (*the-inv-map m y*) = *x*
by (*auto simp*: *the-inv-map-def ran-def inj-on-def dom-def*)

lemma *eq-restrict-map-None*:
restrict-map m A x = *None* $\longleftrightarrow$ *x* $\notin$ (*A* $\cap$ *dom m*)
by (*auto simp*: *restrict-map-def split*: *if-split-asm*)

lemma *eq-the-inv-map-None*[*simp*]: *the-inv-map m x* = *None* $\longleftrightarrow$ *x*$\notin$*ran m*
by (*simp add*: *the-inv-map-def2 eq-restrict-map-None*)

lemma *is-inv-unique*:
is-inv f g $\Longrightarrow$ *is-inv f h* $\Longrightarrow$ *g*=*h*
apply (*rule ext*)
subgoal for *x*
apply (*clarsimp simp*: *is-inv-def dom-def Collect-eq ran-def*)
apply (*drule-tac x*=*x* **in** *spec*)+
apply (*cases g x*, *clarsimp*+)
done
done

lemma *range-convergence1*:
⟦ $\forall z.\ x < z \wedge z \leq y \longrightarrow P\ z$; $\forall z > y.\ P\ (z :: 'a :: linorder)$ ⟧ $\Longrightarrow$ $\forall z > x.\ P\ z$
using *not-le* **by** *blast*

lemma *range-convergence2*:
⟦ $\forall z.\ x < z \wedge z \leq y \longrightarrow P\ z$; $\forall z.\ z > y \wedge z < w \longrightarrow P\ (z :: 'a :: linorder)$ ⟧
$\Longrightarrow$ $\forall z.\ z > x \wedge z < w \longrightarrow P\ z$
using *range-convergence1*[**where** *P*=$\lambda z.\ z < w \longrightarrow P\ z$ **and** *x*=*x* **and** *y*=*y*]
by *auto*

lemma *zip-upt-Cons*:
a < *b* $\Longrightarrow$ *zip* [*a* ..< *b*] (*x* # *xs*) = (*a*, *x*) # *zip* [*Suc a* ..< *b*] *xs*
by (*simp add*: *upt-conv-Cons*)

lemma *map-comp-eq*:
f $\circ_m$ *g* = *case-option None f* $\circ$ *g*
apply (*rule ext*)
subgoal for *x* **by** (*cases g x*, *auto*)
done

lemma *dom-If-Some*:
dom ($\lambda x.$ *if x* $\in$ *S then Some v else f x*) = (*S* $\cup$ *dom f*)
by (*auto split*: *if-split*)

lemma *foldl-fun-upd-const*:
foldl (λ*s x*. *s*(*f x* := *v*)) *s xs*
= (λ*x*. *if x* ∈ *f* ‘ *set xs then v else s x*)
by (*induct xs arbitrary*: *s*) *auto*

lemma *foldl-id*:
foldl (λ*s x*. *s*) *s xs* = *s*
by (*induct xs*) *auto*

lemma *SucSucMinus*: *2* ≤ *n* ⟹ *Suc* (*Suc* (*n* − *2*)) = *n* **by** *arith*

lemma *ball-to-all*:
(⋀*x*. (*x* ∈ *A*) = (*P x*)) ⟹ (∀ *x* ∈ *A*. *B x*) = (∀ *x*. *P x* ⟶ *B x*)
by *blast*

lemma *case-option-If*:
case-option P (λ*x*. *Q*) *v* = (*if v* = *None then P else Q*)
by *clarsimp*

lemma *case-option-If2*:
case-option P Q v = *If* (*v* ≠ *None*) (*Q* (*the v*)) *P*
by (*simp split*: *option.split*)

lemma *if3-fold*:
(*if P then x else if Q then y else x*) = (*if P* ∨ ¬ *Q then x else y*)
by *simp*

lemma *rtrancl-insert*:
assumes *x-new*: ⋀*y*. (*x*,*y*) ∉ *R*
shows *R*^* `` *insert x S* = *insert x* (*R*^* `` *S*)
proof −
have *R*^* `` *insert x S* = *R*^* `` ({*x*} ∪ *S*) **by** *simp*
also
have *R*^* `` ({*x*} ∪ *S*) = *R*^* `` {*x*} ∪ *R*^* `` *S*
by (*subst Image-Un*) *simp*
also
have *R*^* `` {*x*} = {*x*}
by (*meson Image-closed-trancl Image-singleton-iff subsetI x-new*)
finally
show *?thesis* **by** *simp*
qed

lemma *ran-del-subset*:
y ∈ *ran* (*f* (*x* := *None*)) ⟹ *y* ∈ *ran f*
by (*auto simp*: *ran-def split*: *if-split-asm*)

lemma *trancl-sub-lift*:
assumes *sub*: ⋀*p p*′. (*p*,*p*′) ∈ *r* ⟹ (*p*,*p*′) ∈ *r*′
shows (*p*,*p*′) ∈ *r*^+ ⟹ (*p*,*p*′) ∈ *r*′^+

by (*fastforce intro*: *trancl-mono sub*)

lemma *trancl-step-lift*:
assumes *x-step*: $\bigwedge p\ p'.\ (p,p') \in r' \Longrightarrow (p,p') \in r \vee (p = x \wedge p' = y)$
assumes *y-new*: $\bigwedge p'.\ \neg(y,p') \in r$
shows $(p,p') \in r'^{\frown}+ \Longrightarrow (p,p') \in r^{\frown}+ \vee ((p,x) \in r^{\frown}+ \wedge p' = y) \vee (p = x \wedge p' = y)$
apply (*erule trancl-induct*)
apply (*drule x-step*)
apply *fastforce*
apply (*erule disjE*)
apply (*drule x-step*)
apply (*erule disjE*)
apply (*drule trancl-trans*, *drule r-into-trancl*, *assumption*)
apply *blast*
apply *fastforce*
apply (*fastforce simp*: *y-new dest*: *x-step*)
done

lemma *rtrancl-simulate-weak*:
assumes *r*: $(x,z) \in R^*$
assumes *s*: $\bigwedge y.\ (x,y) \in R \Longrightarrow (y,z) \in R^* \Longrightarrow (x,y) \in R' \wedge (y,z) \in R'^*$
shows $(x,z) \in R'^*$
apply (*rule converse-rtranclE*[*OF r*])
apply *simp*
apply (*frule* (*1*) *s*)
apply *clarsimp*
by (*rule converse-rtrancl-into-rtrancl*)

lemma *list-case-If2*:
case-list f g xs = *If* (*xs* = []) *f* (*g* (*hd xs*) (*tl xs*))
by (*simp split*: *list.split*)

lemma *length-ineq-not-Nil*:
length xs > *n* $\Longrightarrow$ *xs* $\neq$ []
length xs $\geq$ *n* $\Longrightarrow$ *n* $\neq$ *0* $\longrightarrow$ *xs* $\neq$ []
$\neg$ *length xs* < *n* $\Longrightarrow$ *n* $\neq$ *0* $\longrightarrow$ *xs* $\neq$ []
$\neg$ *length xs* $\leq$ *n* $\Longrightarrow$ *xs* $\neq$ []
by *auto*

lemma *numeral-eqs*:
2 = *Suc* (*Suc 0*)
3 = *Suc* (*Suc* (*Suc 0*))
4 = *Suc* (*Suc* (*Suc* (*Suc 0*)))
5 = *Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))
6 = *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))
by *simp*+

lemma *psubset-singleton*:

$(S \subset \{x\}) = (S = \{\})$
by *blast*

lemma *length-takeWhile-ge*:
$length\ (takeWhile\ f\ xs) = n \Longrightarrow length\ xs = n \vee (length\ xs > n \wedge \neg\ f\ (xs\ !\ n))$
by (*induct xs arbitrary*: *n*) (*auto split*: *if-split-asm*)

lemma *length-takeWhile-le*:
$\neg\ f\ (xs\ !\ n) \Longrightarrow length\ (takeWhile\ f\ xs) \leq n$
apply (*induct xs arbitrary*: *n*; *simp*)
subgoal for *x xs n* **by** (*cases n*; *simp*)
done

lemma *length-takeWhile-gt*:
$n < length\ (takeWhile\ f\ xs)$
$\Longrightarrow (\exists\ ys\ zs.\ length\ ys = Suc\ n \wedge xs = ys\ @\ zs \wedge takeWhile\ f\ xs = ys\ @$
$takeWhile\ f\ zs)$
apply (*induct xs arbitrary*: *n*; *simp split*: *if-split-asm*)
subgoal for *a xs n*
apply (*cases n*; *simp*)
apply (*rule exI*[**where** *x*=[*a*]])
apply *simp*
apply (*erule meta-allE*, *drule*(*1*) *meta-mp*)
apply *clarsimp*
subgoal for - *ys zs*
apply (*rule exI*[**where** *x*=*a* # *ys*])
apply *simp*
done
done
done

lemma *hd-drop-conv-nth2*:
$n < length\ xs \Longrightarrow hd\ (drop\ n\ xs) = xs\ !\ n$
by (*rule hd-drop-conv-nth*) *clarsimp*

lemma *map-upt-eq-vals-D*:
$[\![\ map\ f\ [0\ ..<\ n] = ys;\ m < length\ ys\ ]\!] \Longrightarrow f\ m = ys\ !\ m$
by *clarsimp*

lemma *length-le-helper*:
$[\![\ n \leq length\ xs;\ n \neq 0\ ]\!] \Longrightarrow xs \neq [] \wedge n - 1 \leq length\ (tl\ xs)$
by (*cases xs*, *simp-all*)

lemma *all-ex-eq-helper*:
$(\forall\ v.\ (\exists\ v'.\ v = f\ v' \wedge P\ v\ v') \longrightarrow Q\ v)$
$= (\forall\ v'.\ P\ (f\ v')\ v' \longrightarrow Q\ (f\ v'))$
by *auto*

lemma *nat-less-cases'*:

$(x::nat) < y \Longrightarrow x = y - 1 \vee x < y - 1$
by *auto*

lemma *less-numeral-nat-iff-disj*:
$(n::nat) < numeral\ m \longleftrightarrow n = numeral\ m - 1 \vee n < numeral\ m - 1$
apply *clarsimp*
using *less-SucE numeral-eq-Suc* **by** *presburger*

lemma *filter-to-shorter-upto*:
$n \leq m \Longrightarrow filter\ (\lambda x.\ x < n)\ [0\ ..<\ m] = [0\ ..<\ n]$
by (*induct m*) (*auto elim*: *le-SucE*)

lemma *in-emptyE*: $[\![\ A = \{\};\ x \in A\ ]\!] \Longrightarrow P$ **by** *blast*

lemma *Ball-emptyI*:
$S = \{\} \Longrightarrow (\forall x \in S.\ P\ x)$
by *simp*

lemma *allfEI*:
$[\![\ \forall x.\ P\ x;\ \bigwedge x.\ P\ (f\ x) \Longrightarrow Q\ x\ ]\!] \Longrightarrow \forall x.\ Q\ x$
by *fastforce*

lemma *cart-singleton-empty2*:
$(\{x\} \times S = \{\}) = (S = \{\})$
$(\{\} = S \times \{e\}) = (S = \{\})$
by *auto*

lemma *cases-simp-conj*:
$((P \longrightarrow Q) \wedge (\neg\ P \longrightarrow Q) \wedge R) = (Q \wedge R)$
by *fastforce*

lemma *domE* :
$[\![\ x \in dom\ m;\ \bigwedge r.\ [\![m\ x = Some\ r]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by *clarsimp*

lemma *dom-eqD*:
$[\![\ f\ x = Some\ v;\ dom\ f = S\ ]\!] \Longrightarrow x \in S$
by *clarsimp*

lemma *exception-set-finite1*:
$finite\ \{x.\ P\ x\} \Longrightarrow finite\ \{x.\ (x = y \longrightarrow Q\ x) \wedge P\ x\}$
by (*simp add*: *Collect-conj-eq*)

lemma *exception-set-finite2*:
$finite\ \{x.\ P\ x\} \Longrightarrow finite\ \{x.\ x \neq y \longrightarrow P\ x\}$
by (*simp add*: *imp-conv-disj*)

lemmas *exception-set-finite* = *exception-set-finite1 exception-set-finite2*

lemma *exfEI*:
$[\![\exists x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q\ (f\ x)\ ]\!] \Longrightarrow \exists x.\ Q\ x$
by *fastforce*

lemma *Collect-int-vars*:
$\{s.\ P\ rv\ s\} \cap \{s.\ rv = xf\ s\} = \{s.\ P\ (xf\ s)\ s\} \cap \{s.\ rv = xf\ s\}$
by *auto*

lemma *if-0-1-eq*:
$((if\ P\ then\ 1\ else\ 0) = (case\ Q\ of\ True \Rightarrow of\text{-}nat\ 1 \mid False \Rightarrow of\text{-}nat\ 0)) = (P = Q)$
by (*simp split*: *if-split bool.split*)

lemma *modify-map-exists-cte* :
$(\exists cte.\ modify\text{-}map\ m\ p\ f\ p' = Some\ cte) = (\exists cte.\ m\ p' = Some\ cte)$
by (*simp add*: *modify-map-def*)

lemma *dom-eqI*:
assumes *c1*: $\bigwedge x\ y.\ P\ x = Some\ y \Longrightarrow \exists y.\ Q\ x = Some\ y$
and *c2*: $\bigwedge x\ y.\ Q\ x = Some\ y \Longrightarrow \exists y.\ P\ x = Some\ y$
shows $dom\ P = dom\ Q$
unfolding *dom-def* **by** (*auto simp*: *c1 c2*)

lemma *dvd-reduce-multiple*:
fixes $k :: nat$
shows $(k\ dvd\ k * m + n) = (k\ dvd\ n)$
by (*induct m*) (*auto simp*: *add-ac*)

lemma *image-iff2*:
$inj\ f \Longrightarrow f\ x \in f\ `\ S = (x \in S)$
by (*rule inj-image-mem-iff*)

lemma *map-comp-restrict-map-Some-iff*:
$((g \circ_m (m \mid` S))\ x = Some\ y) = ((g \circ_m m)\ x = Some\ y \wedge x \in S)$
by (*auto simp add*: *map-comp-Some-iff restrict-map-Some-iff*)

lemma *range-subsetD*:
fixes $a :: 'a :: order$
shows $[\![\{a..b\} \subseteq \{c..d\};\ a \leq b\ ]\!] \Longrightarrow c \leq a \wedge b \leq d$
by *simp*

lemma *case-option-dom*:
$(case\ f\ x\ of\ None \Rightarrow a \mid Some\ v \Rightarrow b\ v) = (if\ x \in dom\ f\ then\ b\ (the\ (f\ x))\ else\ a)$
by (*auto split*: *option.split*)

lemma *contrapos-imp*:
$P \longrightarrow Q \Longrightarrow \neg\ Q \longrightarrow \neg\ P$
by *clarsimp*

lemma *filter-eq-If*:
  distinct xs $\Longrightarrow$ *filter* ($\lambda v.\ v = x$) *xs* = (*if x* $\in$ *set xs then* [*x*] *else* [])
  by (*induct xs*) *auto*

lemma (**in** *semigroup-add*) *foldl-assoc*:
shows *foldl* (+) (*x*+*y*) *zs* = *x* + (*foldl* (+) *y zs*)
  by (*induct zs arbitrary*: *y*) (*simp-all add*:*add.assoc*)

lemma (**in** *monoid-add*) *foldl-absorb0*:
shows *x* + (*foldl* (+) *0 zs*) = *foldl* (+) *x zs*
  by (*induct zs*) (*simp-all add*:*foldl-assoc*)

lemma *foldl-conv-concat*:
  foldl (@) *xs xss* = *xs* @ *concat xss*
proof (*induct xss arbitrary*: *xs*)
  case *Nil* **show** *?case* **by** *simp*
next
  interpret *monoid-add* (@) [] **proof qed** *simp-all*
  case *Cons* **then show** *?case* **by** (*simp add*: *foldl-absorb0*)
qed

lemma *foldl-concat-concat*:
  foldl (@) [] (*xs* @ *ys*) = *foldl* (@) [] *xs* @ *foldl* (@) [] *ys*
  by (*simp add*: *foldl-conv-concat*)

lemma *foldl-does-nothing*:
  $[\![\bigwedge x.\ x \in set\ xs \Longrightarrow f\ s\ x = s\ ]\!] \Longrightarrow foldl\ f\ s\ xs = s$
  by (*induct xs*) *auto*

lemma *foldl-use-filter*:
  $[\![\bigwedge v\ x.\ [\![\neg\ g\ x;\ x \in set\ xs\ ]\!] \Longrightarrow f\ v\ x = v\ ]\!] \Longrightarrow foldl\ f\ v\ xs = foldl\ f\ v$ (*filter g xs*)
  by (*induct xs arbitrary*: *v*) *auto*

lemma *map-comp-update-lift*:
  assumes *fv*: *f v* = *Some v'*
  shows (*f* $\circ_m$ (*g*(*ptr* $\mapsto$ *v*))) = ((*f* $\circ_m$ *g*)(*ptr* $\mapsto$ *v'*))
  by (*simp add*: *fv map-comp-update*)

lemma *restrict-map-cong*:
  assumes *sv*: *S* = *S'*
  and *rl*: $\bigwedge p.\ p \in S' \Longrightarrow mp\ p = mp'\ p$
  shows *mp* |` *S* = *mp'* |` *S'*
  using *expand-restrict-map-eq rl sv* **by** *auto*

lemma *case-option-over-if*:
  case-option P Q (*if G then None else Some v*)
      = (*if G then P else Q v*)
  case-option P Q (*if G then Some v else None*)

```
      = (if G then Q v else P)
  by (simp split: if-split)+

lemma map-length-cong:
  ⟦ length xs = length ys; ⋀x y. (x, y) ∈ set (zip xs ys) ⟹ f x = g y ⟧
    ⟹ map f xs = map g ys
  apply atomize
  apply (erule rev-mp, erule list-induct2)
   apply auto
  done

lemma take-min-len:
  take (min (length xs) n) xs = take n xs
  by (simp add: min-def)

lemmas interval-empty = atLeastatMost-empty-iff

lemma fold-and-false[simp]:
  ¬(fold (∧) xs False)
  apply clarsimp
  apply (induct xs)
   apply simp
  apply simp
  done

lemma fold-and-true:
  fold (∧) xs True ⟹ ∀ i < length xs. xs ! i
  apply clarsimp
  apply (induct xs)
   apply simp
  subgoal for a xs i
    by (cases i = 0; cases a; auto)
  done

lemma fold-or-true[simp]:
  fold (∨) xs True
  by (induct xs, simp+)

lemma fold-or-false:
  ¬(fold (∨) xs False) ⟹ ∀ i < length xs. ¬(xs ! i)
  apply (induct xs, simp+)
  subgoal for a xs
    apply (cases a, simp+)
    apply (rule allI)
    subgoal for i by (cases i = 0, simp+)
    done
  done
```

2.1 Take, drop, zip, *list-all* etc rules

method *two-induct* **for** *xs ys* =
((*induct xs arbitrary*: *ys*; *simp?*), (*case-tac ys*; *simp*)*?*)

lemma *map-fst-zip-prefix*:
map fst (*zip xs ys*) ≤ *xs*
by (*two-induct xs ys*)

lemma *map-snd-zip-prefix*:
map snd (*zip xs ys*) ≤ *ys*
by (*two-induct xs ys*)

lemma *nth-upt-0* [*simp*]:
i < *length xs* ⟹ [*0*..<*length xs*] ! *i* = *i*
by *simp*

lemma *take-insert-nth*:
i < *length xs*⟹ *insert* (*xs* ! *i*) (*set* (*take i xs*)) = *set* (*take* (*Suc i*) *xs*)
by (*subst take-Suc-conv-app-nth*, *assumption*, *fastforce*)

lemma *zip-take-drop*:
⟦*n* < *length xs*; *length ys* = *length xs*⟧ ⟹
zip xs (*take n ys* @ *a* # *drop* (*Suc n*) *ys*) =
zip (*take n xs*) (*take n ys*) @ (*xs* ! *n*, *a*) # *zip* (*drop* (*Suc n*) *xs*) (*drop* (*Suc n*) *ys*)
by (*subst id-take-nth-drop*, *assumption*, *simp*)

lemma *take-nth-distinct*:
⟦*distinct xs*; *n* < *length xs*; *xs* ! *n* ∈ *set* (*take n xs*)⟧ ⟹ *False*
by (*fastforce simp*: *distinct-conv-nth in-set-conv-nth*)

lemma *take-drop-append*:
drop a xs = *take b* (*drop a xs*) @ *drop* (*a* + *b*) *xs*
by (*metis append-take-drop-id drop-drop add.commute*)

lemma *drop-take-drop*:
drop a (*take* (*b* + *a*) *xs*) @ *drop* (*b* + *a*) *xs* = *drop a xs*
by (*metis add.commute take-drop take-drop-append*)

lemma *not-prefixI*:
⟦ *xs* ≠ *ys*; *length xs* = *length ys*⟧ ⟹ ¬ *xs* ≤ *ys*
by (*auto elim*: *prefixE*)

lemma *map-fst-zip′*:
length xs ≤ *length ys* ⟹ *map fst* (*zip xs ys*) = *xs*
by (*metis length-map length-zip map-fst-zip-prefix min-absorb1 not-prefixI*)

lemma *zip-take-triv*:

n ≥ *length bs* ⟹ *zip* (*take n as*) *bs* = *zip as bs*
apply (*induct bs arbitrary*: *n as*; *simp*)
subgoal for *a bs n as* **by** (*cases n*; *cases as*; *simp*)
done

lemma *zip-take-triv2*:
length as ≤ *n* ⟹ *zip as* (*take n bs*) = *zip as bs*
apply (*induct as arbitrary*: *n bs*; *simp*)
subgoal for *a bs n as* **by** (*cases n*; *cases as*; *simp*)
done

lemma *zip-take-length*:
zip xs (*take* (*length xs*) *ys*) = *zip xs ys*
by (*metis order-refl zip-take-triv2*)

lemma *zip-singleton*:
ys ≠ [] ⟹ *zip* [*a*] *ys* = [(*a*, *ys* ! *0*)]
by (*cases ys*, *simp-all*)

lemma *zip-append-singleton*:
⟦*i* = *length xs*; *length xs* < *length ys*⟧ ⟹ *zip* (*xs* @ [*a*]) *ys* = (*zip xs ys*) @ [(*a*,*ys* ! *i*)]
by (*induct xs*; *cases ys*; *simp*)
(*clarsimp simp*: *zip-append1 zip-take-length zip-singleton*)

lemma *ranE*:
⟦ *v* ∈ *ran f*; ⋀*x*. *f x* = *Some v* ⟹ *R*⟧ ⟹ *R*
by (*auto simp*: *ran-def*)

lemma *ran-map-option-restrict-eq*:
⟦ *x* ∈ *ran* (*map-option f o g*); *x* ∉ *ran* (*map-option f o* (*g* |` (− {*y*}))) ⟧
⟹ ∃ *v*. *g y* = *Some v* ∧ *f v* = *x*
apply (*clarsimp simp*: *elim*!: *ranE*)
subgoal for *w z*
apply (*cases w* = *y*)
apply *clarsimp*
apply (*erule notE*, *rule ranI*[**where** *a*=*w*])
apply (*simp add*: *restrict-map-def*)
done
done

lemma *map-of-zip-range*:
⟦*length xs* = *length ys*; *distinct xs*⟧ ⟹ (λ*x*. (*the* (*map-of* (*zip xs ys*) *x*))) ` *set xs* = *set ys*
apply (*clarsimp simp*: *image-def*)
apply (*subst ran-map-of-zip* [*symmetric*, **where** *xs*=*xs* **and** *ys*=*ys*]; *simp?*)
apply (*clarsimp simp*: *ran-def*)
apply (*rule equalityI*)
apply *clarsimp*

apply (*rename-tac x*)
apply (*frule-tac x=x* **in** *map-of-zip-is-Some*; *fastforce*)
apply (*clarsimp simp*: *set-zip*)
by (*metis domI dom-map-of-zip nth-mem ranE ran-map-of-zip option.sel*)

lemma *map-zip-fst*:
length xs = *length ys* ⟹ *map* (λ(*x*, *y*). *f x*) (*zip xs ys*) = *map f xs*
by (*two-induct xs ys*)

lemma *map-zip-fst'*:
length xs ≤ *length ys* ⟹ *map* (λ(*x*, *y*). *f x*) (*zip xs ys*) = *map f xs*
by (*metis length-map map-fst-zip' map-zip-fst zip-map-fst-snd*)

lemma *map-zip-snd*:
length xs = *length ys* ⟹ *map* (λ(*x*, *y*). *f y*) (*zip xs ys*) = *map f ys*
by (*two-induct xs ys*)

lemma *map-zip-snd'*:
length ys ≤ *length xs* ⟹ *map* (λ(*x*, *y*). *f y*) (*zip xs ys*) = *map f ys*
by (*two-induct xs ys*)

lemma *map-of-zip-tuple-in*:
⟦(*x*, *y*) ∈ *set* (*zip xs ys*); *distinct xs*⟧ ⟹ *map-of* (*zip xs ys*) *x* = *Some y*
by (*two-induct xs ys*) (*auto intro*: *in-set-zipE*)

lemma *in-set-zip1*:
(*x*, *y*) ∈ *set* (*zip xs ys*) ⟹ *x* ∈ *set xs*
by (*erule in-set-zipE*)

lemma *in-set-zip2*:
(*x*, *y*) ∈ *set* (*zip xs ys*) ⟹ *y* ∈ *set ys*
by (*erule in-set-zipE*)

lemma *map-zip-snd-take*:
map (λ(*x*, *y*). *f y*) (*zip xs ys*) = *map f* (*take* (*length xs*) *ys*)
apply (*subst map-zip-snd'* [*symmetric*, **where** *xs=xs* **and** *ys=take* (*length xs*) *ys*], *simp*)
apply (*subst zip-take-length* [*symmetric*], *simp*)
done

lemma *map-of-zip-is-index*:
⟦*length xs* = *length ys*; *x* ∈ *set xs*⟧ ⟹ ∃ *i*. (*map-of* (*zip xs ys*)) *x* = *Some* (*ys* ! *i*)
apply (*induct rule*: *list-induct2*; *simp*)
apply (*rule conjI*; *clarsimp*)
apply (*metis nth-Cons-0*)
apply (*metis nth-Cons-Suc*)
done

lemma *map-of-zip-take-update*:
⟦*i* < *length xs*; *length xs* ≤ *length ys*; *distinct xs*⟧
⟹ (*map-of* (*zip* (*take i xs*) *ys*))(*xs* ! *i* ↦ (*ys* ! *i*)) = *map-of* (*zip* (*take* (*Suc i*) *xs*) *ys*)
apply (*rule ext*)
subgoal for *x*
apply (*cases x*=*xs* ! *i*; *clarsimp*)
apply (*rule map-of-is-SomeI*[*symmetric*])
apply (*simp add*: *map-fst-zip*′)
apply (*force simp add*: *set-zip*)
apply (*clarsimp simp*: *take-Suc-conv-app-nth zip-append-singleton map-add-def split*: *option.splits*)
done
done

lemma *map-of-zip-is-Some*′:
length xs ≤ *length ys* ⟹ (*x* ∈ *set xs*) = (∃ *y*. *map-of* (*zip xs ys*) *x* = *Some y*)
apply (*subst zip-take-length*[*symmetric*])
apply (*rule map-of-zip-is-Some*)
by (*metis length-take min-absorb2*)

lemma *map-of-zip-inj*:
⟦*distinct xs*; *distinct ys*; *length xs* = *length ys*⟧
⟹ *inj-on* (λ*x*. (*the* (*map-of* (*zip xs ys*) *x*))) (*set xs*)
apply (*clarsimp simp*: *inj-on-def*)
apply (*subst* (*asm*) *map-of-zip-is-Some*, *assumption*)+
apply *clarsimp*
apply (*clarsimp simp*: *set-zip*)
by (*metis nth-eq-iff-index-eq*)

lemma *map-of-zip-inj*′:
⟦*distinct xs*; *distinct ys*; *length xs* ≤ *length ys*⟧
⟹ *inj-on* (λ*x*. (*the* (*map-of* (*zip xs ys*) *x*))) (*set xs*)
apply (*subst zip-take-length*[*symmetric*])
apply (*erule map-of-zip-inj*, *simp*)
by (*metis length-take min-absorb2*)

lemma *list-all-nth*:
⟦*list-all P xs*; *i* < *length xs*⟧ ⟹ *P* (*xs* ! *i*)
by (*metis list-all-length*)

lemma *list-all-update*:
⟦*list-all P xs*; *i* < *length xs*; ⋀*x*. *P x* ⟹ *P* (*f x*)⟧
⟹ *list-all P* (*xs* [*i* := *f* (*xs* ! *i*)])
by (*metis length-list-update list-all-length nth-list-update*)

lemma *list-allI*:
⟦*list-all P xs*; ⋀*x*. *P x* ⟹ *P*′ *x*⟧ ⟹ *list-all P*′ *xs*

by (*metis list-all-length*)

lemma *list-all-imp-filter*:
list-all (λ*x*. *f x* ⟶ *g x*) *xs* = *list-all* (λ*x*. *g x*) [*x*←*xs* . *f x*]
by (*fastforce simp*: *Ball-set-list-all*[*symmetric*])

lemma *list-all-imp-filter2*:
list-all (λ*x*. *f x* ⟶ *g x*) *xs* = *list-all* (λ*x*. ¬*f x*) [*x*←*xs* . (λ*x*. ¬*g x*) *x*]
by (*fastforce simp*: *Ball-set-list-all*[*symmetric*])

lemma *list-all-imp-chain*:
⟦*list-all* (λ*x*. *f x* ⟶ *g x*) *xs*; *list-all* (λ*x*. *f' x* ⟶ *f x*) *xs*⟧
⟹ *list-all* (λ*x*. *f' x* ⟶ *g x*) *xs*
by (*clarsimp simp*: *Ball-set-list-all* [*symmetric*])

lemma *inj-Pair*:
inj-on (*Pair x*) *S*
by (*rule inj-onI*, *simp*)

lemma *inj-on-split*:
inj-on f S ⟹ *inj-on* (λ*x*. (*z*, *f x*)) *S*
by (*auto simp*: *inj-on-def*)

lemma *split-state-strg*:
(∃ *x*. *f s* = *x* ∧ *P x s*) ⟶ *P* (*f s*) *s* **by** *clarsimp*

lemma *theD*:
⟦*the* (*f x*) = *y*; *x* ∈ *dom f* ⟧ ⟹ *f x* = *Some y*
by (*auto simp add*: *dom-def*)

lemma *bspec-split*:
⟦ ∀(*a*, *b*) ∈ *S*. *P a b*; (*a*, *b*) ∈ *S* ⟧ ⟹ *P a b*
by *fastforce*

lemma *set-zip-same*:
set (*zip xs xs*) = *Id* ∩ (*set xs* × *set xs*)
by (*induct xs*) *auto*

lemma *ball-ran-updI*:
(∀ *x* ∈ *ran m*. *P x*) ⟹ *P v* ⟹ (∀ *x* ∈ *ran* (*m* (*y* ↦ *v*)). *P x*)
by (*auto simp add*: *ran-def*)

lemma *not-psubset-eq*:
⟦ ¬ *A* ⊂ *B*; *A* ⊆ *B* ⟧ ⟹ *A* = *B*
by *blast*

lemma *set-as-imp*:
$(A \cap P \cup B \cap -P) = \{s.\ (s \in P \longrightarrow s \in A) \wedge (s \notin P \longrightarrow s \in B)\}$
by *auto*

lemma *in-image-op-plus*:
$(x + y \in (+)\ x\ `\ S) = ((y :: 'a :: ring) \in S)$
by (*simp add*: *image-def*)

lemma *insert-subtract-new*:
$x \notin S \Longrightarrow (insert\ x\ S - S) = \{x\}$
by *auto*

lemmas *zip-is-empty* = *zip-eq-Nil-iff*

lemma *minus-Suc-0-lt*:
$a \neq 0 \Longrightarrow a - Suc\ 0 < a$
by *simp*

lemma *fst-last-zip-upt*:
$zip\ [0\ ..<\ m]\ xs \neq [] \Longrightarrow$
$fst\ (last\ (zip\ [0\ ..<\ m]\ xs)) = (if\ length\ xs < m\ then\ length\ xs - 1\ else\ m - 1)$
apply (*subst last-conv-nth*, *assumption*)
apply (*simp only*: *One-nat-def*)
apply (*subst nth-zip*; *simp*)
apply (*rule order-less-le-trans*[*OF minus-Suc-0-lt*]; *simp*)
apply (*rule order-less-le-trans*[*OF minus-Suc-0-lt*]; *simp*)
done

lemma *neq-into-nprefix*:
$[\![\ x \neq take\ (length\ x)\ y\ ]\!] \Longrightarrow \neg\ x \leq y$
by (*clarsimp simp*: *prefix-def less-eq-list-def*)

lemma *suffix-eqI*:
$[\![\ suffix\ xs\ as;\ suffix\ xs\ bs;\ length\ as = length\ bs;$
$take\ (length\ as - length\ xs)\ as \leq take\ (length\ bs - length\ xs)\ bs]\!] \Longrightarrow as = bs$
by (*clarsimp elim*!: *prefixE suffixE*)

lemma *suffix-Cons-mem*:
$suffix\ (x\ \#\ xs)\ as \Longrightarrow x \in set\ as$
by (*metis in-set-conv-decomp suffix-def*)

lemma *distinct-imply-not-in-tail*:
$[\![\ distinct\ list;\ suffix\ (y\ \#\ ys)\ list]\!] \Longrightarrow y \notin set\ ys$
by (*clarsimp simp*:*suffix-def*)

lemma *list-induct-suffix* [*case-names Nil Cons*]:
assumes *nilr*: $P\ []$

```
  and    consr: ⋀x xs. ⟦P xs; suffix (x # xs) as ⟧ ⟹ P (x # xs)
  shows  P as
proof −
  define as′ where as′ == as

  have suffix as as′ unfolding as′-def by simp
  then show ?thesis
  proof (induct as)
    case Nil show ?case by fact
  next
    case (Cons x xs)

    show ?case
    proof (rule consr)
      from Cons.prems show suffix (x # xs) as unfolding as′-def .
      then have suffix xs as′ by (auto dest: suffix-ConsD simp: as′-def)
      then show P xs using Cons.hyps by simp
    qed
  qed
qed
```

Parallel etc. and lemmas for list prefix

```
lemma prefix-induct [consumes 1, case-names Nil Cons]:
  fixes prefix
  assumes np: prefix ≤ lst
  and base:   ⋀xs. P [] xs
  and rl:     ⋀x xs y ys. ⟦ x = y; xs ≤ ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
  shows P prefix lst
  using np
proof (induct prefix arbitrary: lst)
  case Nil show ?case by fact
next
  case (Cons x xs)

  have prem: (x # xs) ≤ lst by fact
  then obtain y ys where lv: lst = y # ys
    by (rule prefixE, auto)

  have ih: ⋀lst. xs ≤ lst ⟹ P xs lst by fact

  show ?case using prem
    by (auto simp: lv intro!: rl ih)
qed

lemma not-prefix-cases:
  fixes prefix
  assumes pfx: ¬ prefix ≤ lst
  and c1: ⟦ prefix ≠ []; lst = [] ⟧ ⟹ R
  and c2: ⋀a as x xs. ⟦ prefix = a#as; lst = x#xs; x = a; ¬ as ≤ xs⟧ ⟹ R
```

```
  and c3: ⋀a as x xs. ⟦ prefix = a#as; lst = x#xs; x ≠ a⟧ ⟹ R
  shows R
proof (cases prefix)
  case Nil then show ?thesis using pfx by simp
next
  case (Cons a as)

  have c: prefix = a#as by fact

  show ?thesis
  proof (cases lst)
    case Nil then show ?thesis
      by (intro c1, simp add: Cons)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      show ?thesis
      proof (intro c2)
        show ¬ as ≤ xs using pfx c Cons True
          by simp
      qed fact+
    next
      case False
      show ?thesis by (rule c3) fact+
    qed
  qed
qed

lemma not-prefix-induct [consumes 1, case-names Nil Neq Eq]:
  fixes prefix
  assumes np: ¬ prefix ≤ lst
  and base:   ⋀x xs. P (x#xs) []
  and r1:     ⋀x xs y ys. x ≠ y ⟹ P (x#xs) (y#ys)
  and r2:     ⋀x xs y ys. ⟦ x = y; ¬ xs ≤ ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
  shows P prefix lst
  using np
proof (induct lst arbitrary: prefix)
  case Nil then show ?case
    by (auto simp: neq-Nil-conv elim!: not-prefix-cases intro!: base)
next
  case (Cons y ys)

  have npfx: ¬ prefix ≤ (y # ys) by fact
  then obtain x xs where pv: prefix = x # xs
    by (rule not-prefix-cases) auto

  have ih: ⋀prefix. ¬ prefix ≤ ys ⟹ P prefix ys by fact
```

show *?case* **using** *npfx*
by (*simp only*: *pv*) (*erule not-prefix-cases*, *auto intro*: *r1 r2 ih*)
qed

lemma *rsubst*:
$[\![P\ s;\ s = t]\!] \Longrightarrow P\ t$
by *simp*

lemma *ex-impE*: $((\exists x.\ P\ x) \longrightarrow Q) \Longrightarrow P\ x \Longrightarrow Q$
by *blast*

lemma *option-Some-value-independent*:
$[\![f\ x = Some\ v;\ \bigwedge v'.\ f\ x = Some\ v' \Longrightarrow f\ y = Some\ v']\!] \Longrightarrow f\ y = Some\ v$
by *blast*

Some int bitwise lemmas. Helpers for proofs about `NatBitwise.thy`

lemma *int-2p-eq-shiftl*:
$(2::int)\hat{}x = 1 << x$
by (*simp add*: *shiftl-int-def*)

lemma *nat-int-mul*:
$nat\ (int\ a * b) = a * nat\ b$
by (*simp add*: *nat-mult-distrib*)

lemma *int-shiftl-less-cancel*:
$n \leq m \Longrightarrow ((x :: int) << n < y << m) = (x < y << (m - n))$
apply (*drule le-Suc-ex*)
apply (*clarsimp simp*: *shiftl-int-def power-add*)
done

lemma *int-shiftl-lt-2p-bits*:
$0 \leq (x::int) \Longrightarrow x < 1 << n \Longrightarrow \forall i \geq n.\ \neg\ x\ !!\ i$
apply (*clarsimp simp*: *shiftl-int-def*)
by (*metis bit-take-bit-iff not-less take-bit-int-eq-self-iff*)
— TODO: The converse should be true as well, but seems hard to prove.

lemmas *int-eq-test-bit* = *bin-eq-iff*
lemmas *int-eq-test-bitI* = *int-eq-test-bit*[*THEN iffD2*, *rule-format*]

lemma *le-nat-shrink-left*:
$y \leq z \Longrightarrow y = Suc\ x \Longrightarrow x < z$
by *simp*

lemma *length-ge-split*:
$n < length\ xs \Longrightarrow \exists x\ xs'.\ xs = x\ \#\ xs' \wedge n \leq length\ xs'$
by (*cases xs*) *auto*

Support for defining enumerations on datatypes derived from enumerations

lemma *distinct-map-enum*:
⟦ (∀ *x y*. (*F x* = *F y* ⟶ *x* = *y*)) ⟧
⟹ *distinct* (*map F* (*enum-class.enum* :: *'a* :: *enum list*))
by (*simp add*: *distinct-map inj-onI*)

lemma *if-option-None-eq*:
((*if P then None else Some x*) = *None*) = *P*
by (*auto split*: *if-splits*)

lemma *not-in-ran-None-upd*:
x ∉ *ran m* ⟹ *x* ∉ *ran* (*m*(*y* := *None*))
by (*auto simp*: *ran-def split*: *if-split*)

Prevent clarsimp and others from creating Some from not None by folding this and unfolding again when safe.

definition
not-None x = (*x* ≠ *None*)

lemma *sorted-wrt-hd-min*: ⟦ ⋀*x*. *P x x*; *sorted-wrt P xs* ⟧ ⟹ (∀ *x* ∈ *set xs*. *P* (*hd xs*) *x*)
by (*induction xs*) *auto*

lemma *disjoint-no-subset*: *A* ∩ *B* = {} ⟹ *A* ≠ {} ⟹ *A* ⊆ *B* ⟹ *False*
by *blast*

lemma *map-prod-split-case*: *map-prod f g x* = (*case x of* (*a*, *b*) => (*f a*, *g b*))
by (*cases x*) (*auto*)

lemma *map-prod-split-prj*: *map-prod f g x* = (*f* (*fst x*), *g* (*snd x*))
by (*cases x*) (*auto*)

end

Chapter 3

ML Antiquotations

3.1 Building terms: *mk-term*

theory *MkTermAntiquote*
imports
Pure
begin

mk-term: ML antiquotation for building and splicing terms.
See `MkTermAntiquote_Tests.thy` for examples and tests.

ML-file *mkterm-antiquote.ML*

end
mk-term: ML antiquotation for building and splicing terms.

theory *MkTermAntiquote-Tests*
imports
MkTermAntiquote
Main
begin

Basic usage: @{*mk-term pattern...* (*vars*, ...)} (*args*, ...)
The vars should match schematic vars in the pattern, and they are substituted with the given arguments.
Template vars can include type vars, and they should be applied to type arguments.

ML-val ‹
@{*assert*} (@{*mk-term ?x* (*x*)} @{*term x*::'*a*}
= @{*term x*::'*a*});

@{*assert*} (@{*mk-term 0*::'*a*::*zero* ('*a*)} @{*typ nat*}
= @{*term 0*::*nat*});

(∗ *Tuples are used to pass multiple arguments* ∗)
@{*assert*} (@{*mk-term ?xs* @ *?ys* (*ys*, *xs*)} (@{*term* [*True*]}, @{*term* [*False*]})

```
        = @{term [False] @ [True]});
›
```

Note that *mk-term* does not perform full type inference automatically. If some argument types are mismatched or too general, the output may be inconsistent. This may change in future versions.

ML-val ‹
```
@{assert} (@{mk-term ?x = ?y (x, y)} (@{term 0::'a::zero}, @{term Suc 0})
        = (Const (@{const-name HOL.eq}, @{typ 'a::zero ⇒ 'a ⇒ bool}) $
            @{term 0::'a::zero} $ @{term Suc 0}))
›
```

Besides static checking, using *mk-term* also gives a reusable template:

ML-val ‹
```
let
  val mk-pair: term*term -> term = @{mk-term (?a, ?b) (a, b)};
in
  @{assert} (mk-pair (@{term ()}, @{term True}) = @{term ((), True)});
  @{assert} (mk-pair (@{term int 1}, @{term int 2})
          = @{term (int 1, int 2)})
end;
›
```

end

3.2 Term pattern: *term-pat*

theory *TermPatternAntiquote* **imports**
  Pure
begin

term-pat: ML antiquotation for pattern matching on terms.
See `TermPatternAntiquote_Tests.thy` for examples and tests.

ML ‹
```
structure Term-Pattern-Antiquote = struct

val quote-string = quote

(* typ matching; doesn't support matching on named TVars.
 * This is because each TVar is likely to appear many times in the pattern. *)
fun gen-typ-pattern (TVar -) = -
  | gen-typ-pattern (TFree (v, sort)) =
      Term.TFree ( ^ quote-string v ^ , [ ^ commas (map quote-string sort) ^ ])
  | gen-typ-pattern (Type (typ-head, args)) =
       Term.Type ( ^ quote-string typ-head ^ , [ ^ commas (map gen-typ-pattern
args) ^ ])

(* term matching; does support matching on named (non−dummy) Vars.
```

```
 * The ML var generated will be identical to the Var name except in
 * indexed names like ?v1.2, which creates the var v12. *)
fun gen-term-pattern (Var ((-dummy-, -), -)) = -
  | gen-term-pattern (Var ((v, 0), -)) = v
  | gen-term-pattern (Var ((v, n), -)) = v ^ string-of-int n
  | gen-term-pattern (Const (n, typ)) =
      Term.Const ( ^ quote-string n ^ ,  ^ gen-typ-pattern typ ^ )
  | gen-term-pattern (Free (n, typ)) =
      Term.Free ( ^ quote-string n ^ ,  ^ gen-typ-pattern typ ^ )
  | gen-term-pattern (t as f $ x) =
      (* (read-term-pattern -) helpfully generates a dummy var that is
       * applied to all bound vars in scope. We go back and remove them. *)
      let fun default () = ( ^ gen-term-pattern f ^ $  ^ gen-term-pattern x ^ );
      in case strip-comb t of
            (h as Var ((-dummy-, -), -), bs) =>
              if forall is-Bound bs then gen-term-pattern h else default ()
          | - => default () end
  | gen-term-pattern (Abs (-, typ, t)) =
      Term.Abs (-,  ^ gen-typ-pattern typ ^ ,  ^ gen-term-pattern t ^ )
  | gen-term-pattern (Bound n) = Bound  ^ string-of-int n

(* Create term pattern. All Var names must be distinct in order to generate ML
variables. *)
fun term-pattern-antiquote ctxt s =
  let val pat = Proof-Context.read-term-pattern ctxt s
      val add-var-names' = fold-aterms (fn Var (v, -) => curry (::) v | - => I);
      val vars = add-var-names' pat [] |> filter (fn (n, -) => n <> -dummy-)
      val - = if vars = distinct (=) vars then () else
              raise TERM (Pattern contains duplicate vars, [pat])
  in ( ^ gen-term-pattern pat ^ ) end

end;
val - = Context.>> (Context.map-theory (
    ML-Antiquotation.inline @{binding term-pat}
      ((Args.context -- Scan.lift Parse.embedded-inner-syntax)
        >> uncurry Term-Pattern-Antiquote.term-pattern-antiquote)))
›

end


theory TermPatternAntiquote-Tests
imports
  TermPatternAntiquote
  Main
begin

    Term pattern matching utility.
    Instead of writing monstrosities such as
```

```
case t of
  Const ("Pure.imp", _) $
    P $
    Const (@{const_name Trueprop}, _) $
      (Const ("HOL.eq", _) $
        (Const (@{const_name "my_func"}, _) $ x) $ y)
  => (P, x, y)
```

simply use a term pattern with variables:

```
case t of
  @{term_pat "PROP ?P \<Longrightarrow> my_func ?x = ?y"}
=> (P, x, y)
```

Each *term-pat* generates an ML pattern that can be used in any case-expression or name binding. The ML pattern matches directly on the term datatype; it does not perform matching in the Isabelle logic.

Schematic variables in the pattern generate ML variables. The variables must obey ML's pattern matching rules, i.e. each can appear only once.

Due to the difficulty of enforcing this rule for type variables, schematic type variables are ignored and not checked.

Example: evaluate arithmetic expressions in ML.

ML-val ‹
fun eval-num @{*term-pat numeral ?n*} = *HOLogic.dest-numeral n*
  | *eval-num* @{*term-pat Suc ?n*} = *eval-num n* + *1*
  | *eval-num* @{*term-pat 0*} = *0*
  | *eval-num* @{*term-pat 1*} = *1*
  | *eval-num* @{*term-pat ?x* + *?y*} = *eval-num x* + *eval-num y*
  | *eval-num* @{*term-pat ?x* − *?y*} = *eval-num x* − *eval-num y*
  | *eval-num* @{*term-pat ?x* ∗ *?y*} = *eval-num x* ∗ *eval-num y*
  | *eval-num* @{*term-pat ?x div ?y*} = *eval-num x div eval-num y*
  | *eval-num t* = *raise TERM* (*eval-num*, [*t*]);

eval-num @{*term* (*1* + *2*) ∗ *3* − *4 div 5*}
›

Regression test: backslash handling

ML-val ‹
val @{*term-pat* α} = @{*term* α}
›

Regression test: special-casing for dummy vars

ML-val ‹
val @{*term-pat* λ*x y*. -} = @{*term* λ*x y*. *z*}
›

end

theory *Print-Annotated*
imports *Main*
keywords *print-annotated-thm* :: *diag*
begin

ML ‹
signature ANNOTATE-TERM =
sig
val string-of-term: Proof.context -> int option -> term -> string
val string-of-thm: Proof.context -> int option -> thm -> string
val print-thm: Proof.context -> int option -> thm -> unit
end

structure Annotate-Term: ANNOTATE-TERM =
struct

fun string-of-term ctxt margin t =
let
val ctxt' = ctxt
|> Config.put show-markup false
|> Config.put Printer.show-type-emphasis false
|> Config.put show-types false
|> Config.put show-sorts false
|> Config.put show-consts false
in
t
|> singleton (Syntax.uncheck-terms ctxt')
|> Sledgehammer-Isar-Annotate.annotate-types-in-term ctxt'
|> Syntax.unparse-term ctxt'
|> Pretty.string-of-ops (Pretty.pure-output-ops margin)
end

fun string-of-thm ctxt margin thm =
string-of-term ctxt margin (Thm.concl-of thm)

fun print-thm ctxt margin thm =
writeln (Active.sendback-markup-command (lemma \ ^
string-of-thm ctxt margin (Variable.import-vars ctxt thm) ^ \))

val - =
Outer-Syntax.command ***command-keyword*** ‹*print-annotated-thm*›
print theorems with (minimal) complete type annotations
(Scan.option (Args.parens Parse.nat) -- Parse.thm
>> (fn (margin, thm-src) => Toplevel.keep (fn state =>

```
      let
        val ctxt = Toplevel.context-of state
        val [thm] = Attrib.eval-thms ctxt [thm-src]
      in print-thm ctxt margin thm end)));

end
›

end
```

```
theory ML-Fun-Cache
  imports Pure
begin
```

Extension of the basic cache in `Cache`:

- Include some statistics on cache hits / misses
- Retrieval of existing caches
- Support garbage collected caches (via weak references)

Combination of synchronized variables and weak references. Stored value might get garbage collected. Note the difference in the signature (option value at some positions) compared to `Synchronized`.

```
ML ‹
signature SYNCHRONIZED-WEAK =
sig
  type 'a var
  val var: string -> 'a -> 'a var
  val value: 'a var -> 'a option
  val assign: 'a var -> 'a -> unit

  val timed-access: 'a var -> ('a option -> Time.time option) -> ('a option ->
('b * 'a) option) -> 'b option
  val guarded-access: 'a var -> ('a option -> ('b * 'a) option) -> 'b
  val change-result: 'a var -> ('a option -> 'b * 'a) -> 'b
  val change: 'a var -> ('a option -> 'a) -> unit

end;


structure Synchronized-Weak: SYNCHRONIZED-WEAK =
struct
  abstype 'a var = Weak-Var of ('a Unsynchronized.ref option Unsynchronized.ref)
Synchronized.var
  with
```

```
(* basic operations on weak references *)

fun mk-weak x = Weak.weak (SOME (Unsynchronized.ref x))

fun get-val w = case !w of
      SOME r => SOME (!r)
    | NONE => NONE

fun map-val f w = f (get-val w)

(* the main interface *)

fun var name x =
  Weak-Var (Synchronized.var name (mk-weak x))

fun value (Weak-Var x) =
    get-val (Synchronized.value x)

fun assign (Weak-Var x) v = Synchronized.assign x (mk-weak v)

fun apply-access f w =
  case f (get-val w) of
     SOME (res, x) => SOME (res, mk-weak x)
   | NONE => NONE

fun timed-access (Weak-Var x) time-limit f =
  Synchronized.timed-access x (map-val time-limit) (apply-access f)

fun guarded-access var f = the (timed-access var (fn - => NONE) f);

(* unconditional change *)

fun change-result var f = guarded-access var (SOME o f);
fun change var f = change-result var (fn x => ((), f x));
end
end
›


ML ‹
signature MORE-BINDING = sig
  val here-pretty: binding -> Pretty.T
  val here: binding -> string
end

structure More-Binding: MORE-BINDING = struct

fun here-pretty b =
```

```
  let
    val pos = Binding.pos-of b
    val prt0 = Pretty.mark-str-position (pos, Binding.long-name-of b)
  in
    (case Pretty.here pos of
      [] => prt0
    | prts => Pretty.block0 (prt0 :: prts))
  end

val here = Pretty.unformatted-string-of o here-pretty

end

signature FUN-CACHE =
sig

  (* non garbage collected variant of cache *)
   val create: binding -> ('table -> string) -> 'table -> ('table -> 'key ->
'value lazy option) ->
     ('key * 'value lazy -> 'table -> 'table) -> ('key -> 'value) -> 'key ->
'value

  (* garbage collected variant of cache *)
  val create-gc: binding -> ('table -> string) -> 'table -> ('table -> 'key ->
'value lazy option) ->
     ('key * 'value lazy -> 'table -> 'table) -> ('key -> 'value) -> 'key ->
'value

  datatype 'content handler = Handler of {name: binding, hits: unit -> int, misses:
unit -> int,
          reset: unit -> unit, content: unit -> 'content, mk-string: 'content ->
string}

  (* non garbage collected variant of cache *)
  val create-handler: binding -> ('table -> string) -> 'table -> ('table -> 'key
-> 'value lazy option) ->
    ('key * 'value lazy -> 'table -> 'table) -> ('key -> 'value) -> (('key ->
'value) * 'table handler)

  (* garbage collected variant of cache *)
   val create-gc-handler: binding -> ('table -> string) -> 'table -> ('table ->
'key -> 'value lazy option) ->
    ('key * 'value lazy -> 'table -> 'table) -> ('key -> 'value) -> (('key ->
'value) * 'table handler)

  val get-handler: string -> string handler option
  val delete-handler: string -> unit
  val all-handlers: unit -> (string * string handler) list
  val cache-statistics: unit -> (binding * {hits: int, misses: int, content-size: int})
```

```
list

  val reset-cache: string -> unit
  val reset-all-caches: unit -> unit

  val get-info: 'content handler -> {name: binding, hits: int, misses: int, content:
'content, mk-string: 'content -> string}

  val dummy-handler : 'content ->  'content handler;

  val pretty-handler: 'content handler -> Pretty.T
  val string-of-handler: 'content handler -> string
end;

structure Fun-Cache: FUN-CACHE =
struct

datatype 'content handler = Handler of {name: binding, hits: unit -> int, misses:
unit -> int,
  reset: unit -> unit, content: unit -> 'content, mk-string: 'content -> string};

fun dummy-handler c = Handler {name = Binding.name , hits = K 0, misses =
K 0, reset = K (), content = K c, mk-string = K ()};

val caches = Synchronized.var caches (Symtab.empty)

fun add-handler name handler = Synchronized.change caches (fn tab =>
  let
    val raw-name = Binding.name-of name
     val - = if Symtab.defined tab raw-name then tracing (overwriting cache:  ^
More-Binding.here name) else ()
  in Symtab.update (raw-name, handler) tab end)

fun cached-apply name lookup update f x (tab, hits, misses) =
      case lookup tab x of
        SOME y => (y, (tab, hits + 1, misses))
      | NONE =>
          let val y = Lazy.lazy-name name (fn () => f x)
          in (y, (update (x, y) tab, hits, misses + 1)) end

fun gen-apply cached-apply change-result cache =
  change-result cache cached-apply
  |> Lazy.force;

fun create-handler name mk-string empty lookup update f =
  let
    val raw-name = Binding.name-of name
    val empty-cache = (empty, 0, 0)
    val cache = Synchronized.var raw-name empty-cache;
```

```
    fun apply x = gen-apply (cached-apply raw-name lookup update f x) Synchro-
nized.change-result cache

    fun hits () = #2 (Synchronized.value cache)
    fun misses () = #3 (Synchronized.value cache)
    fun content () = #1 (Synchronized.value cache)
    fun string () = mk-string (#1 (Synchronized.value cache))
    fun reset () = Synchronized.change cache (K empty-cache)

    val handler = Handler {name = name, hits = hits, misses = misses, reset =
reset, content = content, mk-string = mk-string}
    val handler' = Handler {name = name, hits = hits, misses = misses, reset =
reset, content = string, mk-string = I}
    val - = add-handler name handler'
  in (apply, handler) end;

fun create name mk-string empty lookup update f = fst (create-handler name mk-string
empty lookup update f)

fun map-default-trace msg default f v =
  case v of SOME x => f x | NONE => (tracing msg; f default)

fun the-default-trace msg default v =
  case v of
   SOME x => x
  | NONE => (tracing msg; default)

fun create-gc-handler name mk-string empty lookup update f =
  let
    val raw-name = Binding.name-of name
    val empty-cache = (empty, 0, 0)
    val cache = Synchronized-Weak.var raw-name empty-cache;
    val the-default = the-default-trace (cache: ^ More-Binding.here name ^ was
garbage collected)
    val map-default = map-default-trace (cache: ^ More-Binding.here name ^ was
garbage collected - starting with empty cache again.)

    fun apply x = gen-apply (map-default empty-cache (cached-apply raw-name
lookup update f x)) Synchronized-Weak.change-result cache

    fun get cache = the-default empty-cache (Synchronized-Weak.value cache)
    fun hits () = #2 (get cache)
    fun misses () = #3 (get cache)
    fun content () = #1 (get cache)
    fun string () = mk-string (#1 (get cache))
    fun reset () = Synchronized-Weak.change cache (K empty-cache)

    val handler = Handler {name = name, hits = hits, misses = misses, reset =
reset, content = content, mk-string = mk-string}
```

```
    val handler' = Handler {name = name, hits = hits, misses = misses, reset =
reset, content = string, mk-string = I}
    val - = add-handler name handler'
  in (apply, handler) end;

fun create-gc name mk-string empty lookup update f = fst (create-gc-handler name
mk-string empty lookup update f)

fun get-handler name =
  Symtab.lookup (Synchronized.value caches) name

fun delete-handler name = Synchronized.change caches (Symtab.delete name)

fun all-handlers () =
  Symtab.dest (Synchronized.value caches)

fun cache-statistics () =  all-handlers () |>
   map (fn (-, Handler h) =>
       (#name h, {hits = #hits h (), misses = #misses h (), content-size =
ML-Heap.obj-size (#content h ())}))

fun reset-cache name =
  case get-handler name of
    SOME (Handler h) => #reset h ()
  | NONE => warning (reset-handler: no handler with name  ^ name)

fun reset-all-caches () =
  all-handlers () |> map (apsnd (fn Handler h => #reset h ())) |> K ()

fun get-info (Handler handler:'content handler) =
  {name = #name handler, hits = (#hits handler) (), misses = (#misses handler)
(), content = (#content handler) (), mk-string = #mk-string handler}

fun pretty-elem name value = Pretty.block (Pretty.breaks [Pretty.str name, Pretty.str
=, Pretty.str value])
fun pretty-info {name, content, hits, misses, mk-string, ...} =
  Pretty.list { } (
   [pretty-elem name (More-Binding.here name),
    pretty-elem hits (string-of-int hits),
    pretty-elem misses (string-of-int misses),
    pretty-elem content (mk-string content)])

fun pretty-handler x = (pretty-info o get-info) x
fun string-of-info x = (Pretty.string-of o pretty-info) x
fun string-of-handler x = (string-of-info o get-info) x

val - =
  ML-system-pp (fn depth => fn pretty => fn handler =>
    Pretty.to-ML (pretty-handler handler));
```

```
end;
›
```

3.2.1 Example

Cache without garbage collection

```
ML-val ‹
fun fib n =
  if n < 3 then 1
  else fib (n−1) + fib (n−2)

val (fib, handler) = Fun-Cache.create-handler
  @{binding fib}
  (fn n => @{make-string} n) (* as the type of n is fixed @{make-string} will pick
a proper print function *)
  Inttab.empty
  Inttab.lookup
  Inttab.update
  fib

val - = tracing (trace 1: ^ Fun-Cache.string-of-handler handler)
val tests1 = map fib (1 upto 6)
val - = (tracing enforcing garbage collection; ML-Heap.full-gc()); (* does not affect
cache *)
val - = tracing (trace 2: ^ Fun-Cache.string-of-handler handler)
val tests2 = map fib (1 upto 6)
val - = tracing (trace 3: ^ Fun-Cache.string-of-handler handler)
›
```

Cache with garbage collection

```
ML-val ‹
fun fib n =
  if n < 3 then 1
  else fib (n−1) + fib (n−2)

val (fib, handler) = Fun-Cache.create-gc-handler
  @{binding fib}
  (fn n => @{make-string} n) (* as the type of n is fixed @{make-string} will pick
a proper print function *)
  Inttab.empty
  Inttab.lookup
  Inttab.update
  fib

val - = tracing (trace 1: ^ Fun-Cache.string-of-handler handler)
val tests1 = map fib (1 upto 6)
val - = tracing (trace 2: ^ Fun-Cache.string-of-handler handler)
```

val - = (tracing enforcing garbage collection; ML-Heap.full-gc()); (∗ should empty cache ∗)
val - = tracing (trace 3: ^ Fun-Cache.string-of-handler handler)
val tests2 = map fib (1 upto 6)
val - = tracing (trace 4: ^ Fun-Cache.string-of-handler handler)
val tests3 = map fib (1 upto 6)
val - = tracing (trace 5: ^ Fun-Cache.string-of-handler handler)
›

end

theory *AutoCorres-Utils*
imports
Print-Annotated
ML-Fun-Cache
keywords *lazy-named-theorems*::*thy-decl*

begin

definition *CONV-ID*:: *'a*::{} ⇒ *'a* **where**
CONV-ID x ≡ *x*

lemma *CONV-ID-intro*: (*x*::*'a*::{}) ≡ *CONV-ID x*
by (*simp add*: *CONV-ID-def*)

ML-file *utils.ML*

definition *FALSE* ≡ (⋀*P*. *PROP P*)
lemma *ex-falso-quodlibet*: *PROP FALSE* ⟹ *PROP P*
by (*simp add*: *FALSE-def*)

ML-file ‹*context-tactical.ML*›
ML-file *lazy-named-theorems.ML*
ML-file *interpretation-data.ML*

definition *sim-set* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a set* ⇒ *'b set* ⇒ *bool* **where**
sim-set R A B ⟷ (∀ *a*∈*A*. ∃ *b*∈*B*. *R a b*)

lemma *sim-set-empty*: *sim-set R* {} *B*
by (*auto simp*: *sim-set-def*)

lemma *sim-set-insert*: *R a b* ⟹ *sim-set R A B* ⟹ *sim-set R* (*insert a A*) (*insert b B*)
by (*auto simp*: *sim-set-def*)

lemma *sim-set-Collect-Ex*:
(⋀*a*. *sim-set R* {*x*. *P a x*} {*x*. *Q a x*}) ⟹ *sim-set R* {*x*. ∃ *a*. *P a x*} {*x*. ∃ *a*. *Q a x*}

by (*fastforce simp*: *sim-set-def*)

lemma *sim-set-Collect-conj*:
(*A* ⟹ *sim-set R* {*x*. *P x*} {*x*. *Q x*}) ⟹ *sim-set R* {*x*. *A* ∧ *P x*} {*x*. *A* ∧ *Q x*}
by (*fastforce simp*: *sim-set-def*)

lemma *sim-set-Collect-eq*:
R a b ⟹ *sim-set R* {*x*. *x* = *a*} {*x*. *x* = *b*}
by (*fastforce simp*: *sim-set-def*)

lemma *sim-set-eq-rel-set*: *sim-set R* = *rel-set R OO* (⊆)
apply (*auto simp*: *sim-set-def rel-set-def relcomp.simps fun-eq-iff OO-def*)[*1*]
subgoal for *A B*
by (*intro exI*[*of* - {*b*∈*B*. ∃ *a*∈*A*. *R a b*}]) *blast*
apply *blast*
done

end

3.3 ML Antiquotation to Match and Instantiate (recombine) cterms and terms

theory *Match-Cterm*
imports
AutoCorres-Utils
TermPatternAntiquote
begin

ML ‹
signature MATCH-CTERM =
sig
val try-match: ('*a* −> '*b*) −> '*a* −> '*b option* (∗ *handles Match* | *Pattern.MATCH* ∗)
val switch: ('*a* −> '*b*) *list* −> '*a* −> '*b* (∗ *raises Match* ∗)

val cterm-match: *cterm* ∗ *cterm* −> *ctyp TVars.table* ∗ *cterm Vars.table*
val cterm-first-order-match: *cterm* ∗ *cterm* −> *ctyp TVars.table* ∗ *cterm Vars.table*

val match: *theory* −> *term* ∗ *term* −> *typ TVars.table* ∗ *term Vars.table*
val first-order-match: *theory* −> *term* ∗ *term* −> *typ TVars.table* ∗ *term Vars.table*

val pattern-vars: *term* −> *term list*
val cterm-pattern-vars: *cterm* −> *cterm list*

val cterm-instantiate: *Position.T* −>
ctyp TVars.table ∗ *cterm Vars.table* −>
cterm list −> *cterm list* −> *cterm* −> *cterm*

```
  val instantiate: Position.T ->
      typ TVars.table * term Vars.table ->
        term list -> term list -> term -> term

  datatype mode = Higher-Order  | First-Order | ML-Pattern
  val parse-mode: Token.T list -> mode * Token.T list
  val parse-morph-mode: Token.T list -> mode * Token.T list
   val ml-match-term: {morph: bool, cert: bool} -> mode -> Proof.context ->
Position.T -> string -> string
end

structure Match-Cterm: MATCH-CTERM =
struct

fun try-match f x = SOME (f x) handle Match => NONE | Pattern.MATCH =>
NONE

fun switch (c::cs) = (fn x => (* to allow partial evaluation on list elements *)
      (case try-match c x of
         SOME v => v
      | NONE => switch cs x))
  | switch [] = raise Match

(* ML code composition *)
fun ml-val-binding (name, value) = implode-space [val, name, =, value];
fun ml-indent n str = String.concat (replicate n  ) ^ str

fun ml-let-binding vals body =
  [let] @
     (map (ml-indent 2) vals) @ [
   in,
     ml-indent 2 body,
   end]

fun ml-list elems   = enclose [ ] (space-implode ,  elems)
fun ml-tuple elems  = enclose ( ) (space-implode ,  elems)
fun ml-record elems = enclose { } (space-implode ,  elems)
val ml-atomic = enclose ( )

fun triv-record-field name = implode-space [name, =, name]
val triv-record-inst = map triv-record-field #> ml-record

fun ml-fun name args body =
  implode-space ([fun, name] @ args @ [=]) ::
  (map (ml-indent 2) body)

fun ml-fn args body =
  implode-space (map (fn arg => implode-space [fn, arg, =>]) args)::
  (map (ml-indent 2) body)
```

```
fun ml-apply f args = implode-space (f :: args)

fun ml-check-match term-pat args ct = (* apply to args to ommit warning on unused
identifiers *)
  ml-apply
    (ml-atomic (implode-space [fn, term-pat, =>, ml-tuple args, | - => raise Match
]))
    [ml-atomic (ml-apply Thm.term-of [ct])]


(* Interface used by antiquotation *)
fun cterm-match (pat, obj) = Thm.match (pat, obj)
  handle Pattern.MATCH => Exn.reraise Pattern.MATCH

fun cterm-first-order-match (pat, obj) = Thm.first-order-match (pat, obj)
  handle Pattern.MATCH => Exn.reraise Pattern.MATCH

fun gen-match match thy (pat, obj) =
  let
    val (Tinsts, tinsts) = match thy (pat, obj) (Vartab.empty, Vartab.empty)
    fun mk-Tinst ((a, i), (S, T)) =
      (((a, i), S), T);
    fun mk-tinst ((x, i), (U, t)) =
      let val T = Envir.subst-type Tinsts U in
        (((x, i), T), t)
      end
  in
    (TVars.build (Vartab.fold (TVars.add o mk-Tinst) Tinsts),
     Vars.build (Vartab.fold (Vars.add o mk-tinst) tinsts))
  end

fun match thy (pat, obj) = gen-match Pattern.match thy (pat, obj)
  handle Pattern.MATCH => Exn.reraise Pattern.MATCH

fun first-order-match thy (pat, obj) = gen-match Pattern.first-order-match thy (pat,
obj)
  handle Pattern.MATCH => Exn.reraise Pattern.MATCH

fun cterm-instantiate pos (tenv, env) vars vals pattern =
  let
    val insts = map (Term.dest-Var o Thm.term-of) vars ~~ vals
    fun upd var value = the-default value (AList.lookup (op =) insts var)
    val env' = Vars.map upd env
  in
    Thm.instantiate-cterm (tenv, env') pattern
  end
  handle
    CTERM (msg, args) =>
```

```
      Exn.reraise (CTERM (msg ^ "\n from antiquotation Match-Cterm.cterm-instantiate
here: " ^ Position.here pos, args));

fun instantiate pos (tenv, env) vars vals pattern =
  let
    val insts = map Term.dest-Var vars ~~ vals
    fun upd var value = the-default value (AList.lookup (op =) insts var)
    val env' = Vars.map upd env
  in
    Term-Subst.instantiate (tenv, env') pattern
  end
  handle
    TERM (msg, args) =>
      Exn.reraise (TERM (msg ^ "\n from antiquotation Match-Cterm.instantiate
here: " ^ Position.here pos, args));

(* Antiquotation *)

fun collect-vars t = fold-aterms (fn Var v => cons v | - => I) t [] |> rev

fun pattern-vars pat = pat
  |> collect-vars
  |> filter-out (fn ((-dummy-, -), -) => true | - => false)
  |> (distinct (op =)) |> map Var

fun cterm-pattern-vars cpat =
  let
    val ctxt = cpat |> Thm.theory-of-cterm |> Proof-Context.init-global
  in
    cpat |> Thm.term-of |> pattern-vars |> map (Thm.cterm-of ctxt)
  end

datatype mode = Higher-Order | First-Order | ML-Pattern

fun ml-match-term (kind as {morph:bool, cert:bool}) (mode:mode) ctxt pos pat-
tern-str =
  let
    val phiN = phi-
    val thyN = thy-

    val match = case (mode, kind) of
          (Higher-Order, {cert = true ,...}) => Match-Cterm.cterm-match
      | (Higher-Order, {cert = false,...}) => ml-apply Match-Cterm.match [thyN]
       | (-           , {cert = true ,...}) => Match-Cterm.cterm-first-order-match
       | (-           , {cert = false,...}) => ml-apply Match-Cterm.first-order-match
[thyN]

    val cterm-of = case kind of
          {morph = false, cert = true } => Thm.cterm-of ML-context
```

```
      | {morph = false, cert = false} =>
        | {morph = true , cert = true }  => (Morphism.cterm  ^ phiN ^  o
Thm.cterm-of ML-context)
      | {morph = true , cert = false}  => Morphism.term  ^ phiN

    val ctN = ct- val envN = env- val instN = inst- val patN = pat- val posN =
pos-

    val instantiateN = instantiate
    val reserved = [ctN, envN, instN, instantiateN, patN, posN, instantiateN]
    val pat = Proof-Context.read-term-pattern ctxt pattern-str
    val vars = collect-vars pat |> filter-out (fn ((-dummy-, -), -) => true | - =>
false)
              |> (mode <> ML-Pattern) ? (distinct (op =))
    val names = map (#1 o #1) vars
    val dups = duplicates (op =) names
    val - = if null dups then ()
       else error (ml-match-cterm: duplicate variables in pattern:  ^Position.here
pos  ^(commas (map quote dups)))
    val clashes = inter (op =) names reserved
    val - = if null clashes then ()
       else error (ml-match-cterm: avoid internal name(s) in pattern:  ^(commas
(map quote clashes)))

    val names-var = map (suffix -var) names

    val ml-pat = ml-apply cterm-of [ml-atomic (ML-Syntax.print-term pat)]
    val ml-pattern-check =
      if mode = ML-Pattern then
        [ml-val-binding (-,
            ml-check-match (Term-Pattern-Antiquote.term-pattern-antiquote ctxt
pattern-str) names ctN)]
      else []

    val pat-vars = case cert of
       true => Match-Cterm.cterm-pattern-vars
     | false => Match-Cterm.pattern-vars

    val inst = case cert of
       true => Thm.instantiate-cterm
     | false => Term-Subst.instantiate
    val instantiate = case cert of
       true => Match-Cterm.cterm-instantiate
     | false => Match-Cterm.instantiate

    val vals = map ml-val-binding [
      (posN, ML-Syntax.print-position pos),
      (ml-list names-var, ml-apply pat-vars [patN]),
      (envN, ml-apply match [ml-tuple [patN, ctN]]),
```

```
        (ml-list names,
          ml-apply map
            [ml-atomic (ml-apply inst [envN]), ml-list names-var]),
        (ml-list names-var,
          ml-apply map
           [ml-atomic (ml-apply inst [ml-tuple [ml-apply fst [envN], Vars.empty]]),
            ml-list names-var])]
    val instantiate = ml-fun instantiateN [ml-record names]
        [ml-apply instantiate [posN, envN, ml-list names-var, ml-list names, patN]]
   val inst = ml-val-binding (instN, triv-record-inst (names @ [instantiateN, ctN]))
    val pat-binding = map ml-val-binding [(patN, ml-pat)]
    val outer-args = if morph then [phiN] else []

    val body = ml-let-binding (ml-pattern-check @ vals @ instantiate @ [inst]) instN
    val args = (if cert then [] else [thyN]) @ [ctN]
    val res = ml-fn args body |> space-implode \n
     val outer-body = ml-let-binding pat-binding res (* partial application of phi to
pattern *)
    val outer-res = ml-fn outer-args outer-body |> space-implode \n
    val - = Utils.verbose-msg 5 ctxt (fn - => ml-match-cterm: ^ outer-res)
  in
    outer-res
  end


val parse-mode = Scan.optional
      (Args.parens (
         Args.$$$ ho >> K Higher-Order ||
         Args.$$$ fo >> K First-Order ||
         Args.$$$ ml >> K ML-Pattern))
      Higher-Order;

val parse-morph-mode = Scan.optional
      (Args.parens (
         Args.$$$ ho >> K Higher-Order ||
         Args.$$$ fo >> K First-Order))
      Higher-Order;

end

val - = Theory.setup
   (ML-Antiquotation.value-embedded @{binding cterm-match}
        ((Args.context -- Scan.lift (Match-Cterm.parse-mode -- Parse.position
Parse.embedded-inner-syntax))
        >> (fn (ctxt, (mode, (pattern-str, pos))) =>
              Match-Cterm.ml-match-term {morph = false, cert = true} mode ctxt
pos pattern-str)) #>
   (ML-Antiquotation.value-embedded @{binding cterm-morph-match}
    ((Args.context -- Scan.lift (Match-Cterm.parse-morph-mode -- Parse.position
```

```
Parse.embedded-inner-syntax))
      >> (fn (ctxt, (mode, (pattern-str, pos))) =>
           Match-Cterm.ml-match-term {morph = true, cert = true} mode ctxt
pos pattern-str))) #>
  (ML-Antiquotation.value-embedded @{binding match}
    ((Args.context -- Scan.lift (Match-Cterm.parse-mode -- Parse.position
Parse.embedded-inner-syntax))
      >> (fn (ctxt, (mode, (pattern-str, pos))) =>
           Match-Cterm.ml-match-term {morph = false, cert = false} mode ctxt
pos pattern-str))) #>
  (ML-Antiquotation.value-embedded @{binding morph-match}
   ((Args.context -- Scan.lift (Match-Cterm.parse-morph-mode -- Parse.position
Parse.embedded-inner-syntax))
      >> (fn (ctxt, (mode, (pattern-str, pos))) =>
           Match-Cterm.ml-match-term {morph = true, cert = false} mode ctxt
pos pattern-str))))
›
```

The antiquotation yields a function that matches the schematic variables of a pattern with a `cterm`. The matched parts are returned as `cterm` in a record, where the field name is derived from the original schematic variable name. Moreover a function *instantiate* is returned that can be used to instantiate the matched pattern with other `cterm`s.

The antiquotation makes use of the kernel operations `Thm.match` / `Thm.first_order_match`. These operations are efficient in the sense that no costly re-certification of subterms has to be performed.

ML-val ‹
```
val {f, g, instantiate, ct-} = @{cterm-match λx. ?f x + ?g x} @{cterm λx. p x +
q x}
val twisted = instantiate {f = g, g = f}
```
›

Dummy pattern can also be supplied. The matched values for the dummy patterns are not part of the matching result but still considered in *instantiate*.

ML-val ‹
```
val {f, g, instantiate, ...} = @{cterm-match λx. ?f x + ?g x + -} @{cterm λx. p
x + q + r x}
val twisted = instantiate {f = g, g = f}
```
›

ML-val ‹
```
val {f, g, ...} = @{cterm-match λx. ?f x + ?g x} @{cterm λx. p x + q}
```
›

ML-val ‹

```
val {f1, f2, ...} = @{cterm-match λx. ?f1.0 x + ?f2.0 x} @{cterm λx. p x + q}
›
```

ML-val ‹
```
val {f, ...} = @{cterm-match λx. ?f x} @{cterm ‹λx. f›}
```
›

There is also a variant for first-order-matching.

ML-val ‹
```
let
  val - = @{cterm-match (fo) λx. ?f x} @{cterm ‹λx. f›}
in
  ()
end
handle Pattern.MATCH => tracing (this is not first order)
```
›

There is also the corresponding antiquotations for plain terms.

ML-val ‹
```
val {f, g, instantiate, ...} = @{match λx. ?f x + ?g x} @{theory} @{term λx. p x
+ q x}
val twisted = instantiate {f = g, g = f}
```
›

There you can also see the different workings of first-order-matching. Note the eta expanded match result in the higher-order variant before.

ML-val ‹
```
val {f, g, instantiate, ...} = @{match (fo) λx. ?f x + ?g x} @{theory} @{term λx.
p x + q x}
val twisted = instantiate {f = g, g = f}
```
›

There is a variant for ML-pattern-matching. This means, that before using first-order-matching it is tested whether the term actually matches the underlying ML-pattern with ML-pattern matching.

ML-val ‹
```
val {f, g, instantiate, ...} = @{cterm-match (ml) λx. ?f x + ?g x} @{cterm λx. p
x + q x}
val twisted = instantiate {f = g, g = f}
```
›

Note that as with all ML-Patterns a variable might only appear once. So the following example is a valid first-order or higher-order pattern but not a valid ML-pattern

ML-val ‹
```
val {f, ...} = @{cterm-match ?f + ?f} @{cterm a + a}
```
›

This is what happens conceptually in the expanded antiquotation.

ML-val ‹
val ctxt = @{context}
val pat = Proof-Context.read-term-pattern ctxt λx. ?f x + ?g x |> Thm.cterm-of ctxt
val [g-var, f-var] = Term.add-vars (Thm.term-of pat) [] |> map (Thm.cterm-of ctxt o Var)
val env = Thm.match (pat, @{cterm λx. a x + b})
val [f, g] = map (Thm.instantiate-cterm env) [f-var, g-var]
val tenv = fst env
val [f-var, g-var] = map (Thm.instantiate-cterm (tenv, Vars.empty)) [f-var, g-var]
fun instantiate {f, g} =
Match-Cterm.cterm-instantiate Position.none env [f-var, g-var] [f, g]
val res = {f = f, g = g, instantiate = instantiate}
val x= instantiate {f=g, g=f}
›

With the verbose flag you can see the generated function.

declare [[*verbose=5*]]

ML-val ‹
val X = @{cterm-match ?X + -} @{cterm ‹a + a›}
›

ML-val ‹
val X = @{cterm-morph-match ?X + -} Morphism.identity @{cterm ‹a + a›}
›

ML-val ‹
val X = @{morph-match ?X + -} Morphism.identity @{theory} @{term ‹a + a›}
›

ML-val ‹
val X = @{cterm-match ‹?X + -›} @{cterm ‹a + a›}
›

declare [[*verbose=0*]]

end

3.4 Record Antiquotation

theory *ML-Record-Antiquotation*
imports *Main*
begin

3.4.1 Motivation

A shortcoming of ML records is the lack of proper update / map functions for the record fields. Manually defining those update functions is considered as painful, as it requires 'quadratic' editing when adding a new field: - add the new field to every record pattern (in every already defined update / map function) - add a new update / map function for the field.

Various workarounds have been proposed. E.g. - Using mutable references as fields to allow for destructive updates - Fancy higher order function-combinators (Fold): http://mlton.org/FunctionalRecordUpdate

Here we develop yet another solution. We provide a ML-Antiquotation that generates the definitions.

For a record specification as datatype `datatype 'a foo = Foo {f1:'a, f2:bool}` it generates the datatype as specified with one constructor `Foo` wrapping the record and additionally all the 'canonical' functions:

- `make_foo`
- `dest_foo`
- `get_f1`
- `get_f2`
- `map_f1`
- `map_f2`

3.4.2 Example

ML-val ‹
datatype 'a *foo* = *Foo of* {*f1*:'a, *f2*:*bool*};

val make-foo = *Foo*;
fun dest-foo (*Foo r*) = *r*;

fun get-f1 (*Foo r*) = #*f1 r*;
fun get-f2 (*Foo r*) = #*f2 r*;

fun map-f1 f (*Foo* {*f1*, *f2*}) =
  Foo {*f1* = *f f1*, *f2* = *f2*};
fun map-f2 f (*Foo* {*f1*, *f2*}) =
  Foo {*f1* = *f1*, *f2* = *f f2*};
›

3.4.3 Implementation

ML ‹

```
structure ML-Record-Antiquotation =
struct

fun remove-comments s =
  let
    val unbalanced-comments-msg = unbalanced ML comments (* ... *)
    fun err-unbalanced () = error unbalanced-comments-msg

    fun rem n [] = if n = 0 then [] else err-unbalanced ()
      | rem n ((::*::)xs) = rem (n + 1) xs
      | rem n (*::)::xs) = if n > 0 then rem (n - 1) xs else err-unbalanced ()
      | rem n (x::xs) = if n = 0 then x::rem n xs else rem n xs
  in
    s |> Symbol.explode |> rem 0 |> implode
  end

fun to-upper-first s =
   case String.explode s of
     [] =>
   | c::cs => String.implode ((Char.toUpper c)::cs)

fun split-first [] = raise List.Empty
  | split-first (x :: xs) = (x, xs);

— We rely on the ML compiler to check the syntax and only provide a hands-on
parser here.
fun parse-record-declaration s =
  let
    fun is-whitespace c = member (op =) [ , \t, \n] (str c)

    fun white-space-explode  = []
      | white-space-explode s = String.fields (is-whitespace) s |> filter-out (fn x =>
x = )

    fun remove p s
      = String.explode s |> filter-out p |> String.implode

    val normalize-spaces = implode-space o white-space-explode;

    val remove-delimiter = remove (fn c => member (op =) [{,(, },),;] (str c))
    val trim = remove (fn c => str c =  )

    fun strip-word word s =
      let
        val (first, rest) = s |> white-space-explode |> split-first
      in
```

```
      if first = word then implode-space rest else error (expecting ' ^ word ^ ')
    end

  val [lhs, rhs] = s |> remove-comments |> normalize-spaces |> space-explode =
|> map normalize-spaces

  fun split-first-word s = s |> white-space-explode |> split-first ||> implode-space
  fun split-last-word s  = s |> white-space-explode |> split-last  |>> implode-space

  fun split-type s =
    let
      val (params, type-name) = s
        |> strip-word datatype
        |> split-last-word
        |>> remove-delimiter
        |>> space-explode ,
    in (params, type-name) end

  val (params, rec-name) = split-type lhs

  fun is-type-parameter s = String.isPrefix ' s orelse String.isPrefix '' s

  fun get-params s = s
    |> white-space-explode
    |> map remove-delimiter
    |> map (space-explode ,)
    |> flat
    |> filter is-type-parameter

  val (constr, field-names-params) =
      rhs
      |> split-first-word
      ||> strip-word of
      ||> remove-delimiter
      ||> space-explode ,
      ||> map (space-explode :)
      ||> map (fn [name, typ] => (trim name, get-params typ))

in ((lhs, rhs), (params, rec-name), constr, field-names-params) end

fun mk-record fields = enclose { } (commas fields);

fun make-name record = make- ^ record;
fun dest-name record = dest- ^ record;
fun map-name field = map- ^ field;
fun get-name field = get- ^ field;

fun mk-record-ass fields values =
  fields ~~ values
```

```
  |> map (fn (f,v) => f ^ " = " ^ v)
  |> commas
  |> enclose "{ }"

fun replace p q = map (fn x => if x = p then q else x)

fun mk-map constr fields n =
  let
    val (field, params) = nth fields n
    val field-names = map fst fields
    val [f] = Name.variant-list field-names ["f"]
  in
    fun "  " ^ map-name field ^
      "  " f " ( " ^ constr ^ "  " ^ mk-record field-names ^ " )  " ^ " = \n " ^
    "  " ^ constr ^ "  " ^ mk-record-ass field-names (replace field (f ^ "  " ^ field) field-names)
^ ";
  end

fun mk-maps constr fields =
 0 upto (length fields) - 1
 |> map (mk-map constr fields)
 |> cat-lines

fun mk-get constr field-name =
  fun "  " ^ get-name field-name ^ "  ( " ^ constr ^ "  " r ") = # " ^ field-name ^ "  " r";;

fun mk-gets constr fields =
  fields |> map fst |> map (mk-get constr) |> cat-lines

fun mk-record-datatype-declaration add-decl s =
  let
   val ((lhs, rhs), record as (params, rec-name), constr, fields) = parse-record-declaration
s;
    val decl = lhs ^ " = " ^ rhs ^ ";;
    val make = val "  " ^ make-name rec-name ^ " = " ^ constr ^ ";;
    val dest = fun "  " ^ dest-name rec-name ^ "  ( " ^ constr ^ "  " r ") = r;;
    val gets = mk-gets constr fields;
    val maps = mk-maps constr fields;
    fun cond-cons P a xs = if P then a::xs else xs
  in
    cat-lines (cond-cons add-decl decl [make, dest, gets, maps])
  end

end
›
```

Test the parser and its output.

ML-val ‹
```
let
```

```
  val test = datatype 'a foo = Foo of {fld1:int, fld2:'a}
in writeln (ML-Record-Antiquotation.mk-record-datatype-declaration true test) end
›
```

setup ‹
```
ML-Antiquotation.inline-embedded (Binding.make (record,here))
  (Scan.lift Parse.embedded-input >> (fn source =>
    let
      val - = ML-Lex.read-source source; — provide some markup as side effect
       val sanitized-str = fst (Input.source-content source) |> ML-Lex.tokenize |>
ML-Lex.flatten
     in ML-Record-Antiquotation.mk-record-datatype-declaration true sanitized-str
end))
›
```

3.4.4 Examples

ML-val ‹
```
structure JustAStruct = struct
datatype bar = Bar of int;

@{record ‹datatype 'a foo =
  Foo of {f1:'a (* comments are removed *),
  f2:bar }
›}

val my-foo = Foo {f1 = 1, f2 = Bar 42};
val my-foo1 = map-f1 (K (Bar 44)) my-foo;
val t = get-f1 my-foo1

val t1 = make-foo {f1=3, f2=Bar 4}
val t2 = dest-foo t1

@{record ‹datatype foo2 = Foo2 of {f3:int foo}›}

end
›
```

Note that the markup of the datatype declaration is limited to basic Lex-markup. Experiments with explicitly evaluating the datatype with e.g. the `ML` function failed. The issue there is that the `ML` function is not evaluated at the point (aka context) exactly in the text position, but with the context of the surrounding ML command.

The following examples try to illustrate the issues.

Works as expected, as every type referred to in the datatype spec is already known at the beginnig of the **ML-val** command.

ML-val ‹

val - = ML ‹datatype foo = Foo of int›;
›

The following fails as 'bar' is not known at the beginning of the context.

ML-val ‹
datatype bar = Bar of int
val - = ML ‹datatype foo = Foo of bar›
handle ERROR x => warning x
›

Adding a semicolon after the definition of bar helps. There the semicolon marks the end of an "evaluation / compilation chunk". So `ML` is evaluated within an already augmented ML-context that knows about *bar*.

ML-val ‹
datatype bar = Bar of int;
val - = ML ‹datatype foo = Foo of bar›
›

In the context of a structure the semicolon is not enough.

ML-val ‹

structure InAStruct =
struct
datatype bar = Bar of int;

val - = ML ‹datatype foo = Foo of bar›
handle ERROR x => warning x
end

›

end

theory *Misc-Antiquotation*
imports *Main*
begin

3.5 Various Antiquotations

3.5.1 Antiquotations for terms with schematic variables

setup ‹
let
val parser = Args.context -- Scan.lift Parse.embedded-inner-syntax
fun pattern (ctxt, str) =
str |> Proof-Context.read-term-pattern ctxt
|> ML-Syntax.print-term
|> ML-Syntax.atomic

```
in
 ML-Antiquotation.inline @{binding pattern} (parser >> pattern)
end
›

ML ‹@{pattern λ(a,b,c). ?g a b c}›

setup ‹
let
type style = term -> term;

fun pretty-term-style ctxt (style: style, t) =
  Document-Output.pretty-term ctxt (style t);

val basic-entity = Document-Output.antiquotation-pretty-source;

in
(* Document antiquotation that allows schematic variables *)
(basic-entity binding ‹pattern› (Term-Style.parse -- Args.term-pattern) pretty-term-style)
end
›
```

3.5.2 Antiquotation for types with schematic variables

```
setup ‹
let
  val parser = Args.context -- Scan.lift Parse.embedded-inner-syntax
  fun typ-pattern (ctxt, str) =
    let
      val ctxt' = Proof-Context.set-mode Proof-Context.mode-schematic ctxt
    in
      str |> Proof-Context.read-typ ctxt'
          |> ML-Syntax.print-typ
          |> ML-Syntax.atomic
    end
in
 ML-Antiquotation.inline @{binding typ-pattern} (parser >> typ-pattern)
end
›

ML ‹@{typ-pattern ?'a list}›
ML ‹Sign.typ-instance @{theory} (@{typ 'a list}, @{typ-pattern ?'a list})›


end
```

Chapter 4

Proof Tools

theory *Tuple-Tools*

imports *Main*
Misc-Antiquotation
TermPatternAntiquote
ML-Fun-Cache

begin

4.1 Tools for handling Tuples

The tools are supposed to help simplifiying expressions containing *case-prod* constructions, like $\lambda(w,\ x,\ y,\ z).\ f\ w\ x\ y\ z$.

- Simprocs for single step splitting and simplification of tuples / *case-prods* instead of repeated application of the built-in variants for pairs
- Looper for the simplifier to instantiate f in $f\ (a,\ b,\ c)$ with $\lambda(a,\ b,\ c).\ g\ a\ b\ c$

4.1.1 Antiquotations for terms with schematic variables

setup ‹
let
val parser = Args.context −− Scan.lift Parse.embedded-inner-syntax
fun pattern (ctxt, str) =
str |> Proof-Context.read-term-pattern ctxt
|> ML-Syntax.print-term
|> ML-Syntax.atomic
in
ML-Antiquotation.inline @{*binding pattern*} (*parser >> pattern*)
end

›

ML ‹@{*pattern* λ(*a*,*b*,*c*). *?g a b c*}›

setup ‹
let
type style = *term* −> *term*;

fun pretty-term-style ctxt (*style*: *style*, *t*) =
  Document-Output.pretty-term ctxt (*style t*);

val basic-entity = *Document-Output.antiquotation-pretty-source*;

in
(∗ *Document antiquotation that allows schematic variables* ∗)
(*basic-entity* ***binding*** ‹*pattern*› (*Term-Style.parse* −− *Args.term-pattern*) *pretty-term-style*)
end
›

Unification only works modulo ordinary beta reduction but does not succeed on tupled-beta reduction. For example the simplifier will not find an instatiation for the variable *?c*

schematic-goal ⋀*a b*. (*case* (*a*, *b*) *of* (*x*, *y*) ⇒ *f x y*) ≡ *?c* (*a*, *b*)
  apply *simp*
  oops

Such equations can typically occur in congruence rules when a pair is splitted by (⋀*x*. *PROP ?P x*) ≡ (⋀*a b*. *PROP ?P* (*a*, *b*)).

lemma *tuple-up-eq-trivial*: *f r* ≡ (λ*x*. *f x*) *r*
  by *simp*

thm *tuple-up-eq-trivial*
lemma *tuple-up-eq2*: *f* (*fst r*) (*snd r*) ≡ (*case r of* (*x1*, *x2*) ⇒ *f x1 x2*)
  by (*simp add*: *split-def*)

Constant to explictly trigger splitting by simproc `SPLIT_simproc` below

definition *SPLIT* :: *'a*::{} ⇒ *'a*
  where *SPLIT P* ≡ *P*

lemma *SPLIT-cong*: *PROP SPLIT P* ≡ *PROP SPLIT P*
  by *simp*

lemma *case-prod-out*: (λ*r*. *f* ((λ(*a*,*b*). (*g a b*)) *r*)) = (λ(*a*, *b*). *f* (*g a b*))
  apply (*rule ext*)
  apply (*simp add*: *split-paired-all*)
  done

lemma *case-prod-eta-contract*: (λ*x*. (*case-prod s*) *x*) ≡ (*case-prod s*)
  by *simp*

definition *ETA-TUPLED* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$
where *ETA-TUPLED f* $\equiv f$

lemma *ETA-TUPLED-trans*: $f \equiv g \Longrightarrow$ *ETA-TUPLED f* $\equiv g$
by (*simp add*: *ETA-TUPLED-def*)

Add the congruence rule *SPLIT PROP ?P* $\equiv$ *SPLIT PROP ?P* to the simpset together with the simproc to avoid descending into P before the split.

ML ‹
```
structure Bin =
struct

type Bin = bool list

fun bin-of-int n =
  if n < 0 then error (bin: ^ cannot convert negativ number ' ^ string-of-int n ^ ')
  else map (fn 0 => false | - => true) (radixpand (2,n));

fun int-of-bin bs =
  let
    fun int-of pos [] = 0
      | int-of pos (false::bs) = int-of (2 * pos) bs
      | int-of pos (true::bs) = pos + int-of (2 * pos) bs
  in int-of 1 (rev bs) end

fun string-of-bin bs =
  let
    fun bit true = #1
      | bit false = #0
  in
     0b ^ String.implode (map bit bs)
  end

val bin-ord = list-ord bool-ord;
structure Bintab = Table(type key = Bin val ord = bin-ord);

val - = @{assert} (int-of-bin (bin-of-int 1) = 1)
val - = @{assert} (int-of-bin (bin-of-int 22) = 22)
val - = @{assert} (int-of-bin (bin-of-int 43) = 43)
val - = @{assert} (is-none (try bin-of-int ~1))
val - = @{assert} (string-of-bin (bin-of-int 0) = 0b0)
val - = @{assert} (string-of-bin (bin-of-int 1) = 0b1)
val - = @{assert} (string-of-bin (bin-of-int 7) = 0b111)
val - = @{assert} (string-of-bin (bin-of-int 23) = 0b10111)
end
```
›

ML ‹
structure Tuple-Tools = struct

— We document the effect of functions using assertions

— t has to be type correct and be equivalent to ct modulo beta/eta reduction and typing
fun assert-cterm′ ctxt t ct =
  let
    val t1 = Envir.beta-eta-contract (Thm.term-of (Thm.cterm-of ctxt t))
    val t2 = Envir.beta-eta-contract (Thm.term-of ct)
  in @{assert} (Term.aconv-untyped (t1, t2)) end

val assert-cterm = assert-cterm′ @{context}

fun assert-term t1 t2 = @{assert} (t1 = t2)

fun gen-mk-prod (t1, T1) (t2, T2) =
  (HOLogic.pair-const T1 T2 $ t1 $ t2, HOLogic.mk-prodT (T1, T2))

fun gen-mk-tuple [] = (@{term ()}, @{typ unit})
  | gen-mk-tuple [(t, T)] = (t, T)
  | gen-mk-tuple (t::ts) = gen-mk-prod t (gen-mk-tuple ts)

— Functions to create various terms on tuples. * The primed variants take a list of element types. * The unprimed variants only take a number (arity of the tuple) and generate default element types.

fun mk-elT-named n i = TFree (n ^ string-of-int i, @{sort type});
val mk-elT = mk-elT-named 'a
fun mk-elT′ Ts i = nth Ts (i−1);
fun mk-tupleT′ Ts i = HOLogic.mk-tupleT (drop (i − 1) Ts);
fun mk-tupleT n i = mk-tupleT′ (map mk-elT (1 upto n)) i;

fun mk-el-name-named x i = x ^ string-of-int i;
val mk-el-name = mk-el-name-named x

fun mk-el′ Ts i = Free (mk-el-name i, mk-elT′ Ts i);
fun mk-el-named′ x Ts i = Free (mk-el-name-named x i, mk-elT′ Ts i);
fun mk-el-named x i = Free (mk-el-name-named x i, mk-elT i);
fun mk-el i = Free (mk-el-name i, mk-elT i);

fun mk-tuple-bounds n = (1 upto n) |> map (fn i => (Bound (n − i), mk-elT i))
|> gen-mk-tuple |> fst
val mk-tuple-packed-name = r;

```
fun mk-tuple n i = HOLogic.mk-tuple (map mk-el (i upto n));
fun mk-tuple-named x n i = HOLogic.mk-tuple (map (mk-el-named x) (i upto n));
fun mk-tuple' Ts n i = HOLogic.mk-tuple (map (mk-el' Ts) (i upto n))
fun mk-tuple-named' x Ts n i = HOLogic.mk-tuple (map (mk-el-named' x Ts) (i
upto n))

fun mk-tuple-packed n = Free (mk-tuple-packed-name, mk-tupleT n 1);

fun mk-P n = Free (P, map mk-elT (1 upto n) ---> @{typ bool});
fun mk-P-frees n = list-comb (mk-P n, map mk-el (1 upto n));
fun mk-P-bounds n = list-comb (mk-P n, map Bound (n-1 downto 0));

fun mk-Prop-P n = Free (P, mk-tupleT n 1 --> @{typ prop});

fun mk-fun (name,resT) i n = Free (name, map mk-elT (i upto (i+n-1)) --->
resT);
fun mk-fun-bounds (name, resT) i n = list-comb (mk-fun (name, resT) i n, map
Bound (n-1 downto 0));

fun mk-fun-with-types Ts (name,resT) = Free (name, map (mk-elT' Ts) (1 upto
(length Ts)) ---> resT);
fun mk-fun-bounds-with-types Ts (name, resT) = list-comb (mk-fun-with-types Ts
(name, resT), map Bound ((length Ts - 1) downto 0));

val mk-f-resT = @{typ 'b};
val mk-f-name = f;
fun mk-f n = Free (mk-f-name, map mk-elT (1 upto n) ---> mk-f-resT);
fun mk-f-frees n = list-comb (mk-f n, map mk-el (1 upto n));
fun mk-f-bounds n = list-comb (mk-f n, map Bound (n-1 downto 0));



fun mk-f-tupled n = Free (mk-f-name, mk-tupleT n 1 --> mk-f-resT)

fun mk-selT n i = mk-tupleT n 1 --> mk-elT i;
fun mk-selT' Ts i = mk-tupleT' Ts i --> mk-elT' Ts i;
fun mk-snd n i = Const (@{const-name ‹snd›}, mk-tupleT n i --> mk-tupleT n
(i+1));
fun mk-snd' Ts i = Const (@{const-name ‹snd›}, mk-tupleT' Ts i --> mk-tupleT'
Ts (i+1));
fun mk-fst n i = Const (@{const-name ‹fst›}, mk-tupleT n i --> mk-elT i);
fun mk-fst' Ts i = Const (@{const-name ‹fst›}, mk-tupleT' Ts i --> mk-elT' Ts
i);

fun mk-snds' Ts i r =
  let
    val n = length Ts;
  in
    if n = 1 then r
```

```
    else
      if i = (n-1) then mk-snd' Ts 1 $ r
      else mk-snd' Ts (n - i) $ (mk-snds' Ts (i+1) r)
  end;

fun mk-snds n i r =
  if i = (n-1) then mk-snd n 1 $ r
  else mk-snd n (n - i) $ (mk-snds n (i+1) r)

fun mk-sel' Ts r i =
  let
    val n = length Ts;
  in
    if n = 1 then r
    else
      if i = 1 then mk-fst' Ts 1 $ r
      else if i = n then mk-snds' Ts 1 r
      else mk-fst' Ts i $ mk-snds' Ts (n - i + 1) r
  end;

val - = assert-term (mk-sel' ([@{typ 'a}, @{typ 'b}, @{typ 'c}, @{typ 'd}]) (Bound
0) 4)
(Const (@{const-name snd}, @{typ 'c × 'd ⇒ 'd}) $
  (Const (@{const-name snd}, @{typ 'b × 'c × 'd ⇒ 'c × 'd}) $
    (Const (@{const-name snd}, @{typ 'a × 'b × 'c × 'd ⇒ 'b × 'c × 'd}) $ Bound
0)))

val - = assert-term (mk-sel' ([@{typ 'a}, @{typ 'b}, @{typ 'c}, @{typ 'd}]) (Bound
0) 3)
(Const (@{const-name fst}, @{typ 'c × 'd ⇒ 'c}) $
  (Const (@{const-name snd}, @{typ 'b × 'c × 'd ⇒ 'c × 'd}) $
    (Const (@{const-name snd}, @{typ 'a × 'b × 'c × 'd ⇒ 'b × 'c × 'd}) $ Bound
0)))

fun mk-sel n r i =
  if i = 1 then mk-fst n 1 $ r
  else if i = n then mk-snds n 1 r
  else mk-fst n i $ mk-snds n (n - i + 1) r

fun mk-eq n r i =
  HOLogic.mk-eq (mk-sel n r i, mk-el i);

fun mk-case-prod-from T t i n =
  let
    fun mk-cp n i =
      if (n - i) <= 1 then
        HOLogic.case-prod-const (mk-elT i, mk-elT n, T) $
          (Abs (mk-el-name i, mk-elT i, Abs (mk-el-name n, mk-elT n, t)))
    else
```

```
      HOLogic.case-prod-const (mk-elT i, mk-tupleT n (i+1), T) $
        (Abs (mk-el-name i, mk-elT i, mk-cp n (i+1)))
  in mk-cp (n+i−1) i end;



fun mk-case-prod T t n =
  mk-case-prod-from T t 1 n

fun mk-case-prod-with-types Ts T t =
  let
    val n = length Ts
    fun mk-cp n i =
      if (n − i) <= 1 then
        HOLogic.case-prod-const (mk-elT' Ts i, mk-elT' Ts n, T) $
          (Abs (mk-el-name i, mk-elT' Ts i, Abs (mk-el-name n, mk-elT' Ts n, t)))
    else
        HOLogic.case-prod-const (mk-elT' Ts i, mk-tupleT' Ts (i+1), T) $
          (Abs (mk-el-name i, mk-elT' Ts i, mk-cp n (i+1)))
  in mk-cp n 1 end;

fun mk-case-prod-P n = mk-case-prod @{typ bool} (mk-P-bounds n) n;
fun mk-case-prod-f n = mk-case-prod mk-f-resT (mk-f-bounds n) n;
fun mk-case-prod-fun (name, resT) i n = mk-case-prod-from resT (mk-fun-bounds
(name, resT) i n) i n;
fun mk-case-prod-fun-with-types Ts (name, resT) = mk-case-prod-with-types Ts
resT (mk-fun-bounds-with-types Ts (name, resT));
fun mk-case-prod-f-tupled-bounds n = mk-case-prod mk-f-resT (mk-f-tupled n $
mk-tuple-bounds n) n

fun argTs-of-TVar n [(TVar ((x,-),-))] = map (mk-elT-named x) (1 upto n)
  | argTs-of-TVar - Ts = Ts

fun mk-case-prod' name T n =
  if n <= 1 then Free (name, T)
  else
    let
      val (domT, rangeT) = dest-funT T
      val argTs = HOLogic.strip-tupleT domT |> argTs-of-TVar n
    in
      if length argTs = n
       then mk-case-prod-fun-with-types argTs (name, rangeT) (* Type was already
instantiated so we take those types *)
      else mk-case-prod-fun (name, rangeT) 1 n
    end

fun mk-tuple-from-type name T n =
  if n <= 1 then Free (name, T)
  else
```

```
    let
      val Ts = HOLogic.strip-tupleT T
    in
      if length Ts = n
      then mk-tuple-named' name Ts n 1
      else mk-tuple-named name n 1
    end

fun mk-tuple-or-case-prod name T n =
  if is-some (try dest-funT T)
  then mk-case-prod' name T n
  else mk-tuple-from-type name T n


fun mk-f-sels n r = list-comb (mk-f n, map (mk-sel n r) (1 upto n));

fun mk-tuple-up-eq n =
  let
    val r = mk-tuple-packed n;
    val lhs = mk-f-sels n r;
    val rhs = mk-case-prod-f n $ r;
  in Logic.mk-equals (lhs, rhs) end

val - = assert-cterm (mk-tuple-up-eq 3)
  @{cterm f (fst r) (fst (snd r)) (snd (snd r)) ≡ case r of (x1, x2, x3) ⇒ f x1 x2
x3}

fun mk-tuple-up-eq-thm ctxt n =
  let
    val fixes = [mk-tuple-packed n,mk-f n]
      |> map (fn (Free (x, -)) => x);
  in
    Goal.prove ctxt fixes [] (mk-tuple-up-eq n) (fn - =>
    simp-tac (put-simpset HOL-basic-ss ctxt addsimps [mk-meta-eq @{thm split-def}])
1)
  end


fun mk-tuple-case-eq n =
  let
     val lhs = mk-case-prod-f n $ mk-tuple n 1
     val rhs = mk-f-frees n
  in
    Logic.mk-equals (lhs, rhs)
  end

val - =  assert-cterm (mk-tuple-case-eq 3)
 @{cterm case (x1, x2, x3) of (x1, x2, x3) ⇒ f x1 x2 x3 ≡ f x1 x2 x3}
```

```
fun mk-tuple-case-eq-thm ctxt n =
  let
    val fixes = mk-f n::map mk-el (1 upto n)
      |> map (fn (Free (x, -)) => x)
  in
    Goal.prove ctxt fixes [] (mk-tuple-case-eq n) (fn - =>
    simp-tac (put-simpset HOL-basic-ss ctxt addsimps [@{thm Product-Type.prod.case}])
1)
  end



fun num-case-prod t =
  case strip-comb t of
   (Const (@{const-name case-prod},-) , (Abs (-, -, cp))::-) => 1 + num-case-prod
cp
  | (Const (@{const-name case-prod},-) , -) => 1
  | - => 0;

val strip-uu = prefix strip- Name.uu-

fun strip-case-prod t =
    case strip-comb t of
      (Const (@{const-name case-prod},-), Abs (x, xT, (Abs (y, yT, -)))::-) =>
         [(x,xT), (y, yT)]
    | (Const (@{const-name case-prod},cpT), Abs (x, xT, cp)::-) =>
        (case strip-comb cp of
          (Const (@{const-name case-prod},-), -) => (x,xT)::strip-case-prod cp
        | - => (* eta expand lost abstraction in case-prod *)
              ([(x,xT), (strip-uu, nth (binder-types cpT) 1)]))
    | (Const (@{const-name case-prod}, cpT), cp::-) =>
          (* eta expand lost abstractions in case-prod *)
          let
            val [xT, yT] = take 2 (binder-types (domain-type cpT))
          in [(strip-uu, xT), (strip-uu, yT)] end
    | - => [];

fun strip-case-prod' (Abs (x, xT, -)) = [(x, xT)]
  | strip-case-prod' t = strip-case-prod t

fun mk-split-tupled-all-lhs n =
  Logic.list-all ([(mk-tuple-packed-name, mk-tupleT n 1)], (mk-Prop-P n) $ Bound
0)

fun mk-prod-bound (i,iT) (t,T) =
  HOLogic.pair-const iT T $ Bound i $ t

fun mk-prod-bounds [(i, iT), (j, jT)] = mk-prod-bound (i,iT) (Bound j, jT)
  | mk-prod-bounds ((i,iT)::xs) =
```

```
    let
      val t = mk-prod-bounds xs
    in mk-prod-bound (i,iT) (t, fastype-of t) end

fun mk-split-tupled-all-rhs n =
  let
    val - = @{assert} (n >= 2)
  in
    Logic.list-all (map (fn i => (mk-el-name i, mk-elT i)) (1 upto n),
      (mk-Prop-P n) $ mk-prod-bounds (map (fn i => (n-i, mk-elT i)) (1 upto
n)))
  end

fun mk-split-tupled-all n =
  Logic.mk-equals (mk-split-tupled-all-lhs n, mk-split-tupled-all-rhs n)

val - = assert-cterm (mk-split-tupled-all 3)
 @{cterm (⋀r. PROP P r) ≡ (⋀x1 x2 x3. PROP P (x1, x2, x3))}

fun mk-split-tupled-all-thm ctxt n =
  Goal.prove ctxt [P] [] (mk-split-tupled-all n) (fn - => simp-tac
    (put-simpset HOL-basic-ss ctxt addsimps @{thms split-paired-all}) 1)

fun mk-eta-tupled-eq n =
  let
    val lhs = mk-f-tupled n
    val rhs = mk-case-prod-f-tupled-bounds n
    val eq = Logic.mk-equals (lhs, rhs)
  in
    eq
  end

fun mk-eta-tupled-eq-thm ctxt n =
      Goal.prove @{context} [mk-f-name] [] (mk-eta-tupled-eq n) (fn - =>
        simp-tac @{context} 1) |> Thm.transfer' ctxt


structure Data = Generic-Data(
  type T = (thm * thm * thm * thm) list;
  val empty = [];
  fun merge (thms1, thms2) = if length thms1 >= length thms2 then thms1 else
thms2;
);


fun get-tuple-up-eq-thm ctxt n =
  if n <= 1 then @{thm tuple-up-eq-trivial}
  else
    Thm.transfer' ctxt (#1 (nth (Data.get (Context.Proof ctxt)) (n - 2)))
```

```
    handle Subscript => mk-tuple-up-eq-thm ctxt n

fun get-split-tupled-all-thm ctxt n =
  if n <= 1 then error (get-split-tupled-all only makes sense for n >= 2)
  else
    Thm.transfer' ctxt (#2 (nth (Data.get (Context.Proof ctxt)) (n - 2)))
    handle Subscript => mk-split-tupled-all-thm ctxt n

fun get-tuple-case-eq-thm ctxt n =
  if n <= 1 then error (get-tuple-case-eq-thm only makes sense for n >= 2)
  else
    Thm.transfer' ctxt (#3 (nth (Data.get (Context.Proof ctxt)) (n - 2)))
    handle Subscript => mk-tuple-case-eq-thm ctxt n

fun get-eta-tupled-eq-thm ctxt n =
  if n <= 1 then @{thm reflexive}
  else
    Thm.transfer' ctxt (#4 (nth (Data.get (Context.Proof ctxt)) (n - 2)))
    handle Subscript => mk-eta-tupled-eq-thm ctxt n

fun liberal-zip (x::xs) (y::ys) = (x,y)::liberal-zip xs ys
  | liberal-zip - [] = []
  | liberal-zip [] - = []

fun get-first-subterm P t =
  if P (Term.head-of t)
  then SOME t
  else
    case t of
      (t1 $ t2) => (case get-first-subterm P t1 of
                      NONE => get-first-subterm P t2
                    | some => some)
    | Abs (-, - , t) => get-first-subterm P t
    | - => NONE
```

— Split a tuple completely in one step, instead of repeated applications of ($\bigwedge x$. *PROP ?P x*) ≡ ($\bigwedge a\ b$. *PROP ?P* (*a*, *b*))

```
val split-tupled-all-simproc =
 Simplifier.make-simproc @{context}{name = split-tupled-all-simproc, kind = Sim-
proc, identifier = [],
   lhss = [Proof-Context.read-term-pattern @{context} ⋀r. PROP ?P r],
    proc = fn - => fn ctxt => fn ct =>
     let
        fun get-tuple-arity (Const (@{const-name Pure.all},-) $ (Abs (-, T, -))) =
            length (HOLogic.flatten-tupleT T)
          | get-tuple-arity - = 0;
        val t = Thm.term-of ct
        val n = get-tuple-arity t
     in
```

```
      if n >= 2
      then
        let
          fun is-case-prod (Const (@{const-name case-prod}, -)) = true
            | is-case-prod - = false

          fun guess-names t = case get-first-subterm is-case-prod t of
                SOME t' => map fst (strip-case-prod t') |> filter-out (fn n => n
= strip-uu)
              | - => []

          val thm = get-split-tupled-all-thm ctxt n
                |> Drule.rename-bvars (liberal-zip (map mk-el-name (1 upto n))
(guess-names t))
        in SOME thm end
      else NONE
    end
  }
```

— Simplify a *case-prod* applied to a tuple constructed from *Pair* in one step instead of repeated application of (*case* (*?x1.0*, *?x2.0*) *of* (*x*, *xa*) ⇒ *?f x xa*) = *?f ?x1.0 ?x2.0* or (*case* (*?a*, *?b*) *of* (*c*, *d*) ⇒ *?f c d*) = *?f ?a ?b*

```
val tuple-case-simproc =
  Simplifier.make-simproc @{context} {name = tuple-simproc, kind = Simproc,
identifier = [],
   lhss = [Proof-Context.read-term-pattern @{context} case-prod ?X (?x,?y)],
    proc = fn - => fn ctxt => fn ct =>
     let
       fun get-tuple-arity
            (t as (Const (@{const-name Product-Type.prod.case-prod}, -) $ - $
                  (p as (Const (@{const-name Pair}, -) $ - $ -)))) =
              Int.min (num-case-prod t + 1, length (HOLogic.strip-tuple p))
         | get-tuple-arity - = 0;
       val n = get-tuple-arity (Thm.term-of ct)
     in
       if n >= 2
       then SOME (get-tuple-case-eq-thm ctxt n)
       else NONE
     end
  }
```

— Instantiaten for tupled-beta reduction

```
local
fun strip-all (Const (@{const-name Pure.all}, - ) $  Abs (x,T,t)) =
  let
    val (vTs, bdy) = strip-all t
  in ((x,T)::vTs, bdy) end
```

```
  | strip-all t = ([], t)

fun dest-pair-bound (Const (@{const-name Product-Type.Pair}, -) $  Bound i $ p)
=
      i::dest-pair-bound p
  | dest-pair-bound (Bound i) = [i]

fun lookup env i = nth env i

fun mk-name env (Bound i) = fst (lookup env i)
  | mk-name env (Var ((x,-),-)) = x
  | mk-name env (Free (x,-)) = x
  | mk-name env - = Name.uu-;

fun mk-var env (Bound i) = SOME (lookup env i)
  | mk-var - - = NONE

fun is-Pair (Const (@{const-name Product-Type.Pair}, -) $ - $ -) = true
  | is-Pair - = false;

fun dest-tuple env (Const (@{const-name Product-Type.Pair}, -) $ t $ p)
    = mk-name env t :: (if is-Pair p then dest-tuple env p else [mk-name env p])

fun dest-tuple' env (Const (@{const-name Product-Type.Pair}, -) $ t $ p)
    = mk-var env t :: (if is-Pair p then dest-tuple' env p else [mk-var env p])

fun mk-case-prod' T seed [(n1,T1), (n2, T2)] =
    (Const (@{const-name case-prod}, (T1 --> T2 --> T) --> HOLogic.mk-prodT
(T1, T2) --> T)$
          Abs (n1, T1, Abs (n2, T2, seed)),  HOLogic.mk-prodT (T1, T2))
  | mk-case-prod' T seed ((n1,T1)::bs) =
      let
        val (t',T2) = mk-case-prod' T seed  bs
        in (Const (@{const-name case-prod}, (T1 --> T2 --> T) --> HO-
Logic.mk-prodT (T1, T2) --> T)$
          Abs (n1, T1, t'), HOLogic.mk-prodT (T1, T2))
      end

fun get-tuple-var env (Var v $
     (p as (Const (@{const-name Product-Type.Pair}, -)$ -$ -))) =
       [(env, v, dest-tuple env p)]
  | get-tuple-var env (t1 $ t2) =
      (case get-tuple-var env t1 of
        [] => get-tuple-var env t2
      | xs => xs)
  | get-tuple-var env (Abs (x, T, t)) =
      get-tuple-var ((x,T)::env) t
  | get-tuple-var env - = [];
```

```
fun dest-tuple'' env t = case try (dest-tuple' env) t of
      SOME xs => xs
    | NONE => [NONE]

fun flatten-upto-first-tuple xxs =
  let
    val (upto-one, more) = chop-prefix (fn xs => length xs <= 1) xxs
  in
    if null more then ([],[])
    else (flat (upto-one), hd more)
  end

(* finds function vars that are applied to a tuple in some argument e.g.
 *   ?f x y (a, b)
 *)
fun get-tuple-vars env (t as (- $ -)) =
   (case strip-comb t of
      (Var v, ts) =>
        let
          val (args, tuple-args) = map (dest-tuple'' env) ts |> flatten-upto-first-tuple
          val rest = map (get-tuple-vars env) ts |> flat
        in
          if null tuple-args then rest
          else (env, v, (args, tuple-args)) :: rest
        end
    | (t, ts) => map (get-tuple-vars env) (t::ts) |> flat)
  | get-tuple-vars env (Abs (x, T, t)) =
      get-tuple-vars ((x,T)::env) t
  | get-tuple-vars - - = [];


fun get-distinct-tuple-vars env t =
  t |> get-tuple-vars env |> distinct (fn ((-,v1,-),(-,v2,-)) => v1 = v2)

in


val list-abs = fold-rev Term.abs

fun map-filter2 - [] [] = []
  | map-filter2 f (x :: xs) (y :: ys) =
      let
        val vs = map-filter2 f xs ys
      in case f x y of SOME v => v :: vs | - => vs end
  | map-filter2 - - - = raise ListPair.UnequalLengths;

fun gen-calc-inst bound get-tuple-vars ctxt max-idx t = try ‹
   get-tuple-vars [] t
```

```
  |> map (fn (-,((x,i), pT), (args, bs)) =>
    let
      val (all-argTs, finalT) = strip-type pT
      val (argTs, ([tupleT], rest-argTs)) = all-argTs |> chop (length args) ||> chop
1

      fun mk-vars bound vs Ts = (vs ~~ Ts)
        |> map (fn (SOME (x, -), T) => (true, (x, T))
              | (NONE, T) => if bound then (false, (Name.uu-, T)) else (true,
(Name.uu-, T)))

       fun filter-tagged tags xs = map-filter2 (fn b => fn x => if b then SOME x
else NONE) tags xs

      val tagged-args = mk-vars false args argTs
      val args = map snd tagged-args
      val tagged-tuple-args = mk-vars bound bs (HOLogic.flatten-tupleT tupleT)
      val tuple-args = map snd tagged-tuple-args
      val tagged-vars = tagged-args @ tagged-tuple-args
      val tags = map fst tagged-vars
      val vars = map snd tagged-vars

      val bnds = length vars - 1 downto 0 |> filter-tagged tags
      val Ts = map snd vars |> filter-tagged tags
      val rT = rest-argTs ---> finalT
      val newVar = Var ((x,max-idx+1), Ts ---> rT)
      val seed = Term.list-comb (newVar, map Bound bnds)
      val case-prod = fst (mk-case-prod' rT seed tuple-args)
      val inst = list-abs args case-prod
    in ((x,i), Thm.cterm-of ctxt inst) end
  ) catch - => []›

fun calc-first-inst bound = gen-calc-inst bound (fn env => take 1 o get-tuple-vars
env)
val calc-inst = calc-first-inst true
val calc-inst' = calc-first-inst false
val calc-insts = gen-calc-inst false get-distinct-tuple-vars

end


(* For a goal with a schematic variable of the form

   ?f (x1,x2,...,xn)

   as it might occur in an congruence rule

      ⋀... x1 ... xn ... . ?X ≡ ?f (x1,x2,...,xn) ...
```

```
  instantiate ?f to λ(x1,x2,...,xn). ?f' x1 x2 ... xn ...

  Currently only instantiates the first such ?f it finds by a canonical traversal of
the term.
  The tupled argument does not have to be the first argument, also works for the
more general case
  like ?f t1 t2 (x1,x2,...,xn) t3 t4

  tuple-inst-tac can be used as a looper.
*)

fun cond-trace ctxt s =
  if Config.get ctxt Simplifier.simp-trace then tracing s else ()

fun tuple-inst-tac ctxt i = SUBGOAL (fn (trm, -) => fn st =>
  let
    val max-idx = Thm.maxidx-of-cterm (Thm.cprop-of st)
  in
    case calc-inst ctxt max-idx trm of
      [] => (cond-trace ctxt tuple-inst-tac: no instantiation found; no-tac st)
    | inst => let
                val - = cond-trace ctxt (tuple-inst-tac: ^@{make-string} inst);
              in PRIMITIVE (Drule.infer-instantiate ctxt inst) st end
  end) i;

fun tuple-insts ctxt thm =
  let
    val max-idx = Thm.maxidx-of-cterm (Thm.cprop-of thm)
  in
    case calc-insts ctxt max-idx (Thm.prop-of thm) of
      [] => thm
    | inst => let
                val - = cond-trace ctxt (tuple-inst-tac: ^@{make-string} inst);
              in Drule.infer-instantiate ctxt inst thm end
  end;

(* case-prod ?f (Pair ...) ... *)
fun is-case-prod-app-Pair t =
  case Term.strip-comb t of
   (Const (@{const-name case-prod}, -), - :: maybe-pair :: -) => (case Term.strip-comb
maybe-pair of
      (Const (@{const-name Pair}, -), -) => true | - => false)
  | - => false

(* apply beta reduction for Pairs *)
fun mksimps ctxt thm =
  let
    val thm' = (if exists-subterm is-case-prod-app-Pair (Thm.prop-of thm)
      then
```

```
        let
         val thm' = Simplifier.simplify (put-simpset HOL-ss ctxt addsimps @{thms
Product-Type.prod.case}) thm
          val - = cond-trace ctxt (Tuple-Tools.mksimps: ^ @{make-string} thm')
        in thm' end (* (%(x,y). f x y) (a,b) = f a b *)
      else thm)
  in (Simpdata.mksimps Simpdata.mksimps-pairs) ctxt thm'
  end

fun beta-tupled-conv ctxt arity ct =
  if arity <= 1 then
    Conv.all-conv ct
  else
    let
      val case-eq-thm = get-tuple-case-eq-thm ctxt arity
    in
      Conv.rewr-conv case-eq-thm ct
    end

fun app-beta-tupled-conv ctxt arity f x =
  Thm.apply f x |> beta-tupled-conv ctxt arity

fun beta-tupled ctxt arity f x = app-beta-tupled-conv ctxt arity f x |> Thm.rhs-of

val SPLIT-simproc =
  Simplifier.make-simproc @{context} {name = SPLIT-simproc, kind = Simproc,
identifier = [],
    lhss = [Proof-Context.read-term-pattern @{context} PROP SPLIT ?P,
            Proof-Context.read-term-pattern @{context} SPLIT ?P],
    proc = fn - => fn ctxt => fn ct =>
     let
       fun get-tuple-arity (Const (@{const-name SPLIT}, -)$
             (Const (@{const-name Pure.all},-) $ (Abs (-, T, -)))) =
           length (HOLogic.flatten-tupleT T)
         | get-tuple-arity - = 0;
       val t = Thm.term-of ct
       val n = get-tuple-arity t
     in
       if n >= 2
       then
         let
           fun is-case-prod (Const (@{const-name case-prod}, -)) = true
             | is-case-prod - = false

           fun guess-names t = case get-first-subterm is-case-prod t of
                 SOME t' => map fst (strip-case-prod t') |> filter-out (fn n => n
= strip-uu)
               | - => []
```

```
        val split-thm = get-split-tupled-all-thm ctxt n
              |> Drule.rename-bvars (liberal-zip (map mk-el-name (1 upto n))
(guess-names t))

        val case-eq-thm = get-tuple-case-eq-thm ctxt n

        val conv = Conv.rewr-conv @{thm SPLIT-def}
            then-conv Conv.rewr-conv split-thm
              then-conv (Conv.bottom-conv (K (Conv.try-conv (Conv.rewr-conv
case-eq-thm))) ctxt)

      in SOME (conv ct) end
    else SOME @{thm SPLIT-def}
  end
 }

fun asserts [] x = x
  | asserts ((cond, msg)::xs) x = if cond x then asserts xs x else error (msg x)

structure Intlisttab = Table(type key = int list val ord = list-ord int-ord);

fun comb-product [] = [[]]
  | comb-product (xs:: yss) =
    let
      val prods = comb-product yss
      val prodss = map (fn x => map (fn ps => x::ps) prods) xs
    in
      flat prodss
    end

val - = @{assert} (comb-product [[1,2],[3,4],[5,6]] =
     [[1, 3, 5], [1, 3, 6], [1, 4, 5], [1, 4, 6], [2, 3, 5], [2, 3, 6], [2, 4, 5], [2, 4,
6]])

(*
 * Generates a custom rule with the tuple arity of an abstracted variable. The
position of the
 * abstraction is specified in the pattern (which is also used as simproc pattern).
Every real variable
 * in that pattern (besides the dummy pattern -) will be considered as an abstraction
to split. E.g.
 * thm: ⋀v. P (?y v) ⟹ seq ?x (λx. ?y x) with pattern seq - ?Y
 * When this is applied to a term seq x (λ(a,b,c). ?y a b c) a rule of arity 3 will be
generated:
 * ⋀a b c. P (?y a b c) ⟹ seq ?x (λ(a, b, c). ?y a b c) which can be matched with
the term.
 *)
fun split-rule-simproc ctxt name pattern rule =
  let
```

```
  val pat =  Proof-Context.read-term-pattern ctxt pattern
  fun dest-eq thm =
    let
      val @{term-pat Trueprop (?lhs = ?rhs)} = thm |> Thm.concl-of
    in (lhs, rhs) end

  val lhs = rule |> dest-eq |> fst

  val (tenv, term-env) = Pattern.match (Proof-Context.theory-of ctxt) (pat, lhs)
(Vartab.empty, Vartab.empty)
  fun get-vars env (Abs (x, T, t)) = get-vars (env @ [(x,T)]) t
    | get-vars env t =
      (case Term.strip-comb t of
        (Var x, args) => (env, x, args) :: flat (map (get-vars env) args)
      | (-, args) => flat (map (get-vars env) args))


  fun msg s x = split-rule-simproc:  ^ s

  fun is-dummy ((n, -)) = (n = -dummy-)
  val rule-subterms = Vartab.dest term-env |> filter-out (is-dummy o fst) |> map
(snd o snd)
  val rule-vars = flat (map (get-vars []) rule-subterms)
    |> map (asserts [
        (fn (env, var, args) => not (null env),  fn - => matched subterm for
splitting has to be abstraction, use dummy pattern '-' to skip subterms ),
        (fn (env, var, args) => not (null args), fn (env, var, args) =>
          (Variable ' ^  Term.string-of-vname' (fst var) ^
           ' has to be applied to bound variable, about to be splitted)),
        (fn (env, var, args) => hd args = Bound (length env - 1), fn (env, var,
args) =>
          (first argument of variable ' ^  Term.string-of-vname' (fst var) ^
           'expected to be bound variable ' ^ (fst (hd env)) ^ '))
        ])
    |> map (#2)


  fun split-rule arities =
    let
      val (names, ctxt') = Variable.add-fixes (map (fst o fst) rule-vars) ctxt
    fun mk-inst ((var, T), (n, name)) i = ((var, Thm.cterm-of ctxt (mk-case-prod-fun
(name, range-type T) i n)), i+n)
      val insts = fold-map mk-inst (rule-vars ~~ (arities ~~ names)) 1 |> fst

      val [rule'] = Drule.infer-instantiate ctxt' insts rule
        |> Simplifier.simplify (put-simpset HOL-basic-ss ctxt' addsimps @{thms
case-prod-out})
      |> mk-meta-eq
      |> single
```

```
        |> Proof-Context.export ctxt' ctxt

      in
        rule'
      end
      val (split-rule, handler) = Fun-Cache.create-handler (Binding.map-name (fn
name => name ^ -cache) name)
       (fn tab => @{make-string} (Intlisttab.dest tab))
       Intlisttab.empty Intlisttab.lookup Intlisttab.update split-rule
     fun split-rule-names arity-names =
       let
         val arities = map fst arity-names
         val names = map snd arity-names
         val rule = split-rule arities
         fun sum ns = fold (fn n => fn m => n + m) ns 0
         val bound-names = map (mk-el-name) (1 upto (sum arities))
        val renamings = (bound-names ~~ flat (names)) |> filter-out (fn (-, n) =>
n = strip-uu)
       in
         Drule.rename-bvars renamings rule
       end

     fun trace-cache-info () =
         tracing (@{make-string} handler)

     (* populate

 cache, and as a side-effect check if splitting works *)
     val - = map split-rule (comb-product (map (K (1 upto 3)) rule-vars))
     val - = trace-cache-info ()
  in
    Simplifier.make-simproc ctxt {name = Binding.name-of name, kind = Simproc,
identifier = [],
      lhss = [pat],
      proc = fn - => fn ctxt => fn ct =>
        let
          val t = Thm.term-of ct
          val (tenv, term-env) = Pattern.match (Proof-Context.theory-of ctxt) (pat,
t) (Vartab.empty, Vartab.empty)
         val subterms = Vartab.dest term-env |> filter-out (is-dummy o fst) |> map
(snd o snd)
              val arity-names = map ((fn vars => (length vars, map fst vars)) o
strip-case-prod) subterms
          val - = asserts [(fn - => length arity-names = length rule-vars, fn - =>
split-rule-simproc: unexpected number of matched subterms)]

        in
          if exists (fn (n,-) => n > 1) arity-names
          then
```

```
          let
            val splitted-rule = split-rule-names arity-names
          in SOME splitted-rule end
        else NONE
      end
  }
 end

fun tuple-inst-simp-tac ctxt i =
  tuple-inst-tac ctxt i THEN
  simp-tac (put-simpset HOL-ss ctxt addsimps @{thms Product-Type.prod.case}) i

fun tuple-refl-tac ctxt i =
 resolve-tac ctxt @{thms refl} i
 ORELSE (
   tuple-inst-simp-tac ctxt i THEN
   resolve-tac ctxt @{thms refl} i
 )

fun split-rule-simprocs ctxt = map (fn (name, rule, pattern) => split-rule-simproc
ctxt name pattern rule)

fun gen-split-rule ctxt0 name-arities thm =
 let
   val ctxt = Variable.set-body false ctxt0
   val case-prod-eta-contract-thm = @{thm case-prod-eta-contract}
   val case-prod-conv = Conv.bottom-conv (
        K (Conv.try-conv (Conv.rewr-conv case-prod-eta-contract-thm)))
   val name-arities = name-arities |> sort-by fst
   val names = map fst name-arities
   val arities = map snd name-arities
   fun eta-contract ctxt thm = Conv.fconv-rule (case-prod-conv ctxt) thm
   val vars = Term.add-vars (Thm.prop-of thm) [] |>
    filter (fn ((n,-) ,-) => member (op =) names n) |> distinct (op =) |> sort-by
(fst o fst)
   val names' = map (fst o fst) vars
   val insts = ((names' ~~ arities) ~~ vars)
     |> map (fn ((f, n), (var,T)) =>
       (var, mk-tuple-or-case-prod f T n))
   val new-names = [] |> fold Term.add-frees (map snd insts) |> map fst
   val (new-names', ctxt') = Variable.add-fixes new-names ctxt
   val - = @{assert} (new-names = new-names')
   val (-, ctxt'') = ctxt' |> Variable.import-terms false (map snd insts)
     |> apsnd (Context-Position.set-visible false)
   val insts = map (apsnd (Thm.cterm-of ctxt'')) insts
   val split-arities = arities |> filter (fn n => n > 1)
   val splits = map (get-split-tupled-all-thm ctxt) split-arities
   val case-eqs = map (get-tuple-case-eq-thm ctxt) split-arities
 in
```

```
    thm
    |> Drule.infer-instantiate ctxt'' insts
    |> Simplifier.asm-full-simplify (put-simpset HOL-basic-ss ctxt'' addsimps
      (@{thms case-prod-out} @ case-eqs @  splits))
    |> eta-contract ctxt''
    |> single
    |> Proof-Context.export ctxt'' ctxt
    |> hd
    |> Drule.zero-var-indexes
  end


fun split-rule ctxt names thm n =
  if n <= 1 then thm
  else gen-split-rule ctxt (map (fn name => (name, n)) names) thm

fun split-rule-bin ctxt names thm bin =
  split-rule ctxt names thm (Bin.int-of-bin bin)


val case-prod-conv = mk-meta-eq @{thm case-prod-conv}

fun print-conv msg (ct:cterm) =
  let
    val - = tracing (msg ^ ": " ^ @{make-string} ct);
  in Conv.all-conv ct end



fun bounded-top-conv i conv ctxt ct =
  if i <= 0 then Conv.all-conv ct
  else
   (conv ctxt then-conv Conv.sub-conv (bounded-top-conv (i - 1) conv) ctxt) ct;

fun bounded-bottom-conv i conv ctxt ct =
  if i <= 0 then Conv.all-conv ct
  else
    (Conv.sub-conv (bounded-bottom-conv (i - 1) conv) ctxt then-conv conv ctxt)
ct;

fun bounded-top-rewrs-conv i rewrs = bounded-top-conv i (K (Conv.try-conv (Conv.rewrs-conv
rewrs)));

fun bounded-bottom-rewrs-conv i rewrs = bounded-bottom-conv i (K
  (Conv.try-conv (Conv.rewrs-conv rewrs)));


fun eta-expand-tupled-conv ctxt ct =
  let
```

```
    val arity-of = HOLogic.flatten-tupleT #> length
    val domT = Thm.typ-of-cterm ct |> domain-type
    val arity = domT |> arity-of
    val name-types = strip-case-prod' (Thm.term-of ct)
    val case-prod-arity = length name-types
   val renamings = map (fn (n, T) => if arity-of T = 1 then SOME n else NONE)
name-types
    val base-conv =
          if case-prod-arity > 0 andalso case-prod-arity < arity then
            let
              val eta-contract-rule = get-eta-tupled-eq-thm ctxt case-prod-arity
                |> Thm.symmetric
             in fn ct => Conv.rewr-conv eta-contract-rule ct handle CTERM - =>
Conv.all-conv ct end
          else Conv.all-conv
    val conv =
          if arity > 1 andalso case-prod-arity < arity then
            base-conv then-conv
             Conv.rewr-conv (Drule.rename-bvars' renamings (get-eta-tupled-eq-thm
ctxt arity)) then-conv
            bounded-bottom-rewrs-conv (arity * 2) [case-prod-conv] ctxt
        else Conv.all-conv
  in
    conv ct
  end
  handle Match => Conv.all-conv ct

fun eta-expand-tuple ctxt = eta-expand-tupled-conv ctxt #> Thm.rhs-of

end
›

simproc-setup ETA-TUPLED (‹ETA-TUPLED f›) = ‹fn - => fn ctxt => fn ct
=>
  let
    val T = Thm.typ-of-cterm ct |> domain-type
    val arity = T |> HOLogic.flatten-tupleT |> length
  in
    if arity <= 1 then
      SOME @{thm ETA-TUPLED-def}
    else
      SOME (@{thm ETA-TUPLED-trans} OF [Tuple-Tools.get-eta-tupled-eq-thm
ctxt arity])
  end›
declare [[simproc del: ETA-TUPLED]]
```

Set up the theorem cache. We could use "dynamic programming" here and rewrite "thm n" with one step of "thm (n - 1)" and *case-prod* = ($\lambda c\ p.\ c\ (fst\ p)\ (snd\ p)$)

setup ‹
Context.theory-map (*Tuple-Tools.Data.map* (*fn* - => *map* (*fn i* =>
(*Thm.trim-context* (*Tuple-Tools.mk-tuple-up-eq-thm* @{*context*} *i*),
Thm.trim-context (*Tuple-Tools.mk-split-tupled-all-thm* @{*context*} *i*),
Thm.trim-context (*Tuple-Tools.mk-tuple-case-eq-thm* @{*context*} *i*),
Thm.trim-context (*Tuple-Tools.mk-eta-tupled-eq-thm* @{*context*} *i*))) (*2 upto 50*)))
›

attribute-setup *split-tuple* = ‹
let
val parse-arity = *Args.*$$$ *arity* |-- *Args.colon* |-- *Parse.nat*
val parse-names = *Parse.and-list1 Parse.short-ident*
in
Scan.lift (*parse-names* -- *parse-arity*) >> (*fn* (*names*, *arity*) =>
let
in
Thm.rule-attribute [] (*fn context* => *fn thm* =>
Tuple-Tools.split-rule (*Context.proof-of context*) *names thm arity*)
end)
end
›

thm *HOL.ext* [*split-tuple f* **and** *g arity*: *3*]

ML-val ‹
val test = @{*pattern* ⋀*a b c s*. *f a b c* ≡ *?c* (*a*,*b*,*k*) *s*}
val x = *Tuple-Tools.calc-inst'* @{*context*} *2 test*
›

Especially in the case of congruence rules we prefer the more restrictive instantiation, where the freshly introduced variable does not depend on 'non-bound' tuple components

ML-val ‹
val test = @{*pattern* ⋀*a b c s*. *f a b c* ≡ *?c* (*a*,*b*,*k*) *s*}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

ML-val ‹
val thm = *Tuple-Tools.get-tuple-up-eq-thm* @{*context*} *1*
›

ML-val ‹
val thm = *Tuple-Tools.get-tuple-up-eq-thm* @{*context*} *0*
›

ML-val ‹
val thm = Tuple-Tools.get-tuple-up-eq-thm @{context} 3
›

ML-val ‹
val thm = Tuple-Tools.get-split-tupled-all-thm @{context} 30
›

ML-val ‹
val thm = Tuple-Tools.get-split-tupled-all-thm @{context} 2
›

ML-val ‹
val thm = Tuple-Tools.get-tuple-case-eq-thm @{context} 2
›

ML-val ‹
val thm = Tuple-Tools.get-eta-tupled-eq-thm @{context} 2
›

declare [[*simp-trace=false*]]

Variable names should be preserved here, since all abstractions stay in place.

lemma $\bigwedge r.\ (\lambda(x,y).\ (x::nat) < y \wedge y < 200)\ r$
apply (*tactic* ‹
simp-tac (put-simpset HOL-basic-ss @{context} addsimprocs [Tuple-Tools.split-tupled-all-simproc, Tuple-Tools.tuple-case-simproc]
delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}) 1
›)
oops

Variable names are not preserved due to eta contraction: $(\lambda(x,\ y).\ x < y) = (\lambda(x,\ y).\ x < y)$

lemma $\bigwedge r.\ (\lambda(x,y).\ (x::nat) < y\ )\ r$
apply (*tactic* ‹
simp-tac (put-simpset HOL-basic-ss @{context} addsimprocs [Tuple-Tools.split-tupled-all-simproc, Tuple-Tools.tuple-case-simproc]
delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}) 1
›)
oops

lemma $(\lambda(x,y,p,z,f).\ (x::nat) < y)\ (a,b,c) = (case\ c\ of\ (p,z,f) \Rightarrow a < b)$
apply (*tactic* ‹
simp-tac (put-simpset HOL-basic-ss @{context} addsimprocs [Tuple-Tools.split-tupled-all-simproc, Tuple-Tools.tuple-case-simproc]
delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}) 1
›)
done

lemma $\bigwedge r.\ (\lambda(x,y,p,z,f).\ (p::nat) < f)\ r$
apply (*tactic* ‹
simp-tac (*put-simpset HOL-basic-ss* @{*context*} *addsimprocs* [*Tuple-Tools.split-tupled-all-simproc*, *Tuple-Tools.tuple-case-simproc*]
delsimps @{*thms Product-Type.prod.case Product-Type.case-prod-conv*}) *1*
›)
oops

lemma $(\lambda(x,y,p).\ (x::nat) < y)\ (a,b,c,d,e) = (a < b)$
apply (*tactic* ‹
simp-tac (*put-simpset HOL-basic-ss* @{*context*} *addsimprocs* [*Tuple-Tools.split-tupled-all-simproc*, *Tuple-Tools.tuple-case-simproc*]
delsimps @{*thms Product-Type.prod.case Product-Type.case-prod-conv*}) *1*
›)
done

ML-val ‹
val test = @{*pattern* $\bigwedge a\ b\ c\ d\ e\ s.\ f\ a\ b\ c \equiv ?c\ a\ b\ (c,d,e)\ s$}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

ML-val ‹
val test = @{*pattern* $\bigwedge a\ b\ c\ s.\ P\ x \Longrightarrow f\ a\ b\ c \equiv ?c\ (a,b,c)\ s$}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

ML-val ‹
val test = @{*pattern* $\bigwedge a\ b\ c\ s.\ P\ x\ =simp=>\ f\ a\ b\ c \equiv ?c\ (a,b,c)\ s$}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

ML-val ‹
val test = @{*pattern* $\bigwedge a\ b\ c\ s.\ f\ a\ b\ c \equiv ?c\ (a,b,k)\ s$}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

ML-val ‹
val test = @{*pattern* $\bigwedge a\ b\ c\ s.\ f\ a\ b\ c \equiv ?c\ (a,b,k)\ s$}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

ML-val ‹
val test = @{*pattern* $\bigwedge a\ b\ c\ s.\ f\ a\ b\ c \equiv ?c\ (a,x+y,x + y + z)\ s$}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

ML-val ‹
val test = @{*pattern* ⋀*a b c s*. *f a b c* ≡ *?c* (*a*,*x*+*y*,*x* + *y* + *z*) *s*}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

ML-val ‹
val test = @{*pattern* (⋀*a b c s*. *P x* ⟹ *f a b c* ≡ *?c* (*a*,*b*,*c*) *s*) ⟹ (⋀*a b c s*. *Q x*)}
val x = *Tuple-Tools.calc-inst* @{*context*} *2 test*
›

lemma *SPLIT* (⋀*r*. ((λ*z*. *Q z* ∧ (*Q z* ⟶ *P z*)) *r*))
≡ *XXX*
apply (*tactic* ‹
simp-tac (@{*context*}
addsimprocs [*Tuple-Tools.SPLIT-simproc*, *Tuple-Tools.tuple-case-simproc*]
|> *Simplifier.add-cong* @{*thm SPLIT-cong*}
|> *Simplifier.add-cong* @{*thm conj-cong*}) *1*
›)
oops

lemma *PROP SPLIT* (⋀*r*. ((λ(*x*,*y*,*z*). *y* < *z* ∧ *z*=*s*) *r*) ⟹ *P r*)
≡ (⋀*x y z*. *y* < *z* ∧ *z* = *s* ⟹ *P* (*x*, *y*, *s*))
apply (*tactic* ‹
asm-full-simp-tac (*put-simpset HOL-basic-ss* @{*context*}
addsimprocs [*Tuple-Tools.SPLIT-simproc*, *Tuple-Tools.tuple-case-simproc*] |> *Simplifier.add-cong* @{*thm SPLIT-cong*}) *1*
›)
done

schematic-goal *ETA-TUPLED* (λ*x*::('*a* ×'*b* × '*c*). *f x*) = *?XXX*
supply [[*simp-trace*]]
thm *Product-Type.cond-case-prod-eta*
supply [[*simproc add*: *ETA-TUPLED*]]
apply (*simp*)
done

context
fixes *f*::('*a* × '*b* × '*c* × '*d*) ⇒ *nat*
fixes *c*::('*a* × '*b* × '*c* × '*d*) ⇒ '*s* ⇒ *bool*
begin

ML-val ‹
val t1 = *Tuple-Tools.eta-expand-tupled-conv* @{*context*} @{*cterm f*}
val t2 = *Tuple-Tools.eta-expand-tupled-conv* @{*context*} @{*cterm* λ(*a*, *b*). *f* (*a*, *b*)}
val t3 = *Tuple-Tools.eta-expand-tupled-conv* @{*context*} @{*cterm* λ(*a*, *b*, *c*). *f* (*a*,

```
b, c)}

val c1 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm c}
val c2 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm λ(a, b) s. c (a,b)
s}
val d1 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm λ(a, (b::nat ×
nat)) s. a < 2}
val d2 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm λ(a, (b::nat ×
nat × nat)) s. a < 2}
val d3 = Tuple-Tools.eta-expand-tupled-conv @{context} @{cterm λ(a, (b::nat ×
nat × nat × nat)) s. a < 2}
›

ML-val ‹
Tuple-Tools.beta-tupled @{context} 3 @{cterm ‹λ(a::nat,b,c). a + b + c›} @{cterm
(x::nat,y::nat,z::nat)}
›
end


end



theory Subgoal-Methods
imports Main
begin
ML ‹
signature SUBGOAL-METHODS =
sig
  val fold-subgoals: Proof.context -> bool -> thm -> thm
  val unfold-subgoals-tac: Proof.context -> tactic
  val distinct-subgoals: Proof.context -> thm -> thm
end;

structure Subgoal-Methods: SUBGOAL-METHODS =
struct

fun max-common-prefix eq (ls :: lss) =
      let
        val ls' = tag-list 0 ls;
        fun all-prefix (i,a) =
          forall (fn ls' => if length ls' > i then eq (a, nth ls' i) else false) lss
        val ls'' = take-prefix all-prefix ls'
      in map snd ls'' end
  | max-common-prefix - [] = [];

fun push-outer-params ctxt th =
```

```
  let
    val ctxt' = ctxt
      |> Simplifier.empty-simpset
      |> Simplifier.add-simp Drule.norm-hhf-eq;
  in
    Conv.fconv-rule
      (Raw-Simplifier.rewrite-cterm (true, false, false) (K (K NONE)) ctxt') th
  end;

fun fix-schematics ctxt raw-st =
  let
    val ((schematic-types, [st']), ctxt1) = Variable.importT [raw-st] ctxt;
    val ((-, inst), ctxt2) =
      Variable.import-inst true [Thm.prop-of st'] ctxt1;

    val schematic-terms = Vars.map (K (Thm.cterm-of ctxt2)) inst;
    val schematics = (schematic-types, schematic-terms);

  in (Thm.instantiate schematics st', ctxt2) end

val strip-params = Term.strip-all-vars;
val strip-prems = Logic.strip-imp-prems o Term.strip-all-body;
val strip-concl = Logic.strip-imp-concl o Term.strip-all-body;



fun fold-subgoals ctxt prefix raw-st =
  if Thm.nprems-of raw-st < 2 then raw-st
  else
    let
      val (st, inner-ctxt) = fix-schematics ctxt raw-st;

      val subgoals = Thm.prems-of st;
      val paramss = map strip-params subgoals;
      val common-params = max-common-prefix (eq-snd (op =)) paramss;

      fun strip-shift subgoal =
        let
          val params = strip-params subgoal;
          val diff = length common-params - length params;
          val prems = strip-prems subgoal;
        in map (Term.incr-boundvars diff) prems end;

      val premss = map (strip-shift) subgoals;

      val common-prems = max-common-prefix (op aconv) premss;

      val common-params = if prefix then common-params else [];
      val common-prems = if prefix then common-prems else [];
```

```
    fun mk-concl subgoal =
      let
        val params = Term.strip-all-vars subgoal;
        val local-params = drop (length common-params) params;
        val prems = strip-prems subgoal;
        val local-prems = drop (length common-prems) prems;
        val concl = strip-concl subgoal;
      in Logic.list-all (local-params, Logic.list-implies (local-prems, concl)) end;

    val goal =
      Logic.list-all (common-params,
       (Logic.list-implies (common-prems,Logic.mk-conjunction-list (map mk-concl
subgoals))));

    val chyp = Thm.cterm-of inner-ctxt goal;

    val (common-params',inner-ctxt') =
      Variable.add-fixes (map fst common-params) inner-ctxt
      |>> map2 (fn (-, T) => fn x => Thm.cterm-of inner-ctxt (Free (x, T)))
common-params;

    fun try-dest rule =
       try (fn () => (@{thm conjunctionD1} OF [rule], @{thm conjunctionD2}
OF [rule])) ();

    fun solve-headgoal rule =
      let
        val rule' = rule
          |> Drule.forall-intr-list common-params'
          |> push-outer-params inner-ctxt';
      in
        (fn st => Thm.implies-elim st rule')
      end;

    fun solve-subgoals rule' st =
      (case try-dest rule' of
        SOME (this, rest) => solve-subgoals rest (solve-headgoal this st)
      | NONE => solve-headgoal rule' st);

    val rule = Drule.forall-elim-list common-params' (Thm.assume chyp);
  in
    st
    |> push-outer-params inner-ctxt
    |> solve-subgoals rule
    |> Thm.implies-intr chyp
    |> singleton (Variable.export inner-ctxt' ctxt)
  end;
```

```
fun distinct-subgoals ctxt raw-st =
  let
    val (st, inner-ctxt) = fix-schematics ctxt raw-st;
    val subgoals = Thm.cprems-of st;
    val atomize = Conv.fconv-rule (Object-Logic.atomize-prems inner-ctxt);

    val rules =
      map (atomize o Raw-Simplifier.norm-hhf inner-ctxt o Thm.assume) subgoals
      |> sort (int-ord o apply2 Thm.nprems-of);

    val st' = st
      |> ALLGOALS (fn i =>
        Object-Logic.atomize-prems-tac inner-ctxt i THEN solve-tac inner-ctxt rules
i)
      |> Seq.hd;

    val subgoals' = subgoals
      |> inter (op aconvc) (Thm.chyps-of st')
      |> distinct (op aconvc);
  in
    Drule.implies-intr-list subgoals' st'
    |> singleton (Variable.export inner-ctxt ctxt)
  end;

(* Variant of filter-prems-tac that recovers premise order *)
fun filter-prems-tac' ctxt pred =
  let
    fun Then NONE tac = SOME tac
      | Then (SOME tac) tac' = SOME (tac THEN' tac');
    fun thins H (tac, n, i) =
      (if pred H then (tac, n + 1, i)
       else (Then tac (rotate-tac n THEN' eresolve-tac ctxt [thin-rl]), 0, i + n));
  in
    SUBGOAL (fn (goal, i) =>
      let val Hs = Logic.strip-assums-hyp goal in
        (case fold thins Hs (NONE, 0, 0) of
          (NONE, -, -) => no-tac
        | (SOME tac, -, n) => tac i THEN rotate-tac (~ n) i)
      end)
  end;

fun trim-prems-tac ctxt rules =
let
  fun matches (prem,rule) =
  let
    val ((-,prem'),ctxt') = Variable.focus NONE prem ctxt;
    val rule-prop = Thm.prop-of rule;
  in is-some (Unify.matcher (Context.Proof ctxt') [rule-prop] [prem']) end;
```

```
in filter-prems-tac' ctxt (not o member matches rules) end;

val adhoc-conjunction-tac = REPEAT-ALL-NEW
  (SUBGOAL (fn (goal, i) =>
    if can Logic.dest-conjunction (Logic.strip-imp-concl goal)
    then resolve0-tac [Conjunction.conjunctionI] i
    else no-tac));

fun unfold-subgoals-tac ctxt =
  TRY (adhoc-conjunction-tac 1)
  THEN (PRIMITIVE (Raw-Simplifier.norm-hhf ctxt));

val - =
  Theory.setup
   (Method.setup @{binding fold-subgoals}
      (Scan.lift (Args.mode prefix) >> (fn prefix => fn ctxt =>
        SIMPLE-METHOD (PRIMITIVE (fold-subgoals ctxt prefix))))
      lift all subgoals over common premises/params #>
    Method.setup @{binding unfold-subgoals}
      (Scan.succeed (fn ctxt => SIMPLE-METHOD (unfold-subgoals-tac ctxt)))
      recover subgoals after folding #>
    Method.setup @{binding distinct-subgoals}
     (Scan.succeed (fn ctxt => SIMPLE-METHOD (PRIMITIVE (distinct-subgoals
ctxt))))
     trim all subgoals to be (logically) distinct #>
    Method.setup @{binding trim}
      (Attrib.thms >> (fn thms => fn ctxt =>
        SIMPLE-METHOD (HEADGOAL (trim-prems-tac ctxt thms))))
     trim all premises that match the given rules);

end;
›

end
```

4.2 Tools for intro-rule based term synthesis *synthesize-rules*

```
theory Synthesize
  imports
  Tuple-Tools
  AutoCorres-Utils
  MkTermAntiquote
keywords
  synthesize-rules::thy-decl and
  add-synthesize-pattern::thy-decl % ML and
  print-synthesize-rules::diag
begin
```

ML-file ‹*synthesize-rules.ML*›

4.2.1 Commands

ML ‹
val - = Outer-Syntax.local-theory ***command-keyword*** ‹*synthesize-rules*›
declare named collection of synthesize rules (pattern indexed intro rules)
(*Parse.and-list1 Parse.binding >>*
(*fold (fn b => (snd o Synthesize-Rules.declare b))*))
›

ML ‹
val - =
Outer-Syntax.local-theory ***command-keyword*** ‹*add-synthesize-pattern*›
add pattern schemes via ML or declarations
(*Parse.and-list1 Parse.binding :|-- (fn rules-names =>*
(***keyword*** ‹*where*› *|-- Synthesize-Rules.pattern-decls >> (fn decls =>*
fold (fn rules-name => Synthesize-Rules.add-pattern-decls rules-name decls) rules-names))
||
(***keyword*** ‹=› *|-- Parse.ML-source >> (fn src =>*
fold (fn rules-name => Synthesize-Rules.add-pattern-ml rules-name src) rules-names))))
›

ML ‹
val - = Outer-Syntax.local-theory ***command-keyword*** ‹*print-synthesize-rules*›
print named collection of synthesize rules (optionally matching a given term)
(*Parse.name-position -- Scan.option Parse.term >>*
(fn (name, term-opt) => Synthesize-Rules.print-rules-cmd name term-opt))
›

4.2.2 ML Antiquotations

ML ‹ *Theory.setup*
(*ML-Antiquotation.inline-embedded* ***binding*** ‹*synthesize-rules-name*›
(*Args.context -- Scan.lift Parse.embedded-position >>*
(*fn (ctxt, name) => ML-Syntax.print-string (Synthesize-Rules.check (Context.Proof ctxt) name |> fst)*)))
›

ML ‹ *Theory.setup*
(*ML-Antiquotation.value-embedded* ***binding*** ‹*synthesize-rules*›
(*Args.context -- Scan.lift Parse.embedded-position >>*
(fn (ctxt, name) => Synthesize-Rules.get-rules ML-context ^
ML-Syntax.print-string (Synthesize-Rules.check (Context.Proof ctxt) name |> fst))))

›

ML ‹
Theory.setup
(*ML-Antiquotation.inline* @{*binding mk-synthesize-pattern*}
((*Args.context* −− *Scan.lift Parse.embedded-inner-syntax* −−
(*Scan.optional* (*Scan.lift* ((*Synthesize-Rules.comma-list Args.name*))) []))
>> (*fn* ((*ctxt*, *pattern-str*), *synth-args*) => *Synthesize-Rules.gen-pattern-fun*
ctxt (*pattern-str*, *synth-args*))))
›

4.2.3 Attributes

attribute-setup *synthesize-rule* = ‹
let
val priority = *Args.*$$$ *priority* |−− *Args.colon* |−− *Parse.int*
val opt-priority = *Scan.optional* (*Scan.lift* (*priority*)) *10*
val only-schematic-goal = *Args.*$$$ *only-schematic-goal* >> (*fn* - => *true*)
val opt-only-schematic-goal = *Scan.optional* (*Scan.lift only-schematic-goal*) *false*
val del = *Args.del* >> (*fn* - => *true*)
val opt-del = *Scan.optional* (*Scan.lift del*) *false*
val split = *Args.*$$$ *split* |−− *Args.colon* |−− *Parse.and-list Parse.short-ident*
val opt-split = *Scan.optional* (*Scan.lift* (*split*)) []
in
Scan.lift (*Parse.and-list1 Parse.name-position*)
−− *opt-priority* −− *opt-split* −− *opt-only-schematic-goal* −− *opt-del* >> (*fn*
((((*rules-names*, *prio*), *splits*), *only-schematic-goal*), *del*) =>
let
in
Thm.declaration-attribute (*fn thm* => *fn context* =>
let
val ctxt = *Context.proof-of context*
val thm-binding1 = *Utils.guess-binding-of-thm ctxt thm*
val thm-binding2 = *if Binding.is-empty thm-binding1 then Binding.make*
(*??*, (*snd* (*hd rules-names*))) *else thm-binding1*
val rules-names = *map* (*Synthesize-Rules.check context* #> *fst*) *rules-names*
fun add-rule rules-name =
if null splits
then Synthesize-Rules.add-rule rules-name {*only-schematic-goal* = *only-schematic-goal*}
thm-binding2 prio thm
else Synthesize-Rules.add-split-rule rules-name {*only-schematic-goal* =
only-schematic-goal} *thm-binding2 prio splits thm* #> *snd*
fun del-rule rules-name =
if null splits
then Synthesize-Rules.del-rule rules-name {*only-schematic-goal* = *only-schematic-goal*}
thm-binding2 prio thm
else Synthesize-Rules.del-split-rule rules-name {*only-schematic-goal* =
only-schematic-goal} *thm-binding2 prio splits thm*
in

```
        context |> fold (if del then del-rule else add-rule) rules-names
      end)
  end)
end
›
```

end

Chapter 5

Rule by Method

theory *Rule-By-Method*
imports
Main
HOL−Eisbach.Eisbach-Tools
begin

ML ‹
signature RULE-BY-METHOD =
sig
val rule-by-tac: *Proof.context −> {vars* : *bool*, *prop*: *bool*} −>
(*Proof.context −> tactic*) −> (*Proof.context −> tactic*) *list −> Position.T*
−> thm
end;

fun atomize ctxt = Conv.fconv-rule (*Object-Logic.atomize ctxt*);

fun fix-schematics ctxt raw-st =
let
val ((*schematic-types*, [*st′*]), *ctxt1*) = *Variable.importT* [*raw-st*] *ctxt*;
fun certify-inst ((-, *terms*), *ctxt*) = (*Vars.map* (*K* (*Thm.cterm-of ctxt*)) *terms*,
ctxt);
val (*schematic-terms*, *ctxt2*) =
Variable.import-inst true [*Thm.prop-of st′*] *ctxt1*
|> *certify-inst*;
val schematics = (*schematic-types*, *schematic-terms*);
in (*Thm.instantiate schematics st′*, *ctxt2*) *end*

fun curry-asm ctxt st = if Thm.nprems-of st = 0 then Seq.empty else
let

val prems = Thm.cprem-of st 1 |> *Thm.term-of* |> *Logic.strip-imp-prems*;

val (*thesis* :: *xs*,*ctxt′*) = *Variable.variant-fixes* (*thesis* :: *replicate* (*length prems*)

```
P) ctxt;

  val rl =
    xs
    |> map (fn x => Thm.cterm-of ctxt' (Free (x, propT)))
    |> Conjunction.mk-conjunction-balanced
     |> (fn xs => Thm.apply (Thm.apply @{cterm Pure.imp} xs) (Thm.cterm-of
ctxt' (Free (thesis,propT))))
    |> Thm.assume
    |> Conjunction.curry-balanced (length prems)
    |> Drule.implies-intr-hyps

  val rl' = singleton (Variable.export ctxt' ctxt) rl;

  in Thm.bicompose (SOME ctxt) {flatten = false, match = false, incremented =
false}
              (false, rl', 1) 1 st end;

val drop-trivial-imp =
let
  val asm =
    Thm.assume (Drule.protect @{cprop (PROP A ⟹ PROP A) ⟹ PROP A})
    |> Goal.conclude;

in
  Thm.implies-elim  asm (Thm.trivial @{cprop PROP A})
  |> Drule.implies-intr-hyps
  |> Thm.generalize (Names.empty, Names.make-set [A]) 1
  |> Drule.zero-var-indexes
 end

val drop-trivial-imp' =
let
  val asm =
    Thm.assume (Drule.protect @{cprop (PROP P ⟹ A) ⟹ A})
    |> Goal.conclude;

  val asm' = Thm.assume @{cprop PROP P == Trueprop A}

in
  Thm.implies-elim asm (asm' COMP Drule.equal-elim-rule1)
  |> Thm.implies-elim (asm' COMP Drule.equal-elim-rule2)
  |> Drule.implies-intr-hyps
  |> Thm.permute-prems 0 ~1
  |> Thm.generalize (Names.empty, Names.make-set [A,P]) 1
  |> Drule.zero-var-indexes
 end

fun atomize-equiv-tac ctxt i =
```

```
  Object-Logic.full-atomize-tac ctxt i
  THEN PRIMITIVE (fn st' =>
  let val (-,[A,-]) = Drule.strip-comb (Thm.cprem-of st' i) in
  if Object-Logic.is-judgment ctxt (Thm.term-of A) then st'
 else error (Failed to fully atomize result:\n ^(Syntax.string-of-term ctxt (Thm.term-of
A))) end)


structure Data = Proof-Data
(
  type T = thm list * bool;
  fun init - = ([],false);
);

val empty-rule-prems = Data.map (K ([],true));

fun add-rule-prem thm = Data.map (apfst (Thm.add-thm thm));

fun with-rule-prems enabled parse =
  Scan.state :|-- (fn context =>
  let
   val context' = Context.proof-of context |> Data.map (K ([Drule.free-dummy-thm],enabled))
                  |> Context.Proof
  in Scan.lift (Scan.pass context' parse) end)


fun get-rule-prems ctxt =
  let
    val (thms,b) = Data.get ctxt
  in if (not b) then [] else thms end


fun zip-subgoal assume tac (ctxt,st : thm) = if Thm.nprems-of st = 0 then Seq.single
(ctxt,st) else
let
  fun bind-prems st' =
  let
    val prems = Thm.cprems-of st';
    val (asms, ctxt') = Assumption.add-assumes prems ctxt;
    val ctxt'' = fold add-rule-prem asms ctxt';
    val st'' = Goal.conclude (Drule.implies-elim-list st' (map Thm.assume prems));
  in (ctxt'',st'') end

  fun defer-prems st' =
  let
    val nprems = Thm.nprems-of st';
    val st'' = Thm.permute-prems 0 nprems (Goal.conclude st');
  in (ctxt,st'') end;
```

```
in
  tac ctxt (Goal.protect 1 st)
  |> Seq.map (if assume then bind-prems else defer-prems)  end


fun zip-subgoals assume tacs pos ctxt st =
let
  val nprems = Thm.nprems-of st;
  val - = nprems < length tacs andalso error (More tactics than rule assumptions
^ Position.here pos);
  val tacs' = map (zip-subgoal assume) (tacs @ (replicate (nprems - length tacs)
(K all-tac)));
  val ctxt' = empty-rule-prems ctxt;
in Seq.EVERY tacs' (ctxt',st) end;

fun rule-by-tac' ctxt {vars,prop} tac asm-tacs pos raw-st =
  let
    val (st,ctxt1) = if vars then (raw-st,ctxt) else fix-schematics ctxt raw-st;

   val ([x],ctxt2) = Proof-Context.add-fixes [(Binding.name Auto-Bind.thesisN,NONE,
NoSyn)] ctxt1;

    val thesis = if prop then Free (x,propT) else Object-Logic.fixed-judgment ctxt2
x;

    val cthesis = Thm.cterm-of ctxt thesis;

    val revcut-rl' = Thm.instantiate' [] ([NONE,SOME cthesis]) @{thm revcut-rl};

    fun is-thesis t = Logic.strip-assums-concl t aconv thesis;

    fun err thm str = error (str ^ Position.here pos ^ \n ^
      (Pretty.string-of (Goal-Display.pretty-goal ctxt thm)));

    fun pop-thesis st =
    let
      val prems = Thm.prems-of st |> tag-list 0;
      val (i,-) = (case filter (is-thesis o snd) prems of
        [] => err st Lost thesis
      | [x] => x
      | - => err st More than one result obtained);
     in st |> Thm.permute-prems 0 i  end

    val asm-st =
    (revcut-rl' OF [st])
    |> (fn st => Goal.protect (Thm.nprems-of st - 1) st)
```

```
    val (ctxt3,concl-st) = case Seq.pull (zip-subgoals (not vars) asm-tacs pos ctxt2
asm-st) of
      SOME (x,-) => x
    | NONE => error (Failed to apply tactics to rule assumptions. ^(Position.here
pos));

    val concl-st-prepped =
      concl-st
      |> Goal.conclude
       |> (fn st => Goal.protect (Thm.nprems-of st) st |> Thm.permute-prems 0
~1 |> Goal.protect 1)

    val concl-st-result = concl-st-prepped
      |> (tac ctxt3
          THEN (PRIMITIVE pop-thesis)
          THEN curry-asm ctxt
            THEN PRIMITIVE (Goal.conclude #> Thm.permute-prems 0 1 #>
Goal.conclude))

    val result = (case Seq.pull concl-st-result of
      SOME (result,-) => singleton (Proof-Context.export ctxt3 ctxt) result
      | NONE => err concl-st-prepped Failed to apply tactic to rule conclusion:)

    val drop-rule = if prop then drop-trivial-imp else drop-trivial-imp'

    val result' = ((Goal.protect (Thm.nprems-of result -1) result) RS drop-rule)
    |> (if prop then all-tac else
       (atomize-equiv-tac ctxt (Thm.nprems-of result)
       THEN resolve-tac ctxt @{thms Pure.reflexive} (Thm.nprems-of result)))
    |> Seq.hd
    |> Raw-Simplifier.norm-hhf ctxt

  in Drule.zero-var-indexes result' end;

fun rule-by-tac is-closed ctxt args tac asm-tacs pos raw-st =
 let val f = rule-by-tac' ctxt args tac asm-tacs pos
  in
   if is-closed orelse Context-Position.is-really-visible ctxt then SOME (f raw-st)
   else try f raw-st
 end

fun pos-closure (scan : 'a context-parser) :
  (('a * (Position.T * bool)) context-parser) = (fn (context,toks) =>
  let
    val (((context',x),tr-toks),toks') = Scan.trace (Scan.pass context (Scan.state --
scan)) toks;
    val pos = Token.range-of tr-toks;
    val is-closed = exists (fn t => is-some (Token.get-value t)) tr-toks
  in ((x,(Position.range-position pos, is-closed)),(context',toks')) end)
```

```
val parse-flags = Args.mode schematic -- Args.mode raw-prop >> (fn (b,b') =>
{vars = b, prop = b'})

fun tac m ctxt =
  NO-CONTEXT-TACTIC ctxt
    (Method.evaluate-runtime m ctxt []);

(* Declare as a mixed attribute to avoid any partial evaluation *)

fun handle-dummy f (context, thm) =
  case (f context thm) of SOME thm' => (NONE, SOME thm')
  | NONE => (SOME context, SOME Drule.free-dummy-thm)

val (rule-prems-by-method : attribute context-parser) = Scan.lift parse-flags :--
(fn flags =>
  pos-closure (Scan.repeat1
    (with-rule-prems (not (#vars flags)) Method.text-closure ||
      Scan.lift (Args.$$$ - >> (K Method.succeed-text)))) >>
        (fn (flags,(ms,(pos, is-closed))) => handle-dummy (fn context =>
          rule-by-tac is-closed (Context.proof-of context) flags (K all-tac) (map tac
ms) pos))

val (rule-concl-by-method : attribute context-parser) = Scan.lift parse-flags :-- (fn
flags =>
  pos-closure (with-rule-prems (not (#vars flags)) Method.text-closure)) >>
    (fn (flags,(m,(pos, is-closed))) => handle-dummy (fn context =>
      rule-by-tac is-closed (Context.proof-of context) flags (tac m) [] pos))

val - = Theory.setup
  (Global-Theory.add-thms-dynamic (@{binding rule-prems},
    (fn context => get-rule-prems (Context.proof-of context))) #>
   Attrib.setup @{binding #} rule-prems-by-method
    transform rule premises with method #>
   Attrib.setup @{binding @} rule-concl-by-method
    transform rule conclusion with method #>
   Attrib.setup @{binding atomized}
    (Scan.succeed (Thm.rule-attribute []
      (fn context => fn thm =>
        Conv.fconv-rule (Object-Logic.atomize (Context.proof-of context)) thm
          |> Drule.zero-var-indexes)))
    atomize rule)
›
```

experiment begin

ML ‹

```
  val [att] = @{attributes [‹erule thin-rl, cut-tac TrueI, fail›]}
  val k = Attrib.attribute @{context} att
```

```
  val - = case (try k (Context.Proof @{context}, Drule.dummy-thm)) of
    SOME - => error Should fail
    | - => ()
  ›

lemmas baz = [[@‹erule thin-rl, rule revcut-rl[of P ⟶ P ∧ P], simp›]] for P

lemmas bazz[THEN impE] = TrueI[@‹erule thin-rl, rule revcut-rl[of P ⟶ P ∧
P], simp›] for P

lemma Q ⟶ Q ∧ Q by (rule baz)

method silly-rule for P :: bool uses rule =
  (rule [[@‹erule thin-rl, cut-tac rule, drule asm-rl[of P]›]])

lemma assumes A shows A by (silly-rule A rule: ‹A›)

lemma assumes A[simp]: A shows A
  apply (match conclusion in P for P ⇒
      ‹rule [[@‹erule thin-rl, rule revcut-rl[of P], simp›]]›)
  done

end

end


theory Option-Scanner
imports ML-Record-Antiquotation
begin
```

The purpose of the option scanner is to provide an interface to scan lists of options (key / value) for toplevel commands. As values may have different types the idea is that the collection of options is represented by an record of optional values. This record can be provided individually for each command. The initial option-record "empty" has every component assigned to `NONE`. To be able to specify all options in a single list, we represent a single parser / scanner for a field as an update function for the option-record, which is composed by the map-functions for the record fields. So we can preserve different types for parsing individual fields (and do not need an universal value type), while the specification list is a monotype.

```
ML ‹

signature OPTION-SCANNER =
  sig
    type 'opt parse = ('opt -> 'opt) parser
    type 'opt field =  string * 'opt parse * 'opt parse option (* name, parser, default
value parser *)
```

```
    type ('opt, 'a) map-field = ('a option -> 'a option) -> ('opt -> 'opt)

    val mk-opt: string -> 'a parser -> (('opt, 'a) map-field) -> 'a option ->
'opt field

    val bool-opt: ('opt, bool) map-field -> bool option -> 'opt field
    val int-opt: ('opt, int) map-field -> int option -> 'opt field
    val real-opt: ('opt, real) map-field -> real option -> 'opt field
    val string-opt: ('opt, string) map-field -> string option -> 'opt field
    val path-opt: ('opt, string * Position.T) map-field -> (string * Position.T)
option -> 'opt field
    val string-list-opt: ('opt, string list) map-field -> string list option -> 'opt
field
    val path-list-opt: ('opt, (string * Position.T) list) map-field -> (string * Posi-
tion.T) list option -> 'opt field

    val scan-bool: bool parser
    val scan-list: 'a parser -> 'b parser -> 'c list parser -> 'c list parser
    val scan-string-list: string list parser

    val get-options: 'opt -> (string * ('opt field)) list -> 'opt parser

  end

structure Option-Scanner:OPTION-SCANNER =
struct

  type 'opt parse = ('opt -> 'opt) parser
  type 'opt field =  string * 'opt parse * 'opt parse option
  type ('opt, 'a) map-field = ('a option -> 'a option) -> ('opt -> 'opt)

  val equals = keyword "="

  val scan-bool = Parse.group (fn - => bool (true / false))
    (Parse.reserved false >> K false || Parse.reserved true >> K true)

  val scan-path = Scan.ahead Parse.not-eof -- Parse.path >> (fn (tok, name)
=> (name, Token.pos-of tok))

  fun scan-list scan-open scan-close scan-entries =
    (scan-open -- scan-close >> K []) ||
    (scan-open |-- scan-entries --| scan-close)

  val scan-string-list = scan-list keyword "[" keyword "]" (Parse.enum , Parse.embedded)
  val scan-path-list = scan-list keyword "[" keyword "]" (Parse.enum , scan-path)

  fun scan-field scan map-field = scan >> (map-field o K o SOME)
  fun scan-default default map-field = Option.map (fn v => Scan.succeed (map-field
(K (SOME v)))) default
```

```
  fun mk-opt (kind:string) scan map-field default = (kind, scan-field scan map-field,
scan-default default map-field)


  fun bool-opt map-field default        = mk-opt bool scan-bool map-field default
  fun int-opt map-field default         = mk-opt int Parse.int map-field default
  fun real-opt map-field default        = mk-opt real Parse.real map-field default
  fun string-opt map-field default      = mk-opt string Parse.embedded map-field
default
  fun path-opt map-field default        = mk-opt path scan-path map-field default
  fun string-list-opt map-field default = mk-opt string list scan-string-list map-field
default
  fun path-list-opt map-field default   = mk-opt path list scan-path-list map-field
default

  fun scan-config-value eq scan-value maybe-default =
    (eq |-- Parse.!!! scan-value) ||
    (case maybe-default of SOME d => d | - => Scan.fail)

  fun scan-option (name, (-, scan-value, maybe-default)) =
   Parse.reserved name -- Parse.!!! (scan-config-value equals scan-value maybe-default)


  fun scan-options opts =
    let
      fun string-of-opt (name, (kind, parser, maybe-default)) = name ^ ": " ^ kind;
      val option-description = map (Pretty.str o string-of-opt) opts |> Pretty.list (
) |> Pretty.string-of;
      val msg = fn () => options ^ option-description
      val one-of = Parse.group msg (Scan.first (map scan-option opts));
    in
       Scan.optional (scan-list keyword‹[› keyword‹]› (Parse.enum , (Parse.!!!
one-of))) []
    end

  fun no-duplicates xs =
    case duplicates (op =) xs of
     [] => ()
    | ds => error (duplicate option(s) defined: " ^ Pretty.string-of (Pretty.list [ ]
(map Pretty.str ds)))

  fun applies fs init =
    let
      fun apply f x = f x
    in fold apply fs init end

  fun eval-options init opts =
```

```
    let
      val - = no-duplicates (map fst opts)
    in
      applies (map snd opts) init
    end

  fun get-options init opts =
   scan-options opts >> eval-options init


end


›
```

ML-val ‹

— Setup step 1: define the options record. By using the record antiquotation the map-functions for the fields will be automatically generated.

```
@{record
  ‹datatype opts = Opts of {i:int option, b:bool option, str:string list option}›
}
```

— Setup step 2: Define the "empty" record.

```
val empty-opts = make-opts {i=NONE, b=NONE, str = NONE}
```

— Setup step 3: Define the options you want to support.

```
val opts = [(i-opt, Option-Scanner.int-opt map-i NONE),
            (b-opt, Option-Scanner.bool-opt map-b (SOME true)),
            (flags, Option-Scanner.string-list-opt map-str NONE)]
```

— Aux functions to showcase the parsing

```
fun filtered-input b =
  filter Token.is-proper (Token.explode (Thy-Header.get-keywords' @{context}) (Binding.pos-of
b) (Binding.name-of b))

fun do-parse parser = Scan.error parser o filtered-input
fun do-parse-error parser input =
  do-parse parser input
  handle ERROR str => (warning str; (empty-opts, []))
```

— Examples for successful parsing.

```
val (opts1, -) = do-parse (Option-Scanner.get-options empty-opts opts)
  @{binding ‹[i-opt=22, b-opt=true]›}
```

```
val (opts2, -) = do-parse (Option-Scanner.get-options empty-opts opts)
  @{binding ‹[i-opt=22, b-opt]›}

val (opts3, -) = do-parse (Option-Scanner.get-options empty-opts opts)
  @{binding ‹[flags=[hallo, echo, otto]]›}

— Examples for error reporting.

val - = do-parse-error (Option-Scanner.get-options empty-opts opts)
  @{binding ‹[b-opt=ddfsf]›}

val - = do-parse-error (Option-Scanner.get-options empty-opts opts)
  @{binding ‹[foo=42]›}

val - = do-parse-error (Option-Scanner.get-options empty-opts opts)
  @{binding ‹[i-opt=22, i-opt=23]›}

›

end


theory Named-Rules
  imports Main
  keywords named-rules::thy-decl
begin
ML-file named-rules.ML
end


theory Subgoals
  imports Main
  keywords prefers :: prf-script % proof and subgoals :: prf-script-goal % proof
begin

definition protected-conjunction :: prop ⇒ prop ⇒ prop (infixr ‹&^&› 2) where
  protected-conjunction A B ≡ (PROP A &&& PROP B)

definition
  protected-prop A ≡ PROP A

lemma protect-prop:
  PROP A if PROP protected-prop A
  using that unfolding protected-prop-def .

ML ‹
fun filter-subgoals (test : cterm -> bool) thm =
  let
```

```
    val indexed-subgoals = tag-list 0 (Thm.cprems-of thm);
  in
    map fst (Library.filter (fn (-, subgoal) => test subgoal) indexed-subgoals)
  end

fun match-cterm pattern cterm =
  (Thm.match (pattern, cterm); true) handle Pattern.MATCH => false

datatype match-kind = Match-Concl | Match-Prems

fun match-cterm-rec pattern cterm =
  if match-cterm pattern cterm then
    true
  else
    match-cterm-rec pattern (Thm.dest-fun cterm)
  handle
    Thm.CTERM - => false

fun dest-all-cterm-all ctxt ct =
  let
    val ((-, ct), ctxt) = Variable.dest-all-cterm ct ctxt
  in
    dest-all-cterm-all ctxt ct
  end
  handle CTERM - => ct

fun match-subgoal (kind : match-kind) (no-match : bool) ctxt (pattern : cterm)
(subgoal : cterm) =
  let
    val cterms = case kind of
      Match-Concl => [dest-all-cterm-all ctxt subgoal |> Drule.strip-imp-concl] |
      Match-Prems => (dest-all-cterm-all ctxt subgoal |> Drule.strip-imp-prems);
    val match = fold (fn cterm => fn b => b orelse match-cterm-rec pattern
      (Thm.dest-arg cterm)) cterms false
  in
    if no-match then not match else match
  end

fun prefer-and-uncurry-subgoals-tac pred ctxt : tactic = fn thm =>
  let
    val indices = filter-subgoals pred thm
  in
    if indices = [] then
      Seq.empty
    else if length indices = 1 then
      CONVERSION (
       Conv.top-conv (K (Conv.rewr-conv @{thm protected-prop-def[symmetric]}))
ctxt)
        1 (Drule.rearrange-prems indices thm)
```

```
    else
      Seq.single (Conjunction.uncurry-balanced
        (length indices)
        (Drule.rearrange-prems indices thm))
  end

fun prefer-and-protect-subgoals-tac pred ctxt =
  prefer-and-uncurry-subgoals-tac pred ctxt
   THEN (REPEAT-DETERM (CHANGED-PROP (CONVERSION ((Conv.top-sweep-conv
(K (Conv.rewr-conv
        @{thm protected-conjunction-def[symmetric]})) ctxt)) 1)))

fun prefer-and-protect-subgoals-tac-pat (kind : match-kind) (no-match : bool) (pattern
: cterm) =
  fn ctxt => prefer-and-protect-subgoals-tac (match-subgoal kind no-match ctxt pat-
tern) ctxt

fun unprotect-subgoals-tac thms ctxt : tactic =
  REPEAT-DETERM (CHANGED-PROP (CONVERSION
    (Conv.top-sweep-conv (K (Conv.rewrs-conv thms)) ctxt) 1))

fun construct-pattern ctxt pattern =
  let
    val pattern = Proof-Context.read-term-pattern ctxt pattern;
    val pattern =
      if Term.fastype-of pattern = @{typ prop} then
        HOLogic.dest-Trueprop pattern
      else
        pattern
  in
    Thm.cterm-of ctxt pattern end

val parse-match-kind = Scan.optional (
  Args.parens (
    Args.$$$ concl     >> K (Match-Concl, false) ||
    Args.$$$ not-concl >> K (Match-Concl, true) ||
    Args.$$$ prems     >> K (Match-Prems, false) ||
    Args.$$$ not-prems >> K (Match-Prems, true))) (Match-Concl, false);

fun unprotect-and-finish thms =
  Seq.make-results o
  Seq.single o
  Proof.refine-singleton
    (Method.Basic (fn - => Method.succeed)) o
  Proof.refine-singleton
    (Method.Basic (fn ctxt => SIMPLE-METHOD
      (unprotect-subgoals-tac thms ctxt))
    )
```

```
val - = Outer-Syntax.command command-keyword ‹prefers›
  select subgoals that match a given pattern
  (parse-match-kind -- Parse.embedded-inner-syntax >> (fn ((kind, invert), pat-
tern) =>
      Toplevel.proofs (fn state => (
        let
          val ctxt = Proof.context-of state;
          val pattern = construct-pattern ctxt pattern
        in
          unprotect-and-finish @{thms protected-conjunction-def protected-prop-def}
o
          Proof.refine-singleton
            (Method.Basic (fn ctxt => SIMPLE-METHOD
                (prefer-and-protect-subgoals-tac-pat kind invert pattern ctxt))) end)
state)));

val - =
  Outer-Syntax.command command-keyword ‹subgoals›
    focus on all subgoals that match a given pattern within backward refinement
     (parse-match-kind -- Parse.embedded-inner-syntax >> (fn ((kind, invert),
pattern) =>
      Toplevel.proofs (fn state => (
        let
          val ctxt = Proof.context-of state;
          val pattern = construct-pattern ctxt pattern
        in
          unprotect-and-finish @{thms protected-conjunction-def protected-prop-def}
o
          #2 o
          Subgoal.subgoal Binding.empty-atts NONE (false, []) o
          Proof.refine-singleton
            (Method.Basic (fn ctxt => SIMPLE-METHOD
                (prefer-and-protect-subgoals-tac-pat kind invert pattern ctxt))) end)
state)))
›
```

Usage examples **lemma**
  $x > 2 \Longrightarrow x > 0 \wedge x > 1$ **for** $x :: nat$
  apply *standard*
  subgoals ‹- > -›
    by *simp+*
  done

lemma
  $x = 2 \Longrightarrow x > 0 \wedge x \leq 3 \wedge x \leq 2$ **for** $x :: nat$
  apply (*intro conjI*)
  prefers ‹- ≤ -›
  by *simp+*

end

5.1 Tagging

theory *Tagging*
imports *Main Subgoals HOL−Eisbach.Eisbach*
keywords *preferT prefersT* :: *prf-script* % *proof*
and *subgoalT subgoalsT* :: *prf-script-goal* % *proof*
begin

5.1.1 Basic Definitions and Theorems

definition *ASM-TAG* (‹¶›) **where**
ASM-TAG t ≡ *True*

definition *TAG* :: *'b* ⇒ *'a* :: {} ⇒ *'a* (‹(‹*open-block notation*=‹*infix TAG*››- | -)› [*13*, *13*] *14*)
where ‹*TAG t x* ≡ *x*›

abbreviation *tag-prop* (‹(‹*open-block notation*=‹*infix TAG*››- || -)› [*0*, *0*] *0*)
where *tag-prop t x* ≡ *PROP TAG t x*

lemma *TAG-cong*[*cong*]: *x* ≡ *y* ⟹ (*tag* | *x*) ≡ *tag* | (*y*::*'a*::{})
by *simp*

lemma *ASM-TAG-cong*[*cong*]: ¶ *tag* ⟷ ¶ *tag*
by *simp*

lemma *ASM-TAG-I*[*intro*!]: ¶ *x* **by** (*simp add*: *ASM-TAG-def*)

lemma *TAG-TrueI*[*intro*!, *simp*]: *tag* | *True*
by (*simp add*: *TAG-def*)

lemma *TAG-False*[*simp*]: (*tag* | *False*) ⟷ *False*
by (*simp add*: *TAG-def*)

lemma *TAG-false*[*elim*!]: (*tag* | *False*) ⟹ *P*
by (*simp add*: *TAG-def*)

lemma *ASM-TAG-aux1*:
PROP P ≡ (*True* ⟹ *PROP P*)
by *auto*

lemma *ASM-TAG-CONV1*:
PROP TAG t P ≡ (*ASM-TAG t* ⟹ *PROP P*)
unfolding *TAG-def ASM-TAG-def* **by** *auto*

lemma *disjE-tagged*:
(*P* ∨ *Q*) ⟹ (¶ *''l''* ⟹ *P* ⟹ *R*) ⟹ (¶ *''r''* ⟹ *Q* ⟹ *R*) ⟹ *R*

by *blast*

lemma *conjI-tagged*:
(¶ ''l'' ⟹ *P*) ⟹ (¶ ''r'' ⟹ *Q*) ⟹ *P* ∧ *Q*
by *blast*

— We will use the syntax ¶ *t* | *A* for tags that should end up in assumptions and conclusions:
lemma *assm-tagE*[*elim*!]:
assumes ¶ *t* | *A*
assumes ¶ *t* ⟹ *t* | *A* ⟹ *P*
shows *P*
using *assms* **unfolding** *TAG-def* **by** *auto*
— Consider (¶ ''l'' | *P*) ∨ (¶ ''r'' | *Q*)

ML ‹
fun unfold-tags ctxt = Local-Defs.unfold ctxt @{*thms TAG-def*}
val untagged-attr = Thm.rule-attribute [] (*fn context => unfold-tags* (*Context.proof-of context*))

fun unfold-tac' ctxt rewrs =
let
val rewrs = map (*Local-Defs.abs-def-rule ctxt*) *rewrs*
in
rewrite-goal-tac ctxt rewrs
end

fun thin-asm-tag-tac ctxt = REPEAT o eresolve-tac ctxt @{*thms thin-rl*[*of* ¶ *t for t*]}

fun untag-tac ctxt = thin-asm-tag-tac ctxt THEN' unfold-tac' ctxt @{*thms TAG-def*}
›

method-setup *untag* =
‹*Args.context* >> (*fn* - => *fn ctxt => SIMPLE-METHOD'* (*untag-tac ctxt*))›
strip tags

Subgoals Test

lemma
assumes *False*
shows
⋀*a b*. *A* ⟹ *B* ⟹ *C* ⟹ ''*foo*'' | *AA a b*
A ⟹ *B* ⟹ *C2* ⟹ ''*bar*'' | *B*
A ⟹ *B* ⟹ *C* ⟹ ''*bar*'' | *C*
A2 ⟹ *B* ⟹ *C22* ⟹ ''*bar*'' | *C*
⋀*b*. *A2* ⟹ *B* ⟹ *C22* ⟹ ''*foo*'' | *CC b*
⋀*a b*. *A2* ⟹ *B* ⟹ *C22* ⟹ *f a b*
⋀*c d*. *A2* ⟹ ¶ ''*foobar*'' ⟹ *B* ⟹ *C22* ⟹ *D c d*

```
  subgoals (concl) f -
    using ‹False› by simp
  subgoals (concl) ''foo'' | -
    using ‹False› by simp+
  subgoals ''bar'' | -
    using ‹False› by simp+
  subgoals (prems) ¶ ''foobar''
  oops
```

5.1.2 Conversions

```
ML ‹
fun extract-tag-trm trm =
  case trm of
    Const ‹Trueprop for Const ‹TAG - - for tag -›› => SOME tag
  | Const ‹TAG - - for tag -› => SOME tag
  | - => NONE

val tag-of-goal = extract-tag-trm o Logic.strip-assums-concl o Thm.prop-of
›

ML ‹
fun prems-conv cv ct =
  Conv.prems-conv (Thm.term-of ct |> Logic.count-prems) cv ct

fun concl-conv cv ct =
  Conv.concl-conv (Thm.term-of ct |> Logic.count-prems) cv ct

fun params-conv cv ctxt ct =
  Conv.params-conv (Thm.term-of ct |> Logic.strip-params |> length) cv ctxt ct
›

ML ‹
fun rm-head-tag-conv ctx = Conv.top-rewrs-conv @{thms TAG-def} ctx |> concl-conv
›

ML ‹
fun get-add-tag-rewrs - tag-ct =
  let
    val ctyp = Thm.ctyp-of-cterm tag-ct
  in [
   instantiate ‹'t = ‹ctyp› and t1 = tag-ct in
      lemma (schematic) ‹(PROP P) ≡ (ASM-TAG t1 ⟹ PROP P)› for t1 :: 't
and P
        by (unfold ASM-TAG-def, rule ASM-TAG-aux1)›
  ]
  end
val it = get-add-tag-rewrs @{context} @{cterm 1}
›
```

lemma *norm-tag*:
Trueprop (*t* | *P*) ≡ (*t* || *P*)
unfolding *TAG-def* **by** *auto*

ML ‹
fun push-concl-tag-to-assms ctxt thm =
let
val tag-opt = *tag-of-goal thm*
val tag = *the-default* @{*term* ''$''} *tag-opt*
val ctag = *Thm.cterm-of ctxt tag*
val push-thms = *get-add-tag-rewrs ctxt ctag*
val rewr-conv = *Conv.rewrs-conv push-thms*
val push-conv = *prems-conv rewr-conv*
val push = *Conv.fconv-rule push-conv*
val norm-conv = *Conv.rewrs-conv* @{*thms norm-tag*} |> *concl-conv*
val norm-conv = *Conv.top-sweep-conv* (*fn* - => *norm-conv*) *ctxt*
val assms-conv = *Conv.rewrs-conv* @{*thms ASM-TAG-CONV1*} |> *concl-conv*
val assms-conv = *Conv.top-sweep-conv* (*fn* - => *assms-conv*) *ctxt*
val rm = *Conv.fconv-rule* (*rm-head-tag-conv ctxt*)
val tags = *Thm.get-tags thm*
in
thm
|> *Conv.fconv-rule* (*prems-conv* (*norm-conv* ~~(~~~~))
|> *Conv.fconv-rule assms-conv*
|> (*case tag-opt of NONE* => *I* | - => *push*)
|> *rm*
|> *Thm.map-tags* (*K tags*)
end
›

ML ‹
val push-tags-attr =
Thm.rule-attribute []
(*fn context* => *push-concl-tag-to-assms* (*Context.proof-of context*))
›

attribute-setup *push-tags* =
‹*Scan.succeed* () >> (*fn* - => *push-tags-attr*)›
‹*push tags to assumptions*›

ML ‹
fun extract-asm-tag-trm trm =
case trm of
Const ‹*Trueprop for* ***Const*** ‹*ASM-TAG - for tag*›› => *SOME tag*
| - => *NONE*

fun mk-add-tag-thms t = [
instantiate ‹'*t* = ‹*Thm.ctyp-of-cterm t*› *and t* = *t in*

```
    lemma (schematic) ‹Trueprop y ≡ Trueprop (t | y)› for t :: 't by (unfold
TAG-def)›,
  instantiate ‹'t = ‹Thm.ctyp-of-cterm t› and t = t in
    lemma (schematic) ‹Trueprop y ≡ (t | Trueprop y)› for t :: 't by (unfold
TAG-def)›
]

fun get-list-tag t =
  let
    val tags = Logic.strip-assums-hyp t |> List.mapPartial extract-asm-tag-trm
    val tag = HOLogic.mk-list @{typ string} tags
  in
    if tags = [] then NONE else SOME tag
  end

fun add-tag-conv assm t ctxt = case get-list-tag t of
    NONE => Conv.all-conv
  | SOME tag =>
  let
    val ctag = Thm.cterm-of ctxt tag
    val push-thms =
      if assm then
        get-add-tag-rewrs ctxt ctag
      else
        mk-add-tag-thms ctag
    val rewr-conv = Conv.rewrs-conv push-thms |> concl-conv
  in
    rewr-conv
  end

fun tidy-tags-tac0 assm ctxt i thm =
  let
    val t = Thm.cprem-of thm i |> Thm.term-of
    val cconv = add-tag-conv assm t |> Conv.top-sweep-conv
    val conv = cconv ctxt |> Conv.try-conv
  in
    (CONVERSION conv THEN' thin-asm-tag-tac ctxt) i thm
  end
  handle THM - => Seq.empty

fun tidy-tags-tac assm ctxt =
  REPEAT o eresolve-tac ctxt @{thms assm-tagE} THEN' tidy-tags-tac0 assm ctxt

val tidy-tags-meth =       fn - => fn ctxt => SIMPLE-METHOD' (tidy-tags-tac
false ctxt)
val tidy-tags-assm-meth =  fn - => fn ctxt => SIMPLE-METHOD' (tidy-tags-tac
true  ctxt)
›
```

method-setup *tidy-tags-tac* =
  ‹*Args.context* >> *tidy-tags-meth*›
  compress tags into list of tags and tag goals

method-setup *tidy-tags-assm-tac* =
  ‹*Args.context* >> *tidy-tags-assm-meth*›
  compress tags into list of tags and tag assumptions

method *tidy-tags* = *changed* ‹*tidy-tags-tac*›

method *tidy-tags-assm* = *changed* ‹*tidy-tags-assm-tac*›

lemma
  $n > 0$ **if** $n > 1 \vee n > 2$ **for** $n :: nat$
  using *that*
  apply (*rule disjE-tagged*; *tidy-tags*)
  oops

lemma
  $n > 0$ **if** (¶ $''l''$ | $n > 1$) $\vee$ (¶ $''r''$ | $n > 2$) **for** $n :: nat$
  using *that*
  apply *standard*
  apply (*all tidy-tags*)
  oops

lemma *rm-ASM-TAG*:
  (*ASM-TAG A* $\Longrightarrow$ *PROP B*) $\equiv$ *PROP B*
  unfolding *ASM-TAG-def* **by** *auto*

ML ‹
```
fun add-tag-conv assm tag ctxt =
  let
    val ctag = Thm.cterm-of ctxt (HOLogic.mk-string tag)
    val push-thms =
      if assm then
        get-add-tag-rewrs ctxt ctag
      else
        mk-add-tag-thms ctag
    val rewr-conv = Conv.rewrs-conv push-thms
  in
    params-conv (fn - => rewr-conv) ctxt
  end

fun add-tags-conv (tag, tags) ctxt =
  let
    fun fold [] = Conv.all-conv
      | fold (- :: ts) = Conv.implies-conv Conv.all-conv (fold ts)
      | fold (t :: ts) = Conv.implies-conv (add-tag-conv false t ctxt) (fold ts)
    val rewr-conv = fold tags
```

```
   val rewr-conv = (if tag = - then rewr-conv else (rewr-conv then-conv add-tag-conv
true tag ctxt))
  in
    params-conv (fn - => rewr-conv) ctxt
  end

fun save-tags-fconv-rule conv thm =
  let
    val tags = Thm.get-tags thm
  in
    Conv.fconv-rule conv thm |> Thm.map-tags (K tags)
  end

fun tag-prems info ctxt =
  let
    fun fold [] = Conv.all-conv
      | fold (t :: ts) = Conv.implies-conv (add-tags-conv t ctxt) (fold ts)
    val conv = fold info
  in
    save-tags-fconv-rule conv
  end

fun add-tags-from-cases use-default-nums ctxt thm =
  let
    val has-cases = (Thm.get-tags thm |> AList.defined (op =)) case-names
     val maybe-info-from-assms = case AList.lookup (op =) (Thm.get-tags thm)
assm-names of
      SOME s => s
        |> space-explode ;
        |> List.map (fn s => if s =  then - else s)
        |> map (rpair []) |> SOME
    | NONE => NONE
    val (info, consumes-n) = Rule-Cases.get thm
    val info = map fst info
    val info = replicate consumes-n (-, []) @ info
    val info =
      if has-cases then
        info
      else case maybe-info-from-assms of
        NONE => if use-default-nums then info else []
      | SOME info => info
  in
    tag-prems info ctxt thm
  end

fun add-tags use-default-nums tags =
  if tags = [] then
    add-tags-from-cases use-default-nums
  else
```

```
    tag-prems (map (fn t => (t, [])) tags)

fun add-tags-from-cases use-default-nums tags = Thm.rule-attribute [] (fn context
=>
  add-tags use-default-nums tags (Context.proof-of context))

fun add-asm-tag tag =
  let
    fun get-tag thm = case tag of
      NONE =>
        AList.lookup (op =) (Thm.get-tags thm) name
        |> Option.map (space-explode . #> rev #> hd)
    | SOME t => SOME t
  in
    Thm.rule-attribute [] (fn context => fn thm =>
      case get-tag thm of
        NONE => thm
      | SOME tag =>
        thm |>
      (add-tag-conv false tag (Context.proof-of context) |> concl-conv |> save-tags-fconv-rule)
      )
  end

val rm-tags = Thm.rule-attribute [] (fn context =>
  Conv.top-rewrs-conv @{thms rm-ASM-TAG TAG-def} (Context.proof-of context)
|> save-tags-fconv-rule)
›

attribute-setup tags =
  ‹Scan.lift (Scan.repeat Args.name) >> add-tags-from-cases true›
  ‹add tags from case names›

attribute-setup tag =
  ‹Scan.lift (Scan.option Args.name) >> add-asm-tag›
  ‹add tag from rule name›

attribute-setup untag =
  ‹Scan.succeed () >> (fn - => rm-tags)›
  ‹strip tags›

thm conjI[tags l r] disjE[tags - l r] conjI[tags l r, untag] disjE[tags - l r, untag]

thm conjI[case-names A B, tags] disjE[case-names l [P] r [Q], consumes 1, tags]
nat-induct[tags]
 conjI[case-names A B, tags, untag] disjE[case-names l [P] r [Q], consumes 1,
tags, untag]
 nat-induct[tags, untag]
```

— the *assm-names* slot is meant to retain assumption names from theorem state-

ments:
thm *conjI*[*tags*] *disjE*[*tagged assm-names* ;*l*;*r*, *tags*]

thm *conjI*[*tag*] *conjI*[*tag conj*]

5.1.3 Globbing

ML ‹
```
datatype 't glob = WILDCARD | ANY | TOKEN of 't

fun
  match-exact [] [] = SOME ([], [])
| match-exact [] - = NONE
| match-exact (TOKEN t :: ps) (t' :: ts) = if t' = t then match-exact ps ts else
NONE
| match-exact (ANY :: ps) (- :: ts) = match-exact ps ts
| match-exact (WILDCARD :: ps) ts = SOME (WILDCARD :: ps, ts)
| match-exact - [] = NONE

fun
  matches-glob [] [] = true
| matches-glob [] - = false
| matches-glob (TOKEN t :: ps) (t' :: ts) = t' = t andalso matches-glob ps ts
| matches-glob (ANY :: ps) (- :: ts) = matches-glob ps ts
| matches-glob (WILDCARD :: ps) (t :: ts) = (
  case match-exact ps (t :: ts) of
    NONE => matches-glob (WILDCARD :: ps) ts
  | SOME (ps, ts) => matches-glob ps ts)
| matches-glob (WILDCARD :: ps) [] = matches-glob ps []
| matches-glob - [] = false

fun assert b = if b then () else error assert

val p1 = [WILDCARD, TOKEN a, WILDCARD, TOKEN c]
val p2 = [ANY, TOKEN a, WILDCARD, WILDCARD, TOKEN c]
val it1 = assert ([
  matches-glob p1 [b, a, c],
  matches-glob p1 [b, a, c, d],
  matches-glob p1 [a, c],
  matches-glob p1 [b, a, d]
] = [true, false, true, false])
val it2 = assert ([
  matches-glob p2 [b, a, c],
  matches-glob p2 [b, a, c, d],
  matches-glob p2 [a, c],
  matches-glob p2 [b, a, d]
] = [true, false, false, false])
```
›

ML ‹
```
fun tag-list-of goal =
  case extract-tag-trm goal of
      NONE => []
    | SOME t => map HOLogic.dest-string (HOLogic.dest-list t)
      handle TERM (dest-list, -) => [HOLogic.dest-string t]

fun matches-glob-term glob = matches-glob glob o tag-list-of
```
›

ML ‹
```
val parse-glob-token = (Parse.sym-ident || Parse.name)
  >> (fn
    * => WILDCARD
  | - => ANY
  | s   => TOKEN s
  )
val parse-glob = Scan.repeat1 parse-glob-token
```
›

5.1.4 Reordering Subgoals

ML ‹
```
val empty = (Vartab.empty, Vartab.empty)

fun matches thy pat trm = (Pattern.match thy (pat, trm) empty; true)
  handle Pattern.MATCH => false

fun matches-tag thy tag goal =
  case extract-tag-trm goal of
      NONE => false
    | SOME t => matches thy tag t

fun filter-thms thy pat = List.filter (fn thm => matches thy pat (Thm.prop-of
thm))

(* Find index of first subgoal in thm whose conclusion matches pred *)
fun find-subgoal pred thm =
  let
    val subgoals = Thm.prems-of thm
  in
    Library.find-index (fn goal => pred (Logic.strip-assums-concl goal)) subgoals
  end

fun prefer-by-pred-tac pred: tactic = fn thm =>
  let
    val i = find-subgoal pred thm + 1
  in
    if i > 0 then Tactic.prefer-tac i thm else no-tac thm
```

```
  end

fun prefer-by-tag-tac thy tag: tactic =
  prefer-by-pred-tac (matches-tag thy tag)

fun tidy-tags-all-meth ctxt = tidy-tags-tac false ctxt |> TRYALL |> SIMPLE-METHOD

fun prefer-pred pred st =
  let
    val st = Proof.assert-no-chain st
    val thy = Proof.theory-of st
  in
    st
    |> Proof.refine-singleton (Method.Basic tidy-tags-all-meth)
    |> Proof.refine-singleton
      (Method.Basic (fn - => METHOD (fn - => prefer-by-pred-tac (pred thy))))
  end

fun prefer-glob glob = prefer-pred (fn - => matches-glob-term glob)
›

ML ‹Outer-Syntax.command command-keyword‹preferT› select subgoal by tag
  (parse-glob >> (Toplevel.proof o prefer-glob))
›

lemma
  TAG ''p'' P TAG ''q'' Q PROP TAG ''r'' R TAG ''p'' P2
  preferT r
  oops

lemma
  ⋀t. TAG ''p'' P ∧ TAG ''q'' Q
  apply (rule conjI)
  preferT q
  oops

lemma
  P ∧ Q
  apply (rule conjI-tagged)
  preferT r
  preferT l
  oops
```

5.1.5 *subgoalT* and *subgoalsT*, and *prefersT*

```
ML ‹
fun orelseOpt f g x = case f x of NONE => g x | y => y
fun assm-name-of-tags trm =
  extract-tag-trm trm |>
```

```
    Option.mapPartial (
      orelseOpt
        (try (HOLogic.dest-string o hd o HOLogic.dest-list))
        (try HOLogic.dest-string)
      )
  |> the-default
›

ML ‹
fun chop-by - [] = []
  | chop-by eq (x :: xs) =
  let
    val (g, rest) = chop-prefix (eq x) xs
  in
    (x :: g) :: chop-by eq rest
  end
fun group-by k = chop-by (fn a => fn b => k a = k b) o sort-by k
›

ML ‹
val tag-name-of-prop = assm-name-of-tags o Thm.prop-of
fun note-of-thms (name, thms) =
  let
    val binding = Binding.qualified-name name
  in
    (((binding, []), [(thms, [untagged-attr])])
  end
fun note-thms' thms-tags ctx =
  let
    val grouped = group-by fst thms-tags |> map (fn g => (fst (hd g), map snd g))
  in
    Proof-Context.note-thmss  (map note-of-thms grouped) ctx |> snd
  end
fun note-thms thms =
  let
    val thms-tags = map (fn thm => (tag-name-of-prop thm, thm)) thms
  in
    note-thms' thms-tags
  end
fun note-thms-tac thms (ctx, thm) = (note-thms thms ctx, thm) |> Seq.succeed |>
Seq.make-results
fun note-thms-meth x = (Scan.succeed () >>
  (fn () => fn - => CONTEXT-METHOD (fn thms => note-thms-tac thms))) x
›

method-setup note-thms = ‹note-thms-meth›

ML ‹
local
```

```
val fact-binding =
  Parse.binding -- Parse.opt-attribs || Parse.attribs >> pair Binding.empty;
val opt-fact-binding =
  Scan.optional fact-binding Binding.empty-atts;

val for-params =
  Scan.optional
    (keyword ‹for› |--
      Parse.!!! ((Scan.option Parse.dots >> is-some) --
        (Scan.repeat1 (Parse.maybe-position Parse.name-position))))
    (false, []);

fun subgoal-cmd binding facts-name-opt param-specs st =
  let
    val (subgoal-focus, -) = Subgoal.subgoal-cmd binding facts-name-opt param-specs
st
    val names = #prems subgoal-focus |> map tag-name-of-prop — read tags first
    val st = Proof.refine-singleton (Method.Basic (fn ctxt => SIMPLE-METHOD'
(untag-tac ctxt))) st
   val (subgoal-focus, st) = Subgoal.subgoal-cmd binding facts-name-opt param-specs
st
    val prems = #prems subgoal-focus
    val prems-names = names ~~ prems — note premises with tags stripped
    val st = Proof.map-context (note-thms' prems-names) st
  in
    st
  end

val - =
  Outer-Syntax.command command-keyword ‹subgoalT›
    subgoal for tags
    (Scan.optional (fact-binding --| Parse.$$$ :) Binding.empty-atts --
     Scan.option parse-glob --
     (Scan.option (keyword ‹premises› |-- Parse.!!! opt-fact-binding)) --
     for-params >> (fn (((binding, glob-opt), prems-opt), fixes-opt) =>
       Toplevel.proof (
           Proof.refine-singleton (Method.Basic (fn ctxt => SIMPLE-METHOD'
(untag-tac ctxt)))
       o subgoal-cmd binding prems-opt fixes-opt
       o (case glob-opt of SOME x => prefer-glob x | - => fn x => x)
       o Proof.refine-singleton (Method.Basic tidy-tags-all-meth))));

val parse-keep =
  Scan.optional (Args.parens (Args.$$$ keep >> K true)) false;

fun tidy-then-protect-meth glob =
  let
    val pred = matches-glob-term glob o Logic.strip-assums-concl o Thm.term-of
    fun tidy-then-protect-tac ctxt =
```

```
      TRYALL (tidy-tags-tac false ctxt)
      THEN  prefer-and-protect-subgoals-tac pred ctxt
    val tidy-then-protect = Method.Basic
      (SIMPLE-METHOD o tidy-then-protect-tac)
  in tidy-then-protect end

fun get-unprotect-thms keep =
  @{thms protected-conjunction-def protected-prop-def} @
  (if keep then [] else @{thms TAG-def})

val - =
  Outer-Syntax.command command-keyword ‹subgoalsT›
    focus on all subgoals that match a given pattern within backward refinement
    (parse-keep -- parse-glob >> (fn (keep, glob) =>
      Toplevel.proofs (
        unprotect-and-finish (get-unprotect-thms keep) o
        #2 o
        Subgoal.subgoal Binding.empty-atts NONE (false, []) o
        Proof.refine-singleton (tidy-then-protect-meth glob)
      )))

val - =
  Outer-Syntax.command command-keyword ‹prefersT›
    focus on all subgoals that match a given pattern within backward refinement
    (parse-keep -- parse-glob >> (fn (keep, glob) =>
      Toplevel.proofs (
          unprotect-and-finish (get-unprotect-thms keep) o
          Proof.refine-singleton (tidy-then-protect-meth glob)
      )))
in end
›

lemma
  ¶ ''tag'' ⟹ TAG ''p'' P ⟹ TAG ''q'' Q ⟹ P ∧ Q
  subgoalT foo[simp]: * tag premises
  proof −
    show ?thesis — correct thesis is retained
      using p q ..
  qed
  thm foo
  done

lemma
  assumes TAG ''p'' P Q False
  shows TAG [''x''] True TAG [''y''] False TAG [''x'', ''y''] False TAG [''a'', ''b'']
True
  using assms
  apply −
  subgoal premises prems
```

```
    using prems
    apply note-thms
    thm p
    oops

lemma
  assumes TAG ''p'' P Q False
  shows TAG [''x''] True TAG [''y''] False TAG [''x'', ''y''] False TAG [''a'', ''b''] True
  using assms
  apply −
  subgoal premises prems
    using prems
    apply note-thms
    thm p
    oops

lemma
  assumes TAG ''p'' P TAG [''q'', ''r''] Q TAG [''q''] QQ False
  shows TAG [''x''] True TAG [''y''] False TAG [''x'', ''y''] False TAG [''a'', ''b''] True
  using assms
  apply −

  subgoalsT (keep) x ∗

  subgoalT x premises prems
    thm p q
    using ‹False› by simp
  subgoalT x - premises prems
    thm p q
    using ‹False› by simp
  done
  subgoalT y premises prems
    thm p q
    using ‹False› by simp
  subgoalsT ∗
    using ‹False› by simp+
  oops

lemma
  assumes TAG ''p'' P TAG [''q'', ''r''] Q False
  shows
     TAG [''x''] True TAG [''y''] False P TAG [''x'', ''y''] False TAG [''a'', ''b''] True
  using assms
  apply −
  subgoalT x premises prems
    using prems apply −
```

using ‹*False*› **by** *simp*

subgoalT *y* **premises** *prems*
using *prems* **apply** −
using ‹*False*› **by** *simp*

subgoalT premises *prems*
using *prems* **apply** −
using ‹*False*› **by** *simp*

subgoalT *x y* **premises** *prems*
using *prems* **apply** −
using ‹*False*› **by** *simp*

subgoalT *a b* **premises** *prems*
using *prems* **apply** −
thm *p q*
using ‹*False*› **by** *simp*
done

lemma
n > *0* **if** (¶ *''l''* | *n* > *1*) ∨ (¶ *''r''* | *n* > *2*) **for** *n* :: *nat*
using *that*
apply *standard*
subgoalT *l* **premises**
using *l* **by** *simp*
subgoalT *r* **premises**
using *r* **by** *simp*
done

lemma
n > *0* **if** (¶ *''l''* | *n* > *9*) ∨ (¶ *''s''* | *n* > *2*) ∨ (¶ *''s''* | *n* > *0*) **for** *n* :: *nat*
using *that*
apply (*elim disjE*)
prefersT *s* — tidies all goals before tags are matched
apply *simp*+
subgoalT *l* **premises**
using *l* **by** *simp*
done

5.1.6 Syntax

ML ‹
structure Tagging-Cartouche-Syntax =
struct
val tagging-cartouche-syntax =
Attrib.setup-config-bool @{*binding tagging-cartouche-syntax*} (*K true*)

local

```
    fun mk-char (s, pos) =
      let
        val c =
          if Symbol.is-ascii s then ord s
          else error (String literal contains illegal symbol:  ^ quote s ^ Position.here
pos);
      in list-comb (Syntax.const const-syntax ‹Char›, String-Syntax.mk-bits-syntax
8 c) end;

    fun mk-string [] = Const (const-syntax ‹Nil›, typ ‹string›)
      | mk-string (s :: ss) =
          Syntax.const const-syntax ‹Cons› $ mk-char s $ mk-string ss;

    fun string-tr content arg =
      let fun err () = raise TERM (string-tr, [arg]) in
        (case arg of
          (c as Const (syntax-const ‹-constrain›, -)) $ Free (s, -) $ p =>
            (case Term-Position.decode-position1 p of
              SOME {pos, ...} => c $ mk-string (content (s, pos)) $ p
            | NONE => err ())
        | - => err ())
      end

  in
    fun asm-tag-tr content ctxt args =
      let fun err () = raise TERM (asm-tag-tr, args) in
        if Config.get ctxt tagging-cartouche-syntax then
          case args of
            [tag] => Syntax.const const-name ‹ASM-TAG› $ string-tr content tag
          | - => err ()
        else raise Match
      end
    fun tag-tr content ctxt args =
      let fun err () = raise TERM (tag-tr, args) in
        if Config.get ctxt tagging-cartouche-syntax then
          case args of
            [tag, trm] => Syntax.const const-name ‹TAG› $ string-tr content tag $
trm
          | - => err ()
        else raise Match
      end
  end

  local
    val bit-type = typ ‹bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool
⇒ char›
    fun mk-char (Const (const-syntax ‹False›, -)) = Const (const-name ‹False›,
typ ‹bool›)
      | mk-char (Const (const-syntax ‹True›,  -)) = Const (const-name ‹True›,
```

```
typ ‹bool›)
    | mk-char (Const (const-syntax ‹Char›, -)) = Const (const-name ‹Char›,
bit-type)
    | mk-char (a $ b) = mk-char a $ mk-char b
    | mk-char - = error Not a syntax bit

  fun mk-string (Const (const-syntax ‹Nil›, typ ‹string›)) = term ‹''''›
    | mk-string (Const (const-syntax ‹Cons›, typ ‹char ⇒ string ⇒ string›) $ x
$ xs) =
      Const (const-name ‹Cons›, typ ‹char ⇒ string ⇒ string›) $ mk-char x $
mk-string xs
    | mk-string - = error Not a string

   fun mk-list (Const (const-syntax ‹Nil›, typ ‹string list›)) = term ‹[]::string
list›
     | mk-list (Const (const-syntax ‹Cons›, typ ‹string ⇒ string list ⇒ string
list›) $ x $ xs) =
      Const (const-name ‹Cons›, typ ‹string ⇒ string list ⇒ string list›)
      $ mk-string x $ mk-list xs
    | mk-list - = error Not a string list

  fun string-tp arg =
    mk-string arg |> HOLogic.dest-string |> cartouche

  fun string-list-tp arg =
   mk-list arg |> HOLogic.dest-list |> map HOLogic.dest-string |> space-implode
|> cartouche

  fun string-or-list-tp t =
    if fastype-of t = typ ‹string list› then string-list-tp t else
    if fastype-of t = typ ‹string›      then string-tp t
    else raise TERM (Not a string tag, [t])
in
  fun asm-tag-tp cnst ctxt args =
    if Config.get ctxt tagging-cartouche-syntax then
      case args of
        [x] => (Free (¶ ^ string-tp x, typ ‹-›)
          handle ERROR - => Const (cnst, typ ‹-›) $ x)
        | - => raise Match
    else raise Match
  fun tag-tp cnst ctxt args =
    if Config.get ctxt tagging-cartouche-syntax then
      case args of
        [tag, trm] => (
          Const (cnst, typ ‹-›) $ Free (string-or-list-tp tag, typ ‹-›) $ trm
          handle TERM - => Const (cnst, typ ‹-›) $ tag $ trm)
      | - => raise Match
    else raise Match
end
```

end
›

syntax *-ASM-TAG* :: ‹*cartouche-position* ⇒ *string*› (¶-)

parse-translation ‹
[(***syntax-const*** ‹*-ASM-TAG*›,
Tagging-Cartouche-Syntax.asm-tag-tr (*Symbol-Pos.cartouche-content o Symbol-Pos.explode*))]
›

syntax *-TAG* :: ‹*cartouche-position* ⇒ *'a* ⇒ *string*› (- | - [*13*, *13*] *14*)

parse-translation ‹
[(***syntax-const*** ‹*-TAG*›,
Tagging-Cartouche-Syntax.tag-tr (*Symbol-Pos.cartouche-content o Symbol-Pos.explode*))]
›

print-translation ‹
[(***const-syntax*** ‹*ASM-TAG*›, *Tagging-Cartouche-Syntax.asm-tag-tp* ***syntax-const*** ‹*-ASM-TAG*›)]
›

syntax *-TAG* :: ‹*cartouche-position* ⇒ *'a* ⇒ *string*› (- | - [*13*, *13*] *14*)

print-translation ‹
[(***const-syntax*** ‹*TAG*›, *Tagging-Cartouche-Syntax.tag-tp* ***syntax-const*** ‹*-TAG*›)]
›

term ‹‹› | *a*›
term ‹‹*ab*› | *a*›
term ‹‹*a-b'*,*c*› | *abc*›
term ¶‹*AB*› | *a* = *b*
term ‹['*'a*', *'b*'] | *t*›
term ‹(‹*a*› | *b*) *x*›
term ‹(*t* | *b*) *x*›
term ‹*t* | *x*›
term ‹¶ *t*›

end

Chapter 6

Verification Condition Generator *runs-to-vcg*

theory *Basic-Runs-To-VCG*
imports
Named-Rules
HOL−Eisbach.Eisbach-Tools
Tagging
begin

6.1 Marked Assumptions

ML ‹

```
structure Marked-Assumptions =
struct

structure Data = Theory-Data
(
  type T = (thm -> Proof.context -> Proof.context) Symtab.table;
  val empty = Symtab.empty;
  val merge = Symtab.merge (K true);
);

fun import-assm ctxt thm =
  ( case head-of (HOLogic.dest-Trueprop (Thm.prop-of thm)) of
      Const (name, -) =>
        (case Symtab.lookup (Data.get (Proof-Context.theory-of ctxt)) name of
          SOME add-thm => SOME (add-thm thm ctxt)
        | NONE => NONE)
    | - => NONE )
  handle TERM - => NONE

fun with-assms ctxt old-prems tac =
```

```
  Subgoal.FOCUS (fn { context = ctxt1, prems, ... } =>
    let
      val (ctxt4, prems') = (ctxt1, old-prems) |> fold (fn thm => fn (ctxt2, prems')
=>
          case import-assm ctxt2 thm of
            SOME ctxt3 => (ctxt3, prems')
          | NONE => (ctxt2, thm::prems')) prems
    in
      tac ctxt4 prems'
    end) ctxt 1

fun add-marker name f = Data.map (Symtab.update (name, f))

end ;

›

attribute-setup parse-ASSMs =
  ‹Scan.succeed (Thm.declaration-attribute (fn thm => Context.map-proof (fn ctxt
=>
    case Marked-Assumptions.import-assm ctxt thm of SOME ctxt => ctxt | NONE
=> ctxt)))›

named-theorems remove-ASSMs Remove the markers for marked assumptions

method cleanup-ASSMs = simp-all only: remove-ASSMs

definition SIMP-ASSM :: bool ⇒ bool where [remove-ASSMs]: SIMP-ASSM P
⟷ P
lemma SIMP-ASSM-D: SIMP-ASSM P ⟹ P by (simp add: SIMP-ASSM-def)
setup ‹ Marked-Assumptions.add-marker @{const-name SIMP-ASSM} (fn thm
=> fn ctxt =>
  ctxt addsimps [thm RS @{thm SIMP-ASSM-D}]) ›
```

6.2 *THEN-ALL-NEW-FORWARD*

ML ‹

```
fun each-protected (t : int -> tactic) : tactic =
  let
    fun work-at i st =
      if Thm.nprems-of st = i then Seq.single st
      else
        rotate-prems i st
        |> Goal.protect 1
        |> t i
        |> Seq.maps (fn st =>
            Goal.conclude st
            |> rotate-prems (~ i)
```

```
        |> work-at (i + Thm.nprems-of st) )

  in work-at 0 end

(* THEN-ALL-NEW-FORWARD m1 m2, applies first m1 and then all all new
subgoals m2 is applied.

m1 THEN-ALL-NEW m2 (i.e. m1; m2) applies m2 in the reverse order. This is
a problem when schematic
variables need to be resolved *)

fun THEN-ALL-NEW-FORWARD' (t1 : tactic) (t2 : int -> tactic) (st : thm) =
  Goal.protect 1 st
  |> t1
  |> Seq.maps (each-protected t2)
  |> Seq.map Goal.conclude

fun THEN-ALL-NEW-FORWARD (t1 : tactic) (t2 : tactic) =
  THEN-ALL-NEW-FORWARD' t1 (K t2)

›

method-setup all-new = ‹
(Method.text-closure -- Method.text-closure) >> (fn (m1, m2) => fn ctxt =>
fn facts =>
  let
    val tac1 = method-evaluate m1 ctxt facts
    val tac2 = method-evaluate m2 ctxt facts
  in Method.RUNTIME (SIMPLE-METHOD (THEN-ALL-NEW-FORWARD tac1
tac2) facts) end)
›
```

6.3 Basic VCG

```
named-theorems runs-to-vcg-cong-state-only
named-theorems runs-to-vcg-weaken
named-theorems runs-to-vcg-cong-program-only
named-rules (intro) runs-to-vcg

named-theorems runs-to-vcg-post-elims
named-theorems runs-to-vcg-post-intros

declare conjI[runs-to-vcg-post-intros]
declare disjE[runs-to-vcg-post-elims]

ML ‹
signature RUNS-TO-VCG =
sig
  val apply-runs-to:
```

```
    bool -> (Proof.context -> int -> tactic) option ->
      (Proof.context -> (unit -> string) -> tactic) ->
        term -> Proof.context -> tactic
  val prepare:
    {do-nosplit: bool, no-unsafe-hyp-subst: bool} -> Proof.context -> tactic
  val runs-to-vcg-tac:
    (Proof.context -> int -> tactic) option ->
      int -> (Proof.context -> (unit -> string) -> tactic) -> bool ->
          {do-nosplit: bool, no-unsafe-hyp-subst: bool} ->
            (Proof.context -> tactic) -> Proof.context -> tactic
  val split-paired-all: Proof.context -> int -> tactic
  val trace-tac: 'a -> (unit -> string) -> 'b -> 'b Seq.seq
  val no-trace-tac: Proof.context -> (unit -> string) -> tactic
  val trace-print-tac: Proof.context -> (unit -> string) -> thm -> thm Seq.seq
end

structure Runs-To-VCG : RUNS-TO-VCG =
struct

(*Print the current proof state and pass it on.*)
fun trace-tac - msg st = (tracing (msg ()); Seq.single st);
fun trace-print-tac ctxt msg st = print-tac ctxt (msg ()) st;
val no-trace-tac = K (K (all-tac)): Proof.context -> (unit -> string) -> tactic

fun split-paired-all ctxt =
  simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms split-paired-all})

fun prepare {do-nosplit, no-unsafe-hyp-subst} ctxt =
let
  val post-intros = Named-Theorems.get ctxt @{named-theorems runs-to-vcg-post-intros}
  val post-elims = Named-Theorems.get ctxt @{named-theorems runs-to-vcg-post-elims}
in
  REPEAT-DETERM-FIRST (fn n =>
    CHANGED (split-paired-all ctxt n)
    ORELSE
    CHANGED (eresolve-tac ctxt @{thms exE bexE conjE} n)
    ORELSE
    CHANGED (resolve-tac ctxt @{thms allI ballI impI} n)
    ORELSE
    CHANGED (full-simp-tac
      ((fold Simplifier.add-cong
        (Named-Theorems.get ctxt @{named-theorems runs-to-vcg-cong-state-only})
ctxt)) n)
    ORELSE
    (if no-unsafe-hyp-subst
      then CHANGED (asm-full-simp-tac (put-simpset HOL-basic-ss ctxt) n)
      else CHANGED (bound-hyp-subst-tac ctxt n))
    ORELSE
    (if do-nosplit then no-tac else CHANGED (eresolve-tac ctxt post-elims n))
```

```
    ORELSE
    CHANGED (resolve-tac ctxt post-intros n))
  ORELSE
  all-tac
end

local
fun parse-program1 ctxt g cong-program-thm =
  let
    fun lhs t = t |> HOLogic.dest-Trueprop |> HOLogic.dest-eq |> #1
    val pat = Thm.concl-of cong-program-thm |> lhs
    val concl = HOLogic.dest-Trueprop (Logic.strip-assums-concl g)
    val (key, -) = case Thm.prems-of cong-program-thm of [prem] =>
          lhs prem |> Term.dest-Var
      | - => error Runs-To-VCG parse-program: wrong format of rule
  in
    case Unify.unifiers (Context.Proof ctxt, Envir.init, [(pat, concl)]) |> Seq.pull
    of SOME ((env,-), -) =>
      (case Vartab.lookup (Envir.term-env env) key of NONE => NONE
          | SOME (-, t) => SOME t)
    | NONE => NONE
  end
in
fun parse-program ctxt t = get-first (parse-program1 ctxt t)
  (Named-Theorems.get ctxt @{named-theorems runs-to-vcg-cong-program-only})
end

fun with-rules tac ctxt =
  let
    fun tac' ctxt thms i = tac ctxt thms
  in
    Named-Rules.with-rules @{named-rules runs-to-vcg} tac' ctxt 1
  end

fun apply-attributes attributes thm ctxt =
  fold (fn attr => fn (thm, ctxt) => Thm.apply-attribute attr thm ctxt) attributes
(thm, ctxt)

fun apply-runs-to add-tags splitter-opt trace prg = with-rules (fn ctxt => fn rules
=>
  let
    val rules =
      if add-tags then
        map (fn thm =>
          apply-attributes
            [add-asm-tag NONE, add-tags-from-cases false [], push-tags-attr]
            thm (Context.Proof ctxt)
          |> fst
        ) rules
```

```
      else rules
    val n = length rules
    fun t i thm msg () =
      let
        val d = Thm-Name.print (Thm.derivation-name thm)
      in
        At_ ^ Syntax.string-of-term ctxt (head-of prg) ^ apply_ ^ msg ^ :_ ^
          implode-space [d, ( ^ string-of-int (i + 1) ^ of_ ^ string-of-int n ^ )]
      end
    fun MAP f = FIRST (map-index f rules)
  in
    TRY (DETERM (
      MAP (fn (i, thm) => resolve-tac ctxt [thm] 1 THEN trace ctxt (t i thm rule))
      ORELSE (case splitter-opt of
          SOME splitter =>
            trace ctxt (fn () => Trying splitter on_ ^ Syntax.string-of-term ctxt prg)
THEN
            splitter ctxt 1
        | - => no-tac)
      ORELSE
      MAP (fn (i, thm) =>
        resolve-tac ctxt
       (Named-Theorems.get ctxt @{named-theorems Basic-Runs-To-VCG.runs-to-vcg-weaken})
1 THEN
        resolve-tac ctxt [thm] 1 THEN
        trace ctxt (t i thm weakened rule))))
  end)

fun runs-to-vcg-tac splitter-opt n trace add-tags options solver ctxt =
let
  fun flat-trace f () =     ^ f ()
  fun runs-to-vcg-loop n trace ctxt prems =
    if n = 0 then all-tac else
    SUBGOAL (fn (g, -) =>
      case parse-program ctxt g of
        SOME prg =>
          THEN-ALL-NEW-FORWARD
            (CHANGED (apply-runs-to add-tags splitter-opt trace prg ctxt))
            (THEN-ALL-NEW-FORWARD
              (prepare options ctxt)
              (Marked-Assumptions.with-assms ctxt prems
                (runs-to-vcg-loop (n - 1) (fn ctxt => fn f => trace ctxt (flat-trace
f)))))
        ORELSE trace ctxt (fn () => No rule applicable for_ ^ Syntax.string-of-term
ctxt prg)
      | NONE =>
        TRY (SOLVED' (Method.insert-tac ctxt prems THEN'
          (K (solver ctxt) ORELSE' assume-tac ctxt)) 1))
      1
```

```
in
  runs-to-vcg-loop n trace ctxt []
end

end
›

method-setup runs-to-vcg =
‹ (Scan.lift (Args.mode trace) --
    Scan.lift (Args.mode tags) --
    Scan.lift (Args.mode nosplit) --
    Scan.lift (Args.mode no-unsafe-hyp-subst) --
    Scan.optional (Scan.lift (Args.parens Parse.nat)) (~1) --
    Scan.option Method.text-closure) >>
    (fn (((((do-trace, do-tags), do-nosplit), do-no-unsafe-hyp-subst), n), text) =>
fn ctxt =>
      let val tac = (case text of
          NONE => (fn - => no-tac)
        | SOME txt => fn ctxt => NO-CONTEXT-TACTIC ctxt (Method.evaluate
txt ctxt []))
      in SIMPLE-METHOD (Runs-To-VCG.runs-to-vcg-tac NONE n
        (if do-trace then Runs-To-VCG.trace-tac else Runs-To-VCG.no-trace-tac)
        do-tags
        {do-nosplit=do-nosplit, no-unsafe-hyp-subst=do-no-unsafe-hyp-subst}
        tac ctxt) end) ›
```

6.4 Setup for Tagging

```
lemma push-tag-to-assm:
  t | P if ¶ t ⟹ P
  using that unfolding ASM-TAG-def TAG-def by simp

lemma tagE:
  tag | Q ⟹ (Q ⟹ P) ⟹ P
  by (auto simp: TAG-def)

bundle basic-vcg-tagging-setup
begin

lemmas [runs-to-vcg-post-intros] = push-tag-to-assm TAG-TrueI ASM-TAG-I
lemmas [runs-to-vcg-post-elims]  = tagE

end

end
```

theory *Runs-To-VCG*
imports
Basic-Runs-To-VCG
begin

lemma *transfer-conj-imp*: *rel-fun* (=) (*rel-fun* (⟶) (⟶)) (∧) (∧)
by (*auto simp*: *rel-fun-def*)

lemma *transfer-all-imp*: *rel-fun* (*rel-set R*) (*rel-fun* (*rel-fun R* (⟶)) (⟶)) (*Ball*) (*Ball*)
by (*auto simp*: *rel-fun-def rel-set-def*)

6.5 *runs-to-vcg*

method-setup *trace-goals* = ‹
Scan.lift Parse.string >>
(*fn msg* => *fn ctxt* => *fn using* => *Method.RUNTIME* (*CONTEXT-TACTIC* (*print-tac ctxt msg*)))
›

method-setup *trace* = ‹
(*Scan.lift Parse.string*) >>
(*fn msg* => *fn ctxt* => *fn using* => *Method.RUNTIME* (*CONTEXT-TACTIC*
(*fn st* => (*tracing msg*; *Seq.single st*))))
›

6.6 Check

thm *runs-to-vcg* [*no-vars*]
end

Chapter 7

Proof Methods

theory *Eisbach-Methods*
imports
Subgoal-Methods
HOL−Eisbach.Eisbach-Tools
Rule-By-Method
begin

7.1 Debugging methods

method *print-concl* = (*match* **conclusion in** *P* **for** *P* ⇒ ‹*print-term P*›)

method-setup *print-raw-goal* = ‹*Scan.succeed* (*fn ctxt* => *fn facts* =>
(*fn* (*ctxt*, *st*) => (*Output.writeln* (*Thm.string-of-thm ctxt st*);
Seq.make-results (*Seq.single* (*ctxt*, *st*)))))›

ML ‹*fun method-evaluate text ctxt facts* =
NO-CONTEXT-TACTIC ctxt
(*Method.evaluate-runtime text ctxt facts*)›

method-setup *print-headgoal* =
‹*Scan.succeed* (*fn ctxt* =>
fn - => *fn* (*ctxt'*, *thm*) =>
((*SUBGOAL* (*fn* (*t*,-) =>
(*Output.writeln*
(*Pretty.string-of* (*Syntax.pretty-term ctxt t*)); *all-tac*)) *1 thm*);
(*Seq.make-results* (*Seq.single* (*ctxt'*, *thm*)))))›

7.2 Simple Combinators

method-setup *defer-tac* = ‹*Scan.succeed* (*fn* - => *SIMPLE-METHOD* (*defer-tac 1*))›
method-setup *prefer-last* = ‹*Scan.succeed* (*fn* - => *SIMPLE-METHOD* (*PRIMITIVE*

```
(Thm.permute-prems 0 ~1)))›

method-setup all =
 ‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
   let
     fun tac i st' =
       Goal.restrict i 1 st'
       |> method-evaluate m ctxt facts
       |> Seq.map (Goal.unrestrict i)

   in SIMPLE-METHOD (ALLGOALS tac) facts end)
›

method-setup determ =
 ‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
   let
     fun tac st' = method-evaluate m ctxt facts st'

   in SIMPLE-METHOD (DETERM tac) facts end)
› ‹Run the given method, but only yield the first result›

ML ‹
fun require-determ (method : Method.method) facts st =
  case method facts st |> Seq.filter-results |> Seq.pull of
    NONE => Seq.empty
  | SOME (r1, rs) =>
      (case Seq.pull rs of
        NONE => Seq.single r1 |> Seq.make-results
      | - => Method.fail facts st);

fun require-determ-method text ctxt =
  require-determ (Method.evaluate-runtime text ctxt);
›

method-setup require-determ =
  ‹Method.text-closure >> require-determ-method›
  ‹Run the given method, but fail if it returns more than one result›

method-setup changed =
 ‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
   let
     fun tac st' = method-evaluate m ctxt facts st'

   in SIMPLE-METHOD (CHANGED tac) facts end)
›


method-setup timeit =
 ‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
```

```
  let
    fun timed-tac st seq = Seq.make (fn () => Option.map (apsnd (timed-tac st))
      (timeit (fn () => (Seq.pull seq))));

    fun tac st' =
      timed-tac st' (method-evaluate m ctxt facts st');

  in SIMPLE-METHOD tac [] end)
›
```

```
method-setup timeout =
‹Scan.lift Parse.int -- Method.text-closure >> (fn (i,m) => fn ctxt => fn facts
=>
  let
    fun str-of-goal th = Pretty.string-of (Goal-Display.pretty-goal ctxt th);

    fun limit st f x = Timeout.apply (Time.fromSeconds i) f x
      handle Timeout.TIMEOUT - => error (Method timed out:\n ^ (str-of-goal
st));

      fun timed-tac st seq = Seq.make (limit st (fn () => Option.map (apsnd
(timed-tac st))
      (Seq.pull seq)));

    fun tac st' =
      timed-tac st' (method-evaluate m ctxt facts st');

  in SIMPLE-METHOD tac [] end)
›
```

```
method repeat-new methods m = (m ; (repeat-new ‹m›)?)
```

The following *fails* and *succeeds* methods protect the goal from the effect of a method, instead simply determining whether or not it can be applied to the current goal. The *fails* method inverts success, only succeeding if the given method would fail.

```
method-setup fails =
‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun fail-tac st' =
      (case Seq.pull (method-evaluate m ctxt facts st') of
         SOME - => Seq.empty
       | NONE => Seq.single st')

  in SIMPLE-METHOD fail-tac facts end)
›
```

```
method-setup succeeds =
```

```
‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun can-tac st' =
      (case Seq.pull (method-evaluate m ctxt facts st') of
         SOME (st'',-) => Seq.single st'
       | NONE => Seq.empty)

  in SIMPLE-METHOD can-tac facts end)
›
```

This method wraps up the "focus" mechanic of match without actually doing any matching. We need to consider whether or not there are any assumptions in the goal, as premise matching fails if there are none.

If the *fails* method is removed here, then backtracking will produce a set of invalid results, where only the conclusion is focused despite the presence of subgoal premises.

```
method focus-concl methods m =
  ((fails ‹erule thin-rl›, match conclusion in - ⇒ ‹m›)
  | match premises (local) in H:- (multi) ⇒ ‹m›)
```

repeat applies a method a specific number of times, like a bounded version of the '+' combinator.

usage: apply (repeat n *text*)

- Applies the method *text* to the current proof state n times. - Fails if *text* can't be applied n times.

```
ML ‹
  fun repeat-tac count tactic =
    if count = 0
    then all-tac
    else tactic THEN (repeat-tac (count − 1) tactic)
›

method-setup repeat = ‹
  Scan.lift Parse.nat −− Method.text-closure >> (fn (count, text) => fn ctxt =>
fn facts =>
    let val tactic = method-evaluate text ctxt facts
    in SIMPLE-METHOD (repeat-tac count tactic) facts end)
›

notepad begin
  fix A B C
  assume assms: A B C
```

repeat: simple repeated application.

```
  have A ∧ B ∧ C ∧ True
```

repeat: fails if method can't be applied the specified number of times.

```
    apply (fails ‹repeat 4 ‹rule conjI, rule assms››)
```

apply (*repeat 3* ‹*rule conjI*, *rule assms*›)
by (*rule TrueI*)

repeat: application with subgoals.

have $A \wedge A\ B \wedge B\ C \wedge C$
apply −

We have three subgoals. This *repeat* call consumes two of them.

apply (*repeat 2* ‹*rule conjI*, (*rule assms*)+›)

One subgoal remaining...

apply (*rule conjI*, (*rule assms*)+)
done

end

Literally a copy of the parser for *subgoal-tac* composed with an analogue of **prefer**.

Useful if you find yourself introducing many new facts via *subgoal-tac*, but prefer to prove them immediately rather than after they're used.

```
setup ‹
  Method.setup binding ‹prop-tac›
    (Args.goal-spec -- Scan.lift (Scan.repeat1 Parse.embedded-inner-syntax --
Parse.for-fixes) >>
      (fn (quant, (props, fixes)) => fn ctxt =>
        (SIMPLE-METHOD'' quant
          (EVERY' (map (fn prop => Rule-Insts.subgoal-tac ctxt prop fixes) props)
            THEN'
            (K (prefer-tac 2))))))
    insert prop (dynamic instantiation), introducing prop subgoal first
›

notepad begin {
  fix xs
  assume assms: list-all even (xs :: nat list)

  from assms have even (sum-list xs)
    apply (induct xs)
     apply simp

    Inserts the desired proposition as the current subgoal.

    apply (prop-tac list-all even xs)
     subgoal by simp

    The prop list-all even xs is now available as an assumption. Let's add
another one.

    apply (prop-tac even (sum-list xs))
     subgoal by simp
```

Now that we've proven our introduced props, use them!

```
  apply clarsimp
  done
}
end
```

7.3 Advanced combinators

7.3.1 Protecting goal elements (assumptions or conclusion) from methods

```
context
begin

private definition protect-concl x ≡ ¬ x
private definition protect-false ≡ False

private lemma protect-start: (protect-concl P ⟹ protect-false) ⟹ P
  by (simp add: protect-concl-def protect-false-def) (rule ccontr)

private lemma protect-end: protect-concl P ⟹ P ⟹ protect-false
  by (simp add: protect-concl-def protect-false-def)

method only-asm methods m =
  (match premises in H[thin]:- (multi,cut) ⇒
    ‹rule protect-start,
     match premises in H'[thin]:protect-concl - ⇒
       ‹insert H,m;rule protect-end[OF H']››)

method only-concl methods m = (focus-concl ‹m›)

end

notepad begin
 fix D C
  assume DC:D ⟹ C
  have D ∧ D ⟹ C ∧ C
  apply (only-asm ‹simp›) — stash conclusion before applying method
  apply (only-concl ‹simp add: DC›) — hide premises from method
  by (rule DC)

end
```

7.3.2 Safe subgoal folding (avoids expanding meta-conjuncts)

Isabelle's goal mechanism wants to aggressively expand meta-conjunctions if they are the top-level connective. This means that *fold-subgoals* will immediately be unfolded if there are no common assumptions to lift over.

To avoid this we simply wrap conjunction inside of conjunction' to hide it from the usual facilities.

context begin

definition
conjunction′ :: *prop* ⇒ *prop* ⇒ *prop* (**infixr** ‹&^&› *2*) **where**
conjunction′ A B ≡ (*PROP A* &&& *PROP B*)

In general the context antiquotation does not work in method definitions. Here it is fine because `Conv.top_sweep_conv` is just over-specified to need a Proof.context when anything would do.

method *safe-meta-conjuncts* =
raw-tactic
‹*REPEAT-DETERM*
(*CHANGED-PROP*
(*PRIMITIVE*
(*Conv.gconv-rule* ((*Conv.top-sweep-conv* (*K* (*Conv.rewr-conv* @{*thm conjunction′-def*[*symmetric*]})) @{*context*})) *1*)))›

method *safe-fold-subgoals* = (*fold-subgoals* (*prefix*), *safe-meta-conjuncts*)

lemma *atomize-conj′* [*atomize*]: (*A* &^& *B*) == *Trueprop* (*A* & *B*)
by (*simp add*: *conjunction′-def*, *rule atomize-conj*)

lemma *context-conjunction′I*:
PROP P ⟹ (*PROP P* ⟹ *PROP Q*) ⟹ *PROP P* &^& *PROP Q*
apply (*simp add*: *conjunction′-def*)
apply (*rule conjunctionI*)
apply *assumption*
apply (*erule meta-mp*)
apply *assumption*
done

lemma *conjunction′I*:
PROP P ⟹ *PROP Q* ⟹ *PROP P* &^& *PROP Q*
by (*rule context-conjunction′I*; *simp*)

lemma *conjunction′E*:
assumes *PQ*: *PROP P* &^& *PROP Q*
assumes *PQR*: *PROP P* ⟹ *PROP Q* ⟹ *PROP R*
shows
PROP R
apply (*rule PQR*)
apply (*rule PQ*[*simplified conjunction′-def*, *THEN conjunctionD1*])
by (*rule PQ*[*simplified conjunction′-def*, *THEN conjunctionD2*])

end

notepad begin

fix D C E

assume DC: $D \wedge C$
have D $C \wedge C$
apply $-$
apply (*safe-fold-subgoals*, *simp*, *atomize* (*full*))
apply (*rule DC*)
done

end

7.4 Utility methods

7.4.1 Finding a goal based on successful application of a method

method-setup *find-goal* =
‹*Method.text-closure* >> (*fn m* => *fn ctxt* => *fn facts* =>
let
fun prefer-first i = *SELECT-GOAL*
(*fn st'* =>
(*case Seq.pull* (*method-evaluate m ctxt facts st'*) *of*
SOME (*st''*,-) => *Seq.single st''*
| *NONE* => *Seq.empty*)) *i THEN prefer-tac i*

in SIMPLE-METHOD (*FIRSTGOAL prefer-first*) *facts end*)›

Ensure that the proof state is in a certain case of a case distinction:

method *in-case* **for** x = *match* **premises in** t = x **for** $t \Rightarrow$ *succeed*

Focus on a case in a case distinction:

method *find-case* **for** x = *find-goal* ‹*in-case x*›

notepad begin

fix A B
assume A: A **and** B: B

have A A B
apply (*find-goal* ‹*match conclusion in B* ⇒ ‹–››)
apply (*rule B*)
by (*rule A*)+

have $A \wedge A$ $A \wedge A$ B
apply (*find-goal* ‹*fails* ‹*simp*››) — find the first goal which cannot be simplified
apply (*rule B*)
by (*simp add*: *A*)+

have B A $A \wedge A$

apply (*find-goal ‹succeeds ‹simp››*) — find the first goal which can be simplified (without doing so)
apply (*rule conjI*)
by (*rule A B*)+

fix *x*::*'a* **and** *S* :: *nat* ⇒ *'a* **and** *T R*
have
x = *T* ⟹ *A*
x = *S 10* ⟹ *B*
x = *R* ⟹ *B*
apply −
apply (*find-case S ?n*)
apply (*fails ‹in-case R›*)
apply (*in-case S ?n*)
by (*rule A B*)+

end

7.4.2 Remove redundant subgoals

Tries to solve subgoals by assuming the others and then using the given method. Backtracks over all possible re-orderings of the subgoals.

context begin

definition *protect* (*PROP P*) ≡ *P*

lemma *protectE*: *PROP protect P* ⟹ (*PROP P* ⟹ *PROP R*) ⟹ *PROP R* **by** (*simp add*: *protect-def*)

private lemmas *protect-thin* = *thin-rl*[**where** *V*=*PROP protect P* **for** *P*]

private lemma *context-conjunction'I-protected*:
assumes *P*: *PROP P*
assumes *PQ*: *PROP protect* (*PROP P*) ⟹ *PROP Q*
shows
PROP P &^& *PROP Q*
apply (*simp add*: *conjunction'-def*)
apply (*rule P*)
apply (*rule PQ*)
apply (*simp add*: *protect-def*)
by (*rule P*)

private lemma *conjunction'-sym*: *PROP P* &^& *PROP Q* ⟹ *PROP Q* &^& *PROP P*
apply (*simp add*: *conjunction'-def*)
apply (*frule conjunctionD1*)
apply (*drule conjunctionD2*)
apply (*rule conjunctionI*)
by *assumption*+

```
private lemmas context-conjuncts'I =
  context-conjunction'I-protected
  context-conjunction'I-protected[THEN conjunction'-sym]

method distinct-subgoals-strong methods m =
  (safe-fold-subgoals,
   (intro context-conjuncts'I;
     (((elim protectE conjunction'E)?, solves ‹m›)
     | (elim protect-thin)?)))?

end

method forward-solve methods fwd m =
  (fwd, prefer-last, fold-subgoals, safe-meta-conjuncts, rule conjunction'I,
   defer-tac, ((intro conjunction'I)?; solves ‹m›))[1]

method frule-solve methods m uses rule = (forward-solve ‹frule rule› ‹m›)
method drule-solve methods m uses rule = (forward-solve ‹drule rule› ‹m›)

notepad begin
  {
  fix A B C D E
  assume ABCD: A ⟹ B ⟹ C ⟹ D
  assume ACD: A ⟹ C ⟹ D
  assume DE: D ⟹ E
  assume B C

  have A ⟹ D
  apply (frule-solve ‹simp add: ‹B› ‹C››  rule: ABCD)
  apply (drule-solve ‹simp add: ‹B› ‹C››  rule: ACD)
  apply (match premises in A ⇒ ‹fail› | - ⇒ ‹–›)
  apply assumption
  done
  }
end


notepad begin
  {
  fix A B C
  assume A: A
  have A B ⟹ A
  apply –
  apply (distinct-subgoals-strong ‹assumption›)
  by (rule A)
```

have $B \Longrightarrow A\ A$
by (*distinct-subgoals-strong* ‹*assumption*›, *rule A*) — backtracking required here
}

{
fix *A B C*

assume *B*: *B*
assume *BC*: $B \Longrightarrow C\ B \Longrightarrow A$
have $A\ B \longrightarrow (A \wedge C)\ B$
apply (*distinct-subgoals-strong* ‹*simp*›, *rule B*) — backtracking required here
by (*simp add*: *BC*)

}
end

7.5 Attribute methods (for use with *rule-by-method* attributes)

method *prove-prop-raw* **for** *P* :: *prop* **methods** *m* =
(*erule thin-rl*, *rule revcut-rl*[*of PROP P*],
solves ‹*match conclusion in - ⇒* ‹*m*››)

method *prove-prop* **for** *P* :: *prop* = (*prove-prop-raw PROP P* ‹*auto*›)

experiment begin

lemma assumes *A*[*simp*]:*A* **shows** *A* **by** (*rule* [[@‹*prove-prop A*›]])

end

7.6 Shortcuts for *prove-prop*.

Note these are less efficient than using the raw syntax because the facts are re-proven every time.

method *ruleP* **for** *P* :: *prop* = (*catch* ‹*rule* [[@‹*prove-prop PROP P*›]]› ‹*fail*›)
method *insertP* **for** *P* :: *prop* = (*catch* ‹*insert* [[@‹*prove-prop PROP P*›]]› ‹*fail*›)[*1*]

experiment begin

lemma assumes *A*[*simp*]:*A* **shows** *A* **by** (*ruleP False* | *ruleP A*)
lemma assumes *A*:*A* **shows** *A* **by** (*ruleP* $\bigwedge P.\ P \Longrightarrow P \Longrightarrow P$, *rule A*, *rule A*)

end

context begin

private definition *bool-protect* (*b*::*bool*) ≡ *b*

lemma *bool-protectD*:
bool-protect P ⟹ *P*
unfolding *bool-protect-def* **by** *simp*

lemma *bool-protectI*:
P ⟹ *bool-protect P*
unfolding *bool-protect-def* **by** *simp*

When you want to apply a rule/tactic to transform a potentially complex goal into another one manually, but want to indicate that any fresh emerging goals are solved by a more brutal method. E.g. *apply* (*solves-emerging frule x=... in my-rulefastforce simp*: ... *intro*!: ...

method *solves-emerging* **methods** *m1 m2* = (*rule bool-protectD*, (*m1* ; (*rule bool-protectI* | (*m2*; *fail*))))

end

end

theory *Less-Monad-Syntax*
imports *HOL−Library.Monad-Syntax*
begin

no-syntax
-thenM :: ['*a*, '*b*] ⇒ '*c* (**infixr** ‹>>› *54*)

no-notation
Monad-Syntax.bind (**infixr** ‹>>=› *54*)

notation (**output**)
bind-do (**infixl** ‹>>=› *54*)

translations
CONST bind-do <= *CONST bind*

end

Chapter 8

Option Monad (State Reader)

theory *Reader-Monad*
imports
More-Lib
Less-Monad-Syntax
begin

type-synonym $('s,'a)$ *lookup* $= 's \Rightarrow 'a$ *option*

Similar to *map-option* but the second function returns option as well

definition
opt-map :: $('s,'a)$ *lookup* $\Rightarrow$ $('a \Rightarrow 'b$ *option*$) \Rightarrow ('s,'b)$ *lookup* (**infixl** ‹|>› *54*)
where
f |> $g \equiv \lambda s.$ *case* f s *of None* $\Rightarrow$ *None* | *Some* $x \Rightarrow g\ x$

abbreviation *opt-map-Some* :: $('s \rightharpoonup 'a) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 's \rightharpoonup 'b$ (**infixl** ‹||>› *54*)
where
f ||> $g \equiv f$ |> (*Some* $\circ$ g)
lemmas *opt-map-Some-def* = *opt-map-def*

lemma *opt-map-cong* [*fundef-cong*]:
$[\![f = f'; \bigwedge v\ s.\ f\ s = \textit{Some}\ v \Longrightarrow g\ v = g'\ v]\!] \Longrightarrow f$ |> $g = f'$ |> g'
by (*rule ext*) (*simp add*: *opt-map-def split*: *option.splits*)

lemma *in-opt-map-eq*:
((*f* |> *g*) *s* = *Some v*) = ($\exists\, v'.$ *f s* = *Some v'* $\wedge$ *g v'* = *Some v*)
by (*simp add*: *opt-map-def split*: *option.splits*)

lemma *opt-mapE*:
$[\![$ (*f* |> *g*) *s* = *Some v*; $\bigwedge v'.\ [\![f\ s = \textit{Some}\ v'; g\ v' = \textit{Some}\ v\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*auto simp*: *in-opt-map-eq*)

lemma *opt-map-upd-None*:

$f(x := None) \mid> g = (f \mid> g)(x := None)$
by (*auto simp*: *opt-map-def*)

lemma *opt-map-upd-Some*:
$f(x \mapsto v) \mid> g = (f \mid> g)(x := g\ v)$
by (*auto simp*: *opt-map-def*)

lemmas *opt-map-upd*[*simp*] = *opt-map-upd-None opt-map-upd-Some*

declare *None-upd-eq*[*simp*]

lemma ⟦ $(f \mid> g)\ x = None$; $g\ v = None$ ⟧ ⟹ $f(x \mapsto v) \mid> g = f \mid> g$
by *simp*

definition
obind :: $('s,'a)$ *lookup* ⇒ $('a \Rightarrow ('s,'b)$ *lookup*$)$ ⇒ $('s,'b)$ *lookup* (**infixl** ‹|>>› *53*)
where
$f \mid>> g \equiv \lambda s.$ *case* $f\ s$ *of None* ⇒ *None* | *Some* $x \Rightarrow g\ x\ s$

adhoc-overloading
Monad-Syntax.bind ⇌ *obind*

definition
ofail = *K None*

definition
oreturn = *K o Some*

definition
oassert $P \equiv$ *if* P *then oreturn* () *else ofail*

definition *oapply* :: $'a \Rightarrow ('a \Rightarrow 'b$ *option*$) \Rightarrow 'b$ *option*
where
oapply $x \equiv \lambda s.\ s\ x$

If the result can be an exception. Corresponding bindE would be analogous to lifting in NonDetMonad.

definition
oreturnOk x = *K* (*Some* (*Inr* x))

definition
othrow e = *K* (*Some* (*Inl* e))

definition
oguard $G \equiv (\lambda s.$ *if* $G\ s$ *then Some* () *else None*)

definition

ocondition c L R ≡ (λs. if c s then L s else R s)

definition
oskip ≡ oreturn ()

Monad laws

lemma *oreturn-bind* [*simp*]: *(oreturn x |>> f) = f x*
by (*auto simp add*: *oreturn-def obind-def K-def*)

lemma *obind-return* [*simp*]: *(m |>> oreturn) = m*
by (*auto simp add*: *oreturn-def obind-def K-def split*: *option.splits*)

lemma *obind-assoc*:
(m |>> f) |>> g = m |>> (λx. f x |>> g)
by (*auto simp add*: *oreturn-def obind-def K-def split*: *option.splits*)

Binding fail

lemma *obind-fail* [*simp*]:
f |>> (λ-. ofail) = ofail
by (*auto simp add*: *ofail-def obind-def K-def split*: *option.splits*)

lemma *ofail-bind* [*simp*]:
ofail |>> m = ofail
by (*auto simp add*: *ofail-def obind-def K-def split*: *option.splits*)

Function package setup

lemma *opt-bind-cong* [*fundef-cong*]:
⟦ f = f'; ⋀v s. f' s = Some v ⟹ g v s = g' v s ⟧ ⟹ f |>> g = f' |>> g'
by (*rule ext*) (*simp add*: *obind-def split*: *option.splits*)

lemma *opt-bind-cong-apply* [*fundef-cong*]:
⟦ f s = f' s; ⋀v. f' s = Some v ⟹ g v s = g' v s ⟧ ⟹ (f |>> g) s = (f' |>> g') s
by (*simp add*: *obind-def split*: *option.splits*)

lemma *oassert-bind-cong* [*fundef-cong*]:
⟦ P = P'; P' ⟹ m = m' ⟧ ⟹ oassert P |>> m = oassert P' |>> m'
by (*auto simp*: *oassert-def*)

lemma *oassert-bind-cong-apply* [*fundef-cong*]:
⟦ P = P'; P' ⟹ m () s = m' () s ⟧ ⟹ (oassert P |>> m) s = (oassert P' |>> m') s
by (*auto simp*: *oassert-def*)

lemma *oreturn-bind-cong* [*fundef-cong*]:
⟦ x = x'; m x' = m' x' ⟧ ⟹ oreturn x |>> m = oreturn x' |>> m'
by *simp*

lemma *oreturn-bind-cong-apply* [*fundef-cong*]:

⟦ $x = x'$; $m\ x'\ s = m'\ x'\ s$ ⟧ ⟹ $(oreturn\ x\ |>>\ m)\ s = (oreturn\ x'\ |>>\ m')\ s$
by *simp*

lemma *oreturn-bind-cong2* [*fundef-cong*]:
⟦ $x = x'$; $m\ x' = m'\ x'$ ⟧ ⟹ $(oreturn\ \$\ x)\ |>>\ m = (oreturn\ \$\ x')\ |>>\ m'$
by *simp*

lemma *oreturn-bind-cong2-apply* [*fundef-cong*]:
⟦ $x = x'$; $m\ x'\ s = m'\ x'\ s$ ⟧ ⟹ $((oreturn\ \$\ x)\ |>>\ m)\ s = ((oreturn\ \$\ x')\ |>>\ m')\ s$
by *simp*

lemma *ocondition-cong* [*fundef-cong*]:
⟦$c = c'$; ⋀s. $c'\ s \Longrightarrow l\ s = l'\ s$; ⋀$s$. $\neg c'\ s \Longrightarrow r\ s = r'\ s$⟧
⟹ $ocondition\ c\ l\ r = ocondition\ c'\ l'\ r'$
by (*auto simp*: *ocondition-def*)

Decomposition

lemma *ocondition-K-true* [*simp*]:
$ocondition\ (\lambda\text{-}.\ True)\ T\ F = T$
by (*simp add*: *ocondition-def*)

lemma *ocondition-K-false* [*simp*]:
$ocondition\ (\lambda\text{-}.\ False)\ T\ F = F$
by (*simp add*: *ocondition-def*)

lemma *ocondition-False*:
⟦ ⋀s. ¬ $P\ s$ ⟧ ⟹ $ocondition\ P\ L\ R = R$
by (*rule ext*, *clarsimp simp*: *ocondition-def*)

lemma *ocondition-True*:
⟦ ⋀s. $P\ s$ ⟧ ⟹ $ocondition\ P\ L\ R = L$
by (*rule ext*, *clarsimp simp*: *ocondition-def*)

lemma *in-oreturn* [*simp*]:
$(oreturn\ x\ s = Some\ v) = (v = x)$
by (*auto simp*: *oreturn-def K-def*)

lemma *oreturnE*:
⟦$oreturn\ x\ s = Some\ v$; $v = x \Longrightarrow P$⟧ ⟹ P
by *simp*

lemma *in-ofail* [*simp*]:
$ofail\ s \neq Some\ v$
by (*auto simp*: *ofail-def K-def*)

lemma *ofailE*:
$ofail\ s = Some\ v \Longrightarrow P$
by *simp*

lemma *in-oassert-eq* [*simp*]:
(*oassert P s = Some v*) = *P*
by (*simp add*: *oassert-def*)

lemma *oassert-True* [*simp*]:
oassert True = oreturn ()
by (*simp add*: *oassert-def*)

lemma *oassert-False* [*simp*]:
oassert False = ofail
by (*simp add*: *oassert-def*)

lemma *oassertE*:
⟦ *oassert P s = Some v*; *P* ⟹ *Q* ⟧ ⟹ *Q*
by *simp*

lemma *in-obind-eq*:
((*f* |>> *g*) *s = Some v*) = (∃ *v'*. *f s = Some v'* ∧ *g v' s = Some v*)
by (*simp add*: *obind-def split*: *option.splits*)

lemma *obind-eqI*:
⟦ *f s = f s'* ; ⋀*x*. *f s = Some x* ⟹ *g x s = g' x s'* ⟧ ⟹ *obind f g s = obind f g' s'*
by (*simp add*: *obind-def split*: *option.splits*)

lemma *obind-eqI-full*:
⟦ *f s = f s'* ; ⋀*x*. ⟦ *f s = Some x*; *f s' = f s* ⟧ ⟹ *g x s = g' x s'* ⟧
⟹ *obind f g s = obind f g' s'*
by (*drule sym*[**where** *s=f s*])
(*clarsimp simp*: *obind-def split*: *option.splits*)

lemma *obindE*:
⟦ (*f* |>> *g*) *s = Some v*;
⋀*v'*. ⟦*f s = Some v'*; *g v' s = Some v*⟧ ⟹ *P*⟧ ⟹ *P*
by (*auto simp*: *in-obind-eq*)

lemma *in-othrow-eq* [*simp*]:
(*othrow e s = Some v*) = (*v = Inl e*)
by (*auto simp*: *othrow-def K-def*)

lemma *othrowE*:
⟦*othrow e s = Some v*; *v = Inl e* ⟹ *P*⟧ ⟹ *P*
by *simp*

lemma *in-oreturnOk-eq* [*simp*]:
(*oreturnOk x s = Some v*) = (*v = Inr x*)
by (*auto simp*: *oreturnOk-def K-def*)

lemma *oreturnOkE*:
⟦*oreturnOk x s = Some v*; *v = Inr x ⟹ P*⟧ *⟹ P*
by *simp*

lemmas *omonadE* [*elim*!] =
opt-mapE obindE oreturnE ofailE othrowE oreturnOkE oassertE

lemma *in-opt-map-Some-eq*:
((*f* ||> *g*) *x = Some y*) = (∃ *v*. *f x = Some v* ∧ *g v = y*)
by (*simp add*: *in-opt-map-eq*)

lemma *in-opt-map-None-eq*[*simp*]:
((*f* ||> *g*) *x = None*) = (*f x = None*)
by (*simp add*: *opt-map-def split*: *option.splits*)

lemma *oreturn-comp*[*simp*]:
oreturn x ∘ *f = oreturn x*
by (*simp add*: *oreturn-def K-def o-def*)

lemma *ofail-comp*[*simp*]:
ofail ∘ *f = ofail*
by (*auto simp*: *ofail-def K-def*)

lemma *oassert-comp*[*simp*]:
oassert P ∘ *f = oassert P*
by (*simp add*: *oassert-def*)

lemma *fail-apply*[*simp*]:
ofail s = None
by (*simp add*: *ofail-def K-def*)

lemma *oassert-apply*[*simp*]:
oassert P s = (*if P then Some* () *else None*)
by (*simp add*: *oassert-def*)

lemma *oreturn-apply*[*simp*]:
oreturn x s = Some x
by *simp*

lemma *oapply-apply*[*simp*]:
oapply x s = s x
by (*simp add*: *oapply-def*)

lemma *obind-comp-dist*:
obind f g o h = obind (*f o h*) (λ*x*. *g x o h*)
by (*auto simp*: *obind-def split*: *option.splits*)

lemma *if-comp-dist*:

(*if P then f else g*) *o h* = (*if P then f o h else g o h*)
by *auto*

8.1 "While" loops over option monad.

This is an inductive definition of a while loop over the plain option monad (without passing through a state)

inductive-set
option-while' :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ '*a option*) ⇒ '*a option rel*
for *C B*
where
final: ¬ *C r* ⟹ (*Some r*, *Some r*) ∈ *option-while' C B*
| *fail*: ⟦ *C r*; *B r* = *None* ⟧ ⟹ (*Some r*, *None*) ∈ *option-while' C B*
| *step*: ⟦ *C r*; *B r* = *Some r'*; (*Some r'*, *sr''*) ∈ *option-while' C B* ⟧
⟹ (*Some r*, *sr''*) ∈ *option-while' C B*

definition
option-while C B r ≡
(*if* (∃ *s*. (*Some r*, *s*) ∈ *option-while' C B*) *then*
(*THE s*. (*Some r*, *s*) ∈ *option-while' C B*) *else None*)

lemma *option-while'-inj*:
assumes (*s*,*s'*) ∈ *option-while' C B* (*s*, *s''*) ∈ *option-while' C B*
shows *s'* = *s''*
using *assms* **by** (*induct rule*: *option-while'.induct*) (*auto elim*: *option-while'.cases*)

lemma *option-while'-inj-step*:
⟦ *C s*; *B s* = *Some s'*; (*Some s*, *t*) ∈ *option-while' C B* ; (*Some s'*, *t'*) ∈ *option-while' C B* ⟧ ⟹ *t* = *t'*
by (*metis option-while'.step option-while'-inj*)

lemma *option-while'-THE*:
assumes (*Some r*, *sr'*) ∈ *option-while' C B*
shows (*THE s*. (*Some r*, *s*) ∈ *option-while' C B*) = *sr'*
using *assms* **by** (*blast dest*: *option-while'-inj*)

lemma *option-while-simps*:
¬ *C s* ⟹ *option-while C B s* = *Some s*
C s ⟹ *B s* = *None* ⟹ *option-while C B s* = *None*
C s ⟹ *B s* = *Some s'* ⟹ *option-while C B s* = *option-while C B s'*
(*Some s*, *ss'*) ∈ *option-while' C B* ⟹ *option-while C B s* = *ss'*
using *option-while'-inj-step*[*of C s B s'*]
by (*auto simp*: *option-while-def option-while'-THE*
intro: *option-while'.intros*
dest: *option-while'-inj*
elim: *option-while'.cases*)

lemma *option-while-rule*:

assumes *option-while C B s = Some s'*
assumes *I s*
assumes *istep*: $\bigwedge s\ s'.\ C\ s \Longrightarrow I\ s \Longrightarrow B\ s = Some\ s' \Longrightarrow I\ s'$
shows $I\ s' \wedge \neg\ C\ s'$
proof −
{ fix *ss ss'* **assume** $(ss, ss') \in$ *option-while' C B ss = Some s ss' = Some s'*
then have *?thesis* **using** ‹*I s*›
by (*induct arbitrary*: *s*) (*auto intro*: *istep*) **}**
then show *?thesis* **using** *assms*(*1*)
by (*auto simp*: *option-while-def option-while'-THE split*: *if-split-asm*)
qed

lemma *option-while'-term*:
assumes *I r*
assumes *wf M*
assumes *step-less*: $\bigwedge r\ r'.\ [\![I\ r;\ C\ r;\ B\ r = Some\ r']\!] \Longrightarrow (r', r) \in M$
assumes *step-I*: $\bigwedge r\ r'.\ [\![I\ r;\ C\ r;\ B\ r = Some\ r']\!] \Longrightarrow I\ r'$
obtains *sr'* **where** $(Some\ r,\ sr') \in$ *option-while' C B*
apply *atomize-elim*
using *assms*(*2*,*1*)
proof *induct*
case (*less r*)
show *?case*
proof (*cases C r B r rule*: *bool.exhaust*[*case-product option.exhaust*])
case (*True-Some r'*)
then have $(r', r) \in M\ I\ r'$
by (*auto intro*: *less step-less step-I*)
then obtain *sr'* **where** $(Some\ r',\ sr') \in$ *option-while' C B*
by *atomize-elim* (*rule less*)
then have $(Some\ r,\ sr') \in$ *option-while' C B*
using *True-Some* **by** (*auto intro*: *option-while'.intros*)
then show *?thesis* **..**
qed (*auto intro*: *option-while'.intros*)
qed

lemma *option-while-rule'*:
assumes *option-while C B s = ss'*
assumes *wf M*
assumes *I* (*Some s*)
assumes *less*: $\bigwedge s\ s'.\ C\ s \Longrightarrow I\ (Some\ s) \Longrightarrow B\ s = Some\ s' \Longrightarrow (s',\ s) \in M$
assumes *step*: $\bigwedge s\ s'.\ C\ s \Longrightarrow I\ (Some\ s) \Longrightarrow B\ s = Some\ s' \Longrightarrow I\ (Some\ s')$
assumes *final*: $\bigwedge s.\ C\ s \Longrightarrow I\ (Some\ s) \Longrightarrow B\ s = None \Longrightarrow I\ None$
shows $I\ ss' \wedge$ (*case ss' of Some s' ⇒ ¬ C s' | - ⇒ True*)
proof −
define *ss* **where** *ss* ≡ *Some s*
obtain *ss1'* **where** $(Some\ s,\ ss1') \in$ *option-while' C B*
using *assms*(*3*,*2*,*4*,*5*) **by** (*rule option-while'-term*)
then have ∗: $(ss,\ ss') \in$ *option-while' C B* **using** ‹*option-while C B s = ss'*›
by (*auto simp*: *option-while-simps ss-def*)

show *?thesis*
proof (*cases ss'*)
case (*Some s'*) **with** ∗ *ss-def* **show** *?thesis* **using** ‹*I* -›
by (*induct arbitrary*:*s*) (*auto intro*: *step*)
next
case *None* **with** ∗ *ss-def* **show** *?thesis* **using** ‹*I* -›
by (*induct arbitrary*:*s*) (*auto intro*: *step final*)
qed
qed

8.2 Lift *option-while* to the (*'a*, *'s*) *lookup* monad

definition
owhile :: (*'a* ⇒ *'s* ⇒ *bool*) ⇒ (*'a* ⇒ (*'s*,*'a*) *lookup*) ⇒ *'a* ⇒ (*'s*,*'a*) *lookup*
where
owhile c b a ≡ λ*s*. *option-while* (λ*a*. *c a s*) (λ*a*. *b a s*) *a*

lemma *owhile-unroll*:
owhile C B r = *ocondition* (*C r*) (*B r* |>> *owhile C B*) (*oreturn r*)
by (*auto simp*: *ocondition-def obind-def oreturn-def owhile-def*
option-while-simps K-def split: *option.split*)

rule for terminating loops

lemma *owhile-rule*:
assumes *I r s*
assumes *wf M*
assumes *less*: ⋀*r r'*. ⟦*I r s*; *C r s*; *B r s* = *Some r'*⟧ ⟹ (*r'*,*r*) ∈ *M*
assumes *step*: ⋀*r r'*. ⟦*I r s*; *C r s*; *B r s* = *Some r'*⟧ ⟹ *I r' s*
assumes *fail*: ⋀*r*. ⟦*I r s*; *C r s*; *B r s* = *None*⟧ ⟹ *Q None*
assumes *final*: ⋀*r*. ⟦*I r s*; ¬*C r s*⟧ ⟹ *Q* (*Some r*)
shows *Q* (*owhile C B r s*)
proof −
let *?rs'* = *owhile C B r s*
have (*case ?rs' of Some r* ⇒ *I r s* | - ⇒ *Q None*)
∧ (*case ?rs' of Some r'* ⇒ ¬ *C r' s* | - ⇒ *True*)
by (*rule option-while-rule'*[**where** *B*=λ*r*. *B r s* **and** *s*=*r*, *OF* - ‹*wf* -›])
(*auto simp*: *owhile-def intro*: *assms*)
then show *?thesis* **by** (*auto intro*: *final split*: *option.split-asm*)
qed

end

theory *Option-MonadND*
imports
Reader-Monad
begin

definition
ogets :: (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b option*)
where
ogets f ≡ (λ*s*. *Some* (*f s*))

definition
ocatch :: (*'s*,(*'e* + *'a*)) *lookup* ⇒ (*'e* ⇒ (*'s*,*'a*) *lookup*) ⇒ (*'s*, *'a*) *lookup*
(**infix** ‹<*ocatch*>› *10*)
where
f <*ocatch*> *handler* ≡ *do* {
x ← *f*;
case x of Inr b ⇒ *oreturn b* | *Inl e* ⇒ *handler e*
}

definition
odrop :: (*'s*, *'e* + *'a*) *lookup* ⇒ (*'s*, *'a*) *lookup*
where
odrop f ≡ *do* {
x ← *f*;
case x of Inr b ⇒ *oreturn b* | *Inl e* ⇒ *ofail*
}

definition
osequence-x :: (*'s*, *'a*) *lookup list* ⇒ (*'s*, *unit*) *lookup*
where
osequence-x xs ≡ *foldr* (λ*x y*. *do* { *x*; *y* }) *xs* (*oreturn* ())

definition
osequence :: (*'s*, *'a*) *lookup list* ⇒ (*'s*, *'a list*) *lookup*
where
osequence xs ≡ *let mcons* = (λ*p q*. *p* |>> (λ*x*. *q* |>> (λ*y*. *oreturn* (*x*#*y*))))
in foldr mcons xs (*oreturn* [])

definition
omap :: (*'a* ⇒ (*'s*,*'b*) *lookup*) ⇒ *'a list* ⇒ (*'s*, *'b list*) *lookup*
where
omap f xs ≡ *osequence* (*map f xs*)

definition
opt-cons :: *'a option* ⇒ *'a list* ⇒ *'a list* (**infixr** ‹*o*#› *65*)
where
opt-cons x xs ≡ *case x of None* ⇒ *xs* | *Some x'* ⇒ *x'* # *xs*

end

theory *Apply-Trace*
imports
Main
begin

ML-file ‹*ml−helpers/ThmExtras.ML*›

ML ‹

```
signature APPLY-TRACE =
sig
  val apply-results :
    {silent-fail : bool} ->
    (Proof.context -> thm -> ((string * int option) * term) list -> unit) ->
    Method.text-range -> Proof.state -> Proof.state Seq.result Seq.seq

  (* Lower level interface. *)
  val can-clear : theory -> bool
  val clear-deps : thm -> thm
  val join-deps : thm -> thm -> thm
  val used-facts : Proof.context -> thm -> ((string * int option) * term) list
  val pretty-deps: bool -> (string * Position.T) option -> Proof.context -> thm
->
    ((string * int option) * term) list -> Pretty.T
end

structure Apply-Trace : APPLY-TRACE =
struct

(*TODO: Add more robust oracle without hyp clearing *)
fun thm-to-cterm keep-hyps thm =
let

  val thy = Thm.theory-of-thm thm
  val pairs = Thm.tpairs-of thm
  val ceqs = map (Thm.global-cterm-of thy o Logic.mk-equals) pairs
  val hyps = Thm.chyps-of thm
  val prop = Thm.cprop-of thm
  val thm' = if keep-hyps then Drule.list-implies (hyps,prop) else prop

in
  Drule.list-implies (ceqs,thm') end


val (-, clear-thm-deps') =
  Context.>>> (Context.map-theory-result (Thm.add-oracle (binding ‹count-cheat›,
thm-to-cterm false)));
```

```
fun clear-deps thm =
let

  val thm' = try clear-thm-deps' thm
 |> Option.map (fold (fn - => fn t => (@{thm Pure.reflexive} RS t)) (Thm.tpairs-of
thm))

in case thm' of SOME thm' => thm' | NONE => error Can't clear deps here end


fun can-clear thy = Context.subthy(@{theory},thy)

fun join-deps pre-thm post-thm =
let
  val pre-thm' = Thm.flexflex-rule NONE pre-thm |> Seq.hd
    |> Thm.adjust-maxidx-thm (Thm.maxidx-of post-thm + 1)
in
  Conjunction.intr pre-thm' post-thm |> Conjunction.elim |> snd
end

fun get-ref-from-nm' nm =
let
  val exploded = space-explode - nm;
  val base = List.take (exploded, (length exploded) - 1) |> space-implode -
  val idx = List.last exploded |> Int.fromString;
in if is-some idx andalso base <>  then SOME (base, the idx) else NONE end

fun get-ref-from-nm nm = Option.join (try get-ref-from-nm' nm);

fun maybe-nth l = try (curry List.nth l)

fun fact-from-derivation ctxt xnm =
let

  val facts = Proof-Context.facts-of ctxt;
  (* TODO: Check that exported local fact is equivalent to external one *)

  val idx-result =
    let
      val (name', idx) = get-ref-from-nm xnm |> the;
        val entry = try (Facts.retrieve (Context.Proof ctxt) facts) (name', Posi-
tion.none) |> the;
      val thm = maybe-nth (#thms entry) (idx - 1) |> the;
    in SOME (xnm, thm) end handle Option => NONE;

  fun non-idx-result () =
    let
     val entry = try (Facts.retrieve (Context.Proof ctxt) facts) (xnm, Position.none)
```

```
|> the;
      val thm = try the-single (#thms entry) |> the;
    in SOME (#name entry, thm) end handle Option => NONE;

in
  case idx-result of
    SOME thm => SOME thm
  | NONE => non-idx-result ()
end

fun most-local-fact-of ctxt xnm =
let
  val local-name = try (fn xnm => Long-Name.explode xnm |> tl |> tl |> Long-Name.implode)
xnm |> the;
in SOME (fact-from-derivation ctxt local-name |> the) end handle Option =>
  fact-from-derivation ctxt xnm;

fun thms-of (PBody {thms,...}) = thms

fun proof-body-descend' f get-fact (ident, thm-node) deptab = let
  val nm = Thm-Name.short (Proofterm.thm-node-name thm-node)
  val body = Proofterm.thm-node-body thm-node
in
  (if not (f nm) then
     (Inttab.update-new (ident, SOME (nm, get-fact nm |> the)) deptab handle
Inttab.DUP - => deptab)
  else raise Option) handle Option =>
    ((fold (proof-body-descend' f get-fact) (thms-of (Future.join body))
      (Inttab.update-new (ident, NONE) deptab)) handle Inttab.DUP - => deptab)
end

fun used-facts' f get-fact thm =
  let
    val body = thms-of (Thm.proof-body-of thm);

  in fold (proof-body-descend' f get-fact) body Inttab.empty end

fun used-pbody-facts ctxt thm =
  let
    val nm = Thm-Name.short (Thm.get-name-hint thm);
    val get-fact = most-local-fact-of ctxt;
  in
    used-facts' (fn nm' => nm' =  orelse nm' = nm) get-fact thm
    |> Inttab.dest |> map-filter snd |> map snd |> map (apsnd (Thm.prop-of))
  end

fun raw-primitive-text f = Method.Basic (fn - => ((K (fn (ctxt, thm) => Seq.make-results
(Seq.single (ctxt, f thm))))))
```

```
(*Find local facts from new hyps*)
fun used-local-facts ctxt thm =
let
  val hyps = Thm.hyps-of thm
  val facts = Proof-Context.facts-of ctxt |> Facts.dest-static true []

  fun match-hyp hyp =
  let
    fun get (nm,thms) =
      case (get-index (fn t => if (Thm.prop-of t) aconv hyp then SOME hyp else
NONE) thms)
       of SOME t => SOME (nm,t)
        | NONE => NONE


  in
    get-first get facts
  end

in
  map-filter match-hyp hyps end

fun used-facts ctxt thm =
  let
   val used-from-pbody = used-pbody-facts ctxt thm |> map (fn (nm,t) => ((nm,NONE),t))
      val used-from-hyps = used-local-facts ctxt thm |> map (fn (nm,(i,t)) =>
((nm,SOME i),t))
  in
    (used-from-hyps @ used-from-pbody)
  end

(* Perform refinement step, and run the given stateful function
   against computed dependencies afterwards. *)
fun refine args f text state =
let

  val ctxt = Proof.context-of state

  val thm = Proof.simple-goal state |> #goal

  fun save-deps deps = f ctxt thm deps


in
  if (can-clear (Proof.theory-of state)) then
     Proof.refine (Method.Combinator (Method.no-combinator-info,Method.Then,
[raw-primitive-text (clear-deps),text,
     raw-primitive-text (fn thm' => (save-deps (used-facts ctxt thm');join-deps thm
```

```
thm'))])) state
  else
     (if (#silent-fail args) then (save-deps [];Proof.refine text state) else error
Apply-Trace theory must be imported to trace applies)
end

(* Boilerplate from Proof.ML *)


fun method-error kind pos state =
  Seq.single (Proof-Display.method-error kind pos (Proof.raw-goal state));

fun apply args f text = Proof.assert-backward #> refine args f text #>
 Seq.maps-results (Proof.apply ((raw-primitive-text I),(Position.none, Position.none)));

fun apply-results args f (text, range) =
  Seq.APPEND (apply args f text, method-error  (Position.range-position range));


structure Filter-Thms = Named-Thms
(
  val name = @{binding no-trace}
  val description = thms to be ignored from tracing
)

(* Print out the found dependencies. *)
fun pretty-deps only-names query ctxt thm deps =
let
  (* Remove duplicates. *)
 val deps = sort-distinct (prod-ord (prod-ord string-ord (option-ord int-ord)) Term-Ord.term-ord)
deps

  (* Fetch canonical names and theorems. *)
   val deps = map (fn (ident, term) => ThmExtras.adjust-thm-name ctxt ident
term) deps

  (* Remove boring theorems. *)
   val deps = subtract (fn (a, ThmExtras.FoundName (-, thm)) => Thm.eq-thm
(thm, a)
                        | - => false) (Filter-Thms.get ctxt) deps

  val deps = case query of SOME (raw-query,pos) =>
    let
      val pos' = perhaps (try (Position.shift-offsets {remove-id = false} 1)) pos;
      val q = Find-Theorems.read-query pos' raw-query;
       val results = Find-Theorems.find-theorems-cmd ctxt (SOME thm) (SOME
1000000000) false q
                 |> snd
                 |> map ThmExtras.adjusted-thm-name;
```

```
    (* Only consider theorems from our query. *)

    val deps = inter (fn (ThmExtras.FoundName (nmidx,-), ThmExtras.FoundName
(nmidx',-)) => nmidx = nmidx'
                                | - => false) results deps
    in deps end
    | - => deps

in
  if only-names then
    Pretty.block
      (Pretty.separate  (map (ThmExtras.pretty-fact only-names ctxt) deps))
  else
  (* Pretty-print resulting theorems. *)
    Pretty.big-list used theorems:
      (map (Pretty.item o single o ThmExtras.pretty-fact only-names ctxt) deps)

end

val - = Context.>> (Context.map-theory Filter-Thms.setup)

end
›

end
```

```
theory Apply-Trace-Cmd
imports Apply-Trace
keywords apply-trace :: prf-script
begin

ML‹

val - =
  Outer-Syntax.command @{command-keyword apply-trace} initial refinement step
(unstructured)

  (Args.mode only-names -- (Scan.option (Parse.position Parse.cartouche)) --
Method.parse >>
    (fn ((on,query),text) => Toplevel.proofs (Apply-Trace.apply-results {silent-fail
= false}
     (Pretty.writeln ooo (Apply-Trace.pretty-deps on query)) text)));

›
```

lemmas [*no-trace*] = *protectI protectD TrueI Eq-TrueI eq-reflection*

lemma $(a \wedge b) = (b \wedge a)$
apply-trace *auto*
oops

lemma $(a \wedge b) = (b \wedge a)$
apply-trace ‹*intro*› *auto*
oops

lemma
assumes *X*: $b = a$
assumes *Y*: $b = a$
shows
$b = a$
apply-trace (*rule Y*)
oops

locale *Apply-Trace-foo* = **fixes** *b a*
assumes *X*: $b = a$
begin

lemma shows $b = a$ $b = a$
apply −
apply-trace (*rule Apply-Trace-foo.X*)
prefer *2*
apply-trace (*rule X*)
oops
end

experiment begin

Example of trace for grouped lemmas

definition *ex* :: *nat set* **where**
ex = {*1*,*2*,*3*,*4*}

lemma *v1*: *1* ∈ *ex* **by** (*simp add*: *ex-def*)
lemma *v2*: *2* ∈ *ex* **by** (*simp add*: *ex-def*)
lemma *v3*: *3* ∈ *ex* **by** (*simp add*: *ex-def*)

Group several lemmas in a single one

lemmas *vs* = *v1 v2 v3*

lemma *2* ∈ *ex*

apply-trace (*simp add*: *vs*)
oops

end
end

Chapter 9

Mutual CCPO Recursion
fixed-point

theory *Mutual-CCPO-Recursion*
  imports
    HOL−Library.Complete-Partial-Order2
    AutoCorres-Utils
  keywords *fixed-point* :: *thy-goal-stmt*
begin

9.1 Relate orders between locale and type classes

lemma *gfp-lub-fun*: *gfp.lub-fun* = *Inf*
  unfolding *fun-lub-def fun-eq-iff Inf-apply image-def* **by** *simp*

lemma *gfp-le-fun*: *gfp.le-fun* = (≥)
  unfolding *fun-ord-def le-fun-def fun-eq-iff Inf-apply image-def* **by** *simp*

lemma
  shows *fst-prod-lub*: *fst* (*prod-lub luba lubb S*) = *luba* (*fst* ‘ *S*)
    and *snd-prod-lub*: *snd* (*prod-lub luba lubb S*) = *lubb* (*snd* ‘ *S*)
  by (*auto simp*: *prod-lub-def*)

lemma *fun-lub-app*: *fun-lub lub S x* = *lub* ((λ*f*. *f x*) ‘ *S*)
  by (*auto simp*: *fun-lub-def image-def*)

lemma *ccpo-Inf*: *class.ccpo* (*Inf* :: *'a*::*complete-lattice set* ⇒ -) (≥) (*mk-less* (≥))
  by (*rule cont-intro*)

lemma *ccpo-Sup*: *class.ccpo* (*Sup* :: *'a*::*complete-lattice set* ⇒ -) (≤) (*mk-less* (≤))
  by (*rule cont-intro*)

lemma *ccpo-flat*: *class.ccpo* (*flat-lub b*) (*flat-ord b*) (*mk-less* (*flat-ord b*))
  using *Partial-Function.ccpo*[*OF flat-interpretation*] **.**

lemma *ccpo-ccpo-class'*: *class.ccpo* (*Sup* :: *'a::ccpo set* ⇒ -) (≤) (<)
proof qed

lemma *monotone-pair*[*partial-function-mono*]:
monotone R orda f ⟹ *monotone R ordb g* ⟹ *monotone R* (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
by (*simp add*: *monotone-def*)

lemma *monotone-fst'*[*partial-function-mono*]:
monotone orda (*rel-prod ordb ordc*) *f* ⟹ *monotone orda ordb* (λ*x*. *fst* (*f x*))
by (*auto simp add*: *monotone-def rel-prod-sel*)

lemma *monotone-snd'*[*partial-function-mono*]:
monotone orda (*rel-prod ordb ordc*) *f* ⟹ *monotone orda ordc* (λ*x*. *snd* (*f x*))
by (*auto simp add*: *monotone-def rel-prod-sel*)

lemma *monotone-id-fun-elim*[*partial-function-mono*]:
monotone (≥) *ord* (λ*x*. *x*) ⟹ *monotone* (≥) (*fun-ord ord*) (λ*x*. *x*)
by (*auto simp add*: *monotone-def le-fun-def fun-ord-def*)

lemma *monotone-id*[*partial-function-mono*]: *monotone ord ord* (λ*x*. *x*)
by (*auto simp add*: *monotone-def*)

lemma *monotone-abs*:
(⋀*x*. *monotone R Q* (λ*f*. *F f x*)) ⟹ *monotone R* (*fun-ord Q*) (λ*f x*. *F f x*)
by (*simp add*: *monotone-def fun-ord-def*)

lemma *monotone-fun-ord-applyD*:
monotone orda (*fun-ord ordb*) *f* ⟹ *monotone orda ordb* (λ*y*. *f y x*)
by(*auto simp add*: *monotone-def fun-ord-def*)

lemma *admissible-snd*:
cont luba orda (*prod-lub lubb Inf*) (*rel-prod ordb* (≥)) *F* ⟹
ccpo.admissible luba orda (λ*x*. *snd* (*F x*))
unfolding *ccpo.admissible-def cont-def prod-eq-iff*
by (*auto simp*: *prod-lub-def*)

lemma *cont-apply-gfp*: *cont gfp.lub-fun gfp.le-fun Inf* (≤) (λ*x*. *x c*)
by (*rule cont-applyI*) (*simp add*: *cont-def*)

lemma *mcont-mem-gfp*:
mcont gfp.lub-fun gfp.le-fun (*Inf*) (≥) *F* ⟹ *gfp.admissible* (λ*A*. *x* ∈ *F A*)
by (*auto simp*: *mcont-def cont-def ccpo.admissible-def*)

lemma *mcont2mcont-call*:
mcont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *F* ⟹ *mcont luba orda lubb ordb* (λ*x*. *F x c*)
by (*auto simp*: *mcont-def monotone-def cont-def fun-lub-def fun-ord-def le-fun-def*

fun-eq-iff)
(*simp add*: *image-def*)

lemma *preorder-fun-ord* [*partial-function-mono*]: *class.preorder R* (*mk-less R*) ⟹ *class.preorder* (*fun-ord R*) (*mk-less* (*fun-ord R*))
by (*force simp*: *class.preorder-def fun-ord-def mk-less-def*)

lemma *preorder-monotone-const'*[*partial-function-mono*]:
class.preorder leq (*mk-less leq*) ⟹ *monotone ord leq* (λ-. *c*)
by (*rule preorder.monotone-const*)

declare *preorder-rel-prodI* [*partial-function-mono*]
declare *gfp.preorder* [*partial-function-mono*]
declare *lfp.preorder* [*partial-function-mono*]

lemma *option-ord-preorder* [*partial-function-mono*]: *class.preorder option-ord* (*mk-less option-ord*)
by *simp*

9.2 Prove admissibility of *corresXF*

lemma *not-imp-not-iff*: (¬*A* ⟶ ¬*B*) ⟷ (*B* ⟶ *A*)
by *auto*

lemma *mcont-fun-lub-call*:
mcont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* ⟹
mcont luba orda lubb ordb (λ*y*. *f y x*)
by (*simp add*: *mcont-fun-lub-apply*)

lemma *chain-disj-of-subsingleton*:
Complete-Partial-Order.chain ord {*r*. *P r*} ⟹
Complete-Partial-Order.chain ord {*r*. *Q r*} ⟹
(⋀*r q*. *P r* ⟹ *Q q* ⟹ *r* = *q*) ⟹
Complete-Partial-Order.chain ord {*r*. *P r* ∨ *Q r*}
by (*auto simp*: *chain-def*)

definition *subsingleton-set* :: *'a set* ⇒ *bool* **where**
subsingleton-set P ⟷ (∀ *a*∈*P*. ∀ *b*∈*P*. *a* = *b*)

lemma *subsingleton-set-empty*[*iff*]: *subsingleton-set* {}
by (*simp add*: *subsingleton-set-def*)

lemma *subsingleton-set-singleton*[*iff*]: *subsingleton-set* {*x*}
by (*simp add*: *subsingleton-set-def*)

context *ccpo*
begin

lemma *subsingleton-sets-imp-chain*:

subsingleton-set ($\bigcup$ *Ps*) $\Longrightarrow$ *Complete-Partial-Order.chain* ($\leq$) ($\bigcup$ *Ps*)
by (*auto simp*: *subsingleton-set-def Complete-Partial-Order.chain-def Ball-def*)

lemma *Sup-of-subsingleton-sets-eq*:
assumes *Ps*: *subsingleton-set* ($\bigcup$ *Ps*) **and** *P*: $P \in Ps$ **and** *a*: $a \in P$
shows *Sup* ($\bigcup$ *Ps*) = *a*
proof (*intro order.antisym ccpo-Sup-least ccpo-Sup-upper subsingleton-sets-imp-chain*[*OF Ps*])
fix *x* :: *'a* **assume** $x \in \bigcup$ *Ps*
then obtain *Q* **where** $Q \in Ps$ $x \in Q$ **by** *auto*
with *Ps P a* **show** $x \leq a$
by (*auto simp add*: *subsingleton-set-def Ball-def*)
qed (*use P a* **in** *blast*)

lemma *monotone-Sup-of-subsingleton-sets*:
assumes *Ps*: $\bigwedge F$. *subsingleton-set* ($\bigcup$ (*Ps F*))
and $*$: $\bigwedge F\ G$. *ord F G* $\Longrightarrow$ *sim-set* (*sim-set* ($\leq$)) (*Ps F*) (*Ps G*)
shows *monotone ord* ($\leq$) (λF. *Sup* ($\bigcup$ (*Ps F*)))
proof (*intro monotoneI ccpo-Sup-least subsingleton-sets-imp-chain*[*OF Ps*])
fix *F G x* **assume** *ord F G x* $\in \bigcup$ (*Ps F*)
with $*$[*of F G*] **obtain** *y* **where** $x \leq y$ $y \in \bigcup$ (*Ps G*)
by (*fastforce simp*: *sim-set-def*)
then show $x \leq$ *Sup* ($\bigcup$ (*Ps G*))
apply (*intro order-trans*[*OF* ‹$x \leq y$›])
apply (*intro ccpo-Sup-upper subsingleton-sets-imp-chain*[*OF Ps*])
apply *auto*
done
qed

lemma *ccpo-refl*: $x \leq x$ **..**

lemma *fixp-unfold-def*:
fixes *f* :: *'a* $\Rightarrow$ *'a*
assumes *def*: $F \equiv$ *ccpo.fixp Sup* ($\leq$) *f*
assumes *f*: *monotone* ($\leq$) ($\leq$) *f*
shows *F* = *f F*
by (*unfold def*) (*rule fixp-unfold*[*OF f*])

lemma *fixp-induct-def*:
fixes *f* :: *'a* $\Rightarrow$ *'a*
assumes *def*: $F \equiv$ *ccpo.fixp Sup* ($\leq$) *f*
assumes *mono*: *monotone* ($\leq$) ($\leq$) *f*
assumes *adm*: *ccpo.admissible Sup* ($\leq$) *P*
assumes *bot*: *P* (*Sup* {})
assumes *step*: $\bigwedge x$. *P x* $\Longrightarrow$ *P* (*f x*)
shows *P F*
by (*unfold def*) (*rule fixp-induct*[*OF adm mono bot step*])

lemma *induct-Sup-of-subsingleton-sets*:

assumes *Ps*: *subsingleton-set* ($\bigcup$ *Ps*)
and *adm*: *ccpo.admissible Sup* ($\leq$) *P*
and *bot*: *P* (*Sup* {})
and *step*: $\bigwedge p\ x.\ p \in Ps \Longrightarrow x \in p \Longrightarrow P\ x$
shows *P* (*Sup* ($\bigcup$ *Ps*))
proof *cases*
assume $*$: ($\bigcup$ *Ps*) = {} **from** *bot* **show** *?thesis* **unfolding** $*$.
next
assume $*$: ($\bigcup$ *Ps*) $\neq$ {}
show *?thesis*
by (*intro ccpo.admissibleD*[*OF adm subsingleton-sets-imp-chain*[*OF Ps*] $*$])
(*blast intro*: *step*)
qed

lemma *induct-Sup-of-subsingleton-sets-def*:
assumes *F*: *F* $\equiv$ *Sup* ($\bigcup$ *Ps*)
assumes *Ps*: *subsingleton-set* ($\bigcup$ *Ps*)
and *adm*: *ccpo.admissible Sup* ($\leq$) *P*
and *bot*: *P* (*Sup* {})
and *step*: $\bigwedge p\ x.\ p \in Ps \Longrightarrow x \in p \Longrightarrow P\ x$
shows *P F*
unfolding *F* **by** (*rule induct-Sup-of-subsingleton-sets*; *fact*)

end

lemma *flat-lub-empty*: *flat-lub x* {} = *x*
by (*simp add*: *flat-lub-def*)

lemma *monotone-bind-option*[*partial-function-mono*]:
monotone ord option-ord f $\Longrightarrow$ ($\bigwedge a.$ *monotone ord option-ord* ($\lambda x.$ *g x a*)) $\Longrightarrow$
monotone ord option-ord ($\lambda x.$ *Option.bind* (*f x*) (*g x*))
by (*fastforce simp*: *monotone-def flat-ord-def bind-eq-None-conv*)

lemma (**in** *ccpo*) *admissible-ord*: *ccpo.admissible Sup* ($\leq$) ($\lambda x.$ *x* $\leq$ *b*)
by (*clarsimp simp add*: *ccpo.admissible-def intro*!: *ccpo-Sup-least*)

named-theorems *fixed-point-cleanup-simps* ‹*Simp set for fixed-point*›

lemma *monotone-const*[*partial-function-mono*]:
monotone R ($\leq$) ($\lambda x.$ (*c*::'*a*::*order*))
by (*auto simp*: *monotone-def*)

lemma *Sup-empty-ccpo*[*simp*]:
Sup {} = (*bot*::'*a*::{*ccpo*, *order-bot*})
by (*intro le-bot ccpo-Sup-least chain-empty*) *simp*

lemmas [*fixed-point-cleanup-simps*] =
Inf-empty
Sup-empty

prod-lub-empty
fun-lub-empty
flat-lub-empty
Sup-empty-ccpo
fst-conv
snd-conv
split-paired-all
prod.collapse

ML-file ‹*mutual-ccpo-recursion.ML*›

setup ‹
(*Mutual-CCPO-Rec.add-ccpo option Mutual-CCPO-Rec.synth-option*
#> *Mutual-CCPO-Rec.add-ccpo lfp Mutual-CCPO-Rec.synth-lfp*
#> *Mutual-CCPO-Rec.add-ccpo gfp Mutual-CCPO-Rec.synth-gfp*
#> *Mutual-CCPO-Rec.add-ccpo ccpo Mutual-CCPO-Rec.synth-ccpo-class*)
|> *Context.theory-map* ›

no-notation *top* (‹⊤›)
no-notation *bot* (‹⊥›)
no-notation *sup* (**infixl** ‹⊔› *65*)
no-notation *inf* (**infixl** ‹⊓› *70*)

hide-const (**open**) *cont*

end

Part II

C-Parser

Chapter 10

Theory Variants for Target Architectures via *L4V-ARCH*

theory *Target-Architecture*
  imports *Main*
  keywords
    if-architecture-by :: *qed-global* % *proof* **and**
    if-architecture-context :: *thy-decl-block*

begin

ML ‹
structure Target-Architecture =
struct

datatype arch = ARM | ARM64 | ARM-HYP | RISCV64 | X64

val ARM-N   = ARM
val ARM64-N   = ARM64
val ARM-HYP-N = ARM-HYP
val RISCV64-N = RISCV64
val X64-N   = X64
*val architectures = [(ARM-N, (**here**, ARM)), (ARM64-N, (**here**, ARM64)), (ARM-HYP-N, (**here**, ARM-HYP)),*
*(RISCV64-N, (**here**, RISCV64)), (X64-N, (**here**, X64))]*

val rev-architectures = map (fn (name, (-, arch)) => (arch, name)) architectures

val string-of = AList.lookup (op =) rev-architectures #> the

fun check-architecture thy (name, pos) =
  case AList.lookup (op =) architectures name of
    SOME (def-pos, arch) =>
      let
        val markup = Position.entity-markup architecture-variant (name, def-pos)

```
        val - =
          Context-Position.reports (Proof-Context.init-global thy) [(pos, markup),
(pos, Markup.string)]
      in arch end
  | NONE => error (undefined architecture variant " ^ quote name ^ Position.here
pos ^ "\n ^
            known variants: " ^ @{make-string} (map #1 architectures))

  val active = the-default ARM (try (getenv-strict) "L4V-ARCH" |>
      Option.map (fn n => check-architecture @{theory} (n, here)))

end
›


ML ‹
structure Target-Architecture =
struct
  open Target-Architecture

fun arch-parser msg p = (Parse.$$$ "(" |-- Parse.list (Parse.name-position) --|
Parse.$$$ ")")
  :|-- " (fn archs =>
  let
    val archs' = map (check-architecture @{theory}) archs
    val is-active = member (op =) archs' active
   val - = if not is-active then writeln ("active architecture " ^ quote (string-of active)
              ^ " not in " ^ @{make-string} (map string-of archs') ^ ": " ^ msg) else ()
  in
    p is-active
  end)


fun is-active arch = (arch = active);

fun add-path arch =
   if is-active arch then Sign.add-path (string-of arch) else Sign.mandatory-path
(string-of arch)

val - =
  Outer-Syntax.command command-keyword ‹if-architecture-by› conditional ter-
minal backward proof
    (arch-parser aborting proof (oops) (fn is-active =>
      (Method.parse -- Scan.option Method.parse >> (fn (m1, m2) =>
       (Method.report m1;
        Option.map Method.report m2;
        (if is-active then
           Isar-Cmd.terminal-proof (m1, m2)
         else
```

```
        Toplevel.forget-proof (* oops *)))))));

val - =
  Outer-Syntax.command command-keyword ‹if-architecture-context› conditional
context or experiment
    (arch-parser experiment only (fn is-active =>
       (Scan.repeat Parse-Spec.context-element --| Parse.begin
         >> (fn elems =>
             if is-active then
               Toplevel.begin-nested-target (Target-Context.context-begin-nested-cmd []
elems)
             else
               Toplevel.begin-main-target true (Experiment.experiment-cmd elems #>
snd)))));

val - = Theory.setup
  (ML-Antiquotation.conditional binding ‹if-ARM›   (fn - => is-active ARM) #>
    ML-Antiquotation.conditional binding ‹if-ARM64› (fn - => is-active ARM64)
#>
   ML-Antiquotation.conditional binding ‹if-ARM-HYP› (fn - => is-active ARM-HYP)
#>
   ML-Antiquotation.conditional binding ‹if-RISCV64› (fn - => is-active RISCV64)
#>
    ML-Antiquotation.conditional binding ‹if-X64› (fn - => is-active X64))

end
›

end
```

Chapter 11

Unified Memory Model (UMM)

11.1 More Word Lemmas

This is a holding area for Word utility lemmas that are too specific or unpolished for the AFP, but which are reusable enough to be collected together for the rest of L4V. New utility lemmas that only prove facts about words should be added here (in preference to being kept where they were first needed).

theory *Word-Lemmas-Internal*
imports
Word-Lib.Word-Lemmas
Word-Lib.More-Word-Operations
Word-Lib.Many-More
Word-Lib.Word-Syntax
Word-Lib.Syntax-Bundles
begin

unbundle *bit-operations-syntax*
unbundle *bit-projection-infix-syntax*

lemmas *shiftl-nat-def* = *push-bit-eq-mult*[*of* - *a* **for** *a*::*nat*, *folded shiftl-def*]
lemmas *shiftr-nat-def* = *drop-bit-eq-div*[*of* - *a* **for** *a*::*nat*, *folded shiftr-def*]

declare *bit-simps*[*simp*]

lemma *signed-ge-zero-scast-eq-ucast*:
0 <=*s x* ⟹ *scast x* = *ucast x*
by (*simp add*: *scast-eq-ucast word-sle-msb-le*)

lemma *disjCI2*:
(¬ *P* ⟹ *Q*) ⟹ *P* ∨ *Q*

by *blast*

lemma *nat-diff-diff-le-lhs*:
$a + c - b \leq d \Longrightarrow a - (b - c) \leq (d :: nat)$
by *arith*

lemma *is-aligned-obvious-no-wrap'*:
$[\![\textit{is-aligned ptr sz}; x = 2 \hat{} sz - 1]\!]$
$\Longrightarrow ptr \leq ptr + x$
by (*fastforce simp*: *field-simps intro*: *is-aligned-no-overflow*)

lemmas *add-ge0-weak* = *add-increasing*[**where** *'a=int* **and** *b=0*]

lemmas *aligned-sub-aligned* = *Aligned.aligned-sub-aligned'*

lemma *minus-minus-swap*:
$[\![a \leq c; b \leq d; b \leq a; d \leq c ; (d :: nat) - b = c - a]\!]$
$\Longrightarrow a - b = c - d$
by *arith*

lemma *minus-minus-swap'*:
$[\![c \leq a; d \leq b; b \leq a; d \leq c ; (b :: nat) - d = a - c]\!]$
$\Longrightarrow a - b = c - d$
by *arith*

lemmas *word-le-mask-eq* = *le-mask-imp-and-mask*

lemma *int-and-leR*:
$0 \leq b \Longrightarrow a\ AND\ b \leq (b :: int)$
by (*rule AND-upper2*)

lemma *int-and-leL*:
$0 \leq a \Longrightarrow a\ AND\ b \leq (a :: int)$
by (*rule AND-upper1*)

lemma *if-then-1-else-0*:
$(\textit{if } P \textit{ then } 1 \textit{ else } 0) = (0 :: 'a :: zero\text{-}neq\text{-}one) \longleftrightarrow \neg P$
by (*simp split*: *if-split*)

lemma *if-then-0-else-1*:
$(\textit{if } P \textit{ then } 0 \textit{ else } 1) = (0 :: 'a :: zero\text{-}neq\text{-}one) \longleftrightarrow P$
by *simp*

lemmas *if-then-simps* = *if-then-0-else-1 if-then-1-else-0*

lemma *createNewCaps-guard*:
fixes $x :: 'a :: len\ word$
shows $[\![unat\ x = c; b < 2 \hat{} LENGTH('a)]\!]$

$\Longrightarrow (n < \mathit{of\text{-}nat}\ b \wedge n < x) = (n < \mathit{of\text{-}nat}\ (\mathit{min}\ (\mathit{min}\ b\ c)\ c))$
by (*metis* (*no-types*) *min-less-iff-conj nat-neq-iff unat-less-helper unat-ucast-less-no-overflow unsigned-less word-unat.Rep-inverse*)

lemma *bits-2-subtract-ineq*:
$i < (n :: ('a :: \mathit{len})\ \mathit{word})$
$\Longrightarrow 2 \hat{}\ \mathit{bits} + 2 \hat{}\ \mathit{bits} * \mathit{unat}\ (n - (1 + i)) = \mathit{unat}\ (n - i) * 2 \hat{}\ \mathit{bits}$
apply (*simp add*: *unat-sub word-minus-one-le-leq*)
apply (*subst unatSuc*)
apply *clarsimp*
apply *unat-arith*
apply (*simp only*: *mult-Suc-right*[*symmetric*])
apply (*rule trans*[*OF mult.commute*], *rule arg-cong2*[**where** *f*=(∗)], *simp-all*)
apply (*simp add*: *word-less-nat-alt*)
done

lemmas *double-neg-mask* = *neg-mask-combine*

lemmas *int-unat* = *uint-nat*[*symmetric*]

lemmas *word-sub-mono3* = *word-plus-mcs-4'*

lemma *shift-distinct-helper*:
⟦ $(x :: 'a :: \mathit{len}\ \mathit{word}) < \mathit{bnd}$; $y < \mathit{bnd}$; $x \neq y$; $x << n = y << n$; $n < \mathit{LENGTH}('a)$;
$\mathit{bnd} - 1 \leq 2 \hat{}\ (\mathit{LENGTH}('a) - n) - 1$ ⟧
$\Longrightarrow P$
apply (*cases n* = *0*)
apply *simp*
apply (*drule word-plus-mono-right*[**where** *x*=*1*])
apply *simp-all*
apply (*subst word-le-sub1*)
apply (*rule power-not-zero*)
apply *simp*
apply *simp*
apply (*drule*(*1*) *order-less-le-trans*)+
apply (*clarsimp simp*: *bang-eq*)
subgoal for *na*
apply (*drule-tac x*=*na* + *n* **in** *spec*)
apply (*simp add*: *nth-shiftl*)
apply (*cases na* + *n* < *LENGTH*('*a*), *simp-all*)
apply *safe*
apply (*drule*(*1*) *nth-bounded*)
apply *simp*
apply *simp*
apply (*drule*(*1*) *nth-bounded*)
apply *simp*
apply *simp*
done
done

lemma *of-nat-shift-distinct-helper*:
⟦ $x < bnd$; $y < bnd$; $x \neq y$; $(of\text{-}nat\ x :: 'a :: len\ word) << n = of\text{-}nat\ y << n$;
$n < LENGTH('a)$; $bnd \leq 2\ \hat{}\ (LENGTH('a) - n)$ ⟧
⟹ P
apply (*cases* $n = 0$)
apply (*simp add*: *word-unat.Abs-inject unats-def*)
apply (*subgoal-tac* $bnd < 2\ \hat{}\ LENGTH('a)$)
apply (*erule*(1) *shift-distinct-helper*[*rotated*, *rotated*, *rotated*])
defer
apply (*erule*(1) *of-nat-mono-maybe*[*rotated*])
apply (*erule*(1) *of-nat-mono-maybe*[*rotated*])
apply (*simp add*: *word-unat.Abs-inject unats-def*)
apply (*erule order-le-less-trans*)
apply (*rule power-strict-increasing*)
apply *simp*
apply *simp*
apply (*fastforce simp*: *unat-of-nat-minus-1 word-less-nat-alt*)
done

lemmas *pre-helper2* = *add-mult-in-mask-range*[*folded add-mask-fold*]

lemma *ptr-add-distinct-helper*:
⟦ *ptr-add* $(p :: 'a :: len\ word)\ (x * 2\ \hat{}\ n) = ptr\text{-}add\ p\ (y * 2\ \hat{}\ n)$; $x \neq y$;
$x < bnd$; $y < bnd$; $n < LENGTH('a)$;
$bnd \leq 2\ \hat{}\ (LENGTH('a) - n)$ ⟧
⟹ P
apply (*clarsimp simp*: *ptr-add-def word-unat-power*[*symmetric*]
shiftl-t2n[*symmetric*, *simplified mult.commute*])
using *of-nat-shift-distinct-helper*
by *blast*

lemma *unat-sub-le-strg*:
$unat\ v \leq v2 \wedge x \leq v \wedge y \leq v \wedge y < (x :: ('a :: len)\ word)$
⟶ $unat\ (x + (-\ 1\ -\ y)) \leq v2$
apply *clarsimp*
apply (*erule order-trans*[*rotated*])
apply (*fold word-le-nat-alt*)
apply (*rule order-trans*[*rotated*], *assumption*)
apply (*rule order-trans*[*rotated*], *rule word-sub-le*[**where** $y=y + 1$])
apply (*erule Word.inc-le*)
apply (*simp add*: *field-simps*)
done

lemma *multi-lessD*:
⟦ $(a :: nat) * b < c$; $0 < a$; $0 < b$ ⟧
⟹ $a < c \wedge b < c$
by (*cases a*, *simp-all*,*cases b*,*simp-all*)

lemmas *leq-high-bits-shiftr-low-bits-leq-bits* =
leq-high-bits-shiftr-low-bits-leq-bits-mask[*unfolded mask-2pm1*[*of high-bits*]]

lemmas *unat-le-helper* = *word-unat-less-le*

lemmas *word-of-nat-plus* = *of-nat-add*[**where** *'a='a* :: *len word*]
lemmas *word-of-nat-minus* = *of-nat-diff*[**where** *'a='a* :: *len word*]

lemma *word-up-bound*:
(*ptr* :: *'a* :: *len word*) ≤ *2* ^ *LENGTH*(*'a*) − *1*
by *auto*

lemma *base-length-minus-one-inequality*:
assumes *foo*: *wbase* ≤ *2* ^ *sz* − *1*
1 ≤ (*wlength* :: (*'a* :: *len*) *word*)
wlength ≤ *2* ^ *sz* − *wbase*
sz < *LENGTH* (*'a*)
shows *wbase* ≤ *wbase* + *wlength* − *1*
proof −

note *sz-less* = *power-strict-increasing*[*OF foo*(*4*), **where** *a=2*]

from *foo* **have** *plus*: *unat wbase* + *unat wlength* < *2* ^ *LENGTH*(*'a*)
apply −
apply (*rule order-le-less-trans*[*rotated*], *rule sz-less*, *simp*)
apply (*simp add*: *unat-arith-simps split*: *if-split-asm*)
done

from *foo* **show** *?thesis*
by (*simp add*: *unat-arith-simps plus*)
qed

lemmas *from-bool-to-bool-and-1* = *from-to-bool-last-bit*[**where** *x=r* **for** *r*]

lemmas *max-word-neq-0* = *max-word-not-0*

lemmas *word-le-p2m1* = *word-up-bound*[**where** *ptr=w* **for** *w*]

lemma *inj-ucast*:
⟦ *uc* = *ucast*; *is-up uc* ⟧
⟹ *inj uc*
using *down-ucast-inj is-up-down* **by** *blast*

lemma *ucast-eq-0*[*OF refl*]:
⟦ *c* = *ucast*; *is-up c* ⟧
⟹ (*c x* = *0*) = (*x* = *0*)
by (*metis uint-0-iff uint-up-ucast*)

lemmas *is-up-compose'* = *is-up-compose*

lemma *uint-is-up-compose*:
fixes *uc* :: *'a* :: *len word* ⇒ *'b* :: *len word*
and *uc'* :: *'b word* ⇒ *'c* :: *len sword*
assumes *uc* = *ucast*
and *uc'* = *ucast*
and *uuc* = *uc'* ∘ *uc*
shows ⟦ *is-up uc*; *is-up uc'* ⟧
⟹ *uint* (*uuc b*) = *uint b*
apply (*simp add*: *assms*)
apply (*frule is-up-compose*)
apply *simp-all*
apply (*simp only*: *Word.uint-up-ucast*)
done

lemma *uint-is-up-compose-pred*:
fixes *uc* :: *'a* :: *len word* ⇒ *'b* :: *len word*
and *uc'* :: *'b word* ⇒ *'c* :: *len sword*
assumes *uc* = *ucast* **and** *uc'* = *ucast* **and** *uuc* = *uc'* ∘ *uc*
shows ⟦ *is-up uc*; *is-up uc'* ⟧
⟹ *P* (*uint* (*uuc b*)) ⟷ *P*(*uint b*)
apply (*simp add*: *assms*)
apply (*frule is-up-compose*)
apply *simp-all*
apply (*simp only*: *Word.uint-up-ucast*)
done

lemma *is-down-up-sword*:
fixes *uc* :: *'a* :: *len word* ⇒ *'b* :: *len sword*
shows ⟦ *uc* = *ucast*; *LENGTH*(*'a*) < *LENGTH*(*'b*) ⟧
⟹ *is-up uc* = (¬ *is-down uc*)
by (*simp add*: *target-size source-size is-up-def is-down-def*)

lemma *is-not-down-compose*:
fixes *uc* :: *'a* :: *len word* ⇒ *'b* :: *len word*
and *uc'* :: *'b word* ⇒ *'c* :: *len sword*
shows ⟦ *uc* = *ucast*; *uc'* = *ucast*; *LENGTH*(*'a*) < *LENGTH*(*'c*) ⟧
⟹ ¬ *is-down* (*uc'* ∘ *uc*)
unfolding *is-down-def*
by (*simp add*: *Word.target-size Word.source-size*)

lemma *sint-ucast-uint*:
fixes *uc* :: *'a* :: *len word* ⇒ *'b* :: *len word*
and *uc'* :: *'b word* ⇒ *'c* :: *len sword*
assumes *uc* = *ucast* **and** *uc'* = *ucast* **and** *uuc*=*uc'* ∘ *uc*
and *LENGTH*(*'a*) < *LENGTH*(*'c signed*)
shows ⟦ *is-up uc*; *is-up uc'*⟧
⟹ *sint* (*uuc b*) = *uint b*

apply (*simp add*: *assms*)
apply (*frule is-up-compose'*)
apply *simp-all*
apply (*simp add*: *ucast-ucast-b*)
apply (*rule sint-ucast-eq-uint*)
apply (*insert assms*)
apply (*simp add*: *is-down-def target-size source-size*)
done

lemma *sint-ucast-uint-pred*:
fixes *uc* :: *'a* :: *len word* ⇒ *'b* :: *len word*
and *uc'* :: *'b word* ⇒ *'c* :: *len sword*
and *uuc* :: *'a word* ⇒ *'c sword*
assumes *uc* = *ucast* **and** *uc'* = *ucast* **and** *uuc*=*uc'* ∘ *uc*
and *LENGTH*(*'a*) < *LENGTH*(*'c*)
shows ⟦ *is-up uc*; *is-up uc'* ⟧
⟹ *P* (*uint b*) ⟷ *P* (*sint* (*uuc b*))
apply (*simp add*: *assms*)
apply (*insert sint-ucast-uint*[**where** *uc*=*uc* **and** *uc'*=*uc'* **and** *uuc*=*uuc* **and** *b* = *b*])
apply (*simp add*: *assms*)
done

lemma *sint-uucast-uint-uucast-pred*:
fixes *uc* :: *'a* :: *len word* ⇒ *'b* :: *len word*
and *uc'* :: *'b word* ⇒ *'c* :: *len sword*
assumes *uc* = *ucast* **and** *uc'* = *ucast* **and** *uuc*=*uc'* ∘ *uc*
and *LENGTH*(*'a*) < *LENGTH*(*'c*)
shows ⟦ *is-up uc*; *is-up uc'* ⟧
⟹ *P* (*uint*(*uuc b*)) ⟷ *P* (*sint* (*uuc b*))
apply (*simp add*: *assms*)
apply (*insert sint-ucast-uint*[**where** *uc*=*uc* **and** *uc'*=*uc'* **and** *uuc*=*uuc* **and** *b* = *b*])
apply (*insert uint-is-up-compose-pred*[**where** *uc*=*uc* **and** *uc'*=*uc'* **and** *uuc*=*uuc* **and** *b*=*b*])
apply (*simp add*: *assms uint-is-up-compose-pred*)
done

lemma *unat-minus'*:
fixes *x* :: *'a* :: *len word*
shows *x* ≠ *0* ⟹ *unat* (−*x*) = *2* ^ *LENGTH*(*'a*) − *unat x*
by (*simp add*: *unat-minus wsst-TYs*(*3*))

lemma *word-nth-neq*:
n < *LENGTH*(*'a*) ⟹ (~~ *x* :: *'a* :: *len word*) !! *n* = (¬ *x* !! *n*)
by (*simp add*: *word-size word-ops-nth-size*)

lemma *word-wrap-of-natD*:
fixes *x* :: *'a* :: *len word*

```
  assumes wraps: ¬ x ≤ x + of-nat n
  shows   ∃k. x + of-nat k = 0 ∧ k ≤ n
proof −
  show ?thesis
  proof (rule exI [where x = unat (− x)], intro conjI)
    show x + of-nat (unat (−x)) = 0
      by simp
  next
    show unat (−x) ≤ n
    proof (subst unat-minus')
      from wraps show x ≠ 0
        by (rule contrapos-pn, simp add: not-le)
    next
      show 2 ^ LENGTH('a) − unat x ≤ n using wraps
        apply (simp add: no-olen-add-nat le-diff-conv not-less)
        apply (erule order-trans)
        apply (simp add: unat-of-nat)
        done
    qed
  qed
qed

lemma two-bits-cases:
  ⟦ LENGTH('a) > 2; (x :: 'a :: len word) && 3 = 0 ⟹ P; x && 3 = 1 ⟹ P;
     x && 3 = 2 ⟹ P; x && 3 = 3 ⟹ P ⟧
   ⟹ P
  apply (frule and-mask-cases[where n=2 and x=x, simplified mask-eq])
  using upt-conv-Cons by auto[1]

lemma zero-OR-eq:
  y = 0 ⟹ (x || y) = x
  by simp

declare is-aligned-neg-mask-eq[simp]
declare is-aligned-neg-mask-weaken[simp]

lemmas mask-in-range = neg-mask-in-mask-range[folded add-mask-fold]
lemmas aligned-range-offset-mem = aligned-offset-in-range[folded add-mask-fold]
lemmas range-to-bl' = mask-range-to-bl'[folded add-mask-fold]
lemmas range-to-bl = mask-range-to-bl[folded add-mask-fold]
lemmas aligned-ranges-subset-or-disjoint = aligned-mask-range-cases[folded add-mask-fold]
lemmas aligned-range-offset-subset = aligned-mask-range-offset-subset[folded add-mask-fold]
lemmas aligned-diff = aligned-mask-diff[unfolded mask-2pm1]
lemmas aligned-ranges-subset-or-disjoint-coroll = aligned-mask-ranges-disjoint[folded
add-mask-fold]
lemmas distinct-aligned-addresses-accumulate = aligned-mask-ranges-disjoint2[folded
add-mask-fold]

lemmas bang-big = test-bit-over
```

lemma *unat-and-mask-le*:
  $n < LENGTH('a) \Longrightarrow unat\ (x\ \&\&\ mask\ n) \leq 2\hat{}n$ **for** $x::'a::len\ word$
  by (*simp add*: *and-mask-less'* *order-less-imp-le* *unat-less-power*)

lemma *sign-extend-less-mask-idem*:
  $[\![w \leq mask\ n;\ n < size\ w]\!] \Longrightarrow sign\text{-}extend\ n\ w = w$
  by (*simp add*: *sign-extend-def* *le-mask-imp-and-mask* *le-mask-high-bits*)

lemma *word-and-le*:
  $a \leq c \Longrightarrow (a :: 'a :: len\ word)\ \&\&\ b \leq c$
  by (*subst and.commute*) (*erule word-and-le'*)

lemma *le-smaller-mask*:
  $[\![x \leq mask\ n;\ n \leq m]\!] \Longrightarrow x \leq mask\ m$ **for** $x :: 'a::len\ word$
  by (*erule* (*1*) *order.trans*[*OF - mask-mono*])

lemma *upcast-less-unat-less*:
  assumes *less*: $UCAST('a \to 'b)\ x < UCAST('a \to 'b)\ (of\text{-}nat\ y)$
  assumes *len*: $LENGTH('a::len) \leq LENGTH('b::len)$
  assumes *bound*: $y < 2\ \hat{}\ LENGTH('a)$
  shows $unat\ x < y$
 by (*rule unat-mono*[*OF less*, *simplified unat-ucast-up-simp*[*OF len*] *unat-of-nat-eq*[*OF bound*]])

lemma *word-ctz-max*:
  $word\text{-}ctz\ w \leq size\ w$
  unfolding *word-ctz-def*
  by (*rule order-trans*[*OF List.length-takeWhile-le*], *clarsimp simp*: *word-size*)

lemma *scast-of-nat-small*:
  $x < 2\ \hat{}\ (LENGTH('a) - 1) \Longrightarrow scast\ (of\text{-}nat\ x :: 'a :: len\ word) = (of\text{-}nat\ x :: 'b :: len\ word)$
  apply *transfer*
  apply *simp*
  by (*metis One-nat-def id-apply int-eq-sint of-int-eq-id signed-of-nat*)

lemmas *casts-of-nat-small* = *ucast-of-nat-small scast-of-nat-small*

— The conditions under which ‘takeWhile P xs = take n xs‘ and ‘dropWhile P xs = drop n xs‘
definition *list-while-len* **where**
  $list\text{-}while\text{-}len\ P\ n\ xs \equiv (\forall i.\ i < n \longrightarrow i < length\ xs \longrightarrow P\ (xs\ !\ i)) \wedge (n < length\ xs \longrightarrow \neg\ P\ (xs\ !\ n))$

lemma *list-while-len-iff-takeWhile-eq-take*:
  $list\text{-}while\text{-}len\ P\ n\ xs \longleftrightarrow takeWhile\ P\ xs = take\ n\ xs$
  unfolding *list-while-len-def*

apply (*rule iffI*[*OF takeWhile-eq-take-P-nth*], *simp+*)
apply (*intro conjI allI impI*)
apply (*rule takeWhile-take-has-property-nth*, *clarsimp*)
apply (*drule-tac f=length* **in** *arg-cong*, *simp*)
apply (*induct xs arbitrary*: *n*; *clarsimp split*: *if-splits*)
done

lemma *list-while-len-exists*:
$\exists n.$ *list-while-len P n xs*
apply (*induction xs*; *simp add*: *list-while-len-def*)
apply (*erule exE*)
subgoal for *x xs n*
apply (*cases P x*)
apply (*rule exI*[**where** *x=Suc n*], *clarsimp*)
subgoal for *i* **by** (*cases i*; *simp*)
subgoal by (*rule exI*[**where** *x=0*], *simp*)
done
done

lemma *takeWhile-truncate*:
length (*takeWhile P xs*) $\leq$ *m*
$\Longrightarrow$ *takeWhile P* (*take m xs*) = *takeWhile P xs*
apply (*cut-tac list-while-len-exists*[**where** *P=P* **and** *xs=xs*], *clarsimp*)
subgoal for *n*
apply (*cases n* $\leq$ *m*)
apply (*subgoal-tac list-while-len P n* (*take m xs*))
apply(*simp add*: *list-while-len-iff-takeWhile-eq-take*)
apply (*clarsimp simp*: *list-while-len-def*)
apply (*simp add*: *list-while-len-iff-takeWhile-eq-take*)
done
done

lemma *word-clz-shiftr-1*:
fixes *z*::*'a*::*len word*
assumes *wordsize*: *1* < *LENGTH*(*'a*)
shows *z* $\neq$ *0* $\Longrightarrow$ *word-clz* (*z* >> *1*) = *word-clz z* + *1*
supply *word-size* [*simp*]
apply (*clarsimp simp*: *word-clz-def*)
using *wordsize* **apply** (*subst bl-shiftr*, *simp*)
apply (*subst takeWhile-append2*; *simp add*: *wordsize*)
apply (*subst takeWhile-truncate*)
using *word-clz-nonzero-max*[**where** *w=z*]
apply (*clarsimp simp*: *word-clz-def*)
apply *simp+*
done

lemma *shiftr-Suc*:
fixes *x*::*'a*::*len word*
shows *x* >> (*Suc n*) = *x* >> *n* >> *1*

by (*clarsimp simp*: *shiftr-shiftr*)

lemma *word-clz-shiftr*:
fixes *z* :: *'a*::*len word*
shows *n* < *LENGTH*('*a*) ⟹ *mask n* < *z* ⟹ *word-clz* (*z* >> *n*) = *word-clz z* + *n*
apply (*simp add*: *le-mask-iff Not-eq-iff*[*THEN iffD2*, *OF le-mask-iff*, *simplified not-le*]
word-neq-0-conv)
apply (*induction n*; *simp*)
subgoal for *n*
apply (*subgoal-tac* 0 < *z* >> *n*)
apply (*subst shiftr-Suc*)
apply (*subst word-clz-shiftr-1*; *simp*)
apply (*clarsimp simp*: *word-less-nat-alt shiftr-div-2n' div-mult2-eq*)
apply (*cases unat z div* 2 ^ *n* = 0; *simp*)
apply (*clarsimp simp*: *div-eq-0-iff*)
done
done

lemma *mask-to-bl-exists-True*:
x && *mask n* ≠ 0 ⟹ ∃ *m*. (*rev* (*to-bl x*)) ! *m* ∧ *m* < *n*
apply (*subgoal-tac* ¬(∀ *m*. *m* < *n* ⟶ ¬(*rev* (*to-bl x*)) ! *m*), *fastforce*)
apply (*intro notI*)
apply (*subgoal-tac x* && *mask n* = 0, *clarsimp*)
apply (*clarsimp simp*: *eq-zero-set-bl in-set-conv-nth*)
apply (*subst* (*asm*) *to-bl-nth*, *clarsimp simp*: *word-size*)
apply (*clarsimp simp*: *word-size*)
by (*meson nth-mask test-bit-bl word-and-nth*)

lemma *word-ctz-shiftr-1*:
fixes *z*::*'a*::*len word*
assumes *wordsize*: 1 < *LENGTH*('*a*)
shows *z* ≠ 0 ⟹ 1 ≤ *word-ctz z* ⟹ *word-ctz* (*z* >> 1) = *word-ctz z* − 1
supply *word-size* [*simp*]
apply (*clarsimp simp*: *word-ctz-def*)
using *wordsize* **apply** (*subst bl-shiftr*, *simp*)
apply (*simp add*: *rev-take*)
apply (*subgoal-tac*
length (*takeWhile Not* (*rev* (*to-bl z*))) − *Suc* 0
= *length* (*takeWhile Not* (*take* 1 (*rev* (*to-bl z*)) @ *drop* 1 (*rev* (*to-bl z*)))) − *Suc* 0)
apply (*subst* (*asm*) *takeWhile-append2*)
apply *clarsimp*
apply (*cases rev* (*to-bl z*); *simp*)
apply *clarsimp*
apply (*subgoal-tac* ∃ *m*. (*rev* (*to-bl z*)) ! *m* ∧ *m* < *LENGTH*('*a*), *clarsimp*)
subgoal for *m*
apply (*cases m*)

```
    apply (cases rev (to-bl z); simp)
   subgoal for nat
    apply (subst takeWhile-append1, subst in-set-conv-nth)
     apply (rule exI[where x=nat])
     apply (intro conjI)
      apply (clarsimp simp: wordsize)
    using wordsize apply linarith
     apply (rule refl, clarsimp)
    apply simp
    done
   done
  apply (rule mask-to-bl-exists-True, simp)
 apply simp
 done

lemma word-ctz-bound-below-helper:
 fixes x :: 'a::len word
 assumes sz: n ≤ LENGTH('a)
 shows x && mask n = 0
       ⟹ to-bl x = (take (LENGTH('a) − n) (to-bl x) @ replicate n False)
 apply (subgoal-tac replicate n False = drop (LENGTH('a) − n) (to-bl x))
  apply (subgoal-tac True ∉ set (drop (LENGTH('a) − n) (to-bl x)))
   apply (drule list-of-false, clarsimp simp: sz)
  apply (drule-tac sym[where t=drop n x for n x], clarsimp)
 apply (rule sym)
 apply (rule is-aligned-drop; clarsimp simp: is-aligned-mask sz)
 done

lemma word-ctz-bound-below:
 fixes x :: 'a::len word
 assumes sz[simp]: n ≤ LENGTH('a)
 shows x && mask n = 0 ⟹ n ≤ word-ctz x
 apply (clarsimp simp: word-ctz-def)
 apply (subst word-ctz-bound-below-helper[OF sz]; simp)
 apply (subst takeWhile-append2; clarsimp)
 done

lemma word-ctz-bound-above:
 fixes x :: 'a::len word
 shows x && mask n ≠ 0 ⟹ word-ctz x < n
 apply (cases n ≤ LENGTH('a))
  apply (frule mask-to-bl-exists-True, clarsimp)
 subgoal for m
   apply (clarsimp simp: word-ctz-def)
   apply (subgoal-tac m < length ((rev (to-bl x))))
    apply (subst id-take-nth-drop[where xs=rev (to-bl x)], assumption)
    apply (subst takeWhile-tail, simp)
    apply (rule order.strict-trans1)
     apply (rule List.length-takeWhile-le)
```

```
    apply simp
   apply (erule order.strict-trans2, clarsimp)
   done
 apply (simp add: not-le)
 apply (erule le-less-trans[OF word-ctz-max, simplified word-size])
 done

lemma word-ctz-shiftr:
 fixes z::'a::len word
 assumes nz: z ≠ 0
 shows n < LENGTH('a) ⟹ n ≤ word-ctz z ⟹ word-ctz (z >> n) = word-ctz
z − n
 apply (induction n; simp)
 subgoal for n
   apply (subst shiftr-Suc)
   apply (subst word-ctz-shiftr-1, simp)
    apply clarsimp
    apply (subgoal-tac word-ctz z < n, clarsimp)
    apply (rule word-ctz-bound-above, clarsimp simp: word-size)
    apply (subst (asm) and-mask-eq-iff-shiftr-0[symmetric], clarsimp simp: nz)
   apply (rule word-ctz-bound-below, clarsimp simp: word-size)
   apply (rule mask-zero)
   apply (rule is-aligned-shiftr, simp add: is-aligned-mask)
   apply (cases z && mask (Suc n) = 0, simp)
   apply (frule word-ctz-bound-above[rotated]; clarsimp simp: word-size)
   apply simp
   done
 done

— Useful for solving goals of the form ‘(w::32 word) <= 0xFFFFFFFF‘ by simplification
lemma word-less-max-simp:
 fixes w :: 'a::len word
 assumes max-w = −1
 shows w ≤ max-w
 unfolding assms by simp

lemma word-and-mask-word-ctz-zero:
 assumes l = word-ctz w
 shows w && mask l = 0
 unfolding word-ctz-def assms
 apply (word-eqI)
 apply (drule takeWhile-take-has-property-nth)
 apply (simp add: test-bit-bl)
 done

lemma word-ctz-len-word-and-mask-zero:
 fixes w :: 'a::len word
 shows word-ctz w = LENGTH('a) ⟹ w = 0
```

by (*drule sym*, *drule word-and-mask-word-ctz-zero*, *simp*)

lemma *word-le-1*:
fixes $w :: 'a::len\ word$
shows $w \leq 1 \longleftrightarrow w = 0 \vee w = 1$
using *dual-order.antisym lt1-neq0 word-zero-le* **by** *blast*

lemma *less-ucast-ucast-less'*:
$x < UCAST('b \to 'a)\ y \Longrightarrow UCAST('a \to 'b)\ x < y$
for $x :: 'a::len\ word$ **and** $y :: 'b::len\ word$
by (*clarsimp simp*: *order.strict-iff-order dual-order.antisym le-ucast-ucast-le*)

lemma *ucast-up-less-bounded-implies-less-ucast-down'*:
assumes *len*: $LENGTH('a::len) < LENGTH('b::len)$
assumes *bound*: $y < 2 \hat{} LENGTH('a)$
assumes *less*: $UCAST('a \to 'b)\ x < y$
shows $x < UCAST('b \to 'a)\ y$
apply (*rule le-less-trans*[*OF - ucast-mono*[*OF less bound*]])
using *len* **by** (*simp add*: *is-down ucast-down-ucast-id*)

lemma *ucast-up-less-bounded-iff-less-ucast-down'*:
assumes *len*: $LENGTH('a::len) < LENGTH('b::len)$
assumes *bound*: $y < 2 \hat{} LENGTH('a)$
shows $UCAST('a \to 'b)\ x < y \longleftrightarrow x < UCAST('b \to 'a)\ y$
apply (*rule iffI*)
prefer *2*
apply (*simp add*: *less-ucast-ucast-less'*)
using *assms* **by** (*rule ucast-up-less-bounded-implies-less-ucast-down'*)

lemma *word-of-int-word-of-nat-eqD*:
$[\![word\text{-}of\text{-}int\ x = (word\text{-}of\text{-}nat\ y :: 'a :: len\ word);\ 0 \leq x;\ x < 2\hat{}LENGTH('a);$
$y < 2\hat{}LENGTH('a)]\!]$
$\Longrightarrow nat\ x = y$
by (*metis nat-eq-numeral-power-cancel-iff of-nat-inj word-of-int-nat zless2p zless-nat-conj*)

lemma *ucast-down-0*:
$[\![UCAST('a::len \to 'b::len)\ x = 0;\ unat\ x < 2\hat{}LENGTH('b)]\!] \Longrightarrow x = 0$
by (*metis Word.of-nat-unat unat-0 unat-eq-of-nat word-unat-eq-iff*)

lemma *uint-minus-1-eq*:
‹*uint* $(-\ 1 :: 'a\ word) = 2 \hat{} LENGTH('a::len) - 1$›
by *transfer* (*simp add*: *mask-eq-exp-minus-1*)

lemma *FF-eq-minus-1*:
‹*0xFF* $= (-\ 1 :: 8\ word)$›
by *simp*

lemma *shiftl-t2n'*:
$w << n = w * (2 \hat{} n)$ **for** $w :: 'a::len\ word$

by (*simp add*: *shiftl-t2n*)

end

theory *Word-Lemmas-32-Internal*
imports *Word-Lib.Word-Lib-Sumo Word-Lib.Machine-Word-32 Word-Lemmas-Internal*
begin

lemmas *sint-eq-uint-32* = *sint-eq-uint-2pl*[**where** *'a=32*, *simplified*]

lemmas *sle-positive-32* = *sle-le-2pl*[**where** *'a=32*, *simplified*]

lemmas *sless-positive-32* = *sless-less-2pl*[**where** *'a=32*, *simplified*]

lemma *zero-le-sint-32*:
⟦ *0* ≤ (*a* :: *word32*); *a* < *0x80000000* ⟧
⟹ *0* ≤ *sint a*
by (*clarsimp simp*: *sint-eq-uint-32 unat-less-helper*)

lemmas *unat-add-simple* = *iffD1*[*OF unat-add-lem*[**where** *'a* = *32*, *folded word-bits-def*]]

lemma *upto-enum-inc-1*:
a < *2* ^ *word-bits* − *1*
⟹ [(*0*:: *'a* :: *len word*) *.e.* *1* + *a*] = [*0.e.a*] @ [(*1*+*a*)]
using *upper-trivial upto-enum-inc-1-len* **by** *force*

lemmas *upt-enum-offset-trivial* =
upt-enum-offset-trivial[**where** *'a=32*, *folded word-bits-def*]

lemmas *unat32-eq-of-nat* = *unat-eq-of-nat*[**where** *'a=32*, *folded word-bits-def*]

declare *mask-32-max-word*[*simp*]

lemma *le-32-mask-eq*:
(*bits* :: *word32*) ≤ *32* ⟹ *bits* && *mask 6* = *bits*
by (*fastforce elim*: *le-less-trans intro*: *less-mask-eq*)

lemmas *scast-1-32*[*simp*] = *scast-1*[**where** *'a=32*]

lemmas *mask-32-id*[*simp*] = *mask-len-id*[**where** *'a=32*, *folded word-bits-def*]

lemmas *t2p-shiftr-32* = *t2p-shiftr*[**where** *'a=32*, *folded word-bits-def*]

lemma *mask-eq1-nochoice*:
(*x* :: *word32*) && *1* = *x*
⟹ *x* = *0* ∨ *x* = *1*
using *mask-eq1-nochoice len32* **by** *force*

lemmas *const-le-unat-word-32* = *const-le-unat*[**where** *'a=32*, *folded word-bits-def*]

lemmas *createNewCaps-guard-helper* =
createNewCaps-guard[**where** *'a=32*, *folded word-bits-def*]

lemma *word-log2-max-word32*[*simp*]:
word-log2 (*w* :: *32 word*) < *32*
using *word-log2-max*[**where** *w=w*]
by (*simp add*: *word-size*)

lemma *mapping-two-power-16-64-inequality*:
assumes *sz*: *sz* ≤ *4* **and** *len*: *unat* (*len* :: *word32*) = *2* ^ *sz*
shows *unat* (*len* ∗ *8* − *1*) ≤ *127*
using *pow-sub-less*[**where** *'a=32* **and** *b=3*, *simplified*]
proof −
have *len2*: *len* = *2* ^ *sz*
apply (*rule word-unat.Rep-eqD*, *simp only*: *len*)
using *sz*
apply *simp*
done

show *?thesis* **using** *two-power-increasing-less-1*[**where** *'a=32* **and** *n=sz* + *3*
and *m=7*]
apply (*simp add*: *word-le-nat-alt sz power-add len2 field-simps bintrunc-Suc-numeral*)
using *le-trans take-bit-nat-less-eq-self* **by** *blast*
qed

lemmas *pre-helper2-32* = *pre-helper2*[**where** *'a=32*, *folded word-bits-def*]

lemmas *of-nat-shift-distinct-helper-machine* =
of-nat-shift-distinct-helper[**where** *'a=32*, *folded word-bits-def*]

lemmas *ptr-add-distinct-helper-32* =
ptr-add-distinct-helper[**where** *'a=32*, *folded word-bits-def*]

lemmas *mask-out-eq-0-32* = *mask-out-eq-0*[**where** *'a=32*, *folded word-bits-def*]

lemmas *neg-mask-mask-unat-32* = *neg-mask-mask-unat*[**where** *'a=32*, *folded word-bits-def*]

lemmas *unat-less-iff-32* = *unat-less-iff*[**where** *'a=32*, *folded word-bits-def*]

lemmas *is-aligned-no-overflow3-32* = *is-aligned-no-overflow3*[**where** *'a=32*, *folded word-bits-def*]

lemmas *unat-ucast-16-32* = *unat-signed-ucast-less-ucast*[**where** *'a=16* **and** *'b=32*, *simplified*]

lemma *scast-mask-8*:
  scast (*mask 8* :: *sword32*) = (*mask 8* :: *word32*)
  by (*clarsimp simp*: *mask-eq*)

lemmas *ucast-le-8-32-equiv* = *ucast-le-up-down-iff*[**where** *'a=8* **and** *'b=32*, *simplified*]

lemma *signed-unat-minus-one-32*:
  unat (*−1* :: *32 signed word*) = *4294967295*
  by (*simp del*: *word-pow-0 diff-0 add*: *unat-sub-if' minus-one-word*)

lemmas *two-bits-cases-32* = *two-bits-cases*[**where** *'a=32*, *simplified*]

lemmas *word-ctz-not-minus-1-32* = *word-ctz-not-minus-1*[**where** *'a=32*, *simplified*]

lemmas *sint-ctz-32* = *sint-ctz*[**where** *'a=32*, *simplified*]

lemmas *scast-specific-plus32* =
  scast-of-nat-signed-to-unsigned-add[**where** *'a=32* **and** *x=word-ctz x* **and** *y=0x20*
for *x*,
simplified]
lemmas *scast-specific-plus32-signed* =
  scast-of-nat-unsigned-to-signed-add[**where** *'a=32* **and** *x=word-ctz x* **and** *y=0x20*
for *x*,
simplified]

lemma *neq-0-unat*: *x ≠ 0 ⟹ 0 < unat x* **for** *x*::*machine-word*
  by (*simp add*: *unat-gt-0*)

end

theory *Word-Lemmas-64-Internal*
imports *Word-Lib.Word-Lib-Sumo Word-Lib.Word-64 Word-Lib.Machine-Word-64*
begin

lemmas *word-and-max-simps* = *word-and-max-simps word64-and-max-simp*

lemmas *unat-add-simple* = *iffD1*[*OF unat-add-lem*[**where** *'a* = *64*, *folded word-bits-def*]]

lemma *unat-length-4-helper*:
  assumes *unat* (*l*::*machine-word*) = *length args* ¬ *l* < *4*
  shows ∃ *x xa xb xc xs*. *args* = *x*#*xa*#*xb*#*xc*#*xs*
proof −
  from *assms*(*2*) **have** *l* ≥ *4* **by** *auto*
  hence *unat l* ≥ *4* **by** (*simp add*: *const-le-unat*)

with *assms* **have** *length args* $\geq$ *4* **by** *auto*
thus *?thesis* **by** (*auto simp*: *numeral-eq-Suc Suc-le-length-iff*)
qed

lemma *ucast-drop-big-mask*:
UCAST(*64* $\rightarrow$ *16*) (*x* && *0xFFFF*) = *UCAST*(*64* $\rightarrow$ *16*) *x*
by *word-bitwise*

lemma *first-port-last-port-compare*:
UCAST(*16* $\rightarrow$ *32 signed*) (*UCAST*(*64* $\rightarrow$ *16*) (*xa* && *0xFFFF*))
<*s UCAST*(*16* $\rightarrow$ *32 signed*) (*UCAST*(*64* $\rightarrow$ *16*) (*x* && *0xFFFF*))
= (*UCAST*(*64* $\rightarrow$ *16*) *xa* < *UCAST*(*64* $\rightarrow$ *16*) *x*)
apply (*clarsimp simp*: *word-sless-alt ucast-drop-big-mask*)
apply (*subst sint-ucast-eq-uint*, *clarsimp simp*: *is-down*)+
by (*simp add*: *word-less-alt*)

lemma *machine-word-and-3F-less-40*:
(*w* :: *machine-word*) && *0x3F* < *0x40*
by (*rule word-and-less'*, *simp*)

lemmas *unat64-eq-of-nat* = *unat-eq-of-nat*[**where** *'a=64*, *folded word-bits-def*]

lemma *unat-mask-3-less-8*:
unat (*p* && *mask 3* :: *word64*) < *8*
apply (*rule unat-less-helper*)
apply (*rule order-le-less-trans*, *rule word-and-le1*)
apply (*simp add*: *mask-def*)
done

lemma *scast-specific-plus64*:
scast (*of-nat* (*word-ctz x*) + *0x20* :: *64 signed word*) = *of-nat* (*word-ctz x*) + (*0x20* :: *machine-word*)
by (*metis of-nat-add of-nat-numeral scast-of-nat*)

lemma *scast-specific-plus64-signed*:
scast (*of-nat* (*word-ctz x*) + *0x20* :: *machine-word*) = *of-nat* (*word-ctz x*) + (*0x20* :: *64 signed word*)
by (*metis scast-scast-id*(*2*) *scast-specific-plus64*)

lemmas *mask-64-id*[*simp*] = *mask-len-id*[**where** *'a=64*, *folded word-bits-def*]
mask-len-id[**where** *'a=64*, *simplified*]

lemma *neq-0-unat*: *x* $\neq$ *0* $\Longrightarrow$ *0* < *unat x* **for** *x*::*machine-word*
by (*simp add*: *unat-gt-0*)

end

11.2 Distinct Proposition

theory *Distinct-Prop*
imports
Word-Lib.Many-More
HOL−Library.Prefix-Order
begin

primrec
distinct-prop :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a\ list \Rightarrow bool)$
where
distinct-prop P [] = *True*
| *distinct-prop P* $(x \# xs) = ((\forall y{\in}set\ xs.\ P\ x\ y) \wedge distinct\text{-}prop\ P\ xs)$

primrec
distinct-sets :: $'a\ set\ list \Rightarrow bool$
where
distinct-sets [] = *True*
| *distinct-sets* $(x\#xs) = (x \cap \bigcup\ (set\ xs) = \{\} \wedge distinct\text{-}sets\ xs)$

declare *distinct-sets.simps* [*simp del*]

lemma *distinct-prop-map*:
distinct-prop P (*map f xs*) = *distinct-prop* $(\lambda x\ y.\ P\ (f\ x)\ (f\ y))\ xs$
by (*induct xs*) *auto*

lemma *distinct-prop-append*:
distinct-prop P (*xs* @ *ys*) =
(*distinct-prop P xs* ∧ *distinct-prop P ys* ∧ $(\forall x \in set\ xs.\ \forall y \in set\ ys.\ P\ x\ y)$)
by (*induct xs arbitrary*: *ys*) (*auto simp*: *conj-comms ball-Un*)

lemma *distinct-prop-distinct*:
$[\![\ distinct\ xs;\ \bigwedge x\ y.\ [\![\ x \in set\ xs;\ y \in set\ xs;\ x \neq y\ ]\!] \Longrightarrow P\ x\ y\ ]\!] \Longrightarrow$ *distinct-prop P xs*
by (*induct xs*) *auto*

lemma *distinct-prop-True* [*simp*]:
distinct-prop $(\lambda x\ y.\ True)\ xs$
by (*induct xs*, *auto*)

lemma *distinct-prefix*:
$[\![\ distinct\ xs;\ ys \leq xs\ ]\!] \Longrightarrow distinct\ ys$
apply (*induct xs arbitrary*: *ys*; *clarsimp*)
subgoal for *a xs ys*
apply (*cases ys*; *clarsimp*)
by (*fastforce simp*: *less-eq-list-def dest*: *set-mono-prefix*)
done

lemma *distinct-sets-prop*:
distinct-sets xs = *distinct-prop* (λ*x y*. *x* ∩ *y* = {}) *xs*
by (*induct xs*) (*auto simp add*: *distinct-sets.simps*)

lemma *distinct-take-strg*:
distinct xs ⟶ *distinct* (*take n xs*)
by *simp*

lemma *distinct-prop-prefixE*:
⟦ *distinct-prop P ys*; *prefix xs ys* ⟧ ⟹ *distinct-prop P xs*
apply (*induct xs arbitrary*: *ys*; *clarsimp*)
subgoal for *a xs ys*
apply (*cases ys*; *clarsimp*)
by (*fastforce dest*: *set-mono-prefix*)
done

lemma *distinct-sets-union-sub*:
⟦*x* ∈ *A*; *distinct-sets* [*A*,*B*]⟧ ⟹ *A* ∪ *B* − {*x*} = *A* − {*x*} ∪ *B*
by (*auto simp*: *distinct-sets-def*)

lemma *distinct-sets-append*:
distinct-sets (*xs* @ *ys*) ⟹ *distinct-sets xs* ∧ *distinct-sets ys*
apply (*subst distinct-sets-prop*)+
apply (*subst* (*asm*) *distinct-sets-prop*)
apply (*subst* (*asm*) *distinct-prop-append*)
apply *clarsimp*
done

lemma *distinct-sets-append1*:
distinct-sets (*xs* @ *ys*) ⟹ *distinct-sets xs*
by (*drule distinct-sets-append*, *simp*)

lemma *distinct-sets-append2*:
distinct-sets (*xs* @ *ys*) ⟹ *distinct-sets ys*
by (*drule distinct-sets-append*, *simp*)

lemma *distinct-sets-append-Cons*:
distinct-sets (*xs* @ *a* # *ys*) ⟹ *distinct-sets* (*xs* @ *ys*)
apply (*subst distinct-sets-prop*)+
apply (*subst* (*asm*) *distinct-sets-prop*)
apply (*subst distinct-prop-append*)
apply (*subst* (*asm*) *distinct-prop-append*)
apply *clarsimp*
done

lemma *distinct-sets-append-Cons-disjoint*:
distinct-sets (*xs* @ *a* # *ys*) ⟹ *a* ∩ ⋃ (*set xs*) = {}
apply (*subst* (*asm*) *distinct-sets-prop*)
apply (*subst* (*asm*) *distinct-prop-append*)

apply (*subst Int-commute*)
apply (*subst Union-disjoint*)
apply *clarsimp*
done

lemma *distinct-prop-take*:
⟦*distinct-prop P xs*; *i* < *length xs*⟧ ⟹ *distinct-prop P* (*take i xs*)
by (*metis take-is-prefix distinct-prop-prefixE*)

lemma *distinct-sets-take*:
⟦*distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*take i xs*)
by (*simp add*: *distinct-sets-prop distinct-prop-take*)

lemma *distinct-prop-take-Suc*:
⟦*distinct-prop P xs*; *i* < *length xs*⟧ ⟹ *distinct-prop P* (*take* (*Suc i*) *xs*)
by (*metis distinct-prop-take not-less take-all*)

lemma *distinct-sets-take-Suc*:
⟦*distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*take* (*Suc i*) *xs*)
by (*simp add*: *distinct-sets-prop distinct-prop-take-Suc*)

lemma *distinct-prop-rev*:
distinct-prop P (*rev xs*) = *distinct-prop* (λ*y x*. *P x y*) *xs*
by (*induct xs*) (*auto simp*: *distinct-prop-append*)

lemma *distinct-sets-rev* [*simp*]:
distinct-sets (*rev xs*) = *distinct-sets xs*
apply (*unfold distinct-sets-prop*)
apply (*subst distinct-prop-rev*)
apply (*subst Int-commute*)
apply *clarsimp*
done

lemma *distinct-sets-drop*:
⟦*distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*drop i xs*)
apply (*cases i=0*, *simp*)
apply (*subst distinct-sets-rev* [*symmetric*])
apply (*subst rev-drop*)
apply (*subst distinct-sets-take*, *simp-all*)
done

lemma *distinct-sets-drop-Suc*:
⟦*distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*drop* (*Suc i*) *xs*)
apply (*subst distinct-sets-rev* [*symmetric*])
apply (*subst rev-drop*)
apply (*subst distinct-sets-take*, *simp-all*)
done

lemma *distinct-sets-take-nth*:

⟦*distinct-sets xs*; *i* < *length xs*; *x* ∈ *set* (*take i xs*)⟧ ⟹ *x* ∩ *xs* ! *i* = {}
apply (*drule* (*1*) *distinct-sets-take-Suc*)
apply (*subst* (*asm*) *take-Suc-conv-app-nth*, *assumption*)
apply (*unfold distinct-sets-prop*)
apply (*subst* (*asm*) *distinct-prop-append*)
apply *clarsimp*
done

lemma *distinct-sets-drop-nth*:
⟦*distinct-sets xs*; *i* < *length xs*; *x* ∈ *set* (*drop* (*Suc i*) *xs*)⟧ ⟹ *x* ∩ *xs* ! *i* = {}
apply (*drule* (*1*) *distinct-sets-drop*)
apply (*subst* (*asm*) *drop-Suc-nth*, *assumption*)
apply (*fastforce simp add*: *distinct-sets.simps*)
done

lemma *distinct-sets-append-distinct*:
⟦*x* ∈ *set xs*; *y* ∈ *set ys*; *distinct-sets* (*xs* @ *ys*)⟧ ⟹ *x* ∩ *y* = {}
unfolding *distinct-sets-prop* **by** (*clarsimp simp*: *distinct-prop-append*)

lemma *distinct-sets-update*:
⟦*a* ⊆ *xs* ! *i*; *distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*xs*[*i* := *a*])
apply (*subst distinct-sets-prop*)
apply (*subst* (*asm*) *distinct-sets-prop*)
apply (*subst upd-conv-take-nth-drop*, *simp*)
apply (*subst distinct-prop-append*)
apply (*intro conjI*)
apply (*erule* (*1*) *distinct-prop-take*)
apply (*rule conjI*|*clarsimp*)+
apply (*fold distinct-sets-prop*)
apply (*drule* (*1*) *distinct-sets-drop*)
apply (*subst* (*asm*) *drop-Suc-nth*, *assumption*)
apply (*fastforce simp add*: *distinct-sets.simps*)
apply (*drule* (*1*) *distinct-sets-drop*)
apply (*subst* (*asm*) *drop-Suc-nth*, *assumption*)
apply (*clarsimp simp add*: *distinct-sets.simps*)
apply *clarsimp*
apply (*rule conjI*)
apply (*drule* (*2*) *distinct-sets-take-nth*)
apply *blast*
apply *clarsimp*
apply (*thin-tac P* ⊆ *Q* **for** *P Q*)
apply (*subst* (*asm*) *id-take-nth-drop*, *assumption*)
apply (*drule distinct-sets-append-Cons*)
apply (*erule* (*2*) *distinct-sets-append-distinct*)
done

lemma *distinct-sets-map-update*:
⟦*distinct-sets* (*map f xs*); *i* < *length xs*; *f a* ⊆ *f*(*xs* ! *i*)⟧
⟹ *distinct-sets* (*map f* (*xs*[*i* := *a*]))

by (*metis distinct-sets-update length-map map-update nth-map*)

lemma *Union-list-update*:
⟦*i* < *length xs*; *distinct-sets* (*map f xs*)⟧
⟹ (⋃*x*∈*set* (*xs* [*i* := *a*]). *f x*) = (⋃*x*∈*set xs*. *f x*) − *f* (*xs* ! *i*) ∪ *f a*
apply (*induct xs arbitrary*: *i*; *clarsimp*)
subgoal for *x xs i*
apply (*cases i*; (*clarsimp*, *fastforce simp add*: *distinct-sets.simps*))
done
done

lemma *fst-enumerate*:
i < *length xs* ⟹ *fst* (*enumerate n xs* ! *i*) = *i* + *n*
by (*metis add.commute fst-conv nth-enumerate-eq*)

lemma *snd-enumerate*:
i < *length xs* ⟹ *snd* (*enumerate n xs* ! *i*) = *xs* ! *i*
by (*metis nth-enumerate-eq snd-conv*)

lemma *enumerate-member*:
assumes *i* < *length xs*
shows (*n* + *i*, *xs* ! *i*) ∈ *set* (*enumerate n xs*)
proof −
have *pair-unpack*: ⋀*a b x*. ((*a*, *b*) = *x*) = (*a* = *fst x* ∧ *b* = *snd x*) **by** *fastforce*
from *assms* **have** (*n* + *i*, *xs* ! *i*) = *enumerate n xs* ! *i*
by (*auto simp*: *fst-enumerate snd-enumerate pair-unpack*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *distinct-prop-nth*:
⟦ *distinct-prop P ls*; *n* < *n′*; *n′* < *length ls* ⟧ ⟹ *P* (*ls* ! *n*) (*ls* ! *n′*)
apply (*induct ls arbitrary*: *n n′*; *simp*)
subgoal for *l ls n n′*
apply (*cases n′*; *simp*)
apply (*cases n*; *simp*)
done
done

lemma *distinct-prop-iff*:
⟦ ⋀*x y*. *P x y* ⟷ *Q x y* ⟧ ⟹ *distinct-prop P xs* ⟷ *distinct-prop Q xs*
by (*induction xs*) *auto*

lemma *distinct-prop-impl*:
⟦ ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *P x y* ⟹ *Q x y*; *distinct-prop P xs* ⟧ ⟹
distinct-prop Q xs
by (*induction xs*) *auto*

lemma *distinct-iff-distinct-prop-ne*: *distinct xs* ⟷ *distinct-prop* (≠) *xs*
by (*induction xs*) *auto*

end

theory *WordSetup*
imports
Word-Lemmas-32-Internal
Word-Lemmas-64-Internal
Distinct-Prop
begin

lemma *distinct-prop-enum*:
⟦ ⋀*x y*. ⟦ *x* ≤ *stop*; *y* ≤ *stop*; *x* ≠ *y* ⟧ ⟹ *P x y* ⟧
⟹ *distinct-prop P* [*0* :: *'a* :: *len word .e. stop*]
apply (*simp add*: *upto-enum-def distinct-prop-map del*: *upt.simps*)
apply (*rule distinct-prop-distinct*)
apply *simp*
apply (*simp add*: *less-Suc-eq-le del*: *upt.simps*)
apply (*erule-tac x*=*of-nat x* **in** *meta-allE*)
apply (*erule-tac x*=*of-nat y* **in** *meta-allE*)
apply (*frule-tac y*=*x* **in** *unat-le*)
apply (*frule-tac y*=*y* **in** *unat-le*)
apply (*erule word-unat.Rep-cases*)+
apply (*simp add*: *toEnum-of-nat*[*OF unat-lt2p*]
word-le-nat-alt)
done

lemma *distinct-prop-enum-step*:
⟦ ⋀*x y*. ⟦ *x* ≤ *stop div step*; *y* ≤ *stop div step*; *x* ≠ *y* ⟧ ⟹ *P* (*x* ∗ *step*) (*y* ∗ *step*) ⟧
⟹ *distinct-prop P* [*0*, *step .e. stop*]
apply (*simp add*: *upto-enum-step-def distinct-prop-map*)
apply (*rule distinct-prop-enum*)
apply *simp*
done

lemmas *word-bits-def* =
Machine-Word-64-Basics.word-bits-def Machine-Word-32-Basics.word-bits-def

lemmas *word-size-def* =
Machine-Word-64-Basics.word-size-def Machine-Word-32-Basics.word-size-def

lemmas *word-bits-size* =
Machine-Word-64.word-bits-size Machine-Word-32.word-bits-size

lemmas *word-bits-len-of* =
Machine-Word-64.word-bits-len-of Machine-Word-32.word-bits-len-of

lemmas *word-bits-conv* =
Machine-Word-64-Basics.word-bits-conv Machine-Word-32-Basics.word-bits-conv

hide-const (**open**) *Machine-Word-32-Basics.word-bits*
hide-const (**open**) *Machine-Word-64-Basics.word-bits*
hide-const (**open**) *Machine-Word-32-Basics.word-size*
hide-const (**open**) *Machine-Word-64-Basics.word-size*
hide-const (**open**) *Machine-Word-32-Basics.word-size-bits*
hide-const (**open**) *Machine-Word-64-Basics.word-size-bits*

end

theory *Addr-Type-ARM*
imports
Target-Architecture
WordSetup
begin

if-architecture-context (*ARM*)
begin
type-synonym *addr-bitsize* = *32*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *32*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *2*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *19*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *13*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-32-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-32-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-32-Basics.word-size-bits*
type-synonym *char-c* = *8 word*
end

end

theory *Addr-Type-ARM64*
imports
Target-Architecture

WordSetup
begin

if-architecture-context (*ARM64*)
begin
type-synonym *addr-bitsize* = *64*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *64*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *3*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *26*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *20*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-64-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-64-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-64-Basics.word-size-bits*
type-synonym *char-c* = *8 signed word*
end

end

theory *Addr-Type-ARM-HYP*
imports
Target-Architecture
WordSetup
begin

if-architecture-context (*ARM-HYP*)
begin
type-synonym *addr-bitsize* = *32*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *32*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *2*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *19*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *13*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-32-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-32-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-32-Basics.word-size-bits*
type-synonym *char-c* = *8 word*
end

end

theory *Addr-Type-RISCV64*
imports
Target-Architecture
WordSetup
begin

if-architecture-context (*RISCV64*)
begin
type-synonym *addr-bitsize* = *64*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *64*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *3*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *26*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *20*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-64-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-64-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-64-Basics.word-size-bits*
type-synonym *char-c* = *8 word*
end

end

theory *Addr-Type-X64*
imports
Target-Architecture
WordSetup
begin

if-architecture-context (*X64*)
begin

type-synonym *addr-bitsize* = *64*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *64*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *3*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *26*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *20*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-64-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-64-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-64-Basics.word-size-bits*
type-synonym *char-c* = *8 word*
end

end

theory *Addr-Type*
imports
ARM/Addr-Type-ARM
ARM64/Addr-Type-ARM64
ARM-HYP/Addr-Type-ARM-HYP
RISCV64/Addr-Type-RISCV64
X64/Addr-Type-X64
begin

typ *addr-bitsize*
term *array-outer-max-size-exponent*
type-synonym *addr* = *addr-bitsize word*

declare *addr-align-def*[*simp*]
declare *addr-bitsize-def*[*simp*]

definition *addr-card* :: *nat* **where**
addr-card ≡ *card* (*UNIV*::*addr set*)

lemma *addr-card*:
addr-card = *2*^*addr-bitsize*
by (*simp add*: *addr-card-def card-word*)

lemma *len-of-addr-card*:
2 ^ *len-of TYPE*(*addr-bitsize*) = *addr-card*
by (*simp add*: *addr-card*)

lemma *of-nat-addr-card* [*simp*]:
of-nat addr-card = (*0*::*addr*)
by (*simp add*: *addr-card*)

end

theory *CTypesBase*
imports
Addr-Type

begin

11.3 Type setup

type-synonym *byte* = *8 word*

type-synonym *memory* = *addr* ⇒ *byte*
type-synonym *'a mem-upd* = *addr* ⇒ *'a* ⇒ *memory* ⇒ *memory*
type-synonym *'a mem-read* = *addr* ⇒ *memory* ⇒ *'a*

class *unit-class* =
  assumes *there-is-only-one*: *x* = *y*

instantiation *unit* :: *unit-class*
begin
instance by (*intro-classes*, *simp*)
end

11.3.1 Pointers

datatype *'a ptr* = *Ptr addr*

abbreviation
  NULL :: *'a ptr* **where**
  NULL ≡ *Ptr 0*

ML ‹
structure Ptr-Syntax =
struct

  val show-ptr-types = *Attrib.setup-config-bool* @{*binding show-ptr-types*} (*K true*)

  fun ptr-tr' cnst ctxt typ ts = *if Config.get ctxt show-ptr-types then*
      case Term.strip-type typ of
        ([@{*typ addr*}], *Type* (@{*type-name ptr*}, [*T*])) =>
          list-comb
            (*Syntax.const cnst* $ *Syntax-Phases.term-of-typ ctxt T*
            , *ts*)
        | - => *raise Match*
  else raise Match

  fun ptr-coerce-tr' cnst ctxt typ ts = *if Config.get ctxt show-ptr-types then*
      case Term.strip-type typ of
        ([*Type* (@{*type-name ptr*}, [*S*])], *Type* (@{*type-name ptr*}, [*T*])) =>
          list-comb
            (*Syntax.const cnst* $ *Syntax-Phases.term-of-typ ctxt S*
                                $ *Syntax-Phases.term-of-typ ctxt T*
            , *ts*)
        | - => *raise Match*

else raise Match
end
›

syntax
-Ptr :: *type* ⇒ *logic* (‹(‹*indent=1 notation=‹mixfix PTR››PTR/(1'(-'))*)›)
syntax-consts
-Ptr == *Ptr*
translations
PTR('*a*) => *CONST Ptr* :: (*addr* ⇒ '*a ptr*)
typed-print-translation
‹ [(@{*const-syntax Ptr*}, *Ptr-Syntax.ptr-tr'* @{*syntax-const -Ptr*})] ›

primrec
ptr-val :: '*a ptr* ⇒ *addr*
where
ptr-val-def: *ptr-val* (*Ptr a*) = *a*

primrec
ptr-coerce :: '*a ptr* ⇒ '*b ptr* **where**
ptr-coerce (*Ptr a*) = *Ptr a*

syntax
-Ptr-coerce :: *type* ⇒ *type* ⇒ *logic*
(‹(‹*indent=1 notation=‹mixfix PTR-COERCE››PTR'-COERCE/(‹indent=1 notation=‹infix coerce››'*(- → -'))*)›)
syntax-consts
-Ptr-coerce == *ptr-coerce*
translations
PTR-COERCE('*a* → '*b*) => *CONST ptr-coerce* :: ('*a ptr* ⇒ '*b ptr*)
typed-print-translation
‹ [(@{*const-syntax ptr-coerce*}, *Ptr-Syntax.ptr-coerce-tr'* @{*syntax-const -Ptr-coerce*})]
›

definition

ptr-less :: '*a ptr* ⇒ '*a ptr* ⇒ *bool* (**infixl** ‹<$_p$› *50*) **where**
p <$_p$ *q* ≡ *ptr-val p* < *ptr-val q*

definition
ptr-le :: '*a ptr* ⇒ '*a ptr* ⇒ *bool* (**infixl** ‹≤$_p$› *50*) **where**
p ≤$_p$ *q* ≡ *ptr-val p* ≤ *ptr-val q*

instantiation *ptr* :: (*type*) *ord*
begin

definition
ptr-less-def': *p* < *q* ≡ *p* <$_p$ *q*
definition

ptr-le-def': $p \leq q \equiv p \leq_p q$

instance ..

end

lemma *ptr-val-case*: *ptr-val p* = (*case p of Ptr v* ⇒ *v*)
by (*cases p*) *simp*

instantiation *ptr* :: (*type*) *linorder*
begin
instance
by (*intro-classes*)
(*unfold ptr-le-def' ptr-le-def ptr-less-def' ptr-less-def ptr-val-case*,
auto split: *ptr.splits*)
end

11.3.2 Raw heap

A raw map from addresses to bytes

type-synonym *heap-mem* = *addr* ⇒ *byte*

For heap *h*, pointer *p* and nat *n*, *heap-list h n p* returns the list of bytes in the heap taken from addresses $\{p..+n\}$

primrec
heap-list :: *heap-mem* ⇒ *nat* ⇒ *addr* ⇒ *byte list*
where
heap-list-base: *heap-list h 0 p* = []
| *heap-list-rec*: *heap-list h* (*Suc n*) *p* = *h p* # *heap-list h n* (*p* + *1*)

11.4 Intervals

For word *a* and nat *b*, $\{a..+b\}$ is the set of words *x*, with *unat* $(x - a) < b$.

definition
intvl :: *'a*::*len word* × *nat* ⇒ *'a*::*len word set* **where**
intvl x ≡ {*z*. ∃ *k*. *z* = *fst x* + *of-nat k* ∧ *k* < *snd x*}

abbreviation
intvl-abbr :: *'a*::*len word* ⇒ *nat* ⇒ *'a word set*
(‹(‹*open-block notation*=‹*mixfix intvl*››{-..+-})›)
where {*a*..+*b*} ≡ *intvl* (*a*,*b*)

11.5 *dt-tuple*: a reimplementation of 3 item tuples

datatype
(*'a*, *'b*, *'c*) *dt-tuple* = *DTuple* (*dt-fst*: *'a*) (*dt-snd*: *'b*) (*dt-trd*: *'c*)

lemma *dt-prj-simps*[*simp*]:
dt-fst (*DTuple a b c*) = *a*
dt-snd (*DTuple a b c*) = *b*
dt-trd (*DTuple a b c*) = *c*
by (*auto*)

lemma *split-DTuple-All*:
(∀ *x*. *P x*) = (∀ *a b c*. *P* (*DTuple a b c*))
apply (*rule iffI*; *clarsimp*)
subgoal for *x* **by** (*cases x*, *simp*)
done

lemma *surjective-dt-tuple*:
p = *DTuple* (*dt-fst p*) (*dt-snd p*) (*dt-trd p*)
by (*cases p*) *simp*

lemma *split-DTuple-all*[*no-atp*]: (⋀*x*. *PROP P x*) ≡ (⋀*a b c*. *PROP P* (*DTuple a b c*))
proof
fix *a b c*
assume ⋀*x*. *PROP P x*
then show *PROP P* (*DTuple a b c*) **.**
next
fix *x*
assume ⋀*a b c*. *PROP P* (*DTuple a b c*)
from ‹*PROP P* (*DTuple* (*dt-fst x*) (*dt-snd x*) (*dt-trd x*))› **show** *PROP P x* **by** *simp*
qed

type-synonym *normalisor* = *byte list* ⇒ *byte list*

11.6 Properties of pointers

lemma *Ptr-ptr-val* [*simp*]:
Ptr (*ptr-val p*) = *p*
by (*cases p*) *simp*

lemma *ptr-val-ptr-coerce* [*simp*]:
ptr-val (*ptr-coerce p*) = *ptr-val p*
by (*cases p*) *simp*

lemma *Ptr-ptr-coerce* [*simp*]:
Ptr (*ptr-val p*) = *ptr-coerce p*
by (*cases p*) *simp*

lemma *ptr-coerce-id* [*simp*]:
ptr-coerce p = *p*
by (*cases p*) *simp*

lemma *ptr-coerce-idem* [*simp*]:
ptr-coerce (*ptr-coerce p*) = *ptr-coerce p*
by (*cases p*) *simp*

lemma *ptr-val-inj* [*simp*]:
(*ptr-val p* = *ptr-val q*) = (*p* = *q*)
by (*cases p*, *cases q*) *auto*

lemma *ptr-coerce-NULL* [*simp*]:
(*ptr-coerce p* = *NULL*) = (*p* = *NULL*)
by (*cases p*) *simp*

lemma *NULL-ptr-val*:
(*p* = *NULL*) = (*ptr-val p* = *0*)
by (*cases p*) *simp*

lemma *ptr-NULL-conv*: *ptr-coerce NULL* = *NULL*
by *simp*

instantiation *ptr* :: (*type*) *finite*
begin
instance
by (*intro-classes*)
(*auto intro*!: *finite-code finite-imageD* [**where** *f*=*ptr-val*] *injI*)
end

11.7 Properties of the raw heap

lemma *heap-list-length* [*simp*]:
length (*heap-list h n p*) = *n*
by (*induct n arbitrary*: *p*) *auto*

lemma *heap-list-split*:
shows *k* ≤ *n* ⟹ *heap-list h n x* = *heap-list h k x* @ *heap-list h* (*n* − *k*) (*x* + *of-nat k*)
proof (*induct n arbitrary*: *k x*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*) **thus** *?case*
by (*cases k*, *auto simp*: *ac-simps*)
qed

lemma *heap-list-split2*:
heap-list h (*x* + *y*) *p* = *heap-list h x p* @ *heap-list h y* (*p* + *of-nat x*)
by (*subst heap-list-split* [**where** *k*=*x*], *auto*)

11.8 Properties of intervals

lemma *intvlI*:
$x < n \Longrightarrow p + \textit{of-nat}\ x \in \{p..+n\}$
by (*force simp*: *intvl-def*)

lemma *intvlD*:
$q \in \{p..+n\} \Longrightarrow \exists k.\ q = p + \textit{of-nat}\ k \wedge k < n$
by (*force simp*: *intvl-def*)

lemma *intvl-empty* [*simp*]:
$\{p..+0\} = \{\}$
by (*fast dest*: *intvlD*)

lemma *intvl-Suc*:
$q \in \{p..+\textit{Suc}\ 0\} \Longrightarrow p = q$
by (*force dest*: *intvlD*)

lemma *intvl-self*:
$0 < n \Longrightarrow x \in \{x..+n\}$
by (*force simp*: *intvl-def*)

lemma *intvl-start-inter*:
$[\![\ 0 < m;\ 0 < n\ ]\!] \Longrightarrow \{p..+m\} \cap \{p..+n\} \neq \{\}$
by (*force simp*: *disjoint-iff-not-equal dest*: *intvl-self*)

lemma *intvl-overflow*:
assumes *2^len-of TYPE('a)* $\leq n$
shows $\{(p::'a::\textit{len word})..+n\} = \textit{UNIV}$
proof −
have *witness*:
$\bigwedge x.\ x = p + \textit{of-nat}\ (\textit{unat}\ (x - p)) \wedge \textit{unat}\ (x - p) < n$
using *assms* **by** *simp unat-arith*
show *?thesis* **unfolding** *intvl-def* **by** (*auto intro*!: *witness*)
qed

lemma *intvl-self-offset*:
fixes $p::'a::\textit{len word}$
assumes *a*: *2^len-of TYPE('a)* $- n < x$ **and** *b*: $x <$ *2^len-of TYPE('a)* **and**
c: $(p::'a::\textit{len word}) \notin \{p + \textit{of-nat}\ x..+n\}$
shows *False*
proof −
let *?j* = *2^len-of TYPE('a)* $- x$
from *b* **have** *b'*: *of-nat x + of-nat ?j* $= (0::'a::\textit{len word})$ **using** *of-nat-2p* **by** *auto*
moreover from *a b* **have** *?j* $< n$ **by** *arith*
with *b b' c* **show** *?thesis* **by** (*force simp*: *intvl-def*)
qed

lemma *intvl-mem-offset*:
⟦ $q \in \{p..+unat\ x\}$; $q \notin \{p..+unat\ y\}$; $unat\ y \leq unat\ x$ ⟧ ⟹
$q \in \{p + y..+unat\ x - unat\ y\}$
apply (*clarsimp simp*: *intvl-def*)
subgoal for *k*
apply (*rule exI* [**where** *x*=*k* − *unat y*])
apply *auto*
done
done

lemma *intvl-plus-sub-offset*:
$x \in \{p + y..+q - unat\ y\} \Longrightarrow x \in \{p..+q\}$
apply (*clarsimp simp*: *intvl-def*)
subgoal for *k*
apply (*rule exI* [**where** *x*=*k* + *unat y*])
apply *auto*
done
done

lemma *intvl-plus-sub-Suc*:
$x \in \{p + 1..+q - Suc\ 0\} \Longrightarrow x \in \{p..+q\}$
by (*rule intvl-plus-sub-offset* [**where** *y*=*1*], *simp*)

lemma *intvl-neq-start*:
⟦ (*q*::'*a*::*len word*) ∈ {*p*..+*n*}; *p* ≠ *q* ⟧ ⟹ *q* ∈ {*p* + *1*..+*n* − *Suc 0*}
apply (*clarsimp simp*: *intvl-def*)
by (*metis One-nat-def Suc-eq-plus1-left add.commute gr0-conv-Suc*
less-diff-conv of-nat-Suc of-nat-gt-0)

lemmas *unat-simps'* =
word-arith-nat-defs word-unat.eq-norm len-of-addr-card mod-less

lemma *intvl-offset-nmem*:
⟦ $q \in \{(p::'a::len\ word)..+unat\ x\}$; $y \leq\ 2\hat{}len\text{-}of\ TYPE('a) - unat\ x$ ⟧ ⟹
$q \notin \{p + x..+y\}$
apply (*clarsimp simp*: *intvl-def*)
apply (*simp only*: *unat-simps'*)
apply (*subst* (*asm*) *word-unat.Abs-inject*)
apply (*auto simp*: *unats-def*)
done

lemma *intvl-Suc-nmem'* [*simp*]:
$n < 2\hat{}len\text{-}of\ TYPE('a) \Longrightarrow (p::'a::len\ word) \notin \{p + 1..+n - Suc\ 0\}$
by (*clarsimp simp*: *intvl-def*)
(*unat-arith*, *simp add*: *unat-simps' take-bit-nat-eq-self*)

lemma *intvl-Suc-nmem''*:
$n \leq 2\hat{}len\text{-}of\ TYPE('a) \Longrightarrow (p::'a::len\ word) \notin \{p + 1..+n - Suc\ 0\}$

by (*simp add*: *intvl-offset-nmem intvl-self*)

lemma *intvl-start-le*:
$x \leq y \Longrightarrow \{p..+x\} \subseteq \{p..+y\}$
by (*force simp*: *intvl-def*)

lemma *intvl-sub-eq*:
assumes $x \leq y$
shows $\{p + x..+unat\ (y - x)\} = \{p..+unat\ y\} - \{p..+unat\ x\}$
proof −
have *unat* y − *unat* $x \leq 2\ \widehat{}$ *len-of TYPE*('*a*) − *unat* x
by (*insert unat-lt2p* [*of y*], *arith*)
moreover have $x \leq y$ **by** *fact*
moreover hence *unat* $(y - x) =$ *unat* y − *unat* x
by (*simp add*: *word-le-nat-alt*, *unat-arith*)
ultimately show *?thesis*
by (*force dest*: *intvl-offset-nmem intvl-mem-offset elim*: *intvl-plus-sub-offset*
simp: *word-le-nat-alt*)

qed

lemma *intvl-disj-offset*:
$\{x + a..+c\} \cap \{x + b..+d\} = \{\} = (\{a..+c\} \cap \{b..+d\} = \{\})$
by (*force simp*: *intvl-def*)

lemma *intvl-sub-offset*:
unat $x+y \leq z \Longrightarrow \{k+x..+y\} \subseteq \{k..+z\}$
apply(*clarsimp simp*: *intvl-def*)
subgoal for k
apply(*rule exI* [**where** x=*unat* $x +\ k$])
apply *clarsimp*
done
done

lemma *disjnt-intvl-offsetp*[*simp*]:
disjnt $\{a + x\ ..+\ n\}\ \{a + y\ ..+\ m\} \longleftrightarrow$ *disjnt* $\{x\ ..+\ n\}\ \{y\ ..+\ m\}$
by (*simp add*: *disjnt-def intvl-disj-offset*)

lemma *intvl-eq-of-nat-Ico-add*: $\{$*of-nat* n::'a::*len word*..+ $m\} =$ *of-nat* ' $\{n\ ..<\ n + m\}$
by (*force simp*: *image-iff intvl-def Bex-def nat-le-iff-add simp flip*: *of-nat-add*)

lemma *intvl-le*:
assumes *n2* + *off1* ≤ *n1*
shows $\{p +$ *of-nat off1*..+*n2*$\} \subseteq \{p..+n1\}$
using *assms*
by (*auto simp add*: *intvl-def*)
(*metis add.commute add-mono-thms-linordered-field*(*1*) *less-le-trans word-of-nat-plus*)

lemma *intvl-disj-left*:
  fixes *a b* :: *addr*
  assumes *a-n*: *unat a* + *n* ≤ *unat b* **and** *b-m*: *unat b* + *m* ≤ *addr-card* + *unat a*
  shows {*a* ..+ *n*} ∩ {*b* ..+ *m*} = {}
proof (*safe dest*!: *intvlD intro*!: *empty-iff*[*THEN iffD2*])
  fix *i j* **assume** *i*: *i* < *n* **and** *j*: *j* < *m* **and** *eq*: *a* + *of-nat i* = *b* + *of-nat j*
  have *a-i-eq*: *unat* (*a* + *of-nat i*) = *unat a* + *of-nat i*
    using *a-n i*
    by (*metis* (*mono-tags*, *lifting*) *Abs-fnat-hom-add add-left-mono le-unat-uoi*
        less-or-eq-imp-le of-nat-id word-unat.Rep-inverse)
  from *a-n b-m* **have** *n-m*: *n* + *m* ≤ *addr-card*
    by *simp*
  show *False*
  proof *cases*
    assume *unat b* + *j* < *addr-card*
    then have *unat* (*b* + *of-nat j*) = *unat b* + *of-nat j*
    by (*metis len-of-addr-card of-nat-add of-nat-id unat-of-nat-eq word-unat.Rep-inverse*)
    with *eq a-i-eq* **have** *unat a* + *i* = *unat b* + *j*
      unfolding *word-unat-eq-iff* **by** *simp*
    with *a-n i* **show** *False* **by** *simp*
  next
    assume ¬ *unat b* + *j* < *addr-card*
    with *j n-m* **have** *unat* (*b* + *of-nat j*) + *addr-card* = *unat b* + *j*
      unfolding *addr-card-def card-word unat-arith-simps*(*5*)
      by (*simp add*: *unat-of-nat-eq*)
    then have *unat a* + *i* + *addr-card* = *unat b* + *j*
      using *a-i-eq eq* **unfolding** *word-unat-eq-iff* **by** *simp*
    with *b-m j* **show** *False* **by** *simp*
  qed
qed

end

theory *CTypesDefs*
imports
  CTypesBase
begin

11.9 C types setup

type-synonym *field-name* = *string*
type-synonym *qualified-field-name* = *field-name list*

type-synonym *typ-name = string*

A *typ-desc* wraps a *typ-struct* with a typ name. A *typ-struct* is either a Scalar, with size, alignment and either a field description (for *typ-info*) or a 'normalisor' (for *typ-uinfo*), or an Aggregate, with a list of *typ-desc*, field name, and field descripton (for the complete sub-structure) or unit (for *typ-uinfo*). The field description for aggregates is an extension of the original work of H. Tuch. It is used to make the construction of a new structure from nested structures / arrays more efficient. Properties like commutation of fields can be expressed and proven for the toplevel fields only, without having to re-examine the nested leafs of the tree.

datatype
('a,'b) typ-desc = TypDesc nat ('a, 'b) typ-struct typ-name
and *('a,'b) typ-struct = TypScalar nat nat 'a |*
TypAggregate (('a, 'b) typ-desc, field-name, 'b) dt-tuple list

datatype-compat *dt-tuple*
datatype-compat *typ-desc typ-struct*
print-theorems

lemma *typ-desc-induct*:
⟦⋀nat typ-struct list. P2 typ-struct ⟹ P1 (TypDesc nat typ-struct list); ⋀nat1 nat2 a. P2 (TypScalar nat1 nat2 a);
⋀list. P3 list ⟹ P2 (TypAggregate list); P3 []; ⋀dt-tuple list. ⟦P4 dt-tuple; P3 list⟧ ⟹ P3 (dt-tuple # list);
⋀typ-desc list b. P1 typ-desc ⟹ P4 (DTuple typ-desc list b)⟧
⟹ P1 typ-desc
by (*rule compat-typ-desc.induct*)

lemma *typ-struct-induct*:
⟦⋀nat typ-struct list. P2 typ-struct ⟹ P1 (TypDesc nat typ-struct list); ⋀nat1 nat2 a. P2 (TypScalar nat1 nat2 a);
⋀list. P3 list ⟹ P2 (TypAggregate list); P3 []; ⋀dt-tuple list. ⟦P4 dt-tuple; P3 list⟧ ⟹ P3 (dt-tuple # list);
⋀typ-desc list b. P1 typ-desc ⟹ P4 (DTuple typ-desc list b)⟧
⟹ P2 typ-struct
by (*rule compat-typ-struct.induct*)

lemma *typ-list-induct*:
⟦⋀nat typ-struct list. P2 typ-struct ⟹ P1 (TypDesc nat typ-struct list); ⋀nat1 nat2 a. P2 (TypScalar nat1 nat2 a);
⋀list. P3 list ⟹ P2 (TypAggregate list); P3 []; ⋀dt-tuple list. ⟦P4 dt-tuple; P3 list⟧ ⟹ P3 (dt-tuple # list);
⋀typ-desc list b. P1 typ-desc ⟹ P4 (DTuple typ-desc list b)⟧
⟹ P3 list
by (*rule compat-typ-desc-char-list-dt-tuple-list.induct*)

lemma *typ-dt-tuple-induct*:

⟦⋀*nat typ-struct list*. *P2 typ-struct* ⟹ *P1* (*TypDesc nat typ-struct list*); ⋀*nat1 nat2 a*. *P2* (*TypScalar nat1 nat2 a*);
⋀*list*. *P3 list* ⟹ *P2* (*TypAggregate list*); *P3* []; ⋀*dt-tuple list*. ⟦*P4 dt-tuple*; *P3 list*⟧ ⟹ *P3* (*dt-tuple* # *list*);
⋀*typ-desc list b*. *P1 typ-desc* ⟹ *P4* (*DTuple typ-desc list b*)⟧
⟹ *P4 dt-tuple*
by (*rule compat-typ-desc-char-list-dt-tuple.induct*)

— Declare as default induct rule with old case names
lemmas *typ-desc-typ-struct-inducts* [*case-names*
TypDesc TypScalar TypAggregate Nil-typ-desc Cons-typ-desc DTuple-typ-desc, *induct type*] =
typ-desc-induct typ-struct-induct typ-list-induct typ-dt-tuple-induct

— Make sure list induct rule is tried first
declare *list.induct* [*induct type*]

type-synonym (*'a*, *'b*) *typ-tuple* = ((*'a*, *'b*) *typ-desc*,*field-name*, *'b*) *dt-tuple*

type-synonym *typ-uinfo* = (*normalisor*, *unit*) *typ-desc*
type-synonym *typ-uinfo-struct* = (*normalisor*, *unit*) *typ-struct*
type-synonym *typ-uinfo-tuple* = (*normalisor*, *unit*) *typ-tuple*

record *'a field-desc* =
field-access :: *'a* ⇒ *byte list* ⇒ *byte list*
field-update :: *byte list* ⇒ *'a* ⇒ *'a*
field-sz :: *nat*

type-synonym (*'a*, *'b*) *typ-info* = (*'a field-desc*, *'b*) *typ-desc*
type-synonym (*'a*, *'b*) *typ-info-struct* = (*'a field-desc*, *'b*) *typ-struct*
type-synonym (*'a*, *'b*) *typ-info-tuple* = (*'a field-desc*, *'b*) *typ-tuple*

type-synonym *'a xtyp-tuple* = (*'a*, *'a*) *typ-tuple*
type-synonym *'a xtyp-info* = (*'a field-desc*, *'a field-desc*) *typ-desc*
type-synonym *'a xtyp-info-struct* = (*'a field-desc*, *'a field-desc*) *typ-struct*
type-synonym *'a xtyp-info-tuple* = *'a field-desc xtyp-tuple*

definition *fu-commutes* :: (*'b* ⇒ *'a* ⇒ *'a*) ⇒ (*'c* ⇒ *'a* ⇒ *'a*) ⇒ *bool* **where**
fu-commutes f g ≡ ∀ *v bs bs'*. *f bs* (*g bs' v*) = *g bs'* (*f bs v*)

size-td returns the sum of the sizes of all Scalar fields comprising a *typ-desc* i.e. the overall size of the type

primrec
size-td :: (*'a*, *'b*) *typ-desc* ⇒ *nat* **and**
size-td-struct :: (*'a*, *'b*) *typ-struct* ⇒ *nat* **and**
size-td-list :: (*'a*, *'b*) *typ-tuple list* ⇒ *nat* **and**
size-td-tuple :: (*'a*, *'b*) *typ-tuple* ⇒ *nat*
where
tz0: *size-td* (*TypDesc algn st nm*) = *size-td-struct st*

| *tz1*: *size-td-struct* (*TypScalar n algn d*) = *n*
| *tz2*: *size-td-struct* (*TypAggregate xs*) = *size-td-list xs*

| *tz3*: *size-td-list* [] = *0*
| *tz4*: *size-td-list* (*x*#*xs*) = *size-td-tuple x* + *size-td-list xs*

| *tz5*: *size-td-tuple* (*DTuple t n d*) = *size-td t*

access-ti overlays the byte-wise representation of an object on a given byte list, given the *typ-info* (i.e. the layout)

primrec
access-ti :: ('*a*, '*b*) *typ-info* ⇒ ('*a* ⇒ *byte list* ⇒ *byte list*) **and**
access-ti-struct :: ('*a*, '*b*) *typ-info-struct* ⇒
('*a* ⇒ *byte list* ⇒ *byte list*) **and**
access-ti-list :: ('*a*, '*b*) *typ-info-tuple list* ⇒
('*a* ⇒ *byte list* ⇒ *byte list*) **and**
access-ti-tuple :: ('*a*, '*b*) *typ-info-tuple* ⇒ ('*a* ⇒ *byte list* ⇒ *byte list*)
where
fa0: *access-ti* (*TypDesc algn st nm*) = *access-ti-struct st*

| *fa1*: *access-ti-struct* (*TypScalar n algn d*) = *field-access d*
| *fa2*: *access-ti-struct* (*TypAggregate xs*) = *access-ti-list xs*

| *fa3*: *access-ti-list* [] = (λ*v bs*. [])
| *fa4*: *access-ti-list* (*x*#*xs*) =
(λ*v bs*. *access-ti-tuple x v* (*take* (*size-td-tuple x*) *bs*) @
access-ti-list xs v (*drop* (*size-td-tuple x*) *bs*))

| *fa5*: *access-ti-tuple* (*DTuple t nm d*) = *access-ti t*

access-ti$_0$ overlays the representation of an object on a list of zero bytes

definition *access-ti*$_0$ :: ('*a*, '*b*) *typ-info* ⇒ ('*a* ⇒ *byte list*) **where**
access-ti$_0$ *t* ≡ λ*v*. *access-ti t v* (*replicate* (*size-td t*) *0*)

update-ti updates an object, given a list of bytes (the representation of the new value), and the *typ-info*

primrec
update-ti :: ('*a*, '*b*) *typ-info* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **and**
update-ti-struct :: ('*a*, '*b*) *typ-info-struct* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **and**
update-ti-list :: ('*a*, '*b*) *typ-info-tuple list* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **and**
update-ti-tuple :: ('*a*, '*b*) *typ-info-tuple* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*)
where
fu0: *update-ti* (*TypDesc algn st nm*) = *update-ti-struct st*

| *fu1*: *update-ti-struct* (*TypScalar n algn d*) = *field-update d*
| *fu2*: *update-ti-struct* (*TypAggregate xs*) = *update-ti-list xs*

| *fu3*: *update-ti-list* [] = (λ*bs*. *id*)

| *fu4*: *update-ti-list* (*x*#*xs*) = (λ*bs v*.
update-ti-tuple x (*take* (*size-td-tuple x*) *bs*)
(*update-ti-list xs* (*drop* (*size-td-tuple x*) *bs*) *v*))

| *fu5*: *update-ti-tuple* (*DTuple t nm d*) = *update-ti t*

update-ti-t updates an object only if the length of the supplied representation equals the object size

definition *update-ti-t* :: ('*a*, '*b*) *typ-info* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **where**
update-ti-t t ≡ λ*bs*. *if length bs* = *size-td t then*
update-ti t bs else id

definition *update-ti-struct-t* :: ('*a*, '*b*) *typ-info-struct* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **where**
update-ti-struct-t t ≡ λ*bs*. *if length bs* = *size-td-struct t then*
update-ti-struct t bs else id

definition *update-ti-list-t* :: ('*a*, '*b*) *typ-info-tuple list* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **where**
update-ti-list-t t ≡ λ*bs*. *if length bs* = *size-td-list t then*
update-ti-list t bs else id

definition *update-ti-tuple-t* :: ('*a*, '*b*) *typ-info-tuple* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **where**
update-ti-tuple-t t ≡ λ*bs*. *if length bs* = *size-td-tuple t then*
update-ti-tuple t bs else id

lemma *update-ti-t-struct-t* [*simp*]: *update-ti-t* (*TypDesc algn st nm*) = *update-ti-struct-t st*
apply (*rule ext*)
apply (*simp add*: *update-ti-t-def update-ti-struct-t-def*)
done

lemma *update-ti-update-ti-t*:
length bs = *size-td s* ⟹ *update-ti s bs v* = *update-ti-t s bs v*
unfolding *update-ti-t-def* **by** *simp*

field-desc generates the access/update pair for a field, given the field's *type-desc*

definition *field-desc* :: ('*a*, '*b*) *typ-info* ⇒ '*a field-desc* **where**
field-desc t ≡ (| *field-access* = *access-ti t*,
field-update = *update-ti-t t*, *field-sz* = *size-td t* |)

declare *field-desc-def* [*simp add*]

definition *field-desc-struct* :: ('*a*, '*b*) *typ-info-struct* ⇒ '*a field-desc* **where**
field-desc-struct t ≡ (| *field-access* = *access-ti-struct t*,
field-update = *update-ti-struct-t t*, *field-sz* = *size-td-struct t* |)

declare *field-desc-struct-def* [*simp add*]

definition *field-desc-list* :: ('*a*, '*b*) *typ-info-tuple list* ⇒ '*a field-desc*
where
field-desc-list t ≡ (| *field-access* = *access-ti-list t*,
field-update = *update-ti-list-t t*, *field-sz* = *size-td-list t* |)

declare *field-desc-list-def* [*simp add*]

definition *field-desc-tuple* :: ('*a*, '*b*) *typ-info-tuple* ⇒ '*a field-desc*
where
field-desc-tuple t ≡ (| *field-access* = *access-ti-tuple t*,
field-update = *update-ti-tuple-t t*, *field-sz* = *size-td-tuple t* |)

declare *field-desc-tuple-def* [*simp add*]

primrec
map-td :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒ ('*a*, '*c*) *typ-desc* ⇒ ('*b*, '*d*) *typ-desc* **and**
map-td-struct :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒ ('*a*, '*c*) *typ-struct* ⇒ ('*b*, '*d*) *typ-struct* **and**
map-td-list :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒ ('*a*, '*c*) *typ-tuple list* ⇒ ('*b*, '*d*) *typ-tuple list* **and**
map-td-tuple :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒ ('*a*, '*c*) *typ-tuple* ⇒ ('*b*, '*d*) *typ-tuple*
where
mat0: *map-td f g* (*TypDesc algn st nm*) = *TypDesc algn* (*map-td-struct f g st*) *nm*

| *mat1*: *map-td-struct f g* (*TypScalar n algn d*) = *TypScalar n algn* (*f n algn d*)
| *mat2*: *map-td-struct f g* (*TypAggregate xs*) = *TypAggregate* (*map-td-list f g xs*)

| *mat3*: *map-td-list f g* [] = []
| *mat4*: *map-td-list f g* (*x*#*xs*) = *map-td-tuple f g x* # *map-td-list f g xs*

| *mat5*: *map-td-tuple f g* (*DTuple t n d*) = *DTuple* (*map-td f g t*) *n* (*g d*)

definition *field-norm* :: *nat* ⇒ *nat* ⇒ ('*a*,'*b*) *field-desc-scheme* ⇒ (*byte list* ⇒ *byte list*)
where
field-norm ≡ λ*n algn d bs*.
if length bs = *n then*
field-access d (*field-update d bs undefined*) (*replicate n 0*) *else*
[]

definition *export-uinfo* :: ('*a*, '*b*) *typ-info* ⇒ *typ-uinfo* **where**
export-uinfo t ≡ *map-td field-norm* (λ-. ()) *t*

primrec
  wf-desc :: ('*a*, '*b*) *typ-desc* ⇒ *bool* **and**
  wf-desc-struct :: ('*a*, '*b*) *typ-struct* ⇒ *bool* **and**
  wf-desc-list :: ('*a*, '*b*) *typ-tuple list* ⇒ *bool* **and**
  wf-desc-tuple :: ('*a*, '*b*) *typ-tuple* ⇒ *bool*
where
  wfd0: *wf-desc* (*TypDesc algn ts n*) = *wf-desc-struct ts*

| *wfd1*: *wf-desc-struct* (*TypScalar n algn d*) = *True*
| *wfd2*: *wf-desc-struct* (*TypAggregate ts*) = *wf-desc-list ts*

| *wfd3*: *wf-desc-list* [] = *True*
| *wfd4*: *wf-desc-list* (*x*#*xs*) = (*wf-desc-tuple x* ∧ ¬ *dt-snd x* ∈ *dt-snd* ' *set xs* ∧
        wf-desc-list xs)

| *wfd5*: *wf-desc-tuple* (*DTuple x n d*) = *wf-desc x*

primrec
  wf-size-desc :: ('*a*, '*b*) *typ-desc* ⇒ *bool* **and**
  wf-size-desc-struct :: ('*a*, '*b*) *typ-struct* ⇒ *bool* **and**
  wf-size-desc-list :: ('*a*, '*b*) *typ-tuple list* ⇒ *bool* **and**
  wf-size-desc-tuple :: ('*a*, '*b*) *typ-tuple* ⇒ *bool*
where
  wfsd0: *wf-size-desc* (*TypDesc algn ts n*) = *wf-size-desc-struct ts*

| *wfsd1*: *wf-size-desc-struct* (*TypScalar n algn d*) = (*0* < *n*)
| *wfsd2*: *wf-size-desc-struct* (*TypAggregate ts*) =
        (*ts* ≠ [] ∧ *wf-size-desc-list ts*)

| *wfsd3*: *wf-size-desc-list* [] = *True*
| *wfsd4*: *wf-size-desc-list* (*x*#*xs*) =
        (*wf-size-desc-tuple x* ∧ *wf-size-desc-list xs*)

| *wfsd5*: *wf-size-desc-tuple* (*DTuple x n d*) = *wf-size-desc x*

definition
  typ-struct :: ('*a*, '*b*) *typ-desc* ⇒ ('*a*, '*b*) *typ-struct*
where
  typ-struct t = (*case t of TypDesc algn st sz* ⇒ *st*)

lemma *typ-struct* [*simp*]:
  typ-struct (*TypDesc algn st sz*) = *st*
  by (*simp add*: *typ-struct-def*)

primrec
  typ-name :: ('*a*, '*b*) *typ-desc* ⇒ *typ-name*
where

typ-name (*TypDesc algn st nm*) = *nm*

primrec
norm-tu :: *typ-uinfo* ⇒ *normalisor* **and**
norm-tu-struct :: *typ-uinfo-struct* ⇒ *normalisor* **and**
norm-tu-list :: *typ-uinfo-tuple list* ⇒ *normalisor* **and**
norm-tu-tuple :: *typ-uinfo-tuple* ⇒ *normalisor*
where
tn0: *norm-tu* (*TypDesc algn st nm*) = *norm-tu-struct st*

| *tn1*: *norm-tu-struct* (*TypScalar n aln f*) = *f*
| *tn2*: *norm-tu-struct* (*TypAggregate xs*) = *norm-tu-list xs*

| *tn3*: *norm-tu-list* [] = (λ*bs*. [])
| *tn4*: *norm-tu-list* (*x*#*xs*) = (λ*bs*.
norm-tu-tuple x (*take* (*size-td-tuple x*) *bs*) @
norm-tu-list xs (*drop* (*size-td-tuple x*) *bs*))

| *tn5*: *norm-tu-tuple* (*DTuple t n d*) = *norm-tu t*

class *c-type-name* =
fixes
typ-name-itself :: *'a itself* ⇒ *typ-name*

class *c-type* = *c-type-name* +
fixes
typ-info-t :: *'a itself* ⇒ *'a xtyp-info*
assumes *type-name-itself-typ-name*: *typ-name-itself T* = *typ-name* (*typ-info-t T*)

instance *c-type* ⊆ *type* **..**

definition (**in** *c-type*) *typ-uinfo-t* :: *'a itself* ⇒ *typ-uinfo* **where**
typ-uinfo-t t ≡ *export-uinfo* (*typ-info-t TYPE*(*'a*))

definition (**in** *c-type*) *to-bytes* :: *'a* ⇒ *byte list* ⇒ *byte list* **where**
to-bytes v ≡ *access-ti* (*typ-info-t TYPE*(*'a*)) *v*

definition (**in** *c-type*) *from-bytes* :: *byte list* ⇒ *'a* **where**
from-bytes bs ≡
field-update (*field-desc* (*typ-info-t TYPE*(*'a*))) *bs undefined*

type-synonym (*'a*, *'b*) *flr* = ((*'a*, *'b*) *typ-desc* × *nat*) *option*

primrec

field-lookup :: ('*a*, '*b*) *typ-desc* ⇒ *qualified-field-name* ⇒ *nat* ⇒ ('*a*, '*b*) *flr* **and**
field-lookup-struct :: ('*a*, '*b*) *typ-struct* ⇒ *qualified-field-name* ⇒ *nat* ⇒
('*a*, '*b*) *flr* **and**
field-lookup-list :: ('*a*, '*b*) *typ-tuple list* ⇒ *qualified-field-name* ⇒ *nat* ⇒
('*a*, '*b*) *flr* **and**
field-lookup-tuple :: ('*a*, '*b*) *typ-tuple* ⇒ *qualified-field-name* ⇒ *nat* ⇒ ('*a*, '*b*) *flr*
where
fl0: *field-lookup* (*TypDesc algn st nm*) *f m* =
(*if f*=[] *then Some* (*TypDesc algn st nm*,*m*) *else field-lookup-struct st f m*)

| *fl1*: *field-lookup-struct* (*TypScalar n algn d*) *f m* = *None*
| *fl2*: *field-lookup-struct* (*TypAggregate xs*) *f m* = *field-lookup-list xs f m*

| *fl3*: *field-lookup-list* [] *f m* = *None*
| *fl4*: *field-lookup-list* (*x*#*xs*) *f m* = (
case field-lookup-tuple x f m of
None ⇒ *field-lookup-list xs f* (*m* + *size-td* (*dt-fst x*)) |
Some y ⇒ *Some y*)

| *fl5*: *field-lookup-tuple* (*DTuple t nm d*) *f m* =
(*if nm*=*hd f* ∧ *f* ≠ [] *then field-lookup t* (*tl f*) *m else None*)

lemma *field-lookup-wf-desc-pres*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
wf-desc t ⟹ *field-lookup t f n* = *Some* (*s*, *m*) ⟹ *wf-desc s*
wf-desc-struct st ⟹ *field-lookup-struct st f n* = *Some* (*s*, *m*) ⟹ *wf-desc s*
wf-desc-list ts ⟹ *field-lookup-list ts f n* = *Some* (*s*, *m*) ⟹ *wf-desc s*
wf-desc-tuple x ⟹ *field-lookup-tuple x f n* = *Some* (*s*, *m*) ⟹ *wf-desc s*
by (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *n s m f* **and** *n s m f* **and** *n s m f*
and *n s m f*)
(*auto split*: *if-split-asm option.splits*)

definition *map-td-flr* :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒
(('*a*,'*c*) *typ-desc* × *nat*) *option* ⇒ ('*b*, '*d*) *flr*
where
map-td-flr f g ≡ *case-option None* (λ(*s*,*n*). *Some* (*map-td f g s*,*n*))

definition
import-flr :: (*nat* ⇒ *nat* ⇒ '*b* ⇒ '*a*) ⇒ ('*d* ⇒ '*c*) ⇒ ('*a*, '*c*) *flr* ⇒ (('*b*,'*d*)
typ-desc × *nat*) *option* ⇒ *bool*
where
import-flr f g s k ≡ *case-option* (*k*=*None*)
(λ(*s*,*m*). *case-option False* (λ(*t*,*n*). *n*=*m* ∧ *map-td f g t*=*s*) *k*)

s

definition
field-offset-untyped :: ('*a*, '*b*) *typ-desc* ⇒ *qualified-field-name* ⇒ *nat*
where
field-offset-untyped t n ≡ *snd* (*the* (*field-lookup t n 0*))

definition (**in** *c-type*)
field-offset :: '*a itself* ⇒ *qualified-field-name* ⇒ *nat*
where
field-offset t n ≡ *field-offset-untyped* (*typ-uinfo-t TYPE*('*a*)) *n*

definition (**in** *c-type*)
field-ti :: '*a itself* ⇒ *qualified-field-name* ⇀ '*a xtyp-info*
where
field-ti t n ≡ *case-option None* (*Some* ∘ *fst*)
(*field-lookup* (*typ-info-t TYPE*('*a*)) *n 0*)

definition (**in** *c-type*)
field-size :: '*a itself* ⇒ *qualified-field-name* ⇒ *nat*
where
field-size t n ≡ *size-td* (*the* (*field-ti t n*))

definition (**in** *c-type*)
field-lvalue :: '*a ptr* ⇒ *qualified-field-name* ⇒ *addr*
(‹(‹*open-block notation*=‹*mixfix field-lvalue*››&'(-→-'))›)
where
&(*p*→*f*) ≡ *ptr-val* (*p*::'*a ptr*) + *of-nat* (*field-offset TYPE*('*a*) *f*)

definition (**in** *c-type*)
size-of :: '*a itself* ⇒ *nat* **where**
size-of t ≡ *size-td* (*typ-info-t TYPE*('*a*))

lemma (**in** *c-type*) *size-of-fold*: *size-td* (*typ-info-t TYPE*('*a*)) = *size-of TYPE*('*a*)
by (*simp add*: *size-of-def*)

definition (**in** *c-type*)
norm-bytes :: '*a itself* ⇒ *normalisor* **where**
norm-bytes t ≡ *norm-tu* (*export-uinfo* (*typ-info-t t*))

definition (**in** *c-type*) *to-bytes-p* :: '*a* ⇒ *byte list* **where**
to-bytes-p v ≡ *to-bytes v* (*replicate* (*size-of TYPE*('*a*)) *0*)

definition (**in** *c-type*) *zero* ::'*a* **where**
zero ≡ *from-bytes* (*replicate* (*size-of TYPE*('*a*)) *0*)

hide-const (**open**) *zero* — mandatory qualifier: *c-type-class.zero*

syntax
-zero :: *type* ⇒ *logic* (‹(‹*indent=1 notation=‹mixfix ZERO›› ZERO/(1 '(-'))›)
syntax-consts
-zero == *c-type-class.zero*
translations
ZERO('*a*) => *CONST c-type-class.zero* :: ('*a*)

typed-print-translation ‹
let
val show-zero-types = *Attrib.setup-config-bool* @{*binding show-zero-types*} (*K true*);

fun zero-tr' ctxt typ - =
if Config.get ctxt show-zero-types then
Syntax.const @{*syntax-const -zero*} $ *Syntax-Phases.term-of-typ ctxt typ*
else
raise Match;

in [(@{*const-syntax c-type-class.zero*}, *zero-tr'*)]
end›

primrec
align-td-wo-align :: ('*a*, '*b*) *typ-desc* ⇒ *nat* **and**
align-td-wo-align-struct :: ('*a*, '*b*) *typ-struct* ⇒ *nat* **and**
align-td-wo-align-list :: ('*a*, '*b*) *typ-tuple list* ⇒ *nat* **and**
align-td-wo-align-tuple :: ('*a*, '*b*) *typ-tuple* ⇒ *nat*
where
al0: *align-td-wo-align* (*TypDesc algn st nm*) = *align-td-wo-align-struct st*

| *al1*: *align-td-wo-align-struct* (*TypScalar n algn d*) = *algn*
| *al2*: *align-td-wo-align-struct* (*TypAggregate xs*) = *align-td-wo-align-list xs*

| *al3*: *align-td-wo-align-list* [] = *0*
| *al4*: *align-td-wo-align-list* (*x*#*xs*) = *max* (*align-td-wo-align-tuple x*) (*align-td-wo-align-list xs*)

| *al5*: *align-td-wo-align-tuple* (*DTuple t n d*) = *align-td-wo-align t*

primrec
align-td :: ('*a*, '*b*) *typ-desc* ⇒ *nat* **and**
align-td-struct :: ('*a*, '*b*) *typ-struct* ⇒ *nat* **and**
align-td-list :: ('*a*, '*b*) *typ-tuple list* ⇒ *nat* **and**
align-td-tuple :: ('*a*, '*b*) *typ-tuple* ⇒ *nat*
where
al0: *align-td* (*TypDesc algn st nm*) = *algn*

| *al1*: *align-td-struct* (*TypScalar n algn d*) = *algn*
| *al2*: *align-td-struct* (*TypAggregate xs*) = *align-td-list xs*

| *al3*: *align-td-list* [] = *0*
| *al4*: *align-td-list* (*x*#*xs*) = *max* (*align-td-tuple x*) (*align-td-list xs*)

| *al5*: *align-td-tuple* (*DTuple t n d*) = *align-td t*

primrec
wf-align :: ('*a*, '*b*) *typ-desc* ⇒ *bool* **and**
wf-align-struct :: ('*a*, '*b*) *typ-struct* ⇒ *bool* **and**
wf-align-list :: ('*a*, '*b*) *typ-tuple list* ⇒ *bool* **and**
wf-align-tuple :: ('*a*, '*b*) *typ-tuple* ⇒ *bool*
where
wfal0: *wf-align* (*TypDesc algn ts n*) = (*align-td-wo-align-struct ts* ≤ *algn* ∧
align-td-struct ts ≤ *algn* ∧
wf-align-struct ts)

| *wfal1*: *wf-align-struct* (*TypScalar n algn d*) = *True*
| *wfal2*: *wf-align-struct* (*TypAggregate ts*) = (*wf-align-list ts*)

| *wfal3*: *wf-align-list* [] = *True*
| *wfal4*: *wf-align-list* (*x*#*xs*) =
(*wf-align-tuple x* ∧ *wf-align-list xs*)

| *wfal5*: *wf-align-tuple* (*DTuple x n d*) = *wf-align x*

definition (**in** *c-type*) *align-of* :: '*a itself* ⇒ *nat* **where**
align-of t ≡ *2*^(*align-td* (*typ-info-t TYPE*('*a*)))

lemma *align-td-wo-align-le-align-td*:
fixes *t*::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
fs::('*a*,'*b*) *typ-info-tuple list* **and**
f::('*a*,'*b*) *typ-info-tuple*
shows
wf-align t ⟹ *align-td-wo-align t* ≤ *align-td t*
wf-align-struct st ⟹ *align-td-wo-align-struct st* ≤ *align-td-struct st*
wf-align-list fs ⟹ *align-td-wo-align-list fs* ≤ *align-td-list fs*
wf-align-tuple f ⟹ *align-td-wo-align-tuple f* ≤ *align-td-tuple f*
apply (*induct t* **and** *st* **and** *fs* **and** *f rule*: *typ-desc-typ-struct-inducts*)
apply *auto*
done

lemma (**in** *c-type*) *align-td-wo-align-le-align-of*:
assumes *wf*: *wf-align* (*typ-info-t TYPE*('*a*))
shows *2*^(*align-td-wo-align* (*typ-info-t TYPE*('*a*))) ≤ *align-of* (*TYPE*('*a*))
using *align-td-wo-align-le-align-td* (*1*) [*OF wf*]
by (*simp add*: *align-of-def*)

definition (**in** *c-type*)
ptr-add :: *'a ptr* ⇒ *int* ⇒ *'a ptr* (**infixl** ‹+$_p$› *65*)
where
ptr-add (*a* :: *'a ptr*) *w* ≡
Ptr (*ptr-val a* + *of-int w* ∗ *of-nat* (*size-of* (*TYPE*(*'a*))))

lemma (**in** *c-type*) *ptr-add-def'*:
ptr-add (*Ptr p* :: (*'a*) *ptr*) *n*
= (*Ptr* (*p* + *of-int n* ∗ *of-nat* (*size-of TYPE*(*'a*))))
by (*cases p*, *auto simp*: *ptr-add-def scast-id*)

definition (**in** *c-type*)
ptr-sub :: *'a ptr* ⇒ *'a ptr* ⇒ *addr-bitsize signed word* (**infixl** ‹−$_p$› *65*)
where
ptr-sub (*a* :: *'a ptr*) *p* ≡
ucast (*ptr-val a* − *ptr-val p*) *div of-nat* (*size-of* (*TYPE*(*'a*)))

definition (**in** *c-type*) *ptr-aligned* :: *'a ptr* ⇒ *bool* **where**
ptr-aligned p ≡ *align-of TYPE*(*'a*) *dvd unat* (*ptr-val* (*p*::*'a ptr*))

type-synonym *'a ptr-guard* = *'a ptr* ⇒ *bool*

definition (**in** *c-type*) *c-null-guard* :: *'a ptr-guard* **where**
c-null-guard ≡ λ*p*. *0* ∉ {*ptr-val p*..+*size-of TYPE*(*'a*)}

definition (**in** *c-type*) *c-guard* :: *'a ptr-guard* **where**
c-guard ≡ λ*p*. *ptr-aligned p* ∧ *c-null-guard p*

primrec
td-set :: (*'a*, *'b*) *typ-desc* ⇒ *nat* ⇒ ((*'a*, *'b*) *typ-desc* × *nat*) *set* **and**
td-set-struct :: (*'a*, *'b*) *typ-struct* ⇒ *nat* ⇒ ((*'a*, *'b*) *typ-desc* × *nat*) *set* **and**
td-set-list :: (*'a*, *'b*) *typ-tuple list* ⇒ *nat* ⇒ ((*'a*,*'b*) *typ-desc* × *nat*) *set* **and**
td-set-tuple :: (*'a*, *'b*) *typ-tuple* ⇒ *nat* ⇒ ((*'a*, *'b*) *typ-desc* × *nat*) *set*
where
ts0: *td-set* (*TypDesc algn st nm*) *m* = {(*TypDesc algn st nm*,*m*)} ∪ *td-set-struct st m*

| *ts1*: *td-set-struct* (*TypScalar n algn d*) *m* = {}
| *ts2*: *td-set-struct* (*TypAggregate xs*) *m* = *td-set-list xs m*

| *ts3*: *td-set-list* [] *m* = {}
| *ts4*: *td-set-list* (*x*#*xs*) *m* = *td-set-tuple x m* ∪ *td-set-list xs* (*m* + *size-td* (*dt-fst x*))

| *ts5*: *td-set-tuple* (*DTuple t nm d*) *m* = *td-set t m*

instantiation *typ-desc* :: (*type*, *type*) *ord*

begin

definition
typ-tag-le-def: $s \leq (t::('a, 'b)\ typ\text{-}desc) \equiv (\exists n.\ (s,n) \in td\text{-}set\ t\ 0)$
definition
typ-tag-lt-def: $s < (t::('a, 'b)\ typ\text{-}desc) \equiv s \leq t \wedge s \neq t$
instance ..

end

definition
fd-cons-double-update :: $('a, 'x)$ *field-desc-scheme* $\Rightarrow$ *bool*
where
fd-cons-double-update d $\equiv$
$(\forall v\ bs\ bs'.\ length\ bs = length\ bs' \longrightarrow field\text{-}update\ d\ bs\ (field\text{-}update\ d\ bs'\ v) = field\text{-}update\ d\ bs\ v)$

definition
fd-cons-update-access :: $('a, 'x)$ *field-desc-scheme* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
fd-cons-update-access d n $\equiv$
$(\forall v\ bs.\ length\ bs = n \longrightarrow field\text{-}update\ d\ (field\text{-}access\ d\ v\ bs)\ v = v)$

definition
norm-desc :: $('a, 'x)$ *field-desc-scheme* $\Rightarrow$ *nat* $\Rightarrow$ (*byte list* $\Rightarrow$ *byte list*)
where
norm-desc d n $\equiv \lambda bs.\ field\text{-}access\ d\ (field\text{-}update\ d\ bs\ undefined)\ (replicate\ n\ 0)$

definition
fd-cons-length :: $('a, 'x)$ *field-desc-scheme* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
fd-cons-length d n $\equiv \forall v\ bs.\ length\ bs = n \longrightarrow length\ (field\text{-}access\ d\ v\ bs) = n$

definition
fd-cons-access-update :: $('a, 'x)$ *field-desc-scheme* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
fd-cons-access-update d n $\equiv \forall bs\ bs'\ v\ v'.\ length\ bs = n \longrightarrow$
$length\ bs' = n \longrightarrow$
$field\text{-}access\ d\ (field\text{-}update\ d\ bs\ v)\ bs' = field\text{-}access\ d\ (field\text{-}update\ d\ bs\ v')\ bs'$

definition
fd-cons-update-normalise :: $('a, 'x)$ *field-desc-scheme* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
fd-cons-update-normalise d n $\equiv$
$(\forall v\ bs.\ length\ bs{=}n \longrightarrow field\text{-}update\ d\ (norm\text{-}desc\ d\ n\ bs)\ v = field\text{-}update\ d\ bs\ v)$

definition
fd-cons-desc :: ('a,'x) *field-desc-scheme* ⇒ *nat* ⇒ *bool*
where
fd-cons-desc d n ≡ *fd-cons-double-update d* ∧
fd-cons-update-access d n ∧
fd-cons-access-update d n ∧
fd-cons-length d n

definition
fd-cons :: ('a, 'b) *typ-info* ⇒ *bool*
where
fd-cons t ≡ *fd-cons-desc* (*field-desc t*) (*size-td t*)

definition
fd-cons-struct :: ('a, 'b) *typ-info-struct* ⇒ *bool*
where
fd-cons-struct t ≡ *fd-cons-desc* (*field-desc-struct t*) (*size-td-struct t*)

definition
fd-cons-list :: ('a, 'b) *typ-info-tuple list* ⇒ *bool*
where
fd-cons-list t ≡ *fd-cons-desc* (*field-desc-list t*) (*size-td-list t*)

definition
fd-cons-tuple :: ('a, 'b) *typ-info-tuple* ⇒ *bool*
where
fd-cons-tuple t ≡ *fd-cons-desc* (*field-desc-tuple t*) (*size-td-tuple t*)

definition
fa-fu-ind :: 'a *field-desc* ⇒ 'a *field-desc* ⇒ *nat* ⇒ *nat* ⇒*bool*
where
fa-fu-ind d d' n n' ≡ ∀ *v bs bs'*. *length bs* = *n* ⟶ *length bs'* = *n'* ⟶
field-access d (*field-update d' bs v*) *bs'* = *field-access d v bs'*

definition
wf-fdp :: (('a, 'b) *typ-info* × *qualified-field-name*) *set* ⇒ *bool*
where
wf-fdp t ≡ ∀ *x m*. (*x*,*m*) ∈ *t* ⟶ (*fd-cons x* ∧ (∀ *y n*. (*y*,*n*) ∈ *t* ∧ ¬ *m* ≤ *n* ∧ ¬ *n* ≤ *m*
⟶ *fu-commutes* (*field-update* (*field-desc x*)) (*field-update* (*field-desc y*)) ∧
fa-fu-ind (*field-desc x*) (*field-desc y*) (*size-td y*) (*size-td x*)))

lemma *wf-fdp-list*:
wf-fdp (*xs* ∪ *ys*) ⟹ *wf-fdp xs* ∧ *wf-fdp ys*
by (*auto simp*: *wf-fdp-def*)

primrec
wf-fd :: (*'a*, *'b*) *typ-info* ⇒ *bool* **and**
wf-fd-struct :: (*'a*, *'b*) *typ-info-struct* ⇒ *bool* **and**
wf-fd-list :: (*'a*, *'b*) *typ-info-tuple list* ⇒ *bool* **and**
wf-fd-tuple :: (*'a*, *'b*) *typ-info-tuple* ⇒ *bool*
where
wffd0: *wf-fd* (*TypDesc algn ts n*) = (*wf-fd-struct ts*)

| *wffd1*: *wf-fd-struct* (*TypScalar n algn d*) = *fd-cons-struct* (*TypScalar n algn d*::(*'a*,*'b*) *typ-info-struct*)
| *wffd2*: *wf-fd-struct* (*TypAggregate ts*) = *wf-fd-list ts*

| *wffd3*: *wf-fd-list* [] = *True*
| *wffd4*: *wf-fd-list* (*x*#*xs*) = (*wf-fd-tuple x* ∧ *wf-fd-list xs* ∧
fu-commutes (*update-ti-tuple-t x*) (*update-ti-list-t xs*) ∧
fa-fu-ind (*field-desc-tuple x*) (*field-desc-list xs*) (*size-td-list xs*) (*size-td-tuple x*)∧
fa-fu-ind (*field-desc-list xs*) (*field-desc-tuple x*) (*size-td-tuple x*) (*size-td-list xs*))

| *wffd5*: *wf-fd-tuple* (*DTuple x n d*) = *wf-fd x*

definition
tf-set :: (*'a*, *'b*) *typ-info* ⇒ ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set*
where
tf-set td ≡ {(*s*,*f*) | *s f*. ∃ *n*. *field-lookup td f 0* = *Some* (*s*,*n*)}

definition
tf-set-struct :: (*'a*, *'b*) *typ-info-struct* ⇒ ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set*
where
tf-set-struct td ≡ {(*s*,*f*) | *s f*. ∃ *n*. *field-lookup-struct td f 0* = *Some* (*s*,*n*)}

definition
tf-set-list :: (*'a*, *'b*) *typ-info-tuple list* ⇒ ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set*
where
tf-set-list td ≡ {(*s*,*f*) | *s f*. ∃ *n*. *field-lookup-list td f 0* = *Some* (*s*,*n*)}

definition
tf-set-tuple :: (*'a*, *'b*) *typ-info-tuple* ⇒ ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set*
where
tf-set-tuple td ≡ {(*s*,*f*) | *s f*. ∃ *n*. *field-lookup-tuple td f 0* = *Some* (*s*,*n*)}

record *'a leaf-desc* =
lf-fd :: *'a field-desc*
lf-sz :: *nat*

lf-fn :: *qualified-field-name*

primrec
lf-set :: (*'a*, *'b*) *typ-info* ⇒ *qualified-field-name* ⇒ *'a leaf-desc set* **and**
lf-set-struct :: (*'a*, *'b*) *typ-info-struct* ⇒ *qualified-field-name* ⇒ *'a leaf-desc set*
and
lf-set-list :: (*'a*, *'b*) *typ-info-tuple list* ⇒ *qualified-field-name* ⇒ *'a leaf-desc set*
and
lf-set-tuple :: (*'a*, *'b*) *typ-info-tuple* ⇒ *qualified-field-name* ⇒ *'a leaf-desc set*
where
fds0: *lf-set* (*TypDesc algn st nm*) *fn* = *lf-set-struct st fn*

| *fds1*: *lf-set-struct* (*TypScalar n algn d*) *fn* = {(⦇ *lf-fd* = *d*, *lf-sz* = *n*, *lf-fn* = *fn* ⦈)}
| *fds2*: *lf-set-struct* (*TypAggregate xs*) *fn* = *lf-set-list xs fn*

| *fds3*: *lf-set-list* [] *fn* = {}
| *fds4*: *lf-set-list* (*x*#*xs*) *fn* = *lf-set-tuple x fn* ∪ *lf-set-list xs fn*

| *fds5*: *lf-set-tuple* (*DTuple t n d*) *fn* = *lf-set t* (*fn*@[*n*])

definition
wf-lf :: *'a leaf-desc set* ⇒ *bool*
where
wf-lf D ≡ ∀ *x*. *x* ∈ *D* ⟶ (*fd-cons-desc* (*lf-fd x*) (*lf-sz x*) ∧ (∀ *y*. *y* ∈ *D* ⟶ *lf-fn y* ≠ *lf-fn x*
⟶ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) ∧
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*)))

definition
ti-ind :: *'a leaf-desc set* ⇒ *'a leaf-desc set* ⇒ *bool*
where
ti-ind X Y ≡ ∀ *x y*. *x* ∈ *X* ∧ *y* ∈ *Y* ⟶ (
fu-commutes (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) ∧
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) ∧
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*))

definition
t2d :: ((*'a*, *'b*) *typ-info* × *qualified-field-name*) ⇒ *'a leaf-desc*
where
t2d x ≡ ⦇ *lf-fd* = *field-desc* (*fst x*), *lf-sz* = *size-td* (*fst x*), *lf-fn* = *snd x*⦈

definition
fd-consistent :: (*'a*, *'b*) *typ-info* ⇒ *bool*
where

fd-consistent t ≡ ∀ *f s n. field-lookup t f 0 = Some* (*s*,*n*)
⟶ *fd-cons s*

class *wf-type* = *c-type* +
assumes *wf-desc* [*simp*]: *wf-desc* (*typ-info-t TYPE*('*a*))
assumes *wf-size-desc* [*simp*]: *wf-size-desc* (*typ-info-t TYPE*('*a*))
assumes *wf-lf* [*simp*]: *wf-lf* (*lf-set* (*typ-info-t TYPE*('*a*)) [])

definition
super-update-bs :: *byte list* ⇒ *byte list* ⇒ *nat* ⇒ *byte list*
where
super-update-bs v bs n ≡ *take n bs* @ *v* @
drop (*n* + *length v*) *bs*

definition
disj-fn :: *qualified-field-name* ⇒ *qualified-field-name* ⇒ *bool*
where
disj-fn s t ≡ ¬ *s* ≤ *t* ∧ ¬ *t* ≤ *s*

definition
fs-path :: *qualified-field-name list* ⇒ *qualified-field-name set*
where
fs-path xs ≡ {*x*. ∃ *k*. *k* ∈ *set xs* ∧ *x* ≤ *k*} ∪ {*x*. ∃ *k*. *k* ∈ *set xs* ∧ *k* ≤ *x*}

definition
field-names :: ('*a*, '*b*) *typ-desc* ⇒ *qualified-field-name set*
where
field-names t ≡ {*f*. *field-lookup t f 0* ≠ *None*}

definition
align-field :: ('*a*, '*b*) *typ-desc* ⇒ *bool*
where
align-field ti ≡ ∀ *f s n. field-lookup ti f 0 = Some* (*s*,*n*) ⟶
2^(*align-td s*) *dvd n*

class *mem-type-sans-size* = *wf-type* +
assumes *upd*:
length bs = size-of TYPE('*a*) ⟶
update-ti-t (*typ-info-t TYPE*('*a*)) *bs v*
= *update-ti-t* (*typ-info-t TYPE*('*a*)) *bs w*
assumes *align-size-of*: *align-of* (*TYPE*('*a*)) *dvd size-of TYPE*('*a*)
assumes *align-field*: *align-field* (*typ-info-t TYPE*('*a*))
assumes *wf-align*: *wf-align* (*typ-info-t TYPE*('*a*))

class *mem-type* = *mem-type-sans-size* +
assumes *max-size*: *size-of* (*TYPE*('*a*)) < *addr-card*

primrec
  aggregate :: (*'a*, *'b*) *typ-desc* ⇒ *bool* **and**
  aggregate-struct :: (*'a*, *'b*) *typ-struct* ⇒ *bool*
where
  aggregate (*TypDesc algn st tn*) = *aggregate-struct st*

| *aggregate-struct* (*TypScalar n algn d*) = *False*
| *aggregate-struct* (*TypAggregate ts*) = *True*

class *simple-mem-type* = *mem-type* +
  assumes *simple-tag*: ¬ *aggregate* (*typ-info-t TYPE*(*'a*))

definition
  field-of :: *addr* ⇒ (*'a*, *'b*) *typ-desc* ⇒ (*'a*, *'b*) *typ-desc* ⇒ *bool*
where
  field-of q s t ≡ (*s*,*unat q*) ∈ *td-set t 0*

definition (**in** *c-type*)
  field-of-t :: *'a ptr* ⇒ *'b*::*c-type ptr* ⇒ *bool*
where
  field-of-t p q ≡ *field-of* (*ptr-val p* − *ptr-val q*) (*typ-uinfo-t TYPE*(*'a*))
      (*typ-uinfo-t TYPE*(*'b*))

definition (**in** *c-type*)
  h-val :: *heap-mem* ⇒ *'a ptr* ⇒ *'a*
where
  h-val h ≡ λ*p*. *from-bytes* (*heap-list h* (*size-of TYPE* (*'a*))
      (*ptr-val* (*p*::*'a ptr*)))

primrec
  heap-update-list :: *addr* ⇒ *byte list* ⇒ *heap-mem* ⇒ *heap-mem*
where
  heap-update-list-base: *heap-update-list p* [] *h* = *h*
| *heap-update-list-rec*:
  heap-update-list p (*x*#*xs*) *h* = *heap-update-list* (*p* + *1*) *xs* (*h*(*p*:= *x*))

type-synonym *'a typ-heap-g* = *'a ptr* ⇒ *'a*

definition (**in** *c-type*)
  lift :: *heap-mem* ⇒ *'a typ-heap-g*
where
  lift h ≡ *h-val h*

definition (**in** *c-type*)
  heap-update :: *'a ptr* ⇒ *'a* ⇒ *heap-mem* ⇒ *heap-mem*
where
  heap-update p v h ≡ *heap-update-list* (*ptr-val p*) (*to-bytes v* (*heap-list h* (*size-of* *TYPE*(*'a*)) (*ptr-val p*))) *h*

definition (**in** *c-type*)
heap-update-padding :: *'a ptr* ⇒ *'a* ⇒ *byte list* ⇒ *heap-mem* ⇒ *heap-mem* **where**
heap-update-padding p v bs h =
heap-update-list (*ptr-val p*) (*to-bytes v bs*) *h*

lemma (**in** *c-type*) *heap-update-heap-update-padding-conv*:
heap-update p v h = *heap-update-padding p v* (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*)) *h*
by (*simp add*: *heap-update-def heap-update-padding-def*)

definition (**in** *c-type*) *heap-upd* **where**
heap-upd p f s = *heap-update p* (*f* (*h-val s p*)) *s*

definition *heap-upd-list* :: *nat* ⇒ *addr* ⇒ (*byte list* ⇒ *byte list*) ⇒ *heap-mem* ⇒ *heap-mem* **where**
heap-upd-list n p f h = *heap-update-list p* (*f* (*heap-list h n p*)) *h*

fun
fold-td' :: (*typ-name* ⇒ (*'a* × *field-name*) *list* ⇒ *'a*) × (*'a*, *'b*) *typ-desc* ⇒ *'a*
where
fot0: *fold-td'* (*f*,*TypDesc algn st nm*) = (*case st of*
TypScalar n algn d ⇒ *d* |
TypAggregate ts ⇒ *f nm* (*map* (λ*x*. *case x of DTuple t n d* ⇒ (*fold-td'* (*f*,*t*),*n*)) *ts*))

definition
fold-td :: (*typ-name* ⇒ (*'a* × *field-name*) *list* ⇒ *'a*) ⇒ (*'a*, *'b*) *typ-desc* ⇒ *'a*
where
fold-td ≡ λ*f t*. *fold-td'* (*f*,*t*)

declare *fold-td-def* [*simp*]

definition
fold-td-struct :: *typ-name* ⇒ (*typ-name* ⇒ (*'a* × *field-name*) *list* ⇒ *'a*) ⇒ (*'a*, *'b*) *typ-struct* ⇒ *'a*
where
fold-td-struct tn f st ≡ (*case st of*
TypScalar n algn d ⇒ *d* |
TypAggregate ts ⇒ *f tn* (*map* (λ*x*. *case x of DTuple t n d* ⇒ (*fold-td'* (*f*,*t*),*n*)) *ts*))

declare *fold-td-struct-def* [*simp*]

definition
fold-td-list :: *typ-name* ⇒ (*typ-name* ⇒ (*'a* × *field-name*) *list* ⇒ *'a*) ⇒ (*'a*, *'b*) *typ-tuple list* ⇒ *'a*
where

fold-td-list tn f ts ≡ *f tn* (*map* (λ*x*. *case x of DTuple t n d* ⇒ (*fold-td'* (*f*,*t*),*n*)) *ts*)

declare *fold-td-list-def* [*simp*]

definition
fold-td-tuple :: (*typ-name* ⇒ ('*a* × *field-name*) *list* ⇒ '*a*) ⇒ ('*a*, '*b*) *typ-tuple* ⇒ '*a*
where
fold-td-tuple f x ≡ (*case x of DTuple t n d* ⇒ *fold-td'* (*f*,*t*))

declare *fold-td-tuple-def* [*simp*]

fun
map-td' :: ((*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) × ('*c* ⇒ '*d*)) × ('*a*, '*c*) *typ-desc* ⇒ ('*b*, '*d*) *typ-desc*
where
map-td' ((*f*,*g*),*TypDesc algn st nm*) = (*TypDesc algn* (*case st of*
TypScalar n algn d ⇒ *TypScalar n algn* (*f n algn d*) |
TypAggregate ts ⇒ *TypAggregate* (*map* (λ*x*. *case x of DTuple t n d* ⇒ *DTuple* (*map-td'* ((*f*,*g*),*t*)) *n* (*g d*)) *ts*)) *nm*)

definition
tnSum :: *typ-name* ⇒ (*nat* × *field-name*) *list* ⇒ *nat*
where
tnSum ≡ λ*tn ts*. *foldr* ((+) *o fst*) *ts 0*

definition
tnMax :: *typ-name* ⇒ (*nat* × *field-name*) *list* ⇒ *nat*
where
tnMax ≡ λ*tn ts*. *foldr* (λ*x y*. *max* (*fst x*) *y*) *ts 0*

definition
wfd :: *typ-name* ⇒ (*bool* × *field-name*) *list* ⇒ *bool*
where
wfd ≡ λ*tn ts*. *distinct* (*map snd ts*) ∧ *foldr* (∧) (*map fst ts*) *True*

definition
wfsd :: *typ-name* ⇒ (*bool* × *field-name*) *list* ⇒ *bool*
where
wfsd ≡ λ*tn ts*. *ts* ≠ [] ∧ *foldr* (∧) (*map fst ts*) *True*

definition *component-desc-tuple t* ≡ *case t of* (*DTuple t n d*) ⇒ *d*

lemma *component-desc-tuple-simp* [*simp*]: *component-desc-tuple* (*DTuple t n d*) = *d*
by (*simp add*: *component-desc-tuple-def*)

primrec *split-list*:: *nat list* ⇒ *'a list* ⇒ *'a list list* **where**
  split-list [] *bs* = [*bs*]
| *split-list* (*n*#*ns*) *bs* = *take n bs* # *split-list ns* (*drop n bs*)

lemma *concat-split* [*simp*]: *concat* (*split-list ns bs*) = *bs*
  by (*induct ns arbitrary*: *bs*) *auto*

primrec *split-map'*:: (*'b* ⇒ *nat*) ⇒ (*'b* ⇒ *'a list* ⇒ *'c*) ⇒ *'b list* ⇒ *'a list* ⇒ *'c list* **where**
  split-map' n f [] *bs* = []
| *split-map' n f* (*x*#*xs*) *bs* = *f x* (*take* (*n x*) *bs*) # *split-map' n f xs* (*drop* (*n x*) *bs*)

lemma *length-split-map'* [*simp*]: *length* (*split-map' n f xs bs*) = *length xs*
  by (*induct xs arbitrary*: *bs*) *auto*

lemma *split-map' n f xs bs* = *map* (λ(*f'*,*bs'*). *f' bs'*) (*zip* (*map f xs*) (*split-list* (*map n xs*) *bs*))
  by (*induct xs arbitrary*: *bs*) *auto*

primrec *split-map*:: (*'b* ⇒ *'a list* ⇒ *'c list*) ⇒ *'b list* ⇒ *'a list* ⇒ *'c list list* **where**
  split-map f [] *bs* = []
| *split-map f* (*x*#*xs*) *bs* = *f x bs* # *split-map f xs* (*drop* (*length* (*f x bs*)) *bs*)

lemma *length-split-map* [*simp*]: *length* (*split-map f xs bs*) = *length xs*
  by (*induct xs arbitrary*: *bs*) *auto*

primrec *split-fold*:: (*'b* ⇒ *'a list* ⇒ *'d* ⇒ (*'d* × *'a list*)) ⇒ *'b list* ⇒ *'a list* ⇒ *'d* ⇒ (*'d* × *'a list*) **where**
  split-fold f [] *bs s* = (*s*, *bs*)
| *split-fold f* (*x*#*xs*) *bs s* = (*let* (*s'*, *bs'*) = *f x bs s in split-fold f xs bs' s'*)

definition *apply-field-update*::*'a field-desc* ⇒ *byte list* ⇒ *'a* ⇒ (*'a* × *byte list*) **where**
  apply-field-update d bs s ≡ (*field-update d bs s*, *drop* (*field-sz d*) *bs*)

definition *apply-field-updates* :: *'a field-desc list* ⇒ *byte list* ⇒ *'a* ⇒ (*'a* × *byte list*) **where**
  apply-field-updates ≡ *split-fold apply-field-update*

lemma *apply-field-updates-Nil* [*simp*]: *apply-field-updates* [] *bs s* = (*s*, *bs*)
  by (*simp add*: *apply-field-updates-def*)

lemma *apply-field-updates-Cons* [*simp*]: *apply-field-updates* (*d*#*ds*) *bs s* =
  apply-field-updates ds (*drop* (*field-sz d*) *bs*) (*field-update d bs s*)
  by (*simp add*: *apply-field-updates-def apply-field-update-def*)

definition *component-access-tuple v t bs* ≡ *case t of* (*DTuple t n d*) ⇒ *field-access d v bs*

lemma *component-access-tuple-simp* [*simp*]: *component-access-tuple v* (*DTuple t n d*) = *field-access d v*
by (*auto simp add*: *component-access-tuple-def*)

definition *component-access-struct st v bs* =
(*case st of TypScalar - - d* ⇒ *field-access d v bs*
| *TypAggregate fs* ⇒ *concat* (*split-map* (*component-access-tuple v*) *fs bs*))

lemma *component-access-struct-scalar* [*simp*]:
component-access-struct (*TypScalar sz align d*) = *field-access d*
apply (*rule ext*)
apply (*rule ext*)
apply (*simp add*: *component-access-struct-def*)
done

lemma *component-access-struct-aggregate* [*simp*]:
component-access-struct (*TypAggregate fs*) *v bs* = *concat* (*split-map* (*component-access-tuple v*) *fs bs*)
by (*simp add*: *component-access-struct-def*)

lemma *component-access-struct-aggregate-nil* [*simp*]:
component-access-struct (*TypAggregate* []) = (λ*v bs*. [])
apply (*rule ext*)
apply (*rule ext*)
apply *simp*
done

definition *component-update-tuple t bs* ≡ *case t of* (*DTuple t n d*) ⇒ *field-update d bs*

lemma *component-update-tuple-simp* [*simp*]: *component-update-tuple* (*DTuple t n d*) = *field-update d*
by (*auto simp add*: *component-update-tuple-def*)

definition *component-update-struct st bs v* =
(*case st of TypScalar - - d* ⇒ *field-update d bs v*
| *TypAggregate fs* ⇒ *fst* (*apply-field-updates* (*map dt-trd fs*) *bs v*))

lemma *component-update-struct-scalar* [*simp*]:
component-update-struct (*TypScalar sz align d*) = *field-update d*
apply (*rule ext*)
apply (*rule ext*)

apply (*simp add*: *component-update-struct-def*)
done

lemma *component-update-struct-aggregate* [*simp*]:
component-update-struct (*TypAggregate fs*) *bs v* = *fst* (*apply-field-updates* (*map dt-trd fs*) *bs v*)
by (*simp add*: *component-update-struct-def*)

lemma *component-update-struct-aggregate-nil* [*simp*]:
component-update-struct (*TypAggregate* []) = (λ*bs*. *id*)
apply (*rule ext*)
apply (*rule ext*)
apply *simp*
done

definition *component-sz-struct st* =
(*case st of TypScalar - - d* ⇒ *field-sz d*
| *TypAggregate fs* ⇒ *foldl* (+) *0* (*map* (*field-sz o dt-trd*) *fs*))

lemma *component-sz-struct-scalar* [*simp*]:
component-sz-struct (*TypScalar sz align d*) = *field-sz d*
by (*simp add*: *component-sz-struct-def*)

lemma *component-sz-struct-aggregate* [*simp*]:
component-sz-struct (*TypAggregate fs*) = *foldl* (+) *0* (*map* (*field-sz o dt-trd*) *fs*)
by (*simp add*: *component-sz-struct-def*)

definition *component-desc-struct st* =
(|*field-access* = *component-access-struct st*,
field-update = *component-update-struct st*,
field-sz = *component-sz-struct st*|)

lemma *component-desc-struct-simps* [*simp*]:
field-access (*component-desc-struct st*) = *component-access-struct st*
field-update (*component-desc-struct st*) = *component-update-struct st*
field-sz (*component-desc-struct st*) = *component-sz-struct st*
component-desc-struct (*TypScalar sz align d*) = *d*
by (*auto simp add*: *component-desc-struct-def*)

definition *component-desc t* ≡ *case t of TypDesc algn st n* ⇒ *component-desc-struct st*

lemma *component-desc-simps* [*simp*]: *component-desc* (*TypDesc algn st n*) = *component-desc-struct st*

by (*simp add*: *component-desc-def*)

definition (**in** *c-type*) *xto-bytes* :: *'a* ⇒ *byte list* ⇒ *byte list* **where**
xto-bytes ≡ *field-access* (*component-desc* (*typ-info-t TYPE*(*'a*)))

definition (**in** *c-type*) *xfrom-bytes* :: *byte list* ⇒ *'a* **where**
xfrom-bytes bs ≡ *field-update* (*component-desc* (*typ-info-t TYPE*(*'a*))) *bs undefined*

primrec
toplevel-field-descs :: *'a xtyp-info* ⇒ *'a field-desc list* **and**
toplevel-field-descs-struct :: *'a xtyp-info-struct* ⇒ *'a field-desc list* **and**
toplevel-field-descs-list :: *'a xtyp-info-tuple list* ⇒ *'a field-desc list* **and**
toplevel-field-descs-tuple :: *'a xtyp-info-tuple* ⇒ *'a field-desc list*
where
tfd0: *toplevel-field-descs* (*TypDesc algn st nm*) = *toplevel-field-descs-struct st*

| *tfd1*: *toplevel-field-descs-struct* (*TypScalar n algn d*) = [*d*]
| *tfd2*: *toplevel-field-descs-struct* (*TypAggregate xs*) = *toplevel-field-descs-list xs*

| *tfd3*: *toplevel-field-descs-list* [] = []
| *tfd4*: *toplevel-field-descs-list* (*x*#*xs*) = *toplevel-field-descs-tuple x* @ *toplevel-field-descs-list xs*

| *tfd5*: *toplevel-field-descs-tuple* (*DTuple t nm d*) = [*d*]

locale *padding-base* =
fixes *acc*::*'a* ⇒ *byte list* ⇒ *byte list*
fixes *upd*:: *byte list* ⇒ *'a* ⇒ *'a*
fixes *sz*::*nat*
begin
definition *eq-padding*::*byte list* ⇒ *byte list* ⇒ *bool* **where**
eq-padding bs bs' ≡ *length bs* = *sz* ∧ *length bs'* = *sz* ∧ (∀ *v*. *acc v bs* = *acc v bs'*)
definition *eq-upto-padding*::*byte list* ⇒ *byte list* ⇒ *bool* **where**
eq-upto-padding bs bs' ≡ *length bs* = *sz* ∧ *length bs'* = *sz* ∧ (∀ *v*. *upd bs v* = *upd bs' v*)

definition *is-padding-byte*::*nat* ⇒ *bool* **where**
is-padding-byte i ≡ *i* < *sz* ∧ (∀ *v bs*. *length bs* = *sz* ⟶
(*acc v bs* ! *i* = *bs* ! *i*) ∧
(∀ *b*. *upd bs v* = *upd* (*bs*[*i*:=*b*]) *v*))

definition *is-value-byte*::*nat* ⇒ *bool* **where**
is-value-byte i ≡ *i* < *sz* ∧ (∀ *v bs*. *length bs* = *sz* ⟶
(∀ *bs'*. *length bs'* = *sz* ⟶ (*acc* (*upd bs v*) *bs'* ! *i*) = *bs* ! *i*) ∧

$(\forall\, b.\ acc\ v\ bs = acc\ v\ (bs[i{:=}b])))$

lemma *is-padding-byteI*:
assumes $i < sz$
assumes $\bigwedge v\ bs.\ i < sz \Longrightarrow length\ bs = sz \Longrightarrow acc\ v\ bs\ !\ i = bs\ !\ i$
assumes $\bigwedge v\ bs\ b.\ i < sz \Longrightarrow length\ bs = sz \Longrightarrow upd\ bs\ v = upd\ (bs[i{:=}b])\ v$
shows *is-padding-byte i*
using *assms*
by (*simp add*: *is-padding-byte-def*)

lemma *is-value-byteI*:
assumes $i < sz$
assumes $\bigwedge v\ bs\ bs'.\ i < sz \Longrightarrow length\ bs = sz \Longrightarrow length\ bs' = sz \Longrightarrow$
$(acc\ (upd\ bs\ v)\ bs'\ !\ i) = bs\ !\ i$
assumes $\bigwedge v\ bs\ b.\ i < sz \Longrightarrow length\ bs = sz \Longrightarrow acc\ v\ bs = acc\ v\ (bs[i{:=}b])$
shows *is-value-byte i*
using *assms*
by (*simp add*: *is-value-byte-def*)

lemma *is-padding-byte-acc-id*:
$is\text{-}padding\text{-}byte\ i \Longrightarrow length\ bs = sz \Longrightarrow (acc\ v\ bs\ !\ i) = bs\ !\ i$
by (*simp add*: *is-padding-byte-def*)

lemma *is-value-byte-acc-upd-cancel*:
$is\text{-}value\text{-}byte\ i \Longrightarrow length\ bs = sz \Longrightarrow length\ bs' = sz \Longrightarrow$
$(acc\ (upd\ bs\ v)\ bs'\ !\ i) = bs\ !\ i$
by (*simp add*: *is-value-byte-def*)

lemma *is-padding-byte-eq-upto-padding*: $is\text{-}padding\text{-}byte\ i \Longrightarrow length\ bs = sz \Longrightarrow$ *eq-upto-padding bs* $(bs[i{:=}b])$
by (*simp add*: *eq-upto-padding-def is-padding-byte-def*)

lemma *is-value-byte-eq-padding*: $is\text{-}value\text{-}byte\ i \Longrightarrow length\ bs = sz \Longrightarrow$ *eq-padding bs* $(bs[i{:=}b])$
by (*simp add*: *eq-padding-def is-value-byte-def*)

lemma *is-padding-byte-acc-neq*:
$is\text{-}padding\text{-}byte\ i \Longrightarrow b \neq bs!i \Longrightarrow length\ bs = sz \Longrightarrow acc\ v\ bs \neq acc\ v\ (bs[i{:=}b])$
by (*metis is-padding-byte-def length-list-update nth-list-update-eq*)

lemma *is-padding-byte-acc-eq*:
$is\text{-}padding\text{-}byte\ i \Longrightarrow length\ bs = sz\ \Longrightarrow$
$acc\ v\ bs\ !\ i\ = bs\ !\ i$
by (*auto simp add*: *is-padding-byte-def*)

lemma *is-padding-byte-upd-eq*:
$is\text{-}padding\text{-}byte\ i \Longrightarrow length\ bs = sz \Longrightarrow upd\ bs\ v = upd\ (bs[i{:=}b])\ v$
by (*auto simp add*: *is-padding-byte-def*)

lemma *is-padding-byte-not-eq-padding*: *is-padding-byte i* $\Longrightarrow$ *b* $\neq$ *bs*!*i* $\Longrightarrow$ *length bs* = *sz* $\Longrightarrow$ $\neg$ *eq-padding bs* (*bs*[*i*:=*b*])
by (*simp add*: *padding-base.eq-padding-def padding-base.is-padding-byte-acc-neq*)

lemma *is-value-byte-upd-neq*:
is-value-byte i $\Longrightarrow$ *b* $\neq$ *bs*!*i* $\Longrightarrow$ *length bs* = *sz* $\Longrightarrow$ *upd bs v* $\neq$ *upd* (*bs*[*i*:=*b*]) *v*
by (*metis length-list-update nth-list-update-eq is-value-byte-def*)

lemma *is-value-byte-update-depends*:
assumes *is-value*: *is-value-byte i*
assumes *lbs*: *length bs* = *sz*
shows $\exists$ *b*. *upd* (*bs*[*i* := *b*]) *v* $\neq$ *upd bs v*
proof −
obtain *b* **where** *b* $\neq$ *bs*!*i*
by (*metis len8 len-gt-0 less-le word-power-less-1*)
from *is-value-byte-upd-neq* [*OF is-value this lbs*, *of v*] **show** *?thesis* **by** *metis*
qed

lemma *is-value-byte-acc-eq*:
is-value-byte i $\Longrightarrow$ *length bs* = *sz* $\Longrightarrow$ *acc v bs* = *acc v* (*bs*[*i*:=*b*])
by (*auto simp add*: *is-value-byte-def*)

lemma *is-value-byte-not-eq-upto-padding*:
is-value-byte i $\Longrightarrow$ *b* $\neq$ *bs*!*i* $\Longrightarrow$ *length bs* = *sz* $\Longrightarrow$ $\neg$ *eq-upto-padding bs* (*bs*[*i*:=*b*])
by (*simp add*: *padding-base.eq-upto-padding-def padding-base.is-value-byte-upd-neq*)

lemma *eq-paddingI*[*intro?*]:
assumes *length bs* = *sz*
assumes *length bs′* = *sz*
assumes $\bigwedge$*v*. *acc v bs* = *acc v bs′*
shows *eq-padding bs bs′*
using *assms* **by** (*auto simp add*: *eq-padding-def*)

lemma *eq-upto-paddingI*[*intro?*]:
assumes *length bs* = *sz*
assumes *length bs′* = *sz*
assumes $\bigwedge$*v*. *upd bs v* = *upd bs′ v*
shows *eq-upto-padding bs bs′*
using *assms* **by** (*auto simp add*: *eq-upto-padding-def*)

lemma *eq-padding-length1*: *eq-padding bs bs′* $\Longrightarrow$ *length bs* = *sz*
by (*simp add*: *eq-padding-def*)

lemma *eq-padding-length2*: *eq-padding bs bs′* $\Longrightarrow$ *length bs′* = *sz*
by (*simp add*: *eq-padding-def*)

lemma *eq-upto-padding-length1*: *eq-upto-padding bs bs'* $\Longrightarrow$ *length bs* = *sz*
by (*simp add*: *eq-upto-padding-def*)

lemma *eq-upto-padding-length2*: *eq-upto-padding bs bs'* $\Longrightarrow$ *length bs'* = *sz*
by (*simp add*: *eq-upto-padding-def*)

lemma *eq-padding-length-eq*: *eq-padding bs bs'* $\Longrightarrow$ *length bs* = *length bs'*
by (*simp add*: *eq-padding-def*)

lemma *eq-upto-padding-length-eq*: *eq-upto-padding bs bs'* $\Longrightarrow$ *length bs* = *length bs'*
by (*simp add*: *eq-upto-padding-def*)

lemma *eq-padding-acc*: *eq-padding bs bs'* $\Longrightarrow$ *acc v bs* = *acc v bs'*
by (*simp add*: *eq-padding-def*)

lemma *eq-upto-padding-upd*: *eq-upto-padding bs bs'* $\Longrightarrow$ *upd bs v* = *upd bs' v*
by (*simp add*: *eq-upto-padding-def*)

lemma *eq-padding-refl*[*simp*]: *length bs* = *sz* $\Longrightarrow$ *eq-padding bs bs*
by (*simp add*: *eq-padding-def*)

lemma *eq-upto-padding-refl*[*simp*]: *length bs* = *sz* $\Longrightarrow$ *eq-upto-padding bs bs*
by (*simp add*: *eq-upto-padding-def*)

lemma *eq-padding-sym*: *eq-padding bs bs'* $\longleftrightarrow$ *eq-padding bs' bs*
by (*auto simp add*: *eq-padding-def*)

lemma *eq-padding-symp*: *symp eq-padding*
by (*simp add*: *symp-def eq-padding-sym*)

lemma *eq-upto-padding-sym*: *eq-upto-padding bs bs'* $\longleftrightarrow$ *eq-upto-padding bs' bs*
by (*auto simp add*: *eq-upto-padding-def*)

lemma *eq-upto-padding-symp*: *symp eq-upto-padding*
by (*simp add*: *symp-def eq-upto-padding-sym*)

lemma *eq-padding-trans*: *eq-padding bs1 bs2* $\Longrightarrow$ *eq-padding bs2 bs3* $\Longrightarrow$ *eq-padding bs1 bs3*
by (*auto simp add*: *eq-padding-def*)

lemma *eq-padding-transp*: *transp eq-padding*
unfolding *transp-def*
by (*auto intro*: *eq-padding-trans*)

lemma *eq-upto-padding-trans*: *eq-upto-padding bs1 bs2* $\Longrightarrow$ *eq-upto-padding bs2 bs3* $\Longrightarrow$ *eq-upto-padding bs1 bs3*
by (*auto simp add*: *eq-upto-padding-def*)

lemma *eq-upto-padding-transp*: *transp eq-upto-padding*
unfolding *transp-def*
by (*auto intro*: *eq-upto-padding-trans*)

lemma *eq-padding-to-padding-byte-eq*: *eq-padding bs bs'* $\Longrightarrow$
length bs = *sz* $\wedge$ *length bs'* = *sz* $\wedge$ $(\forall i.$ *is-padding-byte i* $\longrightarrow$ *bs* ! *i* = *bs'* ! *i*)
by (*metis is-padding-byte-acc-id padding-base.eq-padding-def*)

lemma *eq-upto-padding-to-value-byte-eq*: *eq-upto-padding bs bs'* $\Longrightarrow$
length bs = *sz* $\wedge$ *length bs'* = *sz* $\wedge$ $(\forall i.$ *is-value-byte i* $\longrightarrow$ *bs* ! *i* = *bs'* ! *i*)
by (*metis* (*no-types*, *opaque-lifting*) *is-value-byte-acc-upd-cancel local.eq-upto-padding-def*)

end

locale *complement-padding* = *padding-base* +
assumes *complement*: *i* < *sz* $\Longrightarrow$
is-padding-byte i $\neq$ *is-value-byte i*
begin

lemma *eq-padding-eq-upto-padding-complement*:
assumes *lbs*: *length bs* = *sz*
assumes *i-bound*: *i* < *length bs*
assumes *neq*: *b* $\neq$ *bs*!*i*
shows *eq-padding bs* (*bs*[*i* := *b*]) $\neq$ *eq-upto-padding bs* (*bs*[*i* := *b*])
by (*metis* (*no-types*, *opaque-lifting*) *complement i-bound lbs length-list-update neq*
padding-base.eq-paddingI
padding-base.eq-upto-padding-def padding-base.is-padding-byte-not-eq-padding
padding-base.is-padding-byte-upd-eq padding-base.is-value-byte-acc-eq padding-base.is-value-byte-not-eq-upto-p

lemma *eq-padding-padding-byte-id*:
assumes *eq-padding*: *eq-padding bs bs'*
assumes *is-padding*: *is-padding-byte i*
shows *bs*!*i* = *bs'*!*i*
by (*metis eq-padding is-padding is-padding-byte-acc-id padding-base.eq-padding-def*)

lemma *eq-upto-padding-value-byte-id*:
assumes *eq-padding*: *eq-upto-padding bs bs'*
assumes *is-value*: *is-value-byte i*
shows *bs*!*i* = *bs'*!*i*
by (*metis* (*no-types*, *opaque-lifting*) *eq-padding eq-upto-padding-length1*
eq-upto-padding-length2 eq-upto-padding-upd is-value is-value-byte-acc-upd-cancel)

lemma *eq-upto-padding-neq-is-padding-byte*:

assumes *eq-upto-padding*: *eq-upto-padding bs bs′*
assumes *i-bound*: *i* < *length bs*
assumes *neq*: *bs*!*i* ≠ *bs′*!*i*
shows *is-padding-byte i*
by (*metis eq-upto-padding eq-upto-padding-value-byte-id i-bound neq*
complement-padding.complement complement-padding-axioms padding-base.eq-upto-padding-length2
padding-base.eq-upto-padding-length-eq)

lemma *eq-padding-neq-is-value-byte*:
assumes *eq-upto-padding*: *eq-padding bs bs′*
assumes *i-bound*: *i* < *length bs*
assumes *neq*: *bs*!*i* ≠ *bs′*!*i*
shows *is-value-byte i*
by (*metis complement eq-padding-padding-byte-id eq-upto-padding i-bound*
neq padding-base.eq-padding-length2 padding-base.eq-padding-length-eq)

lemma *padding-eq-complement*:
assumes *eq-padding*: *eq-padding bs bs′*
assumes *eq-upto-padding*: *eq-upto-padding bs bs′*
shows *bs* = *bs′*
by (*meson eq-padding eq-padding-padding-byte-id eq-upto-padding eq-upto-padding-length-eq*
eq-upto-padding-neq-is-padding-byte nth-equalityI)

end

locale *padding-lense* = *complement-padding* +
assumes *double-update*: *upd bs* (*upd bs′ s*) = *upd bs s*
assumes *access-update*: *acc* (*upd bs s*) *bs′* = *acc* (*upd bs s′*) *bs′*
assumes *update-access*: *upd* (*acc s bs*) *s* = *s*
assumes *access-size*: *acc s* (*take sz bs*) = *acc s bs*
assumes *access-result-size*: *length* (*acc s bs*) = *sz*
assumes *update-size*: *upd* (*take sz bs*) = *upd bs*
begin

lemma *update-size-ext*: *upd* (*take sz bs*) *s* = *upd bs s*
by (*simp add*: *update-size*)

lemma *update-access-append*: *upd* ((*acc s bs*)@*bs′*) *s* = *s* (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = *upd* (*take sz* ((*acc s bs*)@*bs′*)) *s*
by (*rule update-size-ext* [*symmetric*])
also
from *access-result-size*
have *take sz* ((*acc s bs*)@*bs′*) = *acc s bs*
by *auto*
also note *update-access*
finally show *?thesis* .
qed

lemma *field-access-eq-padding1*: *length bs = sz ⟹ eq-padding (acc v bs) bs*
apply (*rule eq-paddingI*)
apply (*simp add*: *access-result-size*)
apply *simp*
by (*smt* (*verit*) *complement is-padding-byte-acc-id is-value-byte-acc-upd-cancel nth-equalityI access-result-size update-access padding-lense-axioms*)

lemma *field-access-eq-padding2*: *length bs = sz ⟹ eq-padding (acc v2 bs) (acc v2 bs)*
apply (*rule eq-paddingI*)
apply (*simp-all add*: *access-result-size*)
done

lemma *field-access-eq-upto-padding*: *length bs = sz ⟹ length bs = sz ⟹*
eq-upto-padding (acc v bs) (acc v bs′)
apply (*rule eq-upto-paddingI*)
apply (*simp add*: *access-result-size*)
apply (*simp add*: *access-result-size*)
by (*metis access-update double-update update-access*)

lemma *padding-byte-to-eq-padding-eq*:
eq-padding bs bs′
if
length bs = sz
length bs′ = sz
(∀ i. is-padding-byte i ⟶ bs ! i = bs′ ! i)
using *that*(*1*,*2*)
apply (*rule eq-paddingI*)
apply (*rule nth-equalityI*)
unfolding *access-result-size*
apply (*rule refl*)
by (*metis access-result-size complement is-padding-byte-acc-id is-value-byte-acc-upd-cancel that update-access*)

lemma *eq-padding-is-padding-byte-conv*:
eq-padding bs bs′ ⟷ length bs = sz ∧ length bs′ = sz ∧ (∀ i. is-padding-byte i ⟶ bs ! i = bs′ ! i)
using *eq-padding-to-padding-byte-eq padding-byte-to-eq-padding-eq* **by** *blast*

lemma *value-byte-to-eq-upto-padding-eq*:
assumes *lbs*: *length bs = sz*
assumes *lbs′*: *length bs′ = sz*
assumes *is-value*: *∀ i. is-value-byte i ⟶ bs ! i = bs′ ! i*
shows *eq-upto-padding bs bs′*
proof (*rule eq-upto-paddingI* [*OF lbs lbs′*])
fix *v*

have *leq*: *length* (*acc* (*upd bs′ v*) *bs*) = *sz*
by (*simp add*: *access-result-size*)
then have *acc* (*upd bs′ v*) *bs* = *bs*
by (*smt* (*verit*) *eq-padding-neq-is-value-byte field-access-eq-padding1*
is-value is-value-byte-acc-upd-cancel lbs lbs′ nth-equalityI)
then
show *upd bs v* = *upd bs′ v*
by (*metis double-update update-access*)
qed

lemma *eq-upto-padding-is-value-byte-conv*:
eq-upto-padding bs bs′ ⟷ *length bs* = *sz* ∧ *length bs′* = *sz* ∧ (∀ *i*. *is-value-byte i* ⟶ *bs* ! *i* = *bs′* ! *i*)
using *eq-upto-padding-to-value-byte-eq value-byte-to-eq-upto-padding-eq* **by** *blast*

end

locale *wf-field-desc* = *padding-lense* (*field-access d*) (*field-update d*) (*field-sz d*) **for** *d*::′*a field-desc*

locale *field-desc-independent* =
fixes
acc::′*a* ⇒ *byte list* ⇒ *byte list* **and**
upd::*byte list* ⇒ ′*a* ⇒ ′*a* **and**
D:: ′*a field-desc set*
assumes *fu-commutes*: *d* ∈ *D* ⟹ *fu-commutes upd* (*field-update d*)
assumes *acc-upd-old*: *d* ∈ *D* ⟹ *acc* (*field-update d bs v*) *bs′* = *acc v bs′*
assumes *acc-upd-new*: *d* ∈ *D* ⟹ *field-access d* (*upd bs′ v*) *bs* = *field-access d v bs*

primrec
field-descs :: ′*a xtyp-info* ⇒ ′*a field-desc list* **and**
field-descs-struct :: ′*a xtyp-info-struct* ⇒ ′*a field-desc list* **and**
field-descs-list :: ′*a xtyp-info-tuple list* ⇒ ′*a field-desc list* **and**
field-descs-tuple :: ′*a xtyp-info-tuple* ⇒ ′*a field-desc list*
where
fd0: *field-descs* (*TypDesc algn st nm*) = *field-descs-struct st*

| *fd1*: *field-descs-struct* (*TypScalar n algn d*) = [*d*]
| *fd2*: *field-descs-struct* (*TypAggregate xs*) = *field-descs-list xs*

| *fd3*: *field-descs-list* [] = []
| *fd4*: *field-descs-list* (*x*#*xs*) = *field-descs-tuple x* @ *field-descs-list xs*

| *fd5*: *field-descs-tuple* (*DTuple t nm d*) = [*d*] @ *field-descs t*

primrec
wf-component-descs :: ′*a xtyp-info* ⇒ *bool* **and**
wf-component-descs-struct :: ′*a xtyp-info-struct* ⇒ *bool* **and**

```
  wf-component-descs-list :: 'a xtyp-info-tuple list ⇒ bool and
  wf-component-descs-tuple :: 'a xtyp-info-tuple ⇒ bool
where
  wfc0: wf-component-descs (TypDesc algn st nm) = wf-component-descs-struct st

| wfc1: wf-component-descs-struct (TypScalar n algn d) = (n = field-sz d)
| wfc2: wf-component-descs-struct (TypAggregate xs) = (wf-component-descs-list
xs)

| wfc3: wf-component-descs-list [] = True
| wfc4: wf-component-descs-list (x#xs) = (wf-component-descs-tuple x ∧ wf-component-descs-list
xs)

| wfc5: wf-component-descs-tuple (DTuple t nm d) =
         (d = component-desc t  ∧ wf-component-descs t)

primrec field-descs-independent:: 'a field-desc list ⇒ bool
  where
field-descs-independent [] = True |
field-descs-independent (d#ds) =
   (field-desc-independent (field-access d) (field-update d) (set ds) ∧
    field-descs-independent ds)

primrec
  component-descs-independent :: 'a xtyp-info ⇒ bool and
  component-descs-independent-struct :: 'a xtyp-info-struct ⇒ bool and
  component-descs-independent-list :: 'a xtyp-info-tuple list ⇒ bool and
  component-descs-independent-tuple :: 'a xtyp-info-tuple ⇒ bool
where
  cdi0: component-descs-independent (TypDesc algn st nm) = component-descs-independent-struct
st

| cdi1: component-descs-independent-struct (TypScalar n algn d) = True
| cdi2: component-descs-independent-struct (TypAggregate xs) = (component-descs-independent-list
xs)

| cdi3: component-descs-independent-list [] = True
| cdi4: component-descs-independent-list (f#fs) = (field-descs-independent (toplevel-field-descs-list
(f#fs)) ∧
          component-descs-independent-tuple f ∧ component-descs-independent-list
fs)

| cdi5: component-descs-independent-tuple (DTuple ft fn d) = (component-descs-independent
ft)

locale wf-field-descs =
  fixes D::'a field-desc set
  assumes wf-desc[intro]: d ∈ D ⟹ wf-field-desc d
```

lemma *wf-field-descs-union* [*simp*]: *wf-field-descs* (*D* ∪ *E*) = (*wf-field-descs D* ∧ *wf-field-descs E*)
by (*auto simp add*: *wf-field-descs-def*)

lemma *wf-field-descs-empty*[*simp*]: *wf-field-descs* {}
by (*simp add*: *wf-field-descs-def*)

lemma *wf-field-descs-insert* [*simp*]: *wf-field-descs* (*insert d D*) = (*wf-field-desc d* ∧ *wf-field-descs D*)
by (*auto simp add*: *wf-field-descs-def*)

definition *padding-desc*:: *nat* ⇒ *'a field-desc* **where**
padding-desc n = (⦇*field-access* = λ*v bs*. *if n* ≤ *length bs then take n bs else replicate n 0*, *field-update* = λ*bs*. *id*, *field-sz* = *n*⦈)

definition *is-padding-desc d* = (∃ *n*. *d* = *padding-desc n*)

definition *padding-tag n* = *TypDesc 0* (*TypScalar n 0* (*padding-desc n*)) *''!pad-typ''*

definition *is-padding-tag t* = (∃ *n*. *t* = *padding-tag n*)

definition *padding-field-name f* = (∃ *xs*. *f* = *CHR ''!'' # xs*)

lemma *padding-field-name-empty*[*simp*]: *padding-field-name* [] = *False*
by (*simp add*: *padding-field-name-def*)

lemma *padding-field-name-cons*[*simp*]: *padding-field-name* (*f*#*fs*) = (*f* = *CHR ''!''*)
by (*simp add*: *padding-field-name-def*)

definition *qualified-padding-field-name fs* = (∃ *f fs'*. *fs*= *f*#*fs'* ∧ *padding-field-name* (*last fs*))

lemma *qualified-pading-field-name-empty*[*simp*]: *qualified-padding-field-name* [] = *False*
by (*simp add*: *qualified-padding-field-name-def*)

lemma *qualified-pading-field-name-single*[*simp*]: *qualified-padding-field-name* [*f*] = *padding-field-name f*
by (*simp add*: *qualified-padding-field-name-def*)

lemma *qualified-pading-field-name-cons*[*simp*]: *qualified-padding-field-name* (*f*#*fs*) = *padding-field-name* (*last* (*f*#*fs*))
by (*simp add*: *qualified-padding-field-name-def*)

primrec
wf-padding :: (*'a*, *'b*) *typ-info* ⇒ *bool* **and**

wf-padding-struct :: (*'a*, *'b*) *typ-info-struct* ⇒ *bool* **and**
wf-padding-list :: (*'a*, *'b*) *typ-info-tuple list* ⇒ *bool* **and**
wf-padding-tuple :: (*'a*, *'b*) *typ-info-tuple* ⇒ *bool*
where
wf-padding (*TypDesc algn st nm*) = *wf-padding-struct st*

| *wf-padding-struct* (*TypScalar m algn d*) = *True*
| *wf-padding-struct* (*TypAggregate xs*) = *wf-padding-list xs*

| *wf-padding-list* [] = *True*
| *wf-padding-list* (*x*#*xs*) = (*wf-padding-tuple x* ∧ *wf-padding-list xs*)

| *wf-padding-tuple* (*DTuple s f d*) = ((*padding-field-name f* ⟶ *is-padding-tag s*) ∧ *wf-padding s*)

class *xmem-type* = *mem-type* +
assumes *wf-component-descs*: *wf-component-descs* (*typ-info-t TYPE*(*'a*))
assumes *component-descs-independent*: *component-descs-independent* (*typ-info-t TYPE*(*'a*))
assumes *wf-field-descs*: *wf-field-descs* (*set* (*field-descs* (*typ-info-t TYPE*(*'a*))))
assumes *wf-padding*: *wf-padding* (*typ-info-t TYPE*(*'a*))

locale *fields-contained* =
fixes
acc::*'a* ⇒ *byte list* ⇒ *byte list* **and**
upd::*byte list* ⇒ *'a* ⇒ *'a* **and**
D:: *'a field-desc set*
assumes *access-contained*: *d* ∈ *D* ⟹ *acc s* = *acc s'* ⟹ *field-access d s* = *field-access d s'*
assumes *update-contained*: *d* ∈ *D* ⟹ ∃ *bs'*. ∀ *s'*. *acc s'* = *acc s* ⟶ *field-update d bs s'* = *upd bs' s'*

primrec
contained-field-descs :: *'a xtyp-info* ⇒ *bool* **and**
contained-field-descs-struct :: *'a xtyp-info-struct* ⇒ *bool* **and**
contained-field-descs-list :: *'a xtyp-info-tuple list* ⇒ *bool* **and**
contained-field-descs-tuple :: *'a xtyp-info-tuple* ⇒ *bool*
where
wfd0: *contained-field-descs* (*TypDesc algn st nm*) = *contained-field-descs-struct st*

| *wfd1*: *contained-field-descs-struct* (*TypScalar n algn d*) = *True*
| *wfd2*: *contained-field-descs-struct* (*TypAggregate xs*) = *contained-field-descs-list xs*

| *wfd3*: *contained-field-descs-list* [] = *True*
| *wfd4*: *contained-field-descs-list* (*x*#*xs*) = (*contained-field-descs-tuple x* ∧ *contained-field-descs-list xs*)

| *wfd5*: *contained-field-descs-tuple* (*DTuple t nm d*) =
(*contained-field-descs t* ∧
fields-contained (*field-access d*) (*field-update d*) (*set* (*toplevel-field-descs t*)))

class *xmem-contained-type* = *xmem-type* +
assumes *contained-field-descs*: *contained-field-descs* (*typ-info-t* (*TYPE*('*a*)))

In order to construct a type of class *mem-type* we usually construct extended type-info for class *xmem-contained-type*. The extendend type-info is better behaved with respect to composition of sub-structures / arrays.

end

theory *CTypes*
imports
CTypesDefs HOL−Eisbach.Eisbach-Tools
begin

11.10 *super-update-bs*

lemma *length-super-update-bs* [*simp*]:
n + *length v* ≤ *length bs* ⟹ *length* (*super-update-bs v bs n*) = *length bs*
by (*clarsimp simp*: *super-update-bs-def*)

lemma *drop-super-update-bs*:
⟦ *k* ≤ *n*; *n* ≤ *length bs* ⟧ ⟹ *drop k* (*super-update-bs v bs n*) = *super-update-bs v* (*drop k bs*) (*n* − *k*)
by (*simp add*: *super-update-bs-def take-drop*)

lemma *drop-super-update-bs2*:
⟦ *n* ≤ *length bs*; *n* + *length v* ≤ *k* ⟧ ⟹
drop k (*super-update-bs v bs n*) = *drop k bs*
by (*clarsimp simp*: *super-update-bs-def min-def split*: *if-split-asm*)

lemma *take-super-update-bs*:
⟦ *k* ≤ *n*; *n* ≤ *length bs* ⟧ ⟹ *take k* (*super-update-bs v bs n*) = *take k bs*
by (*clarsimp simp*: *super-update-bs-def min-def split*: *if-split-asm*)

lemma *take-super-update-bs2*:
⟦ *n* ≤ *length bs*; *n* + *length v* ≤ *k* ⟧ ⟹
take k (*super-update-bs v bs n*) = *super-update-bs v* (*take k bs*) *n*
apply (*clarsimp simp*: *super-update-bs-def min-def split*: *if-split-asm*)
apply (*cases n*=*k*; *simp add*: *drop-take*)
done

lemma *take-drop-super-update-bs*:
length v = n ⟹ m ≤ length bs ⟹ take n (drop m (super-update-bs v bs m)) = v
by (*simp add*: *super-update-bs-def*)

lemma *super-update-bs-take-drop*:
n + m ≤ length bs ⟹ super-update-bs (take m (drop n bs)) bs n = bs
by (*simp add*: *super-update-bs-def*) (*metis append.assoc append-take-drop-id take-add*)

lemma *super-update-bs-same-length*: *length bs = length xbs ⟹ super-update-bs bs xbs 0 = bs*
by (*simp add*: *super-update-bs-def*)

lemma *super-update-bs-append-drop-first*:
length xbs = m ⟹ n + length bs ≤ m ⟹ drop m (super-update-bs bs (xbs @ ybs) n) = ybs
by (*simp add*: *super-update-bs-def*)

lemma *super-update-bs-append-take-first*:
length xbs = m ⟹ n + length bs ≤ m ⟹ take m (super-update-bs bs (xbs @ ybs) n) = (super-update-bs bs xbs n)
by (*simp add*: *super-update-bs-def*)

lemma *super-update-bs-append-drop-second*:
length xbs = m ⟹ m ≤ n ⟹
drop m (super-update-bs bs (xbs @ ybs) n) = (super-update-bs bs ybs (n − m))
by (*simp add*: *super-update-bs-def*)

lemma *super-update-bs-append-take-second*:
length xbs = m ⟹ m ≤ n ⟹
take m (super-update-bs bs (xbs @ ybs) n) = xbs
by (*simp add*: *super-update-bs-def*)

lemma *super-update-bs-length*: *length bs + n ≤ length xbs ==> length (super-update-bs bs xbs n) = length xbs*
by (*simp add*: *super-update-bs-def*)

lemma *super-update-bs-append2*: *length xbs ≤ n ⟹*
super-update-bs bs (xbs @ ybs) n = xbs @ super-update-bs bs ybs (n − length xbs)
by (*simp add*: *super-update-bs-def*)

lemma *super-update-bs-append1*: *n + length bs ≤ length xbs ⟹*
super-update-bs bs (xbs @ ybs) n = super-update-bs bs xbs n @ ybs
by (*simp add*: *super-update-bs-def*)

11.11 Rest

lemma *fu-commutes*:
fu-commutes f g $\Longrightarrow$ *f bs* (*g bs'* *v*) = *g bs'* (*f bs v*)
by (*simp add*: *fu-commutes-def*)

lemma *size-td-list-append* [*simp*]:
size-td-list (*xs*@*ys*) = *size-td-list xs* + *size-td-list ys*
by (*induct xs*) (*auto*)

lemma *access-ti-append*:
$\bigwedge$*bs*. *length bs* = *size-td-list* (*xs*@*ys*) $\Longrightarrow$
access-ti-list (*xs*@*ys*) *t bs* =
access-ti-list xs t (*take* (*size-td-list xs*) *bs*) @
access-ti-list ys t (*drop* (*size-td-list xs*) *bs*)
proof (*induct xs*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case* **by** (*simp add*: *min-def ac-simps drop-take*)
qed

lemma *update-ti-append* [*simp*]:
$\bigwedge$*bs*. *update-ti-list* (*xs*@*ys*) *bs v* =
update-ti-list xs (*take* (*size-td-list xs*) *bs*)
(*update-ti-list ys* (*drop* (*size-td-list xs*) *bs*) *v*)
proof (*induct xs*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case* **by** (*simp add*: *drop-take ac-simps min-def*)
qed

lemma *update-ti-struct-t-typscalar* [*simp*]:
update-ti-struct-t (*TypScalar n algn d*) =
(*λbs*. *if length bs* = *n then field-update d bs else id*)
by (*rule ext*, *simp add*: *update-ti-struct-t-def*)

lemma *update-ti-list-t-empty* [*simp*]:
update-ti-list-t [] = (*λx*. *id*)
by (*rule ext*, *simp add*: *update-ti-list-t-def*)

lemma *update-ti-list-t-cons* [*simp*]:
update-ti-list-t (*x*#*xs*) = (*λbs v*.
if length bs = *size-td-tuple x* + *size-td-list xs then*
update-ti-tuple-t x (*take* (*size-td-tuple x*) *bs*)
(*update-ti-list-t xs* (*drop* (*size-td-tuple x*) *bs*) *v*) *else*
v)
by (*force simp*: *update-ti-list-t-def update-ti-tuple-t-def min-def*)

lemma *update-ti-append-s* [*simp*]:
⋀*bs*. *update-ti-list-t* (*xs*@*ys*) *bs v* = (
if length bs = *size-td-list xs* + *size-td-list ys then*
update-ti-list-t xs (*take* (*size-td-list xs*) *bs*)
(*update-ti-list-t ys* (*drop* (*size-td-list xs*) *bs*) *v*) *else*
v)
proof (*induct xs*)
case *Nil* **show** *?case* **by** (*simp add*: *update-ti-list-t-def*)
next
case (*Cons x xs*) **thus** *?case* **by** (*simp add*: *min-def drop-take ac-simps*)
qed

lemma *update-ti-tuple-t-dtuple* [*simp*]:
update-ti-tuple-t (*DTuple t f d*) = *update-ti-t t*
by (*rule ext*, *simp add*: *update-ti-tuple-t-def update-ti-t-def*)

---

lemma *field-desc-empty* [*simp*]:
field-desc (*TypDesc algn* (*TypAggregate* []) *nm*) =
(| *field-access* = λ*x bs*. [],
field-update = λ*x*. *id*, *field-sz* = *0* |)
by (*force simp*: *update-ti-t-def*)

lemma *export-uinfo-typdesc-simp* [*simp*]:
export-uinfo (*TypDesc algn st nm*) = *map-td field-norm* (λ-. ()) (*TypDesc algn st nm*)
by (*simp add*: *export-uinfo-def*)

lemma *map-td-list-append* [*simp*]:
map-td-list f g (*xs*@*ys*) = *map-td-list f g xs* @ *map-td-list f g ys*
by (*induct xs*) *auto*

lemma *map-td-id*:
map-td (λ*n algn*. *id*) *id t* = (*t*::('*a*, '*b*) *typ-desc*)
map-td-struct (λ*n algn*. *id*) *id st* = (*st*::('*a*, '*b*) *typ-struct*)
map-td-list (λ*n algn*. *id*) *id ts* = (*ts*::('*a*, '*b*) *typ-tuple list*)
map-td-tuple (λ*n algn*. *id*) *id x* = (*x*::('*a*, '*b*) *typ-tuple*)
by (*induction t* **and** *st* **and** *ts* **and** *x*) *simp-all*

lemma *dt-snd-map-td-list*:
dt-snd ‘ *set* (*map-td-list f g ts*) = *dt-snd* ‘ *set ts*
proof (*induct ts*)
case (*Cons x xs*) **thus** *?case* **by** (*cases x*) *auto*
qed *simp*

lemma *wf-desc-map*:

shows *wf-desc* (*map-td f g t*) = *wf-desc t* **and**
wf-desc-struct (*map-td-struct f g st*) = *wf-desc-struct st* **and**
wf-desc-list (*map-td-list f g ts*) = *wf-desc-list ts* **and**
wf-desc-tuple (*map-td-tuple f g x*) = *wf-desc-tuple x*
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*Cons-typ-desc x xs*) **thus** *?case*
by (*cases x*, *auto simp*: *dt-snd-map-td-list*)
qed *auto*

lemma *wf-desc-list-append* [*simp*]:
wf-desc-list (*xs*@*ys*) =
(*wf-desc-list xs* ∧ *wf-desc-list ys* ∧ *dt-snd* ‘ *set xs* ∩ *dt-snd* ‘ *set ys* = {})
by (*induct xs*) *auto*

lemma *wf-size-desc-list-append* [*simp*]:
wf-size-desc-list (*xs*@*ys*) = (*wf-size-desc-list xs* ∧ *wf-size-desc-list ys*)
by (*induct xs*) *auto*

lemma *norm-tu-list-append* [*simp*]:
norm-tu-list (*xs*@*ys*) *bs* =
norm-tu-list xs (*take* (*size-td-list xs*) *bs*) @ *norm-tu-list ys* (*drop* (*size-td-list xs*) *bs*)
by (*induct xs arbitrary*: *bs*, *auto simp*: *min-def ac-simps drop-take*)

lemma *wf-size-desc-gt*:
shows *wf-size-desc* (*t*::(′*a*, ′*b*) *typ-desc*) ⟹ *0* < *size-td t* **and**
wf-size-desc-struct st ⟹ *0* < *size-td-struct* (*st*::(′*a*,′*b*) *typ-struct*) **and**
⟦ *ts* ≠ []; *wf-size-desc-list ts* ⟧ ⟹ *0* < *size-td-list* (*ts*::(′*a*,′*b*) *typ-tuple list*)
and
wf-size-desc-tuple x ⟹ *0* < *size-td-tuple* (*x*::(′*a*,′*b*) *typ-tuple*)
by (*induct t* **and** *st* **and** *ts* **and** *x rule*: *typ-desc-typ-struct-inducts*, *auto*)

lemma *field-lookup-empty* [*simp*]:
field-lookup t [] *n* = *Some* (*t*,*n*)
by (*cases t*) *clarsimp*

lemma *field-lookup-tuple-empty* [*simp*]:
field-lookup-tuple x [] *n* = *None*
by (*cases x*) *clarsimp*

lemma *field-lookup-list-empty* [*simp*]:
field-lookup-list ts [] *n* = *None*
by (*induct ts arbitrary*: *n*) *auto*

lemma *field-lookup-struct-empty* [*simp*]:
field-lookup-struct st [] *n* = *None*
by (*cases st*) *auto*

lemma *field-lookup-list-append*:

field-lookup-list (*xs*@*ys*) *f n* = (*case field-lookup-list xs f n of*
None ⇒ *field-lookup-list ys f* (*n* + *size-td-list xs*)
| *Some y* ⇒ *Some y*)
proof (*induct xs arbitrary*: *n*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case*
by (*cases x*) (*auto simp*: *ac-simps split*: *option.splits*)
qed

lemma *field-lookup-list-None*:
f ∉ *dt-snd* ‘ *set ts* ⟹ *field-lookup-list ts* (*f*#*fs*) *m* = *None*
proof (*induct ts arbitrary*: *f fs m*)
case (*Cons x* -) **thus** *?case* **by** (*cases x*) *auto*
qed *simp*

lemma *field-lookup-list-Some*:
f ∈ *dt-snd* ‘ *set ts* ⟹ *field-lookup-list ts* [*f*] *m* ≠ *None*
proof (*induct ts arbitrary*: *f m*)
case (*Cons x* -) **thus** *?case* **by** (*cases x*) *auto*
qed *simp*

lemma *field-lookup-offset-le*:
shows ⋀*s m n f*. *field-lookup t f m* = *Some* ((*s*::(*'a*,*'b*) *typ-desc*),*n*) ⟹ *m* ≤ *n*
and
⋀*s m n f*. *field-lookup-struct st f m* = *Some* ((*s*::(*'a*,*'b*) *typ-desc*),*n*) ⟹ *m* ≤ *n* **and**
⋀*s m n f*. *field-lookup-list ts f m* = *Some* ((*s*::(*'a*, *'b*) *typ-desc*),*n*) ⟹ *m* ≤ *n* **and**
⋀*s m n f*. *field-lookup-tuple x f m* = *Some* ((*s*::(*'a*, *'b*) *typ-desc*),*n*) ⟹ *m* ≤ *n*
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*Cons-typ-desc x xs*) **thus** *?case* **by** (*fastforce split*: *option.splits*)
qed (*auto split*: *if-split-asm*)

lemma *field-lookup-offset'*:
shows ⋀*f m m' n t'*. (*field-lookup t f m* = *Some* ((*t'*::(*'a*,*'b*) *typ-desc*),*m* + *n*)) =
(*field-lookup t f m'* = *Some* (*t'*,*m'* + *n*)) **and**
⋀*f m m' n t'*. (*field-lookup-struct st f m* = *Some* ((*t'*::(*'a*,*'b*) *typ-desc*),*m* + *n*)) =
(*field-lookup-struct st f m'* = *Some* (*t'*,*m'* + *n*)) **and**
⋀*f m m' n t'*. (*field-lookup-list ts f m* = *Some* ((*t'*::(*'a*,*'b*) *typ-desc*),*m* + *n*)) =
(*field-lookup-list ts f m'* = *Some* (*t'*,*m'* + *n*)) **and**
⋀*f m m' n t'*. (*field-lookup-tuple x f m* = *Some* ((*t'*::(*'a*,*'b*) *typ-desc*),*m* + *n*)) =
(*field-lookup-tuple x f m'* = *Some* (*t'*,*m'* + *n*))
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*Cons-typ-desc x xs*)

```
show ?case
proof
  assume ls: field-lookup-list (x # xs) f m = Some (t', m + n)
  show field-lookup-list (x # xs) f m' = Some (t', m' + n) (is ?X)
  proof cases
    assume ps: field-lookup-tuple x f m = None
    moreover from this ls have ∃ k. n = size-td (dt-fst x) + k
      by (clarsimp dest!: field-lookup-offset-le, arith)
    moreover have field-lookup-tuple x f m' = None
      apply (rule ccontr)
      using ps
      apply clarsimp
      subgoal premises prems for a b
      proof −
        obtain k where b = m' + k
          using prems field-lookup-offset-le
          by (metis less-eqE)
      then show ?thesis
        using prems
        by (clarsimp simp: Cons-typ-desc [where m'=m])
    qed
    done
    ultimately show ?X using ls
      by (clarsimp simp: add.assoc [symmetric])
        (subst (asm) Cons-typ-desc [where m'=m' + size-td (dt-fst x)], fast)
  next
    assume nps: field-lookup-tuple x f m ≠ None
    moreover from this have field-lookup-tuple x f m' ≠ None
      apply (clarsimp)
      subgoal premises prems for a b
      proof −
        obtain k where b = m + k
          using prems field-lookup-offset-le
          by (metis less-eqE)
        then show ?thesis
          using prems
          apply clarsimp
          apply (subst (asm) Cons-typ-desc [where m'=m'])
          apply fast
          done
      qed
      done
    ultimately show ?X using ls
      by (clarsimp, subst (asm) Cons-typ-desc [where m'=m'], simp)
  qed
next
  assume ls: field-lookup-list (x # xs) f m' = Some (t', m' + n)
  show field-lookup-list (x # xs) f m = Some (t', m + n) (is ?X)
  proof cases
```

```
    assume ps: field-lookup-tuple x f m' = None
    moreover from this ls have ∃ k. n = size-td (dt-fst x) + k
      by (clarsimp dest!: field-lookup-offset-le, arith)
    moreover have field-lookup-tuple x f m = None
      apply (rule ccontr)
      using ps
      apply clarsimp
      subgoal premises prems for a b
      proof −
        obtain k where b = m + k
          using prems field-lookup-offset-le
          by (metis less-eqE)
        then show ?thesis using prems
          by (clarsimp simp: Cons-typ-desc [where m'=m'])
      qed
      done
    ultimately show ?X using ls
      by (clarsimp simp: add.assoc [symmetric])
        (subst (asm) Cons-typ-desc [where m'=m + size-td (dt-fst x)], fast)
  next
    assume nps: field-lookup-tuple x f m' ≠ None
    moreover from this have field-lookup-tuple x f m ≠ None
      apply clarsimp
      subgoal premises prems for a b
      proof −
        obtain k where b = m' + k
          using prems field-lookup-offset-le
          by (metis less-eqE)
        then show ?thesis using prems
          apply clarsimp
          apply (subst (asm) Cons-typ-desc [where m'=m])
          apply fast
          done
      qed
      done
    ultimately show ?X using ls
      by (clarsimp, subst (asm) Cons-typ-desc [where m'=m], simp)
  qed
 qed
qed auto

lemma field-lookup-offset:
  (field-lookup t f m = Some (t',m + n)) = (field-lookup t f 0 = Some (t',n))
  by (simp add: field-lookup-offset' [where m'=0])

lemma field-lookup-offset2:
  field-lookup t f m = Some (t',n) ⟹ field-lookup t f 0 = Some (t',n − m)
  by (simp add: field-lookup-offset-le
          flip: field-lookup-offset [where m=m])
```

lemma *field-lookup-list-offset*:
(*field-lookup-list ts f m* = *Some* (*t'*,*m* + *n*)) = (*field-lookup-list ts f 0* = *Some* (*t'*,*n*))
by (*simp add*: *field-lookup-offset'* [**where** *m'*=*0*])

lemma *field-lookup-list-offset2*:
field-lookup-list ts f m = *Some* (*t'*,*n*) ⟹ *field-lookup-list ts f 0* = *Some* (*t'*,*n* − *m*)
by (*simp add*: *field-lookup-offset-le*
flip: *field-lookup-list-offset* [**where** *m*=*m*])

lemma *field-lookup-list-offset3*:
field-lookup-list ts f 0 = *Some* (*t'*,*n*) ⟹ *field-lookup-list ts f m* = *Some* (*t'*,*m* + *n*)
by (*simp add*: *field-lookup-list-offset*)

lemma *field-lookup-list-offsetD*:
⟦ *field-lookup-list ts f 0* = *Some* (*s*,*k*);
field-lookup-list ts f m = *Some* (*t*,*n*) ⟧ ⟹ *s*=*t* ∧ *n*=*m*+*k*
subgoal premises *prems*
proof −
have ∃ *k*. *n* = *m* + *k* **using** *prems field-lookup-offset-le* **by** (*metis less-eqE*)
then show *?thesis* **using** *prems*
by (*clarsimp simp*: *field-lookup-list-offset*)
qed
done

lemma *field-lookup-offset-None*:
(*field-lookup t f m* = *None*) = (*field-lookup t f 0* = *None*)
by (*auto simp*: *field-lookup-offset2 field-lookup-offset* [**where** *m*=*m*,*symmetric*]
intro: *ccontr*)

lemma *field-lookup-list-offset-None*:
(*field-lookup-list ts f m* = *None*) = (*field-lookup-list ts f 0* = *None*)
by (*auto simp*: *field-lookup-list-offset2 field-lookup-list-offset* [**where** *m*=*m*,*symmetric*]
intro: *ccontr*)

lemma *map-td-size* [*simp*]:
shows *size-td* (*map-td f g t*) = *size-td t* **and**
size-td-struct (*map-td-struct f g st*) = *size-td-struct st* **and**
size-td-list (*map-td-list f g ts*) = *size-td-list ts* **and**
size-td-tuple (*map-td-tuple f g x*) = *size-td-tuple x*
by (*induct t* **and** *st* **and** *ts* **and** *x*, *auto*)

lemma *td-set-map-td1*:
(*s*, *n*) ∈ *td-set t k* ⟹ (*map-td f g s*, *n*) ∈ *td-set* (*map-td f g t*) *k* **and**
(*s*, *n*) ∈ *td-set-struct st k* ⟹ (*map-td f g s*, *n*) ∈ *td-set-struct* (*map-td-struct f g st*) *k* **and**

$(s, n) \in$ *td-set-list ts k* $\Longrightarrow$ $($*map-td f g s*, $n) \in$ *td-set-list* $($*map-td-list f g ts*$)$ *k* **and**
$(s, n) \in$ *td-set-tuple td k* $\Longrightarrow$ $($*map-td f g s*, $n) \in$ *td-set-tuple* $($*map-td-tuple f g td*$)$ *k*
apply $($*induct t* **and** *st* **and** *ts* **and** *td arbitrary*: *n k* **and** *n k* **and** *n k* **and** *n k*$)$
by $($*auto*, *metis dt-tuple.collapse map-td-size*(4) *tz5*$)$

lemma *size-td-tuple-dt-fst*:
size-td-tuple p $=$ *size-td* $($*dt-fst p*$)$
by $($*cases p*, *simp*$)$

lemma *td-set-map-td2*:
$(s', n) \in$ *td-set* $($*map-td f g t*$)$ *k* $\Longrightarrow \exists s.\ (s, n) \in$ *td-set t k* $\wedge$ *s'* $=$ *map-td f g s* **and**
$(s', n) \in$ *td-set-struct* $($*map-td-struct f g st*$)$ *k* $\Longrightarrow \exists s.\ (s, n) \in$ *td-set-struct st k* $\wedge$ *s'* $=$ *map-td f g s* **and**
$(s', n) \in$ *td-set-list* $($*map-td-list f g ts*$)$ *k* $\Longrightarrow \exists s.\ (s, n) \in$ *td-set-list ts k* $\wedge$ *s'* $=$ *map-td f g s* **and**
$(s', n) \in$ *td-set-tuple* $($*map-td-tuple f g td*$)$ *k* $\Longrightarrow \exists s.\ (s, n) \in$ *td-set-tuple td k* $\wedge$ *s'* $=$ *map-td f g s*
proof $($*induct t* **and** *st* **and** *ts* **and** *td arbitrary*: *n k* **and** *n k* **and** *n k* **and** *n k* $)$
case $($*TypDesc nat typ-struct list*$)$
then show *?case* **by** *auto*
next
case $($*TypScalar nat1 nat2 a*$)$
then show *?case* **by** *auto*
next
case $($*TypAggregate list*$)$
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case $($*Cons-typ-desc dt-tuple list*$)$
then show *?case*
apply $($*clarsimp*, *safe*$)$
apply *blast*
by $($*metis map-td-size*(4) *size-td-tuple-dt-fst*$)$
next
case $($*DTuple-typ-desc typ-desc list b*$)$
then show *?case* **by** *auto*
qed

lemma *td-set-offset1*:
$(s, n) \in$ *td-set t k* $\Longrightarrow$ $(s, n + l) \in$ *td-set t* $(k + l)$ **and**
$(s, n) \in$ *td-set-struct st k* $\Longrightarrow$ $(s, n + l) \in$ *td-set-struct st* $(k + l)$ **and**
$(s, n) \in$ *td-set-list xs k* $\Longrightarrow$ $(s, n + l) \in$ *td-set-list xs* $(k + l)$ **and**
$(s, n) \in$ *td-set-tuple td k* $\Longrightarrow$ $(s, n + l) \in$ *td-set-tuple td* $(k + l)$

apply (*induct t* **and** *st* **and** *xs* **and** *td arbitrary*: *s n k l* **and** *s n k l* **and** *s n k l* **and** *s n k l*)
by (*auto*, *smt* (*verit*, *best*) *add.commute add.left-commute*)

lemma *td-set-offset2*:
$(s, n + l) \in$ *td-set t* $(k + l) \Longrightarrow (s, n) \in$ *td-set t k* **and**
$(s, n + l) \in$ *td-set-struct st* $(k + l) \Longrightarrow (s, n) \in$ *td-set-struct st k* **and**
$(s, n + l) \in$ *td-set-list xs* $(k + l) \Longrightarrow (s, n) \in$ *td-set-list xs k* **and**
$(s, n + l) \in$ *td-set-tuple td* $(k + l) \Longrightarrow (s, n) \in$ *td-set-tuple td k*
apply (*induct t* **and** *st* **and** *xs* **and** *td arbitrary*: *s n k l* **and** *s n k l* **and** *s n k l* **and** *s n k l*)
by (*auto*, *smt* (*verit*, *best*) *add.commute add.left-commute*)

lemma *td-set-offset-conv*: $(s, n) \in$ *td-set t k* $\longleftrightarrow (s, n + l) \in$ *td-set t* $(k + l)$
using *td-set-offset1 td-set-offset2* **by** *blast*

lemma *td-set-offset-0-conv*: $(s, n + k) \in$ *td-set t k* $\longleftrightarrow (s, n) \in$ *td-set t 0*
using *td-set-offset-conv* [**where** *k*=*0* **and** *n*=*n* **and** *l*=*k* **and** *s*=*s* **and** *t* =*t*]
by *simp*

lemma *td-set-offset-le'*:
$\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in$ *td-set t m* $\longrightarrow m \leq n$
$\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in$ *td-set-struct st m* $\longrightarrow m \leq n$
$\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in$ *td-set-list ts m* $\longrightarrow m \leq n$
$\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in$ *td-set-tuple x m* $\longrightarrow m \leq n$
by (*induct t* **and** *st* **and** *ts* **and** *x*) *fastforce+*

lemma *td-set-offset-0-conv'*: $(s, n) \in$ *td-set t k* $\longleftrightarrow (\exists\, n'.\ (s, n') \in$ *td-set t 0* $\wedge\ n = n' + k)$
by (*metis add.commute le-Suc-ex td-set-offset-0-conv td-set-offset-le'*(*1*))

lemma *td-set-list-set-td-set1*: $(s, n) \in$ *td-set-list ts k* $\Longrightarrow$
$(\exists\, t\ n'.\ t \in$ *set ts* $\wedge\ (s, n') \in$ *td-set* (*dt-fst t*) *0*)
apply (*induct ts arbitrary*: *n k*)
apply *simp*
apply *simp*
by (*metis dt-tuple.collapse td-set-offset-0-conv' ts5*)

lemma *export-uinfo-size* [*simp*]:
size-td (*export-uinfo t*) = *size-td* (*t*::('*a*,'*b*) *typ-info*)
by (*simp add*: *export-uinfo-def*)

lemma (**in** *c-type*) *typ-uinfo-size* [*simp*]:
size-td (*typ-uinfo-t TYPE*('*a*)) = *size-td* (*typ-info-t TYPE*('*a*))
by (*simp add*: *typ-uinfo-t-def export-uinfo-def*)

lemma *wf-size-desc-map*:
shows *wf-size-desc* (*map-td f g t*) = *wf-size-desc t* **and**

wf-size-desc-struct (*map-td-struct f g st*) = *wf-size-desc-struct st* **and**
wf-size-desc-list (*map-td-list f g ts*) = *wf-size-desc-list ts* **and**
wf-size-desc-tuple (*map-td-tuple f g x*) = *wf-size-desc-tuple x*
proof (*induction t* **and** *st* **and** *ts* **and** *x*)
case (*TypAggregate list*)
then show *?case* **by** (*cases list*) *auto*
qed *auto*

lemma *map-td-flr-Some* [*simp*]:
map-td-flr f g (*Some* (*t*,*n*)) = *Some* (*map-td f g t*,*n*)
by (*clarsimp simp*: *map-td-flr-def*)

lemma *map-td-flr-None* [*simp*]:
map-td-flr f g None = *None*
by (*clarsimp simp*: *map-td-flr-def*)

lemma *field-lookup-map*:
shows ⋀*f m s*. *field-lookup t f m* = *s* ⟹
field-lookup (*map-td fupd g t*) *f m* = *map-td-flr fupd g s* **and**
⋀*f m s*. *field-lookup-struct st f m* = *s* ⟹
field-lookup-struct (*map-td-struct fupd g st*) *f m* = *map-td-flr fupd g s*
and
⋀*f m s*. *field-lookup-list ts f m* = *s* ⟹
field-lookup-list (*map-td-list fupd g ts*) *f m* = *map-td-flr fupd g s* **and**
⋀*f m s*. *field-lookup-tuple x f m* = *s* ⟹
field-lookup-tuple (*map-td-tuple fupd g x*) *f m* = *map-td-flr fupd g s*
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*Cons-typ-desc x xs*) **thus** *?case*
by (*clarsimp*, *cases x*, *auto simp*: *map-td-flr-def split*: *option.splits*)
qed *auto*

lemma *field-lookup-export-uinfo-Some*:
field-lookup (*t*::('*a*,'*b*) *typ-info*) *f m* = *Some* (*s*,*n*) ⟹
field-lookup (*export-uinfo t*) *f m* = *Some* (*export-uinfo s*,*n*)
by (*simp add*: *export-uinfo-def field-lookup-map*)

lemma *field-lookup-struct-export-Some*:
field-lookup-struct (*st*::('*a*,'*b*) *typ-struct*) *f m* = *Some* (*s*,*n*) ⟹
field-lookup-struct (*map-td-struct fupd g st*) *f m* = *Some* (*map-td fupd g s*,*n*)
by (*simp add*: *field-lookup-map*)

lemma *field-lookup-struct-export-None*:
field-lookup-struct (*st*::('*a*, '*b*) *typ-struct*) *f m* = *None* ⟹
field-lookup-struct (*map-td-struct fupd g st*) *f m* = *None*
by (*simp add*: *field-lookup-map*)

lemma *field-lookup-list-export-Some*:
field-lookup-list (*ts*::('*a*, '*b*) *typ-tuple list*) *f m* = *Some* (*s*,*n*) ⟹
field-lookup-list (*map-td-list fupd g ts*) *f m* = *Some* (*map-td fupd g s*,*n*)

by (*simp add*: *field-lookup-map*)

lemma *field-lookup-list-export-None*:
field-lookup-list (*ts*::('*a*, '*b*) *typ-tuple list*) *f m* = *None* ⟹
field-lookup-list (*map-td-list fupd g ts*) *f m* = *None*
by (*simp add*: *field-lookup-map*)

lemma *field-lookup-tuple-export-Some*:
field-lookup-tuple (*x*::('*a*, '*b*) *typ-tuple*) *f m* = *Some* (*s*,*n*) ⟹
field-lookup-tuple (*map-td-tuple fupd g x*) *f m* = *Some* (*map-td fupd g s*,*n*)
by (*simp add*: *field-lookup-map*)

lemma *field-lookup-tuple-export-None*:
field-lookup-tuple (*x*::('*a*, '*b*) *typ-tuple*) *f m* = *None* ⟹
field-lookup-tuple (*map-td-tuple fupd g x*) *f m* = *None*
by (*simp add*: *field-lookup-map*)

lemma *import-fltr-Some* [*simp*]:
import-fltr f g (*Some* (*map-td f g t*,*n*)) (*Some* (*t*,*n*))
by (*clarsimp simp*: *import-fltr-def*)

lemma *import-fltr-None* [*simp*]:
import-fltr f g None None
by (*clarsimp simp*: *import-fltr-def*)

lemma *field-lookup-export-uinfo-rev''*:
⋀*f m s*. *field-lookup* (*map-td fupd g t*) *f m* = *s* ⟹
import-fltr fupd g s ((*field-lookup t f m*)::('*a*,'*b*) *fltr*)
⋀*f m s*. *field-lookup-struct* (*map-td-struct fupd g st*) *f m* = *s* ⟹
import-fltr fupd g s ((*field-lookup-struct st f m*)::('*a*,'*b*) *fltr*)
⋀*f m s*. *field-lookup-list* (*map-td-list fupd g ts*) *f m* = *s* ⟹
import-fltr fupd g s ((*field-lookup-list ts f m*)::('*a*,'*b*) *fltr*)
⋀*f m s*. *field-lookup-tuple* (*map-td-tuple fupd g x*) *f m* = *s* ⟹
import-fltr fupd g s ((*field-lookup-tuple x f m*)::('*a*,'*b*) *fltr*)
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*clarsimp simp*: *import-fltr-def export-uinfo-def*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*

apply *clarsimp*
apply(*clarsimp split*: *option.splits*)
apply(*cases f*, *clarsimp*+)
apply(*rule conjI*, *clarsimp*)
apply(*cases dt-tuple*, *simp*)
apply *clarsimp*
apply(*drule field-lookup-tuple-export-Some* [**where** *fupd*=*fupd* **and** *g*=*g*])
apply *simp*
apply(*rule conjI*; *clarsimp*)
apply(*drule field-lookup-tuple-export-None* [**where** *fupd*=*fupd* **and** *g*=*g*])
apply *simp*
apply *metis*
done
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** (*auto*)
qed

lemma *field-lookup-export-uinfo-rev'*:
(∀ *f m s*. *field-lookup* (*map-td fupd g t*) *f m* = *s* ⟶
import-flr fupd g s ((*field-lookup t f m*)::('*a*,'*b*) *flr*)) ∧
(∀ *f m s*. *field-lookup-struct* (*map-td-struct fupd g st*) *f m* = *s* ⟶
import-flr fupd g s ((*field-lookup-struct st f m*)::('*a*,'*b*) *flr*)) ∧
(∀ *f m s*. *field-lookup-list* (*map-td-list fupd g ts*) *f m* = *s* ⟶
import-flr fupd g s ((*field-lookup-list ts f m*)::('*a*,'*b*) *flr*)) ∧
(∀ *f m s*. *field-lookup-tuple* (*map-td-tuple fupd g x*) *f m* = *s* ⟶
import-flr fupd g s ((*field-lookup-tuple x f m*)::('*a*,'*b*) *flr*))
by (*auto simp*: *field-lookup-export-uinfo-rev''*)

lemma *field-lookup-export-uinfo-Some-rev*:
field-lookup (*export-uinfo* (*t*::('*a*,'*b*) *typ-info*)) *f m* = *Some* (*s*,*n*) ⟹
∃ *k*. *field-lookup t f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*
apply(*insert field-lookup-export-uinfo-rev'* [*of field-norm* (λ-. ()) *t undefined undefined undefined*])
apply *clarsimp*
apply(*drule spec* [**where** *x*=*f*])
apply(*drule spec* [**where** *x*=*m*])
apply(*clarsimp simp*: *import-flr-def export-uinfo-def split*: *option.splits*)
done

lemma (**in** *c-type*) *field-lookup-uinfo-Some-rev*:
field-lookup (*typ-uinfo-t* (*TYPE*('*a*))) *f m* = *Some* (*s*,*n*) ⟹
∃ *k*. *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*
apply (*simp add*: *typ-uinfo-t-def*)
apply (*rule field-lookup-export-uinfo-Some-rev*)

apply *assumption*
done

lemma (**in** *c-type*) *field-lookup-offset-untyped-eq* [*simp*]:
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*,*n*) ⟹
field-offset-untyped (*typ-uinfo-t TYPE*('*a*)) *f* = *n*
apply(*simp add*: *field-offset-untyped-def typ-uinfo-t-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add*: *typ-uinfo-t-def export-uinfo-def*)
done

lemma (**in** *c-type*) *field-lookup-offset-eq* [*simp*]:
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*,*n*) ⟹
field-offset TYPE('*a*) *f* = *n*
by(*simp add*: *field-offset-def*)

lemma *field-offset-self* [*simp*]:
field-offset t [] = *0*
by (*simp add*: *field-offset-def field-offset-untyped-def*)

lemma (**in** *c-type*) *field-ti-self* [*simp*]:
field-ti TYPE('*a*) [] = *Some* (*typ-info-t TYPE*('*a*))
by (*simp add*: *field-ti-def*)

lemma (**in** *c-type*) *field-size-self* [*simp*]:
field-size TYPE('*a*) [] = *size-td* (*typ-info-t TYPE*('*a*))
by (*simp add*: *field-size-def*)

lemma *field-lookup-prefix-None''*:
(∀*f g m*. *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *None* ⟶ *field-lookup t* (*f*@*g*) *m* = *None*)
(∀*f g m*. *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *None* ⟶ *f* ≠ [] ⟶
field-lookup-struct st (*f*@*g*) *m* = *None*)
(∀*f g m*. *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* = *None* ⟶ *f* ≠ [] ⟶
field-lookup-list ts (*f*@*g*) *m* = *None*)
(∀*f g m*. *field-lookup-tuple* (*x*::('*a*,'*b*) *typ-tuple*) *f m* = *None* ⟶ *f* ≠ [] ⟶
field-lookup-tuple x (*f*@*g*) *m* = *None*)
by (*induct t* **and** *st* **and** *ts* **and** *x*)
(*clarsimp split*: *option.splits*)+

lemma *field-lookup-prefix-None'*:
(∀*f g m*. *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *None* ⟶ *field-lookup t* (*f*@*g*) *m* = *None*) ∧
(∀*f g m*. *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *None* ⟶ *f* ≠ [] ⟶
field-lookup-struct st (*f*@*g*) *m* = *None*) ∧
(∀*f g m*. *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* = *None* ⟶ *f* ≠ [] ⟶
field-lookup-list ts (*f*@*g*) *m* = *None*) ∧
(∀*f g m*. *field-lookup-tuple* (*x*::('*a*,'*b*) *typ-tuple*) *f m* = *None* ⟶ *f* ≠ [] ⟶

field-lookup-tuple x (f@g) m = None)
by (*auto simp*: *field-lookup-prefix-None''*)

lemma *field-lookup-prefix-Some''*:
(∀*f g t' m n*. *field-lookup t f m = Some* ((*t'*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc t* ⟶
field-lookup t (*f*@*g*) *m* = *field-lookup t' g n*)
(∀*f g t' m n*. *field-lookup-struct st f m = Some* ((*t'*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-struct st* ⟶
field-lookup-struct st (*f*@*g*) *m* = *field-lookup t' g n*)
(∀*f g t' m n*. *field-lookup-list ts f m = Some* ((*t'*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-list ts* ⟶
field-lookup-list ts (*f*@*g*) *m* = *field-lookup t' g n*)
(∀*f g t' m n*. *field-lookup-tuple x f m = Some* ((*t'*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-tuple x* ⟶
field-lookup-tuple x (*f*@*g*) *m* = *field-lookup t' g n*)
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply(*clarsimp split*: *option.splits*)
apply(*cases dt-tuple*)
subgoal for *f*
apply(*cases f*, *clarsimp*)
by (*clarsimp simp*: *field-lookup-list-None split*: *if-split-asm*)
done
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** *auto*
qed

lemma *field-lookup-prefix-Some'*:
(∀*f g t' m n*. *field-lookup t f m = Some* ((*t'*::('*a*, '*b*) *typ-desc*),*n*) ⟶ *wf-desc t* ⟶
field-lookup t (*f*@*g*) *m* = *field-lookup t' g n*) ∧
(∀*f g t' m n*. *field-lookup-struct st f m = Some* ((*t'*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-struct st* ⟶

field-lookup-struct st (f@g) m = field-lookup t' g n) ∧
(∀f g t' m n. field-lookup-list ts f m = Some ((t'::('a,'b) typ-desc),n) ⟶ wf-desc-list ts ⟶
field-lookup-list ts (f@g) m = field-lookup t' g n) ∧
(∀f g t' m n. field-lookup-tuple x f m = Some ((t'::('a,'b) typ-desc),n) ⟶ wf-desc-tuple x ⟶
field-lookup-tuple x (f@g) m = field-lookup t' g n)
by (*auto simp*: *field-lookup-prefix-Some''*)

lemma *field-lookup-append-eq*:
wf-desc t ⟹
field-lookup t (f @ g) n =
Option.bind (field-lookup t f n) (λ(t', m). field-lookup t' g m)
by (*cases field-lookup t f n*)
(*auto simp add*: *field-lookup-prefix-None''(1)*[*rule-format*]
field-lookup-prefix-Some''(1)[*rule-format*])

lemma *field-lookup-offset-shift*:
NO-MATCH 0 m ⟹ field-lookup t f 0 = Some (t', n) ⟹ field-lookup t f m = Some (t', m + n)
by (*simp add*: *field-lookup-offset*)

lemma *field-lookup-offset-shift'*:
field-lookup t f m = Some (s, k) ⟹ field-lookup t f n = Some (s, k + n − m)
by (*metis CTypes.field-lookup-offset2 Nat.add-diff-assoc2 add.commute field-lookup-offset*
field-lookup-offset-le(1))

lemma *field-lookup-append*:
assumes *t*: *wf-desc t*
and *f*: *field-lookup t f n = Some (u, m)* **and** *g*: *field-lookup u g l = Some (v, k)*
shows *field-lookup t (f @ g) n = Some (v, k + m − l)*
using *field-lookup-prefix-Some''(1)*[*rule-format*, *OF f t*, *of g*]
g[*THEN field-lookup-offset-shift'*, *of m*]
by *simp*

lemma *field-lvalue-empty-simp* [*simp*]:
Ptr &(p→[]) = p
by (*simp add*: *field-lvalue-def field-offset-def field-offset-untyped-def*)

lemma *map-td-align* [*simp*]:
align-td-wo-align (map-td f g t) = align-td-wo-align (t::('a,'b) typ-desc)
align-td-wo-align-struct (map-td-struct f g st) = align-td-wo-align-struct (st::('a,'b) typ-struct)
align-td-wo-align-list (map-td-list f g ts) = align-td-wo-align-list (ts::('a,'b) typ-tuple list)
align-td-wo-align-tuple (map-td-tuple f g x) = align-td-wo-align-tuple (x::('a,'b) typ-tuple)
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *map-td-align'* [*simp*]:
align-td (*map-td f g t*) = *align-td* (*t*::('*a*,'*b*) *typ-desc*)
align-td-struct (*map-td-struct f g st*) = *align-td-struct* (*st*::('*a*,'*b*) *typ-struct*)
align-td-list (*map-td-list f g ts*) = *align-td-list* (*ts*::('*a*,'*b*) *typ-tuple list*)
align-td-tuple (*map-td-tuple f g x*) = *align-td-tuple* (*x*::('*a*,'*b*) *typ-tuple*)
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *typ-uinfo-align* [*simp*]:
align-td-wo-align (*export-uinfo t*) = *align-td-wo-align* (*t*::('*a*,'*b*) *typ-info*)
by (*simp add*: *export-uinfo-def*)

lemma *ptr-aligned-Ptr-0* [*simp*]:
ptr-aligned NULL
by (*simp add*: *ptr-aligned-def*)

lemma *td-set-self* [*simp*]:
(*t*,*m*) ∈ *td-set t m*
by (*cases t*) *simp*

lemma *td-set-wf-size-desc* [*rule-format*]:
(∀ *s m n*. *wf-size-desc t* ⟶ ((*s*::('*a*,'*b*) *typ-desc*),*m*) ∈ *td-set t n* ⟶ *wf-size-desc s*)
(∀ *s m n*. *wf-size-desc-struct st* ⟶ ((*s*::('*a*,'*b*) *typ-desc*),*m*) ∈ *td-set-struct st n* ⟶ *wf-size-desc s*)
(∀ *s m n*. *wf-size-desc-list ts* ⟶ ((*s*::('*a*,'*b*) *typ-desc*),*m*) ∈ *td-set-list ts n* ⟶ *wf-size-desc s*)
(∀ *s m n*. *wf-size-desc-tuple x* ⟶ ((*s*::('*a*,'*b*) *typ-desc*),*m*) ∈ *td-set-tuple x n* ⟶ *wf-size-desc s*)
by (*induct t* **and** *st* **and** *ts* **and** *x*) *force+*

lemma *td-set-size-lte'*:
(∀ *s k m*. ((*s*::('*a*,'*b*) *typ-desc*),*k*) ∈ *td-set t m* ⟶ *size s* = *size t* ∧ *s*=*t* ∧ *k*=*m* ∨ *size s* < *size t*)
(∀ *s k m*. ((*s*::('*a*,'*b*) *typ-desc*),*k*) ∈ *td-set-struct st m* ⟶ *size s* < *size st*)
(∀ *s k m*. ((*s*::('*a*,'*b*) *typ-desc*),*k*) ∈ *td-set-list xs m* ⟶ *size s* < *size-list* (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) *xs*)
(∀ *s k m*. ((*s*::('*a*,'*b*) *typ-desc*),*k*) ∈ *td-set-tuple x m* ⟶ *size s* < *size-dt-tuple size* (λ-. *0*) (λ-. *0*) *x*)
by (*induct t* **and** *st* **and** *xs* **and** *x*) *force+*

lemma *td-set-size-lte*:
(*s*,*k*) ∈ *td-set t m* ⟹ *size s* = *size t* ∧ *s*=*t* ∧ *k*=*m* ∨
size s < *size t*
by (*simp add*: *td-set-size-lte'*)

lemma *td-set-struct-size-lte*:
(*s*,*k*) ∈ *td-set-struct st m* ⟹ *size s* < *size st*
by (*simp add*: *td-set-size-lte'*)

lemma *td-set-list-size-lte*:
(*s*,*k*) ∈ *td-set-list ts m* ⟹ *size s* < *size-list* (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) *ts*
by (*simp add*: *td-set-size-lte'*)

lemma *td-aggregate-not-in-td-set-list* [*simp*]:
¬ (*TypDesc algn* (*TypAggregate xs*) *tn*,*k*) ∈ *td-set-list xs m*
by (*fastforce dest*: *td-set-list-size-lte simp*: *size-char-def*)

lemma *sub-size-td*:
(*s*::('*a*,'*b*) *typ-desc*) ≤ *t* ⟹ *size s* ≤ *size t*
by (*fastforce dest*: *td-set-size-lte simp*: *typ-tag-le-def*)

lemma *sub-tag-antisym*:
⟦ (*s*::('*a*,'*b*) *typ-desc*) ≤ *t*; *t* ≤ *s* ⟧ ⟹ *s*=*t*
apply(*frule sub-size-td*)
apply(*frule sub-size-td* [**where** *t*=*s*])
apply(*clarsimp simp*: *typ-tag-le-def*)
apply(*drule td-set-size-lte*)
apply *clarsimp*
done

lemma *sub-tag-refl*:
(*s*::('*a*,'*b*) *typ-desc*) ≤ *s*
unfolding *typ-tag-le-def* **by** (*cases s*, *fastforce*)

lemma *sub-tag-sub'*:
∀ *s m n*. ((*s*::('*a*,'*b*) *typ-desc*),*n*) ∈ *td-set t m* ⟶ *td-set s n* ⊆ *td-set t m*
∀ *s m n*. ((*s*::('*a*,'*b*) *typ-desc*),*n*) ∈ *td-set-struct ts m* ⟶ *td-set s n* ⊆ *td-set-struct ts m*
∀ *s m n*. ((*s*::('*a*,'*b*) *typ-desc*),*n*) ∈ *td-set-list xs m* ⟶ *td-set s n* ⊆ *td-set-list xs m*
∀ *s m n*. ((*s*::('*a*,'*b*) *typ-desc*),*n*) ∈ *td-set-tuple x m* ⟶ *td-set s n* ⊆ *td-set-tuple x m*
by (*induct t* **and** *ts* **and** *xs* **and** *x*) *fastforce*+

lemma *sub-tag-sub*:
(*s*,*n*) ∈ *td-set t m* ⟹ *td-set s n* ⊆ *td-set t m*
by (*simp add*: *sub-tag-sub'*)

lemma *td-set-fst*:
∀ *m n*. *fst* ‘ *td-set* (*s*::('*a*, '*b*) *typ-desc*) *m* = *fst* ‘ *td-set s n*
∀ *m n*. *fst* ‘ *td-set-struct* (*st*::('*a*,'*b*) *typ-struct*) *m* = *fst* ‘ *td-set-struct st n*
∀ *m n*. *fst* ‘ *td-set-list* (*xs*::('*a*, '*b*) *typ-tuple list*) *m* = *fst* ‘ *td-set-list xs n*
∀ *m n*. *fst* ‘ *td-set-tuple* (*x*::('*a*, '*b*) *typ-tuple*) *m* = *fst* ‘ *td-set-tuple x n*
by (*induct s* **and** *st* **and** *xs* **and** *x*) (*all* ‹*clarsimp*›, *fast*, *fast*)

lemma *sub-tag-trans*:
⟦ (*s*::('*a*,'*b*) *typ-desc*) ≤ *t*; *t* ≤ *u* ⟧ ⟹ *s* ≤ *u*
apply(*clarsimp simp*: *typ-tag-le-def*)

by (*meson sub-tag-sub'*(*1*) *subset-iff td-set-offset-0-conv*)

instantiation *typ-desc* :: (*type*, *type*) *order*
begin
instance
apply *intro-classes*
apply(*fastforce simp*: *typ-tag-lt-def typ-tag-le-def dest*: *td-set-size-lte*)
apply(*rule sub-tag-refl*)
apply(*erule* (*1*) *sub-tag-trans*)
apply(*erule* (*1*) *sub-tag-antisym*)
done
end

lemma *td-set-offset-size''*:
$\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\ t\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\ t$
$\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}struct\ st\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}struct\ st$
$\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}list\ ts\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}list\ ts$
$\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}tuple\ x\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}tuple\ x$
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*cases dt-tuple*)
apply *fastforce*
done
qed *auto*

lemma *td-set-offset-size'*:
$(\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\ t\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\ t) \wedge$
$(\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}struct\ st\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}struct\ st) \wedge$
$(\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}list\ ts\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}list\ ts) \wedge$
$(\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}tuple\ x\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}tuple\ x)$
by (*auto simp*: *td-set-offset-size''*)

lemma *td-set-offset-size*:
$(s,n) \in td\text{-}set\ t\ 0 \Longrightarrow size\text{-}td\ s + n \leq size\text{-}td\ t$
using *td-set-offset-size'* [*of t undefined undefined undefined*] **by** *fastforce*

lemma *td-set-struct-offset-size*:
$(s,n) \in td\text{-}set\text{-}struct\ st\ m \Longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}struct\ st$
using *td-set-offset-size'* [*of undefined st undefined undefined*] **by** *clarsimp*

lemma *td-set-list-offset-size*:
(*s*,*n*) ∈ *td-set-list ts 0* ⟹ *size-td s* + *n* ≤ *size-td-list ts*
using *td-set-offset-size'* [*of undefined undefined ts undefined*]
by *fastforce*

lemma *td-set-offset-size-m*:
(*s*,*n*) ∈ *td-set t m* ⟹ *size-td s* + (*n* − *m*) ≤ *size-td t*
using *insert td-set-offset-size'* [*of t undefined undefined undefined*]
by *clarsimp*

lemma *td-set-list-offset-size-m*:
(*s*,*n*) ∈ *td-set-list t m* ⟹ *size-td s* + (*n* − *m*) ≤ *size-td-list t*
using *insert td-set-offset-size'* [*of undefined undefined t undefined*]
by *clarsimp*

lemma *td-set-tuple-offset-size-m*:
(*s*,*n*) ∈ *td-set-tuple t m* ⟹ *size-td s* + (*n* − *m*) ≤ *size-td-tuple t*
using *td-set-offset-size'* [*of undefined undefined undefined t*]
by *clarsimp*

lemma *td-set-list-offset-le*:
(*s*,*n*) ∈ *td-set-list ts m* ⟹ *m* ≤ *n*
by (*simp add*: *td-set-offset-le'*)

lemma *td-set-tuple-offset-le*:
(*s*,*n*) ∈ *td-set-tuple ts m* ⟹ *m* ≤ *n*
by (*simp add*: *td-set-offset-le'*)

lemma *field-of-self* [*simp*]:
field-of 0 t t
by (*simp add*: *field-of-def*)

lemma *td-set-export-uinfo'*:
∀ *f m n s*. ((*s*::('*a*,'*b*) *typ-info*),*n*) ∈ *td-set t m* ⟶
(*export-uinfo s*,*n*) ∈ *td-set* (*export-uinfo t*) *m*
∀ *f m n s*. ((*s*::('*a*,'*b*) *typ-info*),*n*) ∈ *td-set-struct st m* ⟶
(*export-uinfo s*,*n*) ∈ *td-set-struct* (*map-td-struct field-norm* (λ-. ()) *st*) *m*
∀ *f m n s*. ((*s*::('*a*,'*b*) *typ-info*),*n*) ∈ *td-set-list ts m* ⟶
(*export-uinfo s*,*n*) ∈ *td-set-list* (*map-td-list field-norm* (λ-. ()) *ts*) *m*
∀ *f m n s*. ((*s*::('*a*,'*b*) *typ-info*),*n*) ∈ *td-set-tuple x m* ⟶
(*export-uinfo s*,*n*) ∈ *td-set-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) *m*
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **by**(*cases dt-tuple*) (*simp add*: *export-uinfo-def*)
qed (*auto simp add*: *export-uinfo-def*)

lemma *td-set-export-uinfo*:
(∀*f m n s*. ((*s*::('*a*,'*b*) *typ-info*),*n*) ∈ *td-set t m* ⟶
(*export-uinfo s*,*n*) ∈ *td-set* (*export-uinfo t*) *m*) ∧
(∀*f m n s*. ((*s*::('*a*,'*b*) *typ-info*),*n*) ∈ *td-set-struct st m* ⟶
(*export-uinfo s*,*n*) ∈ *td-set-struct* (*map-td-struct field-norm* (λ-. ()) *st*) *m*) ∧
(∀*f m n s*. ((*s*::('*a*,'*b*) *typ-info*),*n*) ∈ *td-set-list ts m* ⟶
(*export-uinfo s*,*n*) ∈ *td-set-list* (*map-td-list field-norm* (λ-. ()) *ts*) *m*) ∧
(∀*f m n s*. ((*s*::('*a*,'*b*) *typ-info*),*n*) ∈ *td-set-tuple x m* ⟶
(*export-uinfo s*,*n*) ∈ *td-set-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) *m*)
by (*auto simp*: *td-set-export-uinfo'*)

lemma *td-set-export-uinfoD*:
(*s*,*n*) ∈ *td-set t m* ⟹ (*export-uinfo s*,*n*) ∈ *td-set* (*export-uinfo t*) *m*
using *td-set-export-uinfo* [*of t undefined undefined undefined*] **by** *clarsimp*

lemma *td-set-field-lookup''*:
∀*s m n*. *wf-desc t* ⟶ (((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set t m* ⟶
(∃*f*. *field-lookup t f m* = *Some* (*s*,*m*+*n*)))
∀*s m n*. *wf-desc-struct st* ⟶ (((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-struct st m*
⟶
(∃*f*. *field-lookup-struct st f m* = *Some* (*s*,*m*+*n*)))
∀*s m n*. *wf-desc-list ts* ⟶ (((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-list ts m* ⟶
(∃*f*. *field-lookup-list ts f m* = *Some* (*s*,*m*+*n*)))
∀*s m n*. *wf-desc-tuple x* ⟶ (((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-tuple x m* ⟶
(∃*f*. *field-lookup-tuple x f m* = *Some* (*s*,*m*+*n*)))
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case*
apply *clarsimp*
subgoal for *s m n*
apply(*cases s*, *clarsimp*)
apply (*rule conjI*)
apply *clarsimp*
apply(*rule exI* [**where** *x*=[]])
apply *clarsimp*+
apply((*erule allE*)+, *erule* (*1*) *impE*)
apply *clarsimp*
subgoal for - - - *f*
apply(*cases f*, *clarsimp*+)
subgoal for *x xs*
apply(*rule exI* [**where** *x*=*x*#*xs*])
apply *clarsimp*
done
done
done
done
next
case (*TypScalar nat1 nat2 a*)

```
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply clarsimp
    apply(rule conjI, clarsimp)
     apply((erule allE)+, erule (1) impE)
     apply(clarsimp)

    subgoal for s m n f
      apply (cases f, clarsimp+)
      subgoal for x xs
        apply(rule exI [where x=x#xs])
        apply(clarsimp)
        done
      done
    apply clarsimp
    subgoal premises prems for s m n
      using prems(1−3,6 )
        prems(5)[rule-format, of s m + size-td (dt-fst dt-tuple) n − size-td (dt-fst
dt-tuple)]
      apply −
      apply(frule td-set-list-offset-le)
      apply clarsimp
      subgoal for f
        apply(rule exI [where x=f])
        apply(clarsimp split: option.splits)
        apply(cases dt-tuple, clarsimp split: if-split-asm)
        apply(cases f, clarsimp+)
        apply(simp add: field-lookup-list-None)
        done
      done
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply clarsimp
    apply((erule allE)+, erule (1) impE)
    apply clarsimp
    subgoal for s m n f
      apply(rule exI [where x=list#f])
      apply clarsimp
      done
```

done
qed

lemma *td-set-field-lookup'*:
(∀ *s m n. wf-desc t* ⟶ (((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set t m* ⟶
(∃ *f. field-lookup t f m* = *Some* (*s*,*m*+*n*)))) ∧
(∀ *s m n. wf-desc-struct st* ⟶ (((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-struct st m* ⟶
(∃ *f. field-lookup-struct st f m* = *Some* (*s*,*m*+*n*)))) ∧
(∀ *s m n. wf-desc-list ts* ⟶ (((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-list ts m* ⟶
(∃ *f. field-lookup-list ts f m* = *Some* (*s*,*m*+*n*)))) ∧
(∀ *s m n. wf-desc-tuple x* ⟶ (((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-tuple x m* ⟶
(∃ *f. field-lookup-tuple x f m* = *Some* (*s*,*m*+*n*))))
by (*auto simp*: *td-set-field-lookup''*)

lemma *td-set-field-lookup-rev''*:
∀ *s m n.* (∃ *f. field-lookup t f m* = *Some* (*s*,*m*+*n*)) ⟶
((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set t m*
∀ *s m n.* (∃ *f. field-lookup-struct st f m* = *Some* (*s*,*m*+*n*)) ⟶
((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-struct st m*
∀ *s m n.* (∃ *f. field-lookup-list ts f m* = *Some* (*s*,*m*+*n*)) ⟶
((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-list ts m*
∀ *s m n.* (∃ *f. field-lookup-tuple x f m* = *Some* (*s*,*m*+*n*)) ⟶
((*s*::('*a*,'*b*) *typ-desc*),*m* + *n*) ∈ *td-set-tuple x m*
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case*
apply *clarsimp*
subgoal for *s m n f*
apply(*cases f*, *clarsimp*)
apply *clarsimp*
apply((*erule allE*)+, *erule impE*, *fast*)
apply *fast*
done
done
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*

```
    apply clarsimp
    apply(clarsimp split: option.splits)
    subgoal premises prems for s m n f
      using prems (1, 4−5)
        prems(3)[rule-format, where s = s and m = m + size-td (dt-fst dt-tuple)
and n= n − size-td (dt-fst dt-tuple)]
      apply −

      apply(frule field-lookup-offset-le)
      apply clarsimp
      apply fast
      done
    subgoal by auto
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply clarsimp
    subgoal for s m n f
      apply(cases f, clarsimp+)
      apply((erule allE)+, erule impE, fast)
      apply assumption
      done
    done
qed

lemma td-set-field-lookup-rev':
  (∀ s m n. (∃ f. field-lookup t f m = Some (s,m+n)) ⟶
      ((s::('a,'b) typ-desc),m + n) ∈ td-set t m) ∧
    (∀ s m n. (∃ f. field-lookup-struct st f m = Some (s,m+n)) ⟶
      ((s::('a,'b) typ-desc),m + n) ∈ td-set-struct st m) ∧
    (∀ s m n. (∃ f. field-lookup-list ts f m = Some (s,m+n)) ⟶
      ((s::('a,'b) typ-desc),m + n) ∈ td-set-list ts m) ∧
    (∀ s m n. (∃ f. field-lookup-tuple x f m = Some (s,m+n)) ⟶
      ((s::('a,'b) typ-desc),m + n) ∈ td-set-tuple x m)
  by (auto simp: td-set-field-lookup-rev'')

lemma td-set-field-lookup:
  wf-desc t ⟹ k ∈ td-set t 0 = (∃ f. field-lookup t f 0 = Some k)
  using td-set-field-lookup' [of t undefined undefined undefined]
        td-set-field-lookup-rev' [of t undefined undefined undefined]
  apply clarsimp
  apply(cases k, clarsimp)
  by (metis add-0)

lemma td-set-field-lookupD:
  field-lookup t f m = Some k ⟹ k ∈ td-set t m
  using td-set-field-lookup-rev' [of t undefined undefined undefined]
  apply(cases k, clarsimp)
```

by (*metis field-lookup-offset-le*(*1*) *le-iff-add*)

lemma *td-set-struct-field-lookup-structD*:
field-lookup-struct st f m = *Some k* ⟹ *k* ∈ *td-set-struct st m*
using *td-set-field-lookup-rev'* [*of undefined st undefined undefined*]
apply(*cases k*, *clarsimp*)
by (*metis field-lookup-offset-le*(*2*) *le-iff-add*)

lemma *field-lookup-struct-td-simp* [*simp*]:
field-lookup-struct ts f m ≠ *Some* (*TypDesc algn ts nm*, *m*)
by (*fastforce dest*: *td-set-struct-field-lookup-structD td-set-struct-size-lte*)

lemma *td-set-list-field-lookup-listD*:
field-lookup-list xs f m = *Some k* ⟹ *k* ∈ *td-set-list xs m*
using *td-set-field-lookup-rev'* [*of undefined undefined xs undefined*]
apply(*cases k*, *clarsimp*)
by (*metis field-lookup-offset-le*(*3*) *le-Suc-ex*)

lemma *td-set-tuple-field-lookup-tupleD*:
field-lookup-tuple x f m = *Some k* ⟹ *k* ∈ *td-set-tuple x m*
using *td-set-field-lookup-rev'* [*of undefined undefined undefined x*]
apply(*cases k*, *clarsimp*)
by (*metis field-lookup-offset-le*(*4*) *nat-le-iff-add*)

lemma *field-lookup-offset-size''*:
field-lookup t f n = *Some* (*u*, *m*) ⟹ *size-td u* + *m* ≤ *size-td t* + *n*
by (*metis Nat.add-diff-assoc field-lookup-offset-le*(*1*)
le-diff-conv td-set-field-lookupD td-set-offset-size')

lemma *field-lookup-offset-size'*:
assumes *n*: *field-lookup t f 0* = *Some* (*t'*,*n*) **shows** *size-td t'* + *n* ≤ *size-td t*
using *field-lookup-offset-size''*[*OF n*] **by** *simp*

lemma *intvl-subset-of-field-lookup*:
field-lookup t f 0 = *Some* (*u*, *n*) ⟹
{*a* + *of-nat n* ..+ *size-td u*} ⊆ {*a* ..+ *size-td t*}
by (*simp add*: *field-lookup-offset-size' intvl-le*)

lemma *field-lookup-wf-size-desc-gt*:
⟦ *field-lookup t f n* = *Some* (*a*,*b*); *wf-size-desc t* ⟧ ⟹ *0* < *size-td a*
by (*fastforce simp*: *td-set-wf-size-desc wf-size-desc-gt dest*!: *td-set-field-lookupD*)

lemma *field-lookup-inject''*:
∀ *f g m s*. *wf-size-desc t* ⟶ *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *Some s* ∧ *field-lookup t g m* = *Some s* ⟶ *f*=*g*
∀ *f g m s*. *wf-size-desc-struct st* ⟶ *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *Some s* ∧ *field-lookup-struct st g m* = *Some s* ⟶ *f*=*g*
∀ *f g m s*. *wf-size-desc-list ts* ⟶ *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* =

```
Some s ∧ field-lookup-list ts g m = Some s ⟶ f=g
  ∀f g m s. wf-size-desc-tuple x ⟶ field-lookup-tuple (x::('a,'b) typ-tuple) f m =
Some s ∧ field-lookup-tuple x g m = Some s ⟶ f=g
proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by clarsimp fast
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply clarsimp
    subgoal for f g m a b
      apply(clarsimp split: option.splits)
        apply fast
      subgoal
        apply(frule td-set-tuple-field-lookup-tupleD)
        apply(drule field-lookup-offset-le)
        apply(drule td-set-tuple-offset-size-m)
        subgoal premises prems
          using prems(3−8)
          apply(cases dt-tuple, simp)
          subgoal premises prems
          proof −
            have 0 < size-td a
              using prems
              apply −
             apply(clarsimp split: if-split-asm; drule (2) field-lookup-wf-size-desc-gt)
              done
            then show ?thesis
              using prems
              by simp
          qed
          done
        done
      subgoal
        apply(frule td-set-tuple-field-lookup-tupleD)
        apply(drule field-lookup-offset-le)
        apply(drule td-set-tuple-offset-size-m)
        subgoal premises prems
          using prems(3−8)
          apply(cases dt-tuple, simp)
```

```
        subgoal premises prems
        proof −
          have 0 < size-td a
            using prems
            apply −
           apply(clarsimp split: if-split-asm; drule (2) field-lookup-wf-size-desc-gt)
            done
          then show ?thesis
            using prems
            by simp
        qed
        done
      done
    subgoal by best
    done
  done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply clarsimp
    subgoal for f g m a b
      apply(drule spec [where x=tl f])
      apply(drule spec [where x=tl g])
      apply(cases f; simp)
      apply(cases g; simp)
      apply fastforce
      done
    done
qed

lemma field-lookup-inject′:
  (∀f g m s. wf-size-desc t ⟶ field-lookup (t::('a,'b) typ-desc) f m = Some s ∧
field-lookup t g m = Some s ⟶ f=g) ∧
      (∀f g m s. wf-size-desc-struct st ⟶ field-lookup-struct (st::('a,'b) typ-struct)
f m = Some s ∧ field-lookup-struct  st g m = Some s ⟶ f=g) ∧
       (∀f g m s. wf-size-desc-list ts ⟶ field-lookup-list (ts::('a,'b) typ-tuple list) f
m = Some s ∧ field-lookup-list ts g m = Some s ⟶ f=g) ∧
      (∀f g m s. wf-size-desc-tuple x ⟶ field-lookup-tuple (x::('a,'b) typ-tuple) f m
= Some s ∧ field-lookup-tuple x g m = Some s ⟶ f=g)
  by (auto simp: field-lookup-inject′′)


lemma field-lookup-inject:
  ⟦ field-lookup (t::('a,'b) typ-desc) f m = Some s;
     field-lookup t g m = Some s; wf-size-desc t ⟧ ⟹ f=g
  using field-lookup-inject′ [of t undefined undefined undefined]
  apply(cases s)
  apply clarsimp
  apply(drule spec [where x=f])
```

apply(*drule spec* [**where** *x*=*g*])
apply *fast*
done

lemma *fd-cons-update-normalise*:
⟦ *fd-cons-update-access d n*; *fd-cons-access-update d n*;
fd-cons-double-update d; *fd-cons-length d n* ⟧ ⟹
fd-cons-update-normalise d n
apply(*clarsimp simp*: *fd-cons-update-access-def fd-cons-update-normalise-def norm-desc-def*)
subgoal for *v bs*
apply(*drule spec* [**where** *x*=*field-update d bs v*])
apply(*drule spec* [**where** *x*=*replicate* (*length bs*) *0*])
apply *clarsimp*
apply(*clarsimp simp*: *fd-cons-access-update-def*)
apply(*drule spec* [**where** *x*=*bs*])
apply *clarsimp*
apply(*drule spec* [**where** *x*=*replicate* (*length bs*) *0*])
apply *clarsimp*
apply(*drule spec* [**where** *x*=*v*])
apply(*drule spec* [**where** *x*=*undefined*])
apply *clarsimp*
apply(*clarsimp simp*: *fd-cons-double-update-def*)
apply(*drule spec* [**where** *x*=*field-access d* (*field-update d bs undefined*)
(*replicate* (*length bs*) *0*)])
apply(*drule spec* [**where** *x*=*bs*])
apply *clarsimp*
apply(*erule impE*)
apply(*clarsimp simp*: *fd-cons-length-def*)
apply *clarsimp*
done
done

lemma *update-ti-t-update-ti-struct-t* [*simp*]:
update-ti-t (*TypDesc algn st tn*) = *update-ti-struct-t st*
by (*auto simp*: *update-ti-t-def update-ti-struct-t-def*)

lemma *fd-cons-fd-cons-struct* [*simp*]:
fd-cons (*TypDesc algn st tn*) = *fd-cons-struct st*
by (*clarsimp simp*: *fd-cons-def fd-cons-struct-def*)

lemma *update-ti-struct-t-update-ti-list-t* [*simp*]:
update-ti-struct-t (*TypAggregate ts*) = *update-ti-list-t ts*
by (*auto simp*: *update-ti-struct-t-def update-ti-list-t-def*)

lemma *fd-cons-struct-fd-cons-list* [*simp*]:
fd-cons-struct (*TypAggregate ts*) = *fd-cons-list ts*
by (*clarsimp simp*: *fd-cons-struct-def fd-cons-list-def*)

lemma *fd-cons-list-empty* [*simp*]:
fd-cons-list []
by (*clarsimp simp*: *fd-cons-list-def fd-cons-double-update-def*
fd-cons-update-access-def fd-cons-access-update-def fd-cons-length-def
fd-cons-update-normalise-def update-ti-list-t-def fd-cons-desc-def)

lemma *fd-cons-double-update-list-append*:
⟦ *fd-cons-double-update* (*field-desc-list xs*);
fd-cons-double-update (*field-desc-list ys*);
fu-commutes (*field-update* (*field-desc-list xs*)) (*field-update* (*field-desc-list ys*))
⟧ ⟹
fd-cons-double-update (*field-desc-list* (*xs*@*ys*))
by (*auto simp*: *fd-cons-double-update-def fu-commutes-def*)

lemma *fd-cons-update-access-list-append*:
⟦ *fd-cons-update-access* (*field-desc-list xs*) (*size-td-list xs*);
fd-cons-update-access (*field-desc-list ys*) (*size-td-list ys*);
fd-cons-length (*field-desc-list xs*) (*size-td-list xs*);
fd-cons-length (*field-desc-list ys*) (*size-td-list ys*) ⟧ ⟹
fd-cons-update-access (*field-desc-list* (*xs*@*ys*)) (*size-td-list* (*xs*@*ys*))
by (*auto simp*: *fd-cons-update-access-def fd-cons-length-def access-ti-append*)

lemma *min-ll*:
min (*x* + *y*) *x* = (*x*::*nat*)
by *simp*

lemma *fd-cons-access-update-list-append*:
⟦ *fd-cons-access-update* (*field-desc-list xs*) (*size-td-list xs*);
fd-cons-access-update (*field-desc-list ys*) (*size-td-list ys*);
fu-commutes (*field-update* (*field-desc-list xs*)) (*field-update* (*field-desc-list ys*))
⟧ ⟹
fd-cons-access-update (*field-desc-list* (*xs*@*ys*)) (*size-td-list* (*xs*@*ys*))
apply(*clarsimp simp*: *fd-cons-access-update-def*)
subgoal for *bs bs′ v v′*
apply(*drule spec* [**where** *x*=*take* (*size-td-list xs*) *bs*])
apply *clarsimp*
apply(*simp add*: *access-ti-append*)
apply(*drule spec* [**where** *x*=*drop* (*size-td-list xs*) *bs*])
apply *clarsimp*
apply(*drule spec* [**where** *x*=*take* (*size-td-list xs*) *bs′*])
apply(*simp add*: *min-ll*)
apply(*drule spec* [**where** *x*=*update-ti-list-t ys* (*drop* (*size-td-list xs*) *bs*) *v*])
apply(*drule spec* [**where** *x*=*update-ti-list-t ys* (*drop* (*size-td-list xs*) *bs*) *v′*])
apply *simp*
apply(*frule fu-commutes*[**where** *bs*=*take* (*size-td-list xs*) *bs* **and**
bs′=*drop* (*size-td-list xs*) *bs* **and** *v*=*v*])
apply *clarsimp*
apply(*frule fu-commutes*[**where** *bs*=*take* (*size-td-list xs*) *bs* **and**

bs'=drop (size-td-list xs) bs **and** *v=v'*])
apply *clarsimp*
done
done

lemma *fd-cons-length-list-append*:
⟦ *fd-cons-length* (*field-desc-list xs*) (*size-td-list xs*);
fd-cons-length (*field-desc-list ys*) (*size-td-list ys*) ⟧ ⟹
fd-cons-length (*field-desc-list* (*xs*@*ys*)) (*size-td-list* (*xs*@*ys*))
by (*auto simp*: *fd-cons-length-def access-ti-append*)

lemma *wf-fdp-insert*:
wf-fdp (*insert x xs*) ⟹ *wf-fdp* {*x*} ∧ *wf-fdp xs*
by (*auto simp*: *wf-fdp-def*)

lemma *wf-fdp-fd-cons*:
⟦ *wf-fdp X*; (*t*,*m*) ∈ *X* ⟧ ⟹ *fd-cons t*
by (*auto simp*: *wf-fdp-def*)

lemma *wf-fdp-fu-commutes*:
⟦ *wf-fdp X*; (*s*,*m*) ∈ *X*; (*t*,*n*) ∈ *X*; ¬ *m* ≤ *n*; ¬ *n* ≤ *m* ⟧ ⟹
fu-commutes (*field-update* (*field-desc s*)) (*field-update* (*field-desc t*))
by (*auto simp*: *wf-fdp-def*)

lemma *wf-fdp-fa-fu-ind*:
⟦ *wf-fdp X*; (*s*,*m*) ∈ *X*; (*t*,*n*) ∈ *X*; ¬ *m* ≤ *n*; ¬ *n* ≤ *m* ⟧ ⟹
fa-fu-ind (*field-desc s*) (*field-desc t*) (*size-td t*) (*size-td s*)
by (*auto simp*: *wf-fdp-def*)

lemma *wf-fdp-mono*:
⟦ *wf-fdp Y*; *X* ⊆ *Y* ⟧ ⟹ *wf-fdp X*
by (*fastforce simp*: *wf-fdp-def*)

lemma *tf0* [*simp*]:
tf-set (*TypDesc algn st nm*) = {(*TypDesc algn st nm*,[])} ∪ *tf-set-struct st*
by (*auto simp*: *tf-set-def tf-set-struct-def*)

lemma *tf1* [*simp*]: *tf-set-struct* (*TypScalar m algn d*) = {}
by (*clarsimp simp*: *tf-set-struct-def*)

lemma *tf2* [*simp*]: *tf-set-struct* (*TypAggregate xs*) = *tf-set-list xs*
by (*auto simp*: *tf-set-struct-def tf-set-list-def*)

lemma *tf3* [*simp*]: *tf-set-list* [] = {}
by (*simp add*: *tf-set-list-def*)

lemma *tf4*: *tf-set-list* (*x*#*xs*) = *tf-set-tuple x* ∪ {*t*. *t* ∈ *tf-set-list xs* ∧ *snd t* ∉ *snd* ‘ *tf-set-tuple x*}
apply(*clarsimp simp*: *tf-set-list-def tf-set-tuple-def*)

```
  apply(cases x)
  apply clarsimp
  subgoal for x1 a b
    apply(rule equalityI; clarsimp)
    subgoal for a' b'
      apply(cases b'; clarsimp)
      apply(erule disjE; clarsimp?)
      apply(fastforce dest: field-lookup-list-offset2 split: option.splits)[1]
      done
    apply (rule conjI; clarsimp)
    subgoal for   - ys n
      apply(cases ys; clarsimp split: option.splits)
      apply(rule conjI; clarsimp simp: image-def)
       apply(drule field-lookup-list-offset3[where  m=size-td x1])
       apply fast
      apply(drule field-lookup-list-offset3[where  m=size-td x1])
      apply fast
      done
    done
  done

lemma tf4D:
  t ∈ tf-set-list (x#xs) ⟹ t ∈ (tf-set-tuple x ∪ tf-set-list xs)
  by (clarsimp simp: tf4)

lemma tf4' [simp]:  wf-desc-list (x#xs) ⟹
  tf-set-list (x#xs) = tf-set-tuple x ∪ tf-set-list xs
  apply(simp add: tf4)
  apply(rule equalityI; clarsimp)
  subgoal for - - ys
  apply(clarsimp simp: tf-set-tuple-def tf-set-list-def)
  apply(cases x, simp)
  apply(clarsimp split: if-split-asm)
    apply(cases ys; simp)
    subgoal for - - - - - list
      apply(drule field-lookup-list-None [where fs=list and m=0])
      apply fastforce
      done
    done
  done

lemma tf5 [simp]:  tf-set-tuple (DTuple t m d) = {(a,m#b) | a b. (a,b) ∈ tf-set t}
  apply(clarsimp simp: tf-set-tuple-def tf-set-def)
  apply(rule equalityI; clarsimp)
  subgoal for - xs n
  apply(cases xs; clarsimp)
    done
  done
```

lemma *tf-set-self* [*simp*]:
(*t*,[]) ∈ *tf-set t*
by (*clarsimp simp*: *tf-set-def*)

lemma *tf-set-list-mem*:
wf-desc-list ts ⟹ *DTuple t n d* ∈ *set ts* ⟹ (*t*,[*n*]) ∈ *tf-set-list ts*
by (*induct ts*) *auto*

lemma *tf-set-list-append*:
wf-desc-list (*xs*@*ys*) ⟹ *tf-set-list* (*xs*@*ys*) = *tf-set-list xs* ∪ *tf-set-list ys*
apply(*induct xs*; *clarsimp*)
apply(*subst tf4'*; *fastforce*)
done

lemma *lf-set-list-append* [*simp*]:
lf-set-list (*xs*@*ys*) *fn* = *lf-set-list xs fn* ∪ *lf-set-list ys fn*
by (*induct xs*) *auto*

lemma *ti-ind-sym*:
ti-ind X Y ⟹ *ti-ind Y X*
by (*auto simp*: *ti-ind-def fu-commutes-def*)

lemma *ti-ind-sym2*:
ti-ind X Y = *ti-ind Y X*
by (*blast dest*: *ti-ind-sym*)

lemma *ti-ind-list* [*simp*]:
ti-ind (*X* ∪ *Y*) *Z* = (*ti-ind X Z* ∧ *ti-ind Y Z*)
unfolding *ti-ind-def* **by** *auto*

lemma *ti-empty* [*simp*]:
ti-ind {} *X*
by (*simp add*: *ti-ind-def*)

lemma *wf-lf-list*:
lf-fn ‘ *X* ∩ *lf-fn* ‘ *Y* = {} ⟹
wf-lf (*X* ∪ *Y*) = (*wf-lf X* ∧ *wf-lf Y* ∧ *ti-ind X Y*)
unfolding *wf-lf-def ti-ind-def field-desc-def fu-commutes-def*
apply (*rule iffI*; *clarsimp*)
subgoal for *x y*
apply(*frule spec* [**where** *x*=*x*])
apply(*drule spec* [**where** *x*=*y*])
apply *clarsimp*
apply(*drule spec* [**where** *x*=*y*])
apply(*drule spec* [**where** *x*=*x*])
apply *clarsimp*
apply *fastforce*
done

subgoal by *fast*
done

lemma *wf-lf-listD*:
wf-lf $(X \cup Y) \Longrightarrow$ *wf-lf* $X \wedge$ *wf-lf* Y
unfolding *wf-lf-def ti-ind-def field-desc-def fu-commutes-def* **by** *clarsimp*

lemma *ti-ind-fn*:
fixes *t*::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
ts::('*a*,'*b*) *typ-info-tuple list* **and**
x::('*a*,'*b*) *typ-info-tuple*
shows
$\forall$ *fn. ti-ind* (*lf-set t fn*) *Y* = *ti-ind* (*lf-set t* []) *Y*
$\forall$ *fn. ti-ind* (*lf-set-struct st fn*) *Y* = *ti-ind* (*lf-set-struct st* []) *Y*
$\forall$ *fn. ti-ind* (*lf-set-list ts fn*) *Y* = *ti-ind* (*lf-set-list ts* []) *Y*
$\forall$ *fn. ti-ind* (*lf-set-tuple x fn*) *Y* = *ti-ind* (*lf-set-tuple x* []) *Y*
apply(*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp*›)
apply (*fastforce simp*: *ti-ind-def*)
apply *auto*
done

lemma *ti-ind-ld-td'*:
fixes *t*::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
ts::('*a*,'*b*) *typ-info-tuple list* **and**
x::('*a*,'*b*) *typ-info-tuple*
shows
ti-ind (*lf-set t* []) *Y* $\longrightarrow$ *ti-ind* {*t2d* (*t*,[])} *Y*
ti-ind (*lf-set-struct st* []) *Y* $\longrightarrow$ *ti-ind* {⦇ *lf-fd* = *field-desc-struct st*, *lf-sz* = *size-td-struct st*, *lf-fn* = [] ⦈} *Y*
ti-ind (*lf-set-list ts* []) *Y* $\longrightarrow$ *ti-ind* {⦇ *lf-fd* = *field-desc-list ts*, *lf-sz* = *size-td-list ts*, *lf-fn* = [] ⦈} *Y*
ti-ind (*lf-set-tuple x* []) *Y* $\longrightarrow$ *ti-ind* {⦇ *lf-fd* = *field-desc-tuple x*, *lf-sz* = *size-td-tuple x*, *lf-fn* = [] ⦈} *Y*
apply(*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp simp*: *t2d-def*›)
apply((*clarsimp simp*: *ti-ind-def fu-commutes-def fa-fu-ind-def*)+)[*3*]
apply(*subst* (*asm*) *ti-ind-fn*)
apply *simp*
done

lemma *ti-ind-ld-td-struct*:
ti-ind (*lf-set-struct st fn*) *Y* $\Longrightarrow$
ti-ind {⦇ *lf-fd* = *field-desc-struct st*, *lf-sz* = *size-td-struct st*, *lf-fn* = [] ⦈} *Y*
by (*subst* (*asm*) *ti-ind-fn*) (*simp only*: *ti-ind-ld-td'*)

lemma *ti-ind-ld-td-list*:
ti-ind (*lf-set-list ts fn*) *Y* $\Longrightarrow$
ti-ind {⦇ *lf-fd* = *field-desc-list ts*, *lf-sz* = *size-td-list ts*, *lf-fn* = [] ⦈} *Y*

by (*subst* (*asm*) *ti-ind-fn*) (*simp only*: *ti-ind-ld-td'*)

lemma *ti-ind-ld-td-tuple*:
ti-ind (*lf-set-tuple x fn*) *Y* ⟹
ti-ind {⦇ *lf-fd* = *field-desc-tuple x*, *lf-sz* = *size-td-tuple x*, *lf-fn* = [] ⦈} *Y*
by (*subst* (*asm*) *ti-ind-fn*) (*simp only*: *ti-ind-ld-td'*)

lemma *ti-ind-ld'*:
fixes *t*::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
ts::('*a*,'*b*) *typ-info-tuple list* **and**
x::('*a*,'*b*) *typ-info-tuple*
shows
ti-ind (*lf-set t* []) *Y* ⟶ *ti-ind* (*t2d* ‘ (*tf-set t*)) *Y*
ti-ind (*lf-set-struct st* []) *Y* ⟶ *ti-ind* (*t2d* ‘ (*tf-set-struct st*)) *Y*
ti-ind (*lf-set-list ts* []) *Y* ⟶ *ti-ind* (*t2d* ‘ (*tf-set-list ts*)) *Y*
ti-ind (*lf-set-tuple x* []) *Y* ⟶ *ti-ind* (*t2d* ‘ (*tf-set-tuple x*)) *Y*
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case*
apply *clarsimp*
apply(*frule ti-ind-ld-td-struct*)
apply(*subst insert-def*)
apply(*subst ti-ind-list*)
apply(*clarsimp simp*: *ti-ind-def t2d-def*)
done
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **apply** *clarsimp*
apply(*clarsimp simp*: *ti-ind-def t2d-def*)
apply(*drule tf4D*)
apply *clarsimp*
by (*smt* (*verit*, *best*) *field-desc.select-convs*(*2*) *fst-eqD image-eqI*
leaf-desc.select-convs(*1*) *leaf-desc.select-convs*(*2*))
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case*
apply *clarsimp*
apply(*subst* (*asm*) (*2*) *ti-ind-fn*)
apply(*simp add*: *t2d-def*)

apply(*clarsimp simp*: *ti-ind-def*)
by (*metis* (*mono-tags*, *lifting*) *field-desc.select-convs*(*2*) *fst-conv image-eqI*
leaf-desc.select-convs(*1*) *leaf-desc.select-convs*(*2*))
qed

lemma *ti-ind-ld*:
ti-ind (*lf-set t fn*) *Y* ⟹ *ti-ind* (*t2d* ‘ (*tf-set t*)) *Y*
by (*subst* (*asm*) *ti-ind-fn*) (*simp only*: *ti-ind-ld′*)

lemma *ti-ind-ld-struct*:
ti-ind (*lf-set-struct t fn*) *Y* ⟹ *ti-ind* (*t2d* ‘ (*tf-set-struct t*)) *Y*
by (*subst* (*asm*) *ti-ind-fn*) (*simp only*: *ti-ind-ld′*)

lemma *ti-ind-ld-list*:
ti-ind (*lf-set-list t fn*) *Y* ⟹ *ti-ind* (*t2d* ‘ (*tf-set-list t*)) *Y*
by (*subst* (*asm*) *ti-ind-fn*) (*simp only*: *ti-ind-ld′*)

lemma *ti-ind-ld-tuple*:
ti-ind (*lf-set-tuple t fn*) *Y* ⟹ *ti-ind* (*t2d* ‘ (*tf-set-tuple t*)) *Y*
by (*subst* (*asm*) *ti-ind-fn*) (*simp only*: *ti-ind-ld′*)

lemma *lf-set-fn′*:
fixes *t*::(′*a*,′*b*) *typ-info* **and**
st::(′*a*,′*b*) *typ-info-struct* **and**
ts::(′*a*,′*b*) *typ-info-tuple list* **and**
x::(′*a*,′*b*) *typ-info-tuple*
shows
∀ *s fn*. *s* ∈ *lf-set t fn* ⟶ *fn* ≤ *lf-fn s*
∀ *s fn*. *s* ∈ *lf-set-struct st fn* ⟶ *fn* ≤ *lf-fn s*
∀ *s fn*. *s* ∈ *lf-set-list ts fn* ⟶ *fn* ≤ *lf-fn s*
∀ *s fn*. *s* ∈ *lf-set-tuple x fn* ⟶ *fn* ≤ *lf-fn s*
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*DTuple-typ-desc typ-desc list b*)
then show *?case*
apply *clarsimp*
subgoal for *s fn*
apply(*drule spec* [**where** *x*=*s*])
apply(*drule spec* [**where** *x*=*fn* @ [*list*]])
apply(*clarsimp simp*: *prefix-def less-eq-list-def*)
done
done
qed *auto*

lemma *lf-set-fn*:
s ∈ *lf-set* (*t*::(′*a*,′*b*) *typ-info*) *fn* ⟹ *fn* ≤ *lf-fn s*
by (*clarsimp simp*: *lf-set-fn′*)

lemma *ln-fn-disj*:
dt-snd x ∉ *dt-snd* ‘ *set xs* ⟹ *lf-fn* ‘ *lf-set-tuple x fn* ∩ *lf-fn* ‘ *lf-set-list xs fn* = {}

apply (*induct xs arbitrary*: *x*; *clarsimp*)
apply (*rule set-eqI*, *clarsimp*)
apply (*erule disjE*)
apply (*clarsimp dest*!: *lf-set-fn simp*: *split-DTuple-all prefix-def less-eq-list-def*)
apply *fastforce*
done

lemma *wf-lf-fn*:
fixes *t*::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
ts::('*a*,'*b*) *typ-info-tuple list* **and**
x::('*a*,'*b*) *typ-info-tuple*
shows
∀*fn*. *wf-desc t* ⟶ *wf-lf* (*lf-set t fn*) = *wf-lf* (*lf-set t* [])
∀*fn*. *wf-desc-struct st* ⟶ *wf-lf* (*lf-set-struct st fn*) = *wf-lf* (*lf-set-struct st* [])
∀*fn*. *wf-desc-list ts* ⟶ *wf-lf* (*lf-set-list ts fn*) = *wf-lf* (*lf-set-list ts* [])
∀*fn*. *wf-desc-tuple x* ⟶ *wf-lf* (*lf-set-tuple x fn*) = *wf-lf* (*lf-set-tuple x* [])
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case*
apply *clarify*
subgoal for *fn*
apply(*drule spec* [**where** *x*=*fn*])
apply *clarsimp*
done
done
next
case (*TypScalar nat1 nat2 a*)
then show *?case*
apply *clarsimp*
apply(*clarsimp simp*: *wf-lf-def*)
done
next
case (*TypAggregate list*)
then show *?case*
apply *clarify*
subgoal for *fn*
apply(*drule spec* [**where** *x*=*fn*])
apply *clarsimp*
done
done
next
case *Nil-typ-desc*
then show *?case* **by** *clarsimp*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply *clarify*
subgoal for *fn*

```
    apply(drule spec [where x=fn])+
    apply clarsimp
    apply(subst wf-lf-list)
     apply(erule ln-fn-disj)
    apply(subst wf-lf-list)
     apply(erule ln-fn-disj)
    apply clarsimp
    apply(subst ti-ind-fn)
    apply(subst ti-ind-sym2)
    apply(subst ti-ind-fn)
    apply(subst ti-ind-sym2)
    apply(clarsimp)
    done
  done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply clarify
    subgoal for fn
      apply(frule spec [where x=[list]])
      apply(drule spec [where x=fn@[list]])
      apply clarsimp
      done
    done
qed

lemma wf-lf-fd-cons':
  fixes t::('a,'b) typ-info and
    st::('a,'b) typ-info-struct and
    ts::('a,'b) typ-info-tuple list and
    x::('a,'b) typ-info-tuple
  shows
  ∀ m. wf-lf (lf-set t []) ⟶ wf-desc t ⟶ fd-cons t
  ∀ m. wf-lf (lf-set-struct st []) ⟶ wf-desc-struct st ⟶ fd-cons-struct st
  ∀ m. wf-lf (lf-set-list ts []) ⟶ wf-desc-list ts ⟶ fd-cons-list ts
  ∀ m. wf-lf (lf-set-tuple x []) ⟶ wf-desc-tuple x ⟶ fd-cons-tuple x
proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by clarsimp
next
  case (TypScalar nat1 nat2 a)
  then show ?case
    apply(clarsimp simp: wf-lf-def fd-cons-struct-def fd-cons-def)
    apply(clarsimp simp: fd-cons-desc-def fd-cons-double-update-def fd-cons-update-access-def
                         fd-cons-access-update-def fd-cons-length-def)
    done
next
  case (TypAggregate list)
  then show ?case by clarsimp
```

next
case *Nil-typ-desc*
then show *?case* **by** *clarsimp*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply *clarsimp*
apply(*subst* (*asm*) *wf-lf-list*)
apply(*erule ln-fn-disj*)
apply *clarsimp*
apply(*drule ti-ind-ld-td-tuple*)
apply(*drule ti-ind-sym*)
apply(*drule ti-ind-ld-td-list*)
apply(*drule ti-ind-sym*)
apply(*clarsimp simp*: *ti-ind-def*)
apply(*clarsimp simp*: *fd-cons-list-def fd-cons-desc-def*)
apply(*rule conjI*)
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-double-update-def fd-cons-desc-def*)
apply(*simp add*: *fu-commutes-def*)
apply(*rule conjI*)
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-update-access-def fd-cons-desc-def*)
apply(*clarsimp simp*: *fd-cons-length-def*)
apply(*rule conjI*)
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-desc-def*)
apply(*clarsimp simp*: *fd-cons-access-update-def*)
subgoal for *bs bs′ v v′*
apply(*simp add*: *fu-commutes-def*)
apply(*clarsimp simp*: *fa-fu-ind-def*)
subgoal premises *prems*
using *prems* (*1−14*, *16−18*)
prems (*15*)[*rule-format*, **where** *bs′*= *take* (*size-td-tuple dt-tuple*) *bs′* **and**
bs= *take* (*size-td-tuple dt-tuple*) *bs* **and** *v*=*v* **and** *v′*=*v′*]
apply −
apply(*simp add*: *min-ll*)
done
done
apply(*clarsimp simp*: *fd-cons-length-def fd-cons-tuple-def fd-cons-desc-def*)
done
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case*
apply *clarsimp*
apply(*subst* (*asm*) (*2*) *wf-lf-fn*, *assumption*)
apply(*clarsimp simp*: *fd-cons-def fd-cons-tuple-def export-uinfo-def*)
done
qed

lemma *wf-lf-fd-cons*:
⟦ *wf-lf* (*lf-set t fn*); *wf-desc t* ⟧ ⟹ *fd-cons t*

by (*subst* (*asm*) *wf-lf-fn*; *simp only*: *wf-lf-fd-cons'*)

lemma *wf-lf-fd-cons-struct*:
⟦ *wf-lf* (*lf-set-struct t fn*); *wf-desc-struct t* ⟧ ⟹ *fd-cons-struct t*
by (*subst* (*asm*) *wf-lf-fn*; *simp only*: *wf-lf-fd-cons'*)

lemma *wf-lf-fd-cons-list*:
⟦ *wf-lf* (*lf-set-list t fn*); *wf-desc-list t* ⟧ ⟹ *fd-cons-list t*
by (*subst* (*asm*) *wf-lf-fn*; *simp only*: *wf-lf-fd-cons'*)

lemma *wf-lf-fd-cons-tuple*:
⟦ *wf-lf* (*lf-set-tuple t fn*); *wf-desc-tuple t* ⟧ ⟹ *fd-cons-tuple t*
by (*subst* (*asm*) *wf-lf-fn*; *simp only*: *wf-lf-fd-cons'*)

lemma *wf-lf-fdp'*:
∀ *m*. *wf-lf* (*lf-set* (*t*::('*a*,'*b*) *typ-info*) []) ⟶ *wf-desc t* ⟶ *wf-fdp* (*tf-set t*)
∀ *m*. *wf-lf* (*lf-set-struct* (*st*::('*a*,'*b*) *typ-info-struct*) []) ⟶ *wf-desc-struct st* ⟶ *wf-fdp* (*tf-set-struct st*)
∀ *m*. *wf-lf* (*lf-set-list* (*ts*::('*a*,'*b*) *typ-info-tuple list*) []) ⟶ *wf-desc-list ts* ⟶ *wf-fdp* (*tf-set-list ts*)
∀ *m*. *wf-lf* (*lf-set-tuple* (*x*::('*a*,'*b*) *typ-info-tuple*) []) ⟶ *wf-desc-tuple x* ⟶ *wf-fdp* (*tf-set-tuple x*)
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case*
apply (*clarsimp simp*: *wf-fdp-def*)
apply (*fastforce elim*: *wf-lf-fd-cons-struct*)
done
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** (*clarsimp simp*: *wf-fdp-def*)
next
case (*TypAggregate list*)
then show *?case* **by** (*clarsimp simp*: *wf-fdp-def*)
next
case *Nil-typ-desc*
then show *?case* **by** (*clarsimp simp*: *wf-fdp-def*)
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*clarsimp simp*: *wf-fdp-def*)
apply(*subst* (*asm*) *wf-lf-list*)
apply(*erule ln-fn-disj*)
apply *clarsimp*
apply(*drule ti-ind-ld-tuple*)
apply(*drule ti-ind-sym*)
apply(*drule ti-ind-ld-list*)
apply(*drule ti-ind-sym*)

```
    apply(rule conjI, clarsimp)
     apply(erule disjE, fast)
     apply(clarsimp simp: ti-ind-def t2d-def)
    subgoal for x m y n
      apply(drule spec [where x=t2d (x,m)])
      apply(drule spec [where x=t2d (y,n)])
      apply(clarsimp simp: t2d-def image-def)
      apply(erule impE)
       apply(rule conjI)
        apply(rule bexI [where x=(x,m)])
         apply clarsimp
        apply assumption
       apply(rule bexI [where x=(y,n)])
        apply clarsimp
       apply assumption
      apply clarsimp
      done

    apply clarsimp
    subgoal for x m y n
      apply(erule disjE)
       apply(clarsimp simp: ti-ind-def t2d-def)
       apply(drule spec [where x=t2d (y,n)])
       apply(drule spec [where x=t2d (x,m)])
       apply(clarsimp simp: t2d-def image-def)
       apply(erule impE)
        apply(standard)
         apply(rule bexI [where x=(y,n)])
          apply clarsimp
         apply assumption
        apply(rule bexI [where x=(x,m)])
         apply clarsimp
        apply assumption
       apply clarsimp
       apply(clarsimp simp: fu-commutes-def)
      apply fast
      done
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply (clarsimp simp: wf-fdp-def)
  apply(subst (asm) (2) wf-lf-fn, assumption)
  apply(clarsimp simp: wf-fdp-def)
    apply fast
    done
qed

lemma wf-lf-fdp:
```

⟦ *wf-lf* (*lf-set t* []); *wf-desc t* ⟧ ⟹ *wf-fdp* (*tf-set t*)
by (*simp only*: *wf-lf-fdp'*)

lemma *wf-fd-field-lookup* [*rule-format*]:
∀ *f m n s*. *wf-fd* (*t*::('*a*,'*b*) *typ-info*) ⟶ *field-lookup t f m* = *Some* (*s*,*n*) ⟶ *wf-fd s*
∀ *f m n s*. *wf-fd-struct* (*st*::('*a*,'*b*) *typ-info-struct*) ⟶ *field-lookup-struct st f m* = *Some* (*s*,*n*) ⟶ *wf-fd s*
∀ *f m n s*. *wf-fd-list* (*ts*::('*a*,'*b*) *typ-info-tuple list*) ⟶ *field-lookup-list ts f m* = *Some* (*s*,*n*) ⟶ *wf-fd s*
∀ *f m n s*. *wf-fd-tuple* (*x*::('*a*,'*b*) *typ-info-tuple*) ⟶ *field-lookup-tuple x f m* = *Some* (*s*,*n*) ⟶ *wf-fd s*
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*clarsimp split*: *option.splits*)+

lemma *wf-fd-field-lookupD*:
⟦ *field-lookup t f m* = *Some* (*s*,*n*); *wf-fd t* ⟧ ⟹ *wf-fd s*
by (*rule wf-fd-field-lookup*)

lemma *wf-fd-tf-set*:
⟦ *wf-fd t*; ((*s*::('*a*,'*b*) *typ-info*),*m*) ∈ *tf-set t* ⟧ ⟹ *wf-fd s*
by (*fastforce simp*: *tf-set-def wf-fd-field-lookupD*)

lemma *tf-set-field-lookupD*:
field-lookup t f m = *Some* (*s*,*n*) ⟹ (*s*,*f*) ∈ *tf-set t*
unfolding *tf-set-def*
by (*clarsimp simp flip*: *field-lookup-offset*[**where** *m*=*m*] *dest*!: *field-lookup-offset-le*) *arith*

lemma *fu-commutes-ts*:
(⋀ *t*. *t* ∈ *dt-fst* ' *set ts* ⟹ *fu-commutes d* (*update-ti-t t*)) ⟹
fu-commutes d (*update-ti-list-t ts*)
by (*induct ts*; *clarsimp simp*: *fu-commutes-def*) (*clarsimp simp*: *split-DTuple-all*)

lemma *fa-fu-ind-ts*:
(⋀*t*. *t* ∈ *dt-fst* ' *set ts* ⟹ *fa-fu-ind d* (*field-desc t*) (*size-td t*) *n*) ⟹
fa-fu-ind d (| *field-access* = *access-ti-list ts*,
field-update = *update-ti-list-t ts*, *field-sz* = *size-td-list ts*|)
(*size-td-list ts*) *n*
by (*induct ts*; *clarsimp simp*: *fa-fu-ind-def*) (*clarsimp simp*: *split-DTuple-all*)

lemma *fa-fu-ind-ts2*:
(⋀*t*. *t* ∈ *dt-fst* ' *set ts* ⟹ *fa-fu-ind* (*field-desc t*) *d n* (*size-td t*)) ⟹
fa-fu-ind (| *field-access* = *access-ti-list ts*,
field-update = *update-ti-list-t ts*, *field-sz* = *size-td-list ts*|) *d*
n (*size-td-list ts*)
by (*induct ts*; *clarsimp simp*: *fa-fu-ind-def*) (*clarsimp simp*: *split-DTuple-all*)

lemma *wf-fdp-fd* [*rule-format*]:
∀ *m*. *wf-fdp* (*tf-set t*) ⟶ *wf-desc t* ⟶ *wf-fd* (*t*::('*a*,'*b*) *typ-info*)

∀ *m*. (*case st of TypScalar sz algn d* ⇒ *fd-cons-struct* ((*TypScalar sz algn d*)::('*a*,'*b*) *typ-info-struct*)
| - ⇒ *wf-fdp* (*tf-set-struct st*)) ⟶ *wf-desc-struct st* ⟶ *wf-fd-struct* (*st*::('*a*,'*b*) *typ-info-struct*)
∀ *m*. *wf-fdp* (*tf-set-list ts*) ⟶ *wf-desc-list ts* ⟶ *wf-fd-list* (*ts*::('*a*,'*b*) *typ-info-tuple list*)
∀ *m*. *wf-fdp* (*tf-set-tuple x*) ⟶ *wf-desc-tuple x* ⟶ *wf-fd-tuple* (*x*::('*a*,'*b*) *typ-info-tuple*)
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case*
apply(*clarsimp split*: *typ-struct.split-asm*)
apply(*clarsimp simp*: *wf-fdp-def fd-cons-def fd-cons-struct-def*)
apply (*fastforce dest*: *wf-fdp-mono*)
done
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple xs*)
then show *?case*
apply *clarsimp*
apply (*rule conjI*, *fastforce dest*: *wf-fdp-mono*)
apply (*rule conjI*)
subgoal
using *wf-fdp-mono*
by *blast*
subgoal
apply(*clarsimp simp*: *wf-fdp-def*)
apply(*cases dt-tuple*, *clarsimp*)
subgoal for *a b x3*

apply(*frule spec* [**where** *x*=*a*])
apply(*drule spec* [**where** *x*=[*b*]])
apply *clarsimp*
apply (*rule conjI*)
apply(*rule fu-commutes-ts*)
apply *clarsimp*
subgoal for *x*
apply(*drule spec* [**where** *x*=*dt-fst x*], *erule impE*, *rule exI* [**where** *x*=[*dt-snd x*]])
apply (*metis Prefix-Order.Cons-prefix-Cons dt-tuple.exhaust-sel rev-image-eqI tf-set-list-mem*)
apply *simp*

```
        done
      apply(rule conjI)
       apply(rule fa-fu-ind-ts)
       apply clarsimp
      subgoal for x

      apply(drule spec [where x=dt-fst x], erule impE, rule exI [where x=[dt-snd
x]])
      apply (metis Prefix-Order.Cons-prefix-Cons dt-tuple.exhaust-sel rev-image-eqI
tf-set-list-mem)
        apply simp
        done

      apply(rule fa-fu-ind-ts2)
      subgoal for t
        apply(drule spec [where x=t])
        apply clarsimp
        subgoal for x
          apply(cases x, clarsimp)
          subgoal for ba aa
            apply(drule spec [where x=[ba]])
            apply clarsimp
            apply(simp add: tf-set-list-mem)
            apply clarsimp
         by (metis Prefix-Order.Cons-prefix-Cons dt-tuple.sel(2) imageI tf-set-self)

          done
        done
      done
    done
  done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply(clarsimp simp: wf-fdp-def)
    subgoal for x m
      apply(drule spec [where x=x])
      apply(drule spec [where x=list#m])
      apply clarsimp
      subgoal for y n
        apply(drule spec [where x=y])
        apply auto
        done
      done
    done
qed

lemma wf-fdp-fdD:
  ⟦ wf-fdp (tf-set t); wf-desc t ⟧ ⟹ wf-fd (t::('a,'b) typ-info)
```

by (*rule wf-fdp-fd*)

lemma *wf-fdp-fd-listD*:
⟦ *wf-fdp* (*tf-set-list t*); *wf-desc-list t* ⟧ ⟹ *wf-fd-list t*
by (*rule wf-fdp-fd*)

lemma *fd-consistentD*:
⟦ *field-lookup t f 0* = *Some* (*s*,*n*); *fd-consistent t* ⟧
⟹ *fd-cons s*
by (*fastforce simp*: *fd-consistent-def*)

lemma *wf-fd-cons-access-update'* [*rule-format*]:
wf-fd (*t*::('*a*,'*b*) *typ-info*) ⟶ *fd-cons-access-update* (*field-desc t*) (*size-td t*)
wf-fd-struct (*st*::('*a*,'*b*) *typ-info-struct*) ⟶ *fd-cons-access-update* (*field-desc-struct st*) (*size-td-struct st*)
wf-fd-list (*ts*::('*a*,'*b*) *typ-info-tuple list*) ⟶ *fd-cons-access-update* (*field-desc-list ts*) (*size-td-list ts*)
wf-fd-tuple (*x*::('*a*,'*b*) *typ-info-tuple*) ⟶ *fd-cons-access-update* (*field-desc-tuple x*) (*size-td-tuple x*)
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*clarsimp split*: *typ-struct.splits*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case*
apply (*clarsimp split*: *typ-struct.splits*)
apply(*clarsimp simp*: *fd-cons-access-update-def fd-cons-struct-def fd-cons-desc-def*)
done
next
case (*TypAggregate list*)
then show *?case* **by** (*clarsimp split*: *typ-struct.splits*)
next
case *Nil-typ-desc*
then show *?case* **apply** (*clarsimp split*: *typ-struct.splits*)
apply(*clarsimp simp*: *fd-cons-access-update-def*)
done
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*clarsimp split*: *typ-struct.splits*)
apply(*clarsimp simp*: *fd-cons-access-update-def*)
apply(*simp add*: *fu-commutes-def*)
apply(*clarsimp simp*: *fa-fu-ind-def*)
by (*metis* (*no-types*, *lifting*) *add-diff-cancel-left' length-drop length-take min-ll*)
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** (*clarsimp split*: *typ-struct.splits*)
qed

lemma *wf-fd-cons-access-updateD*:
wf-fd t $\Longrightarrow$ *fd-cons-access-update* (*field-desc t*) (*size-td t*)
by (*rule wf-fd-cons-access-update'*)

lemma *wf-fd-cons-access-update-structD*:
wf-fd-struct t $\Longrightarrow$ *fd-cons-access-update* (*field-desc-struct t*) (*size-td-struct t*)
by (*rule wf-fd-cons-access-update'*)

lemma *wf-fd-cons-access-update-listD*:
wf-fd-list t $\Longrightarrow$ *fd-cons-access-update* (*field-desc-list t*) (*size-td-list t*)
by (*rule wf-fd-cons-access-update'*)

lemma *wf-fd-cons-access-update-tupleD*:
wf-fd-tuple t $\Longrightarrow$ *fd-cons-access-update* (*field-desc-tuple t*) (*size-td-tuple t*)
by (*rule wf-fd-cons-access-update'*)

lemma *wf-fd-norm-tu*:
$\forall$ *bs*. *wf-fd t* $\longrightarrow$ *length bs* = *size-td t* $\longrightarrow$ *norm-tu* (*export-uinfo* (*t*::('*a*,'*b*) *typ-info*)) *bs* = (*access-ti t* (*update-ti-t t bs undefined*) (*replicate* (*size-td t*) *0*))
$\forall$ *bs*. *wf-fd-struct st* $\longrightarrow$ *length bs* = *size-td-struct st* $\longrightarrow$ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) (*st*::('*a*,'*b*) *typ-info-struct*)) *bs* = (*access-ti-struct st* (*update-ti-struct-t st bs undefined*) (*replicate* (*size-td-struct st*) *0*))
$\forall$ *bs*. *wf-fd-list ts* $\longrightarrow$ *length bs* = *size-td-list ts* $\longrightarrow$ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) (*ts*::('*a*,'*b*) *typ-info-tuple list*)) *bs* = (*access-ti-list ts* (*update-ti-list-t ts bs undefined*) (*replicate* (*size-td-list ts*) *0*))
$\forall$ *bs*. *wf-fd-tuple x* $\longrightarrow$ *length bs* = *size-td-tuple x* $\longrightarrow$ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) (*x*::('*a*,'*b*) *typ-info-tuple*)) *bs* = (*access-ti-tuple x* (*update-ti-tuple-t x bs undefined*) (*replicate* (*size-td-tuple x*) *0*))
apply(*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp simp*: *export-uinfo-def*›)
apply(*simp add*: *field-norm-def*)
apply(*simp add*: *fu-commutes-def*)
apply(*simp add*: *fa-fu-ind-def*)
apply(*drule wf-fd-cons-access-update-listD*)
apply(*clarsimp simp*: *fd-cons-access-update-def export-uinfo-def min-def*)
done

lemma *wf-fd-norm-tuD*:
⟦ *wf-fd t*; *length bs* = *size-td t* ⟧ $\Longrightarrow$ *norm-tu* (*export-uinfo t*) *bs* = (*access-ti*$_0$ *t* (*update-ti-t t bs undefined*))
using *wf-fd-norm-tu*(*1*) [*of t*] **by** (*clarsimp simp*: *access-ti*$_0$*-def*)

lemma *wf-fd-norm-tu-structD*:
⟦ *wf-fd-struct t*; *length bs* = *size-td-struct t* ⟧ $\Longrightarrow$ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *t*) *bs* = (*access-ti-struct t* (*update-ti-struct-t t bs undefined*) (*replicate* (*size-td-struct t*) *0*))
using *wf-fd-norm-tu*(*2*) [*of t*] **by** *clarsimp*

lemma *wf-fd-norm-tu-listD*:

⟦ *wf-fd-list t*; *length bs = size-td-list t* ⟧ ⟹ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *t*) *bs* =
(*access-ti-list t* (*update-ti-list-t t bs undefined*) (*replicate* (*size-td-list t*) *0*))
using *wf-fd-norm-tu*(*3*) [*of t*] **by** *clarsimp*

lemma *wf-fd-norm-tu-tupleD*:
⟦ *wf-fd-tuple t*; *length bs = size-td-tuple t* ⟧ ⟹ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *t*) *bs* =
(*access-ti-tuple t* (*update-ti-tuple-t t bs undefined*) (*replicate* (*size-td-tuple t*) *0*))
using *wf-fd-norm-tu*(*4*) [*of t*] **by** *clarsimp*

lemma *wf-fd-cons* [*rule-format*]:
wf-fd t ⟶ *fd-cons* (*t*::('*a*,'*b*) *typ-info*)
wf-fd-struct st ⟶ *fd-cons-struct* (*st*::('*a*,'*b*) *typ-info-struct*)
wf-fd-list ts ⟶ *fd-cons-list* (*ts*::('*a*,'*b*) *typ-info-tuple list*)
wf-fd-tuple x ⟶ *fd-cons-tuple* (*x*::('*a*,'*b*) *typ-info-tuple*)
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply *clarsimp*
apply(*clarsimp simp*: *fd-cons-list-def fd-cons-desc-def*)
apply (*rule conjI*)
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-double-update-def fd-cons-desc-def*)
apply(*simp add*: *fu-commutes-def*)
apply (*rule conjI*)
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-update-access-def fd-cons-desc-def*)
apply(*clarsimp simp*: *fd-cons-length-def*)
apply (*rule conjI*)
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-desc-def*)
apply(*clarsimp simp*: *fd-cons-access-update-def fu-commutes-def fa-fu-ind-def*
)
apply (*smt* (*verit*) *diff-add-inverse length-drop length-take min-ll*)
apply(*clarsimp simp*: *fd-cons-length-def fd-cons-tuple-def fd-cons-desc-def*)
done
next
case (*DTuple-typ-desc typ-desc list b*)

then show *?case* **by**(*clarsimp simp*: *fd-cons-def fd-cons-tuple-def export-uinfo-def*)
qed

lemma *wf-fd-consD*:
wf-fd t $\Longrightarrow$ *fd-cons t*
by (*rule wf-fd-cons*)

lemma *wf-fd-cons-structD*:
wf-fd-struct t $\Longrightarrow$ *fd-cons-struct t*
by (*rule wf-fd-cons*)

lemma *wf-fd-cons-listD*:
wf-fd-list t $\Longrightarrow$ *fd-cons-list t*
by (*rule wf-fd-cons*)

lemma *wf-fd-cons-tupleD*:
wf-fd-tuple t $\Longrightarrow$ *fd-cons-tuple t*
by (*rule wf-fd-cons*)

lemma *fd-cons-list-append*:
⟦ *wf-fd-list xs*; *wf-fd-list ys*; *fu-commutes*
(*field-update* (*field-desc-list xs*)) (*field-update* (*field-desc-list ys*)) ⟧ $\Longrightarrow$
fd-cons-list (*xs*@*ys*)
apply(*frule wf-fd-cons-listD*)
apply(*frule wf-fd-cons-listD* [**where** *t*=*ys*])
apply(*unfold fd-cons-list-def fd-cons-desc-def*)
apply(*fastforce intro*: *fd-cons-double-update-list-append fd-cons-update-access-list-append*
fd-cons-access-update-list-append fd-cons-length-list-append)
done

lemma (**in** *wf-type*) *wf-fd* [*simp*]:
wf-fd (*typ-info-t TYPE*('*a*))
by (*fastforce intro*: *wf-fdp-fdD wf-lf-fdp wf-lf*)

lemma (**in** *wf-type*) *fd-cons* [*simp*]:
fd-consistent (*typ-info-t TYPE*('*a*))
unfolding *fd-consistent-def* **by** (*fastforce intro*: *wf-fd-consD wf-fd-field-lookupD*)

lemma (**in** *wf-type*) *field-lvalue-append* [*simp*]:
⟦ *field-ti TYPE*('*a*) *f* = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('*b*::*c-type*);
field-ti TYPE('*b*) *g* = *Some k* ⟧ $\Longrightarrow$
&(((*Ptr* &((*p*::'*a ptr*)$\rightarrow$*f*))::'*b ptr*)$\rightarrow$*g*) = &(*p*$\rightarrow$*f*@*g*)
apply(*clarsimp simp*: *field-lvalue-def c-type-class.field-ti-def field-ti-def field-offset-def*
field-offset-untyped-def typ-uinfo-t-def
split: *option.splits*)
apply(*subst field-lookup-prefix-Some'*)

apply(*fastforce dest*: *field-lookup-export-uinfo-Some*)
apply(*simp add*: *export-uinfo-def wf-desc-map*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add*: *export-uinfo-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add*: *export-uinfo-def*)
apply (*simp add*: *c-type-class.field-lvalue-def c-type-class.field-offset-def*
c-type-class.typ-uinfo-t-def export-uinfo-def field-lookup-offset-shift' field-offset-untyped-def)
done

lemma (**in** *wf-type*) *field-lvalue-cons-unfold'*:
— rhs contains additional type variable *'b*, hence simplifier won't apply this rule
⟦ *field-ti TYPE('a) [f] = Some t*;
export-uinfo t = typ-uinfo-t TYPE('b::c-type);
field-ti TYPE('b) g = Some k ⟧ ⟹
&(p→f#g) = &(((Ptr &((p::'a ptr)→[f]))::'b ptr)→g)
using *field-lvalue-append* [**where** *f=[f]* **and** *g=g*]
by *simp*

lemma (**in** *mem-type*) *field-lvalue-cons-unfold*:
⟦*field-ti TYPE('b) g = Some k*;
export-uinfo t = export-uinfo (typ-info-t TYPE('b::c-type));
field-ti TYPE('a) [f] = Some t⟧ ⟹
&(p→f#g) ≡ &(((Ptr &((p::'a ptr)→[f]))::'b ptr)→g)
using *field-lvalue-cons-unfold'* [*simplified c-type-class.typ-uinfo-t-def*]
apply −
apply (*rule eq-reflection*)
apply *blast*
done

lemma *field-access-update-take-drop* [*rule-format*]:
∀ *f s m n bs bs' v. field-lookup t f m = Some (s,m+n)* ⟶
length bs = size-td t ⟶ *length bs' = size-td s* ⟶ *wf-fd t* ⟶
field-access (field-desc s) (field-update (field-desc t) bs v) bs'
= field-access (field-desc s) (field-update (field-desc s)
(take (size-td (s::('a,'b) typ-info)) (drop n bs)) undefined) bs'
∀ *f s m n bs bs' v. field-lookup-struct st f m = Some (s,m+n)* ⟶
length bs = size-td-struct st ⟶ *length bs' = size-td s* ⟶ *wf-fd-struct st* ⟶
field-access (field-desc s) (field-update (field-desc-struct st) bs v) bs'
= field-access (field-desc s) (field-update (field-desc s)
(take (size-td (s::('a,'b) typ-info)) (drop n bs)) undefined) bs'
∀ *f s m n bs bs' v. field-lookup-list ts f m = Some (s,m+n)* ⟶
length bs = size-td-list ts ⟶ *length bs' = size-td s* ⟶ *wf-fd-list ts* ⟶
field-access (field-desc s) (field-update (field-desc-list ts) bs v) bs'
= field-access (field-desc s) (field-update (field-desc s)
(take (size-td (s::('a,'b) typ-info)) (drop n bs)) undefined) bs'
∀ *f s m n bs bs' v. field-lookup-tuple x f m = Some (s,m+n)* ⟶

```
    length bs = size-td-tuple x ⟶ length bs' = size-td s ⟶ wf-fd-tuple x ⟶
    field-access (field-desc s) (field-update (field-desc-tuple x) bs v) bs'
      = field-access (field-desc s) (field-update (field-desc s)
        (take (size-td (s::('a,'b) typ-info)) (drop n bs)) undefined) bs'
proof (induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case
    apply clarsimp
    apply(fastforce dest!: wf-fd-cons-structD
      simp: fd-cons-struct-def fd-cons-desc-def fd-cons-access-update-def)
    done
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply clarsimp
    subgoal for f s m n bs bs' v
      apply(clarsimp simp: fd-cons-desc-def split: option.splits)
       apply(cases f; clarsimp)
      subgoal premises prems for a lista
        using prems (1−7, 10−12)
          prems(9) [rule-format, where f= a#lista and s=s and m= m + size-td
(dt-fst dt-tuple) and n=n − size-td (dt-fst dt-tuple)]
        apply clarsimp
        apply(frule field-lookup-offset-le)
        apply simp
        apply(cases dt-tuple, clarsimp simp: fu-commutes-def)
        done
      apply(cases f; clarsimp)
      subgoal for a lista
        apply(drule spec [where x=a#lista])
        apply(drule spec [where x=s])
        apply(drule spec [where x=m])
        apply(drule spec [where x=n])
        apply clarsimp
        apply(frule field-lookup-offset-le)
        apply simp
        apply(cases dt-tuple, clarsimp)
        apply(clarsimp split: if-split-asm)
          by(fastforce dest: td-set-field-lookupD td-set-offset-size-m simp: ac-simps
drop-take min-def)
```

done
done
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** *auto*
qed

lemma *field-access-update-take-dropD*:
⟦ *field-lookup t f m = Some (s,m+n)*; *length bs = size-td t*;
length bs′ = size-td s; *wf-fd t* ⟧ ⟹
field-access (*field-desc s*) (*field-update* (*field-desc t*) *bs v*) *bs′*
= *field-access* (*field-desc s*) (*field-update* (*field-desc s*)
(*take* (*size-td* (*s*::(′*a*,′*b*) *typ-info*)) (*drop n bs*)) *undefined*) *bs′*
by (*rule field-access-update-take-drop*)

lemma (**in** *wf-type*) *fi-fa-consistentD*:
⟦ *field-lookup* (*typ-info-t TYPE*(′*a*)) *f 0 = Some* (*d*,*n*);
length bs = size-of TYPE(′*a*) ⟧ ⟹
field-access (*field-desc d*) (*from-bytes bs*) (*replicate* (*size-td d*) *0*) =
norm-tu (*export-uinfo d*) (*take* (*size-td d*) (*drop n bs*))
apply(*clarsimp simp*: *field-offset-def from-bytes-def size-of-def*)
apply(*frule field-lookup-export-uinfo-Some*)
apply(*subst wf-fd-norm-tuD*)
apply(*fastforce intro*: *wf-fd-field-lookupD*)
apply(*clarsimp simp*: *min-def split*: *if-split-asm*)
apply(*drule td-set-field-lookupD*)
apply(*drule td-set-offset-size*)
apply *simp*
apply(*frule field-access-update-take-dropD*[**where** *v*=*undefined* **and** *m*=*0* **and**
bs′=*replicate* (*size-td d*) *0*,*simplified*];
simp?)
apply(*simp add*: *min-def access-ti*$_0$*-def*)
done

lemma *fi-fu-consistent* [*rule-format*]:
∀ *f m n s bs v w*. *field-lookup t f m = Some* (*s*,*n* + *m*) ⟶ *wf-fd t* ⟶
length bs = size-td t ⟶ *length v = size-td* (*s*::(′*a*,′*b*) *typ-info*) ⟶
field-update (*field-desc t*) (*super-update-bs v bs n*) *w* =
field-update (*field-desc s*) *v* (*field-update* (*field-desc t*) *bs w*)
∀ *f m n s bs v w*. *field-lookup-struct st f m = Some* (*s*,*n* + *m*) ⟶ *wf-fd-struct st*
⟶
length bs = size-td-struct st ⟶ *length v = size-td* (*s*::(′*a*,′*b*) *typ-info*) ⟶
field-update (*field-desc-struct st*) (*super-update-bs v bs n*) *w* =
field-update (*field-desc s*) *v* (*field-update* (*field-desc-struct st*) *bs w*)
∀ *f m n s bs v w*. *field-lookup-list ts f m = Some* (*s*,*n* + *m*) ⟶ *wf-fd-list ts* ⟶
length bs = size-td-list ts ⟶ *length v = size-td* (*s*::(′*a*,′*b*) *typ-info*) ⟶
field-update (*field-desc-list ts*) (*super-update-bs v bs n*) *w* =
field-update (*field-desc s*) *v* (*field-update* (*field-desc-list ts*) *bs w*)

```
  ∀f m n s bs v w. field-lookup-tuple x f m = Some (s,n + m) ⟶ wf-fd-tuple x ⟶
      length bs = size-td-tuple x ⟶ length v = size-td (s::('a,'b) typ-info) ⟶
      field-update (field-desc-tuple x) (super-update-bs v bs n) w =
          field-update (field-desc s) v (field-update (field-desc-tuple x) bs w)
proof(induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case
    apply clarsimp
    apply(drule wf-fd-cons-structD)
   apply(clarsimp simp: fd-cons-struct-def fd-cons-double-update-def super-update-bs-def
        fd-cons-desc-def)
    done
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply clarsimp
    subgoal for f m n s bs v w
      apply(cases f; clarsimp)
      apply(clarsimp simp: fd-cons-desc-def split: option.splits)
       apply(clarsimp simp: fu-commutes-def)
      subgoal premises prems for a lista
        using prems (1−8,11−12)
          prems(10)[rule-format, where f=a#lista and s= s and m=m + size-td
(dt-fst dt-tuple)
            and n= n − size-td (dt-fst dt-tuple)]
        apply −
        apply(frule field-lookup-offset-le)
        apply simp
        apply(drule td-set-list-field-lookup-listD)
        apply(drule td-set-list-offset-size-m)
        apply (cases dt-tuple)
        apply(clarsimp simp: drop-super-update-bs take-super-update-bs)
        done
      subgoal for a lista
        apply(simp add: fu-commutes-def)
        by (smt (verit, best) add.commute append-take-drop-id diff-add-inverse2
            length-append length-super-update-bs length-take min-ll
            super-update-bs-append-drop-first super-update-bs-append-take-first
            td-set-offset-size' td-set-tuple-field-lookup-tupleD)
      done
```

```
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed


lemma fi-fu-consistentD:
  ⟦ field-lookup t f 0 = Some (s,n); wf-fd t; length bs = size-td t;
      length v = size-td s ⟧ ⟹
      field-update (field-desc t) (super-update-bs v bs n) w =
          field-update (field-desc s) v (field-update (field-desc t) bs w)
  using fi-fu-consistent(1) [of t f 0] by clarsimp

lemma (in wf-type) norm:
  assumes lbs: length bs = size-of TYPE('a)
  shows from-bytes (norm-bytes TYPE('a) bs) = ((from-bytes bs)::'a)
proof −
  have wf-fd (typ-info-t TYPE('a))
    by simp
  with lbs  show ?thesis
    apply −
    apply(drule wf-fd-consD)
    apply(simp add: from-bytes-def norm-bytes-def)
    apply(clarsimp simp: fd-cons-def fd-cons-desc-def)
    apply(drule (3) fd-cons-update-normalise)
     apply(fastforce simp: fd-cons-update-normalise-def size-of-def wf-fd-norm-tuD
norm-desc-def
        access-ti$_0$-def)
    done
qed

lemma (in wf-type) len:
  length bs = size-of TYPE('a) ⟹
      length (to-bytes (x::'a) bs) = size-of TYPE('a)
  apply(simp add: size-of-def to-bytes-def)
  by (metis fd-cons-def fd-cons-desc-def fd-cons-length-def
      field-desc.simps(1) field-desc-def local.wf-fd wf-fd-consD)


lemma (in wf-type) sz-nzero:
  0 < size-of (TYPE('a))
  unfolding size-of-def
  by (simp add: wf-size-desc-gt)

lemma not-disj-fn-empty1 [simp]:
  ¬ disj-fn [] s
  by (simp add: disj-fn-def)
```

lemma *disj-fn-comm*: *disj-fn a b* $\longleftrightarrow$ *disj-fn b a*
by (*auto simp add*: *disj-fn-def*)

lemma *disj-fn-append-right*: *disj-fn x y* $\Longrightarrow$ *disj-fn x* (*y* @ *z*)
by (*auto simp*: *disj-fn-def less-eq-list-def prefix-def append-eq-append-conv2*)

lemma *disj-fn-cons-consI*[*simp*]: (*x* = *y* $\longrightarrow$ *disj-fn a b*) $\Longrightarrow$ *disj-fn* (*x* # *a*) (*y* # *b*)
by (*auto simp*: *disj-fn-def*)

lemma *fd-path-cons* [*simp*]:
f $\notin$ *fs-path* (*x*#*xs*) = (*disj-fn f x* $\wedge$ *f* $\notin$ *fs-path xs*)
by (*auto simp*: *fs-path-def disj-fn-def*)

lemma *fu-commutes-lookup-disjD*:
⟦ *field-lookup t f m* = *Some* (*d*,*n*); *field-lookup t f′ m′* = *Some* (*d′*,*n′*);
disj-fn f f′; *wf-fdp* (*tf-set t*) ⟧ $\Longrightarrow$
fu-commutes (*field-update* (*field-desc* (*d*::(*′a*,*′b*) *typ-info*)))
(*field-update* (*field-desc d′*))
by (*auto simp*: *disj-fn-def wf-fdp-def dest*!: *tf-set-field-lookupD*)

lemma *field-lookup-fa-fu-lhs*:
∀*f m n s d k*. *field-lookup t f m* = *Some* (*s*,*n*) $\longrightarrow$ *fa-fu-ind* (*field-desc t*) *d k* (*size-td t*)
$\longrightarrow$ *wf-fd t* $\longrightarrow$ *fa-fu-ind* (*field-desc* (*s*::(*′a*,*′b*) *typ-info*)) *d k* (*size-td s*)
∀*f m n s d k*. *field-lookup-struct st f m* = *Some* (*s*,*n*) $\longrightarrow$ *fa-fu-ind* (*field-desc-struct st*) *d k* (*size-td-struct st*)
$\longrightarrow$ *wf-fd-struct st* $\longrightarrow$ *fa-fu-ind* (*field-desc* (*s*::(*′a*,*′b*) *typ-info*)) *d k* (*size-td s*)
∀*f m n s d k*. *field-lookup-list ts f m* = *Some* (*s*,*n*) $\longrightarrow$ *fa-fu-ind* (*field-desc-list ts*) *d k* (*size-td-list ts*)
$\longrightarrow$ *wf-fd-list ts* $\longrightarrow$ *fa-fu-ind* (*field-desc* (*s*::(*′a*,*′b*) *typ-info*)) *d k* (*size-td s*)
∀*f m n s d k*. *field-lookup-tuple x f m* = *Some* (*s*,*n*) $\longrightarrow$ *fa-fu-ind* (*field-desc-tuple x*) *d k* (*size-td-tuple x*)
$\longrightarrow$ *wf-fd-tuple x* $\longrightarrow$ *fa-fu-ind* (*field-desc* (*s*::(*′a*,*′b*) *typ-info*)) *d k* (*size-td s*)
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
next
case (*TypAggregate list*)
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
next
case *Nil-typ-desc*
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
next
case (*Cons-typ-desc dt-tuple list*)

```
  then show ?case
    apply (clarsimp simp: fa-fu-ind-def)
    subgoal for  f m n s d v bs bs'
      apply(clarsimp split: option.splits)
      subgoal premises prems
        using prems (1−8, 10)
        apply −
        apply (rule
            prems (9)[rule-format, where f=f and s = s and m = m + size-td
(dt-fst dt-tuple)
            and n=n and d=d and k=length bs, OF prems(11), simplified])
        subgoal for v ys ys'
          apply(drule spec [where x=v])
          apply(drule spec [where x=ys])
          apply clarsimp
          apply(drule spec [where x=replicate (size-td-tuple dt-tuple) 0 @ ys'])
          apply clarsimp
          apply(drule wf-fd-cons-tupleD)
         apply(clarsimp simp: fd-cons-tuple-def fd-cons-length-def fd-cons-desc-def)
          done
        subgoal by clarsimp
        subgoal by fast
        done
      subgoal
        apply(drule spec [where x=f])
        apply(drule spec [where x=m])
        apply(drule spec [where x=n])
        apply(drule spec [where x=s])
        apply clarsimp
        apply(drule spec [where x=d])
        apply(drule spec [where x=length bs])
        apply(erule impE)
         apply clarsimp
        subgoal for  v ys ys'
          apply(drule spec [where x=v])
          apply(drule spec [where x=ys])
          apply clarsimp
          apply(drule spec [where x=ys'@replicate (size-td-list list) 0])
          apply clarsimp
          apply(drule wf-fd-cons-tupleD)
         apply(clarsimp simp: fd-cons-tuple-def fd-cons-length-def fd-cons-desc-def)
          done
        apply clarsimp
        done
      done
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by (clarsimp simp: fa-fu-ind-def)
```

qed

lemma *field-lookup-fa-fu-lhs-listD*:
⟦ *field-lookup-list ts f m* = *Some* (*s*,*n*); *fa-fu-ind* (*field-desc-list ts*) *d k* (*size-td-list ts*);
wf-fd-list ts ⟧ ⟹ *fa-fu-ind* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) *d k* (*size-td s*)
using *field-lookup-fa-fu-lhs*(*3*) [*of ts*] **by** *blast*

lemma *field-lookup-fa-fu-lhs-tupleD*:
⟦ *field-lookup-tuple x f m* = *Some* (*s*,*n*); *fa-fu-ind* (*field-desc-tuple x*) *d k* (*size-td-tuple x*);
wf-fd-tuple x ⟧ ⟹ *fa-fu-ind* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) *d k* (*size-td s*)
using *field-lookup-fa-fu-lhs*(*4*) [*of x*] **by** *blast*

lemma *field-lookup-fa-fu-rhs*:
∀ *f m n s d k* . *field-lookup t f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind d* (*field-desc t*) (*size-td t*) *k*
⟶ *wf-fd t* ⟶ *fa-fu-ind d* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
∀ *f m n s d k*. *field-lookup-struct st f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind d* (*field-desc-struct st*) (*size-td-struct st*) *k*
⟶ *wf-fd-struct st* ⟶ *fa-fu-ind d* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
∀ *f m n s d k*. *field-lookup-list ts f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind d* (*field-desc-list ts*) (*size-td-list ts*) *k*
⟶ *wf-fd-list ts* ⟶ *fa-fu-ind d* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
∀ *f m n s d k*. *field-lookup-tuple x f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind d* (*field-desc-tuple x*) (*size-td-tuple x*) *k*
⟶ *wf-fd-tuple x* ⟶ *fa-fu-ind d* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
next
case (*TypAggregate list*)
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
next
case *Nil-typ-desc*
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*clarsimp simp*: *fa-fu-ind-def*)
subgoal for *f m n s d v bs bs'*
apply(*clarsimp split*: *option.splits*)
subgoal premises *prems*
thm *prems*
using *prems* (*1*−*8*, *10*)
apply −

apply (*rule*
prems (*9*)[*rule-format*, **where** *f*=*f* **and** *s* = *s* **and** *m* = *m* + *size-td* (*dt-fst dt-tuple*)
and *n*=*n* **and** *d*=*d* **and** *k*=*length bs'*, *OF prems*(*11*), *simplified*])
subgoal for *v ys ys'*
apply(*drule spec* [**where** *x*=*v*])
apply(*drule spec* [**where** *x*=*access-ti-tuple dt-tuple v* (*replicate* (*size-td-tuple dt-tuple*) *0*)@*ys*])
apply *clarsimp*

apply(*drule wf-fd-cons-tupleD*)
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-length-def fd-cons-desc-def*)
apply(*drule spec* [**where** *x*=*ys'*])
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-length-def fd-cons-update-access-def fd-cons-desc-def*)
apply(*clarsimp simp*: *fu-commutes-def*)
done
subgoal by *clarsimp*
subgoal
apply(*drule spec* [**where** *x*=*f*])
apply(*drule spec* [**where** *x*=*m*])
apply(*drule spec* [**where** *x*=*n*])
apply(*drule spec* [**where** *x*=*s*])
apply *clarsimp*
done
done
subgoal
apply(*drule spec* [**where** *x*=*f*])
apply(*drule spec* [**where** *x*=*m*])
apply(*drule spec* [**where** *x*=*n*])
apply(*drule spec* [**where** *x*=*s*])
apply *clarsimp*
apply(*drule spec* [**where** *x*=*d*])
apply(*drule spec* [**where** *x*=*length bs'*])
apply(*erule impE*)
apply *clarsimp*
subgoal for *v ys ys'*
apply(*drule spec* [**where** *x*=*v*])
apply(*drule spec* [**where** *x*=*ys*@*access-ti-list list v* (*replicate* (*size-td-list list*) *0*)])
apply(*drule wf-fd-cons-tupleD*)
apply(*drule wf-fd-cons-listD*)
apply(*clarsimp simp*: *fd-cons-tuple-def fd-cons-list-def fd-cons-length-def fd-cons-update-access-def fd-cons-desc-def*)
done
apply *fastforce*
done
done
done

next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** (*clarsimp simp*: *fa-fu-ind-def*)
qed

lemma *field-lookup-fa-fu-rhs-listD*:
⟦ *field-lookup-list ts f m* = *Some* (*s*,*n*);
fa-fu-ind d (*field-desc-list ts*) (*size-td-list ts*) *k*; *wf-fd-list ts* ⟧ ⟹
fa-fu-ind d (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
using *field-lookup-fa-fu-rhs*(*3*) [*of ts*] **by** *blast*

lemma *field-lookup-fa-fu-rhs-tupleD*:
⟦ *field-lookup-tuple x f m* = *Some* (*s*,*n*);
fa-fu-ind d (*field-desc-tuple x*) (*size-td-tuple x*) *k*; *wf-fd-tuple x* ⟧ ⟹
fa-fu-ind d (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
using *field-lookup-fa-fu-rhs*(*4*) [*of x*] **by** *blast*

lemma *fa-fu-lookup-ind-list-tuple*:
shows
⟦ *field-lookup-tuple x f m* = *Some* (*d'*,*n*); *wf-fd-tuple x*;
field-lookup-list ts f' m' = *Some* (*d*,*n'*); *wf-fd-list ts*;
fa-fu-ind (*field-desc-list ts*) (*field-desc-tuple x*) (*size-td-tuple x*) (*size-td-list ts*)
⟧
⟹ *fa-fu-ind* (*field-desc d*) (*field-desc d'*) (*size-td d'*) (*size-td d*)
apply(*drule* (*2*) *field-lookup-fa-fu-lhs-listD*)
apply(*drule* (*3*) *field-lookup-fa-fu-rhs-tupleD*)
done

lemma *fa-fu-lookup-ind-tuple-list*:
⟦ *field-lookup-tuple x f m* = *Some* (*d*,*n*); *wf-fd-tuple x*;
field-lookup-list ts f' m' = *Some* (*d'*,*n'*); *wf-fd-list ts*;
fa-fu-ind (*field-desc-tuple x*) (*field-desc-list ts*) (*size-td-list ts*) (*size-td-tuple x*)
⟧
⟹ *fa-fu-ind* (*field-desc d*) (*field-desc d'*) (*size-td d'*) (*size-td d*)
apply(*drule* (*2*) *field-lookup-fa-fu-lhs-tupleD*)
apply(*drule* (*3*) *field-lookup-fa-fu-rhs-listD*)
done

lemma *fa-fu-lookup-disj*:
∀ *f m d n f' m' d' n'*. *field-lookup t f m* = *Some* (*d*,*n*) ⟶
field-lookup t f' m' = *Some* (*d'*,*n'*) ⟶ *disj-fn f f'* ⟶
wf-fd t ⟶ *fa-fu-ind* (*field-desc* (*d*::('*a*,'*b*) *typ-info*)) (*field-desc d'*) (*size-td d'*)
(*size-td d*)
∀ *f m d n f' m' d' n'*. *field-lookup-struct st f m* = *Some* (*d*,*n*) ⟶
field-lookup-struct st f' m' = *Some* (*d'*,*n'*) ⟶ *disj-fn f f'* ⟶
wf-fd-struct st ⟶ *fa-fu-ind* (*field-desc* (*d*::('*a*,'*b*) *typ-info*)) (*field-desc d'*)
(*size-td d'*) (*size-td d*)
∀ *f m d n f' m' d' n'*. *field-lookup-list ts f m* = *Some* (*d*,*n*) ⟶

```
      field-lookup-list ts f' m' = Some (d',n') ⟶ disj-fn f f' ⟶
     wf-fd-list ts ⟶ fa-fu-ind (field-desc (d::('a,'b) typ-info)) (field-desc d') (size-td
d') (size-td d)
  ∀f m d n f' m' d' n'. field-lookup-tuple x f m = Some (d,n) ⟶
      field-lookup-tuple x f' m' = Some (d',n') ⟶ disj-fn f f' ⟶
        wf-fd-tuple x ⟶ fa-fu-ind (field-desc (d::('a,'b) typ-info)) (field-desc d')
(size-td d') (size-td d)
proof (induct t and st and ts and x)
  case (TypDesc nat typ-struct list)
  then show ?case by (clarsimp simp: disj-fn-def)
next
  case (TypScalar nat1 nat2 a)
  then show ?case by (clarsimp simp: disj-fn-def)
next
  case (TypAggregate list)
  then show ?case by (clarsimp simp: disj-fn-def)
next
  case Nil-typ-desc
  then show ?case by (clarsimp simp: disj-fn-def)
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply (clarsimp simp: disj-fn-def)
    apply(clarsimp simp: split: option.splits)
    subgoal by blast
    subgoal
      by(drule (3) fa-fu-lookup-ind-list-tuple; simp)
    subgoal by (drule (3) fa-fu-lookup-ind-tuple-list; simp)
    subgoal for f m d n f' m' d' n'
      apply(drule spec [where x=f ])
      apply(drule spec [where x=m])
      apply(drule spec [where x=d])
      apply clarsimp
      by blast
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply(clarsimp simp: disj-fn-def)
    subgoal for f m d n f' m' d' n'
      apply(drule spec [where x=tl f])
      apply(drule spec [where x=m])
      apply(drule spec [where x=d])
      apply clarsimp
      apply(drule spec [where x=tl f'])
      apply(drule spec [where x=m'])
      apply(drule spec [where x=d'])
      apply clarsimp
      apply(cases f; clarsimp)
```

```
    apply(cases f'; clarsimp)
    done
  done
qed

lemma fa-fu-lookup-disjD:
  ⟦ field-lookup t f m = Some (d,n); field-lookup t f' m' = Some (d',n');
      disj-fn f f'; wf-fd t ⟧ ⟹
     fa-fu-ind (field-desc (d::('a,'b) typ-info)) (field-desc d') (size-td d') (size-td d)
 using fa-fu-lookup-disj(1) [of t] by fastforce

lemma field-access-update-disj:
  ⟦ field-lookup t f m = Some (d,n); field-lookup t f' m' = Some (d',n');
      disj-fn f f'; length bs = size-td d'; length bs' = size-td d; wf-fd t ⟧ ⟹
      access-ti d (update-ti-t d' bs v) bs' = access-ti d v bs'
  by (fastforce dest: fa-fu-lookup-disjD simp: fa-fu-ind-def)

lemma td-set-list-intvl-sub:
  (d,n) ∈ td-set-list t m ⟹ {of-nat n..+size-td d} ⊆ {of-nat m..+size-td-list t}
  apply(frule td-set-list-offset-le)
  apply(drule td-set-list-offset-size-m)
  apply clarsimp
  apply(drule intvlD, clarsimp)
  apply(clarsimp simp: intvl-def)
  subgoal for k
    apply(rule exI [where x=k + n − m])
    apply simp
    done
  done

lemma td-set-tuple-intvl-sub:
  (d,n) ∈ td-set-tuple t m ⟹ {of-nat n..+size-td d} ⊆ {of-nat m..+size-td-tuple
t}
  apply(frule td-set-tuple-offset-le)
  apply(drule td-set-tuple-offset-size-m)
  apply clarsimp
  apply(drule intvlD, clarsimp)
  apply(clarsimp simp: intvl-def)
  subgoal for k
    apply(rule exI [where x=k + n − m])
    apply simp
    done
  done

lemma intvl-inter-le:
  assumes inter: a + of-nat k = c + of-nat ka and lt-d: ka < d and lt-ka: k ≤
ka
  shows a ∈ {c..+d}
proof −
```

from *lt-ka inter* **have** $a = c + of\text{-}nat\ (ka - k)$ **by** (*simp add*: *field-simps*)
moreover from *lt-d* **have** $ka - k < d$ **by** *simp*
ultimately show *?thesis* **by** (*force simp*: *intvl-def*)
qed

lemma *intvl-inter*:
assumes *nondisj*: $\{a..+b\} \cap \{c..+d\} \neq \{\}$
shows $a \in \{c..+d\} \vee c \in \{a..+b\}$
proof −
from *nondisj* **obtain** *k ka* **where** $a + of\text{-}nat\ k = c + of\text{-}nat\ ka$
and $k < b$ **and** $ka < d$ **by** (*force simp*: *intvl-def*)
thus *?thesis* **by** (*force intro*: *intvl-inter-le*)
qed

lemma *init-intvl-disj*:
$k + z < addr\text{-}card \Longrightarrow \{(p::addr)+of\text{-}nat\ k..+z\} \cap \{p..+k\} = \{\}$
apply(*cases* $k \neq 0$; *simp*)
apply(*rule ccontr*)
apply(*drule intvl-inter*)
apply(*erule disjE*)
apply(*drule intvlD*, *clarsimp*)
apply(*metis add-lessD1 len-of-addr-card less-trans mod-less order-less-irrefl unat-of-nat*)
apply(*drule intvlD*, *clarsimp*)
apply(*subst* (*asm*) *Abs-fnat-homs*)
apply(*subst* (*asm*) *Word.of-nat-0*)
apply(*subst* (*asm*) *len-of-addr-card*)
apply *clarsimp*
subgoal for k' *q*
apply(*cases q*; *simp*)
done
done

lemma *final-intvl-disj*:
$[\![\ k + z \leq n;\ n < addr\text{-}card\ ]\!] \Longrightarrow$
$\{(p::addr)+of\text{-}nat\ k..+z\} \cap \{p+(of\text{-}nat\ k + of\text{-}nat\ z)..+n - (k+z)\} = \{\}$
apply(*cases* $z \neq 0$; *simp*)
apply(*rule ccontr*)
apply(*drule intvl-inter*)
apply(*erule disjE*)
apply(*drule intvlD*, *clarsimp*)
apply(*subst* (*asm*) *Abs-fnat-homs*)
apply(*subst* (*asm*) *Word.of-nat-0*)
apply(*subst* (*asm*) *len-of-addr-card*)
apply *clarsimp*
subgoal for k' *q*
apply(*cases q*; *simp*)
done
apply(*drule intvlD*, *clarsimp*)
apply(*subst* (*asm*) *word-unat.norm-eq-iff* [*symmetric*])

apply(*subst* (*asm*) *len-of-addr-card*)
apply *simp*
done

lemma *fa-fu-lookup-disj-inter*:
∀*f m d n f' d' n'*. *field-lookup t f m* = *Some* (*d*,*n*) ⟶
field-lookup t f' m = *Some* (*d'*,*n'*) ⟶ *disj-fn f f'* ⟶
wf-fd t ⟶ *size-td t* < *addr-card* ⟶
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} =
{}
∀*f m d n f' m d' n'*. *field-lookup-struct st f m* = *Some* (*d*,*n*) ⟶
field-lookup-struct st f' m = *Some* (*d'*,*n'*) ⟶ *disj-fn f f'* ⟶
wf-fd-struct st ⟶ *size-td-struct st* < *addr-card* ⟶
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} =
{}
∀*f m d n f' m d' n'*. *field-lookup-list ts f m* = *Some* (*d*,*n*) ⟶
field-lookup-list ts f' m = *Some* (*d'*,*n'*) ⟶ *disj-fn f f'* ⟶
wf-fd-list ts ⟶ *size-td-list ts* < *addr-card* ⟶
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} =
{}
∀*f m d n f' m d' n'*. *field-lookup-tuple x f m* = *Some* (*d*,*n*) ⟶
field-lookup-tuple x f' m = *Some* (*d'*,*n'*) ⟶ *disj-fn f f'* ⟶
wf-fd-tuple x ⟶ *size-td-tuple x* < *addr-card* ⟶
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} =
{}
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*clarsimp simp*: *disj-fn-def*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** (*clarsimp simp*: *disj-fn-def*)
next
case (*TypAggregate list*)
then show *?case* **by** (*clarsimp simp*: *disj-fn-def*)
next
case *Nil-typ-desc*
then show *?case* **by** (*clarsimp simp*: *disj-fn-def*)
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*clarsimp simp*: *disj-fn-def*)
apply(*rule set-eqI*)
apply(*clarsimp split*: *option.splits*)
apply *fastforce*
apply(*drule td-set-list-field-lookup-listD*)
apply(*drule td-set-list-intvl-sub*)
apply(*drule td-set-tuple-field-lookup-tupleD*)
apply(*drule td-set-tuple-intvl-sub*)
apply (*cases dt-tuple*)

apply(*fastforce dest*: *init-intvl-disj*)
apply(*drule td-set-list-field-lookup-listD*)
apply(*drule td-set-list-intvl-sub*)
apply(*drule td-set-tuple-field-lookup-tupleD*)
apply(*drule td-set-tuple-intvl-sub*)
apply (*cases dt-tuple*)
apply(*fastforce dest*: *init-intvl-disj*)
apply *fastforce*
done
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case*
apply (*clarsimp simp*: *disj-fn-def*)
apply(*rule set-eqI*, *clarsimp*)
subgoal for *f d n f' m d' n' x*
apply(*cases f*; *clarsimp*)
apply(*cases f'*; *clarsimp*)
apply(*fastforce simp*: *disj-fn-def*)
done
done
qed

lemma *fa-fu-lookup-disj-interD*:
⟦ *field-lookup t f m* = *Some* (*d*,*n*); *field-lookup t f' m* = *Some* (*d'*,*n'*);
disj-fn f f'; *wf-fd t*; *size-td t* < *addr-card* ⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} =
{}
using *fa-fu-lookup-disj-inter*(*1*) [*of t*] **by** *clarsimp*

lemma *fa-fu-lookup-disj-inter-listD*:
⟦ *field-lookup-list ts f m* = *Some* (*d*,*n*);
field-lookup-list ts f' m = *Some* (*d'*,*n'*); *disj-fn f f'*;
wf-fd-list ts; *size-td-list ts* < *addr-card* ⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩
{*of-nat n'*..+*size-td d'*} = {}
using *fa-fu-lookup-disj-inter*(*3*) [*of ts*] **by** *clarsimp*

lemma (**in** *mem-type-sans-size*) *upd-rf*:
length bs = *size-of TYPE*('*a*) ⟹
update-ti-t (*typ-info-t TYPE*('*a*)) *bs v*
= *update-ti-t* (*typ-info-t TYPE*('*a*)) *bs w*
by (*simp add*: *upd*)

lemma (**in** *mem-type-sans-size*) *inv*:
length bs = *size-of TYPE*('*a*) ⟹
from-bytes (*to-bytes* (*x*::'*a*) *bs*) = *x*
unfolding *from-bytes-def to-bytes-def*
by (*metis fd-cons-def fd-cons-desc-def fd-cons-update-access-def field-desc.simps*(*1*)

```
    field-desc.simps(2) field-desc-def local.len local.size-of-fold
    local.to-bytes-def local.upd local.wf-fd wf-fd-consD)

lemma (in mem-type) align:
  align-of (TYPE('a)) dvd addr-card
proof −
  have ∗: align-of TYPE('a) < addr-card
    by (smt (verit, best) int-dvd-int-iff less-imp-of-nat-less local.align-size-of
      local.max-size local.sz-nzero of-nat-0-less-iff of-nat-less-imp-less zdvd-not-zless)
  from ∗ show ?thesis
    apply −
    apply(subst (asm) align-of-def)
    apply(clarsimp simp: dvd-def align-of-def)
    apply(rule exI [where x=2^(len-of TYPE(addr-bitsize) − align-td (typ-info-t
TYPE('a)))])
    apply clarsimp
    subgoal premises prems
    proof −

      from prems
      have align-td (typ-info-t TYPE('a)) < addr-bitsize
        apply −
        apply(rule power-less-imp-less-exp [where a=2]; simp add: addr-card)
        done
      then show ?thesis
        apply(simp add: addr-card flip: power-add)
        done
    qed
    done
qed

lemma (in mem-type) to-bytes-inj:
  to-bytes (v::'a) = to-bytes (v'::'a) ⟹ v=v'
  apply (drule fun-cong [where x=replicate (size-of TYPE('a)) 0])
  apply (drule arg-cong [where f=from-bytes::byte list ⇒ 'a])
  apply (simp add: inv)
  done

lemmas unat-simps = unat-simps' max-size

lemmas (in mem-type) mem-type-simps [simp] = inv len sz-nzero max-size align
lemmas mem-type-simps [simp] = inv len sz-nzero max-size align

lemma (in mem-type) ptr-aligned-plus:
  assumes aligned: ptr-aligned (p::'a ptr)
  shows ptr-aligned (p +_p i)
proof −
  have int (align-of TYPE('a)) dvd (i ∗ int (size-of TYPE('a)))
    by (simp add: align-size-of)
```

with *aligned* **show** *?thesis*
apply (*cases p*, *simp add*: *ptr-aligned-def ptr-add-def scast-id*)
apply (*simp only*: *unat-simps len-signed*)
apply (*metis align align-size-of dvd-add dvd-mod dvd-mult2 mult.commute*)
done
qed

lemma (**in** *mem-type*) *mem-type-self* [*simp*]:
ptr-val (*p*::*'a ptr*) ∈ {*ptr-val p*..+*size-of TYPE*(*'a*)}
by (*rule intvl-self*, *rule sz-nzero*)

lemma (**in** *mem-type*) *intvl-Suc-nmem* [*simp*]:
(*p*::*addr*) ∉ {*p* + *1*..+*size-of TYPE*(*'a*) − *Suc 0*}
by (*rule intvl-Suc-nmem'*, *subst len-of-addr-card*, *rule max-size*)

lemma (**in** *mem-type*) *wf-size-desc-typ-uinfo-t-simp* [*simp*]:
wf-size-desc (*typ-uinfo-t TYPE*(*'a*))
by (*simp add*: *typ-uinfo-t-def export-uinfo-def wf-size-desc-map*)

lemma *aggregate-map* [*simp*]:
aggregate (*map-td f g t*) = *aggregate t*
apply(*cases t*)
subgoal for *x1 st n*
apply(*cases st*; *simp*)
done
done

lemma (**in** *simple-mem-type*) *simple-tag-not-aggregate2* [*simp*]:
typ-uinfo-t TYPE(*'a*) ≠ *TypDesc algn* (*TypAggregate ts*) *tn*
by (*metis aggregate.simps aggregate-map aggregate-struct.simps*(*2*) *export-uinfo-def local.simple-tag local.typ-uinfo-t-def*)

lemma (**in** *simple-mem-type*) *simple-tag-not-aggregate3* [*simp*]:
typ-info-t TYPE(*'a*) ≠ *TypDesc algn* (*TypAggregate ts*) *tn*
by (*metis aggregate.simps aggregate-struct.simps*(*2*) *simple-tag*)

lemma (**in** *mem-type*) *field-of-t-mem*:
field-of-t (*p*::*'a ptr*) (*q*::*'b*::*mem-type ptr*) ⟹
ptr-val p ∈ {*ptr-val q*..+*size-of TYPE*(*'b*)}
apply(*clarsimp simp*: *field-of-t-def field-of-def intvl-def*)
apply(*rule exI* [**where** *x*=*unat* (*ptr-val p* − *ptr-val q*)])
apply *simp*
apply(*drule td-set-offset-size*)
apply(*clarsimp simp*: *size-of-def*)
by (*metis* (*mono-tags*, *lifting*) *add-leD2 add-le-same-cancel2 c-type-class.size-of-def leD local.size-of-def local.sz-nzero nat-less-le*)

lemma *map-td-map*:
map-td f p (*map-td g q t*) = *map-td* (λ*n algn*. *f n algn o g n algn*) (*p o q*) *t*
map-td-struct f p (*map-td-struct g q st*) = *map-td-struct* (λ*n algn*. *f n algn o g n algn*) (*p o q*) *st*
map-td-list f p (*map-td-list g q ts*) = *map-td-list* (λ*n algn*. *f n algn o g n algn*) (*p o q*) *ts*
map-td-tuple f p (*map-td-tuple g q x*) = *map-td-tuple* (λ*n algn*. *f n algn o g n algn*) (*p o q*) *x*
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *field-of-t-simple*:
field-of-t p (*x*::'*a*::*simple-mem-type ptr*) ⟹ *ptr-val p* = *ptr-val x*
apply(*clarsimp simp*: *field-of-t-def*)
apply(*cases typ-uinfo-t TYPE*('*a*))
subgoal for *x1 st n*

apply(*cases st*; *clarsimp*)
apply(*clarsimp simp*: *field-of-def unat-eq-zero*)
done
done

lemma *fold-td'-unfold*:
fold-td' t =
(*let* (*f*,*s*) = *t in*
case s of TypDesc algn' st nm ⇒
(*case st of*
TypScalar n algn d ⇒ *d*
| *TypAggregate ts* ⇒ *f nm* (*map* (λ*x*. *case x of DTuple t n d* ⇒ (*fold-td'* (*f*,*t*),*n*)) *ts*)))
apply (*cases t*, *simp*)
subgoal for *a b*
apply (*cases b*, *simp*)
done
done

lemma *fold-td-alt-def'*:
fold-td f t = (*case t of*
TypDesc algn' st nm ⇒
(*case st of*
TypScalar n algn d ⇒ *d*
| *TypAggregate ts* ⇒ *f nm* (*map* (λ*x*. (*fold-td f* (*dt-fst x*),*dt-snd x*)) *ts*)))
apply(*cases t*)
apply(*clarsimp split*: *typ-desc.split typ-struct.splits dt-tuple.splits*)
by (*metis* (*no-types*, *lifting*) *dt-tuple.case dt-tuple.collapse*)

lemma *fold-td-alt-def*:
fold-td f t ≡ (*case t of*

TypDesc algn' st nm ⇒
(case st of
TypScalar n algn d ⇒ d
| TypAggregate ts ⇒ f nm (map (λx. (fold-td f (dt-fst x),dt-snd x)) ts)))
by (*fastforce simp*: *fold-td-alt-def' simp del*: *fold-td-def*)

lemma *map-td'-map'*:
map-td f g t = (map-td' ((f,g),t))
TypDesc algn (map-td-struct f g st) (typ-name t) = (map-td' ((f,g),TypDesc algn st (typ-name t)))
TypDesc algn (TypAggregate (map-td-list f g ts)) (typ-name t) = map-td' ((f,g),TypDesc algn (TypAggregate ts) (typ-name t))
map-td-tuple f g x = DTuple (map-td' ((f,g),dt-fst x)) (dt-snd x) (g (dt-trd x))
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*auto simp*: *split-DTuple-all*)

lemma *map-td'-map*:
map-td f g t = (case t of TypDesc algn st nm ⇒ TypDesc algn (case st of
TypScalar n algn d ⇒ TypScalar n algn (f n algn d) |
TypAggregate ts ⇒ TypAggregate (map (λx. DTuple (map-td f g (dt-fst x)) (dt-snd x) (g (dt-trd x))) ts)) nm)
apply(*subst map-td'-map'*)
apply(*subst map-td'-map'*)
apply(*cases t*, *simp add*: *typ-struct.splits*)
apply(*auto simp*: *split-DTuple-all*)
done

lemma *map-td-alt-def*:
map-td f g t ≡ (case t of TypDesc algn st nm ⇒ TypDesc algn (case st of
TypScalar n algn d ⇒ TypScalar n algn (f n algn d) |
TypAggregate ts ⇒ TypAggregate (map (λx. DTuple (map-td f g (dt-fst x)) (dt-snd x) (g (dt-trd x))) ts)) nm)
by (*simp add*: *map-td'-map*)

lemma *size-td-fm'*:
size-td (t::('a,'b) typ-desc) = fold-td tnSum (map-td (λn x d. n) g t)
size-td-struct (st::('a,'b) typ-struct) = fold-td-struct (typ-name t) tnSum (map-td-struct (λn x d. n) g st)
size-td-list (ts::('a,'b) typ-tuple list) = fold-td-list (typ-name t) tnSum (map-td-list (λn x d. n) g ts)
size-td-tuple (x::('a,'b) typ-tuple) = fold-td-tuple tnSum (map-td-tuple (λn x d. n) g x)
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*auto simp*: *tnSum-def split*: *dt-tuple.splits*)

lemma *size-td-fm*:
size-td (t::('a,'b) typ-desc) ≡ fold-td tnSum (map-td (λn algn d. n) id t)
using *size-td-fm'(1)* [*of t id*] **by** *clarsimp*

lemma *align-td-wo-align-fm'*:
align-td-wo-align (*t*::('*a*,'*b*) *typ-desc*) = *fold-td tnMax* (*map-td* (λ*n x d*. *x*) *g t*)
align-td-wo-align-struct (*st*::('*a*,'*b*) *typ-struct*) = *fold-td-struct* (*typ-name t*) *tnMax* (*map-td-struct* (λ*n x d*. *x*) *g st*)
align-td-wo-align-list (*ts*::('*a*,'*b*) *typ-tuple list*) = *fold-td-list* (*typ-name t*) *tnMax* (*map-td-list* (λ*n x d*. *x*) *g ts*)
align-td-wo-align-tuple (*x*::('*a*,'*b*) *typ-tuple*) = *fold-td-tuple tnMax* (*map-td-tuple* (λ*n x d*. *x*) *g x*)
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*auto simp*: *tnMax-def split*: *dt-tuple.splits*)

lemma *align-td-wo-align-fm*:
align-td-wo-align (*t*::('*a*,'*b*) *typ-desc*) ≡ *fold-td tnMax* (*map-td* (λ*n algn d*. *algn*) *id t*)
using *align-td-wo-align-fm'*(*1*) [*of t id*] **by** *clarsimp*

thm *case-dt-tuple-def*

lemma *case-dt-tuple*:
snd ‘ *case-dt-tuple* (λ*t n d*. *Pair* (*f t*) *n*) ‘ *X* = *dt-snd* ‘ *X*
by (*force simp*: *image-iff split-DTuple-all split*: *dt-tuple.splits*)

lemma *map-DTuple-dt-snd*:
map-td-tuple f g x = *DTuple a b c* ⟹ *b* = *dt-snd x*
by (*metis dt-tuple.inject map-td'-map'*(*4*))

lemma *wf-desc-fm'*:
wf-desc (*t*::('*a*,'*b*) *typ-desc*) = *fold-td wfd* (*map-td* (λ*n x d*. *True*) *g t*)
wf-desc-struct (*st*::('*a*,'*b*) *typ-struct*) = *fold-td-struct* (*typ-name t*) *wfd* (*map-td-struct* (λ*n x d*. *True*) *g st*)
wf-desc-list (*ts*::('*a*,'*b*) *typ-tuple list*) = *fold-td-list* (*typ-name t*) *wfd* (*map-td-list* (λ*n x d*. *True*) *g ts*)
wf-desc-tuple (*x*::('*a*,'*b*) *typ-tuple*) = *fold-td-tuple wfd* (*map-td-tuple* (λ*n x d*. *True*) *g x*)
supply *split-DTuple-all*[*simp*] *dt-tuple.splits*[*split*]
apply (*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp simp*: *wfd-def image-comp*[*symmetric*]›)
apply (*rule iffI*; *clarsimp*)
apply (*metis* (*no-types*) *dt-prj-simps*(*2*) *dt-snd-map-td-list imageI*)
by (*metis* (*mono-tags*) *case-dt-tuple dt-prj-simps*(*2*) *dt-snd-map-td-list image-eqI*)

lemma *wf-desc-fm*:
wf-desc (*t*::('*a*,'*b*) *typ-desc*) ≡ *fold-td wfd* (*map-td* (λ*n algn d*. *True*) *id t*)
using *wf-desc-fm'*(*1*) [*of t id*] **by** *auto*

lemma *update-tag-list-empty* [*simp*]:
(*map-td-list f g xs* = []) = (*xs* = [])
by (*cases xs*, *auto*)

lemma *wf-size-desc-fm'*:
wf-size-desc (*t*::('*a*,'*b*) *typ-desc*) = *fold-td wfsd* (*map-td* (λ*n x d*. *0* < *n*) *g t*)

wf-size-desc-struct (*st*::('*a*,'*b*) *typ-struct*) = *fold-td-struct* (*typ-name t*) *wfsd* (*map-td-struct* (λ*n x d*. *0* < *n*) *g st*)
ts ≠ [] ⟶ *wf-size-desc-list* (*ts*::('*a*,'*b*) *typ-tuple list*) = *fold-td-list* (*typ-name t*) *wfsd* (*map-td-list* (λ*n x d*. *0* < *n*) *g ts*)
wf-size-desc-tuple (*x*::('*a*,'*b*) *typ-tuple*) = *fold-td-tuple wfsd* (*map-td-tuple* (λ*n x d*. *0* < *n*) *g x*)
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*auto simp*: *wfsd-def split*: *dt-tuple.splits*)

lemma *wf-size-desc-fm*:
wf-size-desc (*t*::('*a*,'*b*) *typ-desc*) ≡ *fold-td wfsd* (*map-td* (λ*n algn d*. *0* < *n*) *id t*)
using *wf-size-desc-fm'*(*1*) [*of t id*] **by** *auto*

typedef *stack-byte* = *UNIV*::*byte set*
by (*simp*)

definition *stack-byte-name* = ''*stack-byte*''

instantiation *stack-byte* :: *c-type*
begin

definition *typ-name-itself* (*x*::*stack-byte itself*) = *stack-byte-name*

definition
typ-info-stack-byte: *typ-info-t* (*x*::*stack-byte itself*) ≡
TypDesc 0 (*TypScalar 1 0*
(⦇*field-access* = λ*v bs*. [*Rep-stack-byte v*],
field-update = λ*bs v*. *if length bs* ≥ *1 then Abs-stack-byte* (*hd bs*) *else Abs-stack-byte* (*0*::*byte*),
field-sz = *1*⦈))
stack-byte-name

instance
by (*intro-classes*) (*simp add*: *typ-name-itself-stack-byte-def typ-info-stack-byte*)
end

lemma *size-of-stack-byte* [*simp*]:
size-of TYPE(*stack-byte*) = *1*
size-td (*typ-info-t TYPE*(*stack-byte*)) = *1*
size-td (*typ-uinfo-t TYPE*(*stack-byte*)) = *1*
by (*simp-all add*: *size-of-def typ-info-stack-byte typ-uinfo-t-def*)

lemma *align-of-stack-byte* [*simp*]:
align-of TYPE(*stack-byte*) = *1*
align-td (*typ-info-t TYPE*(*stack-byte*)) = *0*
align-td (*typ-uinfo-t TYPE*(*stack-byte*)) = *0*
by (*simp-all add*: *align-of-def typ-info-stack-byte typ-uinfo-t-def*)

lemma *length-1-conv*: *length bs* = *Suc 0* ⟷ (∃ *b*. *bs* = [*b*])
by (*cases bs*) *auto*

lemma *padding-lense-stack-byte*: *padding-lense* (λ*v bs*. [*Rep-stack-byte v*])
(λ*bs v*.
if Suc 0 ≤ *length bs then Abs-stack-byte* (*hd bs*) *else Abs-stack-byte 0*)
(*Suc 0*)
apply (*unfold-locales*)
apply (*simp add*: *padding-base.is-padding-byte-def padding-base.is-value-byte-def Abs-stack-byte-inverse*)
subgoal

proof −
have *length* [*1*::*8 word*] = *Suc 0*
by *simp*
then show *?thesis*
by (*metis* (*mono-tags*, *opaque-lifting*) *Abs-stack-byte-inject One-nat-def Rep-stack-byte-inject UNIV-I hd-conv-nth length-append n-not-Suc-n plus-1-eq-Suc self-append-conv2 zero-neq-one*)
qed
apply (*clarsimp*)
apply *clarsimp*
apply (*clarsimp simp add*: *Rep-stack-byte-inverse*)
apply *simp*
apply *simp*
apply *simp*
done

instantiation *stack-byte*:: *xmem-contained-type*
begin
instance
apply (*intro-classes*)
apply (*simp add*: *typ-info-stack-byte*)
apply (*simp add*: *typ-info-stack-byte*)
apply (*simp add*: *typ-info-stack-byte wf-lf-def fd-cons-desc-def fd-cons-double-update-def fd-cons-update-access-def fd-cons-access-update-def Rep-stack-byte-inverse fd-cons-length-def*)
apply (*simp add*: *typ-info-stack-byte*)
apply *simp*
apply (*simp add*: *typ-info-stack-byte align-field-def*)
apply (*simp add*: *typ-info-stack-byte*)
apply (*simp add*: *addr-card*)
apply (*simp add*: *typ-info-stack-byte*)
apply (*simp add*: *typ-info-stack-byte*)
apply (*simp add*: *typ-info-stack-byte wf-field-desc-def padding-lense-stack-byte*)
apply (*simp add*: *typ-info-stack-byte*)
apply (*simp add*: *typ-info-stack-byte*)
done

end

end

theory *Vanilla32-Preliminaries*
imports *CTypes*
begin

11.12 Words and Pointers

instantiation *unit* :: *c-type*
begin
definition [*simp*]: *typ-name-itself* (*x*::*unit itself*) = *''unit''*
definition *typ-info-unit*[*simp*]:
typ-info-t (*x*::*unit itself*) ≡
TypDesc 0 (*TypScalar 1 0*
(⦇*field-access* = (λ*v bs*. [*0*]), *field-update* = (λ*bs v*. ()), *field-sz* = *1*⦈))
''unit''
:: *unit xtyp-info*
instance by *intro-classes* (*simp add*: *typ-name-itself-unit-def*)
end

lemma *typ-name-unit*[*simp*]:
typ-name (*typ-info-t* (*TYPE*(*unit*))) = *''unit''*
typ-name (*typ-uinfo-t TYPE*(*unit*)) = *''unit''*
by (*simp-all add*: *typ-uinfo-t-def*)

instantiation *unit* :: *mem-type*
begin
definition
to-bytes-unit :: *unit* ⇒ *byte list* **where**
to-bytes-unit a ≡ [*0*]

definition
from-bytes-unit :: *byte list* ⇒ *unit* **where**
from-bytes-unit bs ≡ ()

definition
size-of-unit :: *unit itself* ⇒ *nat* **where**
size-of-unit x ≡ *0*

definition
align-of-unit :: *unit itself* ⇒ *nat* **where**
align-of-unit x ≡ *1*

instance
apply *intro-classes*
apply (*auto simp*: *size-of-def align-of-def align-field-def addr-card*
wf-lf-def fd-cons-desc-def

fd-cons-double-update-def fd-cons-update-access-def
fd-cons-access-update-def fd-cons-length-def)
done
end

definition
bogus-log2lessthree (*n*::*nat*) ==
if n = 128 then 4
else if n = 64 then (*3*::*nat*)
else if n = 32 then 2
else if n = 16 then 1
else if n = 8 then 0
else undefined
definition
len-exp (*x*::('*a*::*len*) *itself*) ≡ *bogus-log2lessthree* (*len-of TYPE*('*a*))

lemma *lx8'* [*simp*] : *len-exp* (*x*::*8 itself*) = *0*
by (*simp add*: *len-exp-def bogus-log2lessthree-def*)

lemma *lx16'* [*simp*]: *len-exp* (*x*::*16 itself*) = *1*
by (*simp add*: *len-exp-def bogus-log2lessthree-def*)

lemma *lx32'* [*simp*]: *len-exp* (*x*::*32 itself*) = *2*
by (*simp add*: *len-exp-def bogus-log2lessthree-def*)

lemma *lx64'* [*simp*]: *len-exp* (*x*::*64 itself*) = *3*
by (*simp add*: *len-exp-def bogus-log2lessthree-def*)

lemma *lx128'* [*simp*]: *len-exp* (*x*::*128 itself*) = *4*
by (*simp add*: *len-exp-def bogus-log2lessthree-def*)

lemma *lx-signed'* [*simp*]: *len-exp* (*x*::('*a*::*len*) *signed itself*) = *len-exp* (*TYPE*('*a*))
by (*simp add*: *len-exp-def*)

class *len8* = *len* +

assumes *len8-bytes*: *len-of TYPE*('*a*::*len*) = *8* ∗ (*2*^*len-exp TYPE*('*a*))

assumes *len8-width*: *len-of TYPE*('*a*::*len*) ≤ *128*
begin

lemma *len8-size*:
len-of TYPE('*a*) *div 8* < *addr-card*
apply(*subgoal-tac len-of TYPE*('*a*) ≤ *128*)
apply(*simp add*: *addr-card*)
apply(*rule len8-width*)
done

lemma *len8-dv8*:

8 dvd len-of TYPE('a)
by (*simp add*: *len8-bytes*)

lemma *len8-pow*:
∃k. len-of TYPE('a) div 8 = 2^k
by (*simp add*: *len8-bytes*)

end

fun
nat-to-bin-string :: *nat ⇒ char list*
where
ntbs: *nat-to-bin-string n = (if (n = 0) then ''0'' else (if n mod 2 = 1 then CHR ''1'' else CHR ''0'') # nat-to-bin-string (n div 2))*

declare *nat-to-bin-string.simps* [*simp del*]

lemma *nat-to-bin-string-simps*:
nat-to-bin-string 0 = ''0''
n > 0 ⟹ nat-to-bin-string n =
(if n mod 2 = 1 then CHR ''1'' else CHR ''0'') # nat-to-bin-string (n div 2)
apply (*induct n*, *auto simp*: *nat-to-bin-string.simps*)
done

instance *signed* :: (*len8*) *len8*
apply *intro-classes*
apply (*metis len8-bytes len-exp-def len-signed*)
apply (*metis len8-width len-signed*)
done

end

theory *Word-Mem-Encoding-ARM*
imports *../Vanilla32-Preliminaries*
begin

if-architecture-context (*ARM*)
begin

definition
word-tag :: *'a::len8 word itself ⇒ 'a word xtyp-info*
where
word-tag (w::'a::len8 word itself) ≡
TypDesc (len-exp TYPE('a)) (TypScalar (len-of TYPE('a) div 8) (len-exp TYPE('a))
(⦇field-access = λv bs. (rev o word-rsplit) v,
field-update = λbs v. (word-rcat (rev (take (len-of TYPE('a) div 8) bs))::'a::len8 word),

```
            field-sz = (len-of TYPE('a) div 8)⦈)
          (''word'' @  nat-to-bin-string (len-of TYPE('a)))
end

end



theory Word-Mem-Encoding-ARM64
  imports Vanilla32-Preliminaries
begin

if-architecture-context (ARM64)
begin

definition
  word-tag :: 'a::len8 word itself ⇒ 'a word xtyp-info
where
  word-tag (w::'a::len8 word itself) ≡
    TypDesc (len-exp TYPE('a)) (TypScalar (len-of TYPE('a) div 8)
                    (len-exp TYPE('a))
                    (⦇field-access = λv bs. (rev o word-rsplit) v,
                      field-update = λbs v. (word-rcat (rev (take (len-of TYPE('a)
div 8) bs))::'a::len8 word),
                     field-sz = (len-of TYPE('a) div 8)⦈)
          (''word'' @  nat-to-bin-string (len-of TYPE('a)))
end

end


theory Word-Mem-Encoding-ARM-HYP
  imports ../Vanilla32-Preliminaries
begin

if-architecture-context (ARM-HYP)
begin

definition
  word-tag :: 'a::len8 word itself ⇒ 'a word xtyp-info
where
  word-tag (w::'a::len8 word itself) ≡
     TypDesc (len-exp TYPE('a)) (TypScalar (len-of TYPE('a) div 8) (len-exp
TYPE('a))
            (⦇field-access = λv bs. (rev o word-rsplit) v,
              field-update = λbs v. (word-rcat (rev (take (len-of TYPE('a) div 8)
bs))::'a::len8 word),
             field-sz = (len-of TYPE('a) div 8)⦈)
    (''word'' @  nat-to-bin-string (len-of TYPE('a)) )
```

end

end

theory *Word-Mem-Encoding-RISCV64*
imports *../Vanilla32-Preliminaries*
begin

if-architecture-context (*RISCV64*)
begin

definition
word-tag :: *'a*::*len8 word itself* ⇒ *'a word xtyp-info*
where
word-tag (*w*::*'a*::*len8 word itself*) ≡
TypDesc (*len-exp TYPE*(*'a*)) (*TypScalar* (*len-of TYPE*(*'a*) *div* 8)
(*len-exp TYPE*(*'a*))
(⦇*field-access* = λ*v bs*. (*rev o word-rsplit*) *v*,
field-update = λ*bs v*. (*word-rcat* (*rev* (*take* (*len-of TYPE*(*'a*)
div 8) *bs*))::*'a*::*len8 word*),
field-sz = (*len-of TYPE*(*'a*) *div* 8)⦈))
(*''word''* @ *nat-to-bin-string* (*len-of TYPE*(*'a*)))
end

end

theory *Word-Mem-Encoding-X64*
imports *../Vanilla32-Preliminaries*
begin

if-architecture-context (*X64*)
begin

definition
word-tag :: *'a*::*len8 word itself* ⇒ *'a word xtyp-info*
where
word-tag (*w*::*'a*::*len8 word itself*) ≡
TypDesc (*len-exp TYPE*(*'a*)) (*TypScalar* (*len-of TYPE*(*'a*) *div* 8)
(*len-exp TYPE*(*'a*))
(⦇*field-access* = λ*v bs*. (*rev o word-rsplit*) *v*,
field-update = λ*bs v*. (*word-rcat* (*rev* (*take* (*len-of TYPE*(*'a*)
div 8) *bs*))::*'a*::*len8 word*),
field-sz = (*len-of TYPE*(*'a*) *div* 8)⦈))
(*''word''* @ *nat-to-bin-string* (*len-of TYPE*(*'a*)))
end

end

theory *Word-Mem-Encoding*
imports
ARM / Word-Mem-Encoding-ARM
ARM64 / Word-Mem-Encoding-ARM64
ARM-HYP / Word-Mem-Encoding-ARM-HYP
RISCV64 / Word-Mem-Encoding-RISCV64
X64 / Word-Mem-Encoding-X64
begin

end

theory *Vanilla32*
imports *Word-Mem-Encoding CTypes*
begin

type-synonym *exit-status* = *32 sword*

instantiation *word* :: (*len8*) *c-type*
begin
definition *typ-name-itself* (*w*::*'a*::*len8 word itself*) =
''word'' @ *nat-to-bin-string* (*len-of TYPE*(*'a*))
definition
typ-info-word: *typ-info-t* (*w*::*'a*::*len8 word itself*) ≡ *word-tag w*
instance by (*intro-classes*)
(*simp add*: *typ-name-itself-word-def typ-info-word word-tag-def*)
end

lemma *align-of-word*:
align-of TYPE(*'a*::*len8 word*) = *len-of TYPE*(*'a*) *div 8*
by (*simp add*: *align-of-def typ-info-word word-tag-def len8-bytes*)

instantiation *word* :: (*len8*) *mem-type*
begin

instance
proof *intro-classes*
show ⋀*bs v w*. *length bs* = *size-of TYPE*(*'a word*) ⟶
update-ti-t (*typ-info-t TYPE*(*'a word*)) *bs v* =
update-ti-t (*typ-info-t TYPE*(*'a word*)) *bs w*
by (*simp add*: *typ-info-word word-tag-def size-of-def update-ti-t-def*)
next
show *wf-desc* (*typ-info-t TYPE*(*'a word*))
by (*simp add*: *typ-info-word word-tag-def*)

next
have *8 dvd len-of TYPE('a)* **by** (*rule len8-dv8*)
moreover have *0 < len-of TYPE('a)* **by** *simp*
ultimately have *0 < len-of TYPE('a) div 8* **by** − (*erule dvdE*, *simp*)
thus *wf-size-desc* (*typ-info-t TYPE('a word)*)
by (*simp add*: *typ-info-word word-tag-def len8-pow*)
next
have *8 dvd len-of TYPE('a)* **by** (*rule len8-dv8*)
thus *wf-lf* (*lf-set* (*typ-info-t TYPE('a word)*) [])
by (*auto simp*: *typ-info-word word-tag-def wf-lf-def*
fd-cons-def fd-cons-double-update-def fd-cons-update-access-def
word-rcat-rsplit fd-cons-access-update-def fd-cons-length-def
length-word-rsplit-exp-size' word-size fd-cons-update-normalise-def
fd-cons-desc-def field-norm-def)
next
show *size-of TYPE('a word) < addr-card*
by (*simp add*: *size-of-def typ-info-word word-tag-def*)
(*rule len8-size*)
next
show *align-of TYPE('a word) dvd size-of TYPE('a word)*
by (*clarsimp simp*: *size-of-def align-of-word typ-info-word*
word-tag-def)
next
show *align-field* (*typ-info-t TYPE('a word)*)
by (*clarsimp simp*: *typ-info-word word-tag-def align-field-def*)
next
show *wf-align* (*typ-info-t TYPE('a word)*)
by (*clarsimp simp*: *typ-info-word word-tag-def*)
qed

end

instantiation *word* :: (*len8*) *simple-mem-type*
begin

instance
apply *intro-classes*
apply(*clarsimp simp*: *typ-info-word word-tag-def typ-uinfo-t-def*)
done

end

class *len-eq1* = *len* +
assumes *len-eq1*: *len-of TYPE('a::len) = 1*
instance *num1* :: *len-eq1*
by (*intro-classes*, *simp*)

class *len-lg01* = *len* +

assumes *len-lg01*: *len-of TYPE*('*a*::*len*) ∈ {*1*,*2*}
instance *bit0* :: (*len-eq1*) *len-lg01*
by (*intro-classes*, *simp add*: *len-eq1*)
instance *num1* :: *len-lg01*
by (*intro-classes*, *simp*)

class *len-lg02* = *len* +
assumes *len-lg02*: *len-of TYPE*('*a*::*len*) ∈ {*1*,*2*,*4*}
instance *bit0* :: (*len-lg01*) *len-lg02*
apply *intro-classes*
apply (*insert len-lg01*[**where** '*a* = '*a*])
apply *simp*
done
instance *num1* :: *len-lg02*
by (*intro-classes*, *simp*)

class *len-lg03* = *len* +
assumes *len-lg03*: *len-of TYPE*('*a*::*len*) ∈ {*1*,*2*,*4*,*8*}
instance *bit0* :: (*len-lg02*) *len-lg03*
apply *intro-classes*
apply (*insert len-lg02*[**where** '*a* = '*a*])
apply *simp*
done
instance *num1* :: *len-lg03*
by (*intro-classes*, *simp*)

class *len-lg04* = *len* +
assumes *len-lg04*: *len-of TYPE*('*a*::*len*) ∈ {*1*,*2*,*4*,*8*,*16*}
instance *bit0* :: (*len-lg03*) *len-lg04*
apply *intro-classes*
apply (*insert len-lg03*[**where** '*a* = '*a*])
apply *simp*
done

instance *num1* :: *len-lg04*
by (*intro-classes*, *simp*)
class *len-lg15* = *len* +
assumes *len-lg15*: *len-of TYPE*('*a*::*len*) ∈ {*2*,*4*,*8*,*16*,*32*}
instance *bit0* :: (*len-lg04*) *len-lg15*
apply *intro-classes*
apply (*insert len-lg04*[**where** '*a* = '*a*])
apply *simp*
done

class *len-lg26* = *len* +
assumes *len-lg26*: *len-of TYPE*('*a*::*len*) ∈ {*4*,*8*,*16*,*32*,*64*}

instance *bit0* :: (*len-lg15*) *len-lg26*
apply *intro-classes*

apply (*insert len-lg15*[**where** *'a* = *'a*])
apply *simp*
done

instance *bit0* :: (*len-lg26*) *len8*
apply *intro-classes*
apply *simp*
apply (*insert len-lg26*[**where** *'a* = *'a*])
apply (*simp add*: *len-exp-def*)
apply (*erule disjE*)
apply (*simp add*: *bogus-log2lessthree-def len8-bytes len8-width*)
apply (*erule disjE*)
apply (*simp add*: *bogus-log2lessthree-def*)
apply (*erule disjE*)
apply (*simp add*: *bogus-log2lessthree-def*)
apply (*erule disjE*)
apply (*simp add*: *bogus-log2lessthree-def*)
apply (*simp add*: *bogus-log2lessthree-def*)
apply *simp*
apply *auto*
done

instance *signed* :: (*len-eq1*) *len-eq1* **using** *len-eq1* **by** (*intro-classes*, *simp*)
instance *signed* :: (*len-lg01*) *len-lg01* **using** *len-lg01* **by** (*intro-classes*, *simp*)
instance *signed* :: (*len-lg02*) *len-lg02* **using** *len-lg02* **by** (*intro-classes*, *simp*)
instance *signed* :: (*len-lg03*) *len-lg03* **using** *len-lg03* **by** (*intro-classes*, *simp*)
instance *signed* :: (*len-lg04*) *len-lg04* **using** *len-lg04* **by** (*intro-classes*, *simp*)
instance *signed* :: (*len-lg15*) *len-lg15* **using** *len-lg15* **by** (*intro-classes*, *simp*)
instance *signed* :: (*len-lg26*) *len-lg26* **using** *len-lg26* **by** (*intro-classes*, *simp*)

lemma
to-bytes (*1*256*256*256* + *2*256*256* + *3*256* + (*4*::*32 word*)) *bs* = [*4*, *3*, *2*, *1*]
by (*simp add*: *to-bytes-def typ-info-word word-tag-def Let-def*)

lemma *size-td-words* [*simp*]:
size-td (*typ-info-t TYPE*(*8 word*)) = *1*
size-td (*typ-info-t TYPE*(*16 word*)) = *2*
size-td (*typ-info-t TYPE*(*32 word*)) = *4*
size-td (*typ-info-t TYPE*(*64 word*)) = *8*
size-td (*typ-info-t TYPE*(*128 word*)) = *16*
by (*auto simp*: *typ-info-word word-tag-def*)

lemma *align-td-words* [*simp*]:
align-td (*typ-info-t TYPE*(*8 word*)) = *0*
align-td (*typ-info-t TYPE*(*16 word*)) = *1*
align-td (*typ-info-t TYPE*(*32 word*)) = *2*
align-td (*typ-info-t TYPE*(*64 word*)) = *3*
align-td (*typ-info-t TYPE*(*128 word*)) = *4*

by (*auto simp*: *typ-info-word word-tag-def len8-bytes*)

lemma *size-of-words* [*simp*]:
size-of TYPE(*8 word*) = *1*
size-of TYPE(*16 word*) = *2*
size-of TYPE(*32 word*) = *4*
size-of TYPE(*64 word*) = *8*
size-of TYPE(*128 word*) = *16*
by (*auto simp*: *size-of-def*)

lemma *align-of-words* [*simp*]:
align-of TYPE(*8 word*) = *1*
align-of TYPE(*16 word*) = *2*
align-of TYPE(*32 word*) = *4*
align-of TYPE(*64 word*) = *8*
align-of TYPE(*128 word*) = *16*
by (*auto simp*: *align-of-word*)

lemma *size-td-swords* [*simp*]:
size-td (*typ-info-t TYPE*(*8 sword*)) = *1*
size-td (*typ-info-t TYPE*(*16 sword*)) = *2*
size-td (*typ-info-t TYPE*(*32 sword*)) = *4*
size-td (*typ-info-t TYPE*(*64 sword*)) = *8*
size-td (*typ-info-t TYPE*(*128 sword*)) = *16*
by (*auto simp*: *typ-info-word word-tag-def*)

lemma *align-td-swords* [*simp*]:
align-td (*typ-info-t TYPE*(*8 sword*)) = *0*
align-td (*typ-info-t TYPE*(*16 sword*)) = *1*
align-td (*typ-info-t TYPE*(*32 sword*)) = *2*
align-td (*typ-info-t TYPE*(*64 sword*)) = *3*
align-td (*typ-info-t TYPE*(*128 sword*)) = *4*
by (*auto simp*: *typ-info-word word-tag-def len8-bytes*)

lemma *size-of-swords* [*simp*]:
size-of TYPE(*8 sword*) = *1*
size-of TYPE(*16 sword*) = *2*
size-of TYPE(*32 sword*) = *4*
size-of TYPE(*64 sword*) = *8*
size-of TYPE(*128 sword*) = *16*
by (*auto simp*: *size-of-def*)

lemma *align-of-swords* [*simp*]:
align-of TYPE(*8 sword*) = *1*
align-of TYPE(*16 sword*) = *2*
align-of TYPE(*32 sword*) = *4*
align-of TYPE(*64 sword*) = *8*
align-of TYPE(*128 sword*) = *16*
by (*auto simp*: *align-of-word*)

```
instantiation ptr :: (c-type-name) c-type
begin
definition typ-name-itself (p::'a::c-type-name ptr itself) = typ-name-itself TYPE('a)
@ ''+ptr''
definition
  typ-info-ptr:
  typ-info-t (p::'a::c-type-name ptr itself) ≡ TypDesc addr-align
    (TypScalar (addr-bitsize div 8) addr-align (|
        field-access = λp bs. rev (word-rsplit (ptr-val p)),
      field-update = λbs v. Ptr (word-rcat (rev (take (addr-bitsize div 8) bs))::addr),
        field-sz = (addr-bitsize div 8) |) )
    (typ-name-itself TYPE('a) @ ''+ptr'')

instance by (intro-classes) (simp add: typ-name-itself-ptr-def typ-info-ptr)
end

lemma align-of-ptr [simp]:
  align-of (p::'a::c-type ptr itself) = 2 ^ addr-align
  by (simp add: align-of-def typ-info-ptr)

instantiation ptr :: (c-type) mem-type
begin
instance
  apply (intro-classes)
    apply (auto simp: to-bytes-def from-bytes-def
                length-word-rsplit-exp-size' word-size
                word-rcat-rsplit size-of-def  addr-card
                typ-info-ptr fd-cons-double-update-def
                fd-cons-update-access-def fd-cons-access-update-def
                fd-cons-length-def fd-cons-update-normalise-def field-norm-def
                super-update-bs-def word-rsplit-rcat-size norm-bytes-def
                fd-consistent-def fd-cons-def wf-lf-def
                fd-cons-desc-def align-field-def update-ti-t-def)
  done
end

lemma div-eq: 8 dvd n ⟹ (7 + n::nat) div 8 = n div 8
  by (simp add: div-plus-div-distrib-dvd-right)

lemma length-word-rsplit-len8: length ((word-rsplit::('a::len8 word) ⇒ 8 word list)
w) = (LENGTH('a) div (8::nat))
proof −
  have 8 dvd len-of TYPE('a) by (rule len8-dv8)
  then show ?thesis
    by (simp add: length-word-rsplit-exp-size  div-eq word-size)
qed
```

lemma *all-value-byte-word*: $i < LENGTH('a::len8)\ div\ 8 \Longrightarrow$
padding-base.is-value-byte ($\lambda v\ bs.\ rev\ (word\text{-}rsplit\ v)$)
($\lambda bs\ v.\ word\text{-}rcat\ (rev\ (take\ (LENGTH('a)\ div\ 8)\ bs))::'a\ word$)
$(LENGTH('a)\ div\ 8)\ i$
by (*clarsimp simp add*: *padding-base.is-value-byte-def len8-bytes word-size word-rsplit-rcat-size*)

lemma *no-padding-byte-word*: $i < LENGTH('a::len8)\ div\ 8 \Longrightarrow$
$\neg$ *padding-base.is-padding-byte* ($\lambda v\ bs.\ rev\ (word\text{-}rsplit\ v)$)
($\lambda bs\ v.\ word\text{-}rcat\ (rev\ (take\ (LENGTH('a)\ div\ 8)\ bs))::'a\ word)\ (LENGTH('a)\ div\ 8)\ i$
apply (*clarsimp simp add*: *padding-base.is-padding-byte-def len8-bytes word-size*)
by (*metis* (*mono-tags*, *opaque-lifting*) *Ex-list-of-length len8 len-gt-0 length-list-update less-irrefl-nat nth-list-update unat-mono word-power-less-1*)

instantiation *word* :: (*len8*) *xmem-contained-type*
begin
instance
apply *intro-classes*
apply (*clarsimp simp add*: *typ-info-word word-tag-def*)
apply (*clarsimp simp add*: *typ-info-word word-tag-def*)
apply (*clarsimp simp add*: *typ-info-word word-tag-def*)
apply (*unfold-locales*; *clarsimp simp add*: *padding-base.eq-padding-def padding-base.eq-upto-padding-def length-word-rsplit-exp-size word-size word-rcat-rsplit div-eq length-word-rsplit-len8*)
apply (*simp add*: *all-value-byte-word no-padding-byte-word*)
apply (*simp add*: *wf-padding typ-info-word word-tag-def*)
apply (*clarsimp simp add*: *typ-info-word word-tag-def*)
done
end

instantiation *ptr* :: (*c-type*) *simple-mem-type*
begin
instance
apply *intro-classes*
apply(*clarsimp simp*: *typ-info-ptr typ-uinfo-t-def*)
done
end

lemma *all-value-byte-ptr*: $\bigwedge i.\ i < (addr\text{-}bitsize\ div\ 8) \Longrightarrow$
padding-base.is-value-byte ($\lambda p\ bs.\ rev\ (word\text{-}rsplit\ (ptr\text{-}val\ p))$)
($\lambda bs\ v.\ PTR('a)\ (word\text{-}rcat\ (rev\ (take\ (addr\text{-}bitsize\ div\ 8)\ bs))))$
$(addr\text{-}bitsize\ div\ 8)\ i$
by (*clarsimp simp add*: *padding-base.is-value-byte-def len8-bytes word-size word-rsplit-rcat-size*)

lemma *no-padding-byte-ptr*: $\bigwedge i.\ i < (addr\text{-}bitsize\ div\ 8) \Longrightarrow$

```
        ¬ padding-base.is-padding-byte (λp bs. rev (word-rsplit (ptr-val p)))
           (λbs v. PTR('a::c-type) (word-rcat (rev (take (addr-bitsize div 8) bs))))
(addr-bitsize div 8) i
  apply (clarsimp simp add: padding-base.is-padding-byte-def len8-bytes word-size
)
  by (metis (mono-tags, opaque-lifting) Ex-list-of-length len8 len-gt-0 length-list-update
less-irrefl-nat
      nth-list-update unat-mono word-power-less-1)

instantiation ptr :: (c-type) xmem-contained-type
begin
instance
  apply intro-classes
      apply(clarsimp simp: typ-info-ptr typ-uinfo-t-def)
     apply(clarsimp simp: typ-info-ptr typ-uinfo-t-def)
    apply(clarsimp simp: typ-info-ptr typ-uinfo-t-def)
   apply (unfold-locales; clarsimp simp add: padding-base.eq-padding-def padding-base.eq-upto-padding-def
    length-word-rsplit-exp-size word-size word-rcat-rsplit div-eq length-word-rsplit-len8)
    apply (simp add: no-padding-byte-ptr [simplified] all-value-byte-ptr [simplified])
   apply(clarsimp simp: typ-info-ptr typ-uinfo-t-def)
  apply (clarsimp simp add: typ-info-ptr typ-uinfo-t-def)
  done
end

lemma size-td-ptr [simp]:
  size-td (typ-info-t TYPE('a::c-type ptr)) = addr-bitsize div 8
  by (simp add: typ-info-ptr)

lemma size-of-ptr [simp]:
  size-of TYPE('a::c-type ptr) = addr-bitsize div 8
  by (simp add: size-of-def)

lemma align-td-ptr [simp]: align-td (typ-info-t TYPE('a::c-type ptr)) = addr-align
  by (simp add: typ-info-ptr)

lemma ptr-add-word32-signed [simp]:
  fixes a :: 32 word ptr
  shows ptr-val (a +p x) = ptr-val a + 4 * of-int x
  by (cases a) (simp add: CTypesDefs.ptr-add-def scast-id)

lemma ptr-add-word32 [simp]:
  fixes a :: 32 word ptr
  shows ptr-val (a +p uint x) = ptr-val a + 4 * x
  by (cases a) (simp add: ptr-add-def scast-id)

lemma ptr-add-word64-signed [simp]:
  fixes a :: 64 word ptr
  shows ptr-val (a +p x) = ptr-val a + 8 * of-int x
  by (cases a) (simp add: CTypesDefs.ptr-add-def scast-id)
```

lemma *ptr-add-word64* [*simp*]:
fixes *a* :: *64 word ptr*
shows *ptr-val* (*a* +$_p$ *uint x*) = *ptr-val a* + *8* ∗ *x*
by (*cases a*) (*simp add*: *ptr-add-def scast-id*)

lemma *ptr-add-0-id*[*simp*]:*x* +$_p$ *0* = *x*
by (*simp add*:*CTypesDefs.ptr-add-def*)

lemma *from-bytes-ptr-to-bytes-ptr*:
from-bytes (*to-bytes* (*v*::*addr-bitsize word*) *bs*) = (*Ptr v* :: *'a*::*c-type ptr*)
by (*simp add*: *from-bytes-def to-bytes-def typ-info-ptr word-tag-def*
typ-info-word word-size
length-word-rsplit-exp-size' word-rcat-rsplit update-ti-t-def)

lemma *ptr-aligned-coerceI*:
ptr-aligned (*ptr-coerce x*::*addr ptr*) ⟹ *ptr-aligned* (*x*::*'a*::*mem-type ptr ptr*)
by (*simp add*: *ptr-aligned-def*)

lemma *lift-ptr-ptr* [*simp*]:
⋀*p*::*'a*::*mem-type ptr ptr*.
lift h (*ptr-coerce p*) = *ptr-val* (*lift h p*)
by (*simp add*: *lift-def h-val-def from-bytes-def*
typ-info-word word-tag-def typ-info-ptr update-ti-t-def)
thm *typ-name.simps*

lemma *typ-name-itself-ptr*:
typ-name-itself (*T*::*'a*::*c-type ptr itself*) = *typ-name-itself TYPE*(*'a*) @ *''+ptr''*
by (*simp add*: *typ-info-ptr typ-name-itself-ptr-def*)

lemma *typ-name-itself-word*:
typ-name-itself (*T*::*'a*::*len8 word itself*) = *typ-name* (*typ-info-t T*)
by (*simp add*: *typ-info-word word-tag-def typ-name-itself-word-def*)

lemmas *Vanilla32-tags* [*simp*] =
typ-info-word
typ-info-ptr
typ-name-itself-ptr
word-tag-def
typ-name-itself-word

lemma *ptr-typ-name* [*simp*]:
typ-name (*typ-info-t TYPE*((*'a* :: *c-type*) *ptr*)) = *typ-name-itself TYPE*(*'a*) @
''+ptr''
by *simp*

lemma *word-typ-name* [*simp*]:
typ-name (*typ-info-t TYPE*(*'a*::*len8 word*)) = *''word''* @ *nat-to-bin-string* (*len-of TYPE*(*'a*))

by *simp*

lemma *typ-name-words* [*simp*]:
typ-name (*typ-uinfo-t TYPE*(*8 word*)) = ''*word00010*''
typ-name (*typ-uinfo-t TYPE*(*16 word*)) = ''*word000010*''
typ-name (*typ-uinfo-t TYPE*(*32 word*)) = ''*word0000010*''
typ-name (*typ-uinfo-t TYPE*(*64 word*)) = ''*word00000010*''
typ-name (*typ-uinfo-t TYPE*(*128 word*)) = ''*word000000010*''
typ-name (*typ-info-t TYPE*(*8 word*)) = ''*word00010*''
typ-name (*typ-info-t TYPE*(*16 word*)) = ''*word000010*''
typ-name (*typ-info-t TYPE*(*32 word*)) = ''*word0000010*''
typ-name (*typ-info-t TYPE*(*64 word*)) = ''*word00000010*''
typ-name (*typ-info-t TYPE*(*128 word*)) = ''*word000000010*''
by (*auto simp*: *typ-uinfo-t-def nat-to-bin-string.simps export-uinfo-def*)

lemma *typ-name-swords* [*simp*]:
typ-name (*typ-uinfo-t TYPE*(*8 sword*)) = ''*word00010*''
typ-name (*typ-uinfo-t TYPE*(*16 sword*)) = ''*word000010*''
typ-name (*typ-uinfo-t TYPE*(*32 sword*)) = ''*word0000010*''
typ-name (*typ-uinfo-t TYPE*(*64 sword*)) = ''*word00000010*''
typ-name (*typ-uinfo-t TYPE*(*128 sword*)) = ''*word000000010*''
typ-name (*typ-info-t TYPE*(*8 sword*)) = ''*word00010*''
typ-name (*typ-info-t TYPE*(*16 sword*)) = ''*word000010*''
typ-name (*typ-info-t TYPE*(*32 sword*)) = ''*word0000010*''
typ-name (*typ-info-t TYPE*(*64 sword*)) = ''*word00000010*''
typ-name (*typ-info-t TYPE*(*128 sword*)) = ''*word000000010*''
by (*auto simp*: *typ-uinfo-t-def nat-to-bin-string.simps export-uinfo-def*)

lemma *ptr-arith*[*simp*]:
(x +$_p$ a = y +$_p$ a) = ((x::('a::*c-type*) *ptr*) = (y::'a *ptr*))
by (*clarsimp simp*:*CTypesDefs.ptr-add-def*)

lemma *ptr-arith'*[*simp*]:
(*ptr-coerce* (x +$_p$ a) = *ptr-coerce* (y +$_p$ a)) = ((x::('a::*c-type*) *ptr*) = (y::'a *ptr*))
by (*clarsimp simp*:*CTypesDefs.ptr-add-def*)

lemma *typ-uinfo-t-signed-word-word-conv*: *typ-uinfo-t TYPE*(('a::*len8*) *signed word*)
= *typ-uinfo-t TYPE*('a *word*)
by (*simp add*: *typ-uinfo-t-def field-norm-def*
len8-bytes word-rsplit-rcat-size word-size fun-eq-iff)

end

theory *ML-Infer-Instantiate*
imports *Main*
begin

11.13 Instantiation, inferring type instantiation from term instantiation.

ML-val
‹
instantiate ‹*'a*=***typ*** ‹*nat*› *and* *x*=***term*** ‹*1*::*nat*› *in term* ‹*x + x*› *for* *x*::*'a*::*plus*›
›

This is a variant of of the instantiate antiquotation, e.g.: `\<^instantiate>\<open>'a=\<^typ>\<c`
`and x=\<^term>\<open>1::nat\<close> in term \<open>x + x\<close>`
`for x::\<open>'a::plus\<close>\<close>`.

The instantiate antiquotation requires the explicit instantiation of type-variables, including those that can be inferred from the term instantiation. This makes the code very explicit, predictable and efficient as no types have to be calculated at runtime or be 'guessed' by the reader. On the other hand it can make the resulting code quite verbose when the number of type variables increases.

In this theory we provide a variant, that allows to omit those type-instantiations that are already fixed by the given term instantiations. This follows the same idea as e.g. `Drule.infer_instantiate`.

Moreover, we provide a variant that also takes a morphism into account. The use-case here is that the instantiate expression is written within a locale context but later on is used in an interpretation of the locale (given by morphism phi). Before instantiation the term is exported (according to morphism phi).

ML ‹
```
fun pretty-terms ctxt ts =
  Pretty.list [ ] (map (Syntax.pretty-term ctxt) ts)
fun string-of-terms ctxt ts = pretty-terms ctxt ts |> Pretty.string-of
fun pretty-thms ctxt thms =
  Pretty.list [ ] (map (Thm.pretty-thm ctxt) thms)
fun pretty-big-list-thms name ctxt thms =
  Pretty.big-list name  (map (Thm.pretty-thm-item ctxt) thms)

fun string-of-thms ctxt thms = pretty-thms ctxt thms |> Pretty.string-of
```
›

ML ‹
```
signature ML-INFER-INSTANTIATE =
sig
  val make-cterm: Proof.context -> term -> cterm
  type insts = ((indexname * sort) * typ) list * ((indexname * typ) * term) list
  type cinsts = ((indexname * sort) * ctyp) list * ((indexname * typ) * cterm) list
  val morph-instantiate-term: Position.T -> insts -> term -> Morphism.morphism
-> Proof.context -> term
  val morph-instantiate-cterm: Position.T -> cinsts -> cterm -> Morphism.morphism
-> Proof.context -> cterm
```

```
  val morph-instantiate-thm: Position.T -> cinsts -> thm -> Morphism.morphism
-> Proof.context -> thm
  val morph-instantiate-thms: Position.T -> cinsts -> thm list -> Morphism.morphism
-> Proof.context -> thm list
  val instantiate-term: Position.T -> insts -> term -> Proof.context -> term
  val instantiate-cterm: Position.T -> cinsts -> cterm -> Proof.context ->
cterm
  val instantiate-thm: Position.T -> cinsts -> thm -> Proof.context -> thm
  val instantiate-thms: Position.T -> cinsts -> thm list -> Proof.context ->
thm list
  val get-thms: Proof.context -> int -> thm list
  val get-thm: Proof.context -> int -> thm
end;

structure ML-Infer-Instantiate:ML-INFER-INSTANTIATE =
struct

(* exported operations *)

fun make-cterm ctxt t = Thm.cterm-of ctxt t |> Thm.trim-context-cterm;

type insts = ((indexname * sort) * typ) list * ((indexname * typ) * term) list
type cinsts = ((indexname * sort) * ctyp) list * ((indexname * typ) * cterm) list

type 'a operations = {
  type-of: 'a -> typ,
  transfer: theory -> 'a -> 'a,
  maxidx-of: 'a -> int,
  term-of: 'a -> term}

val term-operations = {
  type-of  = fastype-of,
  transfer  = K I,
  maxidx-of = Term.maxidx-of-term,
  term-of  = I}: term operations

val cterm-operations = {
  type-of  = Thm.typ-of-cterm,
  transfer  = Thm.transfer-cterm,
  maxidx-of = Thm.maxidx-of-cterm,
  term-of  = Thm.term-of}: cterm operations

— cf. Drule.infer_instantiate_types
fun gen-infer (operations:'a operations) ctxt ((xi, T), cu) (tyenv, maxidx) =
  let
    val thy = Proof-Context.theory-of ctxt
    val - = Thm.ctyp-of ctxt T;
    val - = #transfer operations thy cu;
    val U = #type-of operations cu;
```

```
    val maxidx' = maxidx
      |> Integer.max (#2 xi)
      |> Term.maxidx-typ T
      |> Integer.max (#maxidx-of operations cu);
    val (tyenv', maxidx'') = Sign.typ-unify thy (T, U) (tyenv, maxidx')
      handle Type.TUNIFY =>
        let
          val t = Var (xi, T);
          val u = #term-of operations cu;
        in
          raise TERM (ML-Infer-Instantiate.gen-infer: type ^
            Syntax.string-of-typ ctxt (Envir.norm-type tyenv T) ^ of variable ^
           Syntax.string-of-term ctxt (Term.map-types (Envir.norm-type tyenv) t) ^
            \ncannot be unified with type ^
            Syntax.string-of-typ ctxt (Envir.norm-type tyenv U) ^ of term ^
            Syntax.string-of-term ctxt (Term.map-types (Envir.norm-type tyenv) u),
            [t, u])
        end;
  in (tyenv', maxidx'') end;

val infer = gen-infer term-operations
val cterm-infer = gen-infer cterm-operations

fun incr-instT k (((n, i), S), T) = (((n, i + k), S), T)
fun incr-inst k (((n, i), T), t) = (((n, i + k), Logic.incr-tvar k T), t)
val maxidx-ctyp = Term.maxidx-typ o Thm.typ-of
val maxidx-cterm = Integer.max o Thm.maxidx-of-cterm

fun mk-term ctxt P =
  if Thm.typ-of-cterm P = propT then
    P
  else
   instantiate ‹'a=‹Thm.ctyp-of-cterm P› and P = ‹Thm.transfer-cterm (Proof-Context.theory-of
ctxt) P›
     in cterm ‹TERM P› for P::'a›

fun dest-term ct =
  if can Logic.dest-term (Thm.term-of ct) then
    Thm.dest-arg ct
  else
    ct


(* Simultaneous export on instantions and term *)
type 'a morph-operations = {
  mk-typ: Proof.context -> (indexname * sort) -> 'a,
  dest-typ: 'a -> (indexname * sort),
  mk-term: Proof.context -> (indexname * typ) -> 'a,
  dest-term: 'a -> (indexname * typ),
```

```
  wrap: Proof.context -> 'a -> 'a,
  unwrap: 'a -> 'a,
  mk-conjunctions: 'a list -> 'a,
  dest-conjunctions: int -> 'a -> 'a list,
  morph: Morphism.morphism -> 'a -> 'a
}

val term-morph-operations: term morph-operations = {
  mk-typ = fn - => Logic.mk-term o Logic.mk-type o TVar,
  dest-typ = dest-TVar o Logic.dest-type o Logic.dest-term,
  mk-term = fn - => Logic.mk-term o Var,
  dest-term = dest-Var o Logic.dest-term,
  wrap = K I,
  unwrap = I,
  mk-conjunctions =  Logic.mk-conjunction-list,
  dest-conjunctions = fn - =>  Logic.dest-conjunction-list,
  morph = Morphism.term
}

val cterm-morph-operations: cterm morph-operations = {
  mk-typ = fn ctxt => Thm.cterm-of ctxt o Logic.mk-term o Logic.mk-type o TVar,
  dest-typ = dest-TVar o Logic.dest-type o Logic.dest-term o Thm.term-of,
  mk-term = fn ctxt => Thm.cterm-of ctxt o Logic.mk-term o Var,
  dest-term = dest-Var o Logic.dest-term o Thm.term-of,
  wrap = mk-term,
  unwrap =  dest-term,
  mk-conjunctions = Conjunction.mk-conjunction-balanced,
  dest-conjunctions = fn - => Conjunction.dest-conjunctions,
  morph = Morphism.cterm
}

val thm-morph-operations: thm morph-operations = {
   mk-typ = fn ctxt => Thm.reflexive o Thm.cterm-of ctxt o Logic.mk-term o
Logic.mk-type o TVar,
  dest-typ = dest-TVar o Logic.dest-type o Logic.dest-term o fst o Logic.dest-equals
o Thm.concl-of,
  mk-term = fn ctxt => Thm.reflexive o Thm.cterm-of ctxt o Logic.mk-term o Var,
  dest-term = dest-Var o Logic.dest-term o fst o Logic.dest-equals o Thm.concl-of,
  wrap = K I,
  unwrap = I,
  mk-conjunctions = Conjunction.intr-balanced,
  dest-conjunctions = Conjunction.elim-balanced,
  morph = Morphism.thm
}

fun gen-morph-insts (ops:'a morph-operations) ctxt phi t (type-insts, term-insts) =
  if Morphism.is-identity phi then (t, (type-insts, term-insts))
  else
  let
```

```
    val Ts = map (#mk-typ ops ctxt o fst) type-insts
    val ts = map (#mk-term ops ctxt o fst) term-insts
    val n = length Ts + length ts + 1
    val (Ts', t'::ts') = #mk-conjunctions ops (Ts @ #wrap ops ctxt t::ts)
      |> #morph ops phi
      |> #dest-conjunctions ops n
      |> chop (length Ts)
    val type-insts' = map (#dest-typ ops) Ts' ~~ map snd type-insts
    val term-insts' = map (#dest-term ops) ts' ~~ map snd term-insts
    val t' = #unwrap ops t'
  in
    (t', (type-insts', term-insts'))
  end

val morph-insts-term = gen-morph-insts term-morph-operations
val morph-insts-cterm = gen-morph-insts cterm-morph-operations
val morph-insts-thm = gen-morph-insts thm-morph-operations

fun prep-insts (insts: insts) =
  let
    val maxidx = ~1
      |> fold Term.maxidx-typ (map snd (#1 insts))
      |> fold Term.maxidx-term (map snd (#2 insts))
    val insts = (map (incr-instT (maxidx + 1)) (#1 insts), map (incr-inst (maxidx
+ 1)) (#2 insts))
  in (insts, maxidx) end

fun init-tenv tinsts = Vartab.make (map (fn ((xi, S), T) => (xi, (S, T))) tinsts)
fun init-ctenv tinsts = Vartab.make (map (fn ((xi, S), cT) => (xi, (S, Thm.typ-of
cT))) tinsts)

fun morph-instantiate-term pos (insts: insts) t phi ctxt =
  let
    val (t, insts) = morph-insts-term ctxt phi t insts
    val (insts, maxidx) = prep-insts insts
    val t = t |> Logic.incr-indexes ([], maxidx + 1)
    val instT = TVars.make (#1 insts);
    val instantiateT = Term-Subst.instantiateT instT;
    val inst = (map o apfst o apsnd) instantiateT (#2 insts);
    val (tyenv, -) = fold (infer ctxt) inst (init-tenv (#1 insts), maxidx + 1)
    val instT = TVars.build (tyenv |> Vartab.fold (fn (xi, (S, T)) =>
          TVars.add ((xi, S), Envir.norm-type tyenv T)));
    val inst = Vars.build (#2 insts |> fold (fn ((xi, T), t) =>
         Vars.add ((xi, Envir.norm-type tyenv T),
          Term-Subst.instantiate (instT, Vars.empty) t)));
  in Term-Subst.instantiate-beta (instT, inst) t end
   handle TERM (msg, args) => Exn.reraise (TERM (msg ^ Position.here pos,
args));
```

```
fun instantiate-term pos (insts: insts) t = morph-instantiate-term pos (insts: insts)
t Morphism.identity

fun make-cinsts ctxt maxidx (cinsts: cinsts) =
  let
    val cinstT = TVars.make (#1 cinsts);
    val instantiateT = Term-Subst.instantiateT (TVars.map (K Thm.typ-of) cin-
stT);
    val cinst = (map o apfst o apsnd) instantiateT (#2 cinsts);
    val (tyenv, -) = fold (cterm-infer ctxt) cinst (init-ctenv (#1 cinsts), maxidx +
1)
    val cinstT = TVars.build (tyenv |> Vartab.fold (fn (xi, (S, T)) =>
          TVars.add ((xi, S), Thm.ctyp-of ctxt (Envir.norm-type tyenv T))));
    val thy = Proof-Context.theory-of ctxt
    val cinst =
      Vars.build (#2 cinsts |> fold (fn ((xi, T), cu) =>
        Vars.add ((xi, Envir.norm-type tyenv T),
              Thm.instantiate-cterm (cinstT, Vars.empty) (Thm.transfer-cterm thy
cu))));
  in (cinstT, cinst) end;

fun prep-cinsts cinsts =
 let
    val maxidx = ~1
      |> fold maxidx-ctyp (map snd (#1 cinsts))
      |> fold maxidx-cterm (map snd (#2 cinsts))
   val cinsts = (map (incr-instT (maxidx + 1)) (#1 cinsts), map (incr-inst (maxidx
+ 1)) (#2 cinsts))
  in (cinsts, maxidx) end

fun morph-instantiate-cterm pos cinsts ct phi ctxt =
  let
    val (ct, cinsts) = morph-insts-cterm ctxt phi ct cinsts
    val (cinsts, maxidx) = prep-cinsts cinsts
    val ct = Thm.incr-indexes-cterm (maxidx + 1) ct
  in
    Thm.instantiate-beta-cterm (make-cinsts ctxt maxidx cinsts) ct
  end
  handle CTERM (msg, args) => Exn.reraise (CTERM (msg ^ Position.here pos,
args))
       | TERM (msg, args) => Exn.reraise (TERM (msg ^ Position.here pos,
args));

fun instantiate-cterm pos cinsts ct = morph-instantiate-cterm pos cinsts ct Mor-
phism.identity

fun morph-instantiate-thm pos cinsts th phi ctxt =
  let
    val (th, cinsts) = morph-insts-thm ctxt phi th cinsts
```

```
    val (cinsts, maxidx) = prep-cinsts cinsts
    val th = Thm.incr-indexes (maxidx + 1) th
  in
    Thm.instantiate-beta (make-cinsts ctxt maxidx cinsts) th
  end
  handle THM (msg, i, args) => Exn.reraise (THM (msg ^ Position.here pos, i,
args))
       | TERM (msg, args) => Exn.reraise (TERM (msg ^ Position.here pos,
args));

fun instantiate-thm pos cinsts th = morph-instantiate-thm pos cinsts th Morphism.identity

fun morph-instantiate-thms pos cinsts ths phi ctxt = map (fn th => morph-instantiate-thm
pos cinsts th phi ctxt) ths;

fun instantiate-thms pos cinsts ths = morph-instantiate-thms pos cinsts ths Mor-
phism.identity


(* context data *)

structure Data = Proof-Data
(
  type T = int * thm list Inttab.table;
  fun init - = (0, Inttab.empty);
);

fun put-thms ths ctxt =
  let
    val (i, thms) = Data.get ctxt;
    val ctxt' = ctxt |> Data.put (i + 1, Inttab.update (i, map Thm.trim-context
ths) thms);
  in (i, ctxt') end;

fun get-thms ctxt i = the (Inttab.lookup (#2 (Data.get ctxt)) i);
fun get-thm ctxt i = the-single (get-thms ctxt i);


(* ML antiquotation *)

local

val make-keywords =
  Thy-Header.get-keywords'
  #> Keyword.no-major-keywords
  #> Keyword.add-major-keywords [term, prop, cterm, cprop, lemma];

val parse-inst-name = Parse.position (Parse.type-ident >> pair true || Parse.name
>> pair false);
```

```
val parse-inst =
  (parse-inst-name -- (Parse.$$$ = |-- Parse.!!! Parse.embedded-ml) ||
    Scan.ahead parse-inst-name -- Parse.embedded-ml)
  >> (fn (((b, a), pos), ml) => (b, ((a, pos), ml)));

val parse-insts =
  Parse.and-list1 parse-inst >> (List.partition #1 #> apply2 (map #2));

val ml = ML-Lex.tokenize-no-range;
val ml-range = ML-Lex.tokenize-range;
fun ml-parens x = ml ( @ x @ ml );
fun ml-bracks x = ml [ @ x @ ml ];
fun ml-commas xs = flat (separate (ml , ) xs);
val ml-list = ml-bracks o ml-commas;
fun ml-pair (x, y) = ml-parens (ml-commas [x, y]);
val ml-list-pair = ml-list o ListPair.map ml-pair;
val ml-here = ML-Syntax.atomic o ML-Syntax.print-position;

fun get-tfree envT (a, pos) =
  (case AList.lookup (op =) envT a of
    SOME S => (a, S)
  | NONE => error (No occurrence of type variable  ^ quote a ^ Position.here
pos));

fun check-free ctxt env (x, pos) =
  (case AList.lookup (op =) env x of
    SOME T =>
      (Context-Position.reports ctxt (map (pair pos) (Syntax-Phases.markup-free
ctxt x)); (x, T))
  | NONE => error (No occurrence of variable  ^ quote x ^ Position.here pos));

fun missing-instT pos envT instT =
  (case filter-out (fn (a, -) => exists (fn (b, -) => a = b) instT) envT of
    [] => ()
  | bad =>
      error (No instantiation for free type variable(s)  ^ commas-quote (map #1
bad) ^
        Position.here pos))
fun missing-inst pos env inst =
  (case filter-out (fn (a, -) => exists (fn (b, -) => a = b) inst) env of
    [] => ()
  | bad =>
      error (No instantiation for free variable(s)  ^ commas-quote (map #1 bad) ^
        Position.here pos));

fun make-instT (a, pos) T =
  (case try (Term.dest-TVar o Logic.dest-type) T of
    NONE => error (Not a free type variable  ^ quote a ^ Position.here pos)
```

```
  | SOME v => ml (ML-Syntax.print-pair ML-Syntax.print-indexname ML-Syntax.print-sort
v));

fun make-inst (a, pos) t =
  (case try Term.dest-Var t of
    NONE => error (Not a free variable " ^ quote a ^ Position.here pos)
  | SOME v => ml (ML-Syntax.print-pair ML-Syntax.print-indexname ML-Syntax.print-typ
v));

fun make-env ts =
  let
    val envT = fold Term.add-tfrees ts [];
    val env = fold Term.add-frees ts [];
  in (envT, env) end;

fun prepare-insts morph pos {schematic} ctxt1 ctxt0 (instT, inst) ts =
  let
    val (envT, env) = make-env ts;
    val freesT = map (Logic.mk-type o TFree o get-tfree envT) instT;
    val frees = map (Free o check-free ctxt1 env) inst;
    val (ts', (varsT, vars)) =
      Variable.export-terms ctxt1 ctxt0 (ts @ freesT @ frees)
      |> chop (length ts) ||> chop (length freesT);
    val ml-insts = (map2 make-instT instT varsT, map2 make-inst inst vars);
  in
    if schematic then ()
    else
      let val (envT', env') = make-env ts'
        val holes = subtract (eq-fst op =) env' env
        val inferableTs = fold Term.add-tfreesT (map snd holes) []
        val holesT = subtract (eq-fst op =) envT' envT
        val - = if null holesT andalso not morph
           then warning (All type instances are already explict and do not have to be
^
                    inferred from the term instantiations.\n ^
                   Use antiquotation 'instantiate' instead. ^ Position.here pos)
          else ()
        val holesT = subtract (eq-fst op =) inferableTs holesT
      in
        missing-instT pos holesT instT;
        missing-inst pos holes inst
      end;
    (ml-insts, ts')
  end;

fun prepare-ml range kind ml1 ml2 ml-val (ml-instT, ml-inst) ctxt =
  let
    val (ml-name, ctxt') = ML-Context.variant kind ctxt;
    val ml-env = ml (val " ^ ml-name ^ " = ) @ ml ml1 @ ml-parens (ml ml-val) @
```

```
ml ;\n;
    fun ml-body (ml-argsT, ml-args) =
      ml-parens (ml ml2 @
       ml-pair (ml-list-pair (ml-instT, ml-argsT), ml-list-pair (ml-inst, ml-args)) @
        ml-range range (ML-Context.struct-name ctxt ^ . ^ ml-name));
  in ((ml-env, ml-body), ctxt') end;

fun prepare-term morph read range ((((kind, pos), ml1, ml2), schematic), (s, fixes))
insts ctxt =
  let
    val ctxt' = #2 (Proof-Context.add-fixes-cmd fixes ctxt);
    val t = read ctxt' s;
    val ctxt1 = Proof-Context.augment t ctxt';
     val (ml-insts, t') = prepare-insts morph pos schematic ctxt1 ctxt insts [t] |>
the-single;
  in prepare-ml range kind ml1 ml2 (ML-Syntax.print-term t') ml-insts ctxt end;

fun prepare-lemma morph range ((pos, schematic), make-lemma) insts ctxt =
  let
    val (ths, (props, ctxt1)) = make-lemma ctxt
    val (i, thms-ctxt) = put-thms ths ctxt;
    val ml-insts = #1 (prepare-insts morph pos schematic ctxt1 ctxt insts props);
    val ml-instantiate-thm =
      if morph then
        ML-Infer-Instantiate.morph-instantiate-thm
        else
        ML-Infer-Instantiate.instantiate-thm
    val ml-instantiate-thms =
      if morph then
        ML-Infer-Instantiate.morph-instantiate-thms
        else
        ML-Infer-Instantiate.instantiate-thms
    val (ml1, ml2) =
      if length ths = 1
      then (ML-Infer-Instantiate.get-thm ML-context,
            ml-instantiate-thm ^ ml-here pos)
      else (ML-Infer-Instantiate.get-thms ML-context,
            ml-instantiate-thms ^ ml-here pos);
  in prepare-ml range lemma ml1 ml2 (ML-Syntax.print-int i) ml-insts thms-ctxt
end;

fun term-ml morph (kind, pos: Position.T) =
  let
    val ml-instantiate-term =
      if morph then
        ML-Infer-Instantiate.morph-instantiate-term
      else
        ML-Infer-Instantiate.instantiate-term
  in
```

```
    ((kind, pos), , ml-instantiate-term ^ ml-here pos)
  end

fun cterm-ml morph (kind, pos) =
  let
    val ml-instantiate-cterm =
      if morph then
        ML-Infer-Instantiate.morph-instantiate-cterm
      else
        ML-Infer-Instantiate.instantiate-cterm
  in
    ((kind, pos),
      ML-Infer-Instantiate.make-cterm ML-context,
        ml-instantiate-cterm ^ ml-here pos)
  end

val command-name = Parse.position o Parse.command-name;

val parse-schematic = Args.mode schematic >> (fn b => {schematic = b});

fun parse-body morph range =
  (command-name term >> term-ml morph || command-name cterm >> cterm-ml
morph) -- parse-schematic --
    Parse.!!! (Parse.term -- Parse.for-fixes) >> prepare-term morph Syntax.read-term
range ||
  (command-name prop >> term-ml morph || command-name cprop >> cterm-ml
morph) -- parse-schematic --
    Parse.!!! (Parse.term -- Parse.for-fixes) >> prepare-term morph Syntax.read-prop
range ||
  (command-name lemma >> #2) -- parse-schematic -- ML-Thms.embedded-lemma
>> prepare-lemma morph range;

fun antiquotation morph =
 (fn range => fn src => fn ctxt =>
      let
        val (insts, prepare-val) = src
          |> Parse.read-embedded ctxt (make-keywords ctxt)
              ((parse-insts --| Parse.$$$ in) -- parse-body morph range);

        val (((ml-env, ml-body), decls), ctxt1) = ctxt
          |> prepare-val (apply2 (map #1) insts)
          ||>> ML-Context.expand-antiquotes-list (op @ (apply2 (map #2) insts));

        fun decl' ctxt' =
          let val (ml-args-env, ml-args) = split-list (decls ctxt')
          in (ml-env @ flat ml-args-env, ml-body (chop (length (#1 insts)) ml-args))
end;
      in (decl', ctxt1) end)
```

```
val - = Theory.setup
 (ML-Context.add-antiquotation-embedded binding ‹infer-instantiate› (antiquotation
false) #>
  ML-Context.add-antiquotation-embedded binding ‹morph-infer-instantiate› (antiquotation
true))

in end;

end
›
```

Here an example with explicit instantiation.

```
ML-val ‹
val b = Free(n, typ ‹nat›)
val t1 = instantiate ‹'a = typ ‹nat› and a = ‹@{term 1::nat}› and b=‹b›
     in term  ‹a + b› for a b::'a::plus›
›
```

With the new variant the type instantiation can be omitted as it is already inferred from the term instantiations. Note that the result of the antiquotation is function that expects a `Proof.context`. This is used to perform the necessary type inference and provide context sensitive error messages.

```
ML-val ‹
val b = Free(n, typ ‹nat›)
val t1 = infer-instantiate ‹a =term ‹1::nat› and b=‹b›
     in term  ‹a + b› for a b::'a::plus› @{context}
›
```

This can be shortened:

```
ML-val ‹
val b = Free(n, typ ‹nat›)
val t1 = infer-instantiate ‹a = term ‹1::nat› and b=‹b›
     in term  ‹a + b›› @{context}
›
```

Here is an example were not all type variables can be inferred from the term instantiation. Those types can be explicitly instantiated.

```
ML-val ‹
val t1 = infer-instantiate ‹'b = typ ‹int› and a = ‹@{term 1::nat}›
     in term  ‹Pair a› for a ::'a › @{context}
›

ML-val ‹
val t1 = morph-infer-instantiate ‹'b = typ ‹int› and a = ‹@{term 1::nat}›
     in term  ‹Pair a› for a ::'a › Morphism.identity @{context}
›

ML-val ‹
```

```
val v = Proof-Context.read-term-pattern @{context} v
val t1 = infer-instantiate ‹'b = typ ‹int› and a = v
    in term ‹Pair a› for a ::'a › @{context}
›

ML-val ‹
val v = Proof-Context.read-term-pattern @{context} v::'a::type |> Thm.cterm-of
@{context}
val t1 = infer-instantiate ‹'b = ctyp ‹int› and a = v
    in cterm ‹Pair a› for a ::'a › @{context}
›

ML-val ‹
val t1 = infer-instantiate ‹'b = ctyp ‹int› and a = ‹@{cterm 1::nat}›
    in cterm ‹Pair a› for a ::'a › @{context}
›

context
  fixes x::'a::plus
begin
ML-val ‹
val b = Free(n, typ ‹nat›)
val t1 = infer-instantiate ‹a = ‹@{term 1::nat}› and b=‹b›
in term ‹a + b› for a b› @{context}
›
end
```

Note that the antiquotation only makes some local type-inference on the term parameters alone. There might be further use cases to perform a type inference on the whole term instead. This might be further fine-tuned by inserting explicit dummy types to be inferred: -.

```
ML ‹
fun infer-types-simple ctxt =
   singleton (Type-Infer-Context.infer-types-finished ctxt);

val b = Free(n, dummyT)
val t1 = instantiate ‹'a = typ ‹-› and a = ‹@{term 1::nat}› and b=‹b›
in prop ‹a + b = a + b› for a b::'a::plus › |> infer-types-simple @{context}
›

end
```

```
theory Arrays
imports
  HOL−Library.Numeral-Type
  Match-Cterm
```

ML-Infer-Instantiate
begin

definition *has-size* :: *'a set* ⇒ *nat* ⇒ *bool* **where**
has-size s n = (*finite s* ∧ *card s* = *n*)

— If *'a* is not finite, there is no *n* < *CARD*(*'a*)
definition *finite-index* :: *nat* ⇒ *'a*::*finite* **where**
finite-index = (*SOME f*. ∀ *x*. ∃!*n*. *n* < *CARD*(*'a*) ∧ *f n* = *x*)

lemma *card-image-inj*:
⟦ *finite S*; ⋀*x y*. ⟦ *x* ∈ *S*; *y* ∈ *S*; *f x* = *f y* ⟧ ⟹ *x* = *y* ⟧ ⟹ *card* (*f* ‘ *S*) = *card* *S*
by (*induct rule*: *finite-induct*) (*auto simp*: *card-insert-if*)

lemma *has-size-image-inj*:
⟦ *has-size S n*; ⋀*x y*. *x* ∈ *S* ∧ *y* ∈ *S* ∧ *f x* = *f y* ⟹ *x* = *y* ⟧ ⟹ *has-size* (*f* ‘ *S*) *n*
by (*simp add*: *has-size-def card-image-inj*)

lemma *has-size-0*[*simp*]:
has-size S 0 = (*S* = {})
by (*auto simp*: *has-size-def*)

lemma *has-size-suc*:
has-size S (*Suc n*) = (*S* ≠ {} ∧ (∀ *a*. *a* ∈ *S* ⟶ *has-size* (*S* − {*a*}) *n*))
unfolding *has-size-def*
by (*metis Diff-empty Suc-not-Zero bot-least card-Suc-Diff1 card-gt-0-iff finite-Diff-insert nat.inject neq0-conv subsetI subset-antisym*)

lemma *has-index*:
⟦ *finite S*; *card S* = *n* ⟧ ⟹
(∃*f*. (∀ *m*. *m* < *n* ⟶ *f m* ∈ *S*) ∧ (∀ *x*. *x*∈*S* ⟶ (∃!*m*. *m* < *n* ∧ *f m* = *x*)))
proof (*induct n arbitrary*: *S*)
case *0* **thus** *?case* **by** (*auto simp*: *card-eq-0-iff*)
next
case (*Suc n*)
then obtain *b B* **where**
S: *S* = *insert b B* ∧ *b* ∉ *B* ∧ *card B* = *n* ∧ (*n* = *0* ⟶ *B* = {})
by (*auto simp*: *card-Suc-eq*)
with ‹*finite S*› *Suc.hyps* [*of B*]
obtain *f* **where** *IH*: (∀ *m*<*n*. *f m* ∈ *B*) ∧ (∀ *x*. *x* ∈ *B* ⟶ (∃!*m*. *m* < *n* ∧ *f m* = *x*)) **by** *auto*
define *f'* **where** *f'* ≡ λ*m*. *if m* = *card B then b else f m*
from *Suc.prems S IH*
have (∀ *m*<*Suc n*. *f'* *m* ∈ *S*) ∧ (∀ *x*. *x* ∈ *S* ⟶ (∃!*m*. *m* < *Suc n* ∧ *f'* *m* = *x*))
unfolding *f'-def*
apply (*clarsimp*)
apply (*rule conjI*, *metis less-SucE*)

apply (*metis less-SucE less-SucI nat-neq-iff*)
done
thus *?case* **by** *blast*
qed

lemma *finite-index-works*:
$\exists!n.\ n < CARD('n::finite) \wedge$ *finite-index* $n = (i::'n)$
proof −
have $\exists f::nat \Rightarrow 'n.\ \forall i.\ \exists!n.\ n < CARD('n) \wedge f\ n = i$
using *has-index*[**where** $S = UNIV :: 'n\ set$] **by** *simp*
hence $\forall i.\ \exists!n.\ n < CARD('n::finite) \wedge$ *finite-index* $n = (i::'n)$
unfolding *finite-index-def* **by** (*rule someI-ex*)
thus *?thesis* **..**
qed

lemma *finite-index-inj*:
$[\![\ i < CARD('a::finite);\ j < CARD('a)\ ]\!] \Longrightarrow$
((*finite-index* $i :: 'a$) = *finite-index* j) = $(i = j)$
using *finite-index-works*[**where** $i =$ *finite-index* j] **by** *blast*

lemma *forall-finite-index*:
$(\forall k::'a::finite.\ P\ k) = (\forall i.\ i < CARD('a) \longrightarrow P\ (finite\text{-}index\ i))$
by (*metis finite-index-works*)

11.14 Finite Cartesian Products

typedef ('*a*,'*n*::*finite*) *array*
(‹(‹*open-block notation*=‹*mixfix array*››-[-])› [*30*,*0*] *31*) = *UNIV* :: ('*n* => '*a*)
set
by *simp*
setup-lifting *type-definition-array*

lift-definition *index* :: ('*a*,'*n*::*finite*) *array* ⇒ *nat* ⇒ '*a*
(‹(‹*open-block notation*=‹*mixfix array index*››-.[-])› [*900*,*0*] *901*) **is**
$\lambda f\ i.\ f\ (finite\text{-}index\ i)$**.**

lemma *index-legacy-def*: *index x i* = *Rep-array x* (*finite-index i*)
by (*transfer*) *simp*

theorem *array-index-eq*:
$((x::'a['n::finite]) = y) = (\forall i.\ i < CARD('n) \longrightarrow x.[i] = y.[i])$
by (*auto dest*!: *forall-finite-index* [*THEN iffD2*]
simp: *index-def Rep-array-inject* [*symmetric*])

lemmas *cart-eq* = *array-index-eq*

lemma *array-ext*:
fixes $x :: 'a['n::finite]$

shows $(\bigwedge i.\ i < CARD('n) \Longrightarrow x.[i] = y.[i]) \Longrightarrow x = y$
by (*simp add*: *array-index-eq*)

definition $FCP :: (nat \Rightarrow 'a) \Rightarrow 'a['b::finite]$ (**binder** ‹*ARRAY* › *10*) **where**
$FCP \equiv \lambda g.\ SOME\ a.\ \forall i.\ i < CARD('b) \longrightarrow a.[i] = g\ i$

definition $update :: 'a['n::finite] \Rightarrow nat \Rightarrow 'a \Rightarrow 'a['n]$ **where**
$update\ f\ i\ x \equiv FCP\ ((index\ f)(i := x))$

definition $fupdate :: nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a['b::finite] \Rightarrow 'a['b]$ **where**
$fupdate\ i\ f\ x \equiv update\ x\ i\ (f\ (index\ x\ i))$

lemma *fcp-beta*[*simp*]: $i < CARD('n :: finite) \Longrightarrow (FCP\ g :: 'a['n]).[i] = g\ i$
proof −
have $\forall i < CARD('n).\ (FCP\ g :: 'a['n]).[i] = g\ i$
unfolding *FCP-def*
proof (*rule someI-ex*)
let $?g' = \lambda k::'n.\ g\ (SOME\ i.\ i < CARD('n) \wedge finite\text{-}index\ i = k)$
have $\forall i{<}CARD('n).\ (Abs\text{-}array\ ?g').[i] = g\ i$
by (*clarsimp simp*: *index-def Abs-array-inverse*)
(*metis* (*mono-tags*, *lifting*) *finite-index-inj someI-ex*)
thus $\exists x::'a['n].\ \forall i{<}CARD('n).\ x.[i] = g\ i$ **..**
qed
thus $i < CARD('n :: finite) \Longrightarrow (FCP\ g :: 'a['n]).[i] = g\ i$ **by** *auto*
qed

lemma *fcp-unique*:
$(\forall i.\ i < CARD('b::finite) \longrightarrow index\ f\ i = g\ i) =$
$(FCP\ g = (f :: ('a,'b)\ array))$
by (*fastforce simp*: *cart-eq*)

lemma *fcp-eta*[*simp*]:
$(ARRAY\ i.\ g.[i]) = g$
by (*simp add*: *cart-eq*)

lemma *index-update*[*simp*]:
$n < CARD('b::finite) \Longrightarrow index\ (update\ (f::'a['b])\ n\ x)\ n = x$
by (*simp add*: *update-def*)

lemma *index-update2*[*simp*]:
$[\![\ k < CARD('b::finite);\ n \neq k\ ]\!] \Longrightarrow index\ (update\ (f::'a['b])\ n\ x)\ k = index\ f\ k$
by (*simp add*: *update-def*)

lemma *update-update*[*simp*]:
$update\ (update\ f\ n\ x)\ n\ y = update\ f\ n\ y$
by (*simp add*: *cart-eq update-def*)

lemma *update-comm*[*simp*]:
$n \neq m \Longrightarrow update\ (update\ f\ m\ v)\ n\ v' = update\ (update\ f\ n\ v')\ m\ v$

by (*simp add*: *cart-eq update-def*)

lemma *update-index-same* [*simp*]:
update v n (*index v n*) = *v*
by (*simp add*: *cart-eq update-def*)

function *foldli0* :: (*nat* ⇒ *'acc* ⇒ *'a* ⇒ *'acc*) ⇒ *'acc* ⇒ *nat* ⇒ *'a*['*index*::*finite*] ⇒ *'acc* **where**
foldli0 f a i arr = (*if CARD*('*index*) ≤ *i then a else foldli0 f* (*f i a* (*index arr i*)) (*i*+*1*) *arr*)
by *pat-completeness auto*

termination
by (*relation measure* (λ(*f*,*a*,*i*,(*arr*::*'b*['*c*::*finite*])). *CARD*('*c*) − *i*)) *auto*

definition *foldli* :: (*nat* ⇒ *'acc* ⇒ *'a* ⇒ *'acc*) ⇒ *'acc* ⇒ (*'a*,*'i*::*finite*) *array* ⇒ *'acc* **where**
foldli f a arr = *foldli0 f a 0 arr*

definition *map-array* :: (*'a* ⇒ *'b*) ⇒ *'a*['*n*::*finite*] ⇒ *'b*['*n*] **where**
map-array f a ≡ *ARRAY i*. *f* (*a*.[*i*])

definition *map-array2* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'a*['*n*::*finite*] ⇒ *'b*['*n*] ⇒ *'c*['*n*] **where**
map-array2 f a b ≡ *ARRAY i*. *f* (*a*.[*i*]) (*b*.[*i*])

definition *zip-array* :: *'a*['*b*::*finite*] ⇒ *'c*['*b*] ⇒ (*'a* × *'c*)['*b*] **where**
zip-array ≡ *map-array2 Pair*

definition *list-array* :: (*'a*,*'n*::*finite*) *array* ⇒ *'a list* **where**
list-array a = *map* (*index a*) [*0*..<*CARD*('*n*)]

lift-definition *set-array* :: *'a*['*n*::*finite*] ⇒ *'a set* **is** *range* **.**

lemma *set-array-list*:
set-array a = *set* (*list-array a*)
by (*simp add*: *list-array-def index-def set-array.rep-eq image-def*)
(*metis atLeast0LessThan finite-index-works lessThan-iff*)

definition *rel-array* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a*['*n*::*finite*] ⇒ *'b*['*n*] ⇒ *bool* **where**
rel-array f a b ≡ ∀ *i* < *CARD*('*n*). *f* (*a*.[*i*]) (*b*.[*i*])

lemma *map-array-index*:
fixes *a* :: *'a*['*n*::*finite*]
shows *n* < *CARD*('*n*) ⟹ (*map-array f a*).[*n*] = *f* (*a*.[*n*])
by (*metis fcp-beta map-array-def*)

lemma *map-array2-index*:

fixes $a :: 'a['n::finite]$
shows $n < CARD('n) \Longrightarrow (map\text{-}array2\ f\ a\ b).[n] = f\ (a.[n])\ (b.[n])$
by (*metis fcp-beta map-array2-def*)

lemma *zip-array-index*:
fixes $a :: 'a['n::finite]$
shows $n < CARD('n) \Longrightarrow (zip\text{-}array\ a\ b).[n] = (a.[n],b.[n])$
by (*simp add*: *zip-array-def map-array2-index*)

lemma *map-array-id*[*simp*]:
$map\text{-}array\ id = id$
by (*auto simp*: *map-array-index array-ext*)

lemma *map-array-comp*:
$map\text{-}array\ (g \circ f) = map\text{-}array\ g \circ map\text{-}array\ f$
by (*auto simp*: *map-array-def intro*!: *array-ext*)

lemma *list-array-nth*:
fixes $a :: 'a['n::finite]$
shows $n < CARD('n) \Longrightarrow list\text{-}array\ a\ !\ n = index\ a\ n$
by (*simp add*: *list-array-def*)

lemma *list-array-length* [*simp*]:
$length\ (list\text{-}array\ (a :: 'a['n::finite])) = CARD('n)$
by (*simp add*: *list-array-def*)

lemma *in-set-array-index-conv*:
$(z \in set\text{-}array\ (a :: 'a['n::finite])) = (\exists\, n < CARD('n).\ z = a.[n])$
by (*metis in-set-conv-nth list-array-length list-array-nth nth-mem set-array-list*)

lemma *in-set-arrayE* [*elim*!]:
$[\![\ z \in set\text{-}array\ (a :: 'a['n::finite]);\ \bigwedge n\ .\ [\![n < CARD('n);\ z = a.[n]]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*metis in-set-array-index-conv*)

lemma *map-array-setI*:
$(\bigwedge z.\ z \in set\text{-}array\ x \Longrightarrow f\ z = g\ z) \Longrightarrow map\text{-}array\ f\ x = map\text{-}array\ g\ x$
by (*rule array-ext*) (*auto simp*: *map-array-index in-set-array-index-conv*)

lemma *list-array-map-array*:
$list\text{-}array\ (map\text{-}array\ f\ a) = map\ f\ (list\text{-}array\ a)$
by (*simp add*: *list-array-def map-array-index*)

lemma *list-array-FCP* [*simp*]:
$list\text{-}array\ (FCP\ f :: 'a['n]) = map\ f\ [0..<CARD('n::finite)]$
by (*simp add*: *list-array-def*)

lemma *set-array-FCP* [*simp*]:
$set\text{-}array\ (FCP\ f :: 'a['n]) = f\ `\ \{0..<\ CARD('n::finite)\}$

by (*auto simp*: *set-array-list*)

lemma *map-array-set-img*:
set-array ∘ *map-array f* = (‘) *f* ∘ *set-array*
by (*rule ext*) (*auto simp*: *map-array-def in-set-array-index-conv intro*!: *imageI*)

lemma *fcp-cong* [*cong*]:
(⋀*i*. *i* < *CARD*('*n*::*finite*) ⟹ *f i* = *g i*) ⟹ *FCP f* = (*FCP g* :: '*a*['*n*])
by (*rule array-ext*) *simp*

bnf ('*a*,'*n*::*finite*) *array*
map: *map-array*
sets: *set-array*
bd: *card-suc* (*BNF-Cardinal-Arithmetic.csum natLeq* (*card-of* (*UNIV* :: '*n set*)))
rel: *rel-array*
proof −
show *map-array id* = *id* **by** *simp*
next
fix *f* :: '*a* ⇒ '*b* **and** *g* :: '*b* ⇒ '*c*
show *map-array* (*g* ∘ *f*) = *map-array g* ∘ *map-array f*
by (*rule map-array-comp*)
next
fix *x* :: '*a*['*n*::*finite*] **and** *f* :: '*a* ⇒ '*b* **and** *g*
assume ⋀*z*. *z* ∈ *set-array x* ⟹ *f z* = *g z*
thus *map-array f x* = *map-array g x*
by (*rule map-array-setI*)
next
fix *f* :: '*a* ⇒ '*b*
show *set-array* ∘ *map-array f* = (‘) *f* ∘ *set-array*
by (*rule map-array-set-img*)
next
show *card-order* (*card-suc* (*BNF-Cardinal-Arithmetic.csum natLeq* (*card-of UNIV*)))
by (*simp add*: *card-of-card-order-on card-order-card-suc card-order-csum natLeq-card-order*)
next
show *BNF-Cardinal-Arithmetic.cinfinite* (*card-suc* (*BNF-Cardinal-Arithmetic.csum natLeq* (*card-of UNIV*)))
by (*simp add*: *Card-order-csum Cinfinite-card-suc card-of-card-order-on card-order-csum cinfinite-csum natLeq-Cinfinite natLeq-card-order*)
next
fix *R* :: '*a* ⇒ '*b* ⇒ *bool* **and** *S* :: '*b* ⇒ '*c* ⇒ *bool*
show *rel-array R OO rel-array S* ≤ *rel-array* (*R OO S*)
by (*force simp*: *rel-array-def*)
next
fix *R* :: '*a* ⇒ '*b* ⇒ *bool*
{ fix *a* :: ('*a* × '*b*)['*n*::*finite*] **and** *i*
have ⟦*set-array a* ⊆ {(*x*, *y*). *R x y*}; *i* < *CARD*('*n*)⟧ ⟹ *R* (*fst* (*a*.[*i*])) (*snd* (*a*.[*i*]))
by (*meson Collect-case-prodD in-set-array-index-conv subset-iff*)

```
  } note conv = this
  show rel-array R =
        (λx y. ∃ z. set-array z ⊆ {(x, y). R x y} ∧ map-array fst z = x ∧ map-array
snd z = y)
    unfolding rel-array-def
    apply (intro ext iffI)
    subgoal for a b
     apply (rule exI [where x=zip-array a b])
      by (auto intro!: array-ext simp: conv map-array-index zip-array-index)
    subgoal
      by (auto intro!: array-ext simp: conv map-array-index zip-array-index)
    done
next
 show regularCard (card-suc (BNF-Cardinal-Arithmetic.csum natLeq (card-of UNIV)))
     by (simp add: Card-order-csum card-of-card-order-on card-order-csum cinfi-
nite-csum
       natLeq-Cinfinite natLeq-card-order regularCard-card-suc)
next
  fix x :: 'a['n::finite]
  let ?U = UNIV :: 'n set
 have ordLeq3 (card-of (set-array x)) (card-of ?U) by transfer (rule card-of-image)
  also
 have ordLeq3 (card-of ?U) (BNF-Cardinal-Arithmetic.csum natLeq (card-of ?U))
   by (rule ordLeq-csum2) (rule card-of-Card-order)
  finally
   have ordLeq3 (card-of (set-array x)) (BNF-Cardinal-Arithmetic.csum natLeq
(card-of ?U)) .
  then show ordLess2 (card-of (set-array x))
        (card-suc (BNF-Cardinal-Arithmetic.csum natLeq (card-of ?U)))
    using ordLeq-ordLess-trans card-suc-greater card-order-csum natLeq-card-order
        card-of-card-order-on
    by blast
qed

lemma arr-fupdate-same: i < CARD('n) ⟹
  index (fupdate i f (x::'a['n::finite])) i = f (index x i)
  by (simp add: fupdate-def)

lemma arr-fupdate-other: j < CARD('n) ⟹ i ≠ j ⟹
  index (fupdate i f (x::'a['n::finite])) j = (index x j)
  by (simp add: fupdate-def)

lemmas arr-fupdate-simps = arr-fupdate-same arr-fupdate-other

lemma surj-finite-index: surj finite-index
  by (metis finite-index-works surj-def)

lemma finite-index-inj-on: inj-on (finite-index::nat⇒'a::finite) {i. i < CARD('a)}
  by (simp add: finite-index-inj inj-on-def)
```

lemma *finite-index-bij-betw*: *bij-betw* (*finite-index*::*nat*⇒'*a*::*finite*) {*i*. *i* < *CARD*('*a*)} *UNIV*
using *finite-index-inj-on*
by (*metis* (*mono-tags*, *lifting*) *UNIV-I bij-betw-def finite-index-works imageI mem-Collect-eq subsetI subset-antisym*)

simproc-setup passive *fold-update* (‹*Arrays.update a n v*›) = ‹*K* (*fn ctxt* => *fn ct* =>
— Prefer more abstract *fupdate* over *update*
let
val {*a*, *n*, *v*, ...} = @{*cterm-match Arrays.update ?a ?n ?v*} *ct*
val elem = ***infer-instantiate*** ‹*a* = *a and n* = *n in cterm* ‹*a*.[*n*]›› *ctxt*
val elem-t = *Thm.term-of elem*
in
if exists-subterm (*fn t* => *elem-t aconv t*) (*Thm.term-of v*) *then*
SOME (*Drule.infer-instantiate ctxt*
[((*a*,*0*), *a*), ((*n*,*0*), *n*), ((*f*, *0*), *Thm.lambda-name* (*v*, *elem*) *v*)]
@{*thm Arrays.fupdate-def*[*symmetric*]})
else
NONE
end)›

experiment
begin
lemma *Arrays.update a n* (*k* − *a*.[*n*] + *x*) = *fupdate n* (λ*v*. *k* − *v* + *x*) *a*
supply [[*simproc add*: *fold-update*]]
apply (*simp*)
done
end
end

theory *Padding*
imports *Main*
begin

lemma (*0*::*nat*) *mod 2* = *XXX*
apply (*simp*)
oops

lemma (*2* − (*0*::*nat*) *mod 2*) = *XXX*
apply (*simp*)
oops

lemma (*2* − (*0*::*nat*) *mod 2*) *mod 2* = *XXX*
apply (*simp*)
oops

definition *padup* :: *nat* ⇒ *nat* ⇒ *nat* **where**
padup align n ≡ (*align* − *n mod align*) *mod align*

lemma *padup-dvd*:
0 < *b* ⟹ (*padup b n* = *0*) = (*b dvd n*)
unfolding *padup-def*
apply(*subst dvd-eq-mod-eq-0*)
apply(*subst mod-if* [**where** *m*=*b* − *n mod b*])
apply *clarsimp*
apply(*insert mod-less-divisor* [*of b n*])
apply *arith*
done

lemma *dvd-padup-add*:
0 < *x* ⟹ *x dvd y* + *padup x y*
apply(*clarsimp simp*: *padup-def*)
apply(*subst mod-if* [**where** *m*=*x* − *y mod x*])
apply(*clarsimp split*: *if-split-asm*)
apply(*rule conjI*)
apply *clarsimp*
apply(*subst ac-simps*)
apply(*subst diff-add-assoc*)
apply(*rule mod-less-eq-dividend*)
apply(*rule dvd-add*)
apply *simp*
apply(*subst minus-div-mult-eq-mod*[*symmetric*])
apply(*subst diff-diff-right*)
apply(*subst ac-simps*)
apply(*subst minus-mod-eq-mult-div*[*symmetric*])
apply *simp*
apply *simp*
apply(*auto simp*: *dvd-eq-mod-eq-0*)
done

end

theory *Lens*
imports *Main Distinct-Prop*
begin

11.15 Auxiliary Theorems

lemma *distinct-prop-cons*:
list-all (*R x*) *xs* ⟹ *distinct-prop R xs* ⟹ *distinct-prop R* (*x* # *xs*)
by (*simp add*: *list-all-iff*)

lemma *distinct-prop-distrib-all*:
distinct-prop (λ*x y*. ∀ *a*. *R a x y*) *xs* ⟷ (∀ *a*. *distinct-prop* (*R a*) *xs*)
by (*induction xs*) *auto*

lemma *pairwise-set-of-distinct-prop*:
(⋀*a b*. *R a b* ⟷ *R b a*) ⟹ *distinct-prop R xs* ⟹ *pairwise R* (*set xs*)
by (*induction xs*) (*auto simp*: *pairwise-insert*)

lemma *distinct-prop-iff-nth*:
distinct-prop R xs ⟷ (∀ *i j*. *i* < *j* ⟶ *j* < *length xs* ⟶ *R* (*xs*!*i*) (*xs*!*j*))
proof (*induction xs*)
have *split*: (∀ *i*. *P i*) ⟷ *P 0* ∧ (∀ *i*. *P* (*Suc i*)) **for** *P* :: *nat* ⇒ *bool*
by (*metis not0-implies-Suc*)

case (*Cons x xs*)
then have *distinct-prop R* (*x* # *xs*) ⟷
(∀ *j*. *j* < *length xs* ⟶ *R x* (*xs* ! *j*)) ∧
(∀ *i j*. *i* < *j* ⟶ *j* < *length xs* ⟶ *R* (*xs* ! *i*) (*xs* ! *j*))
by (*auto simp add*: *set-conv-nth*)
also have . . . ⟷ (∀ *i j*. *i* < *j* ⟶ *j* < *Suc* (*length xs*) ⟶ *R* ((*x* # *xs*) ! *i*) ((*x* # *xs*) ! *j*))
apply (*subst* (*4*) *split*)
apply (*subst* (*4 6*) *split*)
apply *simp*
done
finally show *?case* **by** *simp*
qed *simp*

lemma *distinct-prop-mono*:
(⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *P x y* ⟹ *R x y*) ⟹
distinct-prop P xs ⟹ *distinct-prop R xs*
by (*induction xs*) *auto*

lemma *distinct-prop-nil*: *distinct-prop R* []
by *simp*

lemma *list-all-concat*: *list-all p* (*concat xs*) = *list-all* (*list-all p*) *xs*
by (*induction xs*) *auto*

lemma *list-all-conj*: *list-all P xs* ⟹ *list-all Q xs* ⟹ *list-all* (λ*x*. *P x* ∧ *Q x*) *xs*
using *Ball-set* **by** *blast*

lemma *list-all-zip-iff-list-all2*:
length as = *length bs* ⟹ *list-all R* (*zip as bs*) ⟷ *list-all2* (λ*a b*. *R* (*a*, *b*)) *as bs*
by (*simp add*: *Ball-set-list-all list-all2-iff*)

lemma *disjnt-comm*: *disjnt A B* ⟷ *disjnt B A*
by (*metis disjnt-sym*)

lemma *disjnt-image*: *disjnt* (*f* ‘ *x*) (*f* ‘ *y*) ⟹ *disjnt x y*
by (*auto simp*: *disjnt-def*)

lemma *disjnt-of-nat*:
s ⊆ {*0*..<*2*^*LENGTH*('*a*::*len*)} ⟹ *t* ⊆ {*0*..<*2*^*LENGTH*('*a*)} ⟹
disjnt ((*of-nat* :: *nat* ⇒ '*a word*) ‘ *s*) (*of-nat* ‘ *t*) ⟷ *disjnt s t*
proof (*rule disjnt-inj-on-iff*[*of* - *Pow* {*0*..<*2*^*LENGTH*('*a*)}])
qed (*simp-all add*: *inj-on-word-of-nat*)

lemma *list-all-zip-zip-cons*:
R a b c ⟹
list-all (λ(*a*, *b*, *c*). *R a b c*) (*zip as* (*zip bs cs*)) ⟹
list-all (λ(*a*, *b*, *c*). *R a b c*) (*zip* (*a*#*as*) (*zip* (*b*#*bs*) (*c*#*cs*)))
by *simp*

lemma *list-all-zip-zip-empty*:
list-all (λ(*a*, *b*, *c*). *R a b c*) (*zip* [] (*zip* [] []))
by *simp*

lemma *list-all-cons*: *P x* ⟹ *list-all P xs* ⟹ *list-all P* (*x* # *xs*)
by *simp*

lemma *list-all-nil*: *list-all P* []
by *simp*

lemma *fold-functor*:
F id = *id* ⟹ (⋀*a b*. *F* (*a* ∘ *b*) = *F a* ∘ *F b*) ⟹ *F* (*fold a xs*) = *fold* (λ*x*. *F* (*a x*)) *xs*
by (*induction xs*; *simp*)

11.16 Legacy definition *fg-cons*

definition *fg-cons* :: ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*a* ⇒ '*a*) ⇒ *bool* **where**
fg-cons acc upd ≡
(∀ *bs v*. *acc* (*upd bs v*) = *bs*) ∧ (∀ *bs bs*' *v*. *upd bs* (*upd bs*' *v*) = *upd bs v*) ∧ (∀ *v*. *upd* (*acc v*) *v* = *v*)

11.17 Lense definition using an update function *lense*

type-synonym '*a upd* = '*a* ⇒ '*a*

named-theorems *update-compose*

locale *lense* =
fixes
get :: '*s* ⇒ '*a* **and**

```
  upd :: 'a upd ⇒ 's upd
  assumes get-upd    [simp]: get (upd f s) = f (get s)
  assumes upd-same         : f (get s) = get s ⟹ upd f s = s
  assumes upd-compose[update-compose, simp]: upd f (upd g s) = upd (f o g) s
begin

lemma upd-comp[simp]: upd f ∘ upd g = upd (f ∘ g)
  by (simp add: fun-eq-iff)
lemma upd-get [simp]: upd (λ-. get s) s = s
  by (simp add: upd-same)
lemma upd-id [simp]: upd (λx. x) s = s
  by (simp add: upd-same)
lemma upd-cong: f (get s) = f' (get s) ⟹ upd f s = upd f' s
  by (metis fun-upd-same get-upd upd-compose upd-same)

lemma upd-comp-fold: upd (fold f xs) = fold (upd ∘ f) xs
  apply (induction xs)
  apply (simp-all add: upd-same del: comp-apply flip: upd-comp)
  apply (simp add: fun-eq-iff comp-def)
  done

lemma fg-cons: fg-cons get (upd ∘ (λx -. x))
  unfolding fg-cons-def o-def by (simp add: comp-def)

end


lemma lenseI:
  fixes get upd
  assumes 1:(⋀s f. get (upd f s) = f (get s)) (⋀s. upd (λx. get s) s = s)
    and 2[cong]: (⋀s f g. f (get s) = g (get s) ⟹ upd g s = upd f s)
    and 3: (⋀s f g. upd f (upd g s) = upd (f o g) s)
  shows lense get upd
  using 1 3
  by (simp add: lense-def)

lemma lenseI-equiv:
  (⋀x. f (g x) = x) ⟹ (⋀x. g (f x) = x) ⟹ lense g (λv x. f (v (g x)))
  by (simp add: lense-def)

lemma lense-comp:
  assumes lense1: lense sel1 upd1
  assumes lense2: lense sel2 upd2
  shows lense (sel2 o sel1) (upd1 o upd2)
proof −
  interpret lense1: lense sel1 upd1 by (rule lense1)
  interpret lense2: lense sel2 upd2 by (rule lense2)
  show ?thesis
    apply (unfold-locales)
```

subgoal by (*simp add*: *comp-def*)
subgoal using *lense1.upd-same lense2.upd-same* **by** *simp*
subgoal using *lense1.upd-compose lense2.upd-compose* **by** (*simp add*: *comp-def*)
done
qed

lemma *lense-compose*:
assumes *lense sel1 upd1 lense sel2 upd2*
shows *lense* (λ*x*. *sel2* (*sel1 x*)) (λ*f*. *upd1* (*upd2 f*))
using *lense-comp*[*OF assms*] **by** (*simp add*: *comp-def*)

lemma *lense-of-fg-cons*:
fg-cons g s ⟹ *lense g* (λ*f x*. *s* (*f* (*g x*)) *x*)
by (*simp add*: *fg-cons-def lense-def*)

lemma *lense-of-fg-cons′*:
fg-cons g (*u* ∘ (λ*x* -. *x*)) ⟹
(⋀*f x*. *u f x* = *u* (λ-. *f* (*g x*)) *x*) ⟹
lense g u
by (*simp-all add*: *fg-cons-def fun-eq-iff comp-def lense-def*) *metis*

11.18 Update for functions

definition *upd-fun* :: *′a* ⇒ *′b upd* ⇒ (*′a* ⇒ *′b*) *upd* **where**
upd-fun i f g = *g*(*i* := *f* (*g i*))

global-interpretation *upd-fun*: *lense* λ*f*. *f a upd-fun a*
by (*simp add*: *lense-def upd-fun-def*)

lemma *upd-fun-ne*: *i* ≠ *j* ⟹ *upd-fun i f g j* = *g j*
by (*simp add*: *upd-fun-def fun-upd-def*)

lemma *upd-fun-commute*: *i* ≠ *j* ⟹ *upd-fun i f* ∘ *upd-fun j g* = *upd-fun j g* ∘ *upd-fun i f*
by (*auto simp*: *upd-fun-def fun-eq-iff fun-upd-def*)

11.19 Scenes

Scenes allow us to represent a projection/lense without referring to a second type.

type-synonym *′a scene* = *′a* ⇒ *′a* ⇒ *′a*

locale *is-scene* =
fixes *s* :: *′a scene*
assumes *left*[*simp*]: *s* (*s a b*) *c* = *s a c*
assumes *right*[*simp*]: *s a* (*s b c*) = *s a c*
assumes *idem*[*simp*]: *s a a* = *a*

lemma *is-scene-all*[*simp*]: *is-scene* (λ*a b. a*)
by (*simp add*: *is-scene-def*)

lemma *is-scene-id*[*simp*]: *is-scene* (λ*a. id*)
by (*simp add*: *is-scene-def*)

lemma *is-scene-flip*: *is-scene m* ⟹ *is-scene* (λ*a b. m b a*)
by (*simp add*: *is-scene-def*)

lemma *is-scene-of-lense*: *lense r w* ⟹ *is-scene* (λ*a. w* (λ*-. r a*))
by (*simp add*: *lense-def is-scene-def K-record-comp*)

lemma *is-scene-of-fg-cons*: *fg-cons r w* ⟹ *is-scene* (λ*a. w* (*r a*))
by (*simp add*: *fg-cons-def is-scene-def*)

lemma *scene-comp-idem*: *is-scene m* ⟹ *m a* ∘ *m a* = *m a*
by (*simp add*: *fun-eq-iff is-scene.right*)

definition *comm-scene* :: ′*a scene* ⇒ ′*a scene* ⇒ *bool* **where**
comm-scene m1 m2 ⟷ (∀ *a b. m1 a* (*m2 a b*) = *m2 a* (*m1 a b*))

lemma *comm-scene-comm*: *comm-scene m1 m2* ⟷ *comm-scene m2 m1*
by (*simp add*: *comm-scene-def*) *metis*

lemma *comm-scene-sym*[*intro*]: *comm-scene m1 m2* ⟹ *comm-scene m2 m1*
by (*simp add*: *comm-scene-comm*)

lemma *comm-sceneD*: *comm-scene m1 m2* ⟹ *m1 a* (*m2 a c*) = *m2 a* (*m1 a c*)
by (*simp add*: *comm-scene-def*)

lemma *comm-scene-all-left*[*simp*]: *is-scene m* ⟹ *comm-scene* (λ*a b. a*) *m*
by (*simp add*: *comm-scene-def is-scene.idem*)

lemma *comm-scene-all-right*[*simp*]: *is-scene m* ⟹ *comm-scene m* (λ*a b. a*)
by (*simp add*: *comm-scene-def is-scene.idem*)

lemma *comm-scene-id-left*[*simp*]: *comm-scene* (λ*a. id*) *m*
by (*simp add*: *comm-scene-def*)

lemma *comm-scene-id-right*[*simp*]: *comm-scene m* (λ*a. id*)
by (*simp add*: *comm-scene-def*)

lemma *comm-scene-refl*[*intro*, *simp*]: *comm-scene m m*
by (*simp add*: *comm-scene-def*)

lemma *is-scene-comp*:
is-scene m1 ⟹ *is-scene m2* ⟹ *comm-scene m1 m2* ⟹ *is-scene* (λ*a. m1 a* ∘ *m2 a*)
by (*simp add*: *is-scene-def comm-scene-def*) *metis*

lemma *comm-scene-comp-left*:
comm-scene m1 m2 $\Longrightarrow$ *comm-scene m1 m* $\Longrightarrow$ *comm-scene m2 m* $\Longrightarrow$ *comm-scene* ($\lambda a.$ *m1 a* $\circ$ *m2 a*) *m*
by (*simp add*: *is-scene-def comm-scene-def*)

lemma *comm-scene-comp-right*:
comm-scene m m1 $\Longrightarrow$ *comm-scene m m2* $\Longrightarrow$ *comm-scene m1 m2* $\Longrightarrow$ *comm-scene m* ($\lambda a.$ *m1 a* $\circ$ *m2 a*)
using *comm-scene-comp-left*[*of m1 m2 m*] **by** (*simp add*: *comm-scene-comm*)

lemma *comm-scene-fold*:
pairwise comm-scene (*insert m* (*set ms*)) $\Longrightarrow$ *comm-scene* ($\lambda a.$ *fold* ($\lambda m.$ *m a*) *ms*) *m*
by (*induction ms arbitrary*: *m*) (*auto intro*!: *comm-scene-comp-left simp*: *pairwise-def*)

lemma *is-scene-fold*:
list-all is-scene ms $\Longrightarrow$ *pairwise comm-scene* (*set ms*) $\Longrightarrow$ *is-scene* ($\lambda a.$ *fold* ($\lambda m.$ *m a*) *ms*)
by (*induction ms*) (*simp-all add*: *is-scene-comp comm-scene-fold pairwise-def*)

lemma *is-scene-fold'*:
list-all ($\lambda x.$ *is-scene* (*m x*)) *ms* $\Longrightarrow$ *pairwise comm-scene* (*m* ‘ *set ms*) $\Longrightarrow$
is-scene ($\lambda a.$ *fold* ($\lambda x.$ *m x a*) *ms*)
using *is-scene-fold*[*of map m ms*] **by** (*simp add*: *list.pred-map comp-def fold-map*)

This expresses "disjointness" on scenes, saying that two scenes occupy disjnt parts of the type.

It is stronger than "commutativity" $\forall\ a\ c.$ *m1 a* (*m2 a b*) = *m2 a* (*m1 a b*) which is enough to show composition, but in our cases we are interested in disjointness. Important difference: a scene *m* "commutes" with itself, but only $\lambda a.$ *id* is disjoint with itself.

definition *disjnt-scene* :: *'a scene* $\Rightarrow$ *'a scene* $\Rightarrow$ *bool* **where**
disjnt-scene m1 m2 $\longleftrightarrow$ ($\forall\ a\ b\ c.$ *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*))

lemma *comm-scene-of-disjnt-scene*[*intro*, *simp*]: *disjnt-scene m1 m2* $\Longrightarrow$ *comm-scene m1 m2*
by (*simp add*: *disjnt-scene-def comm-scene-def*)

lemma *disjnt-scene-comm*: *disjnt-scene m1 m2* $\longleftrightarrow$ *disjnt-scene m2 m1*
by (*simp add*: *disjnt-scene-def*) *metis*

lemma *disjnt-scene-sym*[*intro*]: *disjnt-scene m1 m2* $\Longrightarrow$ *disjnt-scene m2 m1*
by (*simp add*: *disjnt-scene-comm*)

lemma *disjnt-sceneD*: *disjnt-scene m1 m2* $\Longrightarrow$ *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*)
by (*simp add*: *disjnt-scene-def*)

lemma *disjnt-sceneD-left*: *is-scene m1* $\Longrightarrow$ *disjnt-scene m1 m2* $\Longrightarrow$ *m1* (*m2 a b*) *c* = *m1 b c*
by (*simp add*: *disjnt-scene-def is-scene-def*) *metis*

lemma *disjnt-sceneD-right*: *is-scene m2* $\Longrightarrow$ *disjnt-scene m1 m2* $\Longrightarrow$ *m2* (*m1 a b*) *c* = *m2 b c*
using *disjnt-sceneD-left*[*of m2 m1*, *OF* - *disjnt-scene-sym*] .

lemma *disjnt-scene-all-left*[*simp*]: *disjnt-scene* ($\lambda a\ b.\ a$) *m* $\longleftrightarrow$ *m* = ($\lambda a.\ id$)
by (*auto simp add*: *disjnt-scene-def fun-eq-iff*)

lemma *disjnt-scene-all-right*[*simp*]: *disjnt-scene m* ($\lambda a\ b.\ a$) $\longleftrightarrow$ *m* = ($\lambda a.\ id$)
by (*auto simp add*: *disjnt-scene-def fun-eq-iff*)

lemma *disjnt-scene-id-left*[*simp*]: *disjnt-scene* ($\lambda a.\ id$) *m*
by (*simp add*: *disjnt-scene-def*)

lemma *disjnt-scene-id-right*[*simp*]: *disjnt-scene m* ($\lambda a.\ id$)
by (*simp add*: *disjnt-scene-def*)

lemma *disjnt-scene-comp-left*:
comm-scene m1 m2 $\Longrightarrow$ *disjnt-scene m1 m* $\Longrightarrow$ *disjnt-scene m2 m* $\Longrightarrow$
disjnt-scene ($\lambda a.$ *m1 a* $\circ$ *m2 a*) *m*
by (*simp add*: *is-scene-def disjnt-scene-def*)

lemma *disjnt-scene-comp-right*:
disjnt-scene m m1 $\Longrightarrow$ *disjnt-scene m m2* $\Longrightarrow$ *comm-scene m1 m2* $\Longrightarrow$
disjnt-scene m ($\lambda a.$ *m1 a* $\circ$ *m2 a*)
using *disjnt-scene-comp-left*[*of m1 m2 m*] **by** (*simp add*: *disjnt-scene-comm*)

lemma *disjnt-scene-fold*:
list-all (*disjnt-scene m*) *ms* $\Longrightarrow$ *pairwise comm-scene* (*set ms*) $\Longrightarrow$
disjnt-scene ($\lambda a.$ *fold* ($\lambda m.$ *m a*) *ms*) *m*
by (*induction ms arbitrary*: *m*)
(*auto simp*: *pairwise-def intro*!: *disjnt-scene-comp-left comm-scene-fold*)

lemma *fold-disjnt-scene*:
list-all is-scene ms $\Longrightarrow$ *pairwise comm-scene* (*set ms*) $\Longrightarrow$
list-all ($\lambda m.$ *disjnt-scene m m'* $\vee$ *m* = *m'*) *ms* $\Longrightarrow$ *m'* $\in$ *set ms* $\Longrightarrow$
fold ($\lambda m.$ *m* (*m' a b*)) *ms c* = *m' a* (*fold* ($\lambda m.$ *m b*) *ms c*)
proof (*induction ms rule*: *rev-induct*)
case (*snoc m ms*)
show *?case*
proof *cases*
assume *m'* $\in$ *set ms*
with *snoc* **have** *eq*[*simp*]: *fold* ($\lambda m.$ *m* (*m' a b*)) *ms c* = *m' a* (*fold* ($\lambda m.$ *m b*) *ms c*)
and *m-m'*: *disjnt-scene m m'* $\vee$ *m* = *m'*
and *m*: *is-scene m* **and** *m'*: *is-scene m'*

```
    by (auto simp: list-all-iff pairwise-def)

   show ?thesis
    using m-m' m m'
    by (auto simp: disjnt-sceneD-left disjnt-sceneD
                   is-scene.right[OF m'] is-scene.left[OF m'])
  next
   assume m' ∉ set ms
   with snoc.prems have m'-ms: list-all (disjnt-scene m') ms
    and [simp]: list-all is-scene ms pairwise comm-scene (set ms)
    and [simp]: m = m' and m': is-scene m'
    by (auto simp: list-all-iff pairwise-def)

   have [simp]: fold (λm. m (m' a b)) ms c = fold (λm. m b) ms c
    by (rule disjnt-sceneD-left[OF is-scene-fold disjnt-scene-fold[OF m'-ms]]; simp)

   show ?thesis
    by (simp add: is-scene.right[OF m'] is-scene.left[OF m'])
  qed
qed simp

end


theory CompoundCTypes
  imports
    Vanilla32
    Padding
    Lens
begin

lemma simple-type-dest: ¬aggregate t ⟹ ∃ n sz align align' d. t = TypDesc align'
(TypScalar sz align d) n
  apply (cases t)
  subgoal for x1 st n
   apply (cases st)
    apply auto
   done
  done

definition empty-typ-info :: nat ⇒ typ-name ⇒ ('a,'b) typ-desc where
  empty-typ-info algn tn ≡ TypDesc algn (TypAggregate []) tn


primrec
  extend-ti :: 'a xtyp-info ⇒ 'a xtyp-info ⇒ nat ⇒ field-name ⇒ 'a field-desc ⇒ 'a
xtyp-info and
  extend-ti-struct :: 'a xtyp-info-struct ⇒ 'a xtyp-info ⇒ field-name ⇒ 'a field-desc
⇒ 'a xtyp-info-struct
```

where
et0: *extend-ti* (*TypDesc algn' st nm*) *t algn fn d* = *TypDesc* (*max algn'* (*max algn* (*align-td t*))) (*extend-ti-struct st t fn d*) *nm*

| *et1*: *extend-ti-struct* (*TypScalar n sz algn*) *t fn d* = *TypAggregate* [*DTuple t fn d*]
| *et2*: *extend-ti-struct* (*TypAggregate ts*) *t fn d* = *TypAggregate* (*ts*@[*DTuple t fn d*])

lemma *aggregate-empty-typ-info* [*simp*]:
aggregate (*empty-typ-info algn tn*)
by (*simp add*: *empty-typ-info-def*)

lemma *aggregate-extend-ti* [*simp*]:
aggregate (*extend-ti tag t algn f d*)
apply(*cases tag*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*, *auto*)
done
done

lemma *typ-name-extend-ti* [*simp*]: *typ-name* (*extend-ti T t algn fn d*) = *typ-name T*
by (*cases T*) *auto*

definition *update-desc* :: ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*a* ⇒ '*a*) ⇒ '*b field-desc* ⇒ '*a field-desc*
where
update-desc acc upd d ≡ (|*field-access* = (*field-access d*) ∘ *acc*,
field-update = λ*bs v*. *upd* (*field-update d bs* (*acc v*)) *v*, *field-sz* = *field-sz d* |)

lemma *update-desc-id*[*simp*]: *update-desc id* (λ*x* -. *x*) = *id*
by (*simp add*: *update-desc-def fun-eq-iff*)

term *map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*

definition *adjust-ti* :: '*b xtyp-info* ⇒ ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*a* ⇒ '*a*) ⇒ '*a xtyp-info*
where
adjust-ti t acc upd ≡ *map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*

lemma *adjust-ti-adjust-ti*:
adjust-ti (*adjust-ti t g s*) *g' s'* =
adjust-ti t (*g* ∘ *g'*) (λ*v x*. *s'* (*s v* (*g' x*)) *x*)
by (*simp add*: *adjust-ti-def map-td-map update-desc-def*[*abs-def*] *comp-def*)

lemma *typ-desc-size-update-ti* [*simp*]:
(*size-td* (*adjust-ti t acc upd*) = *size-td t*)
by (*simp add*: *adjust-ti-def*)

lemma *export-tag-adjust-ti*[*rule-format*]:
  ∀ *bs. fg-cons acc upd* ⟶ *wf-fd t* ⟶
    export-uinfo (*adjust-ti t acc upd*) = *export-uinfo t*
  ∀ *bs. fg-cons acc upd* ⟶ *wf-fd-struct st* ⟶
    map-td-struct field-norm (λ-. ()) (*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *st*) =
    map-td-struct field-norm (λ-. ()) *st*
  ∀ *bs. fg-cons acc upd* ⟶ *wf-fd-list ts* ⟶
     map-td-list field-norm (λ-. ()) (*map-td-list* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *ts*) =
    map-td-list field-norm (λ-. ()) *ts*
  ∀ *bs. fg-cons acc upd* ⟶ *wf-fd-tuple x* ⟶
    map-td-tuple field-norm (λ-. ()) (*map-td-tuple* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *x*) =
    map-td-tuple field-norm (λ-. ()) *x*
  unfolding *adjust-ti-def*
  by (*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp simp*: *export-uinfo-def*›)
    (*fastforce simp*: *update-desc-def field-norm-def fg-cons-def fd-cons-struct-def*
                fd-cons-access-update-def fd-cons-desc-def)

definition (**in** *c-type*) *ti-typ-combine* ::
  'a itself ⇒ (*'b* ⇒ *'a*) ⇒ (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *nat* ⇒ *field-name* ⇒ *'b xtyp-info* ⇒ *'b xtyp-info*
  where
  ti-typ-combine t-b acc upd algn fn tag ≡
    let ft = *adjust-ti* (*typ-info-t TYPE*(*'a*)) *acc upd*
    in extend-ti tag ft algn fn (|*field-access* = *xto-bytes o acc*, *field-update* = *upd o xfrom-bytes*, *field-sz* = *size-of TYPE*(*'a*)|)

primrec
  padding-fields :: (*'a*,*'b*) *typ-desc* ⇒ *field-name list* **and**
  padding-fields-struct :: (*'a*,*'b*) *typ-struct* ⇒ *field-name list*
where
  pf0: *padding-fields* (*TypDesc algn st tn*) = *padding-fields-struct st*

| *pf1*: *padding-fields-struct* (*TypScalar n algn d*) = []
| *pf2*: *padding-fields-struct* (*TypAggregate xs*) = *filter* (λ*x. hd x* = *CHR ''!''*) (*map dt-snd xs*)

primrec
  non-padding-fields :: (*'a*,*'b*) *typ-desc* ⇒ *field-name list* **and**
  non-padding-fields-struct :: (*'a*,*'b*) *typ-struct* ⇒ *field-name list*
where
  npf0: *non-padding-fields* (*TypDesc algn st tn*) = *non-padding-fields-struct st*

| *npf1*: *non-padding-fields-struct* (*TypScalar n algn d*) = []
| *npf2*: *non-padding-fields-struct* (*TypAggregate xs*) = *filter* (λ*x. hd x* ≠ *CHR ''!''*)

(*map dt-snd xs*)

definition *field-names-list* :: ('*a*,'*b*) *typ-desc* ⇒ *field-name list* **where**
field-names-list tag ≡ *non-padding-fields tag* @ *padding-fields tag*

definition *ti-pad-combine* :: *nat* ⇒ '*a xtyp-info* ⇒ '*a xtyp-info* **where**
ti-pad-combine n tag ≡
let
fn = *foldl* (@) ''!*pad-*'' (*field-names-list tag*);
td = *padding-desc n*;
nf = *TypDesc 0* (*TypScalar n 0 td*) ''!*pad-typ*''
in extend-ti tag nf 0 fn td

lemma *aggregate-ti-pad-combine* [*simp*]:
aggregate (*ti-pad-combine n tag*)
by (*simp add*: *ti-pad-combine-def Let-def*)

definition (**in** *c-type*) *ti-typ-pad-combine* ::
'*a itself* ⇒ ('*b* ⇒ '*a*) ⇒ ('*a* ⇒ '*b* ⇒ '*b*) ⇒ *nat* ⇒ *field-name* ⇒ '*b xtyp-info* ⇒ '*b xtyp-info*
where
ti-typ-pad-combine t acc upd algn fn tag ≡
let
pad = *padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*a*))) (*size-td tag*);
ntag = *if 0* < *pad then ti-pad-combine pad tag else tag*
in
ti-typ-combine t acc upd algn fn ntag

definition *map-align f t* = (*case t of TypDesc algn st n* ⇒ *TypDesc* (*f algn*) *st n*)
lemma *map-align-simp* [*simp*]: *map-align f* (*TypDesc algn st n*) = *TypDesc* (*f algn*) *st n*
by (*simp add*: *map-align-def*)

definition *final-pad* :: *nat* ⇒ '*a xtyp-info* ⇒ '*a xtyp-info* **where**
final-pad algn tag ≡
let n = *padup* (2^(*max algn* (*align-td tag*))) (*size-td tag*)
in map-align (*max algn*) (*if 0* < *n then ti-pad-combine n tag else tag*)

lemma *field-names-list-empty-typ-info* [*simp*]:
set (*field-names-list* (*empty-typ-info algn tn*)) = {}
by (*simp add*: *empty-typ-info-def field-names-list-def*)

lemma *field-names-list-extend-ti* [*simp*]:
set (*field-names-list* (*extend-ti tag t algn fn d*)) = *set* (*field-names-list tag*) ∪ {*fn*}
unfolding *field-names-list-def*

apply(*cases tag*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*; *simp*)
done
done

lemma (**in** *c-type*) *field-names-list-ti-typ-combine* [*simp*]:
set (*field-names-list* (*ti-typ-combine* (*t*::'*a itself*) *f f-upd algn fn tag*))
= *set* (*field-names-list tag*) ∪ {*fn*}
by (*clarsimp simp*: *ti-typ-combine-def Let-def*)

lemma *field-names-list-ti-pad-combine* [*simp*]:
set (*field-names-list* (*ti-pad-combine n tag*))
= *set* (*field-names-list tag*) ∪ {*foldl* (@) ''!*pad-*'' (*field-names-list tag*)}
by (*clarsimp simp*: *ti-pad-combine-def Let-def*)

— matches on padding
lemma *hd-string-hd-fold-eq* [*simp*]:
⟦ *s* ≠ []; *hd s* = *CHR* ''!'' ⟧ ⟹ *hd* (*foldl* (@) *s xs*) = *CHR* ''!''
by (*induct xs arbitrary*: *s*; *clarsimp*)

lemma *field-names-list-ti-typ-pad-combine* [*simp*]:
hd x ≠ *CHR* ''!'' ⟹
x ∈ *set* (*field-names-list* (*ti-typ-pad-combine align t-b f-ab f-upd-ab fn tag*))
= (*x* ∈ *set* (*field-names-list tag*) ∪ {*fn*})
by (*auto simp*: *ti-typ-pad-combine-def Let-def*)

lemma *wf-desc-empty-typ-info* [*simp*]:
wf-desc (*empty-typ-info algn tn*)
by (*simp add*: *empty-typ-info-def*)

lemma *wf-desc-extend-ti*:
⟦ *wf-desc tag*; *wf-desc t*; *f* ∉ *set* (*field-names-list tag*) ⟧ ⟹
wf-desc (*extend-ti tag t algn f d*)
unfolding *field-names-list-def*
apply(*cases tag*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*; *clarsimp*)
done
done

lemma *foldl-append-length*:
length (*foldl* (@) *s xs*) ≥ *length s*
apply(*induct xs arbitrary*: *s*, *clarsimp*)
subgoal for *a list s*
apply(*drule meta-spec* [**where** *x*=*s*@*a*])
apply *clarsimp*
done
done

lemma *foldl-append-nmem*:
s ≠ [] ⟹ *foldl* (@) *s xs* ∉ *set xs*
apply(*induct xs arbitrary*: *s*, *clarsimp*)
subgoal for *a list s*
apply(*drule meta-spec* [**where** *x*=*s*@*a*])
apply *clarsimp*
by (*metis add-le-same-cancel2 foldl-append-length le-zero-eq length-0-conv length-append*)
done

lemma *wf-desc-ti-pad-combine*:
wf-desc tag ⟹ *wf-desc* (*ti-pad-combine n tag*)
apply(*clarsimp simp*: *ti-pad-combine-def Let-def*)
apply(*erule wf-desc-extend-ti*)
apply *simp*
apply(*rule foldl-append-nmem*, *simp*)
done

lemma *wf-desc-adjust-ti* [*simp*]:
wf-desc (*adjust-ti t f g*) = *wf-desc* (*t*::′*a xtyp-info*)
by (*simp add*: *adjust-ti-def wf-desc-map*)

lemma (**in** *wf-type*) *wf-desc-ti-typ-combine*:
⟦ *wf-desc tag*; *fn* ∉ *set* (*field-names-list tag*) ⟧ ⟹
wf-desc (*ti-typ-combine* (*t-b*::′*a itself*) *acc upd-ab algn fn tag*)
by (*fastforce simp*: *ti-typ-combine-def Let-def elim*: *wf-desc-extend-ti*)

lemma (**in** *wf-type*) *wf-desc-ti-typ-pad-combine*:
⟦ *wf-desc tag*; *fn* ∉ *set* (*field-names-list tag*); *hd fn* ≠ *CHR* ′′!′′ ⟧ ⟹
wf-desc (*ti-typ-pad-combine* (*t-b*::′*a itself*) *acc upd algn fn tag*)
unfolding *ti-typ-pad-combine-def Let-def*
by (*auto intro*!: *wf-desc-ti-typ-combine wf-desc-ti-pad-combine*)

lemma *wf-desc-map-align*: *wf-desc tag* ⟹ *wf-desc* (*map-align f tag*)
by (*cases tag*) (*simp*)

lemma *wf-desc-final-pad*:
wf-desc tag ⟹ *wf-desc* (*final-pad algn tag*)
by (*auto simp*: *final-pad-def Let-def wf-desc-map-align wf-desc-ti-pad-combine*)

lemma *wf-size-desc-extend-ti*:
⟦ *wf-size-desc tag*; *wf-size-desc t* ⟧ ⟹ *wf-size-desc* (*extend-ti tag t algn fn d*)
apply(*cases tag*)
subgoal for *x1 typ-struct list*
apply(*cases typ-struct*, *auto*)
done
done

lemma *wf-size-desc-ti-pad-combine*:

⟦ *wf-size-desc tag*; *0 < n* ⟧ ⟹ *wf-size-desc* (*ti-pad-combine n tag*)
by (*fastforce simp*: *ti-pad-combine-def Let-def elim*: *wf-size-desc-extend-ti*)

lemma *wf-size-desc-adjust-ti*:
wf-size-desc (*adjust-ti t f g*) = *wf-size-desc* (*t*::'*a xtyp-info*)
by (*simp add*: *adjust-ti-def wf-size-desc-map*)

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-combine*:
wf-size-desc tag ⟹ *wf-size-desc* (*ti-typ-combine* (*t-b*::'*a itself*) *acc upd-ab algn fn tag*)
by (*fastforce simp*: *wf-size-desc-adjust-ti ti-typ-combine-def Let-def elim*: *wf-size-desc-extend-ti*)

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-pad-combine*:
wf-size-desc tag ⟹
wf-size-desc (*ti-typ-pad-combine* (*t-b*::'*a itself*) *acc upd algn fn tag*)
by (*auto simp*: *ti-typ-pad-combine-def Let-def*
intro: *wf-size-desc-ti-typ-combine*
elim: *wf-size-desc-ti-pad-combine*)

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-combine-empty* [*simp*]:
wf-size-desc (*ti-typ-combine* (*t-b*::'*a itself*) *acc upd algn fn* (*empty-typ-info algn'* *tn*))
by (*clarsimp simp*: *ti-typ-combine-def Let-def empty-typ-info-def wf-size-desc-adjust-ti*)

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-pad-combine-empty* [*simp*]:
wf-size-desc (*ti-typ-pad-combine* (*t-b*::'*a itself*) *acc upd algn fn*
(*empty-typ-info algn'* *tn*))
by (*clarsimp simp*: *ti-typ-pad-combine-def Let-def ti-typ-combine-def empty-typ-info-def*
ti-pad-combine-def wf-size-desc-adjust-ti)

lemma *wf-size-desc-msp-align*:
wf-size-desc tag ⟹ *wf-size-desc* (*map-align f tag*)
by (*cases tag*) (*simp add*: *wf-size-desc-def*)

lemma *wf-size-desc-final-pad*:
wf-size-desc tag ⟹ *wf-size-desc* (*final-pad algn tag*)
by (*fastforce simp*: *final-pad-def Let-def wf-size-desc-msp-align wf-size-desc-ti-pad-combine*)

lemma *wf-fdp-set-comp-simp* [*simp*]:
wf-fdp {(*a*, *n* # *b*) |*a b*. (*a*, *b*) ∈ *tf-set t*} = *wf-fdp* (*tf-set t*)
unfolding *wf-fdp-def* **by** *fastforce*

lemma *lf-set-adjust-ti'*:
∀ *d fn*. *d* ∈ *lf-set* (*map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*) *fn* ⟶
(∃ *y*. *lf-fd d*=*update-desc acc upd* (*lf-fd y*) ∧ *lf-sz d*=*lf-sz y* ∧ *lf-fn d*=*lf-fn y* ∧ *y* ∈ *lf-set t fn*)
∀ *d fn*. *d* ∈ *lf-set-struct* (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *st*) *fn* ⟶

(∃ y. lf-fd d=update-desc acc upd (lf-fd y) ∧ lf-sz d=lf-sz y ∧ lf-fn d=lf-fn y ∧ y ∈ lf-set-struct st fn)
∀ d fn. d ∈ lf-set-list (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) ts) fn ⟶
(∃ y. lf-fd d=update-desc acc upd (lf-fd y) ∧ lf-sz d=lf-sz y ∧ lf-fn d=lf-fn y ∧ y ∈ lf-set-list ts fn)
∀ d fn. d ∈ lf-set-tuple (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd) x) fn ⟶
(∃ y. lf-fd d=update-desc acc upd (lf-fd y) ∧ lf-sz d=lf-sz y ∧ lf-fn d=lf-fn y ∧ y ∈ lf-set-tuple x fn)
unfolding *update-desc-def*
by (*induct t* **and** *st* **and** *ts* **and** *x*) *fastforce+*

lemma *lf-set-adjust-ti*:
⟦ d ∈ lf-set (adjust-ti t acc upd) fn; ⋀y. upd (acc y) y = y ⟧ ⟹
(∃ y. lf-fd d=update-desc acc upd (lf-fd y) ∧ lf-sz d=lf-sz y ∧ lf-fn d=lf-fn y ∧ y ∈ lf-set t fn)
by (*simp add*: *lf-set-adjust-ti' adjust-ti-def*)

lemma *fd-cons-struct-id-simp* [*simp*]:
fd-cons-struct (*TypScalar n algn* (|*field-access* = λv. *id*, *field-update* = λbs. *id*, *field-sz* = *m*|))
by (*auto simp*: *fd-cons-struct-def fd-cons-double-update-def*
fd-cons-update-access-def fd-cons-access-update-def fd-cons-length-def
fd-cons-update-normalise-def fd-cons-desc-def)

lemma *field-desc-adjust-ti*:
fg-cons acc upd ⟶
field-desc (*adjust-ti* (*t*::'*a xtyp-info*) *acc upd*) =
update-desc acc upd (*field-desc t*)
fg-cons acc upd ⟶
field-desc-struct (*map-td-struct* (λn *algn. update-desc acc upd*) (*update-desc acc upd*) *st*) =
update-desc acc upd (*field-desc-struct st*)
fg-cons acc upd ⟶
field-desc-list (*map-td-list* (λn *algn. update-desc acc upd*) (*update-desc acc upd*) *ts*) =
update-desc acc upd (*field-desc-list ts*)
fg-cons acc upd ⟶
field-desc-tuple (*map-td-tuple* (λn *algn. update-desc acc upd*) (*update-desc acc upd*) *x*) =
update-desc acc upd (*field-desc-tuple x*)
unfolding *adjust-ti-def*
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*fastforce simp*: *fg-cons-def update-desc-def update-ti-t-struct-t*)+

lemma *update-ti-t-adjust-ti*:
fg-cons acc upd ⟹ *update-ti-t* (*adjust-ti t acc upd*) *bs v* = *upd* (*update-ti-t t bs*

(*acc v*)) *v*
using *field-desc-adjust-ti*(*1*) [*of acc upd t*]
by (*clarsimp simp*: *update-desc-def*)

declare *field-desc-def* [*simp del*]

lemma (**in** *c-type*) *aggregate-ti-typ-combine* [*simp*]:
aggregate (*ti-typ-combine* (*t-b*::'*a itself*) *acc upd algn fn tag*)
by (*simp add*: *ti-typ-combine-def Let-def*)

lemma (**in** *c-type*) *aggregate-ti-typ-pad-combine* [*simp*]:
aggregate (*ti-typ-pad-combine* (*t-b*::'*a itself*) *acc upd algn fn tag*)
by (*simp add*: *ti-typ-pad-combine-def Let-def*)

lemma *align-of-empty-typ-info* [*simp*]:
align-td-wo-align (*empty-typ-info algn tn*) = *0*
by (*simp add*: *empty-typ-info-def*)

lemma *align-of-empty-typ-info'* [*simp*]:
align-td (*empty-typ-info algn tn*) = *algn*
by (*simp add*: *empty-typ-info-def*)

lemma *align-of-tag-list* [*simp*]:
align-td-wo-align-list (*xs* @ [*DTuple t fn d*]) = *max* (*align-td-wo-align-list xs*) (*align-td-wo-align t*)
by (*induct xs*) *auto*

lemma *align-of-tag-list'* [*simp*]:
align-td-list (*xs* @ [*DTuple t fn d*]) = *max* (*align-td-list xs*) (*align-td t*)
by (*induct xs*) *auto*

lemma *align-of-extend-ti* [*simp*]:
aggregate ti ⟹ *align-td-wo-align* (*extend-ti ti t algn fn d*) = *max* (*align-td-wo-align ti*) (*align-td-wo-align t*)
apply (*cases ti*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*; *clarsimp*)
done
done

lemma *align-of-extend-ti'* [*simp*]:
aggregate ti ⟹ *align-td* (*extend-ti ti t algn fn d*) = *max* (*align-td ti*) (*max algn* (*align-td t*))
apply (*cases ti*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*; *clarsimp*)
done
done

lemma *align-of-adjust-ti* [*simp*]:
align-td-wo-align (*adjust-ti t acc upd*) = *align-td-wo-align* (*t*::'*a xtyp-info*)
by (*simp add*: *adjust-ti-def*)

lemma *align-of-adjust-ti'* [*simp*]:
align-td (*adjust-ti t acc upd*) = *align-td* (*t*::'*a xtyp-info*)
by (*simp add*: *adjust-ti-def*)

lemma (**in** *c-type*) *align-of-ti-typ-combine* [*simp*]:
aggregate ti ⟹
align-td-wo-align (*ti-typ-combine* (*t*::'*a itself*) *acc upd algn fn ti*) =
max (*align-td-wo-align ti*) (*align-td-wo-align* (*typ-info-t* (*TYPE*('*a*))))
by (*clarsimp simp*: *ti-typ-combine-def Let-def align-of-def*)

lemma (**in** *c-type*) *align-of-ti-typ-combine'* [*simp*]:
aggregate ti ⟹
align-td (*ti-typ-combine* (*t*::'*a itself*) *acc upd algn fn ti*) =
max (*align-td ti*) (*max algn* (*align-td* (*typ-info-t TYPE*('*a*))))
by (*clarsimp simp*: *ti-typ-combine-def Let-def align-of-def*)

lemma *align-of-ti-pad-combine* [*simp*]:
aggregate ti ⟹ *align-td-wo-align* (*ti-pad-combine n ti*) = (*align-td-wo-align ti*)
by (*clarsimp simp*: *ti-pad-combine-def Let-def max-def*)

lemma *align-of-ti-pad-combine'* [*simp*]:
aggregate ti ⟹ *align-td* (*ti-pad-combine n ti*) = (*align-td ti*)
by (*clarsimp simp*: *ti-pad-combine-def Let-def max-def*)

lemma *max-2-exp*: *max* ((*2*::*nat*) ^ *a*) (*2* ^ *b*) = *2* ^ (*max a b*)
by (*simp add*: *max-def*)

lemma *align-td-wo-align-map-align*: *align-td-wo-align* (*map-align f t*) = *align-td-wo-align t*
by (*cases t*) *simp*

lemma *align-td-wo-align-final-pad*:
aggregate ti ⟹
align-td-wo-align (*final-pad algn ti*) = (*align-td-wo-align ti*)
by (*simp add*: *final-pad-def Let-def padup-def align-td-wo-align-map-align*)

lemma *align-td-map-align* [*simp*]: *align-td* (*map-align f t*) = *f* (*align-td t*)
by (*cases t*) *simp*

lemma *align-of-final-pad*:
aggregate ti ⟹
align-td (*final-pad algn ti*) = *max algn* (*align-td ti*)
by (*simp add*: *final-pad-def Let-def padup-def align-td-map-align*)

lemma (**in** *c-type*) *align-td-wo-align-ti-typ-pad-combine* [*simp*]:
aggregate ti $\Longrightarrow$
align-td-wo-align (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn ti*) =
max (*align-td-wo-align ti*) (*align-td-wo-align* (*typ-info-t TYPE*('*a*)))
by (*clarsimp simp*: *ti-typ-pad-combine-def Let-def*)

lemma (**in** *c-type*) *align-td-ti-typ-pad-combine* [*simp*]:
aggregate ti $\Longrightarrow$
align-td (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn ti*) =
max (*align-td ti*) (*max algn* (*align-td* (*typ-info-t TYPE*('*a*))))
by (*clarsimp simp*: *ti-typ-pad-combine-def Let-def*)

definition *fu-s-comm-set* :: (*byte list* $\Rightarrow$ '*a* $\Rightarrow$ '*a*) *set* $\Rightarrow$ (*byte list* $\Rightarrow$ '*a* $\Rightarrow$ '*a*) *set* $\Rightarrow$ *bool*
where
fu-s-comm-set X Y $\equiv \forall x\ y.\ x \in X \wedge y \in Y \longrightarrow (\forall v\ bs\ bs'.\ x\ bs\ (y\ bs'\ v) = y\ bs'\ (x\ bs\ v))$

lemma *fc-empty-ti* [*simp*]:
fu-commutes (*update-ti-t* (*empty-typ-info algn tn*)) *f*
by (*auto simp*: *fu-commutes-def empty-typ-info-def*)

lemma *fc-extend-ti*:
⟦ *fu-commutes* (*update-ti-t s*) *h*; *fu-commutes* (*update-ti-t t*) *h* ⟧
$\Longrightarrow$ *fu-commutes* (*update-ti-t* (*extend-ti s t algn fn d*)) *h*
apply(*cases s*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*, *auto simp*: *fu-commutes-def*)
done
done

lemma *fc-update-ti*:
⟦ *fu-commutes* (*update-ti-t ti*) *h*; *fg-cons acc upd*;
$\forall v\ bs\ bs'.$ *upd bs* (*h bs'* *v*) = *h bs'* (*upd bs v*); $\forall bs\ v.$ *acc* (*h bs v*) = *acc v* ⟧
$\Longrightarrow$ *fu-commutes* (*update-ti-t* (*adjust-ti t acc upd*)) *h*
by (*auto simp*: *fu-commutes-def update-ti-t-adjust-ti*)

lemma (**in** *c-type*) *fc-ti-typ-combine*:
⟦ *fu-commutes* (*update-ti-t ti*) *h*; *fg-cons acc upd*;
$\forall v\ bs\ bs'.$ *upd bs* (*h bs'* *v*) = *h bs'* (*upd bs v*); $\forall bs\ v.$ *acc* (*h bs v*) = *acc v* ⟧
$\Longrightarrow$ *fu-commutes* (*update-ti-t* (*ti-typ-combine* (*t*::'*a itself*) *acc upd algn fn ti*)) *h*
apply(*clarsimp simp*: *ti-typ-combine-def Let-def*)
apply(*rule fc-extend-ti*, *assumption*)
apply(*rule fc-update-ti*; *simp*)
done

lemma *fc-ti-pad-combine*:
fu-commutes (*update-ti-t ti*) *f* $\Longrightarrow$ *fu-commutes* (*update-ti-t* (*ti-pad-combine n ti*)) *f*

apply(*clarsimp simp*: *ti-pad-combine-def Let-def*)
apply(*rule fc-extend-ti*, *assumption*)
apply(*auto simp*: *fu-commutes-def padding-desc-def*)
done

lemma (**in** *c-type*) *fc-ti-typ-pad-combine*:
⟦ *fu-commutes* (*update-ti-t ti*) *h*; *fg-cons acc upd*;
∀ *v bs bs'*. *upd bs* (*h bs' v*) = *h bs'* (*upd bs v*); ∀ *bs v*. *acc* (*h bs v*) = *acc v* ⟧
⟹ *fu-commutes* (*update-ti-t* (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn ti*)) *h*
apply(*clarsimp simp*: *ti-typ-pad-combine-def Let-def*)
apply(*rule conjI*; *clarsimp*)
apply(*rule fc-ti-typ-combine*; *assumption?*)
apply(*erule fc-ti-pad-combine*)
apply(*erule* (*3*) *fc-ti-typ-combine*)
done

definition *fu-eq-mask* :: '*a xtyp-info* ⇒ ('*a* ⇒ '*a*) ⇒ *bool* **where**
fu-eq-mask ti f ≡
∀ *bs v v'*. *length bs* = *size-td ti* ⟶ *update-ti-t ti bs* (*f v*) = *update-ti-t ti bs* (*f v'*)

lemma *fu-eq-mask*:
⟦ *length bs* = *size-td ti*; *fu-eq-mask ti id* ⟧ ⟹
update-ti-t ti bs v = *update-ti-t ti bs w*
by (*clarsimp simp*: *fu-eq-mask-def update-ti-t-def*)

lemma *fu-eq-mask-ti-pad-combine*:
⟦ *fu-eq-mask ti f*; *aggregate ti* ⟧ ⟹ *fu-eq-mask* (*ti-pad-combine n ti*) *f*
unfolding *ti-pad-combine-def Let-def*
apply(*cases ti*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*, *auto simp*: *fu-eq-mask-def update-ti-list-t-def padding-desc-def*)
done
done

lemma *fu-eq-mask-map-align*: *fu-eq-mask t f* ⟹ *fu-eq-mask* (*map-align g t*) *f*
by (*cases t*) (*auto simp add*: *fu-eq-mask-def*)

lemma *fu-eq-mask-final-pad*:
⟦ *fu-eq-mask ti f*; *aggregate ti* ⟧ ⟹ *fu-eq-mask* (*final-pad algn ti*) *f*
by(*auto simp*: *final-pad-def Let-def fu-eq-mask-map-align fu-eq-mask-ti-pad-combine*)

definition *upd-local* :: ('*b* ⇒ '*a* ⇒ '*a*) ⇒ *bool* **where**
upd-local g ≡ ∀ *j k v v'*. *g k v* = *g k v'* ⟶ *g j v* = *g j v'*

lemma *fg-cons-upd-local*:
fg-cons f g ⟹ *upd-local g*
apply(*clarsimp simp*: *fg-cons-def upd-local-def*)

subgoal for *j k v v′*
apply(*drule arg-cong* [**where** *f*=*g j*])
apply *simp*
done
done

lemma (**in** *mem-type*) *fu-eq-mask-ti-typ-combine*:
⟦ *fu-eq-mask ti* (λ*v*. (*upd* (*acc undefined*) (*h v*))); *fg-cons acc upd*;
fu-commutes (*update-ti-t ti*) *upd*; *aggregate ti* ⟧ ⟹
fu-eq-mask (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) *h*
apply(*frule fg-cons-upd-local*)
apply(*clarsimp simp*: *ti-typ-combine-def Let-def*)
apply(*cases ti*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*; *clarsimp*)
subgoal for *xs′*
apply(*clarsimp simp*: *fu-eq-mask-def update-ti-t-adjust-ti*)
apply(*clarsimp simp*: *update-ti-list-t-def size-of-def*)
apply(*subst upd* [**where** *w*=*acc undefined*])
apply(*simp add*: *size-of-def*)
apply(*subst upd* [**where** *w*=*acc undefined* **and** *v*=*acc* (*h v′*) **for** *v′*])
apply(*simp add*: *size-of-def*)
apply (*smt* (*verit*, *ccfv-threshold*) *fu-commutes-def length-take min-ll upd-local-def*)
done
done
done

lemma (**in** *mem-type*) *fu-eq-mask-ti-typ-pad-combine*:
⟦ *fu-eq-mask ti* (λ*v*. (*upd* (*acc undefined*) (*h v*))); *fg-cons acc upd*;
fu-commutes (*update-ti-t ti*) *upd*; *aggregate ti* ⟧ ⟹
fu-eq-mask (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*) *h*
by (*fastforce simp*: *ti-typ-pad-combine-def Let-def*
intro: *fu-eq-mask-ti-typ-combine fu-eq-mask-ti-pad-combine fc-ti-pad-combine*)

lemma *fu-eq-mask-empty-typ-info-g*:
∃ *k*. ∀ *v*. *f v* = *k* ⟹ *fu-eq-mask t f*
by (*auto simp*: *fu-eq-mask-def*)

lemma *fu-eq-mask-empty-typ-info*:
∀ *v*. *f v* = *undefined* ⟹ *fu-eq-mask t f*
by (*auto simp*: *fu-eq-mask-def*)

lemma *size-td-extend-ti*:
aggregate s ⟹ *size-td* (*extend-ti s t algn fn d*) = *size-td s* + *size-td t*
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*; *simp*)
done
done

lemma *size-td-ti-pad-combine*:
aggregate ti $\Longrightarrow$ *size-td* (*ti-pad-combine n ti*) = *n* + *size-td ti*
unfolding *ti-pad-combine-def Let-def* **by** (*simp add*: *size-td-extend-ti*)

lemma *size-td-map-align* [*simp*]: *size-td* (*map-align f ti*) = *size-td ti*
by (*cases ti*) *auto*

lemma *align-of-dvd-size-of-final-pad* [*simp*]:
aggregate ti $\Longrightarrow$ 2^*align-td* (*final-pad algn ti*) *dvd size-td* (*final-pad algn ti*)
unfolding *final-pad-def Let-def*
apply (*cases ti*)
apply (*auto simp*: *size-td-ti-pad-combine ac-simps padup-dvd power-le-dvd intro*: *dvd-padup-add*)
done

lemma *size-td-lt-ti-pad-combine*:
aggregate t $\Longrightarrow$ *size-td* (*ti-pad-combine n t*) = *size-td t* + *n*
by (*metis add.commute size-td-ti-pad-combine*)

lemma (**in** *c-type*) *size-td-lt-ti-typ-combine*:
aggregate ti $\Longrightarrow$
size-td (*ti-typ-combine* (*t*::'*a itself*) *f g algn fn ti*) =
size-td ti + *size-td* (*typ-info-t TYPE*('*a*))
by (*clarsimp simp*: *ti-typ-combine-def Let-def size-td-extend-ti*)

lemma (**in** *c-type*) *size-td-lt-ti-typ-pad-combine*:
aggregate ti $\Longrightarrow$
size-td (*ti-typ-pad-combine* (*t*::'*a itself*) *f g algn fn ti*) =
(*let k* = *size-td ti in*
k + *size-td* (*typ-info-t TYPE*('*a*)) + *padup* (2^(*max algn* (*align-td* (*typ-info-t TYPE*('*a*))))) *k*)
unfolding *ti-typ-pad-combine-def Let-def*
by (*auto simp*: *size-td-lt-ti-typ-combine size-td-ti-pad-combine align-of-def max-2-exp*)

lemma *size-td-lt-final-pad*:
aggregate tag $\Longrightarrow$
size-td (*final-pad align tag*) = (*let k*=*size-td tag in k* + *padup* (2^(*max align* (*align-td tag*))) *k*)
by (*auto simp*: *final-pad-def Let-def size-td-ti-pad-combine*)

lemma *size-td-empty-typ-info* [*simp*]:
size-td (*empty-typ-info algn tn*) = *0*
by (*clarsimp simp*: *empty-typ-info-def*)

lemma *wf-lf-empty-typ-info* [*simp*]:
wf-lf {}
by (*auto simp*: *wf-lf-def empty-typ-info-def*)

lemma *lf-fn-disj-fn*:
fn ∉ *set* (*field-names-list* (*TypDesc align* (*TypAggregate xs*) *tn*)) ⟹
lf-fn ‘ *lf-set-list xs* [] ∩ *lf-fn* ‘ *lf-set t* [*fn*] = {}
apply(*induct xs arbitrary*: *fn t tn*, *clarsimp*)
subgoal for *a list fn t tn*
apply(*cases a*, *clarsimp*)
apply(*drule meta-spec* [**where** *x*=*fn*])
apply(*drule meta-spec* [**where** *x*=*t*])
apply(*drule meta-spec* [**where** *x*=*tn*])
apply(*drule meta-mp*, *fastforce simp*: *field-names-list-def split*: *if-split-asm*)
apply(*safe*)
apply(*fastforce dest*!: *lf-set-fn simp*: *field-names-list-def prefix-def less-eq-list-def split*: *if-split-asm*)
by *force*
done

lemma *wf-lf-extend-ti*:
⟦ *wf-lf* (*lf-set t* []); *wf-lf* (*lf-set ti* []); *wf-desc t*; *fn* ∉ *set* (*field-names-list ti*);
ti-ind (*lf-set ti* []) (*lf-set t* []) ⟧ ⟹
wf-lf (*lf-set* (*extend-ti ti t align fn d*) [])
apply(*cases ti*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*; *clarsimp*)
apply(*subst wf-lf-fn*; *simp*)
apply(*subst wf-lf-list*, *erule lf-fn-disj-fn*)
apply(*subst ti-ind-sym2*)
apply(*subst ti-ind-fn*)
apply(*subst ti-ind-sym2*)
apply *clarsimp*
apply(*subst wf-lf-fn*; *simp*)
done
done

lemma *wf-lf-ti-pad-combine*:
wf-lf (*lf-set ti* []) ⟹ *wf-lf* (*lf-set* (*ti-pad-combine n ti*) [])
apply(*clarsimp simp*: *ti-pad-combine-def Let-def padding-desc-def*)
apply(*rule wf-lf-extend-ti*)
apply(*clarsimp simp*: *wf-lf-def fd-cons-desc-def fd-cons-double-update-def fd-cons-update-access-def fd-cons-access-update-def fd-cons-length-def*)
apply *assumption*
apply(*clarsimp*)
apply(*rule foldl-append-nmem*)
apply *clarsimp*
apply(*clarsimp simp*: *ti-ind-def fu-commutes-def fa-fu-ind-def*)
done

lemma *lf-set-map-align* [*simp*]: *lf-set* (*map-align f ti*) = *lf-set ti*

by (*cases ti*) *auto*

lemma *wf-lf-final-pad*:
wf-lf (*lf-set ti* []) ⟹ *wf-lf* (*lf-set* (*final-pad algn ti*) [])
by (*auto simp*: *final-pad-def Let-def elim*: *wf-lf-ti-pad-combine*)

lemma *wf-lf-adjust-ti*:
⟦ *wf-lf* (*lf-set t* []); ⋀*v*. *g* (*f v*) *v* = *v*;
⋀*bs bs'* *v*. *g bs* (*g bs'* *v*) = *g bs v*; ⋀*bs v*. *f* (*g bs v*) = *bs* ⟧
⟹ *wf-lf* (*lf-set* (*adjust-ti t f g*) [])
apply(*clarsimp simp*: *wf-lf-def*)
apply(*drule lf-set-adjust-ti*; *clarsimp*)
apply(*rule conjI*)
apply(*fastforce simp*: *fd-cons-desc-def fd-cons-double-update-def update-desc-def*
fd-cons-update-access-def fd-cons-access-update-def
fd-cons-length-def)
apply(*fastforce simp*: *fu-commutes-def update-desc-def fa-fu-ind-def dest*!: *lf-set-adjust-ti*)
done

lemma *ti-ind-empty-typ-info* [*simp*]:
ti-ind (*lf-set* (*empty-typ-info algn tn*) []) (*lf-set* (*adjust-ti k f g*) [])
by (*clarsimp simp*: *ti-ind-def empty-typ-info-def*)

lemma *ti-ind-extend-ti*:
⟦ *ti-ind* (*lf-set t* []) (*lf-set* (*adjust-ti k acc upd*) []);
ti-ind (*lf-set ti* []) (*lf-set* (*adjust-ti k acc upd*) []) ⟧
⟹ *ti-ind* (*lf-set* (*extend-ti ti t algn n d*) []) (*lf-set* (*adjust-ti k acc upd*) [])
apply(*cases ti*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*; *clarsimp*, *subst ti-ind-fn*, *simp*)
done
done

lemma *ti-ind-ti-pad-combine*:
ti-ind (*lf-set ti* []) (*lf-set* (*adjust-ti k acc upd*) []) ⟹
ti-ind (*lf-set* (*ti-pad-combine n ti*) []) (*lf-set* (*adjust-ti k acc upd*) [])
apply(*clarsimp simp*: *ti-pad-combine-def Let-def padding-desc-def*)
apply(*rule ti-ind-extend-ti*)
apply(*clarsimp simp*: *ti-ind-def fu-commutes-def fa-fu-ind-def*)
apply *assumption*
done

definition *acc-ind* :: (*'a* ⇒ *'b*) ⇒ *'a field-desc set* ⇒ *bool* **where**
acc-ind acc X ≡ ∀ *x bs v*. *x* ∈ *X* ⟶ *acc* (*field-update x bs v*) = *acc v*

definition *fu-s-comm-k* :: *'a leaf-desc set* ⇒ (*'b* ⇒ *'a* ⇒ *'a*) ⇒ *bool* **where**
fu-s-comm-k X upd ≡ ∀ *x*. *x* ∈ *field-update* ‘ *lf-fd* ‘ *X* ⟶ *fu-commutes x upd*

definition *upd-ind* :: *'a leaf-desc set* ⇒ (*'b* ⇒ *'a* ⇒ *'a*) ⇒ *bool* **where**

upd-ind X upd ≡ *fu-s-comm-k X upd*

definition *fa-ind* :: *'a field-desc set* ⇒ (*'b* ⇒ *'a* ⇒ *'a*) ⇒ *bool* **where**
fa-ind X upd ≡ ∀ *x bs v*. *x* ∈ *X* ⟶ *field-access x* (*upd bs v*) = *field-access x v*

lemma *lf-fd-fn*:
∀ *fn*. *lf-fd* ‘ (*lf-set* (*t*::*'a xtyp-info*) *fn*) = *lf-fd* ‘ (*lf-set t* [])
∀ *fn*. *lf-fd* ‘ (*lf-set-struct* (*st*::*'a xtyp-info-struct*) *fn*) = *lf-fd* ‘ (*lf-set-struct st* [])
∀ *fn*. *lf-fd* ‘ (*lf-set-list* (*ts*::*'a xtyp-info-tuple list*) *fn*) = *lf-fd* ‘ (*lf-set-list ts* [])
∀ *fn*. *lf-fd* ‘ (*lf-set-tuple* (*x*::*'a xtyp-info-tuple*) *fn*) = *lf-fd* ‘ (*lf-set-tuple x* [])
by (*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp simp*: *image-Un*›) *metis*+

lemma *lf-set-empty-typ-info* [*simp*]:
lf-set (*empty-typ-info algn tn*) *fn* = {}
by (*clarsimp simp*: *empty-typ-info-def*)

lemma *upd-ind-empty* [*simp*]:
upd-ind {} *upd*
by (*clarsimp simp*: *upd-ind-def fu-s-comm-k-def*)

lemma *upd-ind-extend-ti*:
⟦ *upd-ind* (*lf-set s* []) *upd*; *upd-ind* (*lf-set t* []) *upd* ⟧ ⟹
upd-ind (*lf-set* (*extend-ti s t algn fn d*) []) *upd*
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*)
subgoal
apply (*simp add*: *upd-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn*(*1*))
subgoal
apply (*simp add*: *upd-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn*(*1*))
done
done

lemma (**in** *c-type*) *upd-ind-ti-typ-combine*:
⟦ *upd-ind* (*lf-set ti* []) *h*; ⋀*w u v*. *upd w* (*h u v*) = *h u* (*upd w v*);
⋀*w v*. *acc* (*h w v*) = *acc v*; ⋀*v*. *upd* (*acc v*) *v* = *v* ⟧
⟹ *upd-ind* (*lf-set* (*ti-typ-combine* (*t*::*'a itself*) *acc upd algn fn ti*) []) *h*
apply(*clarsimp simp*: *ti-typ-combine-def Let-def*)
apply(*erule upd-ind-extend-ti*)
apply(*clarsimp simp*: *upd-ind-def fu-s-comm-k-def*)
apply(*drule lf-set-adjust-ti*)
apply *clarsimp*
apply(*clarsimp simp*: *update-desc-def fu-commutes-def*)
done

lemma *upd-ind-ti-pad-combine*:

upd-ind ((*lf-set ti* [])) *upd* ⟹ *upd-ind* ((*lf-set* (*ti-pad-combine n ti*) [])) *upd*
apply(*clarsimp simp*: *ti-pad-combine-def Let-def padding-desc-def*)
apply(*erule upd-ind-extend-ti*)
apply(*auto simp*: *upd-ind-def fu-s-comm-k-def fu-commutes-def*)
done

lemma (**in** *c-type*) *upd-ind-ti-typ-pad-combine*:
⟦ *upd-ind* (*lf-set ti* []) *h*; ⋀*w u v*. *upd w* (*h u v*) = *h u* (*upd w v*);
⋀*w v*. *acc* (*h w v*) = *acc v*; ⋀*v*. *upd* (*acc v*) *v* = *v* ⟧
⟹ *upd-ind* (*lf-set* (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*) []) *h*
unfolding *ti-typ-pad-combine-def Let-def*
by (*fastforce intro*!: *upd-ind-ti-typ-combine upd-ind-ti-pad-combine*)

lemma *acc-ind-empty* [*simp*]:
acc-ind acc {}
by (*clarsimp simp*: *acc-ind-def*)

lemma *acc-ind-extend-ti*:
⟦ *acc-ind acc* (*lf-fd* ‘ *lf-set s* []); *acc-ind acc* (*lf-fd* ‘ *lf-set t* []) ⟧ ⟹
acc-ind acc (*lf-fd* ‘ *lf-set* (*extend-ti s t algn fn d*) [])
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*)
subgoal
apply (*simp add*: *acc-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn*(*1*))
subgoal
apply (*simp add*: *acc-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn*(*1*))
done
done

lemma (**in** *c-type*) *acc-ind-ti-typ-combine*:
⟦ *acc-ind h* (*lf-fd* ‘ *lf-set ti* []); ⋀*v w*. *h* (*upd w v*) = *h v*;
⋀*v*. *upd* (*acc v*) *v* = *v* ⟧
⟹ *acc-ind h* (*lf-fd* ‘ *lf-set* (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) [])
apply(*clarsimp simp*: *ti-typ-combine-def Let-def*)
apply(*erule acc-ind-extend-ti*)
apply(*clarsimp simp*: *acc-ind-def*)
apply(*drule lf-set-adjust-ti*)
apply *clarsimp*
apply(*clarsimp simp*: *update-desc-def*)
done

lemma *acc-ind-ti-pad-combine*:
acc-ind acc (*lf-fd* ‘ (*lf-set t* [])) ⟹ *acc-ind acc* (*lf-fd* ‘ (*lf-set* (*ti-pad-combine n t*) []))
apply(*clarsimp simp*: *ti-pad-combine-def Let-def padding-desc-def*)
apply(*erule acc-ind-extend-ti*)

apply(*auto simp*: *acc-ind-def*)
done

lemma (**in** *c-type*) *acc-ind-ti-typ-pad-combine*:
⟦ *acc-ind h* (*lf-fd* ‘ *lf-set ti* []); ⋀*v w*. *h* (*upd w v*) = *h v*; ⋀*v*. *upd* (*acc v*) *v* = *v* ⟧
⟹ *acc-ind h* (*lf-fd* ‘ *lf-set* (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*)
[])
by (*auto simp*: *ti-typ-pad-combine-def Let-def intro*: *acc-ind-ti-typ-combine acc-ind-ti-pad-combine*)

lemma *fa-ind-empty* [*simp*]:
fa-ind {} *upd*
by (*clarsimp simp*: *fa-ind-def*)

lemma *fa-ind-extend-ti*:
⟦ *fa-ind* (*lf-fd* ‘ *lf-set s* []) *upd*; *fa-ind* (*lf-fd* ‘ *lf-set t* []) *upd* ⟧ ⟹
fa-ind (*lf-fd* ‘ *lf-set* (*extend-ti s t algn fn d*) []) *upd*
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*)
subgoal
apply (*simp add*: *fa-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn*(*1*))
subgoal
apply (*simp add*: *fa-ind-def image-Un fu-s-comm-k-def*)
by (*metis lf-fd-fn*(*1*))
done
done

lemma (**in** *c-type*) *fa-ind-ti-typ-combine*:
⟦ *fa-ind* (*lf-fd* ‘ *lf-set ti* []) *h*; ⋀*v w*. *acc* (*h w v*) = *acc v*;
⋀*v*. *upd* (*acc v*) *v* = *v* ⟧
⟹ *fa-ind* (*lf-fd* ‘ *lf-set* (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) []) *h*
apply(*clarsimp simp*: *ti-typ-combine-def Let-def*)
apply(*erule fa-ind-extend-ti*)
apply(*clarsimp simp*: *fa-ind-def fu-s-comm-k-def*)
apply(*drule lf-set-adjust-ti*)
apply *clarsimp*
apply(*clarsimp simp*: *update-desc-def fu-commutes-def*)
done

lemma *fa-ind-ti-pad-combine*:
fa-ind (*lf-fd* ‘ (*lf-set ti* [])) *upd* ⟹ *fa-ind* (*lf-fd* ‘ (*lf-set* (*ti-pad-combine n ti*) []))
upd
apply(*clarsimp simp*: *ti-pad-combine-def Let-def padding-desc-def*)
apply(*erule fa-ind-extend-ti*)
apply(*auto simp*: *fa-ind-def*)
done

lemma (**in** *c-type*) *fa-ind-ti-typ-pad-combine*:

⟦ *fa-ind* (*lf-fd* ‘ *lf-set ti* []) *h*; ⋀*v w*. *f* (*h w v*) = *f v*;
⋀*v*. *g* (*f v*) *v* = *v* ⟧
⟹ *fa-ind* (*lf-fd* ‘ *lf-set* (*ti-typ-pad-combine* (*t*::′*a itself*) *f g algn fn ti*) []) *h*
by (*auto simp*: *ti-typ-pad-combine-def Let-def intro*: *fa-ind-ti-typ-combine fa-ind-ti-pad-combine*)

lemma (**in** *wf-type*) *wf-lf-ti-typ-combine*:
⟦ *wf-lf* (*lf-set ti* []); *fn* ∉ *set* (*field-names-list ti*);
⋀*v*. *upd* (*acc v*) *v* = *v*; ⋀*w u v*. *upd w* (*upd u v*) = *upd w v*;
⋀*w v*. *acc* (*upd w v*) = *w*;
upd-ind (*lf-set ti* []) *upd*; *acc-ind acc* (*lf-fd* ‘ *lf-set ti* []);
fa-ind (*lf-fd* ‘ *lf-set ti* []) *upd* ⟧ ⟹
wf-lf (*lf-set* (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) [])
apply(*clarsimp simp*: *ti-typ-combine-def Let-def*)
apply(*rule wf-lf-extend-ti*; *simp?*)
apply(*rule wf-lf-adjust-ti*; *simp*)
apply(*clarsimp simp*: *ti-ind-def*)
apply(*drule lf-set-adjust-ti*, *simp*)
apply(*clarsimp simp*: *fu-commutes-def update-desc-def upd-ind-def acc-ind-def fu-s-comm-k-def*
fa-fu-ind-def fa-ind-def)
done

lemma (**in** *wf-type*) *wf-lf-ti-typ-pad-combine*:
⟦ *wf-lf* (*lf-set ti* []); *fn* ∉ *set* (*field-names-list ti*); *hd fn* ≠ *CHR* ′′!′′;
⋀*v*. *upd* (*acc v*) *v* = *v*; ⋀*w u v*. *upd w* (*upd u v*) = *upd w v*;
⋀*w v*. *acc* (*upd w v*) = *w*;
upd-ind (*lf-set ti* []) *upd*; *acc-ind acc* (*lf-fd* ‘ *lf-set ti* []);
fa-ind (*lf-fd* ‘ *lf-set ti* []) *upd* ⟧ ⟹
wf-lf (*lf-set* (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*) [])
apply(*clarsimp simp*: *ti-typ-pad-combine-def Let-def*)
apply (*fastforce intro*!: *wf-lf-ti-typ-combine wf-lf-ti-pad-combine upd-ind-ti-pad-combine*
acc-ind-ti-pad-combine fa-ind-ti-pad-combine)
done

definition *upd-fa-ind X upd* ≡ *upd-ind X upd* ∧ *fa-ind* (*lf-fd* ‘ *X*) *upd*

lemma (**in** *wf-type*) *wf-lf-ti-typ-pad-combine′*:
⟦ *wf-lf* (*lf-set ti* []); *fn* ∉ *set* (*field-names-list ti*); *hd fn* ≠ *CHR* ′′!′′;
⋀*v*. *upd* (*acc v*) *v* = *v*; ⋀*w u v*. *upd w* (*upd u v*) = *upd w v*;
⋀*w v*. *acc* (*upd w v*) = *w*;
upd-fa-ind (*lf-set ti* []) *upd*; *acc-ind acc* (*lf-fd* ‘ *lf-set ti* [])⟧
⟹
wf-lf (*lf-set* (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*) [])
unfolding *upd-fa-ind-def*
by (*erule conjE*) (*rule wf-lf-ti-typ-pad-combine*)

lemma (**in** *c-type*) *upd-fa-ind-ti-typ-pad-combine*:
⟦ *upd-fa-ind* (*lf-set ti* []) *h*; ⋀*w u v*. *upd w* (*h u v*) = *h u* (*upd w v*);

$\bigwedge w\ v.\ acc\ (h\ w\ v) = acc\ v;\ \bigwedge v.\ upd\ (acc\ v)\ v = v\ ]\!]$
$\Longrightarrow$ *upd-fa-ind* (*lf-set* (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn ti*) []) *h*
unfolding *upd-fa-ind-def*
by (*auto intro*: *upd-ind-ti-typ-pad-combine fa-ind-ti-typ-pad-combine*)

lemma *upd-fa-ind-empty* [*simp*]:
upd-fa-ind {} *h*
by (*simp add*: *upd-fa-ind-def*)

lemma *wf-align-empty-typ-info*: *wf-align* (*empty-typ-info algn tn*)
by (*simp add*: *wf-align-def empty-typ-info-def*)

lemma *wf-align-list*: *wf-align t* $\Longrightarrow$ *wf-align-list fs* $\Longrightarrow$ *wf-align-list* (*fs* @ [*DTuple t f d*])
by (*induct fs*) *auto*
lemma *wf-align-struct*: *wf-align t* $\Longrightarrow$ *wf-align-struct st* $\Longrightarrow$ *wf-align-struct* (*extend-ti-struct st t f d*)
apply (*cases st*)
apply *simp*
apply (*simp add*: *wf-align-list*)
done

lemma *align-td-extend-ti*: *align-td* (*extend-ti tag t algn f d*) = *max* (*align-td tag*) (*max algn* (*align-td t*))
apply (*cases tag*)
apply (*simp*)
done

lemma *align-td-struct-extend-ti*: *aggregate-struct st* $\Longrightarrow$
align-td-struct (*extend-ti-struct st t f d*) = *max* (*align-td-struct st*) (*align-td t*)
by (*cases st*) *auto*
lemma *wf-align-extend-ti'*:
assumes *wf-t*: *wf-align t*
assumes *agg*: *aggregate tag*
assumes *wf-tag*: *wf-align tag*
shows *wf-align* (*extend-ti tag t algn f d*)
proof (*cases tag*)
case (*TypDesc algn' st n*)
with *wf-tag agg* **obtain** *le*: *align-td-wo-align-struct st* $\leq$ *algn'*
and *le'*: *align-td-struct st* $\leq$ *algn'*
and *wf-st*: *wf-align-struct st*
and *agg-st*: *aggregate-struct st* **by** *auto*
from *wf-align-struct* [*OF wf-t wf-st*]
have *wf-st*: *wf-align-struct* (*extend-ti-struct st t f d*) .
from *align-td-wo-align-le-align-td* (*2*) [*OF this*]
have *align-td-wo-align-struct* (*extend-ti-struct st t f d*) $\leq$ *align-td-struct* (*extend-ti-struct st t f d*) .
also from *align-td-struct-extend-ti* [*OF agg-st*]
have ... = *max* (*align-td-struct st*) (*align-td t*) .

finally
have *align-td-wo-align-struct* (*extend-ti-struct st t f d*) ≤ *max algn'* (*max algn* (*align-td t*))
using *le'*
by (*metis* (*full-types*) *le-max-iff-disj max.orderE*)
moreover from *align-td-struct-extend-ti* [*OF agg-st*] *le'*
have *align-td-struct* (*extend-ti-struct st t f d*) ≤ *max algn'* (*max algn* (*align-td t*))
by (*metis max.cobounded2 max.mono*)
ultimately show *?thesis*
by (*simp add*: *TypDesc wf-st*)
qed

lemma (**in** *mem-type*) *wf-align-extend-ti*:
assumes *agg*: *aggregate tag*
assumes *wf-tag*: *wf-align tag*
shows *wf-align* (*extend-ti tag* (*typ-info-t* (*TYPE*('*a*))) *algn f d*)
proof −
have *wf-align* (*typ-info-t* (*TYPE*('*a*))) **by** (*rule wf-align*)
from *wf-align-extend-ti'* [*OF this agg wf-tag*] **show** *?thesis* **.**
qed

lemma *wf-align-map-td* [*simp*]:
wf-align (*map-td f g d*) = *wf-align d*
wf-align-struct (*map-td-struct f g ts*) = *wf-align-struct* (*ts*)
wf-align-list (*map-td-list f g fs*) = *wf-align-list fs*
wf-align-tuple (*map-td-tuple f g fd*) = *wf-align-tuple fd*
by (*induct d* **and** *ts* **and** *fs* **and** *fd*) *auto*

lemma *wf-align-adjust-ti*[*simp*]: *wf-align* (*adjust-ti t acc upd*) = *wf-align t*
by (*simp add*: *adjust-ti-def*)

lemma (**in** *mem-type*) *wf-align-ti-typ-combine*:
aggregate tag ⟹ *wf-align tag* ⟹ *wf-align* (*ti-typ-combine* (*TYPE*('*a*)) *acc upd algn fn tag*)
apply (*simp add*: *ti-typ-combine-def Let-def*)
apply (*rule wf-align-extend-ti'*)
apply (*simp add*: *wf-align*)
apply *assumption*
apply *assumption*
done

lemma *wf-align-ti-pad-combine*:
aggregate tag ⟹ *wf-align tag* ⟹ *wf-align* (*ti-pad-combine n tag*)
apply (*simp add*: *ti-pad-combine-def Let-def*)
apply (*rule wf-align-extend-ti'*)
apply *simp*
apply *assumption*
apply *assumption*

done

lemma (**in** *mem-type*) *wf-align-ti-typ-pad-combine*:
aggregate tag $\Longrightarrow$ *wf-align tag* $\Longrightarrow$ *wf-align* (*ti-typ-pad-combine* (*TYPE*('*a*)) *acc upd algn fn tag*)
by (*simp add*: *ti-typ-pad-combine-def Let-def wf-align-ti-pad-combine wf-align-ti-typ-combine*)

lemma *wf-align-map-align*:
assumes *wf-tag*: *wf-align tag*
assumes *mono*: $\bigwedge a.\ a \leq f\ a$
shows *wf-align* (*map-align f tag*)
using *wf-tag mono*
apply (*cases tag*)
using *order-trans* **apply** *auto*
done

lemma *wf-align-final-pad*: *aggregate tag* $\Longrightarrow$ *wf-align tag* $\Longrightarrow$ *wf-align* (*final-pad algn tag*)
by (*auto simp add*: *final-pad-def Let-def max-2-exp wf-align-map-align wf-align-ti-pad-combine*)

lemmas *wf-align-simps* =
wf-align-empty-typ-info
wf-align-ti-typ-pad-combine
wf-align-ti-typ-combine
wf-align-ti-pad-combine
wf-align-final-pad

lemma *align-field-empty-typ-info* [*simp*]:
align-field (*empty-typ-info algn tn*)
by (*clarsimp simp*: *empty-typ-info-def align-field-def*)

lemma *align-td-wo-align-field-lookup*:
$\forall f\ m\ s\ n.$ *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *Some* (*s*,*n*) $\longrightarrow$ *align-td-wo-align s* $\leq$ *align-td-wo-align t*
$\forall f\ m\ s\ n.$ *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *Some* (*s*,*n*) $\longrightarrow$ *align-td-wo-align s* $\leq$ *align-td-wo-align-struct st*
$\forall f\ m\ s\ n.$ *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* = *Some* (*s*,*n*) $\longrightarrow$ *align-td-wo-align s* $\leq$ *align-td-wo-align-list ts*
$\forall f\ m\ s\ n.$ *field-lookup-tuple* (*x*::('*a*,'*b*) *typ-tuple*) *f m* = *Some* (*s*,*n*) $\longrightarrow$ *align-td-wo-align s* $\leq$ *align-td-wo-align-tuple x*
by (*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp*›)
(*fastforce simp*: *max-def split*: *option.splits*)

lemma *align-td-field-lookup*[*rule-format*]:
$\forall f\ m\ s\ n.$ *wf-align t* $\longrightarrow$ *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *Some* (*s*,*n*) $\longrightarrow$ *align-td s* $\leq$ *align-td t*
$\forall f\ m\ s\ n.$ *wf-align-struct st* $\longrightarrow$ *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *Some* (*s*,*n*) $\longrightarrow$ *align-td s* $\leq$ *align-td-struct st*
$\forall f\ m\ s\ n.$ *wf-align-list ts* $\longrightarrow$ *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* =

Some (*s*,*n*) ⟶ *align-td s* ≤ *align-td-list ts*
∀ *f m s n*. *wf-align-tuple x* ⟶ *field-lookup-tuple* (*x*::(′*a*,′*b*) *typ-tuple*) *f m* = *Some* (*s*,*n*) ⟶ *align-td s* ≤ *align-td-tuple x*
apply (*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp*›)
apply (*fastforce simp*: *max-def split*: *option.splits*)+
done

lemma (**in** *mem-type*) *align-td-field-lookup-mem-type*: *field-lookup* (*typ-info-t* (*TYPE*(′*a*))) *f m* = *Some* (*s*, *n*) ⟹
align-td s ≤ *align-td* (*typ-info-t* (*TYPE*(′*a*)))
apply (*rule align-td-field-lookup*(*1*))
apply (*rule wf-align*)
apply *simp*
done

lemma *align-field-extend-ti*:
⟦ *align-field s*; *align-field t*; *wf-align t*; 2^(*align-td t*) *dvd size-td s* ⟧ ⟹
align-field (*extend-ti s t algn fn d*)
apply(*cases s*, *clarsimp*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*, *clarsimp*)
apply(*clarsimp simp*: *align-field-def split*: *option.splits*)
apply(*clarsimp simp*: *align-field-def*)
apply(*subst* (*asm*) *field-lookup-list-append*)
apply(*clarsimp split*: *if-split-asm option.splits*)
subgoal for *x2 f s n*
apply(*cases f*, *clarsimp*)
apply *clarsimp*
apply(*frule field-lookup-offset2*)
apply (*meson align-td-field-lookup*(*1*) *dvd-diffD field-lookup-offset-le*(*1*) *power-le-dvd*)
done
subgoal for *x2 f s n*
by(*cases f*) *auto*
done
done

lemma *align-field-ti-pad-combine*:
align-field ti ⟹ *align-field* (*ti-pad-combine n ti*)
apply(*clarsimp simp*: *ti-pad-combine-def Let-def*)
apply(*erule align-field-extend-ti*)
apply(*clarsimp simp*: *align-field-def*)
apply *clarsimp*
apply *clarsimp*
done

lemma *align-field-map-align* [*simp*]: *align-field* (*map-align f t*) = *align-field t*
by (*cases t*) (*auto simp add*: *align-field-def*)

lemma *align-field-final-pad*:

align-field ti $\Longrightarrow$ *align-field* (*final-pad algn ti*)
apply(*clarsimp simp*: *final-pad-def Let-def split*: *if-split-asm*)
apply(*erule align-field-ti-pad-combine*)
done

lemma *field-lookup-adjust-ti-None*:
$\forall$ *fn m s n. field-lookup* (*adjust-ti t acc upd*) *fn m* = *None* $\longrightarrow$
(*field-lookup t fn m* = *None*)
$\forall$ *fn m s n. field-lookup-struct* (*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *st*)
fn m = *None* $\longrightarrow$
(*field-lookup-struct st fn m* = *None*)
$\forall$ *fn m s n. field-lookup-list* (*map-td-list* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *ts*) *fn m* = *None* $\longrightarrow$
(*field-lookup-list ts fn m* = *None*)
$\forall$ *fn m s n. field-lookup-tuple* (*map-td-tuple* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *x*) *fn m* = *None* $\longrightarrow$
(*field-lookup-tuple x fn m* = *None*)
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*clarsimp simp*: *adjust-ti-def split*: *option.splits*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*clarsimp simp*: *adjust-ti-def split*: *option.splits*)
apply (*cases dt-tuple*)
apply *clarsimp*
done
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** (*clarsimp simp*: *adjust-ti-def split*: *option.splits*)
qed

lemma *field-lookup-adjust-ti'* [*rule-format*]:
$\forall$ *fn m s n. field-lookup* (*adjust-ti t acc upd*) *fn m* = *Some* (*s*,*n*) $\longrightarrow$
($\exists$ *s'. field-lookup t fn m* = *Some* (*s'*,*n*) $\wedge$ *align-td-wo-align s* = *align-td-wo-align s'*)
$\forall$ *fn m s n. field-lookup-struct* (*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *st*)
fn m = *Some* (*s*,*n*) $\longrightarrow$

(∃ s′. field-lookup-struct st fn m = Some (s′,n) ∧ align-td-wo-align s = align-td-wo-align s′)
∀ fn m s n. field-lookup-list (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) ts) fn m = Some (s,n) ⟶
(∃ s′. field-lookup-list ts fn m = Some (s′,n) ∧ align-td-wo-align s = align-td-wo-align s′)
∀ fn m s n. field-lookup-tuple (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd) x) fn m = Some (s,n) ⟶
(∃ s′. field-lookup-tuple x fn m = Some (s′,n) ∧ align-td-wo-align s = align-td-wo-align s′)
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*clarsimp simp*: *adjust-ti-def*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** (*clarsimp simp*: *adjust-ti-def*)
next
case (*TypAggregate list*)
then show *?case* **by** (*clarsimp simp*: *adjust-ti-def*)
next
case *Nil-typ-desc*
then show *?case* **by** (*clarsimp simp*: *adjust-ti-def*)
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*clarsimp simp*: *adjust-ti-def*)
apply(*clarsimp split*: *option.splits*)
apply(*rule conjI*; *clarsimp*)
apply(*cases dt-tuple*, *clarsimp*)
apply(*cases dt-tuple*, *clarsimp split*: *if-split-asm*)
subgoal for *fn*
apply(*drule spec* [**where** *x*=*fn*])
apply *clarsimp*
apply(*fold adjust-ti-def*)
apply(*subst* (*asm*) *field-lookup-adjust-ti-None*; *simp*)
done
apply *fastforce*
done
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** (*clarsimp simp*: *adjust-ti-def*)
qed

lemma *field-lookup-adjust-ti‴* [*rule-format*]:
∀ fn m s n. field-lookup (adjust-ti t acc upd) fn m = Some (s,n) ⟶
(∃ s′. field-lookup t fn m = Some (s′,n) ∧ align-td s = align-td s′)
∀ fn m s n. field-lookup-struct (map-td-struct (λn algn. update-desc acc upd) (update-desc acc upd) st)
fn m = Some (s,n) ⟶

(∃ s′. field-lookup-struct st fn m = Some (s′,n) ∧ align-td s = align-td s′)
∀ fn m s n. field-lookup-list (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) ts) fn m = Some (s,n) ⟶
(∃ s′. field-lookup-list ts fn m = Some (s′,n) ∧ align-td s = align-td s′)
∀ fn m s n. field-lookup-tuple (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd) x) fn m = Some (s,n) ⟶
(∃ s′. field-lookup-tuple x fn m = Some (s′,n) ∧ align-td s = align-td s′)
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*clarsimp simp*: *adjust-ti-def*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply(*clarsimp simp*: *adjust-ti-def*)
apply(*clarsimp split*: *option.splits*)
apply(*rule conjI*; *clarsimp*)
apply(*cases dt-tuple*, *clarsimp*)
apply(*cases dt-tuple*, *clarsimp split*: *if-split-asm*)
subgoal for *fn*
apply(*drule spec* [**where** *x*=*fn*])
apply *clarsimp*
apply(*fold adjust-ti-def*)
apply(*subst* (*asm*) *field-lookup-adjust-ti-None*; *simp*)
done
apply *fastforce*
done
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by**(*clarsimp simp*: *adjust-ti-def*)
qed

lemma *field-lookup-adjust-ti*:
⟦ *field-lookup* (*adjust-ti t acc upd*) *fn m* = *Some* (*s*,*n*) ⟧ ⟹
(∃ *s*′. *field-lookup t fn m* = *Some* (*s*′,*n*) ∧ *align-td-wo-align s* = *align-td-wo-align s*′)
by (*rule field-lookup-adjust-ti*′)

lemma *field-lookup-adjust-ti1*:
⟦ *field-lookup* (*adjust-ti t acc upd*) *fn m* = *Some* (*s*,*n*) ⟧ ⟹
(∃ *s*′. *field-lookup t fn m* = *Some* (*s*′,*n*) ∧ *align-td s* = *align-td s*′)

by (*rule field-lookup-adjust-ti'''*)

lemma *align-adjust-ti*:
align-field ti $\Longrightarrow$ *align-field* (*adjust-ti ti acc upd*)
apply(*clarsimp simp*: *align-field-def*)
apply(*drule field-lookup-adjust-ti1*)
apply *clarsimp*
done

lemma (**in** *mem-type*) *align-field-ti-typ-combine*:
⟦ *align-field ti*; *2* ^ *align-td* (*typ-info-t TYPE*('*a*)) *dvd size-td ti* ⟧ $\Longrightarrow$
align-field (*ti-typ-combine* (*t*::'*a itself*) *acc upd algn fn ti*)
apply(*clarsimp simp*: *ti-typ-combine-def Let-def*)
apply(*rule align-field-extend-ti*, *assumption*)
apply(*rule align-adjust-ti*)
apply(*rule align-field*)
apply (*simp add*: *wf-align*)
apply *simp*
done

lemma (**in** *mem-type*) *align-td-wo-align-type-info-t-le-align-td*:
align-td-wo-align (*typ-info-t TYPE*('*a*)) $\leq$ *align-td* (*typ-info-t TYPE*('*a*))
proof −
have *wf-align* (*typ-info-t TYPE*('*a*)) **by** (*rule wf-align*)
then show *?thesis* **thm** *wf-align.wfal0* **by** (*rule align-td-wo-align-le-align-td*(*1*))
qed

lemma (**in** *mem-type*) *align-field-ti-typ-pad-combine*:
⟦*wf-align ti*; *align-field ti*; *aggregate ti*⟧ $\Longrightarrow$
align-field (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn ti*)

unfolding *ti-typ-pad-combine-def Let-def*
apply (*rule align-field-ti-typ-combine*)
subgoal
apply *clarsimp*
apply (*rule align-field-ti-pad-combine*)
apply *assumption*
done
apply *clarsimp*
apply (*rule conjI*)
subgoal
apply (*clarsimp simp add*: *align-of-def*)
apply (*metis* (*no-types*, *lifting*) *align-td-wo-align-type-info-t-le-align-td dvd-padup-add max.cobounded2 max-2-exp*
power-le-dvd size-td-lt-ti-pad-combine zero-less-numeral zero-less-power)
done
apply (*clarsimp simp add*: *align-of-def*)
by (*simp add*: *max-2-exp padup-dvd power-le-dvd*)

lemma *npf-extend-ti* [*simp*]:
non-padding-fields (*extend-ti s t algn fn d*) = *non-padding-fields s* @
(*if hd fn* = *CHR* ''!'' *then* [] *else* [*fn*])
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*; *simp*)
done
done

lemma *npf-ti-pad-combine* [*simp*]:
non-padding-fields (*ti-pad-combine n tag*) = *non-padding-fields tag*
by (*clarsimp simp*: *ti-pad-combine-def Let-def*)

lemma (**in** *c-type*) *npf-ti-typ-combine* [*simp*]:
hd fn ≠ *CHR* ''!'' ⟹
non-padding-fields (*ti-typ-combine* (*t*::'*a itself*) *acc upd algn fn tag*) = *non-padding-fields tag* @ [*fn*]
by (*clarsimp simp*: *ti-typ-combine-def Let-def*)

lemma (**in** *c-type*) *npf-ti-typ-pad-combine* [*simp*]:
hd fn ≠ *CHR* ''!'' ⟹
non-padding-fields (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn tag*) = *non-padding-fields tag* @ [*fn*]
by (*clarsimp simp*: *ti-typ-pad-combine-def Let-def*)

lemma *non-padding-fields-map-align* [*simp*]:
non-padding-fields (*map-align f t*) = *non-padding-fields t*
by (*cases t*) *simp*

lemma *npf-final-pad* [*simp*]:
non-padding-fields (*final-pad algn tag*) = *non-padding-fields tag*
by (*clarsimp simp*: *final-pad-def Let-def*)

lemma *npf-empty-typ-info* [*simp*]:
non-padding-fields (*empty-typ-info algn tn*) = []
by (*clarsimp simp*: *empty-typ-info-def*)

definition *field-fd'* :: '*a xtyp-info* ⇒ *qualified-field-name* ⇀ '*a field-desc* **where**
field-fd' t f ≡ *case field-lookup t f 0 of None* ⇒ *None* | *Some x* ⇒ *Some* (*field-desc* (*fst x*))

lemma *padup-zero* [*simp*]:
padup n 0 = *0*
by (*clarsimp simp*: *padup-def*)

lemma *padup-same* [*simp*]:
padup n n = *0*
by (*clarsimp simp*: *padup-def*)

lemmas *size-td-simps-0* = *size-td-lt-final-pad size-td-lt-ti-typ-pad-combine*

lemmas *size-td-simps-1* = *size-td-simps-0*
aggregate-ti-typ-pad-combine aggregate-empty-typ-info

lemmas *size-td-simps-2* = *padup-def align-of-final-pad align-of-def*

lemmas *size-td-simps* = *size-td-simps-1 size-td-simps-2*

lemmas *size-td-simps-3* = *size-td-simps-0 size-td-simps-2*

lemma *fu-commutes-sym*: *fu-commutes x y* = *fu-commutes y x*
by (*auto simp add*: *fu-commutes-def*)

lemma *wf-lf-insert-recursion*:
assumes *wf-D*: *wf-lf D*
assumes *cons-x*: *fd-cons-desc* (*lf-fd x*) (*lf-sz x*)
assumes *comm-D*: $\bigwedge y.\ y \in D \Longrightarrow$ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) ∧
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) ∧
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*)
shows *wf-lf* (*insert x D*)
proof −
have *comm-x-D*:
fu-commutes (*field-update* (*lf-fd x′*)) (*field-update* (*lf-fd y*)) **and**
fa-fu-ind (*lf-fd x′*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x′*)
if *x′-in*: $x' \in$ *insert x D* **and** *y-in*: $y \in$ *insert x D* **and** *neq*: *lf-fn y* ≠ *lf-fn x′*
for x' **and** y
proof −
show *fu-commutes* (*field-update* (*lf-fd x′*)) (*field-update* (*lf-fd y*))
proof (*cases* $x' = x$)
case *True*
from *True y-in neq comm-D x′-in* **show** *?thesis*
by *auto*
next
case *False*
note *neq-x′-x* = *this*
with *x′-in* **have** *x′-D*: $x' \in D$
by *auto*
from *comm-D* [*OF this*] **have** *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd x′*))
by *auto*
with *fu-commutes-sym* **have** *comm-x′-x*: *fu-commutes* (*field-update* (*lf-fd x′*)) (*field-update* (*lf-fd x*))
by *auto*
from *neq* **have** *neq′*: $y \neq x'$
by *auto*

```
      show ?thesis
      proof (cases y = x)
        case True
        with comm-x'-x show ?thesis by simp
      next
        case False
        from False y-in have y-D: y ∈ D by simp
        with x'-D neq wf-D
        show ?thesis
          by (auto simp add: wf-lf-def)
      qed
    qed
  next
    show fa-fu-ind (lf-fd x') (lf-fd y) (lf-sz y) (lf-sz x')
    proof (cases x' = x)
      case True
      from True y-in neq comm-D x'-in show ?thesis
        by auto
    next
      case False
      note neq-x'-x = this
      with x'-in have x'-D: x' ∈ D
        by auto
      from comm-D [OF this] have fa-fu-ind (lf-fd x) (lf-fd x') (lf-sz x') (lf-sz x)
and
          fa-fu-x'-x: fa-fu-ind (lf-fd x') (lf-fd x) (lf-sz x) (lf-sz x') by auto
      from neq have neq': y ≠ x'
        by auto
      show ?thesis
      proof (cases y = x)
        case True
        with fa-fu-x'-x show ?thesis by simp
      next
        case False
        from False y-in have y-D: y ∈ D by simp
        with x'-D neq wf-D
        show ?thesis
          by (auto simp add: wf-lf-def)
      qed
    qed
  qed
  with cons-x wf-D
  show ?thesis
    by (auto simp add: wf-lf-def)
qed



lemma wf-lf-insert-recursion':
```

assumes *cons-x*: *fd-cons-desc* (*lf-fd x*) (*lf-sz x*)
assumes *comm-D*: $\bigwedge y.\ y \in D \Longrightarrow$ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) $\wedge$
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) $\wedge$
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*)
assumes *wf-D*: *wf-lf D*
shows *wf-lf* (*insert x D*)
using *wf-D cons-x comm-D*
by (*rule wf-lf-insert-recursion*)

lemma *wf-lf-insert-recursion''*:
assumes *wf-D*: *wf-lf D*
assumes *cons-x*: *fd-cons-desc* (*lf-fd x*) (*lf-sz x*)
assumes *comm-D*: $\bigwedge y.\ y \in D \Longrightarrow$ *lf-fn y* $\neq$ *lf-fn x* $\Longrightarrow$ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) $\wedge$
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) $\wedge$
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*)
shows *wf-lf* (*insert x D*)
proof −
have *comm-x-D*:
fu-commutes (*field-update* (*lf-fd x'*)) (*field-update* (*lf-fd y*)) **and**
fa-fu-ind (*lf-fd x'*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x'*)
if *x'-in*: $x' \in$ *insert x D* **and** *y-in*: $y \in$ *insert x D* **and** *neq*: *lf-fn y* $\neq$ *lf-fn x'*
for x' **and** y
proof −
show *fu-commutes* (*field-update* (*lf-fd x'*)) (*field-update* (*lf-fd y*))
proof (*cases* $x' = x$)
case *True*
from *True y-in neq comm-D x'-in* **show** *?thesis*
by *auto*
next
case *False*
note *neq-x'-x* = *this*
with *x'-in* **have** *x'-D*: $x' \in D$
by *auto*
from *neq* **have** *neq'*: $y \neq x'$
by *auto*
show *?thesis*
proof (*cases* $y = x$)
case *True*
with *neq neq-x'-x neq'*
have *lf-fn x'* $\neq$ *lf-fn x* **by** *simp*
from *comm-D* [*OF x'-D this*] **have** *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd x'*))
by *auto*
with *fu-commutes-sym* **have** *comm-x'-x*: *fu-commutes* (*field-update* (*lf-fd x'*)) (*field-update* (*lf-fd x*))
by *auto*

```
        from True comm-x'-x show ?thesis by simp
      next
        case False
        from False y-in have y-D: y ∈ D by simp
        with x'-D neq wf-D
        show ?thesis
          by (auto simp add: wf-lf-def)
      qed
    qed
  next
    show fa-fu-ind (lf-fd x') (lf-fd y) (lf-sz y) (lf-sz x')
    proof (cases x' = x)
      case True
      from True y-in neq comm-D x'-in show ?thesis
        by auto
    next
      case False
      note neq-x'-x = this
      with x'-in have x'-D: x' ∈ D
        by auto
      from neq have neq': y ≠ x'
        by auto
      show ?thesis
      proof (cases y = x)
        case True
        with neq neq-x'-x neq'
        have lf-fn x' ≠ lf-fn x by simp
         from comm-D [OF x'-D this] have fa-fu-ind (lf-fd x) (lf-fd x') (lf-sz x')
(lf-sz x) and
          fa-fu-x'-x: fa-fu-ind (lf-fd x') (lf-fd x) (lf-sz x) (lf-sz x') by auto
        from True fa-fu-x'-x show ?thesis by simp
      next
        case False
        from False y-in have y-D: y ∈ D by simp
        with x'-D neq wf-D
        show ?thesis
          by (auto simp add: wf-lf-def)
      qed
    qed
  qed
  with cons-x wf-D
  show ?thesis
    by (auto simp add: wf-lf-def)
 qed


lemma wf-field-desc-empty [simp]:
wf-field-desc (|field-access = λv bs. [], field-update = λbs. id, field-sz = 0|)
 by (unfold-locales) (auto simp add: padding-base.eq-padding-def padding-base.eq-upto-padding-def)
```

lemma *field-desc-independent-subset*:
D ⊆ *E* ⟹ *field-desc-independent acc upd E* ⟹ *field-desc-independent acc upd D*
by (*auto simp add*: *field-desc-independent-def*)

lemma *field-desc-independent-union-iff*:
field-desc-independent acc upd (*D* ∪ *E*) =
(*field-desc-independent acc upd D* ∧ *field-desc-independent acc upd E*)
by (*auto simp add*: *field-desc-independent-def*)

lemma *field-desc-independent-unionI*:
field-desc-independent acc upd D ⟹ *field-desc-independent acc upd E* ⟹
field-desc-independent acc upd (*D* ∪ *E*)
by (*simp add*: *field-desc-independent-union-iff*)

lemma *field-desc-independent-unionD1*:
field-desc-independent acc upd (*D* ∪ *E*) ⟹
field-desc-independent acc upd D
by (*simp add*: *field-desc-independent-union-iff*)

lemma *field-desc-independent-unionD2*:
field-desc-independent acc upd (*D* ∪ *E*) ⟹
field-desc-independent acc upd E
by (*simp add*: *field-desc-independent-union-iff*)

lemma *field-desc-independent-insert-iff*:
field-desc-independent acc upd (*insert d D*) =
(*field-desc-independent acc upd* {*d*} ∧ *field-desc-independent acc upd D*)
apply (*subst insert-is-Un*)
apply (*simp only*: *field-desc-independent-union-iff*)
done

lemma *field-desc-independent-insertI*:
field-desc-independent acc upd {*d*} ⟹ *field-desc-independent acc upd D* ⟹
field-desc-independent acc upd (*insert d D*)
apply (*subst insert-is-Un*)
apply (*simp only*: *field-desc-independent-union-iff*)
done

lemma *field-desc-independent-insertD1*:
assumes *ins*: *field-desc-independent acc upd* (*insert d D*)
shows *field-desc-independent acc upd* {*d*}
proof −
from *ins* **have** *field-desc-independent acc upd* ({*d*} ∪ *D*)
by (*simp*)
from *field-desc-independent-unionD1* [*OF this*]
show *?thesis* **.**

qed

lemma *field-desc-independent-insertD2*:
assumes *ins*: *field-desc-independent acc upd* (*insert d D*)
shows *field-desc-independent acc upd D*
proof −
from *ins* **have** *field-desc-independent acc upd* ({*d*} ∪ *D*)
by (*simp*)
from *field-desc-independent-unionD2* [*OF this*]
show *?thesis* **.**
qed

lemma *field-descs-independent-append-first*: *field-descs-independent* (*xs* @ *ys*) ⟹ *field-descs-independent xs*
by (*induct xs arbitrary*: *ys*) (*auto intro*: *field-desc-independent-subset*)

lemma *field-descs-independent-append-second*: *field-descs-independent* (*xs* @ *ys*) ⟹ *field-descs-independent ys*
by (*induct xs arbitrary*: *ys*) (*auto intro*: *field-desc-independent-subset*)

lemma *field-descs-independent-append-first-ind*:
field-descs-independent (*xs* @ *ys*) ⟹ *x* ∈ *set xs* ⟹
field-desc-independent (*field-access x*) (*field-update x*) (*set ys*)
by (*induct xs arbitrary*: *ys*) (*auto simp add*: *field-desc-independent-union-iff*)

lemma *field-descs-independent-appendI1*:
field-descs-independent xs ⟹ *field-descs-independent ys* ⟹
(∀ *x* ∈ *set xs*. *field-desc-independent* (*field-access x*) (*field-update x*) (*set ys*)) ⟹
field-descs-independent (*xs* @ *ys*)
by (*induct xs arbitrary*: *ys*) (*auto simp add*: *field-desc-independent-union-iff*)

lemma *field-desc-independent-symmetric*:
(∀ *x* ∈ *X*. *field-desc-independent* (*field-access x*) (*field-update x*) *Y*) ⟹
(∀ *y* ∈ *Y*. *field-desc-independent* (*field-access y*) (*field-update y*) *X*)
by (*simp add*: *field-desc-independent-def fu-commutes-def*)

lemma *field-desc-independent-symmetric-singleton*:
field-desc-independent (*field-access x*) (*field-update x*) *Y* ⟹ *y* ∈ *Y*
⟹ *field-desc-independent* (*field-access y*) (*field-update y*) {*x*}
using *field-desc-independent-symmetric* **by** *blast*

lemma *field-descs-independent-appendI2*:
assumes *ind-xs*: *field-descs-independent xs*
assumes *ind-ys*: *field-descs-independent ys*
assumes *ind-xs-ys*: ∀ *y* ∈ *set ys*. *field-desc-independent* (*field-access y*) (*field-update y*) (*set xs*)

shows *field-descs-independent* (*xs* @ *ys*)
using *field-desc-independent-symmetric* [*OF ind-xs-ys*] *ind-xs ind-ys*
by (*auto intro*: *field-descs-independent-appendI1*)

lemma *field-desc-indepenent-empty* [*simp*]:
field-desc-independent (*field-access d*) (*field-update d*) {} **by** (*simp add*: *field-desc-independent-def*)

lemma *field-update-apply-field-updates-commute*:
fixes *d*::'*a field-desc*
fixes *ds*::'*a field-desc list*
assumes *ind-d*:*field-desc-independent* (*field-access d*) (*field-update d*) (*set ds*)
assumes *ind-ds*: *field-descs-independent ds*
shows *field-update d bs* (*fst* (*apply-field-updates ds bs*′ *s*)) =
fst (*apply-field-updates ds bs*′ (*field-update d bs s*))
using *ind-d ind-ds*
proof (*induct ds arbitrary*: *d bs bs*′ *s*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons d*′ *ds*′)
from *Cons.prems* **obtain**
ind-d1: *field-desc-independent* (*field-access d*) (*field-update d*) (*insert d*′ (*set ds*′)) **and**
ind-d′: *field-desc-independent* (*field-access d*′) (*field-update d*′) (*set ds*′) **and**
ind-d: *field-desc-independent* (*field-access d*) (*field-update d*) (*set ds*′) **and**
ind-ds′: *field-descs-independent ds*′
by (*auto intro*: *field-desc-independent-subset*)

from *ind-d1* **interpret** *fi*: *field-desc-independent field-access d field-update d insert d*′ (*set ds*′) **.**
from *fi.fu-commutes* [*of d*′]
have *fu-commutes* (*field-update d*) (*field-update d*′) **by** *simp*
then
have *d-d*′*-com*: *field-update d bs* (*field-update d*′ *bs*′ *s*) = (*field-update d*′ *bs*′ (*field-update d bs s*))
by (*auto simp add*: *fu-commutes-def*)

from *Cons.prems* **obtain**
ind-d′: *field-desc-independent* (*field-access d*′) (*field-update d*′) (*set ds*′) **and**
ind-d: *field-desc-independent* (*field-access d*) (*field-update d*) (*set ds*′) **and**
ind-ds′: *field-descs-independent ds*′
by (*auto intro*: *field-desc-independent-subset*)

note *hyp* = *Cons.hyps* [*OF ind-d ind-ds*′]

show *?case* **by** (*simp add*: *hyp d-d*′*-com*)

qed

lemma (**in** *wf-field-desc*) *is-padding-byte-acc-upd-compose*:
assumes *acc-upd*: $\bigwedge v\ s.\ acc\ (upd\ v\ s) = v$
shows *padding-base.is-padding-byte* (*field-access d* ∘ *acc*)
($\lambda bs\ v.\ upd\ (field\text{-}update\ d\ bs\ (acc\ v))\ v$) (*field-sz d*) *i* =
is-padding-byte i
apply (*simp add*: *is-padding-byte-def padding-base.is-padding-byte-def*)
by (*metis acc-upd*)

lemma (**in** *wf-field-desc*) *is-value-byte-acc-upd-compose*:
assumes *acc-upd*: $\bigwedge v\ s.\ acc\ (upd\ v\ s) = v$
shows *padding-base.is-value-byte* (*field-access d* ∘ *acc*)
($\lambda bs\ v.\ upd\ (field\text{-}update\ d\ bs\ (acc\ v))\ v$) (*field-sz d*) *i* =
is-value-byte i
apply (*simp add*: *is-value-byte-def padding-base.is-value-byte-def*)
by (*metis acc-upd*)

lemma (**in** *wf-field-desc*) *wf-field-desc-update-desc*:
assumes *double-update-upd*: $\bigwedge v\ w\ s.\ upd\ v\ (upd\ w\ s) = upd\ v\ s$
assumes *acc-upd*: $\bigwedge v\ s.\ acc\ (upd\ v\ s) = v$
assumes *upd-acc*: $\bigwedge s.\ upd\ (acc\ s)\ s = s$
shows *wf-field-desc* (*update-desc acc upd d*) (**is** *wf-field-desc ?upd*)
proof (*unfold-locales*)
fix *i*
assume *i-bound*: *i* < *field-sz* (*update-desc acc upd d*)
show *padding-base.is-padding-byte* (*field-access* (*update-desc acc upd d*))
(*field-update* (*update-desc acc upd d*))
(*field-sz* (*update-desc acc upd d*)) *i* ≠
padding-base.is-value-byte (*field-access* (*update-desc acc upd d*))
(*field-update* (*update-desc acc upd d*))
(*field-sz* (*update-desc acc upd d*)) *i*
using *complement i-bound*
by (*simp add*: *update-desc-def is-padding-byte-acc-upd-compose* [*of acc upd*, *OF acc-upd*]
is-value-byte-acc-upd-compose [*of acc upd*, *OF acc-upd*])
next
fix *bs bs′ s*
show *field-update ?upd bs* (*field-update ?upd bs′ s*) = *field-update ?upd bs s*
by (*simp add*: *update-desc-def double-update double-update-upd acc-upd*)
next
fix *bs s bs′ s′*
show *field-access ?upd* (*field-update ?upd bs s*) *bs′* = *field-access ?upd* (*field-update ?upd bs s′*) *bs′*
by (*simp add*: *update-desc-def access-update acc-upd*)
next
fix *s bs*
show *field-update ?upd* (*field-access ?upd s bs*) *s* = *s*
by (*simp add*: *update-desc-def update-access upd-acc*)

next
fix *s bs*
show (*field-access ?upd s* (*take* (*field-sz ?upd*) *bs*)) = *field-access ?upd s bs*
by (*simp add*: *update-desc-def access-size*)
next
fix *s bs*
show *length* (*field-access ?upd s bs*) = *field-sz ?upd*
by (*simp add*: *update-desc-def access-result-size*)
next
fix *bs*
show *field-update ?upd* (*take* (*field-sz ?upd*) *bs*) = *field-update ?upd bs*
by (*simp add*: *update-desc-def update-size*)
qed

lemma *toplevel-field-descs-subset*:
fixes *t*::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
set (*toplevel-field-descs t*) ⊆ *set* (*field-descs t*)
set (*toplevel-field-descs-struct st*) ⊆ *set* (*field-descs-struct st*)
set (*toplevel-field-descs-list fs*) ⊆ *set* (*field-descs-list fs*)
set (*toplevel-field-descs-tuple f*) ⊆ *set* (*field-descs-tuple f*)
by (*induct t* **and** *st* **and** *fs* **and** *f*) *auto*

lemma
fixes *t*::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
lf-fd ‘ *lf-set t xs* ⊆ *set* (*field-descs t*)
lf-fd ‘ *lf-set-struct st xs* ⊆ *set* (*field-descs-struct st*)
lf-fd ‘ *lf-set-list fs xs* ⊆ *set* (*field-descs-list fs*)
lf-fd ‘ *lf-set-tuple f xs* ⊆ *set* (*field-descs-tuple f*)
apply (*induction t* **and** *st* **and** *fs* **and** *f arbitrary*: *xs*) **apply** *auto*
oops

lemma *lf-set-subset-field-descs*:
fixes *t*::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*

shows
$\bigwedge xs.$ *lf-fd ‘ lf-set t xs* $\subseteq$ *set* (*field-descs t*)
$\bigwedge xs.$ *lf-fd ‘ lf-set-struct st xs* $\subseteq$ *set* (*field-descs-struct st*)
$\bigwedge xs.$ *lf-fd ‘ lf-set-list fs xs* $\subseteq$ *set* (*field-descs-list fs*)
$\bigwedge xs.$ *lf-fd ‘ lf-set-tuple f xs* $\subseteq$ *set* (*field-descs-tuple f*)
by (*induction t* **and** *st* **and** *fs* **and** *f*) (*auto simp add*: *image-subset-iff*)

lemma *wf-field-descs-empty-typ-info* [*simp*]: *wf-field-descs* (*set* (*field-descs* (*empty-typ-info algn t*)))
by (*simp add*: *empty-typ-info-def wf-field-descs-def*)

lemma *field-descs-map*:
field-descs (*map-td* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *t*) =
map (*update-desc acc upd*) (*field-descs t*)
field-descs-struct (*map-td-struct* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *st*) =
map (*update-desc acc upd*) (*field-descs-struct st*)
field-descs-list (*map-td-list* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *fs*) =
map (*update-desc acc upd*) (*field-descs-list fs*)
field-descs-tuple (*map-td-tuple* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *f*) =
map (*update-desc acc upd*) (*field-descs-tuple f*)
by (*induct t* **and** *st* **and** *fs* **and** *f*) *auto*

lemma *component-update-access*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs t*))
shows *field-update* (*component-desc t*)
(*field-access* (*component-desc t*) *s bs*) *s* = *s*
proof (*cases t*)
case (*TypDesc algn st n*)
show *?thesis*
proof (*cases st*)
case (*TypScalar sz align d*)
from *TypDesc TypScalar* **have** *d* ∈ *set* (*field-descs t*) **by** *simp*
with *wf* **have** *wf-field-desc d* **by** (*simp add*: *wf-field-descs-def*)
then interpret *wf-d*: *wf-field-desc d* **.**
from *TypDesc TypScalar wf-d.update-access* **show** *?thesis*
by *simp*
next
case (*TypAggregate fs*)
from *wf* **have** *wf-field-descs* (*set* (*field-descs-list fs*)) **by** (*simp add*: *TypDesc TypAggregate*)
then
have *fst* (*apply-field-updates* (*map dt-trd fs*)
(*concat* (*split-map* (*component-access-tuple s*) *fs bs*)) *s*) = *s*

```
    proof (induct fs arbitrary: s bs)
      case Nil
      then show ?case by simp
    next
      case (Cons f fs')

      obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

      from Cons f obtain
        wf-d: wf-field-desc d and
        wf-ft: wf-field-descs (set (field-descs ft)) and
        wf-fs': wf-field-descs (set (field-descs-list fs')) by auto


      from wf-d interpret wf-d: wf-field-desc d .
      from Cons.hyps [OF wf-fs']
      have fst (apply-field-updates (map dt-trd fs')
            (concat (split-map (component-access-tuple s) fs' (drop (field-sz d) bs)))
s) = s .
      then
      show ?case
        by (simp add: Cons f wf-d.access-result-size wf-d.update-access-append)
    qed
    with TypDesc TypAggregate show ?thesis by simp
  qed
qed

lemma toplevel-field-descs-tuple-dt-trd: toplevel-field-descs-tuple t = [dt-trd t]
  by (cases t) simp
lemma toplevel-field-descs-list-map: toplevel-field-descs-list fs = map dt-trd fs
  by (induct fs) (auto simp add: toplevel-field-descs-tuple-dt-trd)

lemma component-double-update:
  assumes wf:wf-field-descs (set (field-descs t))
  assumes ind: field-descs-independent (toplevel-field-descs t)
  showsfield-update (component-desc t) bs (field-update (component-desc t) bs' s)
=
       field-update (component-desc t) bs s
proof (cases t)
  case (TypDesc align st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp
    with wf have wf-field-desc d by (simp add: wf-field-descs-def)
    then interpret wf-d: wf-field-desc d .
    from TypDesc TypScalar wf-d.double-update show ?thesis
      by simp
  next
```

```
    case (TypAggregate fs)
   from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
   from ind have ind: field-descs-independent (toplevel-field-descs-list fs) by (simp
add: TypDesc TypAggregate)
    from wf ind
    have fst (apply-field-updates (map dt-trd fs) bs
           (fst (apply-field-updates (map dt-trd fs) bs' s))) =
         fst (apply-field-updates (map dt-trd fs) bs s)
    proof (induct fs arbitrary: s bs bs')
      case Nil
      then show ?case by simp
    next
      case (Cons f fs')
      obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

      from Cons.prems f obtain
        wf-d: wf-field-desc d and
        wf-ft: wf-field-descs (set (field-descs ft)) and
        wf-fs': wf-field-descs (set (field-descs-list fs')) and
        ind-d: field-desc-independent (field-access d) (field-update d)
          (set (toplevel-field-descs-list fs')) and
        ind-fs': field-descs-independent (toplevel-field-descs-list fs')
        by auto


      from wf-d interpret wf-d: wf-field-desc d .
      from field-update-apply-field-updates-commute [OF ind-d ind-fs']
      have d-ds'-commute: field-update d bs
            (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs')
                  (field-update d bs' s))) =
          fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs') (field-update
d bs s))
        by (simp add: toplevel-field-descs-list-map wf-d.double-update)


       note hyp = Cons.hyps [OF wf-fs' ind-fs', where bs=(drop (field-sz d) bs)
and s=(field-update d bs s)]
      show ?case
        by (simp add: Cons f wf-d.access-result-size d-ds'-commute hyp)
    qed
    then show ?thesis by (simp add: TypDesc TypAggregate)
  qed
qed


lemma component-access-update:
  assumes wf:wf-field-descs (set (field-descs t))
  assumes ind: field-descs-independent (toplevel-field-descs t)
```

```
  shows field-access (component-desc t) (field-update (component-desc t) bs s) bs'
=
       field-access (component-desc t) (field-update (component-desc t) bs s') bs'
proof (cases t)
  case (TypDesc algn st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp
    with wf have wf-field-desc d by (simp add: wf-field-descs-def)
    then interpret wf-d: wf-field-desc d .
    from TypDesc TypScalar wf-d.access-update show ?thesis
      by simp
  next
    case (TypAggregate fs)
   from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
   from ind have ind: field-descs-independent (toplevel-field-descs-list fs) by (simp
add: TypDesc TypAggregate)
    from wf ind
    have concat (split-map
          (component-access-tuple (fst (apply-field-updates (map dt-trd fs) bs s))) fs
bs') =
         concat (split-map
         (component-access-tuple (fst (apply-field-updates (map dt-trd fs) bs s'))) fs
bs')
    proof (induct fs arbitrary: bs s bs' s')
      case Nil
      then show ?case by simp
    next
      case (Cons f fs')
      obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

      from Cons.prems f obtain
        wf-d: wf-field-desc d and
        wf-ft: wf-field-descs (set (field-descs ft)) and
        wf-fs': wf-field-descs (set (field-descs-list fs')) and
        ind-d: field-desc-independent (field-access d) (field-update d)
          (set (toplevel-field-descs-list fs')) and
        ind-fs': field-descs-independent (toplevel-field-descs-list fs')
        by auto

      from wf-d interpret wf-d: wf-field-desc d .

      note hyp = Cons.hyps [OF wf-fs' ind-fs']
      from field-update-apply-field-updates-commute [OF ind-d ind-fs']
      have eq1: fst ((apply-field-updates (map dt-trd fs') (drop (field-sz d) bs))
                (field-update d bs s)) =
              field-update d bs
```

```
            (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs) s))
        (is - = ?rhs)
      by (simp add: toplevel-field-descs-list-map)
    have eq2: field-access d ?rhs bs' = field-access d (field-update d bs s) bs'
      by (simp add: wf-d.access-update)
    from field-update-apply-field-updates-commute [OF ind-d ind-fs']
    have eq3: fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs)
                (field-update d bs s')) =
              field-update d bs
               (fst (apply-field-updates (map dt-trd fs') (drop (field-sz d) bs) s'))
        (is - = ?rhs)
      by (simp add: toplevel-field-descs-list-map)
    have eq4: field-access d ?rhs bs' = field-access d (field-update d bs s) bs'
      by (simp add: wf-d.access-update)
    show ?case
      apply (simp add: f eq1 eq2 eq3 eq4 wf-d.access-result-size)
      apply (simp only: eq1 [symmetric] )
      apply (simp only: eq3 [symmetric] )
      apply (simp add: hyp)
      done
  qed

  then show ?thesis
    by (simp add: TypDesc TypAggregate)
 qed
qed

lemma sum-nat-foldl: (n::nat) + foldl (+) m xs = foldl (+) (n + m) xs
 by (induct xs arbitrary: n m) (auto simp add: semiring-normalization-rules(25))

lemma sum-nat-foldl-le: (n::nat) ≤ foldl (+) n xs
 by (induct xs arbitrary: n) (auto elim: add-leE)

lemma sum-add-sub-foldl: foldl (+) (m + n) ns - n = foldl (+) (m::nat) ns
 apply (induct ns arbitrary: m n)
  apply simp
 apply (metis diff-add-inverse2 semiring-normalization-rules(24) sum-nat-foldl)
 done

lemma sum-sub-zero-foldl: foldl (+) n ns - (n::nat) = foldl (+) 0 ns
 using sum-add-sub-foldl [where m=0, simplified]
 by simp

lemma drop-take-sub: drop n (take (foldl (+) n ns) bs) = take (foldl (+) 0 ns)
(drop n bs)
 by (simp add: drop-take sum-sub-zero-foldl)


lemma component-access-size:
```

```
  assumes wf: wf-field-descs (set (field-descs t))
  shows field-access (component-desc t) s (take (field-sz (component-desc t)) bs)
=
        field-access (component-desc t) s bs
proof (cases t)
  case (TypDesc algn st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp
    with wf have wf-field-desc d by (simp add: wf-field-descs-def)
    then interpret wf-d: wf-field-desc d .
    from TypDesc TypScalar wf-d.access-size show ?thesis
      by simp
  next
    case (TypAggregate fs)
   from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
    from wf
    have concat (split-map (component-access-tuple s) fs
           (take (foldl (+) 0 (map (field-sz ∘ dt-trd) fs)) bs)) =
         concat (split-map (component-access-tuple s) fs bs)
    proof (induct fs arbitrary: s bs)
      case Nil
      then show ?case by simp
    next
      case (Cons f fs')
      obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

      from Cons.prems f obtain
        wf-d: wf-field-desc d and
        wf-ft: wf-field-descs (set (field-descs ft)) and
        wf-fs': wf-field-descs (set (field-descs-list fs'))
        by auto

      from wf-d interpret wf-d: wf-field-desc d .

      have field-sz d ≤ foldl (+) (field-sz d) (map (λf'. field-sz (dt-trd f')) fs') (is
?n ≤ ?m)
        by (rule sum-nat-foldl-le)
      then
      have eq1: take ?n (take ?m bs) = take ?n bs
        by (simp add: min-def)

      from wf-d.access-size [where bs=take ?m bs]
      have field-access d s (take ?m bs) = field-access d s (take ?n (take ?m bs))
        by (simp)
      also have ... = field-access d s (take ?n bs) by (simp only: eq1)
      also have ... = field-access d s bs by (simp add: wf-d.access-size)
```

```
      finally
      have eq-head: field-access d s (take (foldl (+) (field-sz d) (map (λf'. field-sz
(dt-trd f')) fs')) bs) =
                    field-access d s bs .

      from Cons.hyps [OF wf-fs']
      show ?case
        by (simp add: f eq-head wf-d.access-result-size drop-take-sub)
    qed

    then show ?thesis
      by (simp add: TypDesc TypAggregate)
  qed
qed

lemma component-access-result-size:
  assumes wf: wf-field-descs (set (field-descs t))
  shows length (field-access (component-desc t) s bs) = field-sz (component-desc
t)
proof (cases t)
  case (TypDesc algn st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp
    with wf have wf-field-desc d by (simp add: wf-field-descs-def)
    then interpret wf-d: wf-field-desc d .
    from TypDesc TypScalar wf-d.access-result-size show ?thesis
      by simp
  next
    case (TypAggregate fs)
   from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
    from wf
    have length (concat (split-map (component-access-tuple s) fs bs)) =
          foldl (+) 0 (map (field-sz ∘ dt-trd) fs)
    proof (induct fs arbitrary: s bs)
      case Nil
      then show ?case by simp
    next
      case (Cons f fs')
      obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

      from Cons.prems f obtain
        wf-d: wf-field-desc d and
        wf-ft: wf-field-descs (set (field-descs ft)) and
        wf-fs': wf-field-descs (set (field-descs-list fs'))
        by auto
```

```
    from wf-d interpret wf-d: wf-field-desc d .

    note hyp = Cons.hyps [OF wf-fs']
    show ?case
      by (simp add: hyp wf-d.access-result-size f sum-nat-foldl)
   qed
   then show ?thesis by (simp add: TypDesc TypAggregate)
  qed
qed

lemma component-update-size:
  assumes wf: wf-field-descs (set (field-descs t))
  shows field-update (component-desc t) (take (field-sz (component-desc t)) bs) s
=
        field-update (component-desc t) bs s
proof (cases t)
  case (TypDesc align st n)
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from TypDesc TypScalar have d ∈ set (field-descs t) by simp
    with wf have wf-field-desc d by (simp add: wf-field-descs-def)
    then interpret wf-d: wf-field-desc d .
    from TypDesc TypScalar wf-d.update-size show ?thesis
      by simp
  next
    case (TypAggregate fs)
   from wf have wf: wf-field-descs (set (field-descs-list fs)) by (simp add: TypDesc
TypAggregate)
    then have fst (apply-field-updates (map dt-trd fs)
                (take (foldl (+) 0 (map (field-sz ∘ dt-trd) fs)) bs) s) =
              fst (apply-field-updates (map dt-trd fs) bs s)
    proof (induct fs arbitrary: s bs)
      case Nil
      then show ?case by simp
    next
      case (Cons f fs')
      obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

      from Cons.prems f obtain
        wf-d: wf-field-desc d and
        wf-ft: wf-field-descs (set (field-descs ft)) and
        wf-fs': wf-field-descs (set (field-descs-list fs'))
        by auto

      from wf-d interpret wf-d: wf-field-desc d .

      have field-sz d ≤ foldl (+) (field-sz d) (map (λa. field-sz (dt-trd a)) fs') (is
?n ≤ ?m)
```

```
      by (rule sum-nat-foldl-le)
    then
    have eq1: take ?n (take ?m bs) = take ?n bs
      by (simp add: min-def)

    from wf-d.update-size [where bs=take ?m bs]
    have field-update d ((take ?m) bs) s = field-update d (take ?n (take ?m bs))
s by simp
    also have ... = field-update d (take ?n bs) s by (simp only: eq1)
    also from wf-d.update-size [where bs=bs]
    have ... = field-update d bs s by simp
    finally have eq: field-update d ((take ?m) bs) s = field-update d bs s .


    note hyp = Cons.hyps [OF wf-fs', simplified comp-def]

    show ?case
      apply (simp add: f )
      apply (simp only: eq)
      apply (simp only: drop-take-sub)
      apply (simp only: hyp)
      done
  qed
  then show ?thesis by (simp add: TypDesc TypAggregate)
 qed
qed


lemma apply-field-updates-access-cancel:
 assumes wf: wf-field-descs (set (field-descs-list fs))
 assumes ind: field-descs-independent (toplevel-field-descs-list fs)
 shows fst (apply-field-updates (map dt-trd fs) (concat (split-map (component-access-tuple
v) fs bs)) v) = v
 using wf ind
proof (induct fs arbitrary: v bs)
 case Nil
 then show ?case by simp
next
 case (Cons f fs')
 obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp
 from Cons.prems f obtain
   wf-d: wf-field-desc d and
   wf-ft: wf-field-descs (set (field-descs ft)) and
   wf-fs': wf-field-descs (set (field-descs-list fs')) and
   ind-d: field-desc-independent (field-access d) (field-update d)
         (set (toplevel-field-descs-list fs')) and
   ind-fs': field-descs-independent (toplevel-field-descs-list fs')
   by auto
```

```
  from wf-d interpret wf-d: wf-field-desc d .

  from wf-d.update-size [of (field-access d v bs @
     concat (split-map (component-access-tuple v) fs' (drop (length (field-access d v
bs)) bs)))]
  have eq1: field-update d
           (field-access d v bs @
              concat
                (split-map (component-access-tuple v) fs'
                 (drop (length (field-access d v bs)) bs)))
         v = field-update d (field-access d v bs) v

    by  (simp add: wf-d.access-result-size)

  note hyp = Cons.hyps [OF wf-fs' ind-fs']
  show ?case apply (simp add: Cons.prems f)
    apply (simp add: eq1)
    apply (simp add: wf-d.access-result-size wf-d.update-access hyp)
    done
qed

lemma apply-field-updates-double:
  assumes wf: wf-field-descs (set (field-descs-list fs))
  assumes ind: field-descs-independent (toplevel-field-descs-list fs)
  shows fst (apply-field-updates (map dt-trd fs) bs
           (fst (apply-field-updates (map dt-trd fs) bs' v))) =
         fst (apply-field-updates (map dt-trd fs) bs v)
  using wf ind
proof (induct fs arbitrary: v bs bs')
  case Nil
  then show ?case by simp
next
  case (Cons f fs')
  obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp
  from Cons.prems f obtain
    wf-d: wf-field-desc d and
    wf-ft: wf-field-descs (set (field-descs ft)) and
    wf-fs': wf-field-descs (set (field-descs-list fs')) and
    ind-d: field-desc-independent (field-access d) (field-update d)
           (set (toplevel-field-descs-list fs')) and
    ind-fs': field-descs-independent (toplevel-field-descs-list fs')
    by auto

  note commute = field-update-apply-field-updates-commute [OF ind-d ind-fs', sim-
plified toplevel-field-descs-list-map]
  note hyp = Cons.hyps [OF wf-fs' ind-fs']
  from wf-d interpret wf-d: wf-field-desc d .
  show ?case apply (simp add: Cons.prems f)
    by (simp add: commute hyp wf-d.double-update)
```

qed

lemma *list-append-eq-split-conv*: *length xs1 = length xs2 ⟹ xs1 @ ys1 = xs2 @ ys2 ⟷ xs1 = xs2 ∧ ys1 = ys2*
by *auto*

lemma *component-complement-padding*:
assumes *wf*: *wf-field-descs* (*set* (*field-descs t*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs t*)
assumes *i-bound*: *i* < *field-sz* (*component-desc t*)
shows *padding-base.is-padding-byte* (*field-access* (*component-desc t*)) (*field-update* (*component-desc t*)) (*field-sz* (*component-desc t*)) *i* ≠ *padding-base.is-value-byte* (*field-access* (*component-desc t*)) (*field-update* (*component-desc t*)) (*field-sz* (*component-desc t*)) *i*
proof (*cases t*)
case (*TypDesc align st n*)
show *?thesis*
proof (*cases st*)
case (*TypScalar sz align d*)
from *TypDesc TypScalar* **have** *d* ∈ *set* (*field-descs t*) **by** *simp*
with *wf* **have** *wf-field-desc d* **by** (*simp add*: *wf-field-descs-def*)
then interpret *wf-d*: *wf-field-desc d* **.**
from *TypDesc TypScalar wf-d.complement* **show** *?thesis*
using *i-bound*
by *auto*
next
case (*TypAggregate fs*)
from *wf* **have** *wf*: *wf-field-descs* (*set* (*field-descs-list fs*)) **by** (*simp add*: *TypDesc TypAggregate*)
from *ind* **have** *ind*: *field-descs-independent* (*toplevel-field-descs-list fs*) **by** (*simp add*: *TypDesc TypAggregate*)
from *i-bound* **have** *i-bound′*: *i* < *foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) *fs*) **by** (*simp add*: *TypDesc TypAggregate*)
from *wf ind i-bound′*
have *padding-base.is-padding-byte* (*component-access-struct* (*TypAggregate fs*)) (*component-update-struct* (*TypAggregate fs*)) (*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) *fs*)) *i* = (¬ *padding-base.is-value-byte* (*component-access-struct* (*TypAggregate fs*)) (*component-update-struct* (*TypAggregate fs*)) (*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) *fs*)) *i*)
proof (*induct fs arbitrary*: *i*)
case *Nil*
then show *?case* **by** (*simp add*: *padding-base.is-padding-byte-def padding-base.is-value-byte-def*)
next
case (*Cons f fs′*)
obtain *ft fn d* **where** *f*: *f* = *DTuple ft fn d* **by** (*cases f*) *simp*
from *Cons.prems f* **obtain**
wf-d: *wf-field-desc d* **and**
wf-ft: *wf-field-descs* (*set* (*field-descs ft*)) **and**

wf-fs': *wf-field-descs* (*set* (*field-descs-list fs'*)) **and**
ind-d: *field-desc-independent* (*field-access d*) (*field-update d*)
(*set* (*toplevel-field-descs-list fs'*)) **and**
ind-fs': *field-descs-independent* (*toplevel-field-descs-list fs'*) **and**
i-bound: *i* < *foldl* (+) (*field-sz d*) (*map* (λ*a*. *field-sz* (*dt-trd a*)) *fs'*)
by (*auto*)

have *split-foldl*:
(*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs'*))) =
field-sz d + (*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) *fs'*))
by (*auto simp add*: *f sum-nat-foldl*)

from *wf-d* **interpret** *wf-d*: *wf-field-desc d* **.**

note *commute* = *field-update-apply-field-updates-commute* [*OF ind-d ind-fs'*, *symmetric*, *simplified toplevel-field-descs-list-map*]
note *hyp* = *Cons.hyps* [*OF wf-fs' ind-fs'*]
show *?case*
proof (*cases i* < *field-sz d*)
case *True*
note *i-in-d* = *this*
show *?thesis*
proof (*cases wf-d.is-padding-byte i*)
case *True*
note *is-padding-d* = *this*
with *True wf-d.complement i-in-d* **have** *not-value-d*: ¬ *wf-d.is-value-byte i* **by** *simp*
have *padding-base.is-padding-byte*
(*component-access-struct* (*TypAggregate* (*f* # *fs'*)))
(*component-update-struct* (*TypAggregate* (*f* # *fs'*)))
(*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs'*))) *i*
proof (*rule padding-base.is-padding-byteI*)
show *i* < *foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs'*))
using *i-bound*
by (*simp add*: *f comp-def*)
next
fix *v*::'*a* **and** *bs*::*byte list*
assume *i* < *foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs'*))
assume *length bs* = *foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs'*))
hence *lbs*: *length bs* = *foldl* (+) (*field-sz d*) (*map* (λ*x*. *field-sz* (*dt-trd x*)) *fs'*)
by (*simp add*: *f comp-def*)
hence *lbs'*: *field-sz d* ≤ *length bs*
using *sum-nat-foldl-le* **by** *auto*
hence *ltake*: *length* (*take* (*field-sz d*) *bs*) = *field-sz d* **by** *simp*
from *wf-d.access-size*
have *acc-d-take*: *field-access d v bs* = *field-access d v* (*take* (*field-sz d*) *bs*) **by** *simp*
from *wf-d.access-result-size* **have** *acc-d-sz*: *length* (*field-access d v bs*) =

```
field-sz d .
          have field-access d v bs ! i = bs ! i
            using i-in-d
         by (simp add: acc-d-take wf-d.access-result-size wf-d.is-padding-byte-acc-id
[OF is-padding-d ltake])
          then
          show component-access-struct (TypAggregate (f # fs')) v bs ! i = bs ! i
            using i-bound lbs i-in-d
            by (simp add: Cons f nth-append acc-d-sz)
        next
          fix v::'a and bs::byte list and b::byte
          assume i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
          assume length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
         hence lbs: length bs = foldl (+) (field-sz d) (map (λx. field-sz (dt-trd x))
fs')
            by (simp add: f comp-def)
          hence lbs': field-sz d ≤ length bs
            using sum-nat-foldl-le by auto
         with i-in-d have drop-eq: (drop (field-sz d) (bs[i := b])) = drop (field-sz
d) bs
            by (meson drop-update-cancel)
          from lbs' have ltake: length (take (field-sz d) bs) = field-sz d by simp
          have (field-update d (bs[i := b]) v) = field-update d bs v
            apply (subst (1 2) wf-d.update-size [symmetric])
            using wf-d.is-padding-byte-upd-eq [OF is-padding-d ltake] ltake
            by (metis take-update-swap)
          then
          show component-update-struct (TypAggregate (f # fs')) bs v =
             component-update-struct (TypAggregate (f # fs')) (bs[i := b]) v
            by (simp add: Cons f drop-eq)
        qed
        moreover
        {
          assume is-value: padding-base.is-value-byte
               (component-access-struct (TypAggregate (f # fs')))
               (component-update-struct (TypAggregate (f # fs')))
              (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
          have wf-d.is-value-byte i
          proof (rule wf-d.is-value-byteI)
            show i < field-sz d by (rule i-in-d)
          next
            fix v::'a and bs::byte list and bs'::byte list
            assume lbs: length bs = field-sz d
            assume lbs': length bs' = field-sz d
            obtain xbs where
              lxbs:length xbs = (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) and
              bs: bs = take (field-sz d) xbs and
              split-xbs: xbs = take (field-sz d) xbs @ drop (field-sz d) xbs
              using lbs append-eq-conv-conj
```

unfolding *split-foldl*
by (*metis* (*no-types*, *opaque-lifting*) *Ex-list-of-length length-append*)
obtain *xbs'* **where**
lxbs':*length xbs'* = (*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs'*))) **and**
bs': *bs'* = *take* (*field-sz d*) *xbs'* **and**
split-xbs': *xbs'* = *take* (*field-sz d*) *xbs'* @ *drop* (*field-sz d*) *xbs'*
using *lbs' append-eq-conv-conj*
unfolding *split-foldl*
by (*metis* (*no-types*, *opaque-lifting*) *Ex-list-of-length length-append*)
have *xbs-i*: *xbs* ! *i* = *bs* ! *i*
using *bs split-xbs'*
by (*simp add*: *bs i-in-d*)

have *upd-xbs*: *field-update d xbs v* = *field-update d bs v*
apply (*subst split-xbs*)
apply (*subst wf-d.update-size* [*symmetric*])
apply *simp*
apply (*simp add*: *bs* [*symmetric*])
done

from *commute* [**where** *bs'*=(*drop* (*field-sz d*) *xbs*) **and** *bs* = *bs* **and** *s*=*v*]
have *acc-upd*: *field-access d*
(*fst* (*apply-field-updates* (*map dt-trd fs'*) (*drop* (*field-sz d*) *xbs*)
(*field-update d bs v*)))
bs' = *field-access d* (*field-update d bs v*) *bs'*
using *wf-d.access-update* **by** *auto*

from *padding-base.is-value-byte-acc-upd-cancel* [*OF is-value lxbs lxbs'*]
have *component-access-struct* (*TypAggregate* (*f* # *fs'*))
(*component-update-struct* (*TypAggregate* (*f* # *fs'*)) *xbs v*) *xbs'* !
i = *xbs* ! *i* .
then
show *field-access d* (*field-update d bs v*) *bs'* ! *i* = *bs* ! *i*
using *lbs' lbs' i-in-d*
apply (*simp add*: *f xbs-i*)
apply (*subst* (*asm*) *wf-d.access-size* [*symmetric*])
apply (*simp add*: *bs'* [*symmetric*])
apply (*simp add*: *wf-d.access-result-size nth-append upd-xbs acc-upd*)
done

next
fix *v*::*'a* **and** *bs*::*byte list* **and** *b*::*byte*
assume *lbs*: *length bs* = *field-sz d*
obtain *xbs* **where**
lxbs:*length xbs* = (*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs'*))) **and**
bs: *bs* = *take* (*field-sz d*) *xbs* **and**
split-xbs: *xbs* = *take* (*field-sz d*) *xbs* @ *drop* (*field-sz d*) *xbs*

```
      using lbs append-eq-conv-conj
      unfolding split-foldl
      by (metis (no-types, opaque-lifting) Ex-list-of-length length-append)
    have eq1: field-access d v xbs = field-access d v bs
      using bs wf-d.access-size by simp
    have eq2: field-access d v (xbs[i := b]) = field-access d v (bs[i := b])
      using i-in-d bs
      by (metis take-update-swap wf-d.access-size)

    from padding-base.is-value-byte-acc-eq [OF is-value lxbs]
    have component-access-struct (TypAggregate (f # fs')) v xbs =
      component-access-struct (TypAggregate (f # fs')) v (xbs[i := b]) .
    then
    show field-access d v bs = field-access d v (bs[i := b])
      apply (simp add: f)
      apply (subst (asm) list-append-eq-split-conv)
       apply (simp add: wf-d.access-result-size)
      apply (clarsimp simp add: eq1 eq2)
      done
  qed
 }
 then
 have ¬ padding-base.is-value-byte
       (component-access-struct (TypAggregate (f # fs')))
       (component-update-struct (TypAggregate (f # fs')))
      (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
  using not-value-d
   by blast
 ultimately show ?thesis by blast
next
 case False
 note not-padding = this
 with wf-d.complement i-in-d have is-value: wf-d.is-value-byte i by simp
 have padding-base.is-value-byte
  (component-access-struct (TypAggregate (f # fs')))
  (component-update-struct (TypAggregate (f # fs')))
  (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
 proof (rule padding-base.is-value-byteI)
  show i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
    using i-bound by (simp add: f comp-def)
 next
  fix v::'a and bs::byte list and bs'::byte list
  assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
  assume lbs': length bs' = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
  from lbs have lbs-take: length (take (field-sz d) bs) = field-sz d
    using lbs split-foldl by (simp add: f comp-def)
  from lbs' have lbs'-take: length (take (field-sz d) bs') = field-sz d
    using lbs split-foldl by (simp add: f comp-def)
```

```
  show component-access-struct (TypAggregate (f # fs')) 
        (component-update-struct (TypAggregate (f # fs')) bs v) bs' !
        i = bs ! i
    apply (simp add: f)
    apply (subst wf-d.access-size [symmetric])
    using i-in-d
    apply (simp add: wf-d.access-result-size nth-append)
    apply (subst commute)
    apply (subst wf-d.access-update [where s'=v])
     using wf-d.is-value-byte-acc-upd-cancel [OF is-value lbs-take lbs'-take,
of v]
    apply (simp add: wf-d.update-size)
    done
 next
  fix v::'a and bs::byte list and b::byte
  assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
  from lbs have lbs-take: length (take (field-sz d) bs) = field-sz d
    using lbs split-foldl by (simp add: f comp-def)
  have acc-eq: field-access d v bs = field-access d v (bs[i:=b])
    apply (subst (1 2) wf-d.access-size [symmetric])
    using wf-d.is-value-byte-acc-eq [OF is-value lbs-take] i-in-d
    by (metis take-update-swap)

  show component-access-struct (TypAggregate (f # fs')) v bs =
    component-access-struct (TypAggregate (f # fs')) v (bs[i := b])
    apply (simp add: f acc-eq)
    using lbs-take i-in-d
    by (simp add: wf-d.access-result-size)
 qed
 moreover
 {
  assume is-padding: padding-base.is-padding-byte
    (component-access-struct (TypAggregate (f # fs')))
    (component-update-struct (TypAggregate (f # fs')))
    (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
  have wf-d.is-padding-byte i
  proof (rule wf-d.is-padding-byteI)
    from i-in-d show i < field-sz d .
  next
    fix v::'a and bs::byte list
    assume lbs: length bs = field-sz d
    obtain xbs where
     lxbs:length xbs = (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) and
      bs: bs = take (field-sz d) xbs and
      split-xbs: xbs = take (field-sz d) xbs @ drop (field-sz d) xbs
      using lbs append-eq-conv-conj
      unfolding split-foldl
      by (metis (no-types, opaque-lifting) Ex-list-of-length length-append)
     from padding-base.is-padding-byte-acc-eq [OF is-padding lxbs, where
```

v=v]

show *field-access d v bs* ! *i* = *bs* ! *i*
apply (*simp add*: *f*)
by (*simp add*: *bs i-in-d nth-append wf-d.access-result-size wf-d.access-size*)
next
fix *v*::′*a* **and** *bs*::*byte list* **and** *b*::*byte*
assume *lbs*: *length bs* = *field-sz d*
obtain *xbs* **where**
lxbs:*length xbs* = (*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs*′))) **and**
bs: *bs* = *take* (*field-sz d*) *xbs* **and**
split-xbs: *xbs* = *take* (*field-sz d*) *xbs* @ *drop* (*field-sz d*) *xbs*
using *lbs append-eq-conv-conj*
unfolding *split-foldl*
by (*metis* (*no-types*, *opaque-lifting*) *Ex-list-of-length length-append*)
from *padding-base.is-padding-byte-upd-eq* [*OF is-padding lxbs*, **where** *v=v*]
show *field-update d bs v* = *field-update d* (*bs*[*i* := *b*]) *v*
apply (*simp add*: *f*)
by (*metis bs commute take-update-swap wf-d.access-update wf-d.double-update wf-d.update-access wf-d.update-size-ext*)
qed
}
with *not-padding*
have ¬ *padding-base.is-padding-byte*
(*component-access-struct* (*TypAggregate* (*f* # *fs*′)))
(*component-update-struct* (*TypAggregate* (*f* # *fs*′)))
(*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) (*f* # *fs*′))) *i* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed
next
case *False*
note *i-notin-d* = *this*
with *i-bound* **obtain** *n j* **where**
j-bound: *j* < *foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) *fs*′) **and**
j: *j* = *i* − *field-sz d* **and**
i: *i* = *j* + *field-sz d*
apply (*subst* (*asm*) (*2*) *sum-nat-foldl* [**where** *m* = *0*, *simplified*, *symmetric*])
by (*fastforce simp add*: *comp-def*)
note *complement-fs*′*-j* = *hyp* [*OF j-bound*]
show *?thesis*
proof (*cases padding-base.is-padding-byte* (*component-access-struct* (*TypAggregate fs*′))
(*component-update-struct* (*TypAggregate fs*′))
(*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) *fs*′)) *j*)
case *True*
note *is-padding-j* = *this*
with *complement-fs*′*-j* **have** *not-is-value-j*:
¬ *padding-base.is-value-byte* (*component-access-struct* (*TypAggregate fs*′))

```
        (component-update-struct (TypAggregate fs'))
        (foldl (+) 0 (map (field-sz ∘ dt-trd) fs')) j by blast
    have padding-base.is-padding-byte
      (component-access-struct (TypAggregate (f # fs')))
      (component-update-struct (TypAggregate (f # fs')))
      (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
    proof (rule padding-base.is-padding-byteI)
      from i-bound show i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
by (simp add: f comp-def)
    next
      fix v::'a and bs::byte list
      assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
      from lbs have lbs-drop: length (drop (field-sz d) bs) = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs')
        apply (subst (asm) split-foldl)
        apply (simp add: f)
        done
      show component-access-struct (TypAggregate (f # fs')) v bs ! i = bs ! i
        apply (simp add: f)
        using i-notin-d i
        apply (simp add: nth-append wf-d.access-result-size)
        using padding-base.is-padding-byte-acc-eq [OF is-padding-j lbs-drop, of
v]
        apply simp
          by (metis add.commute lbs le-add-same-cancel1 nth-drop split-foldl
sum-nat-foldl-le)
    next
      fix v::'a and bs::byte list and b::byte
      assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
      from lbs have lbs-drop: length (drop (field-sz d) bs) = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs')
        apply (subst (asm) split-foldl)
        apply (simp add: f)
        done
      from i-notin-d
      have tk-eq: take (field-sz d) (bs[i := b]) = take (field-sz d) bs
        by (meson not-less take-update-cancel)
      have upd-eq: field-update d bs v = field-update d (bs[i := b]) v
        apply (subst (1 2) wf-d.update-size [symmetric])
        apply (simp add: tk-eq)
        done
      show component-update-struct (TypAggregate (f # fs')) bs v =
        component-update-struct (TypAggregate (f # fs')) (bs[i := b]) v
        apply (simp add: f)
        apply (simp add: upd-eq [symmetric])
          using padding-base.is-padding-byte-upd-eq [OF is-padding-j lbs-drop,
where v= (field-update d bs v) and b=b]
        apply (simp)
        by (metis drop-update-swap i j le-add2)
```

```
        qed
        moreover
        {
          assume is-value: padding-base.is-value-byte
            (component-access-struct (TypAggregate (f # fs')))
            (component-update-struct (TypAggregate (f # fs')))
            (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
          have padding-base.is-value-byte (component-access-struct (TypAggregate
fs'))
                (component-update-struct (TypAggregate fs'))
                (foldl (+) 0 (map (field-sz ∘ dt-trd) fs')) j
          proof (rule padding-base.is-value-byteI)
            from j-bound show j < foldl (+) 0 (map (field-sz ∘ dt-trd) fs') .
          next
            fix v::'a and bs::byte list and bs'::byte list
            assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) fs')
            assume lbs': length bs' = foldl (+) 0 (map (field-sz ∘ dt-trd) fs')
            obtain d-bs v-bs where v-bs: v-bs = field-access d v d-bs and
              l-v-bs: length v-bs = field-sz d
              using wf-d.access-result-size by auto
            obtain xbs where
              xbs-def: xbs = v-bs @ bs and
              xbs: xbs = take (field-sz d) xbs @ bs and
              lxbs: length xbs = field-sz d + length bs
              by (simp add: l-v-bs)
            obtain xbs' where
              xbs': xbs' = take (field-sz d) xbs' @ bs' and
              lxbs': length xbs' = field-sz d + length bs'
              by (metis Ex-list-of-length append-eq-conv-conj length-append)
            have xbs-i: xbs ! i = bs ! j
               by (metis add.commute append-take-drop-id i le-add2 lxbs nth-drop
same-append-eq xbs)
            have drop-xbs': drop (field-sz d) xbs' = bs'
              by (metis append-take-drop-id same-append-eq xbs')
            have drop-xbs: drop (field-sz d) xbs = bs
              by (metis append-take-drop-id same-append-eq xbs)
            have upd-xbs: field-update d xbs v = v
              by (simp add: v-bs wf-d.update-access-append xbs-def)
            show component-access-struct (TypAggregate fs')
                   (component-update-struct (TypAggregate fs') bs v) bs' !
                   j = bs ! j
              apply (simp )
              using padding-base.is-value-byte-acc-upd-cancel
                [OF is-value, where bs=xbs and bs'=xbs', simplified split-foldl,
                  OF lxbs [simplified lbs] lxbs' [simplified lbs'], where v = v]
               apply (simp add: xbs-i f nth-append wf-d.access-result-size i-notin-d
drop-xbs' drop-xbs
                  j [symmetric] upd-xbs)
              done
```

```
        next
          fix v::'a and bs::byte list and b::byte
          assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) fs')
          obtain d-bs v-bs where v-bs: v-bs = field-access d v d-bs and
            l-v-bs: length v-bs = field-sz d
            using wf-d.access-result-size by auto
          obtain xbs where
            xbs-def: xbs = v-bs @ bs and
            xbs: xbs = take (field-sz d) xbs @ bs and
            lxbs: length xbs = field-sz d + length bs
            by (simp add: l-v-bs)
          have xbs-i: xbs[i := b] = bs [j := b]

            by (metis calculation component-sz-struct-aggregate is-value l-v-bs lbs
len-gt-0
                      length-append less-irrefl padding-base.is-padding-byte-def
padding-base.is-value-byte-upd-neq
                split-foldl word-power-less-1 xbs-def)
          have drop-xbs: drop (length (field-access d v xbs)) xbs = bs
            by (simp add: l-v-bs wf-d.access-result-size xbs-def)

          show component-access-struct (TypAggregate fs') v bs =
            component-access-struct (TypAggregate fs') v (bs[j := b])
            apply (simp )
            using padding-base.is-value-byte-acc-eq [OF is-value, where bs=xbs,
simplified split-foldl,
                OF lxbs [simplified lbs], where v=v and b=b]
            apply (simp add: f)
            apply (subst (asm) list-append-eq-split-conv)
             apply (simp add: wf-d.access-result-size)
            apply (clarsimp simp add: drop-xbs)
            apply (simp add: xbs-i)
           by (metis (no-types, opaque-lifting) append-Nil append-eq-append-conv
append-eq-conv-conj
                l-v-bs length-list-update wf-d.access-result-size xbs-def xbs-i)
        qed
      }
      with not-is-value-j
      have ¬ padding-base.is-value-byte
        (component-access-struct (TypAggregate (f # fs')))
        (component-update-struct (TypAggregate (f # fs')))
        (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i by blast

      ultimately show ?thesis by blast
    next
      case False
      note not-is-padding-j = this
      with complement-fs'-j have is-value-j:
        padding-base.is-value-byte (component-access-struct (TypAggregate fs'))
```

```
        (component-update-struct (TypAggregate fs'))
        (foldl (+) 0 (map (field-sz ∘ dt-trd) fs')) j by blast
    have padding-base.is-value-byte
        (component-access-struct (TypAggregate (f # fs')))
        (component-update-struct (TypAggregate (f # fs')))
        (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
    proof (rule padding-base.is-value-byteI)
      from i-bound show i < foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
by (simp add: f comp-def)
    next
      fix v::'a and bs::byte list and bs'::byte list
      assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
      assume lbs': length bs' = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
      from lbs have lbs-drop: length (drop (field-sz d) bs) = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs')
        apply (subst (asm) split-foldl)
        apply (simp add: f)
        done
      from lbs' have lbs'-drop: length (drop (field-sz d) bs') = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs')
        apply (subst (asm) split-foldl)
        apply (simp add: f)
        done
      have drop-j: drop (field-sz d) bs ! j = bs ! i
        by (metis add.commute i lbs le-add-same-cancel1 nth-drop split-foldl
zero-le)
      show component-access-struct (TypAggregate (f # fs'))
          (component-update-struct (TypAggregate (f # fs')) bs v) bs' !
          i = bs ! i
           apply (simp add: f nth-append i-notin-d wf-d.access-result-size
j[symmetric])
        using padding-base.is-value-byte-acc-upd-cancel [OF is-value-j lbs-drop
lbs'-drop, where v = field-update d bs v]
        apply (simp add: drop-j)
        done
    next
      fix v::'a and bs::byte list and b::byte
      assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))
      from lbs have lbs-drop: length (drop (field-sz d) bs) = foldl (+) 0 (map
(field-sz ∘ dt-trd) fs')
        apply (subst (asm) split-foldl)
        apply (simp add: f)
        done
      have acc-eq: field-access d v bs = field-access d v (bs[i := b])
        by (metis i le-add2 take-update-cancel wf-d.access-size)
      show component-access-struct (TypAggregate (f # fs')) v bs =
          component-access-struct (TypAggregate (f # fs')) v (bs[i := b])
        apply (simp add: f)
        apply (subst list-append-eq-split-conv)
```

```
            apply (simp add: wf-d.access-result-size)
           apply (simp add: acc-eq)
           using padding-base.is-value-byte-acc-eq [OF is-value-j lbs-drop, where
v=v and b=b]
           apply simp
           by (metis drop-update-swap i j le-add2 wf-d.access-result-size)
         qed
         moreover
         {
          assume is-padding: padding-base.is-padding-byte
                   (component-access-struct (TypAggregate (f # fs')))
                   (component-update-struct (TypAggregate (f # fs')))
                   (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i
         have padding-base.is-padding-byte (component-access-struct (TypAggregate
fs'))
                  (component-update-struct (TypAggregate fs'))
                  (foldl (+) 0 (map (field-sz ∘ dt-trd) fs')) j
          proof (rule padding-base.is-padding-byteI)
           from j-bound show j < foldl (+) 0 (map (field-sz ∘ dt-trd) fs') .
          next
           fix v::'a and bs::byte list
           assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) fs')
           obtain d-bs v-bs where v-bs: v-bs = field-access d v d-bs and
             l-v-bs: length v-bs = field-sz d
             using wf-d.access-result-size by auto
           obtain xbs where
             xbs-def: xbs = v-bs @ bs and
             xbs: xbs = take (field-sz d) xbs @ bs and
             lxbs: length xbs = field-sz d + length bs
             by (simp add: l-v-bs)
           have drop-eq: (drop (field-sz d) xbs) = bs
             by (simp add: l-v-bs xbs-def)
           have xbs-i: xbs ! i = bs ! j
             by (simp add: i-notin-d j l-v-bs nth-append xbs-def)
           show component-access-struct (TypAggregate fs') v bs ! j = bs ! j
               using padding-base.is-padding-byte-acc-eq [OF is-padding, where
bs=xbs, simplified split-foldl,
                OF lxbs [simplified lbs], where v = v]
                apply (simp add: f nth-append wf-d.access-result-size i-notin-d j
[symmetric] drop-eq xbs-i)
             done
          next
           fix v::'a and bs::byte list and b::byte
           assume lbs: length bs = foldl (+) 0 (map (field-sz ∘ dt-trd) fs')
           obtain d-bs v-bs where v-bs: v-bs = field-access d v d-bs and
             l-v-bs: length v-bs = field-sz d
             using wf-d.access-result-size by auto
           obtain xbs where
             xbs-def: xbs = v-bs @ bs and
```

```
        xbs: xbs = take (field-sz d) xbs @ bs and
        lxbs: length xbs = field-sz d + length bs
        by (simp add: l-v-bs)
      have drop-eq: (drop (field-sz d) xbs) = bs
        by (simp add: l-v-bs xbs-def)

      have xbs-i: xbs [i := b] = bs [j := b]
          by (metis calculation is-padding l-v-bs lbs len-gt-0 length-append
less-irrefl
          padding-base.is-padding-byte-acc-neq padding-base.is-value-byte-def
split-foldl word-power-less-1 xbs-def)
      have upd-eq: (field-update d xbs v) = v
        by (simp add: v-bs wf-d.update-access-append xbs-def)
       have eq: fst (apply-field-updates (map dt-trd fs') (bs[j := b]) v) = fst
(apply-field-updates (map dt-trd fs') bs v)
     by (metis calculation is-padding lbs list-update-id lxbs padding-base.is-padding-byte-acc-neq
          padding-base.is-value-byte-acc-eq split-foldl)
        show component-update-struct (TypAggregate fs') bs v =
            component-update-struct (TypAggregate fs') (bs[j := b]) v
          using padding-base.is-padding-byte-upd-eq [OF is-padding, where
bs=xbs, simplified split-foldl,
          OF lxbs [simplified lbs], where v = v and b = b]
        apply (simp add: f drop-eq xbs-i upd-eq eq)
        done
      qed
     }
     with not-is-padding-j
     have ¬ padding-base.is-padding-byte
          (component-access-struct (TypAggregate (f # fs')))
          (component-update-struct (TypAggregate (f # fs')))
          (foldl (+) 0 (map (field-sz ∘ dt-trd) (f # fs'))) i by blast
     ultimately show ?thesis by blast
   qed
  qed
 qed

 then show ?thesis
  by (simp add: TypDesc TypAggregate)
qed
qed


theorem wf-field-desc-component-desc:
  assumes wf:wf-field-descs (set (field-descs t))
  assumes ind: field-descs-independent (toplevel-field-descs t)
  shows wf-field-desc (component-desc t)
  apply (unfold-locales)
      apply (rule component-complement-padding [OF wf ind], assumption)
     apply (rule component-double-update [OF wf ind])
```

```
    apply (rule component-access-update [OF wf ind])
   apply (rule component-update-access [OF wf])
  apply (rule component-access-size [OF wf])
 apply (rule component-access-result-size [OF wf])
apply (rule ext, rule component-update-size [OF wf])
done
```

```
lemma field-desc-list-append [simp]: field-descs-list (fs1 @ fs2) = field-descs-list
fs1 @ field-descs-list fs2
  by (induct fs1 arbitrary: fs2) auto
```

```
lemma toplevel-field-desc-list-append [simp]:
toplevel-field-descs-list (fs1 @ fs2) = toplevel-field-descs-list fs1 @ toplevel-field-descs-list
fs2
  by (induct fs1 arbitrary: fs2) auto

corollary wf-field-descs-struct-append-first:
  assumes wf:wf-field-descs (set (field-descs-struct (TypAggregate (fs1 @ fs2))))
   assumes ind: field-descs-independent (toplevel-field-descs-struct (TypAggregate
(fs1 @ fs2)))
  shows wf-field-desc (component-desc-struct (TypAggregate fs1))
proof −
  fix algn
  from wf have wf': wf-field-descs (set (field-descs (TypDesc algn (TypAggregate
fs1) [])))
    by simp
  from ind
 have ind': field-descs-independent (toplevel-field-descs (TypDesc algn (TypAggregate
fs1) []))
    by (simp add: field-descs-independent-append-first)
  from wf-field-desc-component-desc [OF wf' ind']
  show ?thesis
    by simp
qed

corollary wf-field-descs-struct-append-second:
  assumes wf:wf-field-descs (set (field-descs-struct (TypAggregate (fs1 @ fs2))))
   assumes ind: field-descs-independent (toplevel-field-descs-struct (TypAggregate
(fs1 @ fs2)))
  shows wf-field-desc (component-desc-struct (TypAggregate fs2))
proof −
  fix algn
  from wf have wf': wf-field-descs (set (field-descs (TypDesc algn (TypAggregate
fs2) [])))
    by simp
```

from *ind*
have *ind'*: *field-descs-independent* (*toplevel-field-descs* (*TypDesc align* (*TypAggregate fs2*) []))
by (*simp add*: *field-descs-independent-append-second*)
from *wf-field-desc-component-desc* [*OF wf' ind'*]
show *?thesis*
by *simp*
qed

corollary *wf-field-descs-component-desc*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs t*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs t*)
shows *wf-field-descs* (*insert* (*component-desc t*) (*set* (*field-descs t*)))
using *wf-field-desc-component-desc* [*OF wf ind*] *wf*
by *simp*

lemma *size-td-field-sz*:
fixes
t::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
wf-component-descs t ⟹ *size-td t* = *field-sz* (*component-desc t*)
wf-component-descs-struct st ⟹ *size-td-struct st* = *field-sz* (*component-desc-struct st*)
wf-component-descs-list fs ⟹ *size-td-list fs* = *field-sz* (*component-desc-struct* (*TypAggregate fs*))
wf-component-descs-tuple f ⟹ *size-td-tuple f* = *field-sz* (*dt-trd f*)
proof (*induct t* **and** *st* **and** *fs* **and** *f*)
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply (*cases dt-tuple*)
apply (*clarsimp simp add*: *sum-nat-foldl*)
done
qed *auto*

lemma (**in** *c-type*) *simple-type-to-xto-eq*:
fixes *v*::'*a*
assumes *simple*: ¬ *aggregate* (*typ-info-t TYPE*('*a*))
shows *to-bytes v* = *xto-bytes v*
proof −
from *simple-type-dest* [*OF simple*]
show *?thesis*
by (*clarsimp simp add*: *xto-bytes-def to-bytes-def*)
qed

lemma *field-desc-independent-empty* [*simp*]: *field-desc-independent acc upd* {}
by (*simp add*: *field-desc-independent-def*)

lemma *component-descs-field-descs-independent*:
component-descs-independent t ⟹ *field-descs-independent* (*toplevel-field-descs t*)
apply (*cases t*)
subgoal for *x1 st nm*
apply (*cases st*)
apply (*simp*)
subgoal for *fs*
apply (*cases fs*)
apply *simp*
apply *simp*
done
done
done

lemma *component-descs-list-field-descs-independent*:
component-descs-independent-list fs ⟹ *field-descs-independent* (*toplevel-field-descs-list fs*)
apply (*cases fs*)
apply *simp*
apply *simp*
done

theorem *access-ti-component-desc-compatible*:
⋀*bs v*::′*a*. ⟦*wf-component-descs t*; *component-descs-independent t*;
wf-field-descs (*set* (*field-descs t*))⟧
⟹ *access-ti t v bs* = *field-access* (*component-desc t*) *v bs*
⋀*bs v*::′*a*. ⟦*wf-component-descs-struct st*; *component-descs-independent-struct st*;
wf-field-descs (*set* (*field-descs-struct st*))⟧
⟹ *access-ti-struct st v bs* = *field-access* (*component-desc-struct st*) *v bs*
⋀*bs v*::′*a*. ⟦*wf-component-descs-list fs*; *component-descs-independent-list fs*;
wf-field-descs (*set* (*field-descs-list fs*))⟧
⟹ *access-ti-list fs v bs* = *field-access* (*component-desc-struct* (*TypAggregate fs*)) *v bs*
⋀*bs v*::′*a*. ⟦*wf-component-descs-tuple f*; *component-descs-independent-tuple f*;
wf-field-descs (*set* (*field-descs* (*dt-fst f*)))⟧
⟹ *access-ti-tuple f v bs* = *field-access* (*dt-trd f*) *v bs*
proof (*induct t* **and** *st* **and** *fs* **and** *f*)
case (*TypDesc algn st n*)
then show *?case* **by** *simp*
next
case (*TypScalar sz align d*)

```
  then show ?case by simp
next
  case (TypAggregate fs)
  then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc f fs')
  obtain ft fn d where f: f = DTuple ft fn d by (cases f) simp

  from Cons-typ-desc.prems obtain
    wf-descs-f: wf-component-descs-tuple f and wf-descs-fs': wf-component-descs-list
fs' and
    ind: field-descs-independent (toplevel-field-descs-tuple f @ toplevel-field-descs-list
fs') and
      ind-cf: component-descs-independent-tuple f and
      ind-cfs': component-descs-independent-list fs' and
      wf-field-desc-f: wf-field-descs (set (field-descs (dt-fst f))) and
      wf-d: wf-field-desc d and wf-ft: wf-field-descs (set (field-descs ft)) and
      wf-field-descs-fs': wf-field-descs (set (field-descs-list fs'))
    by (auto simp add: f)


  note ind-fs' = field-descs-independent-append-second [OF ind]

  from wf-d interpret wf-d: wf-field-desc d.


  have wf-fd-fs': wf-field-descs (insert (component-desc-struct (TypAggregate fs'))
        (set (field-descs-list fs')))
  proof −
    fix algn
    from wf-field-descs-fs'
    have wf:wf-field-descs (set (field-descs (TypDesc algn (TypAggregate fs') [])))
      by simp
    from ind-fs'
   have ind: field-descs-independent (toplevel-field-descs (TypDesc algn (TypAggregate
fs') []))
      by simp
    from wf-field-descs-component-desc [OF wf ind] show ?thesis
      by simp
  qed


  from Cons-typ-desc.hyps(1) [OF wf-descs-f ind-cf wf-field-desc-f ]
  have eq-f: access-ti-tuple f v (take (size-td ft) bs) =
            field-access (dt-trd f) v (take (size-td ft) bs).
  from Cons-typ-desc.hyps(2) [OF wf-descs-fs' ind-cfs' wf-field-descs-fs' ]
```

have *eq-fs'*: *access-ti-list fs' v* (*drop* (*size-td ft*) *bs*) =
field-access (*component-desc-struct* (*TypAggregate fs'*)) *v*
(*drop* (*size-td ft*) *bs*) **.**

from *wf-descs-f* **have** *sz-eq*: *field-sz d* = *size-td ft* **by** (*simp add*: *f size-td-field-sz*)

from *sz-eq wf-d.access-size*
have *eq-access-take*: *field-access d v* (*take* (*size-td ft*) *bs*) = *field-access d v bs*
by (*simp*)

from *wf-descs-f* **have** *sz-eq*: *field-sz d* = *size-td ft* **by** (*simp add*: *f size-td-field-sz*)

from *eq-f eq-fs' eq-access-take*
show *?case* **by** (*simp add*: *f wf-d.access-result-size size-td-field-sz sz-eq*)
next
case (*DTuple-typ-desc ft fn d*)
from *DTuple-typ-desc.prems* **obtain**
d: *d* = *component-desc ft* **and**
wf-ft: *wf-component-descs ft* **and**
ind-cf: *component-descs-independent ft* **and**
wf-descs-ft: *wf-field-descs* (*set* (*field-descs ft*))
by *auto*

from *wf-field-descs-component-desc* [*OF wf-descs-ft*] *ind-cf*
have *wf-desc-ft*: *wf-field-desc* (*component-desc ft*)
by (*auto intro*: *component-descs-field-descs-independent*)
from *component-descs-field-descs-independent* [*OF ind-cf*]
have *field-descs-independent* (*toplevel-field-descs ft*) **.**

from *DTuple-typ-desc.hyps* [*OF wf-ft ind-cf wf-descs-ft*]

show *?case* **by** (*simp add*: *d*)
qed

theorem *update-ti-component-desc-compatible*:
⋀*bs v*::'*a*. ⟦*wf-component-descs t*; *component-descs-independent t*;
wf-field-descs (*set* (*field-descs t*))⟧
⟹ *update-ti t bs v* = *field-update* (*component-desc t*) *bs v*
⋀*bs v*::'*a*. ⟦*wf-component-descs-struct st*; *component-descs-independent-struct st*;
wf-field-descs (*set* (*field-descs-struct st*))⟧
⟹ *update-ti-struct st bs v* = *field-update* (*component-desc-struct st*) *bs v*
⋀*bs v*::'*a*. ⟦*wf-component-descs-list fs*; *component-descs-independent-list fs*;
wf-field-descs (*set* (*field-descs-list fs*))⟧
⟹ *update-ti-list fs bs v* = *field-update* (*component-desc-struct* (*TypAggregate fs*)) *bs v*
⋀*bs v*::'*a*. ⟦*wf-component-descs-tuple f*; *component-descs-independent-tuple f*;
wf-field-descs (*set* (*field-descs* (*dt-fst f*)))⟧
⟹ *update-ti-tuple f bs v* = *field-update* (*dt-trd f*) *bs v*

proof (*induct t* **and** *st* **and** *fs* **and** *f*)
case (*TypDesc algn st n*)
then show *?case* **by** *simp*
next
case (*TypScalar sz align d*)
then show *?case* **by** *simp*
next
case (*TypAggregate fs*)
then show *?case* **by** *simp*
next
case *Nil-typ-desc*
then show *?case* **by** *simp*
next
case (*Cons-typ-desc f fs'*)
obtain *ft fn d* **where** *f*: *f* = *DTuple ft fn d* **by** (*cases f*) *simp*

from *Cons-typ-desc.prems* **obtain**
wf-descs-f: *wf-component-descs-tuple f* **and** *wf-descs-fs'*: *wf-component-descs-list fs'* **and**
ind: *field-descs-independent* (*toplevel-field-descs-tuple f* @ *toplevel-field-descs-list fs'*) **and**
ind-cf: *component-descs-independent-tuple f* **and**
ind-cfs': *component-descs-independent-list fs'* **and**
ind-d: *field-desc-independent* (*field-access d*) (*field-update d*) (*set* (*toplevel-field-descs-list fs'*)) **and**
wf-field-desc-f: *wf-field-descs* (*set* (*field-descs* (*dt-fst f*))) **and**
wf-d: *wf-field-desc d* **and** *wf-ft*: *wf-field-descs* (*set* (*field-descs ft*)) **and**
wf-field-descs-fs': *wf-field-descs* (*set* (*field-descs-list fs'*))
by (*auto simp add*: *f*)

note *ind-fs'* = *field-descs-independent-append-second* [*OF ind*]

from *wf-d* **interpret** *wf-d*: *wf-field-desc d*.

from *Cons-typ-desc.hyps*(*2*) [*OF wf-descs-fs' ind-cfs' wf-field-descs-fs'*]
have *eq-fs'*: *update-ti-list fs'* (*drop* (*size-td ft*) *bs*) *v* =
fst (*apply-field-updates* (*map dt-trd fs'*) (*drop* (*size-td ft*) *bs*) *v*) (**is** *?upd-ti-fs'* = *?upd-fs'*)
by *simp*

from *Cons-typ-desc.hyps*(*1*) [*OF wf-descs-f ind-cf wf-field-desc-f*]
have *eq-f*: *update-ti ft* (*take* (*size-td ft*) *bs*) *?upd-ti-fs'* =
field-update d (*take* (*size-td ft*) *bs*) *?upd-fs'* **by** (*simp add*: *f eq-fs'*)
also
from *field-update-apply-field-updates-commute* [*OF ind-d ind-fs'*, **where** *bs*= (*take* (*size-td ft*) *bs*) **and** *bs'*=(*drop* (*size-td ft*) *bs*)]
have ... = *fst* (*apply-field-updates* (*toplevel-field-descs-list fs'*) (*drop* (*size-td ft*) *bs*)

```
            (field-update d (take (size-td ft) bs) v))
    by (simp add: toplevel-field-descs-list-map)
  finally have eq-ft: update-ti ft (take (size-td ft) bs) ?upd-ti-fs' =
    fst (apply-field-updates (toplevel-field-descs-list fs') (drop (size-td ft) bs)
          (field-update d (take (size-td ft) bs) v)) .

  from size-td-field-sz(1) [of ft] wf-descs-f
  have sz-d: field-sz d = size-td ft
    by (simp add: f)


  show ?case
    apply (simp add: f )
    apply (simp add: eq-ft toplevel-field-descs-list-map wf-d.update-size [simplified
sz-d] sz-d)
    done
next
  case (DTuple-typ-desc ft fn d)
  then show ?case by simp
qed


context xmem-type
begin


lemma wf-field-desc-component-desc: wf-field-desc (component-desc (typ-info-t (TYPE('a))))
  using wf-field-descs component-descs-independent
 by (auto intro: wf-field-desc-component-desc component-descs-field-descs-independent)

lemma wf-field-descs': wf-field-descs (insert (component-desc (typ-info-t (TYPE('a))))
                          (set (field-descs (typ-info-t (TYPE('a)))))))
  using wf-field-descs
  by (auto intro: wf-field-desc-component-desc)

lemma wf-field-desc-t: wf-field-desc (component-desc (typ-info-t (TYPE('a))))
  using wf-field-descs' by (simp)

sublocale xmem-type-wf-field-desc: wf-field-desc component-desc (typ-info-t (TYPE('a)))
  by (rule wf-field-desc-t)

lemma xfrom-bytes-xto-bytes-inv: xfrom-bytes (xto-bytes (v::'a) bs) = v
proof −
  define t where t = (typ-info-t TYPE('a))
  have wf-comp-descs-t: wf-component-descs t by (simp add: wf-component-descs
t-def)
 have ind-t: component-descs-independent t by (simp add: component-descs-independent
t-def)
  from wf-field-descs
```

have *wf-field-descs-t*: *wf-field-descs* (*set* (*field-descs t*))
by (*simp add*: *t-def*)
have *xto*: *xto-bytes v bs* = *field-access* (*component-desc t*) *v bs* (**is** (- = *?acc*))
by (*simp add*: *xto-bytes-def t-def*)

from *wf-field-desc-component-desc* **have** *wf-field-desc* (*component-desc t*) **by** (*simp add*: *t-def*)
then interpret *wf-t*: *wf-field-desc component-desc t*.
from *wf-t.access-result-size*
have *len-acc*: *length ?acc* = *field-sz* (*component-desc t*) .
from *size-td-field-sz*(*1*) [*OF wf-comp-descs-t*]
have *size-t*: *size-td t* = *field-sz* (*component-desc t*).
have *len-acc'*: *size-td t* = *length ?acc* **by** (*simp add*: *len-acc size-t*)
from *size-t len-acc*
have *len-acc''*: *length ?acc* = *size-of TYPE*('*a*) **by** (*simp add*: *size-of-def t-def*)
from *len-acc''* *len-acc'*
have *len-acc'''*: *size-of TYPE*('*a*) = *size-td* (*typ-info-t TYPE*('*a*)) **by** (*simp add*: *t-def*)

from *update-ti-component-desc-compatible*(*1*) [*OF wf-comp-descs-t ind-t wf-field-descs-t*]
have *field-update* (*component-desc t*) (*field-access* (*component-desc t*) *v bs*) *undefined* =
update-ti t (*field-access* (*component-desc t*) *v bs*) *undefined* **by** *simp*
also
from *upd* [*rule-format*, *OF len-acc''*, *folded t-def*, **where** *v*= *undefined* **and** *w* = *v*] *len-acc' t-def*
have ... = *update-ti t* (*field-access* (*component-desc t*) *v bs*) *v* **by** (*simp add*: *update-ti-t-def*)
also
from *update-ti-component-desc-compatible*(*1*) [*OF wf-comp-descs-t ind-t wf-field-descs-t*]
have ... = *field-update* (*component-desc t*) (*field-access* (*component-desc t*) *v bs*) *v* .
also
from *wf-t.update-access*
have ... = *v* .
finally
have *field-update* (*component-desc t*) (*field-access* (*component-desc t*) *v bs*) *undefined* = *v* .

with *len-acc* **show** *?thesis*
by (*simp add*: *xto xfrom-bytes-def t-def*)
qed

lemma *xto-bytes-inj*:
xto-bytes (*v*::'*a*) = *xto-bytes* (*v'*::'*a*) ⟹ *v* = *v'*
apply (*drule fun-cong* [**where** *x*=*replicate* (*size-of TYPE*('*a*)) *0*])
apply (*drule arg-cong* [**where** *f*=*xfrom-bytes*::*byte list* ⇒ '*a*])

apply (*simp add*: *xfrom-bytes-xto-bytes-inv*)
done

lemma *field-update-update-ti-t*:
assumes *len*: *length bs* = *size-td* (*typ-info-t* (*TYPE*('*a*)))
shows *field-update* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) *bs* = *update-ti-t* (*typ-info-t* (*TYPE*('*a*))) *bs*
proof −
from *len*
update-ti-component-desc-compatible(*1*) [*OF wf-component-descs component-descs-independent wf-field-descs*]
show *?thesis*
by (*simp add*: *update-ti-t-def fun-eq-iff*)
qed

lemma *field-update-update-ti*:
shows *field-update* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) = *update-ti* (*typ-info-t* (*TYPE*('*a*)))
proof −
from
update-ti-component-desc-compatible(*1*) [*OF wf-component-descs component-descs-independent wf-field-descs*]
show *?thesis*
by (*simp add*: *fun-eq-iff*)
qed

lemma *field-access-access-ti*:
shows *field-access* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) = *access-ti* (*typ-info-t* (*TYPE*('*a*)))
proof −
from
access-ti-component-desc-compatible(*1*) [*OF wf-component-descs component-descs-independent wf-field-descs*]
show *?thesis*
by (*simp add*: *fun-eq-iff*)
qed

lemma *field-sz-size-td*:
field-sz (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) = *size-td* (*typ-info-t* (*TYPE*('*a*)))
by (*simp add*: *local.wf-component-descs size-td-field-sz*(*1*))

lemma *field-sz-size-of*:
field-sz (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) = *size-of* (*TYPE*('*a*))
by (*simp add*: *size-of-def field-sz-size-td*)

lemma *xto-bytes-size*: *xto-bytes* (*v*::'*a*) (*take* (*size-td* (*typ-info-t TYPE*('*a*))) *bs*) = *xto-bytes v bs*

proof −
from *xmem-type-wf-field-desc.access-size size-td-field-sz*(*1*) [*OF wf-component-descs*]
show *?thesis*
by (*simp add*: *xto-bytes-def*)
qed

lemma *xto-bytes-result-size*: *length* (*xto-bytes* (*v*::′*a*) *bs*) = *size-td* (*typ-info-t TYPE*(′*a*))
proof −
from *xmem-type-wf-field-desc.access-result-size size-td-field-sz*(*1*) [*OF wf-component-descs*]
show *?thesis*
by (*simp add*: *xto-bytes-def*)
qed

lemma *xfrom-bytes-size*: (*xfrom-bytes*::*byte list* ⇒ ′*a*) (*take* (*size-td* (*typ-info-t TYPE*(′*a*))) *bs*) = *xfrom-bytes bs*
proof −
from *xmem-type-wf-field-desc.update-size* [**where** *bs*=*bs*] *size-td-field-sz*(*1*) [*OF wf-component-descs*]
show *?thesis*
by (*simp add*: *xfrom-bytes-def*)
qed

lemma *entire-update*:
shows *field-update* (*component-desc* (*typ-info-t* (*TYPE*(′*a*)))) *bs v* = *field-update* (*component-desc* (*typ-info-t* (*TYPE*(′*a*)))) *bs w*
by (*metis xmem-type-wf-field-desc.double-update xfrom-bytes-def xfrom-bytes-xto-bytes-inv*)

lemma *update-access-complete*:
shows *field-update* (*component-desc* (*typ-info-t* (*TYPE*(′*a*)))) (*field-access* (*component-desc* (*typ-info-t* (*TYPE*(′*a*)))) *v bs*) *w* = *v*
by (*metis entire-update xmem-type-wf-field-desc.update-access*)

end

lemma *contained-field-descs-list-all*: *contained-field-descs-list xs* = *list-all contained-field-descs-tuple xs*
by (*induct xs*) *auto*

lemma *toplevel-field-descs-map*:
toplevel-field-descs (*map-td* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *t*)
=
map (*update-desc acc upd*) (*toplevel-field-descs t*)
toplevel-field-descs-struct (*map-td-struct* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *st*) =
map (*update-desc acc upd*) (*toplevel-field-descs-struct st*)

toplevel-field-descs-list (*map-td-list* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *fs*) =
map (*update-desc acc upd*) (*toplevel-field-descs-list fs*)
toplevel-field-descs-tuple (*map-td-tuple* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *f*) =
map (*update-desc acc upd*) (*toplevel-field-descs-tuple f*)
by (*induct t* **and** *st* **and** *fs* **and** *f*) *auto*

lemma *toplevel-field-descs-adjust-ti*: *toplevel-field-descs* (*adjust-ti t acc upd*) = *map* (*update-desc acc upd*) (*toplevel-field-descs t*)
by (*simp add*: *adjust-ti-def toplevel-field-descs-map*)

theorem *contained-field-descs-empty-typ-info* [*simp*]: *contained-field-descs* (*empty-typ-info algn n*)
by (*simp add*: *empty-typ-info-def*)

lemma *fields-contained-update-desc*:
assumes *contained-t*: *fields-contained* (*field-access d*) (*field-update d*) (*set* (*toplevel-field-descs t*))
shows *fields-contained* (*field-access* (*update-desc acc upd d*))
(*field-update* (*update-desc acc upd d*))
(*update-desc acc upd* ‘ *set* (*toplevel-field-descs t*))
proof (*rule fields-contained.intro*)
fix *d′ s s′*
assume *d′-in*: *d′* ∈ *update-desc acc upd* ‘ *set* (*toplevel-field-descs t*)
assume *acc-contained*: *field-access* (*update-desc acc upd d*) *s* =
field-access (*update-desc acc upd d*) *s′*

interpret *ft*: *fields-contained* (*field-access d*) (*field-update d*) *set* (*toplevel-field-descs t*)
by (*rule contained-t*)

from *acc-contained* **have** *acc*: *field-access d* (*acc s*) = *field-access d* (*acc s′*)
by (*simp add*: *update-desc-def*)

from *d′-in* **obtain** *d″* **where**
d″-in: *d″* ∈ *set* (*toplevel-field-descs t*) **and**
d′: *d′* = *update-desc acc upd d″*
by *auto*

from *ft.access-contained* [*OF d″-in acc*]
have *field-access d″* (*acc s*) = *field-access d″* (*acc s′*)**.**
with *d′*
show *field-access d′ s* = *field-access d′ s′*
by (*simp add*: *update-desc-def*)
next
fix *d′ bs s*
assume *d′-in*: *d′* ∈ *update-desc acc upd* ‘ *set* (*toplevel-field-descs t*)

interpret *ft*: *fields-contained* (*field-access d*) (*field-update d*) *set* (*toplevel-field-descs t*)
by (*rule contained-t*)

from *d′-in* **obtain** *d″* **where**
d″-in: $d'' \in$ *set* (*toplevel-field-descs t*) **and**
d′: $d' =$ *update-desc acc upd d″*
by *auto*
from *ft.update-contained* [*OF d″-in*]
obtain *bs′* **where** *cont*: $\forall\, s'.$ *field-access d* (*acc s′*) = *field-access d* (*acc s*) $\longrightarrow$
field-update d″ bs (*acc s′*) = *field-update d bs′* (*acc s′*)
by *fastforce*

from *cont*
show $\exists\, bs'.\ \forall\, s'.$ *field-access* (*update-desc acc upd d*) $s' =$
field-access (*update-desc acc upd d*) $s \longrightarrow$ *field-update d′ bs s′* =
field-update (*update-desc acc upd d*) *bs′ s′*
by (*auto simp add*: *update-desc-def d′*)
qed

lemma *contained-field-descs-update-desc*:
contained-field-descs t $\Longrightarrow$
contained-field-descs (*map-td* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *t*)
contained-field-descs-struct st $\Longrightarrow$
contained-field-descs-struct (*map-td-struct* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *st*)
contained-field-descs-list fs $\Longrightarrow$
contained-field-descs-list (*map-td-list* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *fs*)
contained-field-descs-tuple f $\Longrightarrow$
contained-field-descs-tuple (*map-td-tuple* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *f*)
proof (*induct t* **and** *st* **and** *fs* **and** *f*)
case (*TypDesc st n*)
then show *?case* **by** *simp*
next
case (*TypScalar n sz d*)
then show *?case* **by** *simp*
next
case (*TypAggregate fs*)
then show *?case* **by** *simp*
next
case *Nil-typ-desc*
then show *?case* **by** *simp*
next
case (*Cons-typ-desc f fs*)

```
  then show ?case by simp
next
  case (DTuple-typ-desc ft fn d)
  note ‹contained-field-descs-tuple (DTuple ft fn d)›
  then obtain
    contained-ft: contained-field-descs ft and
   fields-contained-ft: fields-contained (field-access d) (field-update d) (set (toplevel-field-descs
ft))
    by simp
  from contained-ft DTuple-typ-desc(1)
  have contained-map: contained-field-descs
     (map-td (λ- -. update-desc acc upd) (update-desc acc upd) ft)
    by simp

  from fields-contained-update-desc [OF fields-contained-ft]
  have fields-contained (field-access (update-desc acc upd d))
     (field-update (update-desc acc upd d))
     (update-desc acc upd ‘ set (toplevel-field-descs ft)) .

  then have fields-contained (field-access (update-desc acc upd d))
         (field-update (update-desc acc upd d))
         (set (toplevel-field-descs (map-td (λ- -. update-desc acc upd) (update-desc
acc upd) ft)))
    by (simp add: toplevel-field-descs-map)
  with DTuple-typ-desc show ?case by simp
qed

theorem contained-field-descs-adjust-ti:
  fixes t::'a xtyp-info
  assumes contained-t: contained-field-descs t
  shows contained-field-descs (adjust-ti t acc upd)
  using contained-field-descs-update-desc(1) [OF contained-t]
  by (simp add: adjust-ti-def)

theorem contained-field-descs-extend-ti:
  assumes contained-t: contained-field-descs t
  assumes contained-ft: contained-field-descs ft
  assumes fields-contained-ft: fields-contained (field-access d) (field-update d) (set
(toplevel-field-descs ft))
  shows contained-field-descs (extend-ti t ft algn n d)
  using contained-t contained-ft fields-contained-ft
  apply (cases t)
  subgoal for x1 x2 x3
    apply  simp
    apply (cases x2)
     apply simp
    apply (simp add: contained-field-descs-list-all)
    done
  done
```

theorem *contained-field-descs-ti-pad-combine*:
fixes *t*::*'a xtyp-info*
assumes *cont-t*: *contained-field-descs t*
shows *contained-field-descs* (*ti-pad-combine n t*)
proof −
define *d*::*'a field-desc* **where**
d ≡ (|*field-access* = λ*v bs*. *if n* ≤ *length bs then take n bs else replicate n 0*,
field-update = λ*bs*. *id*, *field-sz* = *n*|)
define *ft*::*'a xtyp-info* **where** *ft* ≡ *TypDesc 0* (*TypScalar n 0 d*) *''!pad-typ''*
define *fn* **where** *fn* ≡ (*foldl* (@) *''!pad-''* (*field-names-list t*))
have *contained-field-descs* (*extend-ti t ft 0 fn d*)
by (*rule contained-field-descs-extend-ti* [*OF cont-t*]) (*auto simp add*: *ft-def d-def fields-contained-def*)

then show *?thesis*
by (*simp add*: *ti-pad-combine-def Let-def d-def ft-def fn-def padding-desc-def*)
qed

lemma *contained-field-descs-simp*[*simp*]:
contained-field-descs (*map-align f t*) = *contained-field-descs t*
by (*cases t*) (*simp*)

theorem *contained-field-descs-final-pad*:
fixes *t*::*'a xtyp-info*
assumes *cont-t*: *contained-field-descs t*
shows *contained-field-descs* (*final-pad algn t*)
using *cont-t*
by (*simp add*: *final-pad-def contained-field-descs-ti-pad-combine Let-def*)

lemma (**in** *xmem-type*) *fields-contained-update-desc-mem-type*:
fixes *acc*:: *'b* ⇒ *'a*
fixes *D*:: *'a field-desc set*
shows *fields-contained* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*) (*update-desc acc upd* ‘ *D*)
proof (*rule fields-contained.intro*)
fix *d s s'*
assume *d-in*: *d* ∈ *update-desc acc upd* ‘ *D*
assume *to-btyes-eq*: (*xto-bytes* ∘ *acc*) *s* = (*xto-bytes* ∘ *acc*) *s'*

hence *acc s* = *acc s'*
by (*auto intro*: *xto-bytes-inj*)

with *d-in*
show *field-access d s* = *field-access d s'*
by (*auto simp add*: *update-desc-def*)
next
fix *d bs s*
assume *d-in*: *d* ∈ *update-desc acc upd* ‘ *D*

from *d-in* **obtain** d' **where** *d'-in*: $d' \in D$ **and** *d*: $d =$ *update-desc acc upd* d'
by *auto*

define x **where** $x \equiv \lambda s.$ *field-update* d' *bs* (*acc* s)

from *xfrom-bytes-xto-bytes-inv* **obtain** bs' **where**
cont: $\forall s'.$ *acc* $s' =$ *acc* $s \longrightarrow$ *xfrom-bytes* (*xto-bytes* (x s') bs') = (x s')
by *auto*

from *d'-in cont*
show $\exists bs'.\ \forall s'.$ (*xto-bytes* $\circ$ *acc*) $s' =$ (*xto-bytes* $\circ$ *acc*) $s \longrightarrow$
field-update d bs $s' =$ (*upd* $\circ$ *xfrom-bytes*) bs' s'
by (*metis* (*no-types*, *lifting*) *comp-apply d field-desc.simps*(2) *update-desc-def x-def xto-bytes-inj*)
qed

locale *wf-field* =
fixes *acc*::'*s* ⇒ '*a*
fixes *upd*::'*a* ⇒ '*s* ⇒ '*s*
assumes *double-update-upd*: $\bigwedge v\ w\ s.$ *upd* v (*upd* w s) = *upd* v s
assumes *acc-upd*: $\bigwedge v\ s.$ *acc* (*upd* v s) = v
assumes *upd-acc*: $\bigwedge s.$ *upd* (*acc* s) $s = s$

locale *wf-cfield* = *wf-field* +
constrains *acc*::'*s* ⇒ '*a*::*c-type*
locale *wf-xfield* = *wf-cfield* +
constrains *acc*::'*s* ⇒ '*a*::*xmem-type*

lemma (**in** *wf-field*) *wf-field-descs-adjust-ti*:
assumes *wf-t*: *wf-field-descs* (*set* (*field-descs t*))
shows *wf-field-descs* (*set* (*field-descs* (*adjust-ti t acc upd*)))
proof −
have *wf-field-descs* (*update-desc acc upd* ‘ *set* (*field-descs t*))
proof (*rule wf-field-descs.intro*)
fix *d*
assume $d \in$ *update-desc acc upd* ‘ *set* (*field-descs t*)
then obtain d' **where** *d'-in*: $d' \in$ *set* (*field-descs t*) **and** *d*: $d =$ *update-desc acc upd* d'
by *auto*
from *wf-t d'-in* **have** *wf-field-desc* d'
by (*rule wf-field-descs.wf-desc*)
then interpret *wf-d'*: *wf-field-desc* d' **.**

```
    from wf-d'.wf-field-desc-update-desc [of upd acc, OF double-update-upd acc-upd
upd-acc]
    have wf-field-desc (update-desc acc upd d') .
    then
    show wf-field-desc d
      by (simp add: d)
  qed
  then show ?thesis
    by (simp add: adjust-ti-def field-descs-map)
qed



lemma field-descs-list-append [simp]: field-descs-list (fs @ fs') = field-descs-list fs
@ field-descs-list fs'
  by (induct fs arbitrary: fs') auto

lemma wf-field-descs-extend-ti:
  assumes wf-t: wf-field-descs (set (field-descs t))
  assumes wf-ft: wf-field-descs (set (field-descs ft))
  assumes wf-d: wf-field-desc d
  shows wf-field-descs (set (field-descs (extend-ti t ft algn fn d)))
proof (cases t)
  case (TypDesc algn st n)
  note desc = this
  show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    with wf-ft wf-d show ?thesis
      by (simp add: desc)
  next
    case (TypAggregate fs)
    note aggr = this
    show ?thesis
    proof (cases fs)
      case Nil
      then show ?thesis
        using wf-ft wf-d by (simp add: desc aggr)
    next
      case (Cons f fs')
      note cons = this
      obtain ft' fn fd where f: f = DTuple ft' fn fd by (cases f)
      show ?thesis
        using wf-ft wf-t wf-d by (auto simp add: desc aggr cons f wf-field-descs-def)
    qed
  qed
qed
```

lemma *padding-pad*: *complement-padding* ($\lambda v\ bs.$ *if* $n \leq$ *length bs then take n bs else replicate n 0*) ($\lambda bs.$ *id*) *n*
apply (*unfold-locales*)
by (*auto simp add*: *padding-base.is-padding-byte-def padding-base.is-value-byte-def*)

(*metis* (*mono-tags*, *opaque-lifting*) *Ex-list-of-length len8 len-not-eq-0 less-irrefl neq0-conv nth-list-update-eq word-power-less-1*)

lemma *wf-field-desc-pad*:
wf-field-desc
(|*field-access* = $\lambda v\ bs.$ *if* $n \leq$ *length bs then take n bs else replicate n 0*, *field-update* = $\lambda bs.$ *id*, *field-sz* = n|)
apply (*intro-locales*)
apply *simp*
apply (*rule padding-pad*)
apply (*unfold-locales*)
apply *auto*
done

lemma *wf-field-descs-ti-pad-combine*: *wf-field-descs* (*set* (*field-descs t*)) $\Longrightarrow$
wf-field-descs (*set* (*field-descs* (*ti-pad-combine n t*)))
apply (*simp add*: *ti-pad-combine-def Let-def*)
apply (*rule wf-field-descs-extend-ti*)
apply (*auto simp add*: *wf-field-desc-pad padding-desc-def*)
done

lemma *set-field-descs-map-align*[*simp*]: *set* (*field-descs* (*map-align f t*)) = *set* (*field-descs t*)
by (*cases t*) *simp*

lemma *wf-field-descs-final-pad*: *wf-field-descs* (*set* (*field-descs t*)) $\Longrightarrow$
wf-field-descs (*set* (*field-descs* (*final-pad algn t*)))
apply (*clarsimp simp add*: *final-pad-def Let-def*)
apply (*rule wf-field-descs-ti-pad-combine*)
apply *simp*
done

lemma (**in** *wf-xfield*) *padding-lift*:
shows *complement-padding* (*xto-bytes* $\circ$ *acc*) (*upd* $\circ$ *xfrom-bytes*) (*size-of TYPE*('*a*))
proof −

thm *xmem-type-wf-field-desc.access-update*
thm *xmem-type-wf-field-desc.is-padding-byte-acc-upd-compose*
show *?thesis*
proof

```
    fix i
    assume i-bound: i < size-of TYPE('a)

    {
      fix bs v
      have upd (field-update (component-desc (typ-info-t TYPE('a))) bs (acc v)) v
=
          upd (field-update (component-desc (typ-info-t TYPE('a))) bs undefined) v
           by (simp add: entire-update [where bs = bs and v = acc v and w =
undefined])
    } note tweak-eq = this

   have sz-eq: field-sz (component-desc (typ-info-t TYPE('a))) = size-of TYPE('a)
      by (simp add: size-of-def size-td-field-sz(1) wf-component-descs)

   from xmem-type-wf-field-desc.is-padding-byte-acc-upd-compose [where acc=acc
and upd=upd, OF acc-upd, of i]
        xmem-type-wf-field-desc.is-value-byte-acc-upd-compose [where acc=acc and
upd=upd, OF acc-upd, of i]
    show padding-base.is-padding-byte (xto-bytes ∘ acc) (upd ∘ xfrom-bytes)
          (size-of TYPE('a)) i ≠
         padding-base.is-value-byte (xto-bytes ∘ acc) (upd ∘ xfrom-bytes)
          (size-of TYPE('a)) i
       apply (simp add: xto-bytes-def xfrom-bytes-def [abs-def] comp-def tweak-eq
sz-eq)
      using xmem-type-wf-field-desc.complement i-bound sz-eq
       by metis
  qed
qed




lemma (in wf-xfield) wf-field-desc-lift:
  shows wf-field-desc (|field-access = xto-bytes ∘ acc,
         field-update = upd ∘ xfrom-bytes,
         field-sz = size-of (TYPE('a))|) (is wf-field-desc ?lift)
  apply (intro-locales, simp, rule padding-lift, unfold-locales)
proof −
  fix bs bs' s
  show field-update ?lift bs (field-update ?lift bs' s) = field-update ?lift bs s
    by (simp add: double-update-upd)
next
  fix bs s bs' s'
  show field-access ?lift (field-update ?lift bs s) bs' = field-access ?lift (field-update
?lift bs s') bs'
    by (simp add: acc-upd)
next
  fix s bs
```

show *field-update ?lift* (*field-access ?lift s bs*) *s* = *s*
by (*simp add*: *upd-acc xfrom-bytes-xto-bytes-inv*)
next
fix *s bs*
show *field-access ?lift s* (*take* (*field-sz ?lift*) *bs*) = *field-access ?lift s bs*
by (*simp add*: *xto-bytes-size size-of-def*)
next
fix *s bs*
show *length* (*field-access ?lift s bs*) = *field-sz ?lift*
by (*simp add*: *xto-bytes-result-size size-of-def*)
next
fix *bs*
show *field-update ?lift* (*take* (*field-sz ?lift*) *bs*) = *field-update ?lift bs*
by (*simp add*: *xfrom-bytes-size size-of-def*)
qed

lemma (**in** *wf-xfield*) *wf-field-descs-ti-typ-combine*:
assumes *wf-ft*: *wf-field-descs* (*set* (*field-descs ft*))
shows *wf-field-descs* (*set* (*field-descs* (*ti-typ-combine* (*TYPE*('*a*)) *acc upd algn fn ft*)))
apply (*simp add*: *ti-typ-combine-def*)
apply (*rule wf-field-descs-extend-ti*)
apply (*rule wf-ft*)
apply (*rule wf-field-descs-adjust-ti*)
apply (*rule wf-field-descs*)
apply (*rule wf-field-desc-lift*)
done

lemma (**in** *wf-xfield*) *wf-field-descs-ti-typ-pad-combine*:
assumes *wf-ft*: *wf-field-descs* (*set* (*field-descs ft*))
shows *wf-field-descs* (*set* (*field-descs* (*ti-typ-pad-combine* (*TYPE*('*a*)) *acc upd algn fn ft*)))
proof (*cases* 0 < *padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*a*))) (*size-td ft*))
case *True*
then show *?thesis*
apply (*simp add*: *ti-typ-pad-combine-def Let-def*)
apply (*rule wf-field-descs-ti-typ-combine*)
apply (*rule wf-field-descs-ti-pad-combine*)
apply (*rule wf-ft*)
done
next
case *False*
then show *?thesis*
apply (*simp add*: *ti-typ-pad-combine-def Let-def*)
apply (*rule wf-field-descs-ti-typ-combine*)
apply (*rule wf-ft*)

```
    done
qed

lemma wf-component-descs-empty-typ-info [simp]: wf-component-descs (empty-typ-info
algn n)
  by (simp add: empty-typ-info-def)

lemma wf-component-descs-list-append [simp]:
  wf-component-descs-list (fs @ fs') = (wf-component-descs-list fs ∧ wf-component-descs-list
fs')
  by (induct fs arbitrary: fs') auto

lemma wf-component-descs-extend-ti:
  assumes wf-t: wf-component-descs t
  assumes wf-ft: wf-component-descs ft
  assumes d: d = (component-desc ft)
  shows wf-component-descs (extend-ti t ft algn fn d)
  apply (cases t)
  subgoal for x1 st n
    apply (cases st)
     apply (simp add: wf-ft d)
    subgoal for fs
      apply (cases fs)
       apply (simp add: wf-ft d)
      subgoal for f fs'
        apply (cases f)
        apply (insert wf-t)
        apply (simp add: d wf-ft)
        done
      done
    done
  done

lemma wf-component-descs-ti-pad-combine:
  assumes wf-t: wf-component-descs t
  shows wf-component-descs (ti-pad-combine n t)
  apply (simp add: ti-pad-combine-def Let-def padding-desc-def)
  apply (rule wf-component-descs-extend-ti)
    apply (rule wf-t)
   apply simp
  apply simp
  done


lemma wf-component-descs-map-align [simp]: wf-component-descs (map-align f t)
= wf-component-descs t
  by (cases t) simp

lemma wf-component-descs-final-pad:
```

assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs* (*final-pad algn t*)
by (*clarsimp simp add*: *final-pad-def Let-def wf-t wf-component-descs-ti-pad-combine*)

lemma *field-sz-update-desc* [*simp*]: *field-sz* (*update-desc acc upd d*) = *field-sz d*
by (*simp add*: *update-desc-def*)

lemma *component-sz-struct-update-desc-commutes*:
(*component-sz-struct* (*TypAggregate fs*)) =
component-sz-struct (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) (*TypAggregate fs*))
proof (*induct fs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons f fs′*)
obtain *ft fn d* **where** *f*: *f* = *DTuple ft fn d* **by** (*cases f*) *simp*
have *map-eq*: *map* (*field-sz* ∘ *dt-trd*) (*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *fs′*) =
map (*field-sz* ∘ *dt-trd*) *fs′*
apply (*induct fs′*)
apply *simp*
subgoal for *f′ fs″*
apply (*cases f′*)
apply *clarsimp*
done
done
show *?case* **by** (*simp add*: *f map-eq*)
qed

lemma *component-access-struct-update-desc-commutes′*:
component-access-struct (*TypAggregate fs*) (*acc v*) *bs* =
component-access-struct (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) (*TypAggregate fs*)) *v bs*
proof (*induct fs arbitrary*: *bs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons f fs′*)
obtain *ft fn d* **where** *f*: *f* = *DTuple ft fn d* **by** (*cases f*) *simp*

have *d-commutes*: *field-access* (*update-desc acc upd d*) *v bs* = *field-access d* (*acc v*) *bs*
by (*simp add*: *update-desc-def*)

note *hyp* = *Cons.hyps* [**where** *bs*=(*drop* (*length* (*field-access d* (*acc v*) *bs*)) *bs*)]
then

show *?case*
by (*simp add*: *f d-commutes hyp*)
qed

lemma *component-access-struct-update-desc-commutes*:
(*component-access-struct* (*TypAggregate fs*)) ∘ *acc* =
component-access-struct (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) (*TypAggregate fs*))
apply −
apply (*rule ext*)
apply (*rule ext*)
apply (*simp only*: *comp-def*)
apply (*rule component-access-struct-update-desc-commutes′*)
done

lemma (**in** *wf-field*) *lemma-component-update-struct-update-desc-commutes*:
shows *upd* (*component-update-struct* (*TypAggregate fs*) *bs* (*acc v*)) *v* =
component-update-struct (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) (*TypAggregate fs*)) *bs v*
proof (*induct fs arbitrary*: *v bs*)
case *Nil*
then show *?case* **by** (*simp add*: *upd-acc*)
next
case (*Cons f fs′*)
obtain *ft fn d* **where** *f*: *f* = *DTuple ft fn d* **by** (*cases f*) *simp*

have *acc-update-desc*: *acc* (*field-update* (*update-desc acc upd d*) *bs v*) = *field-update d bs* (*acc v*)
by (*simp add*: *update-desc-def acc-upd*)

have *upd-eq*: *upd* (*fst* (*apply-field-updates* (*map dt-trd fs′*) (*drop* (*field-sz d*) *bs*) (*field-update d bs* (*acc v*)))) (*field-update* (*update-desc acc upd d*) *bs v*) =
upd (*fst* (*apply-field-updates* (*map dt-trd fs′*) (*drop* (*field-sz d*) *bs*) (*field-update d bs* (*acc v*)))) *v*
by (*simp add*: *update-desc-def double-update-upd*)

note *hyp* = *Cons.hyps* [**where** *bs* = (*drop* (*field-sz d*) *bs*) **and** *v*=*field-update* (*update-desc acc upd d*) *bs v*]

from *hyp*
show *?case*
by (*simp add*: *f acc-update-desc upd-eq*)
qed

lemma (**in** *wf-field*) *update-desc-component-desc-struct-commutes*:
shows *update-desc acc upd* (*component-desc-struct* (*TypAggregate fs*)) =

component-desc-struct (*map-td-struct* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) (*TypAggregate fs*))
using *lemma-component-update-struct-update-desc-commutes*
component-access-struct-update-desc-commutes [**where** *acc*=*acc* **and** *upd*=*upd* **and** *fs*=*fs*]
component-sz-struct-update-desc-commutes [**where** *acc*=*acc* **and** *upd*=*upd* **and** *fs*=*fs*]
apply (*simp only*: *update-desc-def*)
apply (*fastforce simp add*: *component-desc-struct-def*)
done

lemma (**in** *wf-field*) *update-desc-component-desc-commutes*:
shows *update-desc acc upd* (*component-desc t*) =
component-desc (*map-td* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *t*)
using *update-desc-component-desc-struct-commutes*
apply (*cases t*)
apply *simp*
subgoal for *x1 st n*
apply (*cases st*)
apply *simp*
apply *simp*
done
done

lemma (**in** *wf-field*) *wf-component-descs-map-td-update-desc*:
shows
wf-component-descs t
⟹ *wf-component-descs* (*map-td* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *t*)
wf-component-descs-struct st
⟹ *wf-component-descs-struct* (*map-td-struct* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *st*)
wf-component-descs-list fs
⟹ *wf-component-descs-list* (*map-td-list* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *fs*)
wf-component-descs-tuple f
⟹ *wf-component-descs-tuple* (*map-td-tuple* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *f*)
apply (*induct t* **and** *st* **and** *fs* **and** *f*)
apply (*auto simp add*: *update-desc-component-desc-commutes*)
done

lemma (**in** *wf-field*) *wf-component-descs-adjust-ti*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs* (*adjust-ti t acc upd*)
using *wf-component-descs-map-td-update-desc*(*1*) [*OF wf-t*]
by (*simp add*: *adjust-ti-def*)

lemma (**in** *wf-xfield*) *update-desc-component-desc*:
shows
update-desc acc upd (*component-desc* (*typ-info-t TYPE*('*a*))) =
(|*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*a*)|)
proof −
have *wf*: *wf-component-descs* (*typ-info-t TYPE*('*a*)) **by** (*rule wf-component-descs*)
have *eq*: ⋀*bs v*. *field-update* (*component-desc* (*typ-info-t TYPE*('*a*))) *bs* (*acc v*)
=
field-update (*component-desc* (*typ-info-t TYPE*('*a*))) *bs undefined*
by (*simp add*: *entire-update*)
show *?thesis*
using *size-td-field-sz*(*1*) [*OF wf*]
apply (*simp add*: *xto-bytes-def size-of-def update-desc-def*)
apply (*rule ext*)
apply (*rule ext*)
apply (*simp add*: *xfrom-bytes-def eq*)
done
qed

lemma (**in** *wf-xfield*) *wf-component-descs-ti-typ-combine*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs* (*ti-typ-combine ft acc upd algn fn t*)
proof −
have *wf-ft*: *wf-component-descs* (*typ-info-t TYPE*('*a*)) **by** (*rule wf-component-descs*)
show *?thesis*
apply (*simp add*: *ti-typ-combine-def Let-def*)
apply (*rule wf-component-descs-extend-ti*)
apply (*rule wf-t*)
apply (*rule wf-component-descs-adjust-ti*)
apply (*rule wf-ft*)
apply (*simp add*: *adjust-ti-def Let-def update-desc-component-desc*
update-desc-component-desc-commutes[*symmetric*])
done
qed

lemma (**in** *wf-xfield*) *wf-component-descs-ti-typ-pad-combine*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs* (*ti-typ-pad-combine ft acc upd algn fn t*)
apply (*cases* *0* < *padup* (*max* (*2* ^ *algn*) (*align-of TYPE*('*a*))) (*size-td t*))
apply (*simp add*: *ti-typ-pad-combine-def Let-def*)
apply (*rule wf-component-descs-ti-typ-combine*)
apply (*rule wf-component-descs-ti-pad-combine* [*OF wf-t*])
apply (*simp add*: *ti-typ-pad-combine-def Let-def*)
apply (*rule wf-component-descs-ti-typ-combine* [*OF wf-t*])
done

lemma *component-descs-independent-empty-typ-info* [*simp*]:
component-descs-independent (*empty-typ-info align n*)
by (*simp add*: *empty-typ-info-def*)

lemma *component-descs-independent-list-appendI*:
assumes *ind-xs-ys*: ∀ *x* ∈ *set* (*toplevel-field-descs-list xs*).
field-desc-independent (*field-access x*) (*field-update x*) (*set* (*toplevel-field-descs-list ys*))
assumes *ind-xs*: *component-descs-independent-list xs*
assumes *ind-ys*: *component-descs-independent-list ys*
shows *component-descs-independent-list* (*xs* @ *ys*)
using *ind-xs-ys ind-xs ind-ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
obtain *ft′ fn′ d′* **where** *x*: *x* = *DTuple ft′ fn′ d′* **by** (*cases x*) *simp*

from *Cons.prems* **obtain**

find-d′-ys: *field-desc-independent* (*field-access d′*) (*field-update d′*)
(*set* (*toplevel-field-descs-list ys*)) **and**
find-xs-ys: ∀ *x*∈*set* (*toplevel-field-descs-list xs*).
field-desc-independent (*field-access x*) (*field-update x*) (*set* (*toplevel-field-descs-list ys*)) **and**
find-d′-xs: *field-desc-independent* (*field-access d′*) (*field-update d′*) (*set* (*toplevel-field-descs-list xs*)) **and**
find-xs: *field-descs-independent* (*toplevel-field-descs-list xs*) **and**
ind-ft′: *component-descs-independent ft′* **and**
ind-ys: *component-descs-independent-list ys* **and**
ind-xs:*component-descs-independent-list xs*
by (*clarsimp simp add*: *x*)

from *find-d′-xs find-d′-ys*
have *find-d′*: *field-desc-independent* (*field-access d′*) (*field-update d′*)
(*set* (*toplevel-field-descs-list xs*) ∪ *set* (*toplevel-field-descs-list ys*))
by (*simp add*: *field-desc-independent-union-iff*)

from *component-descs-list-field-descs-independent* [*OF ind-xs*]
have *tl-xs*: *field-descs-independent* (*toplevel-field-descs-list xs*).
from *component-descs-list-field-descs-independent* [*OF ind-ys*]
have *tl-ys*: *field-descs-independent* (*toplevel-field-descs-list ys*).

from *field-descs-independent-appendI1* [*OF tl-xs tl-ys find-xs-ys*]
have *tl-xs-ys*: *field-descs-independent*
(*toplevel-field-descs-list xs* @ *toplevel-field-descs-list ys*) .

```
  from Cons.hyps [OF find-xs-ys ind-xs ind-ys]
  have ind-xs-ys: component-descs-independent-list (xs @ ys) .
  from find-d' tl-xs-ys ind-ft' ind-xs-ys
  show ?case
    by (simp add: x)
qed

lemma component-descs-independent-extend-ti:
  assumes ind-t: component-descs-independent t
  assumes ind-ft: component-descs-independent ft
 assumes ind-d-t: field-desc-independent (field-access d) (field-update d) (set (toplevel-field-descs
t))
  shows component-descs-independent (extend-ti t ft align fn d)
proof (cases t)
  case (TypDesc align st nn)
  then show ?thesis
  proof (cases st)
    case (TypScalar sz align d)
    from ind-ft show ?thesis
      by (simp add: TypDesc TypScalar)
  next
    case (TypAggregate fs)
    then show ?thesis
    proof (cases fs)
      case Nil
      from ind-ft show ?thesis
        by (simp add: TypDesc TypAggregate Nil)
    next
      case (Cons f fs')
      obtain ft' fn' d' where f: f = DTuple ft' fn' d' by (cases f) simp
      from ind-t obtain
        field-ind-d'-fs': field-desc-independent (field-access d') (field-update d')
                          (set (toplevel-field-descs-list fs')) and
        field-ind-fs': field-descs-independent (toplevel-field-descs-list fs') and
        comp-ind-ft': component-descs-independent ft' and
        comp-ind-fs': component-descs-independent-list fs'
        by (simp add: TypDesc TypAggregate Cons f)

    from ind-d-t obtain ind-d-d': field-desc-independent (field-access d) (field-update
d) {d'} and
     ind-d-fs': field-desc-independent (field-access d) (field-update d) (set (toplevel-field-descs-list
fs'))
        apply  (simp add: TypDesc TypAggregate Cons f)
        supply field-desc-independent-insert-iff[iff]
        by fastforce


      from field-desc-independent-symmetric [of {d} {d'}] ind-d-d'
```

have *field-ind-d′-d*: *field-desc-independent* (*field-access d′*) (*field-update d′*) {*d*}
by *auto*

from *field-ind-d′-fs′ field-ind-d′-d*
have *field-ind-d′-d-fs′*: *field-desc-independent* (*field-access d′*) (*field-update d′*)
(*insert d* (*set* (*toplevel-field-descs-list fs′*)))
apply (*subst insert-is-Un*)
apply (*simp only*: *field-desc-independent-union-iff*)
done

have *ind-d*: *field-descs-independent* [*d*] **by** *simp*
from *field-desc-independent-symmetric* [*of set* [*d*] *set* (*toplevel-field-descs-list fs′*)] *ind-d-fs′*
have *ind-fs′-d*: ∀ *x*∈*set* (*toplevel-field-descs-list fs′*).
field-desc-independent (*field-access x*) (*field-update x*) (*set* [*d*]) **by** *simp*
from *field-descs-independent-appendI1* [*OF field-ind-fs′ ind-d ind-fs′-d*]
have *field-ind-fs′-d*:*field-descs-independent* (*toplevel-field-descs-list fs′* @ [*d*]) **.**

from *ind-fs′-d ind-ft*
have *comp-ind-fs′-d*: *component-descs-independent-list* (*fs′* @ [*DTuple ft fn d*])
apply −
apply (*rule component-descs-independent-list-appendI* [*OF - comp-ind-fs′*])
apply *simp*
apply *simp*
done
from *field-ind-d′-d-fs′ field-ind-fs′-d comp-ind-ft′ comp-ind-fs′-d*
show *?thesis*
by (*simp add*: *TypDesc TypAggregate Cons f*)
qed
qed
qed

lemma *fu-commutes-id1* [*simp*]: *fu-commutes* (λ*bs. id*) *upd*
by (*auto simp add*: *fu-commutes-def*)

lemma *fu-commutes-id2* [*simp*]: *fu-commutes upd* (λ*bs. id*)
by (*auto simp add*: *fu-commutes-def*)

lemma *field-desc-independent-pad*: *field-desc-independent* (λ*v bs. if n* ≤ *length bs then take n bs else replicate n 0*) (λ*bs. id*) *D*
by (*rule field-desc-independent.intro*) *auto*

lemma *component-descs-independent-ti-pad-combine*:

assumes *ind-t*: *component-descs-independent t*
shows *component-descs-independent* (*ti-pad-combine n t*)
apply (*simp add*: *ti-pad-combine-def Let-def padding-desc-def*)
apply (*rule component-descs-independent-extend-ti* [*OF ind-t*])
apply *simp*
apply (*simp add*: *field-desc-independent-pad*)
done

lemma (**in** *wf-field*) *field-desc-independent-update-desc*:
assumes *ind*: *field-desc-independent* (*field-access d*) (*field-update d*) *D*
shows *field-desc-independent* (*field-access* (*update-desc acc upd d*)) (*field-update* (*update-desc acc upd d*))
(*update-desc acc upd ‘ D*) (**is** *field-desc-independent ?acc ?upd ?U*)
proof −
from *ind*
interpret *ind*: *field-desc-independent field-access d field-update d D* .
show *?thesis*
proof
fix *ud*
assume *ud* ∈ *?U*
show *fu-commutes ?upd* (*field-update ud*)
by (*smt* (*verit*) ‹*ud* ∈ *update-desc acc upd ‘ D*› *acc-upd double-update-upd field-desc.select-convs*(*2*)
field-desc-independent.fu-commutes fu-commutes-def imageE ind.field-desc-independent-axioms update-desc-def)
next
fix *ud bs v bs′*
assume *ud* ∈ *?U*
show *?acc* (*field-update ud bs v*) *bs′* = *?acc v bs′*
by (*smt* (*verit*) ‹*ud* ∈ *update-desc acc upd ‘ D*› *acc-upd field-desc.select-convs*(*1*) *field-desc.select-convs*(*2*)
field-desc-independent.acc-upd-old imageE ind.field-desc-independent-axioms o-apply update-desc-def)
next
fix *ud bs′ v bs*
assume *ud* ∈ *?U*
show *field-access ud* (*?upd bs′ v*) *bs* = *field-access ud v bs*
by (*smt* (*verit*) ‹*ud* ∈ *update-desc acc upd ‘ D*› *acc-upd field-desc.select-convs*(*1*) *field-desc.select-convs*(*2*)
field-desc-independent.acc-upd-new imageE ind.field-desc-independent-axioms o-apply update-desc-def)
qed
qed

lemma (**in** *wf-field*) *field-descs-independent-update-desc*:
field-descs-independent (*toplevel-field-descs t*)

```
  ⟹ field-descs-independent (toplevel-field-descs
       (map-td (λn algn. update-desc acc upd) (update-desc acc upd) t))
  field-descs-independent (toplevel-field-descs-struct st)
  ⟹ field-descs-independent (toplevel-field-descs-struct
       (map-td-struct (λn algn. update-desc acc upd) (update-desc acc upd) st))
  field-descs-independent (toplevel-field-descs-list fs)
  ⟹ field-descs-independent (toplevel-field-descs-list
       (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) fs))
  field-descs-independent (toplevel-field-descs-tuple f)
  ⟹ field-descs-independent (toplevel-field-descs-tuple
       (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd) f))
proof (induct t and st and fs and f)
  case (TypDesc st n)
  then show ?case by simp
next
  case (TypScalar sz align d)
  then show ?case by simp
next
  case (TypAggregate fs)
  then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc f fs)

  obtain ft' fn' d' where f: f = DTuple ft' fn' d' by (cases f) simp

  from Cons-typ-desc.prems
  have ind-f-fs: field-descs-independent
    (toplevel-field-descs-tuple f @ toplevel-field-descs-list fs) by simp

  from field-descs-independent-append-first [OF ind-f-fs]
  have ind-f: field-descs-independent (toplevel-field-descs-tuple f) .

  from field-descs-independent-append-second [OF ind-f-fs]
  have ind-fs: field-descs-independent (toplevel-field-descs-list fs) .


  from Cons-typ-desc.hyps(1) [OF ind-f]
  have ind-upd-f: field-descs-independent
                  (toplevel-field-descs-tuple
                     (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc
upd) f)).

  from Cons-typ-desc.hyps(2) [OF ind-fs]
  have ind-upd-fs: field-descs-independent
                   (toplevel-field-descs-list
                      (map-td-list (λn algn. update-desc acc upd) (update-desc acc
```

upd) *fs*)) .

from *field-descs-independent-append-first-ind* [*OF ind-f-fs*]
have *field-desc-independent* (*field-access d′*) (*field-update d′*) (*set* (*toplevel-field-descs-list fs*))
by (*auto simp add*: *f*)
from *field-desc-independent-update-desc* [*OF this*]
have *field-desc-independent* (*field-access* (*update-desc acc upd d′*)) (*field-update* (*update-desc acc upd d′*))
(*update-desc acc upd* ‘ *set* (*toplevel-field-descs-list fs*)) .
then
have *upd-d′-fs*: *field-desc-independent* (*field-access* (*update-desc acc upd d′*)) (*field-update* (*update-desc acc upd d′*))
(*set* (*toplevel-field-descs-list*
(*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *fs*)))
by (*simp add*: *toplevel-field-descs-map*)
from *ind-upd-fs upd-d′-fs*
have *ind-upd-f-fs*: *field-descs-independent*
(*toplevel-field-descs-tuple*
(*map-td-tuple* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *f*) @
toplevel-field-descs-list
(*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *fs*))
by (*simp add*: *f*)
from *Cons-typ-desc ind-upd-f-fs*
show *?case* **by** *simp*
next
case (*DTuple-typ-desc ft fn d*)
then show *?case* **by** *simp*
qed

lemma (**in** *wf-field*) *component-descs-independent-update-desc*:
component-descs-independent t
⟹ *component-descs-independent*
(*map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*)
component-descs-independent-struct st
⟹ *component-descs-independent-struct*
(*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *st*)
component-descs-independent-list fs
⟹ *component-descs-independent-list*
(*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *fs*)
component-descs-independent-tuple f
⟹ *component-descs-independent-tuple*
(*map-td-tuple* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *f*)
proof (*induct t* **and** *st* **and** *fs* **and** *f*)

```
  case (TypDesc st n)
then show ?case by simp
next
  case (TypScalar sz align d)
  then show ?case by simp
next
case (TypAggregate fs)
then show ?case by simp
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc f fs)

  obtain ft' fn' d' where f: f = DTuple ft' fn' d' by (cases f) simp

  from Cons-typ-desc.prems obtain
    find-f-fs: field-descs-independent
                (toplevel-field-descs-tuple f @ toplevel-field-descs-list fs) and
    ind-f: component-descs-independent-tuple f and
    ind-fs: component-descs-independent-list fs
    by simp


  from field-descs-independent-append-first [OF find-f-fs]
  have fi-f: field-descs-independent (toplevel-field-descs-tuple f) .

  from Cons-typ-desc.hyps(1) [OF ind-f]
  have ind-upd-f: component-descs-independent-tuple
         (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd) f) .
  from field-descs-independent-update-desc(4) [OF fi-f]
  have field-descs-independent (toplevel-field-descs-tuple
        (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd) f)) .
  with ind-upd-f
  have ind-upd-f': component-descs-independent-list
         (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) [f])  by
simp


  from Cons-typ-desc.hyps(2) [OF ind-fs]
  have ind-upd-fs: component-descs-independent-list
         (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) fs) .
  from component-descs-list-field-descs-independent [OF this]
  have find-upd-fs: field-descs-independent (toplevel-field-descs-list
         (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) fs)) .
  from field-descs-independent-append-first-ind [OF find-f-fs]
 have find-fs: field-desc-independent (field-access d') (field-update d') (set (toplevel-field-descs-list
fs))
    by (auto simp add: f)
```

from *field-desc-independent-update-desc* [*OF this*]
have *field-desc-independent* (*field-access* (*update-desc acc upd d′*)) (*field-update* (*update-desc acc upd d′*))
(*update-desc acc upd ‘ set* (*toplevel-field-descs-list fs*)) **.**
then
have *upd-d′-fs*: *field-desc-independent* (*field-access* (*update-desc acc upd d′*)) (*field-update* (*update-desc acc upd d′*))
(*set* (*toplevel-field-descs-list*
(*map-td-list* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *fs*)))
by (*simp add*: *toplevel-field-descs-map*)

from *upd-d′-fs find-upd-fs*
have *ind-f-fs*: *field-descs-independent*
(*toplevel-field-descs-tuple*
(*map-td-tuple* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *f*) @
toplevel-field-descs-list
(*map-td-list* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *fs*))
by (*simp add*: *f*)
from *Cons-typ-desc ind-f-fs* **show** *?case*
by (*simp*)
next
case (*DTuple-typ-desc ft fn d*)
then show *?case* **by** *simp*
qed

lemma (**in** *wf-field*) *component-descs-independent-adjust-ti*:
assumes *ind-t*: *component-descs-independent t*
shows *component-descs-independent* (*adjust-ti t acc upd*)
apply (*simp add*: *adjust-ti-def*)
apply (*rule component-descs-independent-update-desc*(*1*) [*OF ind-t*])
done

theorem (**in** *wf-xfield*) *component-descs-independent-ti-typ-combine*:
fixes
ft::′*a*::*xmem-type itself* **and**
t:: ′*s xtyp-info*
assumes *ind-t*: *component-descs-independent t*
assumes *ind-acc-upd*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs t*))
shows *component-descs-independent* (*ti-typ-combine ft acc upd algn fn t*)
apply (*simp add*: *ti-typ-combine-def*)
apply (*rule component-descs-independent-extend-ti* [*OF ind-t*])
apply (*rule component-descs-independent-adjust-ti* [*OF component-descs-independent*])
apply *simp*
apply (*rule ind-acc-upd*)

done

lemma (**in** *wf-xfield*) *wf-field-desc-extend-ti*:
assumes *ind-t*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*) (*set* (*toplevel-field-descs t*))
assumes *ind-d*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*) {*d*}
shows
field-desc-independent (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs* (*extend-ti t ft align fn d*)))
proof (*cases t*)
case (*TypDesc align′ st nn*)
then show *?thesis*
proof (*cases st*)
case (*TypScalar sz align d*)
from *ind-d* **show** *?thesis*
by (*simp add*: *TypDesc TypScalar*)
next
case (*TypAggregate fs*)
then show *?thesis*
proof (*cases fs*)
case *Nil*
from *ind-d* **show** *?thesis*
by (*simp add*: *TypDesc TypAggregate Nil*)
next
case (*Cons f fs′*)
obtain *ft′ fn′ d′* **where** *f*: *f* = *DTuple ft′ fn′ d′* **by** (*cases f*) *simp*
from *ind-d ind-t*
show *?thesis*
apply (*simp add*: *TypDesc TypAggregate Cons f*)
by (*meson field-desc-independent-insert-iff*)
qed
qed
qed

lemma (**in** *wf-xfield*) *field-desc-independent-ti-pad-combine*:
assumes *ind-acc-upd*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs t*))
shows *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs* (*ti-pad-combine n t*)))
apply (*simp add*: *ti-pad-combine-def Let-def padding-desc-def*)
apply (*rule wf-field-desc-extend-ti* [*OF ind-acc-upd*])
by (*auto simp add*: *field-desc-independent-def*)

theorem (**in** *wf-xfield*) *component-desc-independent-ti-typ-pad-combine*:
fixes
ft::′*a*::*xmem-type itself* **and**
t:: ′*s xtyp-info*
assumes *ind-t*: *component-descs-independent t*

assumes *ind-acc-upd*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs t*))
shows *component-descs-independent* (*ti-typ-pad-combine ft acc upd algn fn t*)

apply (*cases 0 < padup* (*max* (*2 ^ algn*) (*align-of TYPE*('*a*))) (*size-td t*))
apply (*simp add*: *ti-typ-pad-combine-def Let-def*)
apply (*rule component-descs-independent-ti-typ-combine*)
apply (*rule component-descs-independent-ti-pad-combine* [*OF ind-t*])
apply (*rule field-desc-independent-ti-pad-combine* [*OF ind-acc-upd*])
apply (*simp add*: *ti-typ-pad-combine-def Let-def*)
apply (*rule component-descs-independent-ti-typ-combine* [*OF ind-t ind-acc-upd*])
done

lemma *component-descs-independent-map-align*[*simp*]:
component-descs-independent (*map-align f t*) = *component-descs-independent t*
by (*cases t*) *simp*

lemma *component-descs-independent-final-pad*:
assumes *ind-t*: *component-descs-independent t*
shows *component-descs-independent* (*final-pad algn t*)
apply (*clarsimp simp add*: *final-pad-def Let-def ind-t*)
apply (*rule component-descs-independent-ti-pad-combine* [*OF ind-t*])
done

theorem (**in** *xmem-contained-type*) *contained-field-descs-ti-typ-combine*:
fixes
ft::'*a itself* **and**
t:: '*b xtyp-info*
assumes *contained-t*: *contained-field-descs t*
shows *contained-field-descs* (*ti-typ-combine ft acc upd algn fn t*)
proof −
from *contained-field-descs-adjust-ti* [*OF contained-field-descs*]
have *contained-field-descs* (*adjust-ti* (*typ-info-t TYPE*('*a*)) *acc upd*) **.**
moreover
define *d*::'*b field-desc*
where *d* ≡ (|*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*a*)|)

from *fields-contained-update-desc-mem-type* [*of acc upd*]
have *fields-contained* (*field-access d*) (*field-update d*)
(*set* (*toplevel-field-descs* (*adjust-ti* (*typ-info-t TYPE*('*a*)) *acc upd*)))
by (*simp add*: *toplevel-field-descs-adjust-ti d-def*)
ultimately
show *?thesis*
using *contained-t*
by (*simp add*: *ti-typ-combine-def contained-field-descs-extend-ti d-def*)
qed

theorem (**in** *xmem-contained-type*) *contained-field-descs-ti-typ-pad-combine*:
fixes
ft::*'a itself* **and**
t:: *'b xtyp-info*
assumes *contained-t*: *contained-field-descs t*
shows *contained-field-descs* (*ti-typ-pad-combine ft acc upd algn fn t*)
using *contained-t contained-field-descs*
by (*simp add*: *ti-typ-pad-combine-def Let-def contained-field-descs-ti-typ-combine contained-field-descs-ti-pad-combine*)

locale *ti-ind'* =
fixes *X* :: *'a leaf-desc set*
fixes *Y* :: *'a leaf-desc set*
assumes *fu-commutes*: $x \in X \implies y \in Y \implies$ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*))
assumes *fa-fu-ind-X*: $x \in X \implies y \in Y \implies$ *fa-fu-ind* (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*)
assumes *fa-fu-ind-Y*: $x \in X \implies y \in Y \implies$ *fa-fu-ind* (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*)

lemma *ti-ind'-ti-ind*: *ti-ind' X Y* = *ti-ind X Y*
by (*auto simp add*: *ti-ind-def ti-ind'-def*)

lemma *fields-contained-transitive*:
assumes *d-in*: $d \in D$
assumes *d-contains* : *fields-contained* (*field-access d*) (*field-update d*) *X*
assumes *contained*: *fields-contained acc upd D*
shows *fields-contained acc upd X*
using *d-in d-contains contained* **unfolding** *fields-contained-def*
by *metis*

lemma *fields-contained-singleton* [*simp*]:
fields-contained (*field-access d*) (*field-update d*) {*d*}
by (*auto simp add*: *fields-contained-def*)

lemma *contained-field-descs-leaf*:
contained-field-descs t $\implies$ *ld* $\in$ *lf-fd* ‘ (*lf-set t* []) $\implies$
$\exists\, d \in$ *set* (*toplevel-field-descs t*). *fields-contained* (*field-access d*) (*field-update d*) {*ld*}
contained-field-descs-struct st $\implies$ *ld* $\in$ *lf-fd* ‘ (*lf-set-struct st* []) $\implies$
$\exists\, d \in$ *set* (*toplevel-field-descs-struct st*). *fields-contained* (*field-access d*) (*field-update d*) {*ld*}
contained-field-descs-list fs $\implies$ *ld* $\in$ *lf-fd* ‘ (*lf-set-list fs* []) $\implies$
$\exists\, d \in$ *set* (*toplevel-field-descs-list fs*). *fields-contained* (*field-access d*) (*field-update d*) {*ld*}

contained-field-descs-tuple f ⟹ *ld* ∈ *lf-fd* ‘ (*lf-set-tuple f* []) ⟹
∃ *d* ∈ *set* (*toplevel-field-descs-tuple f*). *fields-contained* (*field-access d*) (*field-update d*) {*ld*}
proof (*induct t* **and** *st* **and** *fs* **and** *f*)
case (*TypDesc st n*)
then show *?case* **by** *auto*
next
case (*TypScalar n sz d*)
then show *?case* **by** *auto*
next
case (*TypAggregate fs*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc f fs*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc ft fn d*)
then show *?case*
by (*metis contained-field-descs-tuple.simps toplevel-field-descs-tuple.simps fields-contained-transitive insertI1 lf-fd-fn*(*1*) *lf-set-tuple.simps list.simps*(*15*))
qed

lemma *wf-field-desc-wf-lf*:
wf-field-desc d ⟹ *sz* = *field-sz d* ⟹ *wf-lf* {(| *lf-fd* = *d*, *lf-sz* = *sz*, *lf-fn* = *xs* |)}
by (*simp add*: *wf-field-desc-def*
fd-cons-access-update-def fd-cons-desc-def fd-cons-double-update-def fd-cons-length-def
fd-cons-update-access-def padding-lense.access-result-size padding-lense.access-update
padding-lense.double-update padding-lense.update-access wf-lf-def)

lemma *wf-lf-unionI*: *wf-lf A* ⟹ *wf-lf B* ⟹ *ti-ind′ A B* ⟹ *wf-lf* (*A* ∪ *B*)
by (*auto simp add*: *wf-lf-def ti-ind′-def fu-commutes-def*)

lemma *fields-contained-subset*:
assumes *cont-X*: *fields-contained acc upd X*
assumes *subs*: *Y* ⊆ *X*
shows *fields-contained acc upd Y*
proof −
from *cont-X* **interpret** *X*: *fields-contained acc upd X* **.**
show *?thesis*
proof
fix *d s s′*

```
    assume d: d ∈ Y
    assume acc: acc s = acc s'
    show field-access d s = field-access d s'
      using d subs acc X.access-contained by blast
  next
    fix d bs s
    assume d: d ∈ Y
    show ∃ bs'. ∀ s'. acc s' = acc s ⟶ field-update d bs s' = upd bs' s'
      using d subs X.update-contained by blast
  qed
qed


lemma fields-contained-unionD1:
  assumes fields-contained acc upd (X ∪ Y)
  shows fields-contained acc upd X
  using assms
  by (auto intro: fields-contained-subset)


lemma fields-contained-unionD2:
  assumes fields-contained acc upd (X ∪ Y)
  shows fields-contained acc upd Y
  using assms
  by (auto intro: fields-contained-subset)

lemma fields-contained-unionI:
  assumes cont-X: fields-contained acc upd X
  assumes cont-Y: fields-contained acc upd Y
  shows fields-contained acc upd (X ∪ Y)
proof −
  from cont-X interpret X: fields-contained acc upd X .
  from cont-Y interpret Y: fields-contained acc upd Y .
  show ?thesis
  proof
    fix d s s'
    assume d: d ∈ X ∪ Y
    assume acc: acc s = acc s'
    show field-access d s = field-access d s'
      using d acc X.access-contained Y.access-contained by blast
  next
    fix d bs s
    assume d: d ∈ X ∪ Y
    show ∃ bs'. ∀ s'. acc s' = acc s ⟶ field-update d bs s' = upd bs' s'
      using d X.update-contained Y.update-contained by blast
  qed
qed
```

lemma *fields-contained-insertI*:
assumes *cont-x*: *fields-contained acc upd* $\{x\}$
assumes *cont-Y*: *fields-contained acc upd Y*
shows *fields-contained acc upd* (*insert x Y*)
using *fields-contained-unionI* [*OF cont-x cont-Y*]
by *simp*

lemma *fields-contained-toplevel-to-field-descs*:
$\bigwedge$(*acc*::'*a* ⇒ *byte list* ⇒ *byte list*) *upd*.
contained-field-descs t ⟹ *fields-contained acc upd* (*set* (*toplevel-field-descs t*))
⟹ *fields-contained acc upd* (*set* (*field-descs t*))
$\bigwedge$(*acc*::'*a* ⇒ *byte list* ⇒ *byte list*) *upd*.
contained-field-descs-struct st ⟹ *fields-contained acc upd* (*set* (*toplevel-field-descs-struct st*))
⟹ *fields-contained acc upd* (*set* (*field-descs-struct st*))
$\bigwedge$(*acc*::'*a* ⇒ *byte list* ⇒ *byte list*) *upd*.
contained-field-descs-list fs ⟹ *fields-contained acc upd* (*set* (*toplevel-field-descs-list fs*))
⟹ *fields-contained acc upd* (*set* (*field-descs-list fs*))
$\bigwedge$(*acc*::'*a* ⇒ *byte list* ⇒ *byte list*) *upd*.
contained-field-descs-tuple f ⟹ *fields-contained acc upd* (*set* (*toplevel-field-descs-tuple f*))
⟹ *fields-contained acc upd* (*set* (*field-descs-tuple f*))
proof (*induct t* **and** *st* **and** *fs* **and** *f*)
case (*TypDesc st n*)
then show *?case* **by** *simp*
next
case (*TypScalar st align d*)
then show *?case* **by** *simp*
next
case (*TypAggregate fs*)
then show *?case* **by** *simp*
next
case *Nil-typ-desc*
then show *?case* **by** *simp*
next
case (*Cons-typ-desc f fs*)
from *Cons-typ-desc.prems* **obtain**
fcont-f-fs: *fields-contained acc upd*
(*set* (*toplevel-field-descs-tuple f*) ∪ *set* (*toplevel-field-descs-list fs*))
and
cont-f: *contained-field-descs-tuple f* **and**
cont-fs: *contained-field-descs-list fs* **by** *clarsimp*

note *fcont-f* = *fields-contained-unionD1* [*OF fcont-f-fs*]
note *fcont-fs* = *fields-contained-unionD2* [*OF fcont-f-fs*]

from *Cons-typ-desc.hyps*(*1*) [*OF cont-f fcont-f*] *Cons-typ-desc.hyps*(*2*) [*OF cont-fs fcont-fs*]
have *fields-contained acc upd* (*set* (*field-descs-tuple f*) ∪ *set* (*field-descs-list fs*))
by (*rule fields-contained-unionI*)

then show *?case*
by *clarsimp*

next
case (*DTuple-typ-desc ft fn d*)
then show *?case*
by (*fastforce intro*: *fields-contained-insertI fields-contained-transitive*)
qed

lemma *fu-commutes-intro*:
assumes ⋀*v bs bs'*. *f bs* (*g bs' v*) = *g bs'* (*f bs v*)
shows *fu-commutes f g*
using *assms* **by** (*simp add*: *fu-commutes-def*)

lemma (**in** *wf-field-desc*) *access-same-update-id*:
assumes *field-access d* (*field-update d bs v*) *bs'* = *field-access d v bs'*
shows *field-update d bs v* = *v*
by (*metis assms double-update update-access*)

lemma (**in** *wf-field-desc*) *access-same-update-id'*:
assumes *upd-inv*: *field-update d bs v* = *v*
assumes *acc-same*: *field-access d v bs* = *field-access d v' bs*
shows *field-update d bs v'* = *v'*
by (*metis acc-same access-update double-update upd-inv update-access*)

lemma (**in** *wf-field-desc*) *update-access-unequal*:
assumes *neq-upd*: *field-update d bs v* ≠ *v*
shows *field-access d v bs'* ≠ *field-access d* (*field-update d bs v*) *bs'*
by (*metis access-same-update-id neq-upd*)

lemma (**in** *wf-field-desc*) *access-eq-update-eq*:
assumes ⋀*bs'*. *field-access d* (*field-update d bs v*) *bs'* = *field-access d v bs'*
shows *field-update d bs v* = *v*
by (*metis assms double-update update-access*)

lemma *field-desc-independent-contained-transitive*:
assumes *cont*: *fields-contained* (*field-access e*) (*field-update e*) *D*

```
  assumes ind:  field-desc-independent (field-access d) (field-update d) {e}
  shows field-desc-independent (field-access d) (field-update d) D
proof −
  from cont interpret cont-e: fields-contained field-access e field-update e D .
  from ind interpret ind-e: field-desc-independent field-access d field-update d {e}
.
  show ?thesis
  proof
    fix f
    assume f-in: f ∈ D
    show fu-commutes (field-update d) (field-update f)
    proof (rule fu-commutes-intro)
      fix v bs bs'

      from ind-e.acc-upd-new [of e]
      have e-d-ind: field-access e (field-update d bs v) = field-access e v
        by (rule ext) auto

      from cont-e.update-contained [OF f-in, where s=v] e-d-ind
      obtain bs1 where f-e-v: field-update f bs' v = field-update e bs1 v and
                       f-e-upd-v: field-update f bs' (field-update d bs v) = field-update e
bs1 (field-update d bs v)

        by blast
      have field-update d bs (field-update f bs' v) = field-update d bs (field-update e
bs1 v)
        by (simp add: f-e-v)
      also
       from ind-e.fu-commutes [of e] have ... = field-update e bs1 (field-update d
bs v)
        by (auto simp add: fu-commutes-def)
     also have ... = field-update f bs' (field-update d bs v) by (simp add: f-e-upd-v)
      finally
      show field-update d bs (field-update f bs' v) =
           field-update f bs' (field-update d bs v).
    qed
  next
    fix f bs v bs'
    assume f-in: f ∈ D
    show field-access d (field-update f bs v) bs' = field-access d v bs'
      by (metis cont-e.update-contained f-in ind-e.acc-upd-old insertI1)
  next
    fix f bs' v bs
    assume f-in: f ∈ D
    show field-access f (field-update d bs' v) bs = field-access f v bs
    proof −
     from ind-e.acc-upd-new [of e]
     have e-d-ind: field-access e (field-update d bs' v) = field-access e v
       by (rule ext) auto
```

```
    from cont-e.access-contained [OF f-in e-d-ind]
    show ?thesis by simp
  qed
 qed
qed


lemma field-desc-independent-contained-toplevel-to-field-descs:
  assumes cont-fs: contained-field-descs-list fs
  assumes ind-tl: field-desc-independent (field-access d) (field-update d)
          (set (toplevel-field-descs-list fs))
  shows field-desc-independent (field-access d) (field-update d)
        (set (field-descs-list fs))
  using cont-fs ind-tl
proof (induct fs)
  case Nil
  then show ?case by simp
next
  case (Cons f fs)
  obtain ft fn d' where f: f = DTuple ft fn d' by (cases f) simp

  from Cons.prems obtain
    ind-d-tl: field-desc-independent (field-access d) (field-update d)
              (insert d' (set (toplevel-field-descs-list fs))) and
    cont-ft: contained-field-descs ft and
    cont-d'-ft: fields-contained (field-access d') (field-update d')
                (set (toplevel-field-descs ft)) and
    cont-fs: contained-field-descs-list fs
    by (simp add: f)


  note ind-d-fs-tl = field-desc-independent-insertD2 [OF ind-d-tl]

  from Cons.hyps [OF cont-fs ind-d-fs-tl]
  have ind-d-fs: field-desc-independent (field-access d) (field-update d)
                 (set (field-descs-list fs)).

  from fields-contained-toplevel-to-field-descs(1) [OF cont-ft cont-d'-ft]
  have fcont-d'-ft: fields-contained (field-access d') (field-update d')
                    (set (field-descs ft)) .

  from field-desc-independent-insertD1 [OF ind-d-tl]
  have ind-d-d': field-desc-independent (field-access d) (field-update d) {d'} .

  from field-desc-independent-contained-transitive [OF fcont-d'-ft ind-d-d']
  have field-desc-independent (field-access d) (field-update d) (set (field-descs ft))
.
```

from *field-desc-independent-unionI* [*OF this ind-d-fs*] *field-desc-independent-insertD1* [*OF ind-d-tl*]
have *field-desc-independent* (*field-access d*) (*field-update d*)
(*insert d'* (*set* (*field-descs ft*) ∪ *set* (*field-descs-list fs*)))
by (*blast intro*: *field-desc-independent-insertI*)
then show *?case*
by (*clarsimp simp add*: *f*)
qed

theorem *component-descs-independent-contained-wf-lf*:
fixes *t*::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
⋀*ps*. ⟦*wf-desc t*; *wf-component-descs t*; *wf-field-descs* (*set* (*field-descs t*)); *component-descs-independent t*;
contained-field-descs t ⟧
⟹ *wf-lf* (*lf-set t ps*)
⋀*ps*. ⟦*wf-desc-struct st*; *wf-component-descs-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));
component-descs-independent-struct st; *contained-field-descs-struct st*⟧
⟹ *wf-lf* (*lf-set-struct st ps*)
⋀*ps*. ⟦*wf-desc-list fs*; *wf-component-descs-list fs*; *wf-field-descs* (*set* (*field-descs-list fs*));
component-descs-independent-list fs; *contained-field-descs-list fs*⟧
⟹ *wf-lf* (*lf-set-list fs ps*)
⋀*ps*. ⟦*wf-desc-tuple f*; *wf-component-descs-tuple f*; *wf-field-descs* (*set* (*field-descs-tuple f*));
component-descs-independent-tuple f; *contained-field-descs-tuple f*⟧
⟹ *wf-lf* (*lf-set-tuple f ps*)
proof (*induct t* **and** *st* **and** *fs* **and** *f*)
case (*TypDesc algn st n*)
then show *?case* **by** *simp*
next
case (*TypScalar sz align d*)
then show *?case* **by** (*simp add*: *wf-field-desc-wf-lf*)
next
case (*TypAggregate fs*)
then show *?case* **by** *simp*
next
case *Nil-typ-desc*
then show *?case* **by** *simp*
next
case (*Cons-typ-desc f fs*)
obtain *ft' fn' d'* **where** *f*: *f* = *DTuple ft' fn' d'* **by** (*cases f*) *simp*
from *Cons-typ-desc.prems* **obtain**
wf-f: *wf-desc-tuple f* **and**
wf-fs: *wf-desc-list fs* **and**

wf-cd-f: *wf-component-descs-tuple f* **and**
wf-cd-fs: *wf-component-descs-list fs* **and**
wf-fd-f: *wf-field-descs* (*set* (*field-descs-tuple f*)) **and**
wf-fd-fs: *wf-field-descs* (*set* (*field-descs-list fs*)) **and**
ind-f-fs: *field-descs-independent*
(*toplevel-field-descs-tuple f* @ *toplevel-field-descs-list fs*) **and**
cont-f: *contained-field-descs-tuple f* **and**
cont-fs: *contained-field-descs-list fs* **and**
dist-names: *dt-snd f* ∉ *dt-snd ' set fs* **and**
ind-f: *component-descs-independent-tuple f* **and**
ind-fs:*component-descs-independent-list fs*
by *clarsimp*

from *Cons-typ-desc.hyps*(*1*) [*OF wf-f wf-cd-f wf-fd-f ind-f cont-f*]
have *wf-lf-ft'*: *wf-lf* (*lf-set ft'* (*ps* @ [*fn'*])) **by** (*simp add*: *f*)

from *Cons-typ-desc.hyps*(*2*) [*OF wf-fs wf-cd-fs wf-fd-fs ind-fs cont-fs*]
have *wf-lf-fs*: *wf-lf* (*lf-set-list fs ps*) **.**

from *field-descs-independent-append-first-ind* [*OF ind-f-fs*, *of d'*]
have *field-desc-independent* (*field-access d'*) (*field-update d'*)
(*set* (*toplevel-field-descs-list fs*)) **by** (*simp add*: *f*)

from *cont-f field-desc-independent-contained-toplevel-to-field-descs* [*OF cont-fs*,
where *d*=*d'*] *this*
have *ind-d'-fs*: *field-desc-independent* (*field-access d'*) (*field-update d'*)
(*set* (*field-descs-list fs*))
by (*simp add*: *f*)

from *cont-f fields-contained-toplevel-to-field-descs*
have *cont-d'-ft'*: *fields-contained* (*field-access d'*) (*field-update d'*) (*set* (*field-descs ft'*))
by (*auto simp add*: *f*)

have *ti-ind*: *ti-ind'* (*lf-set ft'* (*ps* @ [*fn'*])) (*lf-set-list fs ps*)
proof
fix *x y*
assume *x-in*: *x* ∈ *lf-set ft'* (*ps* @ [*fn'*])
assume *y-in*: *y* ∈ *lf-set-list fs ps*
from *lf-set-subset-field-descs*(*1*) *x-in*
have *x-in'*: *lf-fd x* ∈ *set* (*field-descs ft'*) **by** *blast*
from *lf-set-subset-field-descs*(*3*) *y-in*
have *y-in'*: *lf-fd y* ∈ *set* (*field-descs-list fs*) **by** *blast*

from *field-desc-independent-symmetric-singleton* [*OF ind-d'-fs y-in'*]
have *field-desc-independent* (*field-access* (*lf-fd y*)) (*field-update* (*lf-fd y*)) {*d'*} **.**

```
  from field-desc-independent-contained-transitive [OF cont-d'-ft' this]
  have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y))
        (set (field-descs ft')) .

  from field-desc-independent.fu-commutes [OF this x-in'] fu-commutes-sym
  show fu-commutes (field-update (lf-fd x)) (field-update (lf-fd y))
    by blast
next
  fix x y
  assume x-in: x ∈ lf-set ft' (ps @ [fn'])
  assume y-in: y ∈ lf-set-list fs ps
  from lf-set-subset-field-descs(1) x-in
  have x-in': lf-fd x ∈ set (field-descs ft') by blast
  from lf-set-subset-field-descs(3) y-in
  have y-in': lf-fd y ∈ set (field-descs-list fs) by blast

  from field-desc-independent-symmetric-singleton [OF ind-d'-fs y-in']
  have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y)) {d'} .

  from field-desc-independent-contained-transitive [OF cont-d'-ft' this]
  have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y))
        (set (field-descs ft')) .
  from field-desc-independent.acc-upd-new [OF this x-in']
  show fa-fu-ind (lf-fd x) (lf-fd y) (lf-sz y) (lf-sz x)
    by (auto simp add: fa-fu-ind-def)
next
  fix x y
  assume x-in: x ∈ lf-set ft' (ps @ [fn'])
  assume y-in: y ∈ lf-set-list fs ps
  from lf-set-subset-field-descs(1) x-in
  have x-in': lf-fd x ∈ set (field-descs ft') by blast
  from lf-set-subset-field-descs(3) y-in
  have y-in': lf-fd y ∈ set (field-descs-list fs) by blast

  from field-desc-independent-symmetric-singleton [OF ind-d'-fs y-in']
  have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y)) {d'} .

  from field-desc-independent-contained-transitive [OF cont-d'-ft' this]
  have field-desc-independent (field-access (lf-fd y)) (field-update (lf-fd y))
        (set (field-descs ft')) .
  from field-desc-independent.acc-upd-old [OF this x-in']
  show fa-fu-ind (lf-fd y) (lf-fd x) (lf-sz x) (lf-sz y)
    by (auto simp add: fa-fu-ind-def)
qed

from wf-lf-unionI [OF wf-lf-ft' wf-lf-fs ti-ind]
have wf-lf (lf-set ft' (ps @ [fn']) ∪ lf-set-list fs ps) .

then show ?case by (simp add: f)
```

next
case (*DTuple-typ-desc ft fn d*)
then show *?case* **by** *simp*
qed

definition *wf-align-field ti* ≡ *wf-align ti* ∧ *align-field ti*

lemma *wf-align-field-empty-typ-info*: *wf-align-field* (*empty-typ-info algn n*)
by (*simp add*: *wf-align-field-def wf-align-simps align-field-empty-typ-info*)

lemma (**in** *mem-type*) *wf-align-field-ti-typ-pad-combine*:
aggregate ti ⟹ *wf-align-field ti* ⟹
wf-align-field (*ti-typ-pad-combine* (*TYPE*('*a*)) *acc upd algn fn ti*)
by (*auto simp add*: *wf-align-field-def wf-align-ti-typ-pad-combine align-field-ti-typ-pad-combine*)

lemma (**in** *mem-type*) *wf-align-field-ti-typ-combine*:
aggregate ti ⟹ *wf-align-field ti* ⟹ 2 ^ *align-td* (*typ-info-t TYPE*('*a*)) *dvd size-td ti* ⟹
wf-align-field (*ti-typ-combine* (*TYPE*('*a*)) *acc upd algn fn ti*)
by (*auto simp add*: *wf-align-field-def wf-align-ti-typ-combine align-field-ti-typ-combine*)

lemma *wf-align-field-ti-pad-combine*: *aggregate ti* ⟹ *wf-align-field ti* ⟹ *wf-align-field* (*ti-pad-combine n ti*)
by (*auto simp add*: *wf-align-field-def wf-align-ti-pad-combine align-field-ti-pad-combine*)

lemma *wf-align-field-final-pad*: *aggregate ti* ⟹ *wf-align-field ti* ⟹ *wf-align-field* (*final-pad algn ti*)
by (*auto simp add*: *wf-align-field-def wf-align-final-pad align-field-final-pad*)

lemmas *wf-align-field-simps* =
wf-align-field-empty-typ-info
wf-align-field-ti-typ-pad-combine
wf-align-field-ti-typ-combine
wf-align-field-ti-pad-combine
wf-align-field-final-pad

The following theorem is used to prove that a new type is in class *xmem-contained-type*, for which we have constructed the extended type info with:

- *empty-typ-info*
- *ti-typ-pad-combine* (*ti-typ-combine*, *ti-pad-combine*)
- *final-pad*.

Note that the field-types are already in *xmem-contained-type*.

theorem *tuned-xmem-contained-type-class-intro*:
assumes *wf-desc*: *wf-desc* (*typ-info-t TYPE*('*a*))
assumes *wf-size-desc*: *wf-size-desc* (*typ-info-t TYPE*('*a*))
assumes *align-dvd*: *align-of TYPE*('*a*) *dvd size-of TYPE*('*a*)
assumes *wf-align-field*: *wf-align-field* (*typ-info-t TYPE*('*a*))
assumes *size*: *size-of TYPE*('*a*) < *addr-card*
assumes *entire-update*: ⋀*bs v w*. *length bs* = *size-of TYPE*('*a*)
⟹ *field-update* (*component-desc* (*typ-info-t TYPE*('*a*))) *bs v* =
field-update (*component-desc* (*typ-info-t TYPE*('*a*))) *bs w*
assumes *wf-component-descs*: *wf-component-descs* (*typ-info-t TYPE*('*a*))
assumes *ind*: *component-descs-independent* (*typ-info-t TYPE*('*a*))
assumes *wf-field-descs*: *wf-field-descs* (*set* (*field-descs* (*typ-info-t TYPE*('*a*))))
assumes *contained-field-descs*: *contained-field-descs* (*typ-info-t TYPE*('*a*))
assumes *wf-padding*: *wf-padding* (*typ-info-t TYPE*('*a*))
shows *OFCLASS*('*a*::*c-type*, *xmem-contained-type-class*)
proof
show *wf-desc* (*typ-info-t TYPE*('*a*))
by (*rule wf-desc*)
next
show *wf-size-desc* (*typ-info-t TYPE*('*a*))
by (*rule wf-size-desc*)
next
show *wf-lf* (*lf-set* (*typ-info-t TYPE*('*a*)) [])
proof −

We can show that all leafs of the tree are disjoint, by exploiting the correct nesting of fields in the *toplevel-structure*.

from *component-descs-independent-contained-wf-lf*(*1*) [*OF wf-desc wf-component-descs wf-field-descs ind contained-field-descs*]
show *?thesis* **.**
qed
next
fix *bs v w*
show *length bs* = *size-of TYPE*('*a*) ⟶
update-ti-t (*typ-info-t TYPE*('*a*)) *bs v* =
update-ti-t (*typ-info-t TYPE*('*a*)) *bs w*
proof

The updates are captured by the component-descriptor of the toplevel structure.

assume *len*: *length bs* = *size-of TYPE*('*a*)
from *entire-update* [*OF len*] *wf-field-descs len wf-component-descs*
show *update-ti-t* (*typ-info-t TYPE*('*a*)) *bs v* =
update-ti-t (*typ-info-t TYPE*('*a*)) *bs w*
by (*simp add*: *ind size-of-def update-ti-component-desc-compatible*(*1*) *update-ti-t-def*)
qed
next
show *align-of TYPE*('*a*) *dvd size-of TYPE*('*a*) **by** (*rule align-dvd*)

next
from *wf-align-field*
show *align-field* (*typ-info-t TYPE*('*a*)) **by** (*simp add*: *wf-align-field-def*)
next
show *size-of TYPE*('*a*) < *addr-card* **by** (*rule size*)
next
show *wf-component-descs* (*typ-info-t TYPE*('*a*)) **by** (*rule wf-component-descs*)
next
show *component-descs-independent* (*typ-info-t TYPE*('*a*)) **by** (*rule ind*)
next
show *wf-field-descs* (*set* (*field-descs* (*typ-info-t TYPE*('*a*)))) **by** (*rule wf-field-descs*)
next
show *contained-field-descs* (*typ-info-t TYPE*('*a*)) **by** (*rule contained-field-descs*)
next
from *wf-align-field*
show *wf-align* (*typ-info-t TYPE*('*a*))
by (*simp add*: *wf-align-field-def*)
next
from *wf-padding*
show *wf-padding* (*typ-info-t TYPE*('*a*)) **.**
qed

lemma *field-sz-extend-ti*: *aggregate t* ⟹
field-sz (*component-desc* (*extend-ti t ft algn fn d*)) = *field-sz* (*component-desc t*)
+ *field-sz d*
apply (*cases t*)
subgoal for *x1 st n*
apply (*cases st*)
apply (*simp*)
subgoal for *fs*
apply *simp*
done
done
done

lemma *field-sz-empty-typ-info*: *field-sz* (*component-desc* (*empty-typ-info algn n*))
= *0*
by (*simp add*: *empty-typ-info-def*)

lemma (**in** *c-type*) *field-sz-ti-typ-combine*:
fixes *acc*:: '*s* ⇒ '*a*
assumes *agg*: *aggregate t*
shows *field-sz* (*component-desc* (*ti-typ-combine ft acc upd algn fn t*)) =
field-sz (*component-desc t*) + *size-of TYPE*('*a*)
using *agg*
by (*simp add*: *ti-typ-combine-def Let-def field-sz-extend-ti*)

lemma (**in** *c-type*) *field-sz-ti-pad-combine*:
fixes *acc*:: '*s* ⇒ '*a*

assumes *agg*: *aggregate t*
shows *field-sz* (*component-desc* (*ti-pad-combine n t*)) =
field-sz (*component-desc t*) + *n*
using *agg*
by (*simp add*: *ti-pad-combine-def Let-def field-sz-extend-ti padding-desc-def*)

lemma (**in** *c-type*) *field-sz-ti-typ-pad-combine*:
fixes *acc*:: *'s* ⇒ *'a*
assumes *agg*: *aggregate t*
shows *field-sz* (*component-desc* (*ti-typ-pad-combine ft acc upd algn fn t*)) =
field-sz (*component-desc t*) + *size-of TYPE*('*a*) + *padup* (*max* (*2* ^ *algn*)
(*align-of TYPE*('*a*))) (*size-td t*)
using *agg*
by (*simp add*: *ti-typ-pad-combine-def Let-def field-sz-ti-pad-combine field-sz-ti-typ-combine*)

lemma *component-desc-map-align*[*simp*]:*component-desc* (*map-align f t*) = *component-desc t*
by (*cases t*) *simp*

lemma *field-sz-final-pad*:
assumes *agg*: *aggregate t*
shows *field-sz* (*component-desc* (*final-pad algn t*)) =
field-sz (*component-desc t*) + *padup* (*2*^(*max algn* (*align-td t*))) (*size-td t*)
using *agg*
by (*simp add*: *final-pad-def Let-def field-sz-ti-pad-combine*)

lemma *split-fold-append*: *split-fold f* (*xs*@*ys*) *bs s* =
(*let* (*s'*, *bs'*) = *split-fold f xs bs s in split-fold f ys bs' s'*)
apply (*induct xs arbitrary*: *bs ys s*)
apply *simp*
apply *clarsimp*
by (*metis prod.case-distrib*)

lemma *apply-field-updates-append*: *apply-field-updates* (*xs*@*ys*) *bs s* =
(*let* (*s'*, *bs'*) = *apply-field-updates xs bs s in apply-field-updates ys bs' s'*)
by (*simp add*: *apply-field-updates-def split-fold-append*)

lemma *snd-apply-field-updates*: *snd* (*apply-field-updates ds bs s*) = (*drop* (*foldl* (+)
0 (*map field-sz ds*)) *bs*)
apply (*induct ds arbitrary*:*bs s*)
apply *simp*
apply (*simp add*: *add.commute sum-nat-foldl*)
done

lemma *field-update-extend-ti*:

assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*extend-ti t ft algn fn d*)) *bs v* =
field-update d ((*drop* (*field-sz* (*component-desc t*))) *bs*)
(*field-update* (*component-desc t*) *bs v*)
proof (*cases t*)
case (*TypDesc algn st n*)
show *?thesis*
proof (*cases st*)
case (*TypScalar sz align d*)
with *agg* **show** *?thesis* **by** (*simp add*: *TypDesc*)
next
case (*TypAggregate fs*)
from *snd-apply-field-updates*
have *fst* (*apply-field-updates* (*map dt-trd fs* @ [*d*]) *bs v*) =
field-update d (*drop* (*foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) *fs*)) *bs*)
(*fst* (*apply-field-updates* (*map dt-trd fs*) *bs v*))
by (*simp add*: *apply-field-updates-append case-prod-beta snd-apply-field-updates*)

then show *?thesis* **by** (*simp add*: *TypDesc TypAggregate*)
qed
qed

lemma *field-update-empty-typ-info*: *field-update* (*component-desc* (*empty-typ-info algn n*)) *bs s* = *s*
by (*simp add*: *empty-typ-info-def*)

lemma (**in** *c-type*) *field-update-ti-typ-combine*:
assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*ti-typ-combine* (*ft*::*'a itself*) *acc upd algn fn t*)) *bs v* =
(*upd o xfrom-bytes*) ((*drop* (*field-sz* (*component-desc t*))) *bs*)
(*field-update* (*component-desc t*) *bs v*)
using *agg*
by (*simp add*: *ti-typ-combine-def field-update-extend-ti*)

lemma *field-update-ti-pad-combine*:
assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*ti-pad-combine n t*)) *bs v* =
field-update (*component-desc t*) *bs v*
using *agg*
by (*simp add*: *ti-pad-combine-def Let-def field-update-extend-ti padding-desc-def*)

lemma (**in** *c-type*) *field-update-ti-typ-pad-combine*:
fixes *acc*:: *'s* ⇒ *'a*
assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*ti-typ-pad-combine ft acc upd algn fn t*)) *bs v* =

(*upd o xfrom-bytes*) (*drop* (*field-sz* (*component-desc t*) + *padup* (*max* (*2* ^ *algn*) (*align-of TYPE*('*a*))) (*size-td t*)) *bs*)
(*field-update* (*component-desc t*) *bs v*)
using *agg*
apply (*clarsimp simp add*: *ti-typ-pad-combine-def Let-def field-update-ti-pad-combine field-update-ti-typ-combine*)
apply (*clarsimp simp add*: *ti-pad-combine-def field-update-extend-ti Let-def field-sz-extend-ti padding-desc-def*)
done

lemma *field-update-final-pad*:
assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*final-pad algn t*)) *bs v* =
field-update (*component-desc t*) *bs v*
using *agg*
by (*simp add*: *final-pad-def field-update-ti-pad-combine Let-def*)

lemma *set-toplevel-field-descs-extend-ti*:
set (*toplevel-field-descs* (*extend-ti t ft algn fn d*)) ⊆ *insert d* (*set* (*toplevel-field-descs t*))
apply (*cases t*)
subgoal for *x1 st n*
apply (*cases st*)
apply *simp*
subgoal for *fs*
apply (*cases fs*)
apply *simp*
subgoal for *f fs'*
apply *simp*
done
done
done
done

lemma *set-toplevel-field-descs-extend-ti-aggregate*:
aggregate t ⟹ *set* (*toplevel-field-descs* (*extend-ti t ft algn fn d*)) = *insert d* (*set* (*toplevel-field-descs t*))
apply (*cases t*)
subgoal for *x1 st n*
apply (*cases st*)
apply *simp*
subgoal for *fs*
apply (*cases fs*)
apply *simp*
subgoal for *f fs'*
apply *simp*
done
done
done

done

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-typ-combine*:
set (*toplevel-field-descs* (*ti-typ-combine ft* (*acc*::*'b* ⇒ *'a*) *upd algn fn t*)) ⊆
insert (⦇*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*, *field-sz* = *size-of TYPE*(*'a*)⦈)
(*set* (*toplevel-field-descs t*))
apply (*simp add*: *ti-typ-combine-def*)
apply (*rule set-toplevel-field-descs-extend-ti*)
done

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-typ-combine-aggregate*:
fixes *acc*::*'s* ⇒ *'a*
assumes *agg*: *aggregate t*
shows *set* (*toplevel-field-descs* (*ti-typ-combine ft acc upd algn fn t*)) =
insert (⦇*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*, *field-sz* = *size-of TYPE*(*'a*)⦈)
(*set* (*toplevel-field-descs t*))
using *agg*
apply (*simp add*: *ti-typ-combine-def*)
apply (*rule set-toplevel-field-descs-extend-ti-aggregate*)
apply *simp*
done

lemma *set-field-descs-extend-ti-aggregate*:
aggregate t ⟹ *set* (*field-descs* (*extend-ti t ft algn fn d*)) = *insert d* (*set* (*field-descs ft*) ∪ *set* (*field-descs t*))
apply (*cases t*)
subgoal for *x1 st n*
apply (*cases st*)
apply *simp*
subgoal for *fs*
apply (*cases fs*)
apply *simp*
subgoal for *f fs'*
apply *auto*
done
done
done
done

lemma (**in** *c-type*) *set-field-descs-ti-typ-combine-aggregate*:
fixes *acc*::*'s* ⇒ *'a*
assumes *agg*: *aggregate t*
shows *set* (*field-descs* (*ti-typ-combine ft acc upd algn fn t*)) =
insert (⦇*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*, *field-sz* = *size-of TYPE*(*'a*)⦈)
(*set* (*field-descs* (*adjust-ti* (*typ-info-t TYPE*(*'a*)) *acc upd*)) ∪ *set* (*field-descs t*))

using *agg*
apply (*simp add*: *ti-typ-combine-def*)
apply (*simp add*: *set-field-descs-extend-ti-aggregate*)
done

locale *padding*=
fixes *d*::'*a field-desc*
assumes *independent*: *field-desc-independent acc upd* {*d*}

lemma *pad-is-padding*: *padding* (*padding-desc n*)
unfolding *padding-desc-def*
apply (*rule padding.intro*)
apply (*rule field-desc-independent.intro*)
apply (*auto simp add*: *fu-commutes-def*)
done

definition *PAD*::'*a field-desc set* **where** *PAD* = {*d*. *padding d*}

lemma *in-PAD-iff*[*iff*]: *d* ∈ *PAD* ⟷ (*padding d*) **by** (*auto simp add*: *PAD-def*)

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-pad-combine-aggregate*:
fixes *acc*::'*s* ⇒ '*a*
assumes *agg*: *aggregate t*
shows *set* (*toplevel-field-descs* (*ti-pad-combine n t*)) ∪ *PAD* =
set (*toplevel-field-descs t*) ∪ *PAD*
using *agg*
by (*auto simp add*: *ti-pad-combine-def Let-def set-toplevel-field-descs-extend-ti-aggregate pad-is-padding*)

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-typ-pad-combine-aggregate*:
fixes *acc*::'*s* ⇒ '*a*
assumes *agg*: *aggregate t*
shows *set* (*toplevel-field-descs* (*ti-typ-pad-combine ft acc upd algn fn t*)) ∪ *PAD*
=
insert (⦇*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*, *field-sz* = *size-of TYPE*('*a*)⦈)
((*set* (*toplevel-field-descs t*)) ∪ *PAD*)
using *agg*
apply (*simp add*: *ti-typ-pad-combine-def Let-def*)
apply (*cases* 0 < *padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*a*))) (*size-td t*))
apply *clarsimp*
using *set-toplevel-field-descs-ti-pad-combine-aggregate* [*OF agg*, *of* (*padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*a*))) (*size-td t*))]
set-toplevel-field-descs-ti-typ-combine-aggregate [*OF aggregate-ti-pad-combine* [*of* (*padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*a*))) (*size-td t*)) *t*], *of ft acc upd algn fn*]
apply *clarsimp*

apply (*simp add*: *set-toplevel-field-descs-ti-typ-combine-aggregate*)
done

lemma *toplevel-field-descs-map-align*[*simp*]: *toplevel-field-descs* (*map-align f t*) = *toplevel-field-descs t*
by (*cases t*) *simp*

lemma *set-toplevel-field-descs-final-pad-aggregate*:
assumes *agg*: *aggregate t*
shows *set* (*toplevel-field-descs* (*final-pad algn t*)) ∪ *PAD* =
set (*toplevel-field-descs t*) ∪ *PAD*
using *agg*
by (*auto simp add*: *final-pad-def Let-def set-toplevel-field-descs-ti-pad-combine-aggregate*)

lemma *set-toplevel-field-descs-empty-typ-info*: *set* (*toplevel-field-descs* (*empty-typ-info algn n*)) = {}
by (*simp add*: *empty-typ-info-def*)

lemmas *set-toplevel-field-descs-combinator-simps* =
set-toplevel-field-descs-empty-typ-info
set-toplevel-field-descs-ti-typ-combine-aggregate
set-toplevel-field-descs-ti-typ-pad-combine-aggregate
set-toplevel-field-descs-final-pad-aggregate

lemma *field-descs-independent-PAD*:
field-desc-independent acc upd (*D* ∪ *PAD*) = *field-desc-independent acc upd D*
by (*auto simp add*: *field-desc-independent-def padding-def*)

lemma *field-desc-independent-PAD-expand*:
field-desc-independent acc upd (*D* ∪ *PAD*) ⟹ *field-desc-independent acc upd D*
by (*simp add*: *field-descs-independent-PAD*)

lemma *field-desc-independent-PAD-collapse*:
field-desc-independent acc upd D ⟹ *field-desc-independent acc upd* (*D* ∪ *PAD*)
by (*simp add*: *field-descs-independent-PAD*)

lemma *insert-union-out*: *insert d* (*X* ∪ *Y*) = *insert d X* ∪ *Y*
by *blast*

lemmas *field-sz-typ-combinators-simps* =
field-sz-final-pad
field-sz-ti-typ-pad-combine
field-sz-ti-typ-combine
field-sz-empty-typ-info

lemmas *field-update-typ-combinators-simps* =
field-update-final-pad

field-update-ti-typ-pad-combine
field-update-ti-typ-combine
field-update-empty-typ-info

lemma *aggregate-map-align*[*simp*]: *aggregate* (*map-align f t*) = *aggregate t*
by (*cases t*) *simp*

lemma *aggregate-final-pad*[*simp*]: *aggregate t* ⟹ *aggregate* (*final-pad algn t*)
by (*simp add*: *final-pad-def Let-def*)

lemmas *aggregate-typ-combinators-simps* =
aggregate-empty-typ-info
aggregate-ti-pad-combine
aggregate-ti-typ-pad-combine
aggregate-final-pad

lemma *wf-padding-empty-typ-info*: *wf-padding* (*empty-typ-info algn tn*)
by (*simp add*: *empty-typ-info-def*)

lemma *is-padding-tag-padding-tag*[*simp*]: *is-padding-tag* (*padding-tag n*)
by (*simp add*: *is-padding-tag-def padding-tag-def*)

lemma *is-padding-tag-pad-typ*[*simp*]: *is-padding-tag* (*TypDesc 0* (*TypScalar n 0* (*padding-desc n*)) ''!*pad-typ*'')
by (*simp add*: *is-padding-tag-def padding-tag-def*)

lemma *wf-padding-padding-tag*: *wf-padding* (*padding-tag n*)
by (*simp add*: *padding-tag-def*)

lemma *wf-padding-tuple-padI*:
is-padding-tag s ⟹ *wf-padding-tuple* (*DTuple s* (*CHR* ''!''#*xs*) *d*)
by (*auto simp add*: *is-padding-tag-def wf-padding-padding-tag*)

lemma *wf-padding-tuple-no-padI*:
¬ *padding-field-name fn* ⟹ *wf-padding s* ⟹ *wf-padding-tuple* (*DTuple s fn d*)
by (*simp*)

lemma *wf-padding-extend-ti*:
fixes
t :: 'a *xtyp-info* **and**
st :: 'a *xtyp-info-struct* **and**
ts :: 'a *xtyp-info-tuple list* **and**
x :: 'a *xtyp-info-tuple*
shows
wf-padding t ⟹ *wf-padding-tuple* (*DTuple s fn d*) ⟹
wf-padding (*extend-ti t s n fn d*)

wf-padding-struct st $\Longrightarrow$ *wf-padding-tuple* (*DTuple s fn d*) $\Longrightarrow$
wf-padding-struct (*extend-ti-struct st s fn d*)

wf-padding-list ts $\Longrightarrow$ *wf-padding-tuple* (*DTuple s fn d*) $\Longrightarrow$
wf-padding-list (*ts* @ [*DTuple s fn d*])

wf-padding-tuple x $\Longrightarrow$ *wf-padding-tuple x*
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *is-padding-tag-update-desc*: *is-padding-tag t* $\Longrightarrow$
($\bigwedge x$. *upd* (*acc x*) *x* = *x*) $\Longrightarrow$
is-padding-tag
(*map-td* (λn *algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*)
by (*auto simp add*: *is-padding-tag-def padding-tag-def padding-desc-def update-desc-def fun-eq-iff*)

lemma *wf-padding-adjust-ti*:
fixes
t :: *'a xtyp-info* **and**
st :: *'a xtyp-info-struct* **and**
ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
assumes *upd-acc-id*: ($\bigwedge x$. *upd* (*acc x*) *x* = *x*)
shows
wf-padding t $\Longrightarrow$
wf-padding (*map-td* (λn *algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*)

wf-padding-struct st $\Longrightarrow$
wf-padding-struct (*map-td-struct* (λn *algn*. *update-desc acc upd*) (*update-desc acc upd*) *st*)

wf-padding-list ts $\Longrightarrow$
wf-padding-list (*map-td-list* (λn *algn*. *update-desc acc upd*) (*update-desc acc upd*) *ts*)

wf-padding-tuple x $\Longrightarrow$
wf-padding-tuple (*map-td-tuple* (λn *algn*. *update-desc acc upd*) (*update-desc acc upd*) *x*)
apply (*induct t* **and** *st* **and** *ts* **and** *x*)
by (*auto simp add*: *is-padding-tag-update-desc upd-acc-id*)

lemma *wf-padding-ti-pad-combine*: *wf-padding t* $\Longrightarrow$ *wf-padding* (*ti-pad-combine n t*)
by (*simp add*: *ti-pad-combine-def Let-def wf-padding-extend-ti*(*1*))

lemma (**in** *c-type*) *wf-padding-ti-typ-combine*:
wf-padding t $\Longrightarrow$ *wf-padding* (*typ-info-t TYPE*('*a*)) $\Longrightarrow$ ($\bigwedge xs$. *fn* $\neq$ *CHR ''!''*#*xs*)

$\Longrightarrow$ ($\bigwedge x.$ *upd* (*acc x*) *x* = *x*) $\Longrightarrow$
wf-padding (*ti-typ-combine* (*TYPE*('*a*)) *acc upd algn fn t*)
apply (*simp add*: *ti-typ-combine-def Let-def*)
apply (*rule wf-padding-extend-ti*(*1*))
apply *assumption*
apply (*simp add*: *adjust-ti-def wf-padding-adjust-ti*(*1*) *padding-field-name-def*)
done

lemma (**in** *c-type*) *wf-padding-ti-typ-pad-combine*:
wf-padding t $\Longrightarrow$ *wf-padding* (*typ-info-t TYPE*('*a*)) $\Longrightarrow$ ($\bigwedge xs.$ *fn* $\neq$ *CHR* ''!''#*xs*)
$\Longrightarrow$ ($\bigwedge x.$ *upd* (*acc x*) *x* = *x*) $\Longrightarrow$
wf-padding (*ti-typ-pad-combine* (*TYPE*('*a*)) *acc upd algn fn t*)
by (*simp add*: *ti-typ-pad-combine-def wf-padding-ti-typ-combine wf-padding-ti-pad-combine Let-def*)

lemma *wf-padding-map-align*: *wf-padding t* $\Longrightarrow$ *wf-padding* (*map-align f t*)
by (*cases t*) (*simp add*: *map-align-def*)

lemma *wf-padding-final-pad*: *wf-padding t* $\Longrightarrow$ *wf-padding* (*final-pad n t*)
by (*simp add*: *final-pad-def wf-padding-ti-pad-combine Let-def wf-padding-map-align*)

lemmas *wf-padding-combinator-simps* =
wf-padding-empty-typ-info
wf-padding-final-pad
wf-padding-ti-typ-pad-combine
wf-padding-ti-typ-combine

end

theory *ArraysMemInstance*
imports *Arrays CompoundCTypes*
begin

primrec (**in** *c-type*)
array-tag-n :: *nat* $\Rightarrow$ ('*a*,'*b*::*finite*) *array xtyp-info*
where
atn-base:
array-tag-n 0 = ((*empty-typ-info* (*align-td* (*typ-uinfo-t TYPE*('*a*))) (*typ-name* (*typ-uinfo-t TYPE*('*a*)) @ ''-array-'' @
nat-to-bin-string (*CARD*('*b*::*finite*))))::('*a*,'*b*) *array xtyp-info*)
| *atn-rec*:
array-tag-n (*Suc n*) = ((*ti-typ-combine TYPE*('*a*)
(*λx. index x n*) (*λx f. update f n x*) *0* (*replicate n CHR* ''1'')
(*array-tag-n n*))::('*a*,'*b*::*finite*) *array xtyp-info*)

definition (**in** *c-type*) *array-tag* :: ('*a*,'*b*::*finite*) *array itself* $\Rightarrow$ ('*a*,'*b*) *array xtyp-info*

where
array-tag t ≡ *array-tag-n* (*CARD*('*b*))

lemma (**in** *c-type*) *typ-name-array-tag*: *typ-name* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) =
(*typ-name* (*typ-uinfo-t TYPE*('*a*)) @ ''-array-'' @ *nat-to-bin-string* (*CARD*('*b*)))
apply (*induct n*)
apply (*simp add*: *empty-typ-info-def*)
apply (*simp add*: *ti-typ-combine-def Let-def adjust-ti-def*)
done

instantiation *array* :: (*c-type*, *finite*) *c-type*
begin

definition *typ-info-array*:
typ-info-t (*w*::('*a*::*c-type*,'*b*::*finite*) *array itself*) ≡ *array-tag w*

definition *typ-name-itself* (*w*::('*a*::*c-type*,'*b*::*finite*) *array itself*) = *typ-name* (*typ-info-t w*)

instance by (*intro-classes*) (*simp add*: *typ-name-itself-array-def*)
end

lemma (**in** *c-type*) *field-names-array-tag-length* [*rule-format*]:
x ∈ *set* (*field-names-list* (*array-tag-n n*)) ⟶ *length x* < *n*
by (*induct n*) (*auto*)

lemma (**in** *c-type*) *replicate-mem-field-names-array-tag* [*simp*]:
replicate n x ∉ *set* (*field-names-list* (*array-tag-n n*))
by (*fastforce dest*: *field-names-array-tag-length*)

lemma (**in** *c-type*) *aggregate-array-tag* [*simp*]:
aggregate (*array-tag-n n*)
by (*cases n*; *simp add*: *c-type.ti-typ-combine-def*)

lemma (**in** *mem-type*) *wf-desc-array-tag* [*simp*]:
wf-desc ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)
by (*induct n*; *simp*) (*fastforce elim*: *wf-desc-ti-typ-combine*)

lemma (**in** *mem-type*) *wf-size-desc-array-tag* [*simp*]:
0 < *n* ⟹ *wf-size-desc* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)
apply(*induct n*; *simp*)
subgoal for *n*
apply(*cases n=0*; *simp*)
apply(*rule wf-size-desc-ti-typ-combine*)
apply *simp*
done
done

lemma (**in** *mem-type*) *upd-ind-array-tag-udpate* [*simp*]:
⟦ $n \leq m$; $n \leq CARD('b)$ ⟧ ⟹
upd-ind (*lf-set* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) []) (λ*x f*. *update f m x*)
by (*induct n*) (*auto elim*: *upd-ind-ti-typ-combine*)

lemma (**in** *mem-type*) *fc-array-tag-udpate* [*simp*]:
⟦ $n \leq m$; $n \leq CARD('b)$ ⟧ ⟹
fu-commutes (*update-ti-t* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)) (λ*x f*. *update f m x*)
by (*induct n*; *fastforce elim*: *fc-ti-typ-combine simp*: *fg-cons-def*)

lemma (**in** *mem-type*) *acc-ind-array-tag-udpate* [*simp*]:
⟦ $n \leq m$; $m < CARD('b)$ ⟧ ⟹
acc-ind (λ*x*. *index x m*) (*lf-fd* ‘ *lf-set* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) [])
by (*induct n*; *fastforce elim*: *acc-ind-ti-typ-combine*)

lemma (**in** *mem-type*) *fa-fu-g-array-tag-udpate* [*simp*]:
⟦ $n \leq m$; $m < CARD('b)$ ⟧ ⟹
fa-ind (*lf-fd* ‘ *lf-set* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) []) (λ*x f*. *update f m x*)
by (*induct n*; *fastforce elim*: *fa-ind-ti-typ-combine*)

lemma (**in** *mem-type*) *wf-fdp-array-tag* [*simp*]:
$n \leq CARD('b)$ ⟹ *wf-lf* (*lf-set* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) [])
by (*induct n*; *fastforce elim*: *wf-lf-ti-typ-combine*)

lemma *upd-local-update* [*simp*]:
upd-local (λ*x f*. *update f n x*)
unfolding *upd-local-def*
by (*metis update-update*)

lemma (**in** *mem-type*) *fu-eq-mask-array-tag* [*simp*, *rule-format*]:
$n \leq CARD('b) \longrightarrow (\forall m.\ (\forall k\ v.\ k < CARD('b) \longrightarrow$
index ((*m v*)::('*a*,'*b*) *array*) *k* = (*if n* ≤ *k then*
index (*undefined*::('*a*,'*b*::*finite*) *array*) *k*
else index v k)) ⟶ *fu-eq-mask* (*array-tag-n n*) *m*)
apply(*induct n*; *clarsimp*)
apply(*rule fu-eq-mask-empty-typ-info*)
apply(*clarsimp simp*: *array-index-eq*)
subgoal for *n m*
apply(*rule fu-eq-mask-ti-typ-combine*; *clarsimp?*)
apply (*drule spec* [**where** *x*=λ*v*. *update* (*m v*) *n* (*index* (*undefined*::'*a*['*b*]) *n*)])
apply(*erule impE*)
apply *clarsimp*
subgoal for *k v*

```
    by (cases k=n) auto
  subgoal premises prems
  proof −
    from prems
    have ∀ v bs. m (update v n bs) = update (m v) n bs
      apply(clarsimp simp: array-index-eq)
      subgoal for v bs i
        apply(cases i=n; clarsimp)
        apply(cases Suc n ≤ i; clarsimp)
        done
      done
    then show ?thesis using prems  by clarsimp
  qed
  subgoal
    by(clarsimp simp: fg-cons-def)
  done
 done

lemma (in c-type) size-td-array-tag [simp]:
  size-td (((array-tag-n n)::('a,'b::finite) array xtyp-info)) =
     n ∗ size-of TYPE('a)
  by (induct n; simp add: size-td-lt-ti-typ-combine size-of-def)

lemma (in c-type) align-td-wo-align-array-tag:
  0 < n ⟹
  align-td-wo-align ((array-tag-n n)::('a,'b::finite) array xtyp-info) = (align-td-wo-align
(typ-info-t (TYPE('a))))
proof (induct n)
  case 0
  then show ?case by clarsimp
next
  case (Suc n)
  then show ?case
    by (cases n = 0) (auto simp: align-of-def max-def)
qed

lemma align-td-export-uinfo[simp]: align-td (export-uinfo t) = align-td t
  apply (cases t)
  apply (simp add: export-uinfo-def)
  done

lemma (in c-type) align-td-uinfo: align-td (typ-uinfo-t (TYPE('a))) = align-td
(typ-info-t (TYPE('a)))
  by (simp add: typ-uinfo-t-def)

lemma (in c-type) align-td-array-tag:
  0 < n ⟹
   align-td ((array-tag-n n)::('a,'b::finite) array xtyp-info) = (align-td (typ-info-t
(TYPE('a))))
```

proof (*induct n*)
case *0*
then show *?case* **by** *clarsimp*
next
case (*Suc n*)
then show *?case*
by (*cases n = 0*) (*auto simp*: *align-of-def max-def align-td-uinfo*)
qed

lemma *align-of-array* [*simp*]:
0 < CARD('b) ⟹ align-of TYPE(('a,'b::finite) array) = align-of TYPE('a::c-type)
by (*simp add*: *align-of-def typ-info-array array-tag-def align-td-array-tag*)

lemma *align-td-wo-align-array-info*: *0 < CARD('b) ⟹ align-td-wo-align* (*typ-info-t TYPE(('a,'b::finite) array)*)
= align-td-wo-align (*typ-info-t TYPE('a::c-type)*)
by (*simp add*: *typ-info-array array-tag-def align-td-wo-align-array-tag*)

lemma *align-td-array-info*: *0 < CARD('b) ⟹ align-td* (*typ-info-t TYPE(('a,'b::finite) array)*)
= align-td (*typ-info-t TYPE('a::c-type)*)
by (*simp add*: *typ-info-array array-tag-def align-td-array-tag*)

lemma (**in** *mem-type*) *align-field-array* [*simp*]:
align-field ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
by *clarsimp* (*metis align-field-ti-typ-combine align-of-def align-size-of dvd-mult size-td-array-tag*)
qed

lemma (**in** *mem-type*) *wf-align-array* [*simp*]:
wf-align ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)
proof (*induct n*)
case *0*
then show *?case* **by** (*simp add*: *wf-align-simps*)
next
case (*Suc n*)
then show *?case* **by** (*simp add*: *wf-align-ti-typ-combine*)
qed

instance *array* :: (*mem-type*,*finite*) *mem-type-sans-size*
apply *intro-classes*

apply(*simp-all add*: *typ-info-array array-tag-def size-of-def norm-bytes-def*)
apply *clarsimp*
apply(*rule fu-eq-mask*)
apply(*simp add*: *size-of-def*)
apply(*rule fu-eq-mask-array-tag*; *simp*)
apply (*clarsimp simp*: *align-of-def typ-info-array array-tag-def align-td-wo-align-array-tag*)
apply (*metis align-of-def align-size-of dvd-mult size-of-def*)
done

declare *atn-base* [*simp del*]
declare *atn-rec* [*simp del*]

lemma *size-of-array* [*simp*]:
size-of TYPE(('*a*,'*b*::*finite*) *array*) = *CARD*('*b*) ∗ *size-of TYPE*('*a*::*c-type*)
by (*simp add*: *size-of-def typ-info-array array-tag-def*)

lemma *size-td-array*:
size-td (*typ-info-t TYPE*(('*a*,'*b*::*finite*) *array*)) = *CARD*('*b*) ∗ *size-of TYPE*('*a*::*c-type*)
by (*simp add*: *size-of-def typ-info-array array-tag-def*)

lemma *align-td-array*:
2^*align-td* (*typ-info-t TYPE*(('*a*,'*b*::*finite*) *array*)) = *align-of TYPE*('*a*::*c-type*)
by (*simp add*: *align-of-def typ-info-array array-tag-def align-td-array-tag*)

lemma *wf-field-array*:
n < *CARD*('*b*) ⟹ *wf-field* (λ*x*. *x*.[*n*]) (λ*x f*. *update* (*f*::('*a*,'*b*::*finite*) *array*) *n x*)
by (*simp add*: *wf-field-def*)

lemma *wf-cfield-array*:
n < *CARD*('*b*) ⟹ *wf-cfield* (λ*x*. *x*.[*n*]) (λ*x f*. *update* (*f*::('*a*::*c-type*,'*b*::*finite*) *array*) *n x*)
by (*simp add*: *wf-cfield-def wf-field-array*)

lemma *wf-xfield-array*:
n < *CARD*('*b*) ⟹ *wf-xfield* (λ*x*. *x*.[*n*]) (λ*x f*. *update* (*f*::('*a*::*xmem-type*,'*b*::*finite*) *array*) *n x*)
by (*simp add*: *wf-xfield-def wf-cfield-array*)

lemma *wf-component-descs-array-tag-n*: *n* ≤ *CARD*('*b*)
⟹ *wf-component-descs* ((*array-tag-n*::*nat* ⇒ ('*a*::*xmem-type*,'*b*::*finite*) *array xtyp-info*) *n*)
apply (*induct n*)
apply (*simp add*: *atn-base*)
apply (*simp add*: *atn-rec*)
apply (*rule wf-xfield.wf-component-descs-ti-typ-combine*)
apply (*simp add*: *wf-xfield-array*)
apply *simp*

done

lemma *wf-component-descs-array*: *wf-component-descs* (*typ-info-t TYPE*('*a*::*xmem-type*['*b*::*finite*]))
apply (*simp add*: *typ-info-array array-tag-def*)
apply (*rule wf-component-descs-array-tag-n*)
apply *simp*
done

lemma (**in** *c-type*) *set-toplevel-field-descs-array-tag-n*:
(*set* (*toplevel-field-descs* ((*array-tag-n*::*nat* ⇒ ('*a*,'*b*::*finite*) *array xtyp-info*) *n*)))
=
{*d*. ∃ *m*. *m* < *n* ∧ *d* = (|*field-access* = *xto-bytes* ∘ (λ*x*. *index x m*),
field-update = (λ*x f*. *update f m x*) ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*a*)|)} (**is** - = *?D n*)
proof (*induct n*)
case *0*
then show *?case* **by** (*simp add*: *atn-base empty-typ-info-def*)
next
case (*Suc n*)
from *Suc.hyps* **have** *hyp*: *set* (*toplevel-field-descs* (*array-tag-n n*)) = *?D n* **.**

show *?case*
proof
show *set* (*toplevel-field-descs* (*array-tag-n* (*Suc n*))) ⊆ *?D* (*Suc n*)
proof
fix *d*
assume *d-in*: *d* ∈ *set* (*toplevel-field-descs* ((*array-tag-n*::*nat* ⇒ ('*a*,'*b*::*finite*) *array xtyp-info*) (*Suc n*)))
show *d* ∈ *?D* (*Suc n*)
proof −
from *d-in* **consider**
(*d-new*) *d* = (|*field-access* = *xto-bytes* ∘ (λ*x*. *x*.[*n*]), *field-update* = (λ*x f*. *update f n x*) ∘ *xfrom-bytes*, *field-sz* = *size-of TYPE*('*a*)|) |
(*d-old*) *d* ∈ *set* (*toplevel-field-descs* (*array-tag-n n*))
by (*auto simp add*: *set-toplevel-field-descs-ti-typ-combine-aggregate atn-rec*)
then show *?thesis*
proof (*cases*)
case *d-new*
then show *?thesis* **by** (*auto simp add*: *comp-def*)
next
case *d-old*
with *hyp less-Suc-eq* **show** *?thesis* **by** (*auto*)
qed
qed
qed
next
show *?D* (*Suc n*) ⊆ *set* (*toplevel-field-descs* (*array-tag-n* (*Suc n*)))
proof

```
      fix d
      assume d-in: d ∈ ?D (Suc n)
       show d ∈ set (toplevel-field-descs ((array-tag-n::nat ⇒ ('a,'b::finite) array
xtyp-info) (Suc n)))
      proof −
        from d-in obtain m where m-bound: m < Suc n and
         d: d = (|field-access = xto-bytes ∘ (λx. x.[m]),
                  field-update = (λx f. update f m x) ∘ xfrom-bytes,
                  field-sz = size-of TYPE('a)|) by (auto simp add: comp-def)
        from m-bound d show ?thesis
          using hyp
          apply (simp add: set-toplevel-field-descs-ti-typ-combine-aggregate atn-rec)
          using not-less-less-Suc-eq by fastforce
      qed
    qed
  qed
qed

lemma (in xmem-type) field-desc-independent-extend-array:
 n < CARD('b) ⟹
        field-desc-independent (xto-bytes ∘ (λx. x.[n]))
         ((λx f. update (f::('a,'b::finite) array) n x) ∘ xfrom-bytes)
         (set (toplevel-field-descs (array-tag-n n)))
  apply (simp add: set-toplevel-field-descs-array-tag-n)
  apply (rule field-desc-independent.intro)
    apply (auto simp add: fu-commutes-def)
  done


lemma component-descs-independent-array-tag-n: n ≤ CARD('b)
⟹ component-descs-independent ((array-tag-n::nat ⇒ ('a::xmem-type,'b::finite)
array xtyp-info) n)
  apply (induct n)
   apply (simp add: atn-base)
  apply (simp add: atn-rec)
  apply (rule wf-xfield.component-descs-independent-ti-typ-combine)
    apply (simp add: wf-xfield-array)
   apply simp
  apply (rule field-desc-independent-extend-array)
  apply simp
  done


lemma component-descs-independent-array: component-descs-independent (typ-info-t
TYPE('a::xmem-type['b::finite]))
  apply (simp add: typ-info-array array-tag-def)
  apply (rule component-descs-independent-array-tag-n)
  apply simp
  done
```

lemma *complement-padding-extend-array*: $n < CARD('b) \Longrightarrow$
complement-padding (*xto-bytes* $\circ$ ($\lambda x.\ x.[n]$))
(($\lambda x\ f.$ *update* (f::('*a*::*xmem-type*,'*b*::*finite*) *array*) *n x*) $\circ$ *xfrom-bytes*) (*size-of* *TYPE*('*a*))
apply (*unfold-locales*)
by (*simp add*: *complement-padding.complement wf-cfield.intro wf-field-def wf-xfield.intro wf-xfield.padding-lift*)

lemma *wf-field-desc-extend-array*: $n <$ *CARD*('*b*) $\Longrightarrow$ *wf-field-desc*
(|*field-access* = *xto-bytes* $\circ$ ($\lambda x.\ x.[n]$),
field-update = ($\lambda x\ f.$ *update* (f::('*a*::*xmem-type*,'*b*::*finite*) *array*) *n x*) $\circ$ *xfrom-bytes*,
field-sz = *size-of* *TYPE*('*a*::*xmem-type*)|)
apply (*intro-locales*)
apply *simp*
apply (*rule complement-padding-extend-array*, *assumption*)
apply (*unfold-locales*)
by (*auto simp add*: *xfrom-bytes-xto-bytes-inv size-of-def xto-bytes-result-size xto-bytes-size xfrom-bytes-size*)

lemma (**in** *xmem-type*) *wf-field-desc-adjust-array-field*: $n <$ *CARD*('*b*) $\Longrightarrow$
wf-field-descs
(*set* (*field-descs*
(*adjust-ti* (*typ-info-t* *TYPE*('*a*)) ($\lambda x.\ x.[n]$)
($\lambda x\ f.$ *update* (f::('*a*,'*b*::*finite*) *array*) *n x*))))
apply (*rule wf-field.wf-field-descs-adjust-ti*)
apply (*rule wf-field-array*, *assumption*)
apply (*simp add*: *wf-field-descs*)
done

lemma *wf-field-descs-array-tag-n*: $n \leq$ *CARD*('*b*)
$\Longrightarrow$ *wf-field-descs* (*set* (*field-descs* ((*array-tag-n*::*nat* $\Rightarrow$ ('*a*::*xmem-type*,'*b*::*finite*) *array xtyp-info*) *n*)))
apply (*induct n*)
apply (*simp add*: *atn-base*)
apply (*simp add*: *atn-rec*)
apply (*simp add*: *set-field-descs-ti-typ-combine-aggregate*)
apply (*simp add*: *wf-field-desc-extend-array wf-field-desc-adjust-array-field*)
done

lemma *wf-field-descs-array*: *wf-field-descs* (*set* (*field-descs* (*typ-info-t* *TYPE*('*a*::*xmem-type*['*b*::*finite*]))))
apply (*simp add*: *typ-info-array array-tag-def*)
apply (*rule wf-field-descs-array-tag-n*)
apply *simp*
done

lemma (**in** *xmem-contained-type*) *contained-field-descs-array-tag-n*:
contained-field-descs ((*array-tag-n*::*nat* ⇒ ('*a*,'*b*::*finite*) *array xtyp-info*) *n*)
apply (*induct n*)
apply (*simp add*: *atn-base*)
apply (*simp add*: *atn-rec*)
apply (*rule contained-field-descs-ti-typ-combine*)
apply *simp*
done

lemma *contained-field-descs-array*: *contained-field-descs* (*typ-info-t TYPE*('*a*::*xmem-contained-type*['*b*::*finite*])
apply (*simp add*: *typ-info-array array-tag-def*)
apply (*rule contained-field-descs-array-tag-n*)
done

lemma *replicate-1-neq-padding*: *replicate n CHR ''1'' ≠ CHR ''!'' # xs*
by (*cases n*) *auto*

lemma (**in** *xmem-type*) *wf-padding-array-tag-n*: *n* ≤ *CARD*('*b*)
⟹ *wf-padding* ((*array-tag-n*::*nat* ⇒ ('*a*,'*b*::*finite*) *array xtyp-info*) *n*)
apply (*induct n*)
apply (*simp add*: *atn-base wf-padding-combinator-simps*)
apply (*simp add*: *atn-rec wf-padding-ti-typ-combine wf-padding replicate-1-neq-padding*)
done

lemma *wf-padding-array*: *wf-padding* (*typ-info-t TYPE*('*a*::*xmem-type*['*b*::*finite*]))
apply (*simp add*: *typ-info-array array-tag-def*)
apply (*rule wf-padding-array-tag-n*)
apply *simp*
done

end

theory *ArchArraysMemInstance*
imports *ArraysMemInstance*
begin

class *array-outer-max-size* = *xmem-contained-type* +
assumes *array-outer-max-size-ax*: *size-of TYPE*('*a*) < *2* ^ *array-outer-max-size-exponent*

class *array-max-count* = *finite* +
assumes *array-max-count-ax*: *CARD* ('*a*) <= *2* ^ *array-outer-max-count-exponent*

instance *array* :: (*array-outer-max-size*, *array-max-count*) *mem-type*
apply *intro-classes*

apply *simp*
apply (*subgoal-tac addr-card = 2 ^(addr-bitsize − array-outer-max-size-exponent)*
∗ 2 ^ array-outer-max-size-exponent)
apply (*erule ssubst*)
apply (*rule less-le-trans*[**where** *y = card* (*UNIV*::*'b set*) *∗ 2 ^array-outer-max-size-exponent*])
apply (*rule mult-less-mono2*)
apply (*rule array-outer-max-size-ax*)
apply *simp*
apply (*rule mult-le-mono1*)
apply (*rule le-trans*[**where** *j = 2 ^ array-outer-max-count-exponent*])
apply (*rule array-max-count-ax*)
apply *simp*
apply *simp*
apply (*simp add*: *addr-card*)
done

instance *array* :: (*array-outer-max-size*, *array-max-count*) *xmem-contained-type*
apply *intro-classes*
apply (*rule wf-component-descs-array*)
apply (*rule component-descs-independent-array*)
apply (*rule wf-field-descs-array*)
apply (*rule wf-padding-array*)
apply (*rule contained-field-descs-array*)
done

class *array-inner-max-size = array-outer-max-size +*
assumes *array-inner-max-size-ax*: *size-of TYPE*(*'a*) *< 2 ^array-inner-max-size-exponent*

instance *array* :: (*array-inner-max-size*, *array-max-count*) *array-outer-max-size*
apply *intro-classes*
apply *simp*
apply (*rule order-less-le-trans*)
apply (*rule mult-le-less-imp-less*)
apply (*rule array-max-count-ax*)
apply (*rule array-inner-max-size-ax*)
apply *simp*
apply *simp*
apply *simp*
done

instance *word* :: (*len8*) *array-outer-max-size*
apply *intro-classes*
apply(*simp add*: *size-of-def*)
apply(*subgoal-tac len-of TYPE*(*'a*) *≤ 128*)
apply *simp*
apply(*rule len8-width*)
done

instance *word* :: (*len8*) *array-inner-max-size*
apply *intro-classes*
apply(*simp add*: *size-of-def*)
apply(*subgoal-tac len-of TYPE*('*a*) ≤ *128*)
apply *simp*
apply(*rule len8-width*)
done

instance *ptr* :: (*c-type*) *array-outer-max-size*
apply *intro-classes*
apply (*simp add*: *size-of-def*)
done

instance *ptr* :: (*c-type*) *array-inner-max-size*
apply *intro-classes*
apply (*simp add*: *size-of-def*)
done

class *lt19* = *finite* +
assumes *lt19-ax*: *CARD* ('*a*) < *2* ^ *19*
class *lt18* = *lt19* +
assumes *lt18-ax*: *CARD* ('*a*) < *2* ^ *18*
class *lt17* = *lt18* +
assumes *lt17-ax*: *CARD* ('*a*) < *2* ^ *17*
class *lt16* = *lt17* +
assumes *lt16-ax*: *CARD* ('*a*) < *2* ^ *16*
class *lt15* = *lt16* +
assumes *lt15-ax*: *CARD* ('*a*) < *2* ^ *15*
class *lt14* = *lt15* +
assumes *lt14-ax*: *CARD* ('*a*) < *2* ^ *14*
class *lt13* = *lt14* +
assumes *lt13-ax*: *CARD* ('*a*) < *2* ^ *13*
class *lt12* = *lt13* +
assumes *lt12-ax*: *CARD* ('*a*) < *2* ^ *12*
class *lt11* = *lt12* +
assumes *lt11-ax*: *CARD* ('*a*) < *2* ^ *11*
class *lt10* = *lt11* +
assumes *lt10-ax*: *CARD* ('*a*) < *2* ^ *10*
class *lt9* = *lt10* +
assumes *lt9-ax*: *CARD* ('*a*) < *2* ^ *9*
class *lt8* = *lt9* +
assumes *lt8-ax*: *CARD* ('*a*) < *2* ^ *8*
class *lt7* = *lt8* +
assumes *lt7-ax*: *CARD* ('*a*) < *2* ^ *7*
class *lt6* = *lt7* +
assumes *lt6-ax*: *CARD* ('*a*) < *2* ^ *6*
class *lt5* = *lt6* +
assumes *lt5-ax*: *CARD* ('*a*) < *2* ^ *5*

class *lt4* = *lt5* +
  assumes *lt4-ax*: *CARD* ('*a*) < *2* ^ *4*
class *lt3* = *lt4* +
  assumes *lt3-ax*: *CARD* ('*a*) < *2* ^ *3*
class *lt2* = *lt3* +
  assumes *lt2-ax*: *CARD* ('*a*) < *2* ^ *2*
class *lt1* = *lt2* +
  assumes *lt1-ax*: *CARD* ('*a*) < *2* ^ *1*

instance *bit0* :: (*lt19*) *array-max-count*
  using *lt19-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit1* :: (*lt19*) *array-max-count*
  using *lt19-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit0* :: (*lt18*) *lt19*
  using *lt18-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit1* :: (*lt18*) *lt19*
  using *lt18-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit0* :: (*lt17*) *lt18*
  using *lt17-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit1* :: (*lt17*) *lt18*
  using *lt17-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit0* :: (*lt16*) *lt17*
  using *lt16-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit1* :: (*lt16*) *lt17*
  using *lt16-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit0* :: (*lt15*) *lt16*
  using *lt15-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit1* :: (*lt15*) *lt16*
  using *lt15-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit0* :: (*lt14*) *lt15*
using *lt14-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit1* :: (*lt14*) *lt15*
using *lt14-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit0* :: (*lt13*) *lt14*
using *lt13-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit1* :: (*lt13*) *lt14*
using *lt13-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit0* :: (*lt12*) *lt13*
using *lt12-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit1* :: (*lt12*) *lt13*
using *lt12-ax*[**where** *'a='a*] **if-architecture-by** (*ARM64*, *RISCV64*, *X64*) *intro-classes simp*

instance *bit0* :: (*lt12*) *array-max-count*
using *lt12-ax*[**where** *'a='a*] **if-architecture-by** (*ARM*, *ARM-HYP*) *intro-classes simp*

instance *bit1* :: (*lt12*) *array-max-count*
using *lt12-ax*[**where** *'a='a*] **if-architecture-by** (*ARM*, *ARM-HYP*) *intro-classes simp*

instance *bit0* :: (*lt11*) *lt12*
using *lt11-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt11*) *lt12*
using *lt11-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt10*) *lt11*
using *lt10-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt10*) *lt11*
using *lt10-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt9*) *lt10*
using *lt9-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt9*) *lt10*
using *lt9-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt8*) *lt9*
using *lt8-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt8*) *lt9*
using *lt8-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt7*) *lt8*
using *lt7-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt7*) *lt8*
using *lt7-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt6*) *lt7*
using *lt6-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt6*) *lt7*
using *lt6-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt5*) *lt6*
using *lt5-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt5*) *lt6*
using *lt5-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt4*) *lt5*
using *lt4-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt4*) *lt5*
using *lt4-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt3*) *lt4*
using *lt3-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt3*) *lt4*
using *lt3-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt2*) *lt3*
using *lt2-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt2*) *lt3*
using *lt2-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit0* :: (*lt1*) *lt2*
using *lt1-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *bit1* :: (*lt1*) *lt2*

using *lt1-ax*[**where** *'a='a*] **by** *intro-classes simp-all*

instance *num1* :: *lt1*
by (*intro-classes*, *simp-all*)

instance *num1* :: *array-max-count*
by (*intro-classes*, *simp*)

end

theory *HeapRawState*
imports *CTypes*
begin

type-synonym *typ-base* = *bool*
datatype *s-heap-index* = *SIndexVal* | *SIndexTyp nat*
datatype *s-heap-value* = *SValue byte* | *STyp typ-uinfo* × *typ-base*

primrec (*nonexhaustive*) *s-heap-tag* :: *s-heap-value* ⇒ *typ-uinfo* × *typ-base* **where**
s-heap-tag (*STyp t*) = *t*

type-synonym *typ-slice* = *nat* ⇀ *typ-uinfo* × *typ-base*

type-synonym *s-addr* = *addr* × *s-heap-index*
type-synonym *heap-state* = *s-addr* ⇀ *s-heap-value*
type-synonym *heap-typ-desc* = *addr* ⇒ *bool* × *typ-slice*
type-synonym *heap-raw-state* = *heap-mem* × *heap-typ-desc*

definition *hrs-mem* :: *heap-raw-state* ⇒ *heap-mem* **where**
hrs-mem ≡ *fst*

definition *hrs-mem-update* :: (*heap-mem* ⇒ *heap-mem*) ⇒ *heap-raw-state* ⇒ *heap-raw-state*
where
hrs-mem-update f ≡ λ(*h*,*d*). (*f h*,*d*)

definition *hrs-htd* :: *heap-raw-state* ⇒ *heap-typ-desc* **where**
hrs-htd ≡ *snd*

definition *hrs-htd-update* :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *heap-raw-state* ⇒
heap-raw-state
where
hrs-htd-update f ≡ λ(*h*,*d*). (*h*,*f d*)

lemma *hrs-comm*:
hrs-htd-update d (*hrs-mem-update h s*) = *hrs-mem-update h* (*hrs-htd-update d s*)
by (*simp add*: *hrs-htd-update-def hrs-mem-update-def split-def*)

lemma *hrs-htd-update-htd-update*:
(λ*s*. *hrs-htd-update d* (*hrs-htd-update d′ s*)) = *hrs-htd-update* (*d* ∘ *d′*)
by (*simp add*: *hrs-htd-update-def split-def*)

lemma *hrs-htd-mem-update* [*simp*]:
hrs-htd (*hrs-mem-update f s*) = *hrs-htd s*
by (*simp add*: *hrs-mem-update-def hrs-htd-def split-def*)

lemma *hrs-mem-htd-update* [*simp*]:
hrs-mem (*hrs-htd-update f s*) = *hrs-mem s*
by (*simp add*: *hrs-htd-update-def hrs-mem-def split-def*)

lemma *hrs-mem-update*:
hrs-mem (*hrs-mem-update f s*) = (*f* (*hrs-mem s*))
by (*simp add*: *hrs-mem-update-def hrs-mem-def split-def*)

lemma *hrs-htd-update*:
hrs-htd (*hrs-htd-update f s*) = (*f* (*hrs-htd s*))
by (*simp add*: *hrs-htd-update-def hrs-htd-def split-def*)

lemmas *hrs-update* = *hrs-mem-update hrs-htd-update*

lemma *hrs-htd-update-comp*: *hrs-htd-update f* ∘ *hrs-htd-update g* = *hrs-htd-update* (*f o g*)
by (*auto simp add*: *hrs-htd-update-def split*: *prod.splits*)

lemma *hrs-mem-update-comp*: *hrs-mem-update f* ∘ *hrs-mem-update g* = *hrs-mem-update* (*f o g*)
by (*auto simp add*: *hrs-mem-update-def split*: *prod.splits*)

lemma *hrs-update-commute*:
hrs-mem-update f ∘ *hrs-htd-update g* = *hrs-htd-update g o hrs-mem-update f*
by (*auto simp add*: *hrs-mem-update-def hrs-htd-update-def split*: *prod.splits*)

end

11.20 More properties of maps plus map disjuction.

theory *MapExtra*
imports *Main*
begin

BEWARE: we are not interested in using the $dom\ x \cap dom\ y = \{\}$ rules from Map for our separation logic proofs. As such, we overwrite the Map rules where that form of disjointness is in the assumption conflicts with a name we want to use with $\bot$.

A note on naming: Anything not involving heap disjuction can potentially be incorporated directly into Map.thy, thus uses m. Anything involving heap disjunction is not really mergeable with Map, is destined for use in separation logic, and hence uses h

---

Things that should go into Option Type

Misc option lemmas

lemma *None-not-eq*: $(None \neq x) = (\exists\, y.\ x = Some\ y)$ **by** $(cases\ x)\ auto$

lemma *None-com*: $(None = x) = (x = None)$ **by** *fast*

lemma *Some-com*: $(Some\ y = x) = (x = Some\ y)$ **by** *fast*

Things that should go into Map.thy

Map intersection: set of all keys for which the maps agree.

definition
$map\text{-}inter :: ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'a\ set$ (**infixl** $\langle\cap_m\rangle$ *70*) **where**
$m_1 \cap_m m_2 \equiv \{x \in dom\ m_1.\ m_1\ x = m_2\ x\}$

Map restriction via domain subtraction

definition
$sub\text{-}restrict\text{-}map :: ('a \rightharpoonup 'b) => 'a\ set => ('a \rightharpoonup 'b)$ (**infixl** $\langle$`–$\rangle$ *110*)
where
m `– $S \equiv (\lambda x.\ if\ x \in S\ then\ None\ else\ m\ x)$

11.20.1 Properties of maps not related to restriction

lemma *empty-forall-equiv*: $(m = Map.empty) = (\forall\, x.\ m\ x = None)$
by (*rule fun-eq-iff*)

lemma *map-le-empty2* [*simp*]:
$(m \subseteq_m Map.empty) = (m = Map.empty)$
by (*auto simp*: *map-le-def*)

lemma *dom-iff*:
$(\exists\, y.\ m\ x = Some\ y) = (x \in dom\ m)$

by *auto*

lemma *non-dom-eval*:
$x \notin dom\ m \Longrightarrow m\ x = None$
by *auto*

lemma *non-dom-eval-eq*:
$x \notin dom\ m = (m\ x = None)$
by *auto*

lemma *map-add-same-left-eq*:
$m_1 = m_1{'} \Longrightarrow (m_0 ++ m_1 = m_0 ++ m_1{'})$
by *simp*

lemma *map-add-left-cancelI* [*intro*!]:
$m_1 = m_1{'} \Longrightarrow m_0 ++ m_1 = m_0 ++ m_1{'}$
by *simp*

lemma *dom-empty-is-empty*:
$(dom\ m = \{\}) = (m = Map.empty)$
proof (*rule iffI*)
assume *a*: $dom\ m = \{\}$
{ assume $m \neq Map.empty$
hence $dom\ m \neq \{\}$
by − (*subst* (*asm*) *empty-forall-equiv*, *simp add*: *dom-def*)
hence *False* **using** *a* **by** *blast*
}
thus $m = Map.empty$ **by** *blast*
next
assume *a*: $m = Map.empty$
thus $dom\ m = \{\}$ **by** *simp*
qed

lemma *map-add-dom-eq*:
$dom\ m = dom\ m' \Longrightarrow m ++ m' = m'$
by (*rule ext*) (*auto simp*: *map-add-def split*: *option.splits*)

lemma *map-add-right-dom-eq*:
$[\![\ m_0 ++ m_1 = m_0{'} ++ m_1{'};\ dom\ m_1 = dom\ m_1{'}\ ]\!] \Longrightarrow m_1 = m_1{'}$
unfolding *map-add-def*
apply (*rule ext*)
apply (*rule ccontr*)
subgoal for *x*
apply (*drule fun-cong* [**where** *x*=*x*], *clarsimp split*: *option.splits*)
apply (*drule sym*, *drule sym*, *force*+)
done
done

lemma *map-le-same-dom-eq*:

$[\![m_0 \subseteq_m m_1 ; dom\ m_0 = dom\ m_1]\!] \Longrightarrow m_0 = m_1$
by (*simp add*: *map-le-antisym map-le-def*)

11.20.2 Properties of map restriction

lemma *restrict-map-cancel*:
$(m \mid` S = m \mid` T) = (dom\ m \cap S = dom\ m \cap T)$
by (*fastforce intro*: *set-eqI dest*: *fun-cong*
simp: *restrict-map-def None-not-eq*
split: *if-split-asm*)

lemma *map-add-restricted-self* [*simp*]:
$m ++ m \mid` S = m$
by (*auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

lemma *map-add-restrict-dom-right* [*simp*]:
$(m ++ m') \mid` dom\ m' = m'$
by (*rule ext*, *auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

lemma *restrict-map-UNIV* [*simp*]:
$m \mid` UNIV = m$
by (*simp add*: *restrict-map-def*)

lemma *restrict-map-dom*:
$S = dom\ m \Longrightarrow m \mid` S = m$
by (*fastforce simp*: *restrict-map-def None-not-eq*)

lemma *restrict-map-subdom*:
$dom\ m \subseteq S \Longrightarrow m \mid` S = m$
by (*fastforce simp*: *restrict-map-def None-com*)

lemma *map-add-restrict*:
$(m_0 ++ m_1) \mid` S = ((m_0 \mid` S) ++ (m_1 \mid` S))$
by (*force simp*: *map-add-def restrict-map-def*)

lemma *map-le-restrict*:
$m \subseteq_m m' \Longrightarrow m = m' \mid` dom\ m$
by (*force simp*: *map-le-def restrict-map-def None-com*)

lemma *restrict-map-le*:
$m \mid` S \subseteq_m m$
by (*auto simp*: *map-le-def*)

lemma *restrict-map-remerge*:
$[\![S \cap T = \{\}]\!] \Longrightarrow m \mid` S ++ m \mid` T = m \mid` (S \cup T)$
by (*rule ext*, *clarsimp simp*: *restrict-map-def map-add-def*
split: *option.splits*)

lemma *restrict-map-empty*:

$dom\ m \cap S = \{\} \Longrightarrow m \mid` S = Map.empty$
by (*fastforce simp*: *restrict-map-def*)

lemma *map-add-restrict-comp-right* [*simp*]:
$(m \mid` S ++ m \mid` (UNIV - S)) = m$
by (*force simp*: *map-add-def restrict-map-def split*: *option.splits*)

lemma *map-add-restrict-comp-right-dom* [*simp*]:
$(m \mid` S ++ m \mid` (dom\ m - S)) = m$
by (*fastforce simp*: *map-add-def restrict-map-def split*: *option.splits*)

lemma *map-add-restrict-comp-left* [*simp*]:
$(m \mid` (UNIV - S) ++ m \mid` S) = m$
by (*subst map-add-comm*, *auto*)

lemma *restrict-self-UNIV*:
$m \mid` (dom\ m - S) = m \mid` (UNIV - S)$
by (*fastforce simp*: *restrict-map-def*)

lemma *map-add-restrict-nonmember-right*:
$x \notin dom\ m' \Longrightarrow (m ++ m') \mid` \{x\} = m \mid` \{x\}$
by (*rule ext*, *auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

lemma *map-add-restrict-nonmember-left*:
$x \notin dom\ m \Longrightarrow (m ++ m') \mid` \{x\} = m' \mid` \{x\}$
by (*rule ext*, *auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

lemma *map-add-restrict-right*:
$x \subseteq dom\ m' \Longrightarrow (m ++ m') \mid` x = m' \mid` x$
by (*rule ext*, *auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

lemma *restrict-map-compose*:
$[\![S \cup T = dom\ m\ ;\ S \cap T = \{\}]\!] \Longrightarrow m \mid` S ++ m \mid` T = m$
by (*fastforce simp*: *map-add-def restrict-map-def*)

lemma *map-le-dom-subset-restrict*:
$[\![m' \subseteq_m m;\ dom\ m' \subseteq S]\!] \Longrightarrow m' \subseteq_m (m \mid` S)$
by (*force simp*: *restrict-map-def map-le-def*)

lemma *map-le-dom-restrict-sub-add*:
$m' \subseteq_m m \Longrightarrow m \mid` (dom\ m - dom\ m') ++ m' = m$
by (*metis map-add-restrict-comp-right-dom map-le-iff-map-add-commute map-le-restrict*)

lemma *subset-map-restrict-sub-add*:
$T \subseteq S \Longrightarrow m \mid` (S - T) ++ m \mid` T = m \mid` S$
by (*fastforce simp*: *restrict-map-def map-add-def split*: *option.splits*)

lemma *restrict-map-sub-union*:
$m \mid` (dom\ m - (S \cup T)) = (m \mid` (dom\ m - T)) \mid` (dom\ m - S)$

by (*auto simp*: *restrict-map-def*)

lemma *prod-restrict-map-add*:
⟦ $S \cup T = U$; $S \cap T = \{\}$ ⟧ ⟹ $m \mid` (X \times S) ++ m \mid` (X \times T) = m \mid` (X \times U)$
by (*auto simp*: *map-add-def restrict-map-def split*: *option.splits*)

Things that should NOT go into Map.thy

11.21 Definitions

Map disjuction

definition
map-disj :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow$ *bool* (**infix** ‹⊥› *51*) **where**
$h_0 \perp h_1 \equiv dom\ h_0 \cap dom\ h_1 = \{\}$

declare *None-not-eq* [*simp*]

Heap monotonicity and the frame property

definition
heap-mono :: $(('a \rightharpoonup 'b) \Rightarrow 'c\ option) \Rightarrow bool$ **where**
heap-mono $f \equiv \forall h\ h'\ v.\ h \perp h' \wedge f\ h = Some\ v \longrightarrow f\ (h ++ h') = Some\ v$

lemma *heap-monoE*:
⟦ *heap-mono* f ; $f\ h = Some\ v$; $h \perp h'$ ⟧ ⟹ $f\ (h ++ h') = Some\ v$
unfolding *heap-mono-def* **by** *blast*

lemma *heap-mono-simp*:
⟦ *heap-mono* f ; $f\ h = Some\ v$; $h \perp h'$ ⟧ ⟹ $f\ (h ++ h') = f\ h$
by (*frule* (*2*) *heap-monoE*, *simp*)

definition
heap-frame :: $(('a \rightharpoonup 'b) \Rightarrow 'c\ option) \Rightarrow bool$ **where**
heap-frame $f \equiv \forall h\ h'\ v.\ h \perp h' \wedge f\ (h ++ h') = Some\ v$
$\longrightarrow (f\ h = Some\ v \vee f\ h = None)$

lemma *heap-frameE*:
⟦ *heap-frame* f ; $f\ (h ++ h') = Some\ v$; $h \perp h'$ ⟧
⟹ $f\ h = Some\ v \vee f\ h = None$
unfolding *heap-frame-def* **by** *fastforce*

11.22 Properties of ($`-$)

lemma *restrict-map-sub-disj*: $h \mid` S \perp h\ `- S$

by (*fastforce simp*: *sub-restrict-map-def restrict-map-def map-disj-def*
split: *option.splits if-split-asm*)

lemma *restrict-map-sub-add*: $h \mid` S ++ h -` S = h$
by (*fastforce simp*: *sub-restrict-map-def restrict-map-def map-add-def*
split: *option.splits if-split*)

11.23 Properties of map disjunction

lemma *map-disj-empty-right* [*simp*]:
$h \perp Map.empty$
by (*simp add*: *map-disj-def*)

lemma *map-disj-empty-left* [*simp*]:
$Map.empty \perp h$
by (*simp add*: *map-disj-def*)

lemma *map-disj-com*:
$h_0 \perp h_1 = h_1 \perp h_0$
by (*simp add*: *map-disj-def*, *fast*)

lemma *map-disjD*:
$h_0 \perp h_1 \Longrightarrow dom\ h_0 \cap dom\ h_1 = \{\}$
by (*simp add*: *map-disj-def*)

lemma *map-disjI*:
$dom\ h_0 \cap dom\ h_1 = \{\} \Longrightarrow h_0 \perp h_1$
by (*simp add*: *map-disj-def*)

11.23.1 Map associativity-commutativity based on map disjuction

lemma *map-add-com*:
$h_0 \perp h_1 \Longrightarrow h_0 ++ h_1 = h_1 ++ h_0$
by (*drule map-disjD*, *rule map-add-comm*, *force*)

lemma *map-add-left-commute*:
$h_0 \perp h_1 \Longrightarrow h_0 ++ (h_1 ++ h_2) = h_1 ++ (h_0 ++ h_2)$
by (*simp add*: *map-add-com map-disj-com*)

lemma *map-add-disj*:
$h_0 \perp (h_1 ++ h_2) = (h_0 \perp h_1 \wedge h_0 \perp h_2)$
by (*simp add*: *map-disj-def*, *fast*)

lemma *map-add-disj'*:
$(h_1 ++ h_2) \perp h_0 = (h_1 \perp h_0 \wedge h_2 \perp h_0)$
by (*simp add*: *map-disj-def*, *fast*)

We redefine (++) associativity to bind to the right, which seems to

be the more common case. Note that when a theory includes Map again, *map-add-assoc* will return to the simpset and will cause infinite loops if its symmetric counterpart is added (e.g. via *map-ac-simps*)

declare *map-add-assoc* [*simp del*]

Since the associativity-commutativity of (++) relies on map disjunction, we include some basic rules into the ac set.

lemmas *map-ac-simps* =
map-add-assoc[*symmetric*] *map-add-com map-disj-com*
map-add-left-commute map-add-disj map-add-disj'

11.23.2 Basic properties

lemma *map-disj-None-right*:
⟦ $h_0 \perp h_1$; $x \in dom\ h_0$ ⟧ ⟹ $h_1\ x = None$
by (*auto simp*: *map-disj-def dom-def*)

lemma *map-disj-None-left*:
⟦ $h_0 \perp h_1$; $x \in dom\ h_1$ ⟧ ⟹ $h_0\ x = None$
by (*auto simp*: *map-disj-def dom-def*)

lemma *map-disj-None-left'*:
⟦ $h_0\ x = Some\ y$; $h_1 \perp h_0$ ⟧ ⟹ $h_1\ x = None$
by (*auto simp*: *map-disj-def*)

lemma *map-disj-None-right'*:
⟦ $h_1\ x = Some\ y$; $h_1 \perp h_0$ ⟧ ⟹ $h_0\ x = None$
by (*auto simp*: *map-disj-def*)

lemma *map-disj-common*:
⟦ $h_0 \perp h_1$; $h_0\ p = Some\ v$; $h_1\ p = Some\ v'$ ⟧ ⟹ *False*
by (*frule* (*1*) *map-disj-None-left'*, *simp*)

11.23.3 Map disjunction and addition

lemma *map-add-eval-left*:
⟦ $x \in dom\ h$; $h \perp h'$ ⟧ ⟹ $(h\ ++\ h')\ x = h\ x$
by (*auto dest*!: *map-disj-None-right simp*: *map-add-def cong*: *option.case-cong*)

lemma *map-add-eval-right*:
⟦ $x \in dom\ h'$; $h \perp h'$ ⟧ ⟹ $(h\ ++\ h')\ x = h'\ x$
by (*auto elim*!: *map-disjD simp*: *map-add-comm map-add-eval-left map-disj-com*)

lemma *map-add-eval-left'*:
⟦ $x \notin dom\ h'$; $h \perp h'$ ⟧ ⟹ $(h\ ++\ h')\ x = h\ x$
by (*clarsimp simp*: *map-disj-def map-add-def split*: *option.splits*)

lemma *map-add-eval-right'*:
⟦ $x \notin dom\ h$; $h \perp h'$ ⟧ ⟹ $(h\ ++\ h')\ x = h'\ x$

by (*clarsimp simp*: *map-disj-def map-add-def split*: *option.splits*)

lemma *map-add-left-dom-eq*:
assumes *eq*: $h_0 ++ h_1 = h_0' ++ h_1'$
assumes *etc*: $h_0 \perp h_1$ $h_0' \perp h_1'$ *dom* $h_0 =$ *dom* h_0'
shows $h_0 = h_0'$
proof −
from *eq* **have** $h_1 ++ h_0 = h_1' ++ h_0'$ **using** *etc* **by** (*simp add*: *map-ac-simps*)
thus *?thesis* **using** *etc*
by (*fastforce elim*!: *map-add-right-dom-eq simp*: *map-ac-simps*)
qed

lemma *map-add-left-eq*:
assumes *eq*: $h_0 ++ h = h_1 ++ h$
assumes *disj*: $h_0 \perp h$ $h_1 \perp h$
shows $h_0 = h_1$
proof (*rule ext*)
fix x
from *eq* **have** *eq'*: $(h_0 ++ h)\ x = (h_1 ++ h)\ x$ **by** *auto*
{ assume $x \in$ *dom* h
hence $h_0\ x = h_1\ x$ **using** *disj* **by** (*simp add*: *map-disj-None-left*)
} moreover {
assume $x \notin$ *dom* h
hence $h_0\ x = h_1\ x$ **using** *disj eq'* **by** (*simp add*: *map-add-eval-left'*)
}
ultimately show $h_0\ x = h_1\ x$ **by** *cases*
qed

lemma *map-add-right-eq*:
$[\![h ++ h_0 = h ++ h_1;\ h_0 \perp h;\ h_1 \perp h]\!] \Longrightarrow h_0 = h_1$
by (*rule map-add-left-eq* [**where** *h*=*h*], *auto simp*: *map-ac-simps*)

lemma *map-disj-add-eq-dom-right-eq*:
assumes *merge*: $h_0 ++ h_1 = h_0' ++ h_1'$ **and** *d*: *dom* $h_0 =$ *dom* h_0' **and**
ab-disj: $h_0 \perp h_1$ **and** *cd-disj*: $h_0' \perp h_1'$
shows $h_1 = h_1'$
proof (*rule ext*)
fix x
from *merge* **have** *merge-x*: $(h_0 ++ h_1)\ x = (h_0' ++ h_1')\ x$ **by** *simp*
with *d ab-disj cd-disj* **show** $h_1\ x = h_1'\ x$
apply (*cases* $h_1\ x$)
apply (*cases* $h_1'\ x$)
apply *force*
apply (*fastforce simp*: *map-disj-def*)
apply (*cases* $h_1'\ x$)
apply *clarsimp*
apply (*simp add*: *Some-com*)
by (*force simp*: *map-disj-def*)+

qed

lemma *map-disj-add-eq-dom-left-eq*:
assumes *add*: $h_0 ++ h_1 = h_0' ++ h_1'$ **and**
dom: $dom\ h_1 = dom\ h_1'$ **and**
disj: $h_0 \perp h_1\ h_0' \perp h_1'$
shows $h_0 = h_0'$
proof −
have $h_1 ++ h_0 = h_1' ++ h_0'$ **using** *add disj* **by** (*simp add*: *map-ac-simps*)
thus *?thesis* **using** *dom disj*
by − (*rule map-disj-add-eq-dom-right-eq*, *auto simp*: *map-disj-com*)
qed

lemma *map-add-left-cancel*:
assumes *disj*: $h_0 \perp h_1\ h_0 \perp h_1'$
shows $(h_0 ++ h_1 = h_0 ++ h_1') = (h_1 = h_1')$
proof (*rule iffI*, *rule ext*)
fix x
assume $(h_0 ++ h_1) = (h_0 ++ h_1')$
hence $(h_0 ++ h_1)\ x = (h_0 ++ h_1')\ x$ **by** *auto*
hence $h_1\ x = h_1'\ x$ **using** *disj*
by − (*cases* $x \in dom\ h_0$,
simp-all add: *map-disj-None-right map-add-eval-right'*)
thus $h_1\ x = h_1'\ x$ **by** *auto*
qed *auto*

lemma *map-add-lr-disj*:
$[\![\ h_0 ++ h_1 = h_0' ++ h_1';\ h_1 \perp h_1'\ ]\!] \Longrightarrow dom\ h_1 \subseteq dom\ h_0'$
apply (*clarsimp simp*: *map-disj-def map-add-def*)
subgoal for $x\ y$
apply (*drule fun-cong* [**where** $x{=}x$])
apply (*auto split*: *option.splits*)
done
done

11.23.4 Map disjunction and updates

lemma *map-disj-update-left* [*simp*]:
$p \in dom\ h_1 \Longrightarrow h_0 \perp h_1(p \mapsto v) = h_0 \perp h_1$
by (*clarsimp simp add*: *map-disj-def*, *blast*)

lemma *map-disj-update-right* [*simp*]:
$p \in dom\ h_1 \Longrightarrow h_1(p \mapsto v) \perp h_0 = h_1 \perp h_0$
by (*simp add*: *map-disj-com*)

lemma *map-add-update-left*:
$[\![\ h_0 \perp h_1\ ;\ p \in dom\ h_0\ ]\!] \Longrightarrow (h_0 ++ h_1)(p \mapsto v) = (h_0(p \mapsto v) ++ h_1)$
by (*drule* (*1*) *map-disj-None-right*)
(*auto simp*: *map-add-def cong*: *option.case-cong*)

lemma *map-add-update-right*:
⟦ $h_0 \perp h_1$; $p \in dom\ h_1$ ⟧ ⟹ $(h_0 ++ h_1)(p \mapsto v) = (h_0 ++ h_1\ (p \mapsto v))$
by (*drule* (*1*) *map-disj-None-left*)
(*auto simp*: *map-add-def cong*: *option.case-cong*)

lemma *map-add3-update*:
⟦ $h_0 \perp h_1$; $h_1 \perp h_2$; $h_0 \perp h_2$; $p \in dom\ h_0$ ⟧
⟹ $(h_0 ++ h_1 ++ h_2)(p \mapsto v) = h_0(p \mapsto v) ++ h_1 ++ h_2$
by (*auto simp*: *map-add-update-left*[*symmetric*] *map-ac-simps*)

11.23.5 Map disjunction and ($\subseteq_m$)

lemma *map-le-override* [*simp*]:
⟦ $h \perp h'$ ⟧ ⟹ $h \subseteq_m h ++ h'$
by (*auto simp*: *map-le-def map-add-def map-disj-def split*: *option.splits*)

lemma *map-leI-left*:
⟦ $h = h_0 ++ h_1$; $h_0 \perp h_1$ ⟧ ⟹ $h_0 \subseteq_m h$ **by** *auto*

lemma *map-leI-right*:
⟦ $h = h_0 ++ h_1$; $h_0 \perp h_1$ ⟧ ⟹ $h_1 \subseteq_m h$ **by** *auto*

lemma *map-disj-map-le*:
⟦ $h_0' \subseteq_m h_0$; $h_0 \perp h_1$ ⟧ ⟹ $h_0' \perp h_1$
by (*force simp*: *map-disj-def map-le-def*)

lemma *map-le-on-disj-left*:
⟦ $h' \subseteq_m h$; $h_0 \perp h_1$; $h' = h_0 ++ h_1$ ⟧ ⟹ $h_0 \subseteq_m h$
unfolding *map-le-def*
apply (*rule ballI*)
subgoal for *a*
apply (*erule ballE* [**where** *x*=*a*], *auto simp*: *map-add-eval-left*)+
done
done

lemma *map-le-on-disj-right*:
⟦ $h' \subseteq_m h$; $h_0 \perp h_1$; $h' = h_1 ++ h_0$ ⟧ ⟹ $h_0 \subseteq_m h$
by (*auto simp*: *map-le-on-disj-left map-ac-simps*)

lemma *map-le-add-cancel*:
⟦ $h_0 \perp h_1$; $h_0' \subseteq_m h_0$ ⟧ ⟹ $h_0' ++ h_1 \subseteq_m h_0 ++ h_1$
by (*auto simp*: *map-le-def map-add-def map-disj-def split*: *option.splits*)

lemma *map-le-override-bothD*:
assumes *subm*: $h_0' ++ h_1 \subseteq_m h_0 ++ h_1$
assumes *disj'*: $h_0' \perp h_1$
assumes *disj*: $h_0 \perp h_1$
shows $h_0' \subseteq_m h_0$

unfolding *map-le-def*
proof (*rule ballI*)
fix *a*
assume *a*: $a \in dom\ h_0{'}$
hence *sumeq*: $(h_0{'} +\!\!+ h_1)\ a = (h_0 +\!\!+ h_1)\ a$
using *subm* **unfolding** *map-le-def* **by** *auto*
from *a* **have** $a \notin dom\ h_1$ **using** *disj'* **by** (*auto dest*!: *map-disj-None-right*)
thus $h_0{'}\ a = h_0\ a$ **using** *a sumeq disj disj'*
by (*simp add*: *map-add-eval-left map-add-eval-left'*)
qed

lemma *map-le-conv*:
$(h_0{'} \subseteq_m h_0 \wedge h_0{'} \neq h_0) = (\exists h_1.\ h_0 = h_0{'} +\!\!+ h_1 \wedge h_0{'} \perp h_1 \wedge h_0{'} \neq h_0)$
unfolding *map-le-def map-disj-def map-add-def*
using *exI*[**where** $x{=}\lambda x.$ *if* $x \notin dom\ h_0{'}$ *then* $h_0\ x$ *else None*]
by (*rule iffI*, *clarsimp*)
(*fastforce intro*: *set-eqI split*: *option.splits if-split-asm*)+

lemma *map-le-conv2*:
$h_0{'} \subseteq_m h_0 = (\exists h_1.\ h_0 = h_0{'} +\!\!+ h_1 \wedge h_0{'} \perp h_1)$
by (*cases* $h_0{'}{=}h_0$, *insert map-le-conv*, *auto intro*: *exI*[**where** $x{=}$*Map.empty*])

11.23.6 Map disjunction and restriction

lemma *map-disj-comp* [*simp*]:
$h_0 \perp h_1\ |`\ (UNIV - dom\ h_0)$
by (*force simp*: *map-disj-def*)

lemma *restrict-map-disj*:
$S \cap T = \{\} \Longrightarrow h\ |`\ S \perp h\ |`\ T$
by (*auto simp*: *map-disj-def restrict-map-def dom-def*)

lemma *map-disj-restrict-dom* [*simp*]:
$h_0 \perp h_1\ |`\ (dom\ h_1 - dom\ h_0)$
by (*force simp*: *map-disj-def*)

lemma *restrict-map-disj-dom-empty*:
$h \perp h' \Longrightarrow h\ |`\ dom\ h' = Map.empty$
by (*fastforce simp*: *map-disj-def restrict-map-def*)

lemma *restrict-map-univ-disj-eq*:
$h \perp h' \Longrightarrow h\ |`\ (UNIV - dom\ h') = h$
by (*rule ext*, *auto simp*: *map-disj-def restrict-map-def*)

lemma *restrict-map-disj-dom*:
$h_0 \perp h_1 \Longrightarrow h\ |`\ dom\ h_0 \perp h\ |`\ dom\ h_1$
by (*auto simp*: *map-disj-def restrict-map-def dom-def*)

lemma *map-add-restrict-dom-left*:

$h \perp h' \Longrightarrow (h +\!\!+ h') \mid` dom\ h = h$
by (*rule ext*, *auto simp*: *restrict-map-def map-add-def dom-def map-disj-def*
split: *option.splits*)

lemma *restrict-map-disj-left*:
$h_0 \perp h_1 \Longrightarrow h_0 \mid` S \perp h_1$
by (*auto simp*: *map-disj-def*)

lemma *restrict-map-disj-right*:
$h_0 \perp h_1 \Longrightarrow h_0 \perp h_1 \mid` S$
by (*auto simp*: *map-disj-def*)

lemmas *restrict-map-disj-both* = *restrict-map-disj-right restrict-map-disj-left*

lemma *map-dom-disj-restrict-right*:
$h_0 \perp h_1 \Longrightarrow (h_0 +\!\!+ h_0') \mid` dom\ h_1 = h_0' \mid` dom\ h_1$
by (*simp add*: *map-add-restrict restrict-map-empty map-disj-def*)

lemma *restrict-map-on-disj*:
$h_0' \perp h_1 \Longrightarrow h_0 \mid` dom\ h_0' \perp h_1$
unfolding *map-disj-def* **by** *auto*

lemma *restrict-map-on-disj'*:
$h_0 \perp h_1 \Longrightarrow h_0 \perp h_1 \mid` S$
by (*auto simp*: *map-disj-def map-add-def*)

lemma *map-le-sub-dom*:
$[\![\ h_0 +\!\!+ h_1 \subseteq_m h\ ;\ h_0 \perp h_1\ ]\!] \Longrightarrow h_0 \subseteq_m h \mid` (dom\ h - dom\ h_1)$
by (*rule map-le-override-bothD*, *subst map-le-dom-restrict-sub-add*)
(*auto elim*: *map-add-le-mapE simp*: *map-ac-simps*)

lemma *map-submap-break*:
$[\![\ h \subseteq_m h'\ ]\!] \Longrightarrow h' = (h' \mid` (UNIV - dom\ h)) +\!\!+ h$
by (*fastforce split*: *option.splits*
simp: *map-le-restrict restrict-map-def map-le-def map-add-def dom-def*)

lemma *map-add-disj-restrict-both*:
$[\![\ h_0 \perp h_1;\ S \cap S' = \{\};\ T \cap T' = \{\}\ ]\!]$
$\Longrightarrow (h_0 \mid` S) +\!\!+ (h_1 \mid` T) \perp (h_0 \mid` S') +\!\!+ (h_1 \mid` T')$
by (*auto simp*: *map-ac-simps intro*!: *restrict-map-disj-both restrict-map-disj*)

end

theory *MapExtraTrans*
imports *MapExtra*
begin

lemma *case-option-None-Some* [*simp*]:
  case-option None Some P = P
  by (*simp split*: *option.splits*)

lemma *heap-merge-dom-exact2*:
  ⟦ *a ++ b = c ++ d*; *dom a = dom c*; *a ⊥ b*; *c ⊥ d* ⟧ ⟹ *a=c ∧ b=d*
  apply (*rule conjI*)
   apply (*erule* (*3*) *map-add-left-dom-eq*)
  apply (*erule* (*3*) *map-disj-add-eq-dom-right-eq*)
  done

lemma *map-add-restrict-sub*:
  ⟦ *dom s = X*; *dom t = X − Y* ⟧ ⟹
      s ++ (t |‘ (X − Y − Z)) = s ++ t ++ s |‘ Z
apply(*rule ext*)
apply(*auto simp*: *restrict-map-def map-add-def split*: *option.splits*)
done

lemma *map-add-restrict-UNIV*:
  ⟦ *dom g ∩ X = {}*; *dom f = dom h* ⟧ ⟹ *f ++ g = f |‘ (UNIV − X) ++ h |‘ X ++ g ++ f |‘ X*
apply(*rule ext*)
apply(*force simp*: *restrict-map-def map-add-def split*: *option.splits*)
done

end

theory *TypHeap*
imports
  Vanilla32
  ArchArraysMemInstance
  HeapRawState
  MapExtraTrans
begin

declare *map-add-assoc* [*simp del*]

definition *wf-heap-val* :: *heap-state* ⇒ *bool* **where**

wf-heap-val s ≡
∀ *x t n v*. *s* (*x*,*SIndexVal*) ≠ *Some* (*STyp t*) ∧ *s* (*x*,*SIndexTyp n*) ≠ *Some* (*SValue v*)

type-synonym *typ-slice-list* = (*typ-uinfo* × *typ-base*) *list*

primrec
typ-slice-t :: *typ-uinfo* ⇒ *nat* ⇒ *typ-slice-list* **and**
typ-slice-struct :: *typ-uinfo-struct* ⇒ *nat* ⇒ *typ-slice-list* **and**
typ-slice-list :: (*typ-uinfo*,*field-name*, *unit*) *dt-tuple list* ⇒ *nat* ⇒ *typ-slice-list* **and**
typ-slice-tuple :: (*typ-uinfo*,*field-name*, *unit*) *dt-tuple* ⇒ *nat* ⇒ *typ-slice-list*
where
tl0: *typ-slice-t* (*TypDesc algn st nm*) *m* = *typ-slice-struct st m* @
[(*if m* = *0 then* ((*TypDesc algn st nm*),*True*) *else*
((*TypDesc algn st nm*),*False*))]

| *tl1*: *typ-slice-struct* (*TypScalar n algn d*) *m* = []
| *tl2*: *typ-slice-struct* (*TypAggregate xs*) *m* = *typ-slice-list xs m*

| *tl3*: *typ-slice-list* [] *m* = []
| *tl4*: *typ-slice-list* (*x*#*xs*) *m* = (*if m* < *size-td* (*dt-fst x*) ∨ *xs* = [] *then*
typ-slice-tuple x m else typ-slice-list xs (*m* − *size-td* (*dt-fst x*)))

| *tl5*: *typ-slice-tuple* (*DTuple t n d*) *m* = *typ-slice-t t m*

definition *list-map* :: *'a list* ⇒ (*nat* ⇀ *'a*) **where**
list-map xs ≡ *map-of* (*zip* [*0*..<*length xs*] *xs*)

definition *s-footprint-untyped* :: *addr* ⇒ *typ-uinfo* ⇒ (*addr* × *s-heap-index*) *set*
where
s-footprint-untyped p t ≡
{(*p* + *of-nat x*,*k*) | *x k*. *x* < *size-td t* ∧
(*k*=*SIndexVal* ∨ (∃ *n*. *k*=*SIndexTyp n* ∧ *n* < *length* (*typ-slice-t t x*)))}

definition (**in** *c-type*) *s-footprint* :: *'a ptr* ⇒ (*addr* × *s-heap-index*) *set* **where**
s-footprint p ≡ *s-footprint-untyped* (*ptr-val p*) (*typ-uinfo-t TYPE*(*'a*))

definition *empty-htd* :: *heap-typ-desc* **where**
empty-htd ≡ λ*x*. (*False*,*Map.empty*)

definition *dom-s* :: *heap-typ-desc* ⇒ *s-addr set* **where**
dom-s d ≡ {(*x*,*SIndexVal*) | *x*. *fst* (*d x*)} ∪
{(*x*,*SIndexTyp n*) | *x n*. *snd* (*d x*) *n* ≠ *None*}

definition *restrict-s* :: *heap-typ-desc* ⇒ *s-addr set* ⇒ *heap-typ-desc* **where**
restrict-s d X ≡
λ*x*. ((*x*,*SIndexVal*) ∈ *X* ∧ *fst* (*d x*), (λ*y*. *if* (*x*,*SIndexTyp y*) ∈ *X then snd* (*d x*) *y else None*))

definition *valid-footprint* :: *heap-typ-desc* ⇒ *addr* ⇒ *typ-uinfo* ⇒ *bool* **where**
valid-footprint d x t ≡
let n = *size-td t in*
0 < *n* ∧ (∀ *y*. *y* < *n* ⟶
list-map (*typ-slice-t t y*) ⊆$_m$ *snd* (*d* (*x* + *of-nat y*)) ∧ *fst* (*d* (*x* + *of-nat y*)))

definition (**in** *c-type*) *h-t-valid* ::
heap-typ-desc ⇒ *'a ptr-guard* ⇒ *'a ptr* ⇒ *bool*
(‹(‹*open-block notation*=‹*mixfix heap-typ-desc*››-,- ⊨$_t$ -)› [*99*,*0*,*99*] *100*)
where *d*,*g* ⊨$_t$ *p* ≡ *valid-footprint d* (*ptr-val* (*p*::*'a ptr*)) (*typ-uinfo-t TYPE*(*'a*)) ∧ *g p*

type-synonym *'a typ-heap* = *'a ptr* ⇀ *'a*

definition *proj-h* :: *heap-state* ⇒ *heap-mem* **where**
proj-h s ≡ λ*x*. *case-option undefined* (*case-s-heap-value id undefined*) (*s* (*x*,*SIndexVal*))

definition *lift-state* :: *heap-raw-state* ⇒ *heap-state* **where**
lift-state ≡ λ(*h*,*d*) (*x*,*y*).
case y of
SIndexVal ⇒ *if fst* (*d x*)*then Some* (*SValue* (*h x*)) *else None*
| *SIndexTyp n* ⇒ *case-option None* (*Some* ∘ *STyp*) (*snd* (*d x*) *n*)

definition *fun2list* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a list* **where**
fun2list f n ≡ *if n=0 then* [] *else map f* [*0*..<*n*]

definition *null-d* :: *heap-state* ⇒ *addr* ⇒ *nat* ⇒ *bool* **where**
null-d s x y ≡ *s* (*x*,*SIndexTyp y*) = *None*

definition *max-d* :: *heap-state* ⇒ *addr* ⇒ *nat* **where**
max-d s x ≡ *1* + (*GREATEST y*. ¬ *null-d s x y*)

definition *proj-d* :: *heap-state* ⇒ *heap-typ-desc* **where**
proj-d s ≡ λ*x*. (*s* (*x*,*SIndexVal*) ≠ *None*,
λ*n*. *case-option None* (*Some* ∘ *s-heap-tag*) (*s* (*x*,*SIndexTyp n*)))

definition (**in** *c-type*) *s-valid* ::
heap-state ⇒ *'a ptr-guard* ⇒ *'a ptr* ⇒ *bool*
(‹(‹*open-block notation*=‹*mixfix heap-state*››-,- ⊨$_s$ -)› [*100*,*0*,*100*] *100*)
where *s*,*g* ⊨$_s$ *p* ≡ *proj-d s*,*g* ⊨$_t$ *p*

definition *heap-list-s* :: *heap-state* ⇒ *nat* ⇒ *addr* ⇒ *byte list* **where**
heap-list-s s n p ≡ *heap-list* (*proj-h s*) *n p*

definition (**in** *c-type*) *lift-typ-heap* :: *'a ptr-guard* ⇒ *heap-state* ⇒ *'a typ-heap* **where**
lift-typ-heap g s ≡
(*Some* ∘ *from-bytes* ∘ *heap-list-s s* (*size-of TYPE*(*'a*)) ∘ *ptr-val*) |` {*p*. *s*,*g* ⊨$_s$ *p*}

definition (**in** *c-type*) *heap-update-s* :: *'a ptr* ⇒ *'a* ⇒ *heap-state* ⇒ *heap-state* **where**
heap-update-s n p s ≡ *lift-state* (*heap-update n p* (*proj-h s*), *proj-d s*)

definition (**in** *c-type*) *lift-t* :: *'a ptr-guard* ⇒ *heap-raw-state* ⇒ *'a typ-heap* **where**
lift-t g ≡ *lift-typ-heap g* ∘ *lift-state*

definition *tag-disj* :: (*'a*, *'b*) *typ-desc* ⇒ (*'a*,*'b*) *typ-desc* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix tag-disj*››- ⊥$_t$ -)› [*90*,*90*] *90*)
where *f* ⊥$_t$ *g* ≡ ¬ (*f* ≤ *g* ∨ *g* ≤ *f*)

definition *ladder-set* :: *typ-uinfo* ⇒ *nat* ⇒ *nat* ⇒ (*typ-uinfo* × *nat*) *set* **where**
ladder-set s n p ≡ {(*t*,*n*+*p*) | *t*. ∃ *k*. (*t*,*k*) ∈ *set* (*typ-slice-t s n*)}

primrec
field-names :: (*'a*,*'b*) *typ-info* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-struct :: (*'a field-desc*,*'b*) *typ-struct* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-list :: ((*'a*,*'b*) *typ-info*,*field-name*, *'b*) *dt-tuple list* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-tuple :: ((*'a*,*'b*) *typ-info*,*field-name*, *'b*) *dt-tuple* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list*
where
tfs0: *field-names* (*TypDesc algn st nm*) *t* = (*if t*=*export-uinfo* (*TypDesc algn st nm*) *then*
[[]] *else field-names-struct st t*)

| *tfs1*: *field-names-struct* (*TypScalar m algn d*) *t* = []
| *tfs2*: *field-names-struct* (*TypAggregate xs*) *t* = *field-names-list xs t*

| *tfs3*: *field-names-list* [] *t* = []
| *tfs4*: *field-names-list* (*x*#*xs*) *t* = *field-names-tuple x t*@*field-names-list xs t*

| *tfs5*: *field-names-tuple* (*DTuple s f d*) *t* = *map* (λ*fs*. *f*#*fs*) (*field-names s t*)

definition *field-typ-untyped* :: (*'a*,*'b*) *typ-desc* ⇒ *qualified-field-name* ⇒ (*'a*,*'b*) *typ-desc* **where**
field-typ-untyped t n ≡ (*fst* (*the* (*field-lookup t n 0*)))

definition (**in** *c-type*) *field-typ* :: *'a itself* ⇒ *qualified-field-name* ⇒ *'a xtyp-info*
where
field-typ t n ≡ *field-typ-untyped* (*typ-info-t TYPE*('*a*)) *n*

definition (**in** *c-type*) *fs-consistent* ::
qualified-field-name list ⇒ *'a itself* ⇒ *'b*::*c-type itself* ⇒ *bool* **where**
fs-consistent fs a b ≡ *set fs* ⊆ *set* (*field-names* (*typ-info-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))

definition (**in** *c-type*) *field-offset-footprint* :: *'a ptr* ⇒ (*qualified-field-name*) *list* ⇒ *'b ptr set*
where
field-offset-footprint p fs ≡ {*Ptr* &(*p*→*k*) | *k*. *k* ∈ *set fs*}

definition *sub-typ* :: *'a*::*c-type itself* ⇒ *'b*::*c-type itself* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix sub-typ*››- ≤$_\tau$ -)› [*51*, *51*] *50*)
where *s* ≤$_\tau$ *t* ≡ *typ-uinfo-t s* ≤ *typ-uinfo-t t*

definition *sub-typ-proper* :: *'a*::*c-type itself* ⇒ *'b*::*c-type itself* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix sub-typ-proper*››- <$_\tau$ -)› [*51*, *51*] *50*)
where *s* <$_\tau$ *t* ≡ *typ-uinfo-t s* < *typ-uinfo-t t*

definition *peer-typ* :: *'a*::*c-type itself* ⇒ *'b* :: *c-type itself* ⇒ *bool*
where
peer-typ a b ≡ *typ-uinfo-t TYPE*('*a*) = *typ-uinfo-t TYPE*('*b*) ∨
typ-uinfo-t TYPE('*a*) ⊥$_t$ *typ-uinfo-t TYPE*('*b*)

definition (**in** *c-type*) *guard-mono* :: *'a ptr-guard* ⇒ *'b*::*c-type ptr-guard* ⇒ *bool*
where
guard-mono g g' ≡
∀ *n f p*. *g p* ∧
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*),*n*)
⟶
g' (*Ptr* (*ptr-val p* + *of-nat n*))

primrec (**in** *c-type*) *sub-field-update-t* ::
qualified-field-name list ⇒ *'a ptr* ⇒ *'a* ⇒ *'b*::*c-type typ-heap* ⇒ *'b typ-heap*
where
sft0: *sub-field-update-t* [] *p v s* = *s*
| *sft1*: *sub-field-update-t* (*f*#*fs*) *p* (*v*::*'a*) *s* =
(*let s'* = *sub-field-update-t fs p v s in*
s'(*Ptr* &(*p*→*f*) ↦ *from-bytes* (*access-ti*$_0$ (*field-typ TYPE*('*a*) *f*) *v*))) |‘
dom (*s*::*'b*::*c-type typ-heap*)

primrec (**in** *c-type*) *update-value-t* :: (*qualified-field-name*) *list* ⇒ *'a* ⇒ *'b* ⇒ *nat*

$\Rightarrow$ *'b::c-type*
where
uvt0: *update-value-t* [] *v w x* = *w*
| *uvt1*: *update-value-t* (*f#fs*) *v* (*w::'b*) *x* = (*if x=field-offset TYPE('b) f then field-update* (*field-desc* (*field-typ TYPE('b) f*)) (*to-bytes-p* (*v::'a*)) (*w::'b::c-type*) *else update-value-t fs v w x*)

definition (**in** *c-type*) *super-field-update-t* :: *'a ptr* $\Rightarrow$ *'a* $\Rightarrow$ *'b::c-type typ-heap* $\Rightarrow$ *'b typ-heap* **where**
super-field-update-t p v s $\equiv$ λq.
if field-of-t p q
then
case-option None
(λw. *Some* (*update-value-t* (*field-names* (*typ-info-t TYPE('b)*) (*typ-uinfo-t TYPE('a)*)))
v w (*unat* (*ptr-val p* − *ptr-val q*))))
(*s q*)
else s q

definition *heap-footprint* :: *heap-typ-desc* $\Rightarrow$ *typ-uinfo* $\Rightarrow$ *addr set* **where**
heap-footprint d t $\equiv$ {*x*. $\exists$ *y*. *valid-footprint d y t* $\wedge$ *x* $\in$ {*y*} $\cup$ {*y*..+*size-td t*}}

definition (**in** *c-type*) *ptr-safe* :: *'a ptr* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *bool* **where**
ptr-safe p d $\equiv$ *s-footprint p* $\subseteq$ *dom-s d*

primrec
htd-update-list :: *addr* $\Rightarrow$ *typ-slice list* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *heap-typ-desc*
where
hul0: *htd-update-list p* [] *d* = *d*
| *hul1*: *htd-update-list p* (*x#xs*) *d* = *htd-update-list* (*p+1*) *xs* (*d*(*p* := (*True*,*snd* (*d p*) ++ *x*)))

definition *dom-tll* :: *addr* $\Rightarrow$ *typ-slice list* $\Rightarrow$ *s-addr set* **where**
dom-tll p xs $\equiv$ {(*p* + *of-nat x*,*SIndexVal*) | *x*. *x* < *length xs*} $\cup$
{(*p* + *of-nat x*,*SIndexTyp n*) | *x n*. *x* < *length xs* $\wedge$ (*xs* ! *x*) *n* $\neq$ *None*}

definition (**in** *c-type*) *typ-slices* :: *'a itself* $\Rightarrow$ *typ-slice list* **where**
typ-slices t $\equiv$ *map* (λn. *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE('a)*) *n*)) [*0*..<*size-of TYPE('a)*]

definition (**in** *c-type*) *ptr-retyp* :: *'a ptr* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *heap-typ-desc* **where**
ptr-retyp p $\equiv$ *htd-update-list* (*ptr-val p*) (*typ-slices TYPE('a)*)

definition (**in** *c-type*) *field-fd* :: *'a itself* $\Rightarrow$ *qualified-field-name* $\Rightarrow$ *'a field-desc* **where**
field-fd t n $\equiv$ *field-desc* (*field-typ t n*)

definition *tag-disj-typ* :: *'a*::*c-type itself* ⇒ *'b*::*c-type itself* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix tag-disj-typ*››- ⊥$_\tau$ -)›)
where *s* ⊥$_\tau$ *t* ≡ *typ-uinfo-t s* ⊥$_t$ *typ-uinfo-t t*

—-

lemma *wf-heap-val-SIndexVal-STyp-simp* [*simp*]:
wf-heap-val s ⟹ *s* (*x*,*SIndexVal*) ≠ *Some* (*STyp t*)
apply(*clarsimp simp*: *wf-heap-val-def*)
apply(*drule spec* [**where** *x*=*x*])
apply *clarsimp*
apply(*cases t*, *simp*)
apply *fast*
done

lemma *wf-heap-val-SIndexTyp-SValue-simp* [*simp*]:
wf-heap-val s ⟹ *s* (*x*,*SIndexTyp n*) ≠ *Some* (*SValue v*)
apply(*unfold wf-heap-val-def*)
apply *clarify*
apply(*drule-tac x*=*x* **in** *spec*)
apply *clarsimp*
done

lemma (**in** *mem-type*) *field-tag-sub*:
field-lookup (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*,*n*) ⟹
{&(*p*→*f*)..+*size-td t*} ⊆ {*ptr-val* (*p*::*'a ptr*)..+*size-of TYPE*(*'a*)}
apply(*clarsimp simp*: *field-ti-def split*: *option.splits*)
apply(*drule intvlD*, *clarsimp simp*: *field-lvalue-def field-offset-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*subst add.assoc*)
apply(*subst Abs-fnat-homs*)
apply(*rule intvlI*)
apply(*simp add*: *size-of-def typ-uinfo-t-def*)
apply(*drule td-set-field-lookupD*)
apply(*drule td-set-offset-size*)
apply(*simp*)
done

lemma *typ-slice-t-not-empty* [*simp*]:
typ-slice-t t n ≠ []
by (*cases t*, *simp*)

lemma *list-map-typ-slice-t-not-empty* [*simp*]:
list-map (*typ-slice-t t n*) ≠ *Map.empty*
by(*simp add*: *list-map-def*)

lemma (**in** *c-type*) *s-footprint*:
s-footprint (*p*::*'a ptr*) =
{(*ptr-val p* + *of-nat x*,*k*) | *x k*.

$x < size\text{-}of\ TYPE('a) \wedge$
$(k{=}SIndexVal \vee (\exists\, n.\ k{=}SIndexTyp\ n \wedge n < length\ (typ\text{-}slice\text{-}t\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ x)))\}$
by (*auto simp*: *s-footprint-def s-footprint-untyped-def size-of-def*)

lemma (**in** *mem-type*) *ptr-val-SIndexVal-in-s-footprint* [*simp*]:
(*ptr-val p*, *SIndexVal*) ∈ *s-footprint* (*p*::'*a ptr*)
apply(*simp add*: *s-footprint*)
apply(*rule exI* [**where** *x=0*])
by *auto*

lemma (**in** *c-type*) *s-footprintI*:
⟦ *n* < *length* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *x*); *x* < *size-of TYPE*('*a*) ⟧ ⟹
(*ptr-val p* + *of-nat x*, *SIndexTyp n*) ∈ *s-footprint* (*p*::'*a ptr*)
apply(*simp add*: *s-footprint*)
apply(*rule exI* [**where** *x=x*])
apply *auto*
done

lemma (**in** *c-type*) *s-footprintI2*:
x < *size-of TYPE*('*a*) ⟹ (*ptr-val p* + *of-nat x*, *SIndexVal*) ∈ *s-footprint* (*p*::'*a ptr*)
apply(*simp add*: *s-footprint*)
apply(*rule exI* [**where** *x=x*])
apply *auto*
done

lemma (**in** *c-type*) *s-footprintD*:
(*x*,*k*) ∈ *s-footprint p* ⟹ *x* ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)}
by (*auto simp*: *s-footprint elim*: *intvlI*)

lemma (**in** *mem-type*) *s-footprintD2*:
(*x*,*SIndexTyp n*) ∈ *s-footprint* (*p*::'*a ptr*) ⟹
n < *length* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*x* − *ptr-val p*)))
by (*clarsimp simp add*: *s-footprint*)
(*metis len-of-addr-card max-size nat-less-le of-nat-inverse order-less-le-trans*)

lemma (**in** *c-type*) *s-footprint-restrict*:
x ∈ *s-footprint p* ⟹ (*s* |` *s-footprint p*) *x* = *s x*
by (*rule restrict-in*)

lemma *restrict-s-fst*:
fst (*restrict-s d X x*) ⟹ *fst* (*d x*)
by (*clarsimp simp*: *restrict-s-def*)

lemma *restrict-s-map-le* [*simp*]:
snd (*restrict-s d X x*) ⊆$_m$ *snd* (*d x*)
by (*auto simp*: *restrict-s-def map-le-def*)

lemma *dom-list-map* [*simp*]:
dom (*list-map xs*) = {*0*..<*length xs*}
by (*auto simp*: *list-map-def*)

lemma *list-map* [*simp*]:
n < *length xs* ⟹ *list-map xs n* = *Some* (*xs* ! *n*)
by (*force simp*: *list-map-def set-zip*)

lemma *list-map-eq*:
list-map xs n = (*if n* < *length xs then Some* (*xs* ! *n*) *else None*)
by (*force simp*: *list-map-def set-zip*)

lemma *valid-footprintI*:
⟦ *0* < *size-td t*; ⋀*y*. *y* < *size-td t* ⟹ *list-map* (*typ-slice-t t y*) ⊆$_m$ *snd* (*d* (*x* + *of-nat y*)) ∧
fst (*d* (*x* + *of-nat y*)) ⟧ ⟹
valid-footprint d x t
by (*simp add*: *valid-footprint-def*)

lemma *valid-footprintD*:
⟦ *valid-footprint d x t*; *y* < *size-td t* ⟧ ⟹
list-map (*typ-slice-t t y*) ⊆$_m$ *snd* (*d* (*x* + *of-nat y*)) ∧
fst (*d* (*x* + *of-nat y*))
by (*simp add*: *valid-footprint-def Let-def*)

lemma (**in** *c-type*) *h-t-valid-taut*:
d,*g* ⊨$_t$ *p* ⟹ *d*,(λ*x*. *True*) ⊨$_t$ *p*
by (*simp add*: *h-t-valid-def*)

lemma (**in** *c-type*) *h-t-valid-restrict*:
restrict-s d (*s-footprint p*),*g* ⊨$_t$ *p* = *d*,*g* ⊨$_t$ *p*
apply (*simp add*: *h-t-valid-def valid-footprint-def Let-def*)
apply (*fastforce simp*: *restrict-s-def map-le-def size-of-def intro*: *s-footprintI s-footprintI2*)
done

lemma (**in** *c-type*) *h-t-valid-restrict2*:
⟦ *d*,*g* ⊨$_t$ *p*; *restrict-s d* (*s-footprint p*) = *restrict-s d'* (*s-footprint p*)
⟧ ⟹ *d'*,*g* ⊨$_t$ (*p*::'*a ptr*)
apply(*clarsimp simp*: *h-t-valid-def valid-footprint-def Let-def*)
apply(*rule conjI*; *clarsimp?*)
apply(*clarsimp simp*: *map-le-def*)
apply(*drule-tac x*=(*ptr-val p* + *of-nat y*) **in** *fun-cong*)
apply(*clarsimp simp*: *restrict-s-def*)
apply(*drule-tac x*=*a* **in** *fun-cong*)
apply(*fastforce simp*: *size-of-def intro*: *s-footprintI split*: *if-split-asm*)
apply(*drule-tac x*=(*ptr-val p* + *of-nat y*) **in** *fun-cong*)
apply(*fastforce simp*: *size-of-def restrict-s-def intro*: *s-footprintI2*)
done

lemma *lift-state-wf-heap-val* [*simp*]:
wf-heap-val (*lift-state* (*h*,*d*))
unfolding *wf-heap-val-def*
by (*auto simp*: *lift-state-def split*: *option.splits*)

lemma *wf-hs-proj-d*:
fst (*proj-d s x*) ⟹ *s* (*x*,*SIndexVal*) ≠ *None*
by (*auto simp*: *proj-d-def*)

lemma (**in** *c-type*) *s-valid-g*:
s,*g* ⊨$_s$ *p* ⟹ *g p*
by (*simp add*: *s-valid-def h-t-valid-def*)

lemma (**in** *c-type*) *lift-typ-heap-if*:
lift-typ-heap g s = (λ(*p*::'*a ptr*). *if s*,*g* ⊨$_s$ *p then Some* (*from-bytes*
(*heap-list-s s* (*size-of TYPE*('*a*)) (*ptr-val p*))) *else None*)
by (*force simp*: *lift-typ-heap-def*)

lemma (**in** *c-type*) *lift-typ-heap-s-valid*:
lift-typ-heap g s p = *Some x* ⟹ *s*,*g* ⊨$_s$ *p*
by (*simp add*: *lift-typ-heap-if split*: *if-split-asm*)

lemma (**in** *c-type*) *lift-typ-heap-g*:
lift-typ-heap g s p = *Some x* ⟹ *g p*
by (*fast dest*: *lift-typ-heap-s-valid s-valid-g*)

lemma *lift-state-empty* [*simp*]:
lift-state (*h*,*empty-htd*) = *Map.empty*
by (*auto simp*: *lift-state-def empty-htd-def split*: *s-heap-index.splits*)

lemma *lift-state-eqI*:
⟦ *h x* = *h*′ *x*; *d x* = *d*′ *x* ⟧ ⟹ *lift-state* (*h*,*d*) (*x*,*k*) = *lift-state* (*h*′,*d*′) (*x*,*k*)
by (*clarsimp simp*: *lift-state-def split*: *s-heap-index.splits*)

lemma *proj-h-lift-state*:
fst (*d x*) ⟹ *proj-h* (*lift-state* (*h*,*d*)) *x* = *h x*
by (*clarsimp simp*: *proj-h-def lift-state-def*)

lemma *lift-state-proj-simp* [*simp*]:
lift-state (*proj-h* (*lift-state* (*h*, *d*)), *d*) = *lift-state* (*h*, *d*)
by (*auto simp*: *lift-state-def proj-h-def split*: *s-heap-index.splits option.splits*)

lemma *f2l-length* [*simp*]:
length (*fun2list f n*) = *n*
by (*simp add*: *fun2list-def*)

lemma *GREATEST-lt* [*simp*]:
0 < *n* ⟹ (*GREATEST x*. *x* < *n*) = *n* − (*1*::*nat*)

by (*rule Greatest-equality*; *simp*)

lemma *fun2list-nth* [*simp*]:
x < *n* ⟹ *fun2list f n* ! *x* = *f x*
by (*clarsimp simp*: *fun2list-def*)

lemma *proj-d-lift-state*:
proj-d (*lift-state* (*h*,*d*)) = *d*
apply(*rule ext*)
subgoal for *x*
apply(*cases d x*)
apply(*auto simp*: *proj-d-def lift-state-def Let-def split*: *option.splits*)
done
done

lemma *lift-state-proj* [*simp*]:
wf-heap-val s ⟹ *lift-state* (*proj-h s*,*proj-d s*) = *s*
apply (*clarsimp simp*: *proj-h-def proj-d-def lift-state-def fun-eq-iff*
split: *if-split-asm s-heap-index.splits option.splits*)
apply *safe*
apply (*metis s-heap-tag.simps s-heap-value.exhaust wf-heap-val-SIndexTyp-SValue-simp*)
apply (*metis id-apply s-heap-value.exhaust s-heap-value.simps*(5) *wf-heap-val-SIndexVal-STyp-simp*)
apply (*metis s-heap-tag.simps s-heap-value.exhaust wf-heap-val-SIndexTyp-SValue-simp*)
done

lemma *lift-state-Some*:
lift-state (*h*,*d*) (*p*,*SIndexTyp n*) = *Some t* ⟹ *snd* (*d p*) *n* = *Some* (*s-heap-tag t*)
apply (*simp add*: *lift-state-def split*: *option.splits split*: *if-split-asm*)
apply (*cases t*; *simp*)
done

lemma *lift-state-Some2*:
snd (*d p*) *n* = *Some t* ⟹
∃ *v*. *lift-state* (*h*,*d*) (*p*,*SIndexTyp n*) = *Some* (*STyp t*)
by (*simp add*: *lift-state-def split*: *option.split*)

lemma (**in** *c-type*) *h-t-s-valid*:
lift-state (*h*,*d*),*g* ⊨*s p* = *d*,*g* ⊨*t p*
by (*simp add*: *s-valid-def proj-d-lift-state*)

lemma (**in** *c-type*) *lift-t*:
lift-typ-heap g (*lift-state s*) = *lift-t g s*
by (*simp add*: *lift-t-def*)

lemma (**in** *c-type*) *lift-t-h-t-valid*:
lift-t g (*h*,*d*) *p* = *Some x* ⟹ *d*,*g* ⊨*t p*
by (*force simp*: *lift-t-def h-t-s-valid dest*: *lift-typ-heap-s-valid*)

lemma (**in** *c-type*) *lift-t-g*:

lift-t g s p = Some x ⟹ g p
by (*force simp*: *lift-t-def dest*: *lift-typ-heap-g*)

lemma (**in** *c-type*) *lift-t-proj* [*simp*]:
wf-heap-val s ⟹ lift-t g (*proj-h s*, *proj-d s*) = *lift-typ-heap g s*
by (*simp add*: *lift-t-def*)

lemma *valid-footprint-Some*:
assumes *valid*: *valid-footprint d p t* **and** *size*: *x* < *size-td t*
shows *fst* (*d* (*p* + *of-nat x*))
proof (*cases of-nat x*=(*0*::*addr*))
case *True*
with *valid* **show** *?thesis* **by** (*force simp add*: *valid-footprint-def Let-def*)
next
case *False*
with *size valid* **show** *?thesis* **by** (*force simp*: *valid-footprint-def Let-def*)
qed

lemma (**in** *c-type*) *h-t-valid-Some*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a ptr*); *x* < *size-of TYPE*('*a*) ⟧ ⟹
fst (*d* (*ptr-val p* + *of-nat x*))
by (*force simp*: *h-t-valid-def size-of-def dest*: *valid-footprint-Some*)

lemma (**in** *c-type*) *h-t-valid-ptr-safe*:
d,*g* ⊨$_t$ (*p*::'*a ptr*) ⟹ *ptr-safe p d*
apply(*clarsimp simp*: *ptr-safe-def h-t-valid-def valid-footprint-def s-footprint-def*
s-footprint-untyped-def dom-s-def size-of-def Let-def)
by (*metis* (*mono-tags*, *opaque-lifting*) *domIff list-map map-le-def option.simps*(*3*)
surj-pair)

lemma (**in** *c-type*) *lift-t-ptr-safe*:
lift-t g (*h*,*d*) (*p*::'*a ptr*) = *Some x* ⟹ *ptr-safe p d*
by (*fast dest*: *lift-t-h-t-valid h-t-valid-ptr-safe*)

lemma (**in** *c-type*) *s-valid-Some*:
⟦ *d*,*g* ⊨$_s$ (*p*::'*a ptr*); *x* < *size-of TYPE*('*a*) ⟧ ⟹
d (*ptr-val p* + *of-nat x*,*SIndexVal*) ≠ *None*
by (*auto simp*: *s-valid-def dest*!: *h-t-valid-Some wf-hs-proj-d split*: *option.splits*)

lemma *heap-list-s-heap-list-dom*:
⋀*n*. (λ*x*. (*x*,*SIndexVal*)) ' {*n*..+*k*} ⊆ *dom-s d* ⟹
heap-list-s (*lift-state* (*h*,*d*)) *k n* = *heap-list h k n*
proof (*induct k*)
case *0* **show** *?case* **by** (*simp add*: *heap-list-s-def*)
next
case (*Suc k*)
hence (λ*x*. (*x*,*SIndexVal*)) ' {*n* + *1*..+*k*} ⊆ *dom-s d*
by (*force intro*: *intvl-plus-sub-Suc subset-trans simp*: *image-def*)
with *Suc* **have** *heap-list-s* (*lift-state* (*h*, *d*)) *k* (*n* + *1*) =

```
    heap-list h k (n + 1) by simp
  moreover from this Suc have (n,SIndexVal) ∈ dom-s d
    by (force simp: dom-s-def image-def intro: intvl-self)
  ultimately show ?case
    by (auto simp add: heap-list-s-def proj-h-lift-state dom-s-def)
qed

lemma (in c-type) heap-list-s-heap-list:
  d,(λx. True) ⊨t (p::'a ptr) ⟹
        heap-list-s (lift-state (h,d)) (size-of TYPE('a)) (ptr-val p)
           = heap-list h (size-of TYPE('a)) (ptr-val p)
  apply(drule h-t-valid-ptr-safe)
  apply(clarsimp simp: ptr-safe-def)
  apply(subst heap-list-s-heap-list-dom)
   apply(clarsimp simp: dom-s-def)
   apply(drule-tac c=(x,SIndexVal) in subsetD)
    apply(clarsimp simp: intvl-def)
    apply(erule s-footprintI2)
   apply clarsimp+
  done

lemma (in c-type) lift-t-if:
  lift-t g (h,d) = (λp. if d,g ⊨t p then Some (h-val h (p::'a ptr)) else None)
  by (force simp: lift-t-def lift-typ-heap-if h-t-s-valid h-val-def
                  heap-list-s-heap-list h-t-valid-taut)

lemma (in c-type) lift-lift-t:
  d,g ⊨t (p::'a ptr) ⟹ lift h p = the (lift-t g (h,d) p)
  by (simp add: lift-t-if lift-def)

lemma (in c-type) lift-t-lift:
  lift-t g (h,d) (p::'a ptr) = Some v ⟹ lift h p = v
  by (simp add: lift-t-if lift-def split: if-split-asm)

lemma heap-update-list-same:
  ⋀h p k. ⟦ 0 < k; k ≤ addr-card − length v ⟧ ⟹ heap-update-list (p + of-nat k)
v h p = h p
proof (induct v)
  case Nil show ?case by simp
next
  case (Cons x xs)
  have heap-update-list (p + of-nat k) (x # xs) h p =
       heap-update-list (p + of-nat (k + 1)) xs (h(p + of-nat k := x)) p
    by (simp add: ac-simps)
  also have ... = (h(p + of-nat k := x)) p
  proof −
    from Cons have k + 1 ≤ addr-card − length xs by simp
    with Cons show ?thesis by (simp only:)
  qed
```

also have $\ldots = h\ p$
proof $-$
from *Cons* **have** *of-nat* $k \neq (0::addr)$
by $-$ (*erule of-nat-neq-0*, *simp add*: *addr-card*)
thus *?thesis* **by** *clarsimp*
qed
finally show *?case* **.**
qed

lemma *heap-list-update*:
$\bigwedge h\ p.$ *length* $v \leq$ *addr-card* $\Longrightarrow$
heap-list (*heap-update-list* $p\ v\ h$) (*length* v) $p = v$
proof (*induct* v)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* $x\ xs$)
hence *heap-update-list* (p + *of-nat 1*) xs ($h(p := x)$) $p = (h(p := x))\ p$
by $-$ (*rule heap-update-list-same*, *auto*)
with *Cons* **show** *?case* **by** *simp*
qed

lemma (**in** *mem-type*) *heap-list-update-to-bytes*:
heap-list (*heap-update-list* p (*to-bytes* ($v::'a$) (*heap-list* h (*size-of* $TYPE('a)$) p)) h)
(*size-of* $TYPE('a)$) p = *to-bytes* v (*heap-list* h (*size-of* $TYPE('a)$) p)
by (*metis* (*mono-tags*) *heap-list-length heap-list-update len less-imp-le max-size*)

lemma (**in** *mem-type*) *heap-list-update-to-bytes-padding*:
length bs = *size-of* $TYPE('a) \Longrightarrow$ *heap-list* (*heap-update-list* p (*to-bytes* ($v::'a$) bs) h)
(*size-of* $TYPE('a)$) p = *to-bytes* v bs
by (*metis heap-list-update local.len local.max-size nless-le*)

lemma (**in** *mem-type*) *h-val-heap-update*[*simp*]:
h-val (*heap-update* $p\ v\ h$) $p = (v::'a)$
by (*simp add*: *h-val-def heap-update-def heap-list-update-to-bytes inv*)

lemma (**in** *mem-type*) *h-val-heap-update-padding*:
fixes p:: *'a ptr*
shows *length* bs = *size-of* $TYPE('a) \Longrightarrow$ *h-val* (*heap-update-padding* $p\ v\ bs\ h$) $p = v$
apply (*simp add*: *heap-update-padding-def h-val-def*)
by (*metis heap-list-update inv le-eq-less-or-eq len max-size*)

lemma *heap-list-update-disjoint-same*:
shows $\bigwedge q.$ $\{p..+$*length* $v\} \cap \{q..+k\} = \{\} \Longrightarrow$
heap-list (*heap-update-list* $p\ v\ h$) $k\ q$ = *heap-list* $h\ k\ q$
proof (*induct* k)
case *0* **show** *?case* **by** *simp*

next
case (*Suc n*)
hence {*p*..+*length v*} ∩ {*q* + *1*..+*n*} = {}
by (*force intro*: *intvl-plus-sub-Suc*)
with *Suc* **have** *heap-list* (*heap-update-list p v h*) *n* (*q* + *1*) =
heap-list h n (*q* + *1*) **by** *simp*
moreover have *heap-update-list* (*q* + *of-nat* (*unat* (*p* − *q*))) *v h q* = *h q*
proof (*cases v*)
case *Nil* **thus** *?thesis* **by** *simp*
next
case (*Cons y ys*)
with *Suc* **have** *0* < *unat* (*p* − *q*)
by (*cases p*=*q*)
(*simp add*: *intvl-start-inter unat-gt-0*)+
moreover have *unat* (*p* − *q*) ≤ *addr-card* − *length v* (**is** *?G*)
proof (*rule ccontr*)
assume ¬ *?G*
moreover from *Suc* **have** *q* ∉ {*p*..+*length v*}
by (*fast intro*: *intvl-self*)
ultimately show *False*
by (*simp only*: *linorder-not-le len-of-addr-card* [*symmetric*])
(*frule-tac p*=*q* **in** *intvl-self-offset*, *force*+)
qed
ultimately show *?thesis* **by** (*rule heap-update-list-same*)
qed
ultimately show *?case* **by** *simp*
qed

lemma *heap-update-nmem-same*:
assumes *nmem*: *q* ∉ {*p*..+*length v*}
shows *heap-update-list p v h q* = *h q*
proof −
from *nmem* **have** *heap-list* (*heap-update-list p v h*) *1 q* = *heap-list h 1 q*
by − (*rule heap-list-update-disjoint-same*, *force dest*: *intvl-Suc*)
thus *?thesis* **by** *simp*
qed

lemma *heap-update-mem-same*:
⟦ *q* ∈ {*p*..+*length v*}; *length v* < *addr-card* ⟧ ⟹
heap-update-list p v h q = *heap-update-list p v h′ q*
by (*induct v arbitrary*: *p h h′*; *simp*)
(*fastforce dest*: *intvl-neq-start simp*: *heap-update-list-same* [**where** *k*=*1*, *simplified*])

lemma *sub-tag-proper-TypScalar* [*simp*]:
¬ *t* < *TypDesc algn′* (*TypScalar n algn d*) *nm*
by (*simp add*: *typ-tag-lt-def typ-tag-le-def*)

lemma *tag-disj-com* [*simp*]:

$f \perp_t g = g \perp_t f$
by (*force simp*: *tag-disj-def*)

lemma *typ-slice-set'*:
$\forall m\ n.\ fst\ `\ set\ (typ\text{-}slice\text{-}t\ s\ n) \subseteq fst\ `\ td\text{-}set\ s\ m$
$\forall m\ n.\ fst\ `\ set\ (typ\text{-}slice\text{-}struct\ st\ n) \subseteq fst\ `\ td\text{-}set\text{-}struct\ st\ m$
$\forall m\ n.\ fst\ `\ set\ (typ\text{-}slice\text{-}list\ xs\ n) \subseteq fst\ `\ td\text{-}set\text{-}list\ xs\ m$
$\forall m\ n.\ fst\ `\ set\ (typ\text{-}slice\text{-}tuple\ x\ n) \subseteq fst\ `\ td\text{-}set\text{-}tuple\ x\ m$
apply(*induct s* **and** *st* **and** *xs* **and** *x*, *all* ‹*clarsimp simp*: *ladder-set-def*›)
apply *auto*[*1*]
apply (*rule conjI*; *clarsimp*)
apply *force*
apply(*thin-tac All P* **for** *P*)
apply *force*
done

lemma *typ-slice-set*:
$fst\ `\ set\ (typ\text{-}slice\text{-}t\ s\ n) \subseteq fst\ `\ td\text{-}set\ s\ m$
using *typ-slice-set'*(*1*) [*of s*] **by** *clarsimp*

lemma *typ-slice-struct-set*:
$(s,t) \in set\ (typ\text{-}slice\text{-}struct\ st\ n) \Longrightarrow \exists k.\ (s,k) \in td\text{-}set\text{-}struct\ st\ m$
using *typ-slice-set'*(*2*) [*of st*] **by** *force*

lemma *typ-slice-set-sub*:
$typ\text{-}slice\text{-}t\ s\ m \leq typ\text{-}slice\text{-}t\ t\ n \Longrightarrow$
$fst\ `\ set\ (typ\text{-}slice\text{-}t\ s\ m) \subseteq fst\ `\ set\ (typ\text{-}slice\text{-}t\ t\ n)$
by (*force simp*: *image-def prefix-def less-eq-list-def*)

lemma *ladder-set-self*:
$s \in fst\ `\ set\ (typ\text{-}slice\text{-}t\ s\ n)$
by (*cases s*) (*auto simp*: *ladder-set-def*)

lemma *typ-slice-sub*:
$typ\text{-}slice\text{-}t\ s\ m \leq typ\text{-}slice\text{-}t\ t\ n \Longrightarrow s \leq t$
apply(*drule typ-slice-set-sub*)
using *ladder-set-self* [*of s m*] *typ-slice-set* [*of t n 0*]
apply(*force simp*: *typ-tag-le-def*)
done

lemma *typ-slice-self*:
$(s, True) \in set\ (typ\text{-}slice\text{-}t\ s\ 0)$
by (*cases s*) *simp*

lemma *typ-slice-struct-nmem*:
$(TypDesc\ algn\ st\ nm,n) \notin set\ (typ\text{-}slice\text{-}struct\ st\ k)$
by (*fastforce dest*: *typ-slice-struct-set td-set-struct-size-lte*)

lemma *typ-slice-0-prefix*:

$0 < n \Longrightarrow \neg$ *typ-slice-t t 0* $\leq$ *typ-slice-t t n* $\wedge \neg$ *typ-slice-t t n* $\leq$ *typ-slice-t t 0*
by (*cases t*) (*fastforce simp*: *less-eq-list-def typ-slice-struct-nmem dest*: *set-mono-prefix*)

lemma *prefix-eq-nth*:
xs $\leq$ *ys* = ((∀ *i*. *i* < *length xs* ⟶ *xs* ! *i* = *ys* ! *i*) ∧ *length xs* $\leq$ *length ys*)
apply(*rule iffI*; *clarsimp simp*: *less-eq-list-def prefix-def nth-append*)
by (*metis append-take-drop-id nth-take-lemma order-refl take-all*)

lemma *map-prefix-same-cases*:
⟦ *list-map xs* $\subseteq_m$ *f*; *list-map ys* $\subseteq_m$ *f* ⟧ $\Longrightarrow$ *xs* $\leq$ *ys* ∨ *ys* $\leq$ *xs*
using *linorder-linear*[**where** *x*=*length xs* **and** *y*=*length ys*]
apply(*clarsimp simp*: *prefix-eq-nth map-le-def prefix-def*)
apply (*erule disjE*)
apply *clarsimp*
subgoal for *i*
by ((*drule-tac x*=*i* **in** *bspec*, *simp*)+, *force dest*: *sym*)
apply *clarsimp*
subgoal for *i*
by ((*drule-tac x*=*i* **in** *bspec*, *simp*)+, *force dest*: *sym*)
done

lemma *list-map-mono*:
xs $\leq$ *ys* $\Longrightarrow$ *list-map xs* $\subseteq_m$ *list-map ys*
by (*auto simp*: *map-le-def prefix-def nth-append less-eq-list-def*)

lemma *map-list-map-trans*:
⟦ *xs* $\leq$ *ys*; *list-map ys* $\subseteq_m$ *f* ⟧ $\Longrightarrow$ *list-map xs* $\subseteq_m$ *f*
apply(*drule list-map-mono*)
apply(*erule* (*1*) *map-le-trans*)
done

lemma *valid-footprint-le*:
valid-footprint d x t $\Longrightarrow$ *size-td t* $\leq$ *addr-card*
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply(*rule ccontr*)
apply(*frule-tac x*=*addr-card* **in** *spec*)
apply(*drule-tac x*=*0* **in** *spec*)
apply *clarsimp*
apply(*drule* (*1*) *map-prefix-same-cases*)
apply(*simp add*: *typ-slice-0-prefix addr-card*)
done

lemma *typ-slice-True-set′*:
∀ *s k m*. (*s*,*True*) ∈ *set* (*typ-slice-t t k*) ⟶ (*s*,*k*+*m*) ∈ *td-set t m*
∀ *s k m*. (*s*,*True*) ∈ *set* (*typ-slice-struct st k*) ⟶ (*s*,*k*+*m*) ∈ *td-set-struct st m*
∀ *s k m*. (*s*,*True*) ∈ *set* (*typ-slice-list xs k*) ⟶ (*s*,*k*+*m*) ∈ *td-set-list xs m*
∀ *s k m*. (*s*,*True*) ∈ *set* (*typ-slice-tuple x k*) ⟶ (*s*,*k*+*m*) ∈ *td-set-tuple x m*
proof (*induct t* **and** *st* **and** *xs* **and** *x*)
case (*TypDesc nat typ-struct list*)

```
  then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
   apply clarsimp
   apply(cases dt-tuple, clarsimp)
    subgoal for s k m a b
   apply(thin-tac All P for P)
   apply(drule spec [where x=s])
   apply(drule spec [where x=k − size-td a])
   apply clarsimp
   apply(drule spec [where x=m + size-td a])
     apply simp
     done
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed

lemma typ-slice-True-set:
  (s,True) ∈ set (typ-slice-t t k) ⟹ (s,k+m) ∈ td-set t m
  by (simp add: typ-slice-True-set′)

lemma typ-slice-True-prefix:
  typ-slice-t s 0 ≤ typ-slice-t t k ⟹ (s,k) ∈ td-set t 0
  using typ-slice-self [of s] typ-slice-True-set [of s t k 0]
  by (force simp: less-eq-list-def dest: set-mono-prefix)

lemma tag-sub-prefix [simp]:
  t < s ⟹ ¬ typ-slice-t s m ≤ typ-slice-t t n
  by (fastforce dest: typ-slice-sub)

lemma tag-disj-prefix [simp]:
  s ⊥t t ⟹ ¬ typ-slice-t s m ≤ typ-slice-t t n
  by (auto dest: typ-slice-sub simp: tag-disj-def typ-slice-sub)

lemma typ-slice-0-True′:
  ∀ x. x ∈ set (typ-slice-t t 0) ⟶ snd x = True
  ∀ x. x ∈ set (typ-slice-struct st 0) ⟶ snd x = True
```

```
  ∀x. x ∈ set (typ-slice-list xs 0) ⟶ snd x = True
  ∀x. x ∈ set (typ-slice-tuple y 0) ⟶ snd x = True
  by (induct t and st and xs and y)  auto

lemma typ-slice-0-True:
  x ∈ set (typ-slice-t t 0) ⟹ snd x = True
  by (simp add: typ-slice-0-True')

lemma typ-slice-False-self:
  k ≠ 0 ⟹ (t,False) ∈ set (typ-slice-t t k)
  by (cases t) simp

lemma tag-prefix-True:
  typ-slice-t s k ≤ typ-slice-t t 0 ⟹ k = 0
  using typ-slice-0-True [of (s,False) t]
  apply(clarsimp simp: less-eq-list-def dest!: set-mono-prefix)
  apply(rule ccontr)
  apply(fastforce dest!: typ-slice-False-self[where t=s and k=k])
  done

lemma valid-footprint-neq-nmem:
  assumes valid-p: valid-footprint d p f and
          valid-q: valid-footprint d q g and
              neq: p ≠ q and
             disj: f ⊥t g ∨ f=g
  shows p ∉ {q..+size-td g} (is ?G)
proof −
  from assms show ?thesis
    apply(clarsimp simp: valid-footprint-def intvl-def Let-def)
    apply(erule disjE)
     apply(drule-tac x=0 in spec)
     apply (fastforce dest: map-prefix-same-cases)
    apply(drule-tac x=0 in spec)
    apply(drule-tac x=k in spec)
    apply clarsimp
   by (metis add.comm-neutral map-prefix-same-cases neq semiring-1-class.of-nat-0

      typ-slice-0-prefix zero-less-iff-neq-zero)
qed

lemma valid-footprint-sub:
  assumes valid-p: valid-footprint d p s
  assumes valid-q: valid-footprint d q t
  assumes sub: ¬ t < s
  shows p ∉ {q..+size-td t} ∨ field-of (p − q) (s) (t) (is ?G)
proof −
  from assms show ?thesis
    apply clarsimp
    apply(insert valid-footprint-le[OF valid-q])
```

```
    apply(clarsimp simp: valid-footprint-def Let-def)
    apply(drule-tac x=0 in spec)
    apply clarsimp
    apply(drule intvlD)
    apply clarsimp
    apply(drule-tac x=k in spec)
    apply clarsimp
    apply(drule (1) map-prefix-same-cases)
    apply(erule disjE)
     prefer 2
     apply(frule typ-slice-sub)
     apply(subgoal-tac k = 0)
      prefer 2
      apply(rule ccontr, simp)
      apply(simp add: typ-slice-0-prefix)
     apply simp

    apply(drule typ-slice-True-prefix)
    apply(clarsimp simp: field-of-def)
    apply(simp only: unat-simps)
    done
qed

lemma valid-footprint-sub2:
  ⟦ valid-footprint d p s; valid-footprint d q t; ¬ t < s ⟧ ⟹
      q ∉ {p..+size-td s} ∨ p=q
  apply(clarsimp simp: valid-footprint-def Let-def)
  apply(drule intvlD)
  apply clarsimp
  subgoal for k
    apply(drule spec [where x=k])
    apply clarsimp
    apply(drule spec [where x=0])
    apply clarsimp
    apply(drule (1) map-prefix-same-cases)
    apply(cases k=0)
     apply simp
    apply(erule disjE)
     prefer 2
     apply(frule typ-slice-sub)
     apply(drule order-le-imp-less-or-eq[where x=t])
     apply clarsimp
     apply(simp add: typ-slice-0-prefix)
    apply(drule tag-prefix-True)
    apply simp
    done
  done

lemma valid-footprint-neq-disjoint:
```

⟦ *valid-footprint d p s*; *valid-footprint d q t*; ¬ $t < s$;
¬ *field-of* $(p - q)$ (s) (t) ⟧ ⟹ $\{p..+size\text{-}td\ s\} \cap \{q..+size\text{-}td\ t\} = \{\}$
apply(*rule ccontr*)
apply(*drule intvl-inter*)
apply(*erule disjE*)
apply(*drule* (*2*) *valid-footprint-sub*)
apply *clarsimp*
apply(*frule* (*1*) *valid-footprint-sub2*, *assumption*)
apply(*frule* (*1*) *valid-footprint-sub2*)
apply *simp*
apply *simp*
apply(*clarsimp simp*: *field-of-def*)
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply(*drule-tac x=0* **in** *spec*)+
apply *clarsimp*
apply(*drule* (*1*) *map-prefix-same-cases* [**where** *xs=typ-slice-t s 0*])
apply(*erule disjE*)
apply(*drule typ-slice-True-prefix*)
apply *simp*
apply(*drule typ-slice-sub*)
apply(*drule order-le-imp-less-or-eq*)
apply *simp*
done

lemma *sub-typ-proper-not-same* [*simp*]:
¬ $t <_\tau t$
by (*simp add*: *sub-typ-proper-def*)

lemma *sub-typ-proper-not-simple* [*simp*]:
¬ $TYPE('a::c\text{-}type) <_\tau TYPE('b::simple\text{-}mem\text{-}type)$
apply(*cases typ-uinfo-t* $TYPE('b)$)
subgoal for *x1 typ-struct*
by(*cases typ-struct*, *auto simp*: *sub-typ-proper-def*)
done

lemma *field-of-sub*:
field-of p s t ⟹ $s \leq t$
by (*auto simp*: *field-of-def typ-tag-lt-def typ-tag-le-def*)

lemma *h-t-valid-neq-disjoint*:
⟦ *d*,*g* ⊨$_t$ $(p::'a::c\text{-}type\ ptr)$; *d*,*g*′ ⊨$_t$ $(q::'b::c\text{-}type\ ptr)$;
¬ $TYPE('b) <_\tau TYPE('a)$; ¬ *field-of-t p q* ⟧ ⟹ $\{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\}$
∩
$\{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}$
by (*fastforce dest*: *valid-footprint-neq-disjoint*
simp: *size-of-def h-t-valid-def sub-typ-proper-def field-of-t-def*)

lemma *field-ti-sub-typ*:
⟦ *field-ti* $(TYPE('b::mem\text{-}type))$ f = *Some t*; *export-uinfo t* = $(typ\text{-}uinfo\text{-}t\ TYPE('a::c\text{-}type))$

⟧ ⟹
    $TYPE('a) \leq_\tau TYPE('b)$
  by (*auto simp*: *field-ti-def sub-typ-def typ-tag-le-def typ-uinfo-t-def*
        dest!: *td-set-export-uinfoD td-set-field-lookupD*
        split: *option.splits*)

lemma *h-t-valid-neq-disjoint-simple*:
  ⟦ *d,g* ⊨$_t$ (*p*::'*a*::*simple-mem-type ptr*); *d,g'* ⊨$_t$ (*q*::'*b*::*simple-mem-type ptr*) ⟧
    ⟹ *ptr-val p* ≠ *ptr-val q* ∨ *typ-uinfo-t TYPE*('*a*) = *typ-uinfo-t TYPE*('*b*)
  apply(*clarsimp simp*: *h-t-valid-def valid-footprint-def Let-def*)
  apply(*drule-tac x=0* **in** *spec*)+
  apply *clarsimp*
 apply(*drule* (*1*) *map-prefix-same-cases*[**where** *xs=typ-slice-t* (*typ-uinfo-t TYPE*('*a*))
0])
  apply(*erule disjE*; *drule typ-slice-sub*)
   apply(*cases typ-info-t TYPE*('*b*), *rename-tac typ-struct xs*)
  subgoal for *x1 typ-struct xs*
    apply(*cases typ-struct*; *fastforce simp*: *typ-tag-le-def typ-uinfo-t-def*)
    done
  apply(*cases typ-info-t TYPE*('*a*), *rename-tac typ-struct xs*)
  subgoal for *x1 typ-struct xs*
    apply(*cases typ-struct*; *simp add*: *typ-tag-le-def typ-uinfo-t-def*)
    done
  done

lemma *h-val-heap-same*:
  fixes *p*::'*a*::*mem-type ptr* **and** *q*::'*b*::*c-type ptr*
  assumes *val-p*: *d,g* ⊨$_t$ *p* **and** *val-q*: *d,g'* ⊨$_t$ *q* **and**
        subt: ¬ *TYPE*('*a*) <$_\tau$ *TYPE*('*b*) **and** *nf*: ¬ *field-of-t q p*
  shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
proof −
  from *val-p val-q subt nf*
  have {*ptr-val p..+length* (*to-bytes v* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)))}
∩
       {*ptr-val q..+size-of TYPE*('*b*)} = {}
    by (*force dest*: *h-t-valid-neq-disjoint*)
  hence *heap-list* (*heap-update-list* (*ptr-val p*) (*to-bytes v* (*heap-list h* (*size-of TYPE*('*a*))
(*ptr-val p*))) *h*)
                  (*size-of TYPE*('*b*)) (*ptr-val q*) = *heap-list h* (*size-of TYPE*('*b*))
(*ptr-val q*)
    by − (*erule heap-list-update-disjoint-same*)
  thus *?thesis* **by** (*simp add*: *h-val-def heap-update-def*)
qed

lemma *h-val-heap-same-padding*:
  fixes *p*::'*a*::*mem-type ptr* **and** *q*::'*b*::*c-type ptr*
  assumes *val-p*: *d,g* ⊨$_t$ *p* **and** *val-q*: *d,g'* ⊨$_t$ *q* **and**
        subt: ¬ *TYPE*('*a*) <$_\tau$ *TYPE*('*b*) **and** *nf*: ¬ *field-of-t q p*

assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *h-val* (*heap-update-padding p v bs h*) *q* = *h-val h q*
proof −
from *val-p val-q subt nf*
have {*ptr-val p..+length* (*to-bytes v bs*)} ∩
{*ptr-val q..+size-of TYPE*('*b*)} = {}
using *lbs*
by (*force dest*: *h-t-valid-neq-disjoint*)
hence *heap-list* (*heap-update-list* (*ptr-val p*) (*to-bytes v bs*) *h*)
(*size-of TYPE*('*b*)) (*ptr-val q*) = *heap-list h* (*size-of TYPE*('*b*))
(*ptr-val q*)
by − (*erule heap-list-update-disjoint-same*)
thus *?thesis* **by** (*simp add*: *h-val-def heap-update-padding-def*)
qed

lemma *peer-typI*:
typ-uinfo-t TYPE('*a*) ⊥$_t$ *typ-uinfo-t TYPE*('*b*) ⟹ *peer-typ* (*a*::'*a*::*c-type itself*)
(*b*::'*b*::*c-type itself*)
by (*simp add*: *peer-typ-def*)

lemma *peer-typD*:
peer-typ TYPE('*a*::*c-type*) *TYPE*('*b*::*c-type*) ⟹ ¬ *TYPE*('*a*) <$_\tau$ *TYPE*('*b*)
by (*clarsimp simp*: *peer-typ-def tag-disj-def sub-typ-proper-def order-less-imp-le*)

lemma *peer-typ-refl* [*simp*]:
peer-typ t t
by (*simp add*: *peer-typ-def*)

lemma *peer-typ-simple* [*simp*]:
peer-typ TYPE('*a*::*simple-mem-type*) *TYPE*('*b*::*simple-mem-type*)
apply(*clarsimp simp*: *peer-typ-def tag-disj-def typ-tag-le-def typ-uinfo-t-def*)
apply(*erule disjE*)
apply(*cases typ-info-t TYPE*('*b*))
subgoal for *x1 typ-struct xs*
by(*cases typ-struct*; *simp*)
apply(*cases typ-info-t TYPE*('*a*))
subgoal for *x1 typ-struct xs*
by(*cases typ-struct*; *simp*)
done

lemmas *peer-typ-nlt* = *peer-typD*

lemma *peer-typ-not-field-of*:
⟦ *peer-typ TYPE*('*a*::*c-type*) *TYPE*('*b*::*c-type*); *ptr-val p* ≠ *ptr-val q* ⟧ ⟹
¬ *field-of-t* (*q*::'*b ptr*) (*p*::'*a ptr*)
by (*fastforce simp*: *peer-typ-def field-of-t-def field-of-def tag-disj-def typ-tag-le-def*
unat-eq-zero
dest: *td-set-size-lte*)

lemma *h-val-heap-same-peer*:
⟦ *d*,*g* ⊨$_t$ (*p*::′*a*::*mem-type ptr*); *d*,*g*′ ⊨$_t$ (*q*::′*b*::*c-type ptr*);
ptr-val p ≠ *ptr-val q*; *peer-typ TYPE*(′*a*) *TYPE*(′*b*) ⟧ ⟹
h-val (*heap-update p v h*) *q* = *h-val h q*
apply(*erule* (*1*) *h-val-heap-same*)
apply(*erule peer-typ-nlt*)
apply(*erule* (*1*) *peer-typ-not-field-of*)
done

lemma *h-val-heap-same-peer-padding*:
⟦ *d*,*g* ⊨$_t$ (*p*::′*a*::*mem-type ptr*); *d*,*g*′ ⊨$_t$ (*q*::′*b*::*c-type ptr*);
ptr-val p ≠ *ptr-val q*; *peer-typ TYPE*(′*a*) *TYPE*(′*b*); *length bs* = *size-of TYPE*(′*a*) ⟧ ⟹
h-val (*heap-update-padding p v bs h*) *q* = *h-val h q*
apply(*erule* (*1*) *h-val-heap-same-padding*)
apply(*erule peer-typ-nlt*)
apply(*erule* (*1*) *peer-typ-not-field-of*)
apply *assumption*
done

lemma *field-offset-footprint-cons-simp* [*simp*]:
field-offset-footprint p (*x*#*xs*) = {*Ptr* &(*p*→*x*)} ∪ *field-offset-footprint p xs*
unfolding *field-offset-footprint-def* **by** (*cases x*) *auto*

lemma *heap-list-update-list*:
⟦ *n* + *x* ≤ *length v*; *length v* < *addr-card* ⟧ ⟹
heap-list (*heap-update-list p v h*) *n* (*p* + *of-nat x*) = *take n* (*drop x v*)
apply(*induct v arbitrary*: *n x p h*; *clarsimp*)
subgoal for *a list n x p h*
apply(*cases x*; *clarsimp*)
apply(*cases n*; *clarsimp*)
apply (*rule conjI*)
apply(*subgoal-tac heap-update-list* (*p* + *of-nat 1*) *list* (*h*(*p* := *a*)) *p* = *a*, *simp*)
apply(*subst heap-update-list-same*; *simp*)
apply(*metis add.right-neutral drop0 semiring-1-class.of-nat-0*)
subgoal for *nat*
apply(*drule meta-spec* [**where** *x*=*n*])
apply(*drule meta-spec* [**where** *x*=*nat*])
apply(*drule meta-spec* [**where** *x*=*p*+*1*])
apply (*simp add*: *add.assoc*)
done
done
done

lemma *typ-slice-td-set′*:
∀ *s m n k*. (*s*,*m* + *n*) ∈ *td-set t m* ∧ *k* < *size-td s* ⟶
typ-slice-t s k ≤ *typ-slice-t t* (*n* + *k*)
∀ *s m n k*. (*s*,*m* + *n*) ∈ *td-set-struct st m* ∧ *k* < *size-td s* ⟶

typ-slice-t s k ≤ *typ-slice-struct st* (*n* + *k*)
∀ *s m n k*. (*s*,*m* + *n*) ∈ *td-set-list ts m* ∧ *k* < *size-td s* ⟶
typ-slice-t s k ≤ *typ-slice-list ts* (*n* + *k*)
∀ *s m n k*. (*s*,*m* + *n*) ∈ *td-set-tuple x m* ∧ *k* < *size-td s* ⟶
typ-slice-t s k ≤ *typ-slice-tuple x* (*n* + *k*)
apply(*induct t* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp simp*: *split-DTuple-all*›)
apply(*safe*; *clarsimp*)
apply(*drule-tac x*=*s* **in** *spec*)
apply(*drule-tac x*=*m* **in** *spec*)
apply(*drule-tac x*=*0* **in** *spec*)
apply *fastforce*
apply (*safe*; *clarsimp?*)
apply(*fastforce dest*: *td-set-list-offset-le*)
apply(*fastforce dest*: *td-set-offset-size-m*)
apply(*rotate-tac*)
apply(*drule-tac x*=*s* **in** *spec*)
apply(*rename-tac a list s m n k*)
apply(*drule-tac x*=*m* + *size-td a* **in** *spec*)
apply(*drule-tac x*=*n* − *size-td a* **in** *spec*)
apply(*drule-tac x*=*k* **in** *spec*)
apply(*erule impE*)
apply(*subgoal-tac m* + *size-td a* + (*n* − *size-td a*) = *m* + *n*, *simp*)
apply(*fastforce dest*: *td-set-list-offset-le*)
apply(*fastforce dest*: *td-set-list-offset-le*)
done

lemma *typ-slice-td-set*:
⟦ (*s*,*n*) ∈ *td-set t 0*; *k* < *size-td s* ⟧ ⟹
typ-slice-t s k ≤ *typ-slice-t t* (*n* + *k*)
using *typ-slice-td-set'*(*1*) [*of t*]
apply(*drule-tac x*=*s* **in** *spec*)
apply(*drule-tac x*=*0* **in** *spec*)
apply *clarsimp*
done

lemma *typ-slice-td-set-list*:
⟦ (*s*,*n*) ∈ *td-set-list ts 0*; *k* < *size-td s* ⟧ ⟹
typ-slice-t s k ≤ *typ-slice-list ts* (*n* + *k*)
using *typ-slice-td-set'*(*3*) [*of ts*]
apply(*drule-tac x*=*s* **in** *spec*)
apply(*drule-tac x*=*0* **in** *spec*)
apply *clarsimp*
done

lemma *h-t-valid-sub*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*b*::*mem-type ptr*);
field-ti TYPE('*b*) *f* = *Some t*; *export-uinfo t* = (*typ-uinfo-t TYPE*('*a*)) ⟧ ⟹
d,(λ*x*. *True*) ⊨$_t$ ((*Ptr* &(*p*⟶*f*))::'*a*::*mem-type ptr*)
apply(*clarsimp simp*: *h-t-valid-def field-ti-def valid-footprint-def Let-def split*: *op-*

tion.splits)
apply(*rule conjI*)
apply(*simp add*: *typ-uinfo-t-def export-uinfo-def size-of-def* [*symmetric*, **where** *t*=*TYPE*('*a*)])
apply *clarsimp*
apply(*drule-tac x*=*field-offset TYPE*('*b*) *f* + *y* **in** *spec*)
apply(*erule impE*)
apply(*fastforce simp*: *field-offset-def field-offset-untyped-def typ-uinfo-t-def size-of-def*
dest: *td-set-offset-size td-set-field-lookupD field-lookup-export-uinfo-Some*)
apply *clarsimp*
apply(*frule td-set-field-lookupD*)
apply(*clarsimp simp*: *field-lvalue-def ac-simps*)
apply(*drule td-set-export-uinfoD*)
apply(*drule typ-slice-td-set*)
apply(*simp add*: *size-of-def typ-uinfo-t-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add*: *field-offset-def ac-simps field-offset-untyped-def typ-uinfo-t-def export-uinfo-def*)
apply(*erule* (*1*) *map-list-map-trans*)
done

lemma *size-of-tag*:
size-td (*typ-uinfo-t t*) = *size-of t*
by (*simp add*: *size-of-def typ-uinfo-t-def*)

lemma *size-of-neq-implies-typ-uinfo-t-neq* [*simp*]:
size-of TYPE('*a*::*c-type*) ≠ *size-of TYPE*('*b*::*c-type*) ⟹ *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
by (*metis size-of-tag*)

lemma *guard-mono-self*[*simp*]:
guard-mono g g
by (*fastforce dest*: *td-set-field-lookupD td-set-size-lte simp*: *guard-mono-def*)

lemma (**in** *c-type*) *field-lookup-offset-size*:
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*) ⟹
size-td t + *n* ≤ *size-td* (*typ-info-t TYPE*('*a*))
by (*fastforce dest*: *td-set-field-lookupD td-set-offset-size*)

lemma (**in** *mem-type*) *sub-h-t-valid'*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a ptr*);
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*),*n*);
guard-mono g (*g'*::'*b*::*mem-type ptr-guard*) ⟧ ⟹
d,*g'* ⊨$_t$ ((*Ptr* (*ptr-val p* + *of-nat n*))::'*b*::*mem-type ptr*)
apply(*clarsimp simp*: *h-t-valid-def c-type-class.h-t-valid-def guard-mono-def valid-footprint-def Let-def size-of-tag*)
apply(*drule-tac x*=*n*+*y* **in** *spec*, *erule impE*;
fastforce simp: *ac-simps size-of-def c-type-class.size-of-def typ-uinfo-t-def c-type-class.typ-uinfo-t-def*

dest: *td-set-field-lookupD typ-slice-td-set map-list-map-trans td-set-offset-size*)
done

lemma (**in** *mem-type*) *sub-h-t-valid*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a ptr*); (*typ-uinfo-t TYPE*('*b*),*n*) ∈ *td-set* (*typ-uinfo-t TYPE*('*a*)) *0* ⟧ ⟹
d,(λ*x*. *True*) ⊨$_t$ ((*Ptr* (*ptr-val p* + *of-nat n*))::'*b*::*mem-type ptr*)
apply(*subst* (*asm*) *td-set-field-lookup*)
apply(*simp add*: *typ-uinfo-t-def export-uinfo-def wf-desc-map*)
apply(*fastforce intro*: *sub-h-t-valid'* *simp*: *guard-mono-def*)
done

lemma (**in** *mem-type*) *h-t-valid-mono*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*,*n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*); *guard-mono g g'* ⟧ ⟹
d,*g* ⊨$_t$ (*p*::'*a ptr*) ⟶ *d*,*g'* ⊨$_t$ (*Ptr* (&(*p*→*f*))::'*b*::*mem-type ptr*)
apply(*clarsimp simp*: *field-lvalue-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*clarsimp simp*: *field-offset-def c-type-class.field-offset-def typ-uinfo-t-def c-type-class.typ-uinfo-t-def field-offset-untyped-def*)
apply(*rule sub-h-t-valid'*; *fast?*)
apply(*fastforce simp*: *c-type-class.typ-uinfo-t-def typ-uinfo-t-def*)
done

lemma (**in** *mem-type*) *s-valid-mono*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*,*n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*); *guard-mono g g'* ⟧ ⟹
d,*g* ⊨$_s$ (*p*::'*a ptr*) ⟶ *d*,*g'* ⊨$_s$ (*Ptr* (&(*p*→*f*))::'*b*::*mem-type ptr*)
unfolding *s-valid-def c-type-class.s-valid-def* **by** (*rule h-t-valid-mono*)

lemma *take-heap-list-le*:
k ≤ *n* ⟹ *take k* (*heap-list h n x*) = *heap-list h k x*
apply (*induct n arbitrary*: *k x*; *clarsimp*)
subgoal for *n k x* **by** (*cases k*; *simp*)
done

lemma *drop-heap-list-le*:
k ≤ *n* ⟹ *drop k* (*heap-list h n x*) = *heap-list h* (*n* − *k*) (*x* + *of-nat k*)
apply (*induct n arbitrary*: *k x*; *clarsimp*)
subgoal for *n k x* **by** (*cases k*; *simp add*: *ac-simps*)
done

lemma (**in** *mem-type*) *h-val-field-from-bytes'*:
⟦ *field-ti TYPE*('*a*) *f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*('*b*::{*mem-type*})) ⟧ ⟹
h-val h (*Ptr* &(*pa*→*f*) :: '*b ptr*) = *from-bytes* (*access-ti*$_0$ *t* (*h-val h pa*))
apply (*clarsimp simp*: *field-ti-def h-val-def c-type-class.h-val-def split*: *option.splits*)
apply (*frule field-lookup-export-uinfo-Some*)

apply (*frule-tac bs=heap-list h* (*size-of TYPE('a)*) (*ptr-val pa*) **in** *fi-fa-consistentD*)
apply *simp*
apply (*clarsimp simp*: *field-lvalue-def field-offset-def field-offset-untyped-def typ-uinfo-t-def field-names-def access-ti$_0$-def*)
apply (*subst drop-heap-list-le*)
apply(*simp add*: *size-of-def*)
apply(*drule td-set-field-lookupD*)
apply(*drule td-set-offset-size*)
apply *simp*
apply(*subst take-heap-list-le*)
apply(*simp add*: *size-of-def*)
apply(*drule td-set-field-lookupD*)
apply(*drule td-set-offset-size*)
apply *simp*
apply (*fold c-type-class.norm-bytes-def*)
apply (*subgoal-tac size-td t = size-of TYPE('b)*)
apply (*clarsimp simp*: *wf-type-class.norm*)
apply(*clarsimp simp*: *c-type-class.size-of-def*)
apply(*subst c-type-class.typ-uinfo-size* [*symmetric*])
apply(*unfold c-type-class.typ-uinfo-t-def*)
apply(*drule sym*)
apply *simp*
done

lemma (**in** *mem-type*) *h-val-field-from-bytes*:
⟦ *field-ti TYPE('a) f = Some t*;
export-uinfo t = export-uinfo (*typ-info-t TYPE('b::*{*mem-type*}*)*) ⟧ ⟹
h-val (*hrs-mem h*) (*Ptr &(pa→f) :: 'b ptr*) = *from-bytes* (*access-ti$_0$ t* (*h-val* (*hrs-mem h*) *pa*))
by (*simp add*: *h-val-field-from-bytes'*)

lemma (**in** *mem-type*) *lift-typ-heap-mono*:
⟦ *field-lookup* (*typ-info-t TYPE('a)*) *f 0 = Some* (*t*,*n*);
lift-typ-heap g s (*p::'a ptr*) = *Some v*;
export-uinfo t = typ-uinfo-t TYPE('b); *guard-mono g g'* ⟧ ⟹
lift-typ-heap g' s (*Ptr* (*&(p→f)*)*::'b::mem-type ptr*) = *Some* (*from-bytes* (*access-ti$_0$ t v*))
apply(*clarsimp simp*: *c-type-class.lift-typ-heap-if lift-typ-heap-if split*: *if-split-asm*)
apply(*rule conjI*; *clarsimp?*)
apply(*clarsimp simp*: *heap-list-s-def*)
apply(*frule field-lookup-export-uinfo-Some*)
apply(*frule-tac bs=heap-list* (*proj-h s*) (*size-of TYPE('a)*) (*ptr-val p*) **in** *fi-fa-consistentD*; *simp*)
apply(*simp add*: *c-type-class.field-lvalue-def field-lvalue-def c-type-class.field-offset-def field-offset-def field-offset-untyped-def typ-uinfo-t-def c-type-class.typ-uinfo-t-def field-names-def access-ti$_0$-def*)
apply(*subst drop-heap-list-le*)

apply(*simp add*: *size-of-def*)
apply(*drule td-set-field-lookupD*)
apply(*drule td-set-offset-size*)
apply *simp*
apply(*subst take-heap-list-le*)
apply(*simp add*: *size-of-def*)
apply(*drule td-set-field-lookupD*)
apply(*drule td-set-offset-size*)
apply *simp*
apply(*fold c-type-class.norm-bytes-def*)
apply(*subgoal-tac size-td t = size-of TYPE('b)*)
apply(*simp add*: *wf-type-class.norm*)
apply(*clarsimp simp*: *c-type-class.size-of-def size-of-def*)
apply(*subst c-type-class.typ-uinfo-size* [*symmetric*])
apply(*unfold c-type-class.typ-uinfo-t-def*)[*1*]
apply(*drule sym*)
apply *simp*
apply (*fastforce dest*: *s-valid-mono*)
done

lemma (**in** *mem-type*) *lift-t-mono*:
⟦ *field-lookup* (*typ-info-t TYPE('a)*) *f 0* = *Some* (*t*,*n*);
lift-t g s (*p*::'*a ptr*) = *Some v*;
export-uinfo t = *typ-uinfo-t TYPE('b)*; *guard-mono g g'* ⟧ ⟹
lift-t g' s (*Ptr* (&(*p*→*f*))::'*b*::*mem-type ptr*) = *Some* (*from-bytes* (*access-ti*$_0$ *t v*))
by (*clarsimp simp*: *lift-t-def c-type-class.lift-t-def lift-typ-heap-mono*)

lemma *align-td-wo-align-field-lookupD*:
field-lookup (*t*::('*a*,'*b*) *typ-desc*) *f m* = *Some* (*s*, *n*) ⟹ *align-td-wo-align s* ≤ *align-td-wo-align t*
by(*simp add*: *align-td-wo-align-field-lookup*)

lemma (**in** *c-type*) *align-td-wo-align-uinfo*:
align-td-wo-align (*typ-uinfo-t TYPE('a)*) = *align-td-wo-align* (*typ-info-t TYPE('a)*)
by (*clarsimp simp*: *typ-uinfo-t-def*)

lemma (**in** *c-type*) *align-td-uinfo*:
align-td (*typ-uinfo-t TYPE('a)*) = *align-td* (*typ-info-t TYPE('a)*)
by (*clarsimp simp*: *typ-uinfo-t-def export-uinfo-def*)

lemma *align-td-export-uinfo*: *align-td* (*export-uinfo t*) = *align-td t*
by (*clarsimp simp add*: *export-uinfo-def*)

lemma (**in** *mem-type*) *align-field-uinfo*:
align-field (*typ-uinfo-t TYPE('a)*) = *align-field* (*typ-info-t TYPE('a)*)
apply (*standard*)
apply (*simp add*: *align-field*)
apply (*clarsimp simp add*: *typ-uinfo-t-def align-field-def align-field*)

apply (*drule field-lookup-export-uinfo-Some-rev*)
apply (*clarsimp simp add*: *align-td-export-uinfo*)
done

lemma (**in** *mem-type*) *ptr-aligned-mono'*:
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*),*n*) ⟹
ptr-aligned (*p*::'*a ptr*) ⟶ *ptr-aligned* (*Ptr* (&(*p*→*f*))::'*b*::*mem-type ptr*)
apply(*clarsimp simp*: *ptr-aligned-def c-type-class.ptr-aligned-def*
align-of-def c-type-class.align-of-def
field-lvalue-def c-type-class.field-lvalue-def)
apply(*subgoal-tac align-field* (*typ-uinfo-t* (*TYPE*('*a*))))
apply(*subst* (*asm*) *align-field-def*)
apply(*drule-tac x*=*f* **in** *spec*)
apply(*drule-tac x*=*typ-uinfo-t TYPE*('*b*) **in** *spec*)
apply(*drule-tac x*=*n* **in** *spec*)
apply *clarsimp*
apply(*simp add*: *c-type-class.align-td-uinfo*)
apply(*clarsimp simp*: *field-offset-def field-offset-untyped-def*)
apply(*subst unat-word-ariths*)
apply(*rule dvd-mod*)
apply(*rule dvd-add*)
subgoal
apply (*drule field-lookup-uinfo-Some-rev*)
apply *clarsimp*
by (*metis ArraysMemInstance.align-td-export-uinfo c-type-class.typ-uinfo-t-def*
align-td-field-lookup-mem-type power-le-dvd)
subgoal
apply(*subst unat-of-nat*)
apply(*subst mod-less*)
apply(*frule td-set-field-lookupD*)
apply(*drule td-set-offset-size*)
apply(*subst len-of-addr-card*)
using *local.max-size local.size-of-def* **apply** *force*
apply *assumption*
done
subgoal
apply(*subst len-of-addr-card*)
apply(*subgoal-tac align-of TYPE*('*b*) *dvd addr-card*)
apply(*subst* (*asm*) *c-type-class.align-of-def*)
apply *simp*
apply *simp*
done
apply(*subst align-field-uinfo*)
apply(*rule align-field*)
done

lemma (**in** *mem-type*) *ptr-aligned-mono*:
guard-mono (*ptr-aligned*::'*a ptr-guard*) (*ptr-aligned*::'*b*::*mem-type ptr-guard*)
apply(*clarsimp simp*: *guard-mono-def*)

apply(*frule ptr-aligned-mono'*)
apply(*fastforce simp*: *field-lvalue-def field-offset-def field-offset-untyped-def*)
done

lemma (**in** *wf-type*) *wf-desc-typ-tag* [*simp*]:
wf-desc (*typ-uinfo-t TYPE*('*a*))
by (*simp add*: *typ-uinfo-t-def export-uinfo-def wf-desc-map*)

lemma (**in** *c-type*) *sft1'*:
sub-field-update-t (*f*#*fs*) *p* (*v*::'*a*) *s* =
(*let s'* = *sub-field-update-t fs p* (*v*::'*a*) *s in*
s'(*Ptr* &(*p*→*f*) ↦ *from-bytes* (*access-ti*$_0$ (*field-typ TYPE*('*a*) *f*) *v*))) |`
dom (*s*::'*b*::*c-type typ-heap*)
by (*rule sft1*)

lemma *size-map-td*:
size (*map-td f g t*) = *size t*
size (*map-td-struct f g st*) = *size st*
size-list (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) (*map-td-list f g ts*) = *size-list* (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) *ts*
size-dt-tuple size (λ-. *0*) (λ-. *0*) (*map-td-tuple f g x*) = *size-dt-tuple size* (λ-. *0*) (λ-. *0*) *x*
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *field-names-size'*:
field-names t s ≠ [] ⟶ *size s* ≤ *size* (*t*::('*a*,'*b*) *typ-info*)
field-names-struct st s ≠ [] ⟶ *size s* ≤ *size* (*st*::('*a field-desc*,'*b*) *typ-struct*)
field-names-list ts s ≠ [] ⟶ *size s* ≤ *size-list* (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) (*ts*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple list*)
field-names-tuple x s ≠ [] ⟶ *size s* ≤ *size-dt-tuple size* (λ-. *0*) (λ-. *0*) (*x*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple*)
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*auto simp*: *size-map-td size-char-def*)

lemma *field-names-size*:
f ∈ *set* (*field-names t s*) ⟹ *size s* ≤ *size t*
by (*cases field-names t s*; *simp add*: *field-names-size'*)

lemma *field-names-size-struct*:
f ∈ *set* (*field-names-struct st s*) ⟹ *size s* ≤ *size st*
by (*cases field-names-struct st s*; *simp add*: *field-names-size'*)

lemma *field-names-Some3*:
∀*f m s n*. *field-lookup* (*t*::('*a*,'*b*) *typ-info*) *f m* = *Some* (*s*,*n*) ⟶
f ∈ *set* (*field-names t* (*export-uinfo s*))
∀*f m s n*. *field-lookup-struct* (*st*::('*a field-desc*,'*b*) *typ-struct*) *f m* = *Some* (*s*,*n*) ⟶
f ∈ *set* (*field-names-struct st* (*export-uinfo s*))

$\forall f\ m\ s\ n.$ *field-lookup-list* (*ts*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple list*) *f m* = *Some* (*s*,*n*) ⟶
f ∈ *set* (*field-names-list ts* (*export-uinfo s*))
$\forall f\ m\ s\ n.$ *field-lookup-tuple* (*x*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple*) *f m* = *Some* (*s*,*n*) ⟶
f ∈ *set* (*field-names-tuple x* (*export-uinfo s*))
apply(*induct t* **and** *st* **and** *ts* **and** *x*)
apply *clarsimp*
apply((*erule allE*)+, *erule impE*, *fast*)
apply *simp*
apply(*drule field-names-size-struct*)
apply(*simp add*: *size-map-td*)
apply (*auto split*: *option.splits*)[*4*]
apply *clarsimp*
by (*metis image-eqI list.collapse*)

lemma *field-names-SomeD3*:
field-lookup (*t*::('*a*,'*b*) *typ-info*) *f m* = *Some* (*s*,*n*) ⟹ *f* ∈ *set* (*field-names t* (*export-uinfo s*))
by (*simp add*: *field-names-Some3*)

lemma *empty-not-in-field-names* [*simp*]:
[] ∉ *set* (*field-names-tuple x s*)
by (*cases x*, *auto*)

lemma *empty-not-in-field-names-list* [*simp*]:
[] ∉ *set* (*field-names-list ts s*)
by (*induct ts*, *auto*)

lemma *empty-not-in-field-names-struct* [*simp*]:
[] ∉ *set* (*field-names-struct st s*)
by (*cases st*, *auto*)

lemma *field-names-Some*:
$\forall m\ f.$ *f* ∈ *set* (*field-names* (*t*::('*a*,'*b*) *typ-info*) *s*) ⟶ (*field-lookup t f m* ≠ *None*)
$\forall m\ f.$ *f* ∈ *set* (*field-names-struct* (*st*::('*a field-desc*,'*b*) *typ-struct*) *s*) ⟶ (*field-lookup-struct st f m* ≠ *None*)
$\forall m\ f.$ *f* ∈ *set* (*field-names-list* (*ts*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple list*) *s*) ⟶ (*field-lookup-list ts f m* ≠ *None*)
$\forall m\ f.$ *f* ∈ *set* (*field-names-tuple* (*x*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple*) *s*) ⟶ (*field-lookup-tuple x f m* ≠ *None*)
proof(*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next

case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
apply *clarsimp*
apply (*safe*; *clarsimp?*)
subgoal for *m f*
apply(*drule spec* [**where** *x*=*m*])
apply(*drule spec* [**where** *x*=*f*])
apply *clarsimp*
done
apply(*clarsimp split*: *option.splits*)
done

next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** *force*
qed

lemma *dt-snd-field-names-list-simp* [*simp*]:
f ∉ *dt-snd* ‘ *set xs* ⟹ ¬ *f*#*fs* ∈ *set* (*field-names-list xs s*)
by (*induct xs*; *clarsimp*) (*auto simp*: *split-DTuple-all*)

lemma *field-names-Some2*:
∀ *m f*. *wf-desc t* ⟶ *f* ∈ *set* (*field-names* (*t*::(′*a*,′*b*) *typ-info*) *s*) ⟶
(∃ *n k*. *field-lookup t f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
∀ *m f*. *wf-desc-struct st* ⟶ *f* ∈ *set* (*field-names-struct* (*st*::(′*a field-desc*,′*b*) *typ-struct*) *s*) ⟶
(∃ *n k*. *field-lookup-struct st f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
∀ *m f*. *wf-desc-list ts* ⟶ *f* ∈ *set* (*field-names-list* (*ts*::((′*a*,′*b*) *typ-info*,*field-name*,′*b*) *dt-tuple list*) *s*) ⟶
(∃ *n k*. *field-lookup-list ts f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
∀ *m f*. *wf-desc-tuple x* ⟶ *f* ∈ *set* (*field-names-tuple* (*x*::((′*a*,′*b*) *typ-info*,*field-name*,′*b*) *dt-tuple*) *s*) ⟶
(∃ *n k*. *field-lookup-tuple x f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next

```
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply (clarsimp simp: export-uinfo-def)
    subgoal for m f
      apply(safe; clarsimp?)
       apply(drule spec [where x=m])
       apply(fastforce simp: split: option.split)
      apply (cases dt-tuple)
      apply(clarsimp )
      apply(cases f; fastforce)
      done
    done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    apply  (clarsimp simp: export-uinfo-def)
    subgoal for m f
      apply(cases f; fastforce)
      done
    done
qed

lemma field-names-SomeD2:
  ⟦ f ∈ set (field-names (t::('a,'b) typ-info) s); wf-desc t ⟧ ⟹
      (∃ n k. field-lookup t f m = Some (k,n) ∧ export-uinfo k = s)
  by (simp add: field-names-Some2)

lemma field-names-SomeD:
  f ∈ set (field-names (t::('a,'b) typ-info) s) ⟹ (field-lookup t f m ≠ None)
  by (simp add: field-names-Some)

lemma lift-t-sub-field-update' [rule-format]:
   ⟦ d,g' ⊨_t p; ¬ (TYPE('a) <_τ TYPE('b)) ⟧ ⟹ fs-consistent fs TYPE('a)
TYPE('b) ⟶
    (∀ K. K = UNIV − (((field-offset-footprint p (field-names (typ-info-t TYPE('a))
(typ-uinfo-t TYPE('b))))) − (field-offset-footprint p fs)) ⟶
      lift-t g (heap-update p (v::'a::mem-type) h,d) |` K =
          sub-field-update-t fs p v ((lift-t g (h,d))::'b::mem-type typ-heap) |` K)
  apply(induct fs)
   apply clarsimp
   apply(rule ext)
  subgoal for x
    apply(clarsimp simp: lift-t-if restrict-map-def)
    apply(erule (2) h-val-heap-same)
    apply(clarsimp simp: field-of-t-def field-offset-footprint-def field-of-def)
    apply(cases x)
```

apply(*clarsimp simp*: *field-lvalue-def td-set-field-lookup field-offset-def field-offset-untyped-def*)
subgoal for *xa f*
apply(*drule-tac x=f* **in** *spec*)
apply(*fastforce simp*: *typ-uinfo-t-def dest*: *field-lookup-export-uinfo-Some-rev field-names-SomeD3*)
done
done
subgoal for *a list*
apply *clarify*
apply(*clarsimp simp*: *fs-consistent-def*)
apply (*rule conjI*; *clarsimp*)
subgoal for *y*
apply(*rule ext*)
subgoal for *x*
apply(*cases x* ≠ *Ptr* &(*p*→*a*))
apply(*clarsimp simp*: *restrict-map-def*)
apply(*drule fun-cong* [**where** *x=x*])
apply *clarsimp*
apply(*fastforce simp*: *lift-t-if split*: *if-split-asm*)
apply(*clarsimp simp*: *lift-t-if*)
apply (*rule conjI*)
apply *clarsimp*
apply(*clarsimp simp*: *h-val-def heap-update-def field-lvalue-def*)
apply(*subst heap-list-update-list*; *simp?*)
apply(*simp add*: *size-of-def*)

apply(*subst typ-uinfo-size* [*symmetric*])
apply(*subst typ-uinfo-size* [*symmetric*])
apply(*drule field-names-SomeD2* [**where** *m=0*]; *clarsimp*)
apply(*frule td-set-field-lookupD* [**where** *m=0*])
apply(*clarsimp simp*: *field-offset-def field-offset-untyped-def typ-uinfo-t-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply *simp*
apply(*drule td-set-export-uinfoD*)
apply(*simp add*: *export-uinfo-def*)
apply(*drule td-set-offset-size*)
apply *fastforce*

apply(*drule field-names-SomeD2* [**where** *m=0*], *clarsimp+*)
subgoal for *n k*
apply(*frule fi-fa-consistentD* [**where** *bs=to-bytes v* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))])
apply *simp*
apply(*simp add*: *size-of-def*)
apply(*frule field-lookup-export-uinfo-Some*)
apply(*simp add*: *typ-uinfo-t-def*)
apply(*subgoal-tac size-td k = size-td* (*typ-info-t TYPE*('*b*)))
prefer *2*
apply(*simp flip*: *export-uinfo-size*)

apply *simp*
apply(*rule sym*)
apply(*fold norm-bytes-def*)
apply(*subst typ-uinfo-size* [*symmetric*])
apply(*drule sym* [**where** *t*=*Some* (*k*,*n*)])
apply(*simp only*: *typ-uinfo-t-def*)
apply(*simp*)
apply(*clarsimp simp*: *access-ti*$_0$*-def field-typ-def field-typ-untyped-def*)
apply(*drule sym* [**where** *s*=*Some* (*k*,*n*)])
apply *simp*
apply(*rule norm*)
apply(*simp add*: *size-of-def*)
apply(*subst typ-uinfo-size* [*symmetric*])+
apply(*drule td-set-field-lookupD*, *drule td-set-offset-size*)
apply(*fastforce simp*: *min-def split*: *if-split-asm*)
done
apply(*clarsimp split*: *if-split-asm*)
done
done
apply(*rule ccontr*, *clarsimp*)
apply(*erule notE*, *rule ext*)
subgoal for *x*
apply(*cases x* ≠ *Ptr* &(*p*→*a*))
apply(*clarsimp simp*: *restrict-map-def*)
apply(*drule fun-cong* [**where** *x*=*x*])
apply *clarsimp*
apply(*rule ccontr*, *clarsimp*)
apply(*erule impE* [**where** *P*=*x* ∈ *dom* (*lift-t g* (*h*, *d*))])
apply(*clarsimp simp*: *lift-t-if h-t-valid-def split*: *if-split-asm*)
apply *clarsimp*
apply(*clarsimp simp*: *restrict-map-def*)
apply(*rule conjI*, *clarsimp*)
apply(*clarsimp simp*: *lift-t-if*)
done
done
done

lemma *lift-t-sub-field-update*:
⟦ *d*,*g'* ⊨$_t$ *p*; ¬ (*TYPE*('*a*) <$_\tau$ *TYPE*('*b*))⟧ ⟹
lift-t g (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*) =
sub-field-update-t (*field-names* (*typ-info-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))
p v
((*lift-t g* (*h*,*d*))::'*b*::*mem-type typ-heap*)
apply(*drule lift-t-sub-field-update'* [**where** *fs*=(*field-names* (*typ-info-t TYPE*('*a*))
(*typ-uinfo-t TYPE*('*b*)))] , *assumption*+)
apply(*clarsimp simp*: *fs-consistent-def*)
apply *fast*
apply(*simp add*: *restrict-map-def*)
done

lemma *lift-t-field-ind*:
⟦ *d*,*g'* ⊨$_t$ (*p*::'*b*::*mem-type ptr*); *d*,*ga* ⊨$_t$ (*q*::'*b ptr*);
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *f 0* = *Some* (*a*,*ba*);
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *z 0* = *Some* (*c*,*da*) ;
size-td a = *size-of TYPE*('*a*); *size-td c* = *size-of TYPE*('*c*);
¬ *f* ≤ *z*; ¬ *z* ≤ *f* ⟧ ⟹
lift-t g (*heap-update* (*Ptr* (&(*p*→*f*))) (*v*::'*a*::*mem-type*) *h*,*d*) (*Ptr* (&(*q*→*z*)))
=
((*lift-t g* (*h*,*d*) (*Ptr* (&(*q*→*z*))))::'*c*::*c-type option*)
apply(*clarsimp simp*: *lift-t-if h-val-def heap-update-def*)
apply(*subgoal-tac* (*heap-list* (*heap-update-list* &(*p*→*f*) (*to-bytes v* (*heap-list h* (*size-of TYPE*('*a*)) &(*p*→*f*))) *h*)
(*size-of TYPE*('*c*)) &(*q*→*z*)) =
(*heap-list h* (*size-of TYPE*('*c*)) &(*q*→*z*)))
apply(*drule arg-cong* [**where** *f*=*from-bytes*])
apply *simp*
apply(*rule heap-list-update-disjoint-same*)
apply *simp*
apply(*simp add*: *field-lvalue-def field-offset-def field-offset-untyped-def*)
apply(*simp add*: *typ-uinfo-t-def field-lookup-export-uinfo-Some*)
apply(*frule field-lookup-export-uinfo-Some*[**where** *s*=*c*])
apply(*cases ptr-val p* = *ptr-val q*)
apply *clarsimp*
apply(*subst intvl-disj-offset*)
apply(*drule fa-fu-lookup-disj-interD*)
apply *fast*
apply(*simp add*: *disj-fn-def*)
apply *simp*
apply(*subst size-of-def* [*symmetric*, **where** *t*=*TYPE*('*b*)])
apply *simp*
apply *simp*
apply *clarsimp*
apply(*drule* (*1*) *h-t-valid-neq-disjoint*, *simp*)
apply(*rule peer-typ-not-field-of*; *clarsimp*)
apply(*subgoal-tac* {*ptr-val p* + *of-nat ba*..+*size-td a*} ⊆ {*ptr-val p*..+*size-of TYPE*('*b*)})
apply(*subgoal-tac* {*ptr-val q* + *of-nat da*..+*size-td c*} ⊆ {*ptr-val q*..+*size-of TYPE*('*b*)})
apply *fastforce*
apply(*rule intvl-sub-offset*)
apply(*simp add*: *size-of-def*)
apply(*drule td-set-field-lookupD*[**where** *k*=(*c*,*da*)])
apply(*drule td-set-offset-size*)
apply (*smt* (*verit*, *ccfv-threshold*) *add.commute add-le-cancel-right le-trans le-unat-uoi nat-le-linear*)

apply(*rule intvl-sub-offset*)
apply(*simp add*: *size-of-def*)
apply(*drule td-set-field-lookupD*)
apply(*drule td-set-offset-size*)
by (*smt* (*verit*, *ccfv-threshold*) *add.commute add-le-imp-le-right le-trans le-unat-uoi nat-le-linear*)

lemma (**in** *c-type*) *uvt1'*:
update-value-t (*f*#*fs*) *v* (*w*::'*b*) *x* = (*if x*=*field-offset TYPE*('*b*) *f then update-ti-t* (*field-typ TYPE*('*b*) *f*) (*to-bytes-p* (*v*::'*a*)) (*w*::'*b*::*c-type*) *else update-value-t fs v w x*)
by *simp*

lemma (**in** *c-type*) *field-typ-self* [*simp*]:
field-typ TYPE('*a*) [] = *typ-info-t TYPE*('*a*)
by (*simp add*: *field-typ-def field-typ-untyped-def*)

lemma (**in** *mem-type*) *field-of-t-less-size*:
field-of-t (*p*::'*a ptr*) (*x*::'*b*::*c-type ptr*) ⟹
unat (*ptr-val p* − *ptr-val x*) < *size-of TYPE*('*b*)
apply(*simp add*: *field-of-t-def field-of-def*)
apply(*drule td-set-offset-size*)
apply(*subgoal-tac* 0 < *size-td* (*typ-info-t TYPE*('*a*)))
apply(*simp add*: *c-type-class.size-of-def*)
apply(*simp add*: *size-of-def* [*symmetric*, **where** *t*=*TYPE*('*a*)])
done

lemma *unat-minus*:
x ≠ 0 ⟹ *unat* (− (*x*::*addr*)) = *addr-card* − *unat x*
by (*metis word-bits-def word-bits-size len-of-addr-card unat-minus*)

lemma (**in** *mem-type*) *field-of-t-nmem*:
⟦ *field-of-t p q*; *ptr-val p* ≠ *ptr-val* (*q*::'*b*::*mem-type ptr*) ⟧ ⟹
ptr-val q ∉ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)}
supply *unsigned-of-nat* [*simp del*] *unsigned-of-int* [*simp del*]
apply(*clarsimp simp*: *field-of-t-def field-of-def intvl-def*)
apply(*drule td-set-offset-size*)
apply(*simp add*: *unat-minus size-of-def*)
apply(*subgoal-tac size-td* (*typ-info-t TYPE*('*b*)) < *addr-card*)
apply(*simp only*: *unat-simps*)
using *local.max-size local.size-of-fold* **apply** *auto*[*1*]
by (*metis c-type-class.size-of-def mem-type-class.max-size*)

lemma (**in** *mem-type*) *field-of-t-init-neq-disjoint*:
field-of-t p (*x*::'*b*::*mem-type ptr*) ⟹

```
    {ptr-val (p::'a ptr)..+size-of TYPE('a)} ∩
    {ptr-val x..+unat (ptr-val p − ptr-val x)} = {}
  apply(cases ptr-val p = ptr-val x, simp)
  apply(rule ccontr)
  apply(drule intvl-inter)
  apply(auto simp: field-of-t-nmem le-unat-uoi nat-less-le dest!: intvlD)
  done

lemma (in mem-type) field-of-t-final-neq-disjoint:
  field-of-t (p::'a ptr) (x::'b ptr) ⟹ {ptr-val p..+size-of TYPE('a)} ∩
      {ptr-val p + of-nat (size-of TYPE('a))..+size-of TYPE('b::mem-type) −
          (unat (ptr-val p − ptr-val x) + size-of TYPE('a))} = {}
  supply unsigned-of-nat [simp del] unsigned-of-int [simp del]
  apply(rule ccontr)
  apply(drule intvl-inter)
  apply(erule disjE)
   apply(subgoal-tac ptr-val p ∉ {ptr-val p + of-nat (size-of TYPE('a)) ..+
                                  size-of TYPE('b) − (unat (ptr-val p − ptr-val x) +
                                  size-of TYPE('a))})
    apply simp
   apply(rule intvl-offset-nmem)
    apply(rule intvl-self)
    apply(subst unat-of-nat)
    apply(subst mod-less)
     apply(subst len-of-addr-card)
     apply(rule max-size)
    apply(rule sz-nzero)
   apply(subst len-of-addr-card)
   apply(thin-tac x ∈ S for x S)
   apply(simp add: field-of-t-def field-of-def)
   apply(drule td-set-offset-size)
   apply(simp add: size-of-def)
  subgoal

  proof −
    have size-td (typ-info-t (TYPE('b)::'b itself)) ≤ addr-card
      by (metis c-type-class.size-of-def less-imp-le-nat mem-type-class.max-size)
    then show ?thesis
     by (smt (verit, ccfv-threshold) c-type-class.size-of-def diff-diff-left diff-le-mono2
diff-le-self
          le-diff-conv le-trans le-unat-uoi nat-le-linear)
  qed

  apply(drule intvlD, clarsimp)
  apply(subst (asm) word-unat.norm-eq-iff [symmetric])
  apply(subst (asm) mod-less)
   apply(subst len-of-addr-card)
   apply(rule max-size)
  apply(subst (asm) mod-less)
```

apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply(*rule max-size*)
apply *simp*
done

lemma (**in** *mem-type*) *h-val-super-update-bs*:
field-of-t p x $\Longrightarrow$ *h-val* (*heap-update p* (*v*::*'a*) *h*) (*x*::*'b*::*mem-type ptr*) =
from-bytes (*super-update-bs* (*to-bytes v* (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*))) (*heap-list h* (*size-of TYPE*(*'b*)) (*ptr-val x*)) (*unat* (*ptr-val p* − *ptr-val x*)))
apply(*simp add*: *h-val-def c-type-class.h-val-def*)
apply (*rule arg-cong* [**where** *f* = *c-type-class.from-bytes*::*byte list* ⇒ *'b*])
apply(*simp add*: *heap-update-def c-type-class.heap-update-def super-update-bs-def*)
apply(*subst heap-list-split* [*of unat* (*ptr-val p* − *ptr-val x*) *size-of TYPE*(*'b*)])
apply(*drule field-of-t-less-size*)
apply *simp*
apply *simp*
apply(*subst heap-list-update-disjoint-same*)
apply(*drule field-of-t-init-neq-disjoint*)
apply (*simp add*: *len*)
apply(*subst take-heap-list-le*)
apply(*drule field-of-t-less-size*)
apply *simp*
apply *simp*
apply(*subst heap-list-split* [*of size-of TYPE*(*'a*)
size-of TYPE(*'b*) − *unat* (*ptr-val p* − *ptr-val x*)])
apply(*frule field-of-t-less-size*)
apply(*simp add*: *field-of-t-def field-of-def*)
apply(*drule td-set-offset-size*)
apply(*simp add*: *size-of-def c-type-class.size-of-def*)
apply *clarsimp*
apply (*rule conjI*)
apply(*simp add*: *heap-list-update-to-bytes*)
apply(*subst heap-list-update-disjoint-same*)
apply(*drule field-of-t-final-neq-disjoint*)
apply(*simp*)
apply(*subst drop-heap-list-le*)
apply(*simp add*: *field-of-t-def field-of-def*)
apply(*drule td-set-offset-size*)
apply(*simp add*: *size-of-def c-type-class.size-of-def*)
apply *simp*
done

lemma (**in** *mem-type*) *h-val-super-update-bs-padding*:
field-of-t p x $\Longrightarrow$ *length bs* = *size-of TYPE*(*'a*) $\Longrightarrow$ *h-val* (*heap-update-padding p* (*v*::*'a*) *bs h*) (*x*::*'b*::*mem-type ptr*) =
from-bytes (*super-update-bs* (*to-bytes v bs*) (*heap-list h* (*size-of TYPE*(*'b*)) (*ptr-val x*)) (*unat* (*ptr-val p* − *ptr-val x*)))
apply(*simp add*: *h-val-def c-type-class.h-val-def*)

apply (*rule arg-cong* [**where** *f* = *c-type-class.from-bytes*::*byte list* ⇒ *'b*])
apply(*simp add*: *heap-update-padding-def c-type-class.heap-update-padding-def super-update-bs-def*)
apply(*subst heap-list-split* [*of unat* (*ptr-val p* − *ptr-val x*) *size-of TYPE*(*'b*)])
apply(*drule field-of-t-less-size*)
apply *simp*
apply *simp*

apply(*subst heap-list-update-disjoint-same*)
apply(*drule field-of-t-init-neq-disjoint*)
apply (*simp add*: *len*)
apply(*subst take-heap-list-le*)
apply(*drule field-of-t-less-size*)
apply *simp*
apply *simp*
apply(*subst heap-list-split* [*of size-of TYPE*(*'a*)
size-of TYPE(*'b*) − *unat* (*ptr-val p* − *ptr-val x*)])
apply(*frule field-of-t-less-size*)
apply(*simp add*: *field-of-t-def field-of-def*)
apply(*drule td-set-offset-size*)
apply(*simp add*: *size-of-def c-type-class.size-of-def*)
apply *clarsimp*
apply (*rule conjI*)
apply(*simp add*: *heap-list-update-to-bytes-padding*)
apply(*subst heap-list-update-disjoint-same*)
apply(*drule field-of-t-final-neq-disjoint*)
apply(*simp*)
apply(*subst drop-heap-list-le*)
apply(*simp add*: *field-of-t-def field-of-def*)
apply(*drule td-set-offset-size*)
apply(*simp add*: *size-of-def c-type-class.size-of-def*)
apply *simp*
done

lemma (**in** *c-type*) *update-field-update'*:
n ∈ (λ*f*. *field-offset TYPE*(*'b*) *f*) ' *set fs* ⟹
(∃*f*. *update-value-t fs* (*v*::*'a*) (*v'*::*'b*::*c-type*) *n* =
field-update (*field-desc* (*field-typ TYPE*(*'b*) *f*)) (*to-bytes-p v*) *v'* ∧ *f* ∈ *set fs* ∧ *n* = *field-offset TYPE*(*'b*) *f*)
by (*induct fs*) *auto*

lemma (**in** *c-type*) *update-field-update*:
field-of-t (*p*::*'a ptr*) (*x*::*'b ptr*) ⟹
∃*f*. *update-value-t* (*field-names* (*typ-info-t TYPE*(*'b*)) (*typ-uinfo-t TYPE*(*'a*))) (*v*::*'a*)
(*v'*::*'b*::*mem-type*) (*unat* (*ptr-val p* − *ptr-val x*)) =
field-update (*field-desc* (*field-typ TYPE*(*'b*) *f*)) (*to-bytes-p v*) *v'* ∧
f ∈ *set* (*field-names* (*typ-info-t TYPE*(*'b*)) (*typ-uinfo-t TYPE*(*'a*))) ∧
unat (*ptr-val p* − *ptr-val x*) = *field-offset TYPE*(*'b*) *f*

apply(*rule update-field-update'*)
apply(*clarsimp simp*: *image-def field-offset-def c-type-class.field-offset-def field-of-t-def field-of-def field-offset-untyped-def*)
apply(*simp add*: *td-set-field-lookup*)
apply *clarsimp*
subgoal for *f*
apply(*simp add*: *typ-uinfo-t-def c-type-class.typ-uinfo-t-def*)
apply(*rule bexI* [**where** *x=f*], *simp*)
apply(*drule field-lookup-export-uinfo-Some-rev*)
apply *clarsimp*
apply(*drule field-names-SomeD3*)
apply *clarsimp*
done
done

lemma *length-fa-ti*:
⟦ *wf-fd s*; *length bs* = *size-td s* ⟧ ⟹ *length* (*access-ti s w bs*) = *size-td s*
apply(*drule wf-fd-consD*)
apply(*clarsimp simp*: *fd-cons-def fd-cons-length-def fd-cons-desc-def*)
done

lemma *fa-fu-v*:
⟦ *wf-fd s*; *length bs* = *size-td s*; *length bs'* = *size-td s* ⟧ ⟹
access-ti s (*update-ti-t s bs v*) *bs'* = *access-ti s* (*update-ti-t s bs w*) *bs'*
apply(*drule wf-fd-consD*)
apply(*clarsimp simp*: *fd-cons-def fd-cons-access-update-def fd-cons-desc-def*)
done

lemma *fu-fa*:
⟦ *wf-fd s*; *length bs* = *size-td s* ⟧ ⟹
update-ti-t s (*access-ti s v bs*) *v* = *v*
apply(*drule wf-fd-consD*)
apply(*clarsimp simp*: *fd-cons-def fd-cons-update-access-def fd-cons-desc-def*)
done

lemma *fu-fu*:
⟦ *wf-fd s*; *length bs* = *length bs'* ⟧ ⟹
update-ti-t s bs (*update-ti-t s bs' v*) = *update-ti-t s bs v*
apply(*drule wf-fd-consD*)
apply(*clarsimp simp*: *fd-cons-def fd-cons-double-update-def fd-cons-desc-def*)
done

lemma *fu-fa'-rpbs*:
⟦ *export-uinfo s* = *export-uinfo t*; *length bs* = *size-td s*; *wf-fd s*; *wf-fd t* ⟧ ⟹
update-ti-t s (*access-ti t v bs*) *w* = *update-ti-t s* (*access-ti*$_0$ *t v*) *w*
apply(*clarsimp simp*: *access-ti*$_0$*-def*)
apply(*subgoal-tac size-td s* = *size-td t*)
apply(*frule-tac bs=access-ti t v bs* **in** *wf-fd-norm-tuD*[**where** *t=t*])

apply(*subst length-fa-ti*; *simp*)
apply(*frule-tac bs=access-ti t v bs* **in** *wf-fd-norm-tuD*)
apply(*subst length-fa-ti*; *simp*)
apply(*clarsimp simp*: *access-ti*$_0$*-def*)
apply(*thin-tac norm-tu X Y = Z* **for** *X Y Z*)+
apply(*subst* (*asm*) *fa-fu-v* [**where** *w=v*]; *simp?*)
apply(*simp add*: *length-fa-ti*)
apply(*subst* (*asm*) *fa-fu-v* [**where** *w=w*]; *simp?*)
apply(*simp add*: *length-fa-ti*; *simp*)
apply(*subst* (*asm*) *fu-fa*; (*solves* ‹*simp*›)*?*)
apply(*drule-tac f=update-ti-t s* **in** *arg-cong*)
apply(*drule-tac x=update-ti-t s* (*access-ti t v bs*) *w* **in** *fun-cong*)
apply(*subst* (*asm*) *fu-fa*; (*solves* ‹*simp*›)*?*)
apply(*fastforce simp*: *fu-fu length-fa-ti*)
apply(*drule-tac f=size-td* **in** *arg-cong*)
apply(*simp add*: *export-uinfo-def*)
done

lemma *lift-t-super-field-update*:
⟦ *d,g′* ⊨$_t$ *p*; *TYPE*(*'a*) ≤$_\tau$ *TYPE*(*'b*) ⟧ ⟹
lift-t g (*heap-update p* (*v*::*'a*::*mem-type*) *h,d*) =
super-field-update-t p v ((*lift-t g* (*h,d*))::*'b*::*mem-type typ-heap*)
apply(*rule ext*)
apply(*clarsimp simp*: *super-field-update-t-def split*: *option.splits*)
apply(*rule conjI*; *clarsimp*)
apply(*simp add*: *lift-t-if split*: *if-split-asm*)
apply(*rule conjI*; *clarsimp*)
apply(*simp add*: *lift-t-if h-val-super-update-bs split*: *if-split-asm*)
apply(*drule sym*)
apply(*rename-tac x1 x2*)
apply(*drule-tac v=v* **and** *v′=x2* **in** *update-field-update*)
apply(*clarsimp simp*: *h-val-def*)
apply(*frule-tac m=0* **in** *field-names-SomeD*)
apply *clarsimp*
apply(*subgoal-tac export-uinfo a = typ-uinfo-t TYPE*(*'a*))
prefer *2*
apply(*drule-tac m=0* **in** *field-names-SomeD2*; *clarsimp*)
apply(*simp add*: *from-bytes-def*)
apply(*frule-tac bs=heap-list h* (*size-of TYPE*(*'b*)) (*ptr-val x1*) **and**
v=to-bytes v (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*)) **and**
w=undefined **in** *fi-fu-consistentD*)
apply *simp*
apply(*simp add*: *size-of-def*)
apply(*clarsimp simp*: *size-of-def sub-typ-def typ-tag-le-def simp flip*: *export-uinfo-size*)
apply(*simp add*: *field-offset-def field-offset-untyped-def typ-uinfo-t-def*)
apply(*frule field-lookup-export-uinfo-Some*)
apply(*simp add*: *to-bytes-def to-bytes-p-def*)
apply(*subst fu-fa′-rpbs*; *simp?*)
apply(*simp add*: *size-of-def flip*: *export-uinfo-size*)

apply(*fastforce intro*: *wf-fd-field-lookupD*)
apply(*unfold access-ti$_0$-def*)[*1*]
apply(*simp flip*: *export-uinfo-size*)
apply(*simp add*: *size-of-def access-ti$_0$-def field-typ-def field-typ-untyped-def*)
apply(*frule lift-t-h-t-valid*)
apply(*simp add*: *lift-t-if*)
apply(*cases typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b)*)
apply(*subst h-val-heap-same-peer*; *assumption?*)
apply(*clarsimp simp*: *field-of-t-def*)
apply(*simp add*: *peer-typ-def*)
apply(*drule* (*1*) *h-t-valid-neq-disjoint*)
apply(*simp add*: *sub-typ-proper-def*)
apply(*rule order-less-not-sym*)
apply(*simp add*: *sub-typ-def*)
apply *assumption*
apply(*simp add*: *h-val-def heap-update-def heap-list-update-disjoint-same*)
done

lemma *lift-t-super-field-update-padding*:
⟦ *d,g' ⊨$_t$ p*; *TYPE('a) ≤$_\tau$ TYPE('b)*; *length bs = size-of TYPE('a)* ⟧ ⟹
lift-t g (*heap-update-padding p* (*v*::*'a*::*mem-type*) *bs h,d*) =
super-field-update-t p v ((*lift-t g* (*h,d*))::*'b*::*mem-type typ-heap*)
apply(*rule ext*)
apply(*clarsimp simp*: *super-field-update-t-def split*: *option.splits*)
apply(*rule conjI*; *clarsimp*)
apply(*simp add*: *lift-t-if split*: *if-split-asm*)
apply(*rule conjI*; *clarsimp*)
apply(*simp add*: *lift-t-if h-val-super-update-bs-padding split*: *if-split-asm*)
apply(*drule sym*)
apply(*rename-tac x1 x2*)
apply(*drule-tac v=v* **and** *v'=x2* **in** *update-field-update*)
apply(*clarsimp simp*: *h-val-def*)
apply(*frule-tac m=0* **in** *field-names-SomeD*)
apply *clarsimp*
apply(*subgoal-tac export-uinfo a = typ-uinfo-t TYPE('a)*)
prefer *2*
apply(*drule-tac m=0* **in** *field-names-SomeD2*; *clarsimp*)
apply(*simp add*: *from-bytes-def*)
apply(*frule-tac bs=heap-list h* (*size-of TYPE('b)*) (*ptr-val x1*) **and**
v=to-bytes v bs **and**
w=undefined **in** *fi-fu-consistentD*)
apply *simp*
apply(*simp add*: *size-of-def*)
apply(*clarsimp simp*: *size-of-def sub-typ-def typ-tag-le-def simp flip*: *export-uinfo-size*)
apply(*simp add*: *field-offset-def field-offset-untyped-def typ-uinfo-t-def*)
apply(*frule field-lookup-export-uinfo-Some*)
apply(*simp add*: *to-bytes-def to-bytes-p-def*)
apply(*subst fu-fa'-rpbs*; *simp?*)
apply(*simp add*: *size-of-def flip*: *export-uinfo-size*)

apply(*fastforce intro*: *wf-fd-field-lookupD*)
apply(*unfold access-ti$_0$-def*)[*1*]
apply(*simp flip*: *export-uinfo-size*)
apply(*simp add*: *size-of-def access-ti$_0$-def field-typ-def field-typ-untyped-def*)
apply(*frule lift-t-h-t-valid*)
apply(*simp add*: *lift-t-if*)
apply(*cases typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b)*)
apply(*subst h-val-heap-same-peer-padding*; *assumption?*)
apply(*clarsimp simp*: *field-of-t-def*)
apply(*simp add*: *peer-typ-def*)
apply(*drule* (*1*) *h-t-valid-neq-disjoint*)
apply(*simp add*: *sub-typ-proper-def*)
apply(*rule order-less-not-sym*)
apply(*simp add*: *sub-typ-def*)
apply *assumption*
apply(*simp add*: *h-val-def heap-update-padding-def heap-list-update-disjoint-same*)
done

lemma *field-names-same*:
k = export-uinfo ti ⟹ field-names ti k = [[]]
by (*cases ti*, *clarsimp*)

lemma *lift-t-heap-update*:
d,g ⊨$_t$ p ⟹ lift-t g (*heap-update p v h,d*) =
((*lift-t g* (*h,d*)) (*p ↦* (*v*::*'a*::*mem-type*)))
apply(*subst lift-t-sub-field-update*)
apply *fast*
apply(*simp add*: *sub-typ-proper-def*)
apply(*simp add*: *typ-uinfo-t-def Let-def*)
apply(*subgoal-tac access-ti$_0$* (*typ-info-t TYPE('a)*) = *to-bytes-p*)
apply(*simp add*: *field-names-same*)
apply(*clarsimp simp*: *Let-def to-bytes-p-def*)
apply(*rule conjI*, *clarsimp*)
apply(*rule ext*, *clarsimp simp*: *restrict-self-UNIV*)
apply *clarsimp*
apply(*clarsimp simp*: *h-t-valid-def lift-t-if*)
apply(*rule ext*)
apply(*simp add*: *to-bytes-p-def to-bytes-def size-of-def access-ti$_0$-def*)
done

lemma *field-names-disj*:
typ-uinfo-t TYPE('a::*c-type*) *⊥$_t$ typ-uinfo-t TYPE('b*::*mem-type*) ⟹*
field-names (*typ-info-t TYPE('b*)) (*typ-uinfo-t TYPE('a*)) = []
apply(*rule ccontr*)
apply(*subgoal-tac* (∃ *k*. *k* ∈ *set* (*field-names* (*typ-info-t TYPE('b*)) (*typ-uinfo-t TYPE('a*)))))
apply *clarsimp*
apply(*frule field-names-SomeD2* [**where** *m=0*], *clarsimp+*)
apply(*drule field-lookup-export-uinfo-Some*)

apply(*drule td-set-field-lookupD*)
apply(*fastforce simp*: *tag-disj-def typ-tag-le-def typ-uinfo-t-def*)
apply(*cases field-names* (*typ-info-t TYPE*('*b*)) (*typ-uinfo-t TYPE*('*a*)); *simp*)
apply *fast*
done

lemma *lift-t-heap-update-same*:
⟦ *d*,*g*′ ⊨$_t$ (*p*::'*b*::*mem-type ptr*); *typ-uinfo-t TYPE*('*a*) ⊥$_t$ *typ-uinfo-t TYPE*('*b*) ⟧
⟹
lift-t g (*heap-update p v h*,*d*) = (*lift-t g* (*h*,*d*)::'*a*::*mem-type typ-heap*)
apply(*subst lift-t-sub-field-update*, *simp*)
apply(*fastforce dest*: *order-less-imp-le simp*: *sub-typ-proper-def tag-disj-def*)
apply(*simp add*: *field-names-disj*)
done

lemma *lift-heap-update*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a ptr*); *d*,*g*′ ⊨$_t$ *q* ⟧ ⟹ *lift* (*heap-update p v h*) *q* =
((*lift h*)(*p* := (*v*::'*a*::*mem-type*))) *q*
by (*simp add*: *lift-def h-val-heap-update h-val-heap-same-peer*)

lemma (**in** *mem-type*) *lift-heap-update-p* [*simp*]:
lift (*heap-update p v h*) *p* = (*v*::'*a*)
by (*simp add*: *lift-def heap-update-def h-val-def heap-list-update-to-bytes*)

lemma *lift-heap-update-same*:
⟦ *ptr-val p* ≠ *ptr-val q*; *d*,*g* ⊨$_t$ (*p*::'*a*::*mem-type ptr*);
d,*g*′ ⊨$_t$ (*q*::'*b*::*mem-type ptr*); *peer-typ TYPE*('*a*) *TYPE*('*b*) ⟧ ⟹
lift (*heap-update p v h*) *q* = *lift h q*
by (*simp add*: *lift-def h-val-heap-same-peer*)

lemma *lift-heap-update-same-type*:
fixes *p*::'*a*::*mem-type ptr* **and** *q*::'*b*::*mem-type ptr*
assumes *valid*: *d*,*g* ⊨$_t$ *p* *d*,*g*′ ⊨$_t$ *q*
assumes *type*: *typ-uinfo-t TYPE*('*a*) ⊥$_t$ *typ-uinfo-t TYPE*('*b*)
shows *lift* (*heap-update p v h*) *q* = *lift h q*
proof −
from *valid type* **have** *ptr-val p* ≠ *ptr-val q*
apply(*clarsimp simp*: *h-t-valid-def*)
apply(*drule* (*1*) *valid-footprint-sub*)
apply(*clarsimp simp*: *tag-disj-def order-less-imp-le*)
apply(*fastforce intro*: *intvl-self*
simp: *field-of-def tag-disj-def typ-tag-le-def*
simp flip: *size-of-def* [**where** *t*=*TYPE*('*b*)])
done
thus *?thesis* **using** *valid type*
by (*fastforce elim*: *lift-heap-update-same peer-typI*)
qed

lemma *lift-heap-update-same-ptr-coerce*:

⟦ *ptr-val q* ≠ *ptr-val p*;
d,*g* ⊨$_t$ (*ptr-coerce* (*q*::*'b*::*mem-type ptr*)::*'c*::*mem-type ptr*);
d,*g'* ⊨$_t$ (*p*::*'a*::*mem-type ptr*);
size-of TYPE(*'b*) = *size-of TYPE*(*'c*); *peer-typ TYPE*(*'a*) *TYPE*(*'c*) ⟧ ⟹
lift (*heap-update q v h*) *p* = *lift h p*
apply(*clarsimp simp*: *lift-def h-val-def heap-update-def size-of-def*)
apply(*subst heap-list-update-disjoint-same*)
apply(*drule* (*1*) *h-t-valid-neq-disjoint*)
apply(*erule peer-typD*)
apply(*erule peer-typ-not-field-of*, *simp*)
apply(*simp add*: *size-of-def*)
apply *simp*
done

lemma *heap-footprintI*:
⟦ *valid-footprint d y t*; *x* ∈ {*y*..+*size-td t*} ⟧ ⟹ *x* ∈ *heap-footprint d t*
by (*force simp*: *heap-footprint-def*)

lemma *valid-heap-footprint*:
valid-footprint d x t ⟹ *x* ∈ *heap-footprint d t*
by (*force simp*: *heap-footprint-def*)

lemma *heap-valid-footprint*:
x ∈ *heap-footprint d t* ⟹ ∃ *y*. *valid-footprint d y t* ∧ *x* ∈ {*y*} ∪ {*y*..+*size-td t*}
by (*simp add*: *heap-footprint-def*)

lemma *heap-footprint-Some*:
x ∈ *heap-footprint d t* ⟹ *d x* ≠ (*False*,*Map.empty*)
by (*auto simp*: *heap-footprint-def intvl-def valid-footprint-def Let-def*)

lemma *dom-tll-empty* [*simp*]:
dom-tll p [] = {}
by (*clarsimp simp*: *dom-tll-def*)

lemma *dom-s-upd* [*simp*]:
dom-s (λ*b*. *if b* = *p then* (*True*,*a*) *else d b*) =
(*dom-s d* − {(*p*,*y*) | *y*. *True*}) ∪ {(*p*,*SIndexVal*)} ∪ {(*p*,*SIndexTyp n*) | *n*. *a n* ≠ *None*}
unfolding *dom-s-def* **by** (*cases d p*) *auto*

lemma *dom-tll-cons* [*simp*]:
dom-tll p (*x*#*xs*) = *dom-tll* (*p* + *1*) *xs* ∪ {(*p*,*SIndexVal*)} ∪ {(*p*,*SIndexTyp n*) | *n*. *x n* ≠ *None*}
unfolding *dom-tll-def*
apply (*rule equalityI*; *clarsimp*)
apply (*rule conjI*; *clarsimp*)

subgoal using *less-Suc-eq-0-disj* **by** *auto*[*1*]

subgoal using *less-Suc-eq-0-disj* **by** *force*
subgoal using *less-Suc-eq-0-disj Suc-mono of-nat-Suc*
by (*smt* (*verit*) *Groups.add-ac*(*2*) *Groups.add-ac*(*3*) *UnCI add.right-neutral mem-Collect-eq nth-Cons'* *nth-Cons-Suc semiring-1-class.of-nat-simps*(*1*) *subsetI*)
done

lemma *one-plus-x-zero*:
(*1*::*addr*) + *of-nat x* = *0* ⟹ *x* ≥ *addr-card* − *1*
apply(*simp add*: *addr-card*)
apply(*subst* (*asm*) *of-nat-1* [*symmetric*])
apply(*subst* (*asm*) *Abs-fnat-homs*)
apply(*subst* (*asm*) *Word.of-nat-0*)
apply(*erule exE*)
subgoal for *q*
apply(*cases q*; *simp*)
done
done

lemma *htd-update-list-dom* [*rule-format*, *simp*]:
length xs < *addr-card* ⟶ (∀ *p d*. *dom-s* (*htd-update-list p xs d*) = *dom-s d* ∪ *dom-tll p xs*)
by (*induct xs*; *clarsimp*) (*auto simp*: *dom-s-def*)

lemma *htd-update-list-same*:
shows ⋀*h p k*. ⟦ *0* < *k*; *k* ≤ *addr-card* − *length v* ⟧ ⟹
(*htd-update-list* (*p* + *of-nat k*) *v*) *h p* = *h p*
proof (*induct v*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons x xs*)
have *htd-update-list* (*p* + *of-nat k*) (*x* # *xs*) *h p* =
htd-update-list (*p* + *of-nat* (*k* + *1*)) *xs* (*h*(*p* + *of-nat k* := (*True*,*snd* (*h* (*p* + *of-nat k*)) ++ *x*))) *p*
by (*simp add*: *ac-simps*)
also have ... = (*h*(*p* + *of-nat k* := (*True*,*snd* (*h* (*p* + *of-nat k*)) ++ *x*))) *p*
proof −
from *Cons* **have** *k* + *1* ≤ *addr-card* − *length xs* **by** *simp*
with *Cons* **show** *?thesis* **by** (*simp only*:)
qed
also have ... = *h p*
proof −
from *Cons* **have** *of-nat k* ≠ (*0*::*addr*)

```
    by − (erule of-nat-neq-0, simp add: addr-card)
   thus ?thesis by clarsimp
 qed
 finally show ?case .
qed

lemma htd-update-list-index [rule-format]:
 ∀ p d. length xs < addr-card ⟶ x ∈ {p..+length xs} ⟶
       htd-update-list p xs d x = (True, snd (d x) ++ (xs ! (unat (x − p))))
 apply(induct xs; clarsimp)
 subgoal for x1 xs p d
  apply(cases p=x)
   apply simp
   apply(subst of-nat-1 [symmetric])
   apply(subst htd-update-list-same; simp)
  apply(drule spec [where x=p+1])
  apply(erule impE)
   apply(drule intvl-neq-start; simp)
  apply(subgoal-tac unat (x − p) = unat (x − (p + 1)) + 1)
   apply simp
  apply (clarsimp simp: unatSuc[symmetric] field-simps)
  done
 done

lemma (in c-type) typ-slices-length [simp]:
 length (typ-slices TYPE('a)) = size-of TYPE('a)
 by (simp add: typ-slices-def)

lemma (in c-type) typ-slices-index [simp]:
 n < size-of TYPE('a) ⟹ typ-slices TYPE('a) ! n =
   list-map (typ-slice-t (typ-uinfo-t TYPE('a)) n)
 by (simp add: typ-slices-def)

lemma (in c-type) empty-not-in-typ-slices [simp]:
 Map.empty ∉ set (typ-slices TYPE('a))
 by (auto simp: typ-slices-def dest: sym)

lemma (in c-type) dom-tll-s-footprint [simp]:
 dom-tll (ptr-val p) (typ-slices TYPE('a)) = s-footprint (p::'a ptr)
 by (fastforce simp: list-map-eq typ-slices-def s-footprint-def s-footprint-untyped-def
                     dom-tll-def size-of-def
               split: if-split-asm)

lemma (in mem-type) ptr-retyp-dom [simp]:
 dom-s (ptr-retyp (p::'a ptr) d) = dom-s d ∪ s-footprint p
 by (simp add: ptr-retyp-def)

lemma dom-s-empty-htd [simp]:
 dom-s empty-htd = {}
```

by (*clarsimp simp*: *empty-htd-def dom-s-def*)

lemma *dom-s-nempty*:
d x ≠ (*False*, *Map.empty*) ⟹ ∃ *k*. (*x*,*k*) ∈ *dom-s d*
apply(*clarsimp simp*: *dom-s-def*)
apply(*cases d x*, *clarsimp*)
using *None-not-eq* **by** *fastforce*

lemma (**in** *mem-type*) *ptr-retyp-None*:
x ∉ {*ptr-val p*..+*size-of TYPE*('*a*)} ⟹
ptr-retyp (*p*::'*a ptr*) *empty-htd x* = (*False*,*Map.empty*)
using *ptr-retyp-dom* [*of p empty-htd*]
by (*fastforce dest*: *dom-s-nempty s-footprintD*)

lemma (**in** *mem-type*) *ptr-retyp-footprint*:
x ∈ {*ptr-val p*..+*size-of TYPE*('*a*)} ⟹
ptr-retyp (*p*::'*a ptr*) *d x* =
(*True*,*snd* (*d x*) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*x* − *ptr-val p*))))
apply(*clarsimp simp*: *ptr-retyp-def*)
apply(*subst htd-update-list-index*; *simp*)
apply(*subst typ-slices-index*; *simp?*)
apply(*drule intvlD*, *clarsimp*)
by (*metis le-unat-uoi linorder-not-less nat-less-le*)

lemma (**in** *mem-type*) *ptr-retyp-Some*:
ptr-retyp (*p*::'*a ptr*) *d* (*ptr-val p*) =
(*True*,*snd* (*d* (*ptr-val p*)) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *0*))
by (*simp add*: *ptr-retyp-footprint*)

lemma (**in** *mem-type*) *ptr-retyp-Some2*:
x ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} ⟹ *ptr-retyp p d x* ≠ (*False*,*Map.empty*)
by (*auto simp*: *ptr-retyp-Some ptr-retyp-footprint dest*: *intvl-neq-start*)

lemma *snd-empty-htd* [*simp*]:
snd (*empty-htd x*) = *Map.empty*
by (*auto simp*: *empty-htd-def*)

lemma (**in** *mem-type*) *ptr-retyp-d-empty*:
x ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} ⟹
(∀ *d*. *fst* (*ptr-retyp p d x*)) ∧
snd (*ptr-retyp p d x*) = *snd* (*d x*) ++ *snd* (*ptr-retyp p empty-htd x*)
by (*auto simp*: *ptr-retyp-Some ptr-retyp-footprint dest*: *intvl-neq-start*)

lemma *unat-minus-abs*:
x ≠ *y* ⟹ *unat* ((*x*::*addr*) − *y*) = *addr-card* − *unat* (*y* − *x*)
by (*clarsimp simp*: *unat-sub-if-size*)
(*metis* (*no-types*) *Nat.diff-cancel Nat.diff-diff-right add.commute len-of-addr-card*
nat-le-linear not-le trans-le-add1 unat-lt2p wsst-TYs(*3*))

lemma (**in** *mem-type*) *ptr-retyp-d*:
x ∉ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} ⟹
ptr-retyp p d x = *d x*
apply(*simp add*: *ptr-retyp-def*)
apply(*subgoal-tac* ∃*k*. *ptr-val p* = *x* + *of-nat k* ∧ 0 < *k* ∧ *k* ≤ *addr-card* − *size-of TYPE*('*a*))
apply *clarsimp*
apply(*subst htd-update-list-same*; *simp*)
apply(*rule exI* [**where** *x*=*unat* (*ptr-val p* − *x*)])
apply *clarsimp*
apply(*cases ptr-val p* = *x*)
apply *simp*
apply(*erule notE*)
apply(*rule intvl-self*)
apply *simp*
apply(*rule conjI*)
apply(*subst unat-gt-0*)
apply *clarsimp*
apply(*rule ccontr*)
apply(*erule notE*)
apply(*clarsimp simp*: *intvl-def*)
apply(*rule exI* [**where** *x*=*unat* (*x* − *ptr-val p*)])
apply *simp*
apply(*subst* (*asm*) *unat-minus-abs*; *simp*)
done

lemma (**in** *mem-type*) *ptr-retyp-valid-footprint-disjoint*:
⟦ *valid-footprint d p s*; {*p*..+*size-td s*} ∩ {*ptr-val q*..+*size-of TYPE*('*a*)} = {} ⟧
⟹ *valid-footprint* (*ptr-retyp* (*q*::'*a ptr*) *d*) *p s*
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply((*subst ptr-retyp-d*; *clarsimp*), *fastforce intro*: *intvlI*)+
done

lemma *ptr-retyp-disjoint*:
⟦ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*); {*ptr-val p*..+*size-of TYPE*('*a*)} ∩
{*ptr-val q*..+*size-of TYPE*('*b*)} = {} ⟧ ⟹
ptr-retyp (*p*::'*a*::*mem-type ptr*) *d*,*g* ⊨$_t$ *q*
apply(*clarsimp simp*: *h-t-valid-def*)
apply(*erule ptr-retyp-valid-footprint-disjoint*)
apply(*fastforce simp*: *size-of-def*)
done

lemma *ptr-retyp-d-fst*:
(*x*,*SIndexVal*) ∉ *s-footprint* (*p*::'*a*::*mem-type ptr*) ⟹ *fst* (*ptr-retyp p d x*) = *fst* (*d x*)
apply(*simp add*: *ptr-retyp-def*)
apply(*subgoal-tac* ∃*k*. *ptr-val p* = *x* + *of-nat k* ∧ 0 < *k* ∧ *k* ≤ *addr-card* − *size-of TYPE*('*a*))

apply(*fastforce simp*: *htd-update-list-same*)
apply(*rule exI* [**where** *x*=*unat* (*ptr-val p* − *x*)])
apply *clarsimp*
apply(*cases ptr-val p* = *x*)
apply(*fastforce dest*: *sym*)
apply(*rule conjI*)
apply(*subst unat-gt-0*, *clarsimp*)
apply(*rule ccontr*)
apply(*erule notE*)
apply(*subst* (*asm*) *unat-minus-abs*, *simp*)
apply(*clarsimp simp*: *s-footprint-def s-footprint-untyped-def*)
apply(*rule exI* [**where** *x*=*unat* (*x* − *ptr-val p*)])
apply(*simp add*: *size-of-def*)
done

lemma (**in** *mem-type*) *ptr-retyp-d-eq-fst*:
fst (*ptr-retyp p d x*) =
(*if x* ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} *then True else fst* (*d x*))
by (*auto dest*!: *ptr-retyp-d-empty*[**where** *d*=*d*] *ptr-retyp-d*[**where** *d*=*d*])

lemma (**in** *mem-type*) *ptr-retyp-d-eq-snd*:
snd (*ptr-retyp p d x*) =
(*if x* ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)}
then snd (*d x*) ++ *snd* (*ptr-retyp p empty-htd x*)
else snd (*d x*))
by (*auto dest*!: *ptr-retyp-d-empty*[**where** *d*=*d*] *ptr-retyp-d*[**where** *d*=*d*])

lemma *lift-state-ptr-retyp-d-empty*:
x ∈ {*ptr-val* (*p*::'*a*::*mem-type ptr*)..+*size-of TYPE*('*a*)} ⟹
lift-state (*h*,*ptr-retyp p d*) (*x*,*k*) =
(*lift-state* (*h*,*d*) ++ *lift-state* (*h*,*ptr-retyp p empty-htd*)) (*x*,*k*)
apply(*clarsimp simp*: *lift-state-def map-add-def split*: *s-heap-index.splits*)
apply *safe*
apply(*subst ptr-retyp-d-empty*; *simp*)
apply(*subst ptr-retyp-d-empty*; *simp*)
apply(*subst* (*asm*) *ptr-retyp-d-empty*; *simp*)
apply(*subst ptr-retyp-d-empty*, *simp*)
apply(*auto split*: *option.splits*)
done

lemma *lift-state-ptr-retyp-d*:
x ∉ {*ptr-val* (*p*::'*a*::*mem-type ptr*)..+*size-of TYPE*('*a*)} ⟹
lift-state (*h*,*ptr-retyp p d*) (*x*,*k*) = *lift-state* (*h*,*d*) (*x*,*k*)
by (*clarsimp simp*: *lift-state-def ptr-retyp-d split*: *s-heap-index.splits*)

lemma (**in** *mem-type*) *ptr-retyp-valid-footprint*:
valid-footprint (*ptr-retyp p d*) (*ptr-val* (*p*::'*a ptr*)) (*typ-uinfo-t TYPE*('*a*))
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply(*subst size-of-def* [*symmetric*, **where** *t*=*TYPE*('*a*)])

apply *clarsimp*
apply(*subst ptr-retyp-footprint*)
apply(*rule intvlI*)
apply(*simp add*: *size-of-def*)
apply(*subst ptr-retyp-footprint*)
apply(*rule intvlI*)
apply(*simp add*: *size-of-def*)
apply *clarsimp*
by (*metis* (*mono-tags*) *id-apply len-of-addr-card less-trans map-le-map-add max-size of-nat-eq-id*
size-of-def take-bit-nat-eq-self unsigned-of-nat)

lemma (**in** *mem-type*) *ptr-retyp-h-t-valid*:
g p $\Longrightarrow$ *ptr-retyp p d*,*g* $\models_t$ (*p*::'*a ptr*)
by (*simp add*: *h-t-valid-def ptr-retyp-valid-footprint*)

lemma (**in** *mem-type*) *ptr-retyp-s-valid*:
g p $\Longrightarrow$ *lift-state* (*h*,*ptr-retyp p d*),*g* $\models_s$ (*p*::'*a ptr*)
by (*simp add*: *s-valid-def proj-d-lift-state ptr-retyp-h-t-valid*)

lemma (**in** *mem-type*) *lt-size-of-unat-simps*:
k < *size-of TYPE*('*a*) $\Longrightarrow$ *unat* ((*of-nat k*)::*addr*) < *size-of TYPE*('*a*)
by (*metis le-def le-unat-uoi less-trans*)

lemma (**in** *mem-type*) *ptr-retyp-h-t-valid-same*:
⟦ *d*,*g* $\models_t$ (*p*::'*a ptr*); *x* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)} ⟧ $\Longrightarrow$
snd (*ptr-retyp p d x*) $\subseteq_m$ *snd* (*d x*)
apply(*clarsimp simp*: *h-t-valid-def valid-footprint-def Let-def*)
apply(*subst ptr-retyp-footprint*)
apply *simp*
apply *clarsimp*
apply(*drule-tac x*=*unat* (*x* − *ptr-val p*) **in** *spec*)
apply *clarsimp*
apply(*erule impE*)
apply(*simp only*: *size-of-def* [*symmetric*, **where** *t*=*TYPE*('*a*)])
apply(*drule intvlD*, *clarsimp*)
using *lt-size-of-unat-simps* **apply** *blast*
apply(*fastforce intro*: *map-add-le-mapI*)
done

lemma (**in** *mem-type*) *ptr-retyp-ptr-safe* [*simp*]:
ptr-safe p (*ptr-retyp* (*p*::'*a ptr*) *d*)
by (*force intro*: *h-t-valid-ptr-safe ptr-retyp-h-t-valid*)

lemma (**in** *mem-type*) *lift-state-ptr-retyp-restrict*:
(*lift-state* (*h*, *ptr-retyp p d*) |` {(*x*,*k*). *x* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)}}) =
(*lift-state* (*h*,*d*) |` {(*x*,*k*). *x* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)}}) ++
lift-state (*h*, *ptr-retyp* (*p*::'*a ptr*) *empty-htd*) (**is** *?x* = *?y*)

proof (*rule ext*, *cases*)
fix *x*::*addr* × *s-heap-index*
assume *fst x* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)}
thus *?x x* = *?y x*
apply(*cases x*)
apply(*clarsimp simp*: *lift-state-def map-add-def split*: *s-heap-index.splits*)
apply(*safe*; *clarsimp*)
apply(*drule ptr-retyp-d-empty*, *fast*)
apply(*frule-tac d*=*d* **in** *ptr-retyp-d-empty*, *clarsimp simp*: *map-add-def*)
apply(*clarsimp simp*: *restrict-map-def split*: *option.splits*)
apply(*drule ptr-retyp-d-empty*, *fast*)
apply(*drule ptr-retyp-d-empty*, *fast*)
apply(*frule-tac d*=*d* **in** *ptr-retyp-d-empty*, *clarsimp simp*: *map-add-def*)
apply(*clarsimp simp*: *restrict-map-def split*: *option.splits*)
done
next
fix *x*::*addr* × *s-heap-index*
assume *fst x* ∉ {*ptr-val p*..+*size-of TYPE*('*a*)}
thus *?x x* = *?y x*
by (*cases x*) (*auto simp*: *lift-state-def ptr-retyp-None map-add-def split*: *s-heap-index.splits*)
qed

lemmas *typ-simps* = *lift-t-heap-update lift-t-heap-update-same lift-heap-update*
lift-t-h-t-valid h-t-valid-ptr-safe lift-t-ptr-safe lift-lift-t lift-t-lift
tag-disj-def typ-tag-le-def typ-uinfo-t-def

declare *field-desc-def* [*simp del*]

lemma *field-fd*:
field-fd (*t*::'*a*::*c-type itself*) *n* =
(*case field-lookup* (*typ-info-t TYPE*('*a*)) *n 0 of*
None ⇒ *field-desc* (*fst* (*the* (*None*::('*a xtyp-info* × *nat*) *option*)))
| *Some x* ⇒ *field-desc* (*fst x*))
by (*auto simp*: *field-fd-def field-typ-def field-typ-untyped-def split*: *option.splits*)

declare *field-desc-def* [*simp add*]

lemma (**in** *mem-type*) *super-field-update-lookup*:
assumes *field-lookup* (*typ-info-t TYPE*('*b*)) *f 0* = *Some* (*s*,*n*)
and *typ-uinfo-t TYPE*('*a*) = *export-uinfo s*
and *lift-t g h p* = *Some v'*
shows *super-field-update-t* (*Ptr* (&(*p*→*f*))) (*v*::'*a*) (*lift-t g h*::'*b*::*mem-type typ-heap*)
=
(*lift-t g h*)(*p* ↦ *field-update* (*field-desc s*) (*to-bytes-p v*) *v'*)
proof −
note *unsigned-of-nat* [*simp del*]
from *assms* **have** *size-of TYPE*('*b*) < *addr-card* **by** *simp*
with *assms* **have** [*simp*]: *unat* (*of-nat n* :: *addr-bitsize word*) = *n*

apply(*subst unat-of-nat*)
apply(*subst mod-less*)
apply(*drule td-set-field-lookupD*)+
apply(*drule td-set-offset-size*)+
apply(*subst len-of-addr-card*)
apply(*subst* (*asm*) *c-type-class.size-of-def* [*symmetric*, **where** *t*=*TYPE*('*b*)])+
apply *arith*
apply *simp*
done
from *assms* **show** *?thesis*
apply(*clarsimp simp*: *super-field-update-t-def c-type-class.super-field-update-t-def*)
apply(*rule ext*)
apply(*clarsimp simp*: *field-lvalue-def c-type-class.field-lvalue-def split*: *option.splits*)
apply(*safe*; *clarsimp?*)
apply(*frule-tac v*=*v* **and** *v'*=*v'* **in** *update-field-update*)
apply *clarsimp*
apply(*thin-tac P* = *update-ti-t x y z* **for** *P x y z*)
apply(*clarsimp simp*: *field-of-t-def c-type-class.field-of-t-def field-of-def c-type-class.typ-uinfo-t-def typ-uinfo-t-def*)
apply(*frule-tac m*=*0* **in** *field-names-SomeD2*; *clarsimp*)
apply(*simp add*: *field-typ-def c-type-class.field-typ-def field-typ-untyped-def*)
apply(*frule field-lookup-export-uinfo-Some*)
apply(*frule-tac s*=*k* **in** *field-lookup-export-uinfo-Some*)
apply *simp*

apply(*drule* (*1*) *field-lookup-inject*)
apply(*subst c-type-class.typ-uinfo-t-def* [*symmetric*, **where** *t*=*TYPE*('*b*)])
apply *simp*
apply *simp*
apply(*drule field-of-t-mem*)+
apply(*cases h*)
apply(*clarsimp simp*: *lift-t-if c-type-class.lift-t-if split*: *if-split-asm*)
apply(*drule* (*1*) *h-t-valid-neq-disjoint*)
apply *simp*
apply(*fastforce simp*: *unat-eq-zero field-of-t-def c-type-class.field-of-t-def field-of-def dest*!: *td-set-size-lte*)
apply *fast*
apply(*clarsimp simp*: *field-of-t-def c-type-class.field-of-t-def field-of-def td-set-field-lookup*)
apply(*fastforce simp*: *typ-uinfo-t-def c-type-class.typ-uinfo-t-def dest*: *field-lookup-export-uinfo-Some*)
apply(*clarsimp simp*: *field-of-t-def c-type-class.field-of-t-def field-of-def td-set-field-lookup*)
apply(*fastforce simp*: *typ-uinfo-t-def c-type-class.typ-uinfo-t-def dest*: *field-lookup-export-uinfo-Some*)
done
qed

lemmas *typ-rewrs* =

lift-lift-t
lift-t-heap-update
lift-t-heap-update-same
lift-t-heap-update [*OF lift-t-h-t-valid*]
lift-t-heap-update-same [*OF lift-t-h-t-valid*]
lift-lift-t [*OF lift-t-h-t-valid*]

end

theory *Separation-UMM*
imports *TypHeap*
begin

type-synonym ('*a*,'*b*) *map-assert* = ('*a* ⇀ '*b*) ⇒ *bool*
type-synonym *heap-assert* = (*addr* × *s-heap-index*,*s-heap-value*) *map-assert*

definition *sep-emp* :: ('*a*,'*b*) *map-assert* (‹□›) **where**
sep-emp ≡ (=) *Map.empty*

definition *sep-true* :: ('*a*,'*b*) *map-assert* **where**
sep-true ≡ λ*s*. *True*

definition *sep-false* :: ('*a*,'*b*) *map-assert* **where**
sep-false ≡ λ*s*. *False*

definition
sep-conj :: ('*a*,'*b*) *map-assert* ⇒ ('*a*,'*b*) *map-assert* ⇒ ('*a*,'*b*) *map-assert* (**infixr** ‹∧*› *35*)
where
P ∧* *Q* ≡ λ*s*. ∃ *s*$_0$ *s*$_1$. *s*$_0$ ⊥ *s*$_1$ ∧ *s* = *s*$_1$ ++ *s*$_0$ ∧ *P* *s*$_0$ ∧ *Q* *s*$_1$

definition
sep-impl :: ('*a*,'*b*) *map-assert* ⇒ ('*a*,'*b*) *map-assert* ⇒ ('*a*,'*b*) *map-assert* (**infixr** ‹⟶*› *25*)
where
x ⟶* *y* ≡ λ*s*. ∀ *s*'. *s* ⊥ *s*' ∧ *x* *s*' ⟶ *y* (*s* ++ *s*')

definition
singleton :: '*a*::*c-type ptr* ⇒ '*a* ⇒ *heap-mem* ⇒ *heap-typ-desc* ⇒ *heap-state*
where
singleton p v h d ≡ *lift-state* (*heap-update p v h*,*d*) |` *s-footprint p*

Like in Separation.thy, these arrows are defined using bsub and esub but have an *input* syntax abbreviation with just sub. Why? Because if sub is the only way, people write things like $p \mapsto^i_{(f\ x\ y)}\ v$ instead of $p \mapsto^i_{(f\ x\ y)}$

v. We preserve the sub syntax though, because esub and bsub are a pain to type.

definition
sep-map :: *'a::c-type ptr* ⇒ *'a ptr-guard* ⇒ *'a* ⇒ *heap-assert*
(‹(‹*open-block notation=‹mixfix sep-map›*›- ↦- -)› [*56*,*0*,*51*] *56*)
where
$p \mapsto_g v \equiv \lambda s.$ *lift-typ-heap g s p* = *Some v* ∧ *dom s* = *s-footprint p* ∧ *wf-heap-val s*

notation (*input*)
sep-map (‹(‹*open-block notation=‹mixfix sep-map›*›- ↦- -)› [*56*,*1000*,*51*] *56*)

definition
sep-map-any :: *'a ::c-type ptr* ⇒ *'a ptr-guard* ⇒ *heap-assert*
(‹(‹*open-block notation=‹mixfix sep-map-any›*›- ↦- −)› [*56*,*0*] *56*)
where
$p \mapsto_g - \equiv \lambda s.\ \exists v.\ (p \mapsto_g v)\ s$

notation (*input*)
sep-map-any (‹(‹*open-block notation=‹mixfix sep-map-any›*›- ↦- −)› [*56*,*0*] *56*)

definition
sep-map' :: *'a::c-type ptr* ⇒ *'a ptr-guard* ⇒ *'a* ⇒ *heap-assert*
(‹(‹*open-block notation=‹mixfix sep-map'›*›- ↪- -)› [*56*,*0*,*51*] *56*)
where
$p \hookrightarrow_g v \equiv (p \mapsto_g v) \wedge^* \mathit{sep\text{-}true}$

notation (*input*)
sep-map' (‹(‹*open-block notation=‹mixfix sep-map'›*›- ↪- -)› [*56*,*1000*,*51*] *56*)

definition
sep-map'-any :: *'a ::c-type ptr* ⇒ *'a ptr-guard* ⇒ *heap-assert*
(‹(‹*open-block notation=‹mixfix sep-map'-any›*›- ↪- −)› [*56*,*0*] *56*)
where
$p \hookrightarrow_g - \equiv \lambda s.\ \exists x.\ (p \hookrightarrow_g x)\ s$

notation (*input*)
sep-map'-any (‹(‹*open-block notation=‹mixfix sep-map'-any›*›- ↪- −)› [*56*,*0*] *56*)

syntax
-sep-assert :: *bool* ⇒ *heap-state* ⇒ *bool*
(‹(‹*open-block notation=‹mixfix assertion›*›'(-')^{sep})› [*0*] *100*)

—-

lemma *sep-empD*:
□ *s* ⟹ *s* = *Map.empty*
by (*simp add*: *sep-emp-def*)

lemma *sep-emp-empty* [*simp*]:

□ *Map.empty*
by (*simp add*: *sep-emp-def*)

lemma *sep-true* [*simp*]:
sep-true s
by (*simp add*: *sep-true-def*)

lemma *sep-false* [*simp*]:
¬ *sep-false s*
by (*simp add*: *sep-false-def*)

declare *sep-false-def* [*symmetric*, *simp add*]

lemma *singleton-dom'*:
dom (*singleton p* (*v*::'*a*::*mem-type*) *h d*) = *dom* (*lift-state* (*h*,*d*)) ∩ *s-footprint p*
by (*auto simp*: *singleton-def lift-state-def*
split: *if-split-asm s-heap-index.splits*)

lemma *lift-state-dom*:
d,*g* ⊨$_t$ *p* ⟹ *s-footprint* (*p*::'*a*::*mem-type ptr*) ⊆ *dom* (*lift-state* (*h*,*d*))
supply *unsigned-of-nat* [*simp del*]
apply(*clarsimp simp*: *h-t-valid-def valid-footprint-def Let-def*)
apply(*clarsimp simp*: *lift-state-def split*: *s-heap-index.splits option.splits*)
apply(*rule conjI*; *clarsimp*)
apply(*fastforce dest*: *s-footprintD intvlD simp*: *size-of-def*)
apply(*frule s-footprintD2*)
apply(*drule s-footprintD*)
apply(*drule intvlD*, *clarsimp*)
apply(*rename-tac k*)
apply(*drule-tac x*=*k* **in** *spec*)
apply(*erule impE*)
apply(*simp add*: *size-of-def*)
apply(*subst* (*asm*) *word-unat.eq-norm*)
apply(*subst* (*asm*) *mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply(*rule max-size*)
apply(*force simp*: *map-le-def*)
done

lemma *singleton-dom*:
d,*g* ⊨$_t$ *p* ⟹ *dom* (*singleton p* (*v*::'*a*::*mem-type*) *h d*) = *s-footprint p*
apply(*subst singleton-dom'*)
apply(*fastforce dest*: *lift-state-dom*)
done

lemma *wf-heap-val-restrict* [*simp*]:
wf-heap-val s ⟹ *wf-heap-val* (*s* |` *X*)
unfolding *wf-heap-val-def* **by** (*auto simp*: *restrict-map-def*)

lemma *singleton-wf-heap-val* [*simp*]:
wf-heap-val (*singleton p v h d*)
unfolding *singleton-def* **by** *simp*

lemma *h-t-valid-restrict-proj-d*:
⟦ *proj-d s,g* ⊨$_t$ *p*; ∀ *x*. *x* ∈ *s-footprint p* ⟶ *s x* = *s'* *x* ⟧ ⟹
proj-d s',g ⊨$_t$ *p*
apply(*clarsimp simp*: *h-t-valid-def valid-footprint-def Let-def*)
apply(*rule conjI*)
apply(*drule-tac x*=*y* **in** *spec*)
apply *simp*
apply(*clarsimp simp*: *proj-d-def map-le-def*)
apply(*drule-tac x*=*ptr-val p* + *of-nat y* **in** *spec*)
apply(*drule-tac x*=*SIndexTyp a* **in** *spec*)
apply(*erule impE*)
apply(*erule s-footprintI*)
apply(*simp add*: *size-of-def*)
apply *simp*
apply(*clarsimp simp*: *proj-d-def*)
apply(*drule-tac x*=*y* **in** *spec*)
apply *clarsimp*
apply(*drule-tac x*=*ptr-val p* + *of-nat y* **in** *spec*)
apply(*drule-tac x*=*SIndexVal* **in** *spec*)
apply(*erule impE*)
apply(*rule s-footprintI2*)
apply(*simp add*: *size-of-def*)
apply *force*
done

lemma *s-valid-restrict* [*simp*]:
s |‘ *s-footprint p,g* ⊨$_s$ *p* = *s,g* ⊨$_s$ *p*
by (*fastforce simp*: *s-valid-def elim*: *h-t-valid-restrict-proj-d*)

lemma *proj-h-restrict*:
(*x*,*SIndexVal*) ∈ *X* ⟹ *proj-h* (*s* |‘ *X*) *x* = *proj-h s x*
by (*auto simp*: *proj-h-def*)

lemma *heap-list-s-restrict*:
(λ*x*. (*x*,*SIndexVal*)) ‘ {*p*..+*n*} ⊆ *X* ⟹ *heap-list-s* (*s* |‘ *X*) *n p* = *heap-list-s s n p*
apply(*induct n arbitrary*: *p*)
apply(*simp add*: *heap-list-s-def*)
apply(*clarsimp simp*: *heap-list-s-def*)
apply(*rule conjI*)
apply(*fastforce intro*: *proj-h-restrict intvl-self*)
apply(*fastforce intro*: *intvl-plus-sub-Suc*)
done

lemma *lift-typ-heap-restrict* [*simp*]:
lift-typ-heap g (*s* |‘ *s-footprint p*) *p* = *lift-typ-heap g s p*
apply(*clarsimp simp*: *lift-typ-heap-if*)
apply(*subst heap-list-s-restrict*)
apply *clarsimp*
apply(*drule intvlD*, *clarsimp*)
apply(*erule s-footprintI2*)
apply *simp*
done

lemma *singleton-s-valid*:
d,*g* ⊨$_t$ *p* ⟹ *singleton p* (*v*::′*a*::*mem-type*) *h d*,*g* ⊨$_s$ *p*
by (*simp add*: *singleton-def h-t-s-valid*)

lemma *singleton-lift-typ-heap-Some*:
d,*g* ⊨$_t$ *p* ⟹ *lift-typ-heap g* (*singleton p v h d*) *p* = *Some* (*v*::′*a*::*mem-type*)
by (*simp add*: *singleton-def lift-t lift-t-heap-update h-t-valid-restrict*)

lemma *sep-map-g*:
(*p* ↦$_g$ *v*) *s* ⟹ *g p*
by (*force simp*: *sep-map-def dest*: *lift-typ-heap-g*)

lemma *sep-map-g-sep-false*:
¬ *g p* ⟹ (*p* ↦$_g$ *v*) = *sep-false*
by (*simp add*: *sep-map-def lift-typ-heap-if s-valid-def h-t-valid-def*)

lemma *sep-map-singleton*:
d,*g* ⊨$_t$ *p* ⟹ ((*p*::′*a*::*mem-type ptr*) ↦$_g$ *v*) (*singleton p v h d*)
by (*simp add*: *sep-map-def singleton-lift-typ-heap-Some singleton-dom*)

lemma *sep-mapD*:
(*p* ↦$_g$ *v*) *s* ⟹ *lift-typ-heap g s p* = *Some v* ∧
dom s = *s-footprint p* ∧ *wf-heap-val s*
by (*simp add*: *sep-map-def*)

lemma *sep-map-lift-typ-heapD*:
(*p* ↦$_g$ *v*) *s* ⟹ *lift-typ-heap g s p* = *Some* (*v*::′*a*::*c-type*)
by (*simp add*: *sep-map-def*)

lemma *sep-map-dom-exc*:
(*p* ↦$_g$ (*v*::′*a*::*c-type*)) *s* ⟹ *dom s* = *s-footprint p*
by (*simp add*: *sep-map-def*)

lemma *sep-map-inj*:
⟦ (*p* ↦$_g$ (*v*::′*a*::*c-type*)) *s*; (*p* ↦$_h$ *v*′) *s* ⟧ ⟹ *v* = *v*′
by (*clarsimp simp*: *sep-map-def lift-typ-heap-if split*: *if-split-asm*)

lemma *sep-map-anyI-exc* [*simp*]:
(*p* ↦$_g$ *v*) *s* ⟹ (*p* ↦$_g$ −) *s*

by (*force simp*: *sep-map-any-def*)

lemma *sep-map-anyD-exc*:
(*p* $\mapsto_g$ −) *s* $\Longrightarrow$ $\exists$ *v*. (*p* $\mapsto_g$ *v*) *s*
by (*force simp*: *sep-map-any-def*)

lemma *sep-map-any-singleton*:
d,*g* $\models_t$ *i* $\Longrightarrow$ (*i* $\mapsto_g$ −) (*singleton i* (*v*::'*a*::*mem-type*) *h d*)
by (*unfold sep-map-any-def*, *rule exI* [**where** *x*=*v*], *erule sep-map-singleton*)

lemma *proj-h-heap-merge*:
proj-h (*s* ++ *t*) = (λ*x*. *if* (*x*,*SIndexVal*) ∈ *dom t then proj-h t x else proj-h s x*)
by (*force simp*: *proj-h-def split*: *option.splits*)

lemma *s-valid-heap-merge-right*:
s$_1$,*g* $\models_s$ *p* $\Longrightarrow$ *s*$_0$ ++ *s*$_1$,*g* $\models_s$ *p*
apply (*clarsimp simp*: *s-valid-def h-t-valid-def valid-footprint-def Let-def*)
apply (*rule conjI*)
apply(*drule-tac x*=*y* **in** *spec*, *simp*)
apply *clarsimp*
apply(*erule map-le-trans*)
apply(*clarsimp simp*: *proj-d-def map-le-def split*: *option.splits*)
apply(*drule-tac x*=*y* **in** *spec*, *simp*)
apply(*clarsimp simp*: *proj-d-def map-le-def split*: *option.splits*)
done

lemma *proj-d-map-add-fst*:
fst (*proj-d* (*s* ++ *t*) *x*) = (*if* (*x*,*SIndexVal*) ∈ *dom t then fst* (*proj-d t x*) *else*
fst (*proj-d s x*))
by (*auto simp*: *proj-d-def split*: *option.splits*)

lemma *proj-d-map-add-snd*:
snd (*proj-d* (*s* ++ *t*) *x*) *n* = (*if* (*x*,*SIndexTyp n*) ∈ *dom t then snd* (*proj-d t x*) *n else*
snd (*proj-d s x*) *n*)
by (*auto simp*: *proj-d-def split*: *option.splits*)

lemma *proj-d-restrict-map-fst*:
(*x*,*SIndexVal*) ∈ *X* $\Longrightarrow$ *fst* (*proj-d* (*s* |‘ *X*) *x*) = *fst* (*proj-d s x*)
by (*auto simp*: *proj-d-def*)

lemma *proj-d-restrict-map-snd*:
(*x*,*SIndexTyp n*) ∈ *X* $\Longrightarrow$ *snd* (*proj-d* (*s* |‘ *X*) *x*) *n* = *snd* (*proj-d s x*) *n*
by (*auto simp*: *proj-d-def*)

lemma *s-valid-heap-merge-right2*:
⟦ *s*$_0$ ++ *s*$_1$,*g* $\models_s$ *p*; *s-footprint p* ⊆ *dom s*$_1$ ⟧ $\Longrightarrow$ *s*$_1$,*g* $\models_s$ *p*
apply(*clarsimp simp*: *s-valid-def h-t-valid-def valid-footprint-def Let-def*)
apply(*rule conjI*)

apply(*clarsimp simp*: *map-le-def*)
apply(*subst proj-d-map-add-snd*)
apply(*clarsimp split*: *if-split-asm*)
apply(*subgoal-tac* (*ptr-val p* + *of-nat y*,*SIndexTyp a*) ∈ *s-footprint p*)
apply *fast*
apply(*erule s-footprintI*)
apply(*simp add*: *size-of-def*)
apply(*subgoal-tac* (*ptr-val p* + *of-nat y*,*SIndexVal*) ∈ *s-footprint p*)
apply(*drule* (*1*) *subsetD*)
apply *clarsimp*
apply(*subst* (*asm*) *proj-d-map-add-fst*)
apply(*drule-tac x*=*y* **in** *spec*)
apply(*clarsimp split*: *if-split-asm*)
apply(*rule s-footprintI2*)
apply(*simp add*: *size-of-def*)
done

lemma *heap-list-s-heap-merge-right'*:
⟦ *s*$_1$,*g* ⊨$_s$ (*p*::'*a*::*c-type ptr*); *n* ≤ *size-of TYPE*('*a*) ⟧ ⟹
heap-list-s (*s*$_0$ ++ *s*$_1$) *n* (*ptr-val p* + *of-nat* (*size-of TYPE*('*a*) − *n*))
= *heap-list-s s*$_1$ *n* (*ptr-val p* + *of-nat* (*size-of TYPE*('*a*) − *n*))
proof (*induct n*)
case *0* **thus** *?case* **by** (*simp add*: *heap-list-s-def*)
next
case (*Suc n*)
hence (*ptr-val p* + (*of-nat* (*size-of TYPE*('*a*) − *Suc n*)),*SIndexVal*) ∈ *dom s*$_1$
by − (*drule-tac x*=*size-of TYPE*('*a*) − *Suc n* **in** *s-valid-Some*, *auto*)
with *Suc* **show** *?case*
by (*simp add*: *heap-list-s-def proj-h-heap-merge algebra-simps*)
qed

lemma *heap-list-s-heap-merge-right*:
s$_1$,*g* ⊨$_s$ ((*Ptr p*)::'*a*::*c-type ptr*) ⟹
heap-list-s (*s*$_0$ ++ *s*$_1$) (*size-of TYPE*('*a*)) *p* = *heap-list-s s*$_1$ (*size-of TYPE*('*a*)) *p*
by (*force dest*: *heap-list-s-heap-merge-right'*)

lemma *lift-typ-heap-heap-merge-right*:
lift-typ-heap g s$_1$ *p* = *Some v* ⟹ *lift-typ-heap g* (*s*$_0$ ++ *s*$_1$) (*p*::'*a*::*c-type ptr*) = *Some v*
by (*force simp*: *lift-typ-heap-if s-valid-heap-merge-right heap-list-s-heap-merge-right*
split: *if-split-asm*)

lemma *lift-typ-heap-heap-merge-sep-map*:
(*p* ↦$_g$ *v*) *s*$_1$ ⟹ *lift-typ-heap g* (*s*$_0$ ++ *s*$_1$) *p* = *Some* (*v*::'*a*::*c-type*)
by (*drule sep-map-lift-typ-heapD*, *erule lift-typ-heap-heap-merge-right*)

lemma *sep-conjI*:
⟦ *P s*$_0$; *Q s*$_1$; *s*$_0$ ⊥ *s*$_1$; *s* = *s*$_1$ ++ *s*$_0$ ⟧ ⟹ (*P* ∧* *Q*) *s*

by (*force simp*: *sep-conj-def*)

lemma *sep-conjD*:
(*P* $\wedge^*$ *Q*) *s* $\Longrightarrow$ $\exists$ *s*$_0$ *s*$_1$. *s*$_0$ $\perp$ *s*$_1$ $\wedge$ *s* = *s*$_1$ ++ *s*$_0$ $\wedge$ *P* *s*$_0$ $\wedge$ *Q* *s*$_1$
by (*force simp*: *sep-conj-def*)

lemma *sep-map′I*:
((*p* $\mapsto_g$ *v*) $\wedge^*$ *sep-true*) *s* $\Longrightarrow$ (*p* $\hookrightarrow_g$ *v*) *s*
by (*simp add*: *sep-map′-def*)

lemma *sep-map′D*:
(*p* $\hookrightarrow_g$ *v*) *s* $\Longrightarrow$ ((*p* $\mapsto_g$ *v*) $\wedge^*$ *sep-true*) *s*
by (*simp add*: *sep-map′-def*)

lemma *sep-map′-g-exc*:
(*p* $\hookrightarrow_g$ *v*) *s* $\Longrightarrow$ *g p*
by (*force simp add*: *sep-map′-def dest*: *sep-conjD sep-map-g*)

lemma *sep-conj-sep-true*:
P s $\Longrightarrow$ (*P* $\wedge^*$ *sep-true*) *s*
by (*erule-tac* *s*$_1$=*Map.empty* **in** *sep-conjI*, *simp*+)

lemma *sep-map-sep-map′-exc* [*simp*]:
(*p* $\mapsto_g$ *v*) *s* $\Longrightarrow$ (*p* $\hookrightarrow_g$ *v*) *s*
by (*unfold sep-map′-def*, *erule sep-conj-sep-true*)

lemma *sep-conj-true* [*simp*]:
(*sep-true* $\wedge^*$ *sep-true*) = *sep-true*
apply (*rule ext*)
subgoal for *x*
apply *simp*
apply (*rule sep-conjI* [**where** *s*$_0$=*x* **and** *s*$_1$=*Map.empty*])
apply *auto*
done
done

lemma *sep-conj-assocD*:
assumes *l*: ((*P* $\wedge^*$ *Q*) $\wedge^*$ *R*) *s*
shows (*P* $\wedge^*$ (*Q* $\wedge^*$ *R*)) *s*
proof −
from *l* **obtain** *s′ s*$_2$ **where** *disj-o*: *s′* $\perp$ *s*$_2$ **and** *merge-o*: *s* = *s*$_2$ ++ *s′* **and**
l-o: (*P* $\wedge^*$ *Q*) *s′* **and** *r-o*: *R s*$_2$ **by** (*force dest*: *sep-conjD*)
then obtain *s*$_0$ *s*$_1$ **where** *disj-i*: *s*$_0$ $\perp$ *s*$_1$ **and** *merge-i*: *s′* = *s*$_1$ ++ *s*$_0$ **and**
l-i: *P s*$_0$ **and** *r-i*: *Q s*$_1$ **by** (*force dest*: *sep-conjD*)
from *disj-o disj-i merge-i* **have** *disj-i′*: *s*$_1$ $\perp$ *s*$_2$
by (*force simp*: *map-ac-simps*)
with *r-i* **and** *r-o* **have** *r-o′*: (*Q* $\wedge^*$ *R*) (*s*$_2$ ++ *s*$_1$) **by** (*fast intro*: *sep-conjI*)
from *disj-o merge-i disj-i disj-i′* **have** *s*$_0$ $\perp$ *s*$_2$ ++ *s*$_1$
by (*force simp*: *map-ac-simps*)

with *r-o′ l-i* **have** $(P \wedge^* (Q \wedge^* R))\ ((s_2 +\!\!+ s_1) +\!\!+ s_0)$
by (*force intro*: *sep-conjI*)
moreover from *merge-o merge-i disj-i disj-i′* **have** $s = ((s_2 +\!\!+ s_1) +\!\!+ s_0)$
by (*simp add*: *map-add-assoc* [*symmetric*])
ultimately show *?thesis* **by** *simp*
qed

lemma *sep-conj-com*:
$(P \wedge^* Q) = (Q \wedge^* P)$
by (*rule ext*) (*auto simp*: *map-ac-simps sep-conjI dest*!: *sep-conjD*)

lemma *sep-conj-false-right* [*simp*]:
$(P \wedge^* \mathit{sep\text{-}false}) = \mathit{sep\text{-}false}$
by (*force dest*: *sep-conjD*)

lemma *sep-conj-false-left* [*simp*]:
$(\mathit{sep\text{-}false} \wedge^* P) = \mathit{sep\text{-}false}$
by (*simp add*: *sep-conj-com*)

lemma *sep-conj-comD*:
$(P \wedge^* Q)\ s \Longrightarrow (Q \wedge^* P)\ s$
by (*simp add*: *sep-conj-com*)

lemma *exists-left*:
$(Q \wedge^* (\lambda s.\ \exists x.\ P\ x\ s)) = ((\lambda s.\ \exists x.\ P\ x\ s) \wedge^* Q)$ **by** (*simp add*: *sep-conj-com*)

lemma *sep-conj-assoc*:
$((P \wedge^* Q) \wedge^* R) = (P \wedge^* (Q \wedge^* R))$ (**is** *?x* = *?y*)
proof (*rule ext*, *standard*)
fix *s*
assume *?x s*
thus *?y s* **by** − (*erule sep-conj-assocD*)
next
fix *s*
assume *?y s*
hence $((R \wedge^* Q) \wedge^* P)\ s$ **by** (*simp add*: *sep-conj-com*)
hence $(R \wedge^* (Q \wedge^* P))\ s$ **by** − (*erule sep-conj-assocD*)
thus *?x s* **by** (*simp add*: *sep-conj-com*)
qed

lemma *sep-conj-left-com*:
$(P \wedge^* (Q \wedge^* R)) = (Q \wedge^* (P \wedge^* R))$ (**is** *?x* = *?y*)
proof −
have *?x* = $((Q \wedge^* R) \wedge^* P)$ **by** (*simp add*: *sep-conj-com*)
also have ... = $(Q \wedge^* (R \wedge^* P))$ **by** (*subst sep-conj-assoc*, *simp*)

finally show *?thesis* **by** (*simp add*: *sep-conj-com*)
qed

lemmas *sep-conj-ac* = *sep-conj-com sep-conj-assoc sep-conj-left-com*

lemma *sep-conj-empty*:
(*P* ∧* □) = *P*
proof (*rule ext*, *standard*)
fix *x*
assume (*P* ∧* □) *x*
thus *P x* **by** (*force simp*: *sep-emp-def dest*: *sep-conjD*)
next
fix *x*
assume *P x*
moreover have □ *Map.empty* **by** *simp*
ultimately show (*P* ∧* □) *x* **by** − (*erule* (*1*) *sep-conjI*, *auto*)
qed

lemma *sep-conj-empty′* [*simp*]:
(□ ∧* *P*) = *P*
by (*simp add*: *sep-conj-empty sep-conj-ac*)

lemma *sep-conj-true-P* [*simp*]:
(*sep-true* ∧* (*sep-true* ∧* *P*)) = (*sep-true* ∧* *P*)
by (*simp add*: *sep-conj-ac*)

lemma *sep-map′-unfold-exc*:
(*p* ↪*g* *v*) = ((*p* ↪*g* *v*) ∧* *sep-true*)
by (*simp add*: *sep-map′-def sep-conj-ac*)

lemma *sep-conj-disj*:
((λ*s*. *P s* ∨ *Q s*) ∧* *R*) *s* = ((*P* ∧* *R*) *s* ∨ (*Q* ∧* *R*) *s*) (**is** *?x* = (*?y* ∨ *?z*))
proof
assume *?x*
then obtain s_0 s_1 **where** s_0 ⊥ s_1 **and** *s* = s_1 ++ s_0 **and** *P* s_0 ∨ *Q* s_0 **and**
R s_1
by − (*drule sep-conjD*, *auto*)
moreover from *this* **have** ¬ *?z* ⟹ ¬ *Q* s_0
by − (*clarsimp*, *erule notE*, *erule* (*2*) *sep-conjI*, *simp*)
ultimately show *?y* ∨ *?z* **by** (*force intro*: *sep-conjI*)
next
have *?y* ⟹ *?x*
by (*force simp*: *map-ac-simps intro*: *sep-conjI dest*: *sep-conjD*)
moreover have *?z* ⟹ *?x*
by (*force simp*: *map-ac-simps intro*: *sep-conjI dest*: *sep-conjD*)
moreover assume *?y* ∨ *?z*
ultimately show *?x* **by** *fast*
qed

lemma *sep-conj-conj*:
$((\lambda s.\ P\ s \wedge Q\ s) \wedge^*\ R)\ s \Longrightarrow (P \wedge^*\ R)\ s \wedge (Q \wedge^*\ R)\ s$
by (*force intro*: *sep-conjI dest*!: *sep-conjD*)

lemma *sep-conj-exists1*:
$((\lambda s.\ \exists x.\ P\ x\ s) \wedge^*\ Q) = (\lambda s.\ (\exists x.\ (P\ x \wedge^*\ Q)\ s))$
by (*force intro*: *sep-conjI dest*: *sep-conjD*)

lemma *sep-conj-exists2*:
$(P \wedge^*\ (\lambda s.\ \exists x.\ Q\ x\ s)) = (\lambda s.\ (\exists x.\ (P \wedge^*\ Q\ x)\ s))$
by (*force intro*: *sep-conjI dest*: *sep-conjD*)

lemmas *sep-conj-exists* = *sep-conj-exists1 sep-conj-exists2*

lemma *sep-conj-forall*:
$((\lambda s.\ \forall x.\ P\ x\ s) \wedge^*\ Q)\ s \Longrightarrow (P\ x \wedge^*\ Q)\ s$
by (*force intro*: *sep-conjI dest*: *sep-conjD*)

lemma *sep-conj-impl*:
$[\![\ (P \wedge^*\ Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow P'\ s;\ \bigwedge s.\ Q\ s \Longrightarrow Q'\ s\ ]\!] \Longrightarrow (P' \wedge^*\ Q')\ s$
by (*force intro*: *sep-conjI dest*: *sep-conjD*)

lemma *sep-conj-sep-true-left*:
$(P \wedge^*\ Q)\ s \Longrightarrow (sep\text{-}true \wedge^*\ Q)\ s$
by (*erule sep-conj-impl*, *simp*+)

lemma *sep-conj-sep-true-right*:
$(P \wedge^*\ Q)\ s \Longrightarrow (P \wedge^*\ sep\text{-}true)\ s$
by (*subst* (*asm*) *sep-conj-com*, *drule sep-conj-sep-true-left*,
simp add: *sep-conj-ac*)

lemma *sep-globalise*:
$[\![\ (P \wedge^*\ R)\ s;\ (\bigwedge s.\ P\ s \Longrightarrow Q\ s)\ ]\!] \Longrightarrow (Q \wedge^*\ R)\ s$
by (*fast elim*: *sep-conj-impl*)

lemma *sep-implI*:
$\forall s'.\ s \perp s' \wedge x\ s' \longrightarrow y\ (s +\!\!+ s') \Longrightarrow (x \longrightarrow^*\ y)\ s$
by (*force simp*: *sep-impl-def*)

lemma *sep-implI'*:
assumes x: $\bigwedge h'.\ [\![\ h \perp h';\ x\ h'\ ]\!] \Longrightarrow y\ (h +\!\!+ h')$
shows $(x \longrightarrow^*\ y)\ h$
apply (*simp add*: *sep-impl-def*)
apply (*intro allI impI*)
apply (*erule conjE*)
apply (*erule* (*1*) x)
done

lemma *sep-implD*:

$(x \longrightarrow^* y)\ s \Longrightarrow \forall s'.\ s \perp s' \wedge x\ s' \longrightarrow y\ (s +\!\!+ s')$
by (*force simp*: *sep-impl-def*)

lemma *sep-emp-sep-impl* [*simp*]:
$(\square \longrightarrow^* P) = P$
apply(*rule ext*)
apply(*clarsimp simp*: *sep-impl-def*)
apply(*rule iffI*; *clarsimp?*)
apply(*drule-tac x*=*Map.empty* **in** *spec*)
apply *fastforce*
apply(*fastforce dest*: *sep-empD*)
done

lemma *sep-impl-sep-true* [*simp*]:
$(P \longrightarrow^* \textit{sep-true}) = \textit{sep-true}$
by (*force intro*: *sep-implI*)

lemma *sep-impl-sep-false* [*simp*]:
$(\textit{sep-false} \longrightarrow^* P) = \textit{sep-true}$
by (*force intro*: *sep-implI*)

lemma *sep-impl-sep-true-P*:
$(\textit{sep-true} \longrightarrow^* P)\ s \Longrightarrow P\ s$
by (*auto dest*!: *sep-implD*, *drule-tac x*=*Map.empty* **in** *spec*, *simp*)

lemma *sep-impl-sep-true-false* [*simp*]:
$(\textit{sep-true} \longrightarrow^* \textit{sep-false}) = \textit{sep-false}$
by (*force dest*: *sep-impl-sep-true-P*)

lemma *sep-impl-impl*:
$(P \longrightarrow^* Q \longrightarrow^* R) = (P \wedge^* Q \longrightarrow^* R)$
apply(*rule ext*)
apply(*rule iffI*)
apply(*rule sep-implI*)
apply *clarsimp*
apply(*drule sep-conjD*, *clarsimp*)
apply(*drule sep-implD*)
apply(*drule-tac x*=s_0 **in** *spec*)
apply(*erule impE*)
apply(*clarsimp simp*: *map-disj-def*, *fast*)
apply(*drule sep-implD*)
apply(*drule-tac x*=s_1 **in** *spec*)
apply(*erule impE*)
apply(*clarsimp simp*: *map-disj-def*, *fast*)
apply(*subst map-add-com* [**where** h_0=s_1])
apply(*clarsimp simp*: *map-disj-def*, *fast*)
apply(*subst map-add-assoc*)
apply *simp*
apply(*rule sep-implI*, *clarsimp*)

apply(*rule sep-implI*, *clarsimp*)
apply(*drule sep-implD*)
apply(*drule-tac x*=s' ++ $s'a$ **in** *spec*)
apply(*erule impE*)
apply(*rule conjI*)
apply(*clarsimp simp*: *map-disj-def*, *fast*)
apply(*erule* (*1*) *sep-conjI*)
apply(*clarsimp simp*: *map-disj-def*, *fast*)
apply(*subst map-add-com*)
apply(*clarsimp simp*: *map-disj-def*, *fast*)
apply *simp*
apply(*simp add*: *map-add-assoc*)
done

lemma *sep-conj-sep-impl*:
⟦ $P\ s$; $\bigwedge s.\ (P \wedge^*\ Q)\ s \Longrightarrow R\ s$ ⟧ $\Longrightarrow (Q \longrightarrow^*\ R)\ s$
apply (*rule sep-implI*, *clarsimp*)
proof −
fix s'
assume $P\ s$ **and** $s \perp s'$ **and** $Q\ s'$
hence $(P \wedge^*\ Q)\ (s\ ++\ s')$ **by** (*force simp*: *map-ac-simps intro*: *sep-conjI*)
moreover assume $\bigwedge s.\ (P \wedge^*\ Q)\ s \Longrightarrow R\ s$
ultimately show $R\ (s\ ++\ s')$ **by** *simp*
qed

lemma *sep-conj-sep-impl2*:
⟦ $(P \wedge^*\ Q)\ s$; $\bigwedge s.\ P\ s \Longrightarrow (Q \longrightarrow^*\ R)\ s$ ⟧ $\Longrightarrow R\ s$
by (*force simp*: *map-ac-simps dest*: *sep-implD sep-conjD*)

lemma *sep-map'-anyI-exc* [*simp*]:
$(p \hookrightarrow_g\ v)\ s \Longrightarrow (p \hookrightarrow_g\ -)\ s$
by (*force simp*: *sep-map'-any-def*)

lemma *sep-map'-anyD-exc*:
$(p \hookrightarrow_g\ -)\ s \Longrightarrow \exists v.\ (p \hookrightarrow_g\ v)\ s$
by (*force simp*: *sep-map'-any-def*)

lemma *sep-map'-any-unfold-exc*:
$(i \hookrightarrow_g\ -) = ((i \hookrightarrow_g\ -) \wedge^*\ sep\text{-}true)$
by (*rule ext*, *simp add*: *sep-map'-any-def*)
(*subst sep-map'-unfold-exc*, *subst sep-conj-com*, *subst sep-conj-exists*,
simp add: *sep-conj-ac*)

lemma *sep-map'-inj-exc*:
assumes *pv*: $(p \hookrightarrow_g\ (v::'a::c\text{-}type))\ s$ **and** *pv'*: $(p \hookrightarrow_h\ v')\ s$
shows $v = v'$
proof −
from *pv pv'* **obtain** $s_0\ s_1\ s_0'\ s_1'$ **where** *pv-m*: $(p \mapsto_g\ v)\ s_1$ **and**
pv'-m: $(p \mapsto_h\ v')\ s_1'$ **and** $s_0\ ++\ s_1 = s_0'\ ++\ s_1'$

```
    by (force simp: sep-map'-def map-ac-simps dest!: sep-conjD)
  hence s1 = s1' by (force dest!: map-add-right-dom-eq sep-map-dom-exc)
  with pv-m pv'-m show ?thesis by (force dest: sep-map-inj)
qed

lemma sep-map'-any-dom-exc:
  ((p::'a::mem-type ptr) ↪g _) s ⟹ (ptr-val p,SIndexVal) ∈ dom s
  by (clarsimp simp: sep-map'-def sep-map'-any-def sep-conj-ac
                dest!: sep-conjD)
     (subgoal-tac s1 (ptr-val p,SIndexVal) ≠ None, force simp: map-ac-simps,
      force dest: sep-map-dom-exc)

lemma sep-map'-dom-exc:
  (p ↪g (v::'a::mem-type)) s ⟹ (ptr-val p,SIndexVal) ∈ dom s
  apply(clarsimp simp: sep-map'-def sep-conj-ac dest!: sep-conjD)
  apply(subgoal-tac s1 (ptr-val p, SIndexVal) ≠ None)
   apply(force simp: map-ac-simps)
  apply(drule sep-map-dom-exc)
  apply(subgoal-tac (ptr-val p, SIndexVal) ∈ s-footprint p)
   apply fast
  apply(rule s-footprintI2 [where x=0, simplified])
  apply simp
  done

lemma sep-map'-lift-typ-heapD:
  (p ↪g v) s ⟹
      lift-typ-heap g s p = Some (v::'a::c-type)
  by (force simp: sep-map'-def map-ac-simps dest: sep-conjD
                  lift-typ-heap-heap-merge-sep-map)

lemma sep-map'-merge:
  assumes map'-v: (p ↪g v) s0 ∨ (p ↪g v) s1 and disj: s0 ⊥ s1
  shows (p ↪g v) (s0 ++ s1) (is ?x)
proof cases
  assume (p ↪g v) s0
  with disj show ?x
    apply (clarsimp simp: sep-map'-def sep-conj-ac dest!: sep-conjD)
    subgoal for s0' s1'
      apply (rule sep-conjI [where s1=s1' and s0=s0' ++ s1])
      apply (auto simp: map-ac-simps)
      done
    done
next
  assume ¬ (p ↪g v) s0
  with map'-v have (p ↪g v) s1 by simp
  with disj show ?x
    apply (clarsimp simp: sep-map'-def sep-conj-ac dest!: sep-conjD)
    subgoal for s0' s1'
      apply (rule sep-conjI [where s1=s1' and s0=s0 ++ s0'])
```

```
    apply (auto simp: map-ac-simps)
    done
  done
qed

lemma sep-conj-overlapD:
  ⟦ (P ∧* Q) s; ⋀s. P s ⟹ ((p::'a::mem-type ptr) ↪g −) s;
     ⋀s. Q s ⟹ (p ↪h −) s ⟧ ⟹ False
  apply(drule sep-conjD, clarsimp simp: map-disj-def)
  apply(subgoal-tac (ptr-val p,SIndexVal) ∈ dom s0 ∧ (ptr-val p,SIndexVal) ∈ dom
s1)
   apply fast
  apply(fast intro!: sep-map'-any-dom-exc)
  done

lemma sep-no-skew:
  (λs. (p ↪g v) s ∧ (q ↪h w) s) s ⟹
     p=q ∨ {ptr-val (p::'a::c-type ptr)..+size-of TYPE('a)} ∩
        {ptr-val q..+size-of TYPE('a)} = {}
  apply clarsimp
  apply(drule sep-map'-lift-typ-heapD)+
  apply(clarsimp simp: lift-typ-heap-if s-valid-def split: if-split-asm)
  apply(rule ccontr)
  apply(drule (1) h-t-valid-neq-disjoint; simp?)
  apply(rule peer-typ-not-field-of; simp)
  done

lemma sep-no-skew2:
  ⟦ (λs. (p ↪g v) s ∧ (q ↪h w) s) s; typ-uinfo-t TYPE('a) ⊥t typ-uinfo-t TYPE('b)
⟧
     ⟹ {ptr-val (p::'a::c-type ptr)..+size-of TYPE('a)} ∩
        {ptr-val (q::'b::c-type ptr)..+size-of TYPE('b)} = {}
  apply clarsimp
  apply(drule sep-map'-lift-typ-heapD)+
  apply(clarsimp simp: lift-typ-heap-if s-valid-def split: if-split-asm)
  apply(frule (1) h-t-valid-neq-disjoint[where q=q])
   apply(clarsimp simp: tag-disj-def sub-typ-proper-def)
   apply(simp add: typ-tag-lt-def)
  apply(clarsimp simp: tag-disj-def typ-tag-le-def field-of-t-def field-of-def)
  apply assumption
  done

lemma sep-conj-impl-same:
  (P ∧* (P −→* Q)) s ⟹ Q s
  apply(drule sep-conjD, clarsimp)
  apply(drule sep-implD)
  apply(drule-tac x=s0 in spec)
  apply(clarsimp simp: map-disj-com)
  done
```

definition *pure* :: $('a,'b)$ *map-assert* $\Rightarrow$ *bool* **where**
pure $P \equiv \forall\, s\ s'.\ P\ s = P\ s'$

lemma *pure-sep-true*:
pure sep-true
by (*simp add*: *pure-def*)

lemma *pure-sep-fasle*:
pure sep-true
by (*simp add*: *pure-def*)

lemma *pure-split*:
pure $P = (P = \textit{sep-true} \vee P = \textit{sep-false})$
by (*force simp*: *pure-def*)

lemma *pure-sep-conj*:
$[\![$ *pure* P; *pure* Q $]\!] \Longrightarrow$ *pure* $(P \wedge^*\ Q)$
by (*force simp*: *pure-split*)

lemma *pure-sep-impl*:
$[\![$ *pure* P; *pure* Q $]\!] \Longrightarrow$ *pure* $(P \longrightarrow^*\ Q)$
by (*force simp*: *pure-split*)

lemma *pure-conj-sep-conj*:
$[\![$ $(\lambda s.\ P\ s \wedge Q\ s)\ s$; *pure* $P \vee$ *pure* Q $]\!] \Longrightarrow (P \wedge^*\ Q)\ s$
by (*force simp*: *pure-split sep-conj-ac intro*: *sep-conj-sep-true*)

lemma *pure-sep-conj-conj*:
$[\![$ $(P \wedge^*\ Q)\ s$; *pure* P; *pure* Q $]\!] \Longrightarrow (\lambda s.\ P\ s \wedge Q\ s)\ s$
by (*force simp*: *pure-split*)

lemma *pure-conj-sep-conj-assoc*:
pure $P \Longrightarrow ((\lambda s.\ P\ s \wedge Q\ s) \wedge^*\ R) = (\lambda s.\ P\ s \wedge (Q \wedge^*\ R)\ s)$
by (*force simp*: *pure-split*)

lemma *pure-sep-impl-impl*:
$[\![$ $(P \longrightarrow^*\ Q)\ s$; *pure* P $]\!] \Longrightarrow P\ s \longrightarrow Q\ s$
by (*force simp*: *pure-split dest*: *sep-impl-sep-true-P*)

lemma *pure-impl-sep-impl*:
$[\![$ $P\ s \longrightarrow Q\ s$; *pure* P; *pure* Q $]\!] \Longrightarrow (P \longrightarrow^*\ Q)\ s$
by (*force simp*: *pure-split*)

definition
intuitionistic :: $('a,'b)$ *map-assert* $\Rightarrow$ *bool*
where
$intuitionistic\ P \equiv \forall\, s\ s'.\ P\ s \wedge s \subseteq_m s' \longrightarrow P\ s'$

lemma *intuitionisticI*:
$(\bigwedge s\ s'.\ [\![\ P\ s;\ s \subseteq_m s'\ ]\!] \Longrightarrow P\ s') \Longrightarrow intuitionistic\ P$
by (*unfold intuitionistic-def*, *fast*)

lemma *intuitionisticD*:
$[\![\ intuitionistic\ P;\ P\ s;\ s \subseteq_m s'\ ]\!] \Longrightarrow P\ s'$
by (*unfold intuitionistic-def*, *fast*)

lemma *pure-intuitionistic*:
$pure\ P \Longrightarrow intuitionistic\ P$
by (*clarsimp simp*: *intuitionistic-def pure-def*, *fast*)

lemma *intuitionistic-sep-map′*:
intuitionistic $(p \hookrightarrow_g v)$
proof (*rule intuitionisticI*)
fix $s\ s'$
assume $(p \hookrightarrow_g v)\ s$
then obtain $s_0\ s_1$ **where** $(p \mapsto_g v)\ s_0$ **and** s: $s = s_1 +\!\!+ s_0$
by (*force dest*!: *sep-conjD sep-map′D*)
moreover assume $s \subseteq_m s'$
with s **have** $s' = s' \mid` (UNIV - dom\ s_0) +\!\!+ s_0$
apply *simp*
apply (*drule map-add-le-mapE*)
by (*subst map-le-restrict*) *force*+
ultimately show $(p \hookrightarrow_g v)\ s'$ **by** (*force intro*: *sep-map′I sep-conjI*)
qed

lemma *intuitionistic-sep-conj-sep-true*:
intuitionistic $(P \wedge^* sep\text{-}true)$
by (*rule intuitionisticI*, *drule sep-conjD*, *clarsimp*)
(*erule-tac* $s_1{=}s' \mid` (dom\ s' - dom\ s_0)$ **in** *sep-conjI*, *simp*,
force simp: *map-disj-def*,
force intro: *map-le-dom-restrict-sub-add map-add-le-mapE sym*)

lemma *intuitionistic-sep-impl-sep-true*:
intuitionistic $(sep\text{-}true \longrightarrow^* P)$
apply (*rule intuitionisticI*, *rule sep-implI*, *clarsimp*)
proof −
fix $s\ s'\ s'a$
assume $(sep\text{-}true \longrightarrow^* P)\ s$ **and** *le*: $s \subseteq_m s'$ **and** $s' \perp s'a$
moreover from *this* **have** $P\ (s +\!\!+ (s' \mid` (dom\ s' - dom\ s) +\!\!+ s'a))$
by − (*drule sep-implD*,
drule-tac $x{=}s' \mid` (dom\ s' - dom\ s) +\!\!+ s'a$ **in** *spec*,
force simp: *map-disj-def dest*: *map-disj-map-le*)

moreover have *dom s* ∩ *dom* (*s'* |` (*dom s'* − *dom s*)) = {} **by** *force*
ultimately show *P* (*s'* ++ *s'a*)
by (*force simp*: *map-le-dom-restrict-sub-add map-add-assoc dest*!: *map-add-comm*)
qed

lemma *intuitionistic-conj*:
⟦ *intuitionistic P*; *intuitionistic Q* ⟧ ⟹ *intuitionistic* (λ*s*. *P s* ∧ *Q s*)
by (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

lemma *intuitionistic-disj*:
⟦ *intuitionistic P*; *intuitionistic Q* ⟧ ⟹ *intuitionistic* (λ*s*. *P s* ∨ *Q s*)
by (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

lemma *intuitionistic-forall*:
(⋀*x*. *intuitionistic* (*P x*)) ⟹ *intuitionistic* (λ*s*. ∀ *x*. *P x s*)
by (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

lemma *intuitionistic-exists*:
(⋀*x*. *intuitionistic* (*P x*)) ⟹ *intuitionistic* (λ*s*. ∃ *x*. *P x s*)
by (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

lemma *intuitionistic-sep-conj*:
intuitionistic (*P*::('*a*,'*b*) *map-assert*) ⟹ *intuitionistic* (*P* ∧* *Q*)
apply (*rule intuitionisticI*, *drule sep-conjD*, *clarsimp*)
proof −
fix *s' s*$_0$ *s*$_1$
assume *le*: *s*$_1$ ++ *s*$_0$ ⊆$_m$ *s'* **and** *disj*: *s*$_0$ ⊥ (*s*$_1$::'*a* ⇀ '*b*)
hence *le-restrict*: *s*$_0$ ⊆$_m$ *s'* |` (*dom s'* − *dom s*$_1$)
by − (*rule map-le-dom-subset-restrict*, *erule map-add-le-mapE*,
force simp: *map-disj-def dest*: *map-le-implies-dom-le*
map-add-le-mapE)
moreover assume *intuitionistic P* **and** *P s*$_0$
ultimately have *P* (*s'* |` (*dom s'* − *dom s*$_1$))
by − (*erule* (*2*) *intuitionisticD*)
moreover from *le-restrict* **have** *s'* |` (*dom s'* − *dom s*$_1$) ⊥ *s*$_1$
by (*force simp*: *map-disj-def dest*: *map-le-implies-dom-le*)
moreover from *le disj* **have** *s*$_1$ ⊆$_m$ *s'*
by (*subst* (*asm*) *map-add-comm*, *force simp*: *map-disj-def*)
(*erule map-add-le-mapE*)
hence *s'* = *s*$_1$ ++ *s'* |` (*dom s'* − *dom s*$_1$)
by (*subst map-add-comm*, *force simp*: *map-disj-def*,
force simp: *map-le-dom-restrict-sub-add map-add-comm map-disj-def*)
moreover assume *Q s*$_1$
ultimately show (*P* ∧* *Q*) *s'*
by − (*erule* (*3*) *sep-conjI*)
qed

lemma *intuitionistic-sep-impl*:
intuitionistic (*Q*::('*a*,'*b*) *map-assert*) ⟹ *intuitionistic* (*P* −→* *Q*)

```
  apply (rule intuitionisticI, rule sep-implI, clarsimp)
proof −
  fix s s′ s′a
  assume le: s ⊆m s′ and disj: s′ ⊥ (s′a::′a ⇀ ′b)
  moreover from this have s ++ s′a ⊆m s′ ++ s′a
  proof −
    from le disj have s ⊆m s ++ s′a
      by (subst map-add-com)
         (force simp: map-disj-def dest: map-le-implies-dom-le, simp)
    with le disj show ?thesis
      by − (rule map-add-le-mapI, subst map-add-com,
            auto elim: map-le-trans)
  qed
  moreover assume (P ⟶* Q) s and intuitionistic Q and P s′a
  ultimately show Q (s′ ++ s′a)
    by (fast elim: map-disj-map-le intuitionisticD dest: sep-implD)
qed

lemma strongest-intuitionistic:
  ¬ (∃ Q. (∀ s. (Q s ⟶ (P ∧* sep-true) s)) ∧ intuitionistic Q ∧
      Q ≠ (P ∧* sep-true) ∧ (∀ s. P s ⟶ Q s))
  by (clarsimp, rule ext) (force dest!: sep-conjD intuitionisticD)

lemma weakest-intuitionistic:
  ¬ (∃ Q. (∀ s. ((sep-true ⟶* P) s ⟶ Q s)) ∧ intuitionistic Q ∧
      Q ≠ (sep-true ⟶* P) ∧ (∀ s. Q s ⟶ P s))
  apply (clarsimp, rule ext)
  apply (rename-tac Q x)
  apply (rule iffI; clarsimp?)
  apply (rule sep-implI′)
  apply (rename-tac h′)
 apply (drule-tac s=x and s′=x ++ h′ in intuitionisticD; clarsimp simp: map-ac-simps)
  done

lemma intuitionistic-sep-conj-sep-true-P:
  ⟦ (P ∧* sep-true) s; intuitionistic P ⟧ ⟹ P s
  by (force dest: intuitionisticD sep-conjD)

lemma intuitionistic-sep-conj-sep-true-simp:
  intuitionistic P ⟹ (P ∧* sep-true) = P
  by (fast intro: sep-conj-sep-true elim: intuitionistic-sep-conj-sep-true-P)

lemma intuitionistic-sep-impl-sep-true-P:
  ⟦ P s; intuitionistic P ⟧ ⟹ (sep-true ⟶* P) s
  by (force simp: map-add-com intro: sep-implI dest: intuitionisticD)

lemma intuitionistic-sep-impl-sep-true-simp:
  intuitionistic P ⟹ (sep-true ⟶* P) = P
  by (fast elim: sep-impl-sep-true-P intuitionistic-sep-impl-sep-true-P)
```

definition
dom-exact :: (*'a*,*'b*) *map-assert* ⇒ *bool*
where
dom-exact P ≡ ∀ *s s'*. *P s* ∧ *P s'* ⟶ *dom s* = *dom s'*

lemma *dom-exactI*:
(⋀*s s'*. ⟦ *P s*; *P s'* ⟧ ⟹ *dom s* = *dom s'*) ⟹ *dom-exact P*
by (*unfold dom-exact-def*, *fast*)

lemma *dom-exactD*:
⟦ *dom-exact P*; *P* s_0; *P* s_1 ⟧ ⟹ *dom* s_0 = *dom* s_1
by (*unfold dom-exact-def*, *fast*)

lemma *dom-exact-sep-conj*:
⟦ *dom-exact P*; *dom-exact Q* ⟧ ⟹ *dom-exact* (*P* ∧* *Q*)
by (*rule dom-exactI*, (*drule sep-conjD*)+, *clarsimp*)
(*drule* (*2*) *dom-exactD*, *drule* (*2*) *dom-exactD*, *simp*)

lemma *dom-exact-sep-conj-conj*:
⟦ (*P* ∧* *R*) *s*; (*Q* ∧* *R*) *s*; *dom-exact R* ⟧ ⟹ ((λ*s*. *P s* ∧ *Q s*) ∧* *R*) *s*
apply ((*drule sep-conjD*)+)
by (*clarsimp simp*: *sep-conj-ac*,
rule sep-conjI, *fast*, *rule conjI*)
(*fast*, *drule* (*2*) *dom-exactD*, *drule* (*1*) *map-disj-add-eq-dom-right-eq*,
auto simp: *map-ac-simps*)

lemma *sep-conj-conj-simp*:
dom-exact R ⟹ ((λ*s*. *P s* ∧ *Q s*) ∧* *R*) = (λ*s*. (*P* ∧* *R*) *s* ∧ (*Q* ∧* *R*) *s*)
by (*fast intro*!: *sep-conj-conj dom-exact-sep-conj-conj*)

definition *dom-eps* :: (*'a*,*'b*) *map-assert* ⇒ *'a set* **where**
dom-eps P ≡ *THE x*. ∀ *s*. *P s* ⟶ *x* = *dom s*

lemma *dom-epsI*:
⟦ *dom-exact P*; *P s*; *x* ∈ *dom s* ⟧ ⟹ *x* ∈ *dom-eps P*
by (*unfold dom-eps-def*, *rule theI2* [**where** *a*=*dom s*])
(*fastforce simp*: *dom-exact-def*, *clarsimp*+)

lemma *dom-epsD* [*rule-format*]:
⟦ *dom-exact P*; *P s* ⟧ ⟹ *x* ∈ *dom-eps P* ⟶ *x* ∈ *dom s*
by (*unfold dom-eps-def*, *rule theI2* [**where** *a*=*dom s*])
(*fastforce simp*: *dom-exact-def*, *clarsimp*+)

lemma *dom-eps*:
⟦ *dom-exact P*; *P s* ⟧ ⟹ *dom s* = *dom-eps P*
by (*force intro*: *dom-epsI dest*: *dom-epsD*)

lemma *map-restrict-dom-exact*:
⟦ *dom-exact P*; *P s* ⟧ ⟹ *s* |‘ *dom-eps P* = *s*
by (*force simp*: *restrict-map-def None-com intro*: *dom-epsI*)

lemma *map-restrict-dom-exact2*:
⟦ *dom-exact P*; *P* s_0; s_0 ⊥ s_1 ⟧ ⟹ (s_1 |‘ *dom-eps P*) = *Map.empty*
by (*force simp*: *restrict-map-def map-disj-def dest*: *dom-epsD*)

lemma *map-restrict-dom-exact3*:
⟦ *dom-exact P*; *P s* ⟧ ⟹ *s* |‘ (*UNIV* − *dom-eps P*) = *Map.empty*
by (*force simp*: *restrict-map-def dest*: *dom-epsI*)

lemma *map-add-restrict-dom-exact*:
⟦ *dom-exact P*; s_0 ⊥ s_1; *P* s_1 ⟧ ⟹ (s_1 ++ s_0) |‘ (*dom-eps P*) = s_1
by (*simp add*: *map-add-restrict map-restrict-dom-exact*)
(*subst map-restrict-dom-exact2*, *auto simp*: *map-disj-def*)

lemma *map-add-restrict-dom-exact2*:
⟦ *dom-exact P*; s_0 ⊥ s_1; *P* s_0 ⟧ ⟹ (s_1 ++ s_0) |‘ (*UNIV* − *dom-eps P*) = s_1
by (*force simp*: *map-add-restrict map-disj-def dom-eps*
intro: *restrict-map-subdom dest*: *map-restrict-dom-exact3*)

lemma *dom-exact-sep-conj-forall*:
assumes *sc*: ∀ *x*. (*P x* ∧* *Q*) *s* **and** *de*: *dom-exact Q*
shows ((λ*s*. ∀ *x*. *P x s*) ∧* *Q*) *s*
proof (*rule sep-conjI* [**where** s_0=*s* |‘ (*UNIV* − *dom-eps Q*) **and** s_1=*s* |‘ *dom-eps Q*])
from *sc de* **show** ∀ *x*. *P x* (*s* |‘ (*UNIV* − *dom-eps Q*))
by (*force simp*: *map-add-restrict-dom-exact2 sep-conj-ac dest*: *sep-conjD*)
next
from *sc de* **show** *Q* (*s* |‘ *dom-eps Q*)
by (*force simp*: *map-add-restrict-dom-exact dest*!: *sep-conjD spec*)
next
from *sc de* **show** *s* |‘ (*UNIV* − *dom-eps Q*) ⊥ *s* |‘ *dom-eps Q*
by (*force simp*: *map-add-restrict-dom-exact2 map-ac-simps*
dest: *map-add-restrict-dom-exact dest*!: *sep-conjD spec*)
next
show *s* = *s* |‘ *dom-eps Q* ++ *s* |‘ (*UNIV* − *dom-eps Q*) **by** *simp*
qed

lemma *sep-conj-forall-simp*:
dom-exact Q ⟹ ((λ*s*. ∀ *x*. *P x s*) ∧* *Q*) = (λ*s*. ∀ *x*. (*P x* ∧* *Q*) *s*)
by (*fast dest*: *sep-conj-forall dom-exact-sep-conj-forall*)

lemma *dom-exact-sep-map*:
dom-exact (*i* ↦$_g$ *x*)
by (*clarsimp simp*: *dom-exact-def sep-map-def*)

lemma *dom-exact-P-emp*:
⟦ *dom-exact P*; *P Map.empty*; *P s* ⟧ ⟹ *s* = *Map.empty*
by (*auto simp*: *dom-exact-def*)

definition
strictly-exact :: ('*a*,'*b*) *map-assert* ⇒ *bool*
where
strictly-exact P ≡ ∀ *s s'*. *P s* ∧ *P s'* ⟶ *s* = *s'*

lemma *strictly-exactD*:
⟦ *strictly-exact P*; *P* s_0; *P* s_1 ⟧ ⟹ s_0 = s_1
by (*unfold strictly-exact-def*, *fast*)

lemma *strictly-exactI*:
(⋀*s s'*. ⟦ *P s*; *P s'* ⟧ ⟹ *s* = *s'*) ⟹ *strictly-exact P*
by (*unfold strictly-exact-def*, *fast*)

lemma *strictly-exact-dom-exact*:
strictly-exact P ⟹ *dom-exact P*
by (*force simp*: *strictly-exact-def dom-exact-def*)

lemma *strictly-exact-sep-emp*:
strictly-exact □
by (*force simp*: *strictly-exact-def dest*: *sep-empD*)

lemma *strictly-exact-sep-conj*:
⟦ *strictly-exact P*; *strictly-exact Q* ⟧ ⟹ *strictly-exact* (*P* ∧* *Q*)
by (*force intro*!: *strictly-exactI dest*: *sep-conjD strictly-exactD*)

lemma *strictly-exact-conj-impl*:
⟦ (*Q* ∧* *sep-true*) *s*; *P s*; *strictly-exact Q* ⟧ ⟹ (*Q* ∧* (*Q* ⟶* *P*)) *s*
by (*force intro*: *sep-conjI sep-implI dest*: *strictly-exactD dest*!: *sep-conjD*)

lemma *dom-eps-sep-emp* [*simp*]:
dom-eps □ = {}
apply(*subst dom-eps* [*symmetric*])
apply(*rule strictly-exact-dom-exact*)
apply(*rule strictly-exact-sep-emp*)
apply(*rule sep-emp-empty*)
apply *simp*
done

lemma *dom-eps-sep-map*:
g p ⟹ *dom-eps* (*p* ↦$_g$ (*v*::'*a*::*mem-type*)) = *s-footprint p*
apply(*subst dom-eps* [*symmetric*])
apply(*rule dom-exact-sep-map*)

apply(*rule sep-map-singleton*)
apply(*erule ptr-retyp-h-t-valid*)
apply(*subst singleton-dom*)
apply(*erule ptr-retyp-h-t-valid*)
apply *simp*
done

definition *non-empty* :: ('*a*,'*b*) *map-assert* ⇒ *bool* **where**
non-empty P ≡ ∃ *s*. *P s*

lemma *non-emptyI*:
P s ⟹ *non-empty P*
by (*force simp*: *non-empty-def*)

lemma *non-emptyD*:
non-empty P ⟹ ∃ *s*. *P s*
by (*force simp*: *non-empty-def*)

lemma *non-empty-sep-true*:
non-empty sep-true
by (*simp add*: *non-empty-def*)

lemma *non-empty-sep-false*:
¬ *non-empty sep-false*
by (*simp add*: *non-empty-def*)

lemma *non-empty-sep-emp*:
non-empty □
unfolding *non-empty-def* **by** (*rule exI*, *rule sep-emp-empty*)

lemma *non-empty-sep-map*:
g p ⟹ *non-empty* (*p* ↦$_g$ (*v*::'*a*::*mem-type*))
apply(*unfold non-empty-def*)
apply(*rule exI*, *rule sep-map-singleton*)
apply(*erule ptr-retyp-h-t-valid*)
done

lemma *non-empty-sep-conj*:
⟦ *non-empty P*; *non-empty Q*; *dom-exact P*; *dom-exact Q*;
dom-eps P ∩ *dom-eps Q* = {} ⟧ ⟹ *non-empty* (*P* ∧* *Q*)
apply(*clarsimp simp*: *non-empty-def*)
subgoal for *s s'*
apply(*rule exI* [**where** *x*=*s*++*s'*])
apply(*rule sep-conjI*, *assumption*+)
apply(*clarsimp simp*: *map-disj-def dom-eps*)
apply(*subst map-add-com*)
apply(*clarsimp simp*: *map-disj-def dom-eps*)

apply *simp*
done
done

lemma *non-empty-sep-map'*:
g p $\Longrightarrow$ *non-empty* (*p* $\hookrightarrow_g$ (*v*::'*a*::*mem-type*))
apply(*unfold sep-map'-def*)
apply(*clarsimp simp*: *non-empty-def sep-conj-ac*)
apply(*rule exI* [**where** *x*=*singleton p v h* (*ptr-retyp p d*)])
apply(*rule sep-conjI* [**where** s_0=*Map.empty*])
apply *simp*
apply(*rule sep-map-singleton*)
apply(*erule ptr-retyp-h-t-valid*)
apply(*simp add*: *map-disj-def*)
apply *simp*
done

lemma *non-empty-sep-impl*:
¬ *P Map.empty* $\Longrightarrow$ *non-empty* (*P* $\longrightarrow^*$ *Q*)
apply(*clarsimp simp*: *non-empty-def*)
apply(*rule exI* [**where** *x*=λs. *Some undefined*])
apply(*rule sep-implI*)
apply(*clarsimp simp*: *map-disj-def*)
done

lemma *pure-conj-right*: (*Q* $\wedge^*$ (λs. *P'* $\wedge$ *Q' s*)) = (λs. *P'* $\wedge$ (*Q* $\wedge^*$ *Q'*) *s*)
by (*rule ext*, *standard*, *standard*, *clarsimp dest*!: *sep-conjD*)
(*erule sep-conj-impl*, *auto*)

lemma *pure-conj-right'*: (*Q* $\wedge^*$ (λs. *P' s* $\wedge$ *Q'*)) = (λs. *Q'* $\wedge$ (*Q* $\wedge^*$ *P'*) *s*)
by (*simp add*: *conj-comms pure-conj-right*)

lemma *pure-conj-left*: ((λs. *P'* $\wedge$ *Q' s*) $\wedge^*$ *Q*) = (λs. *P'* $\wedge$ (*Q'* $\wedge^*$ *Q*) *s*)
by (*simp add*: *pure-conj-right sep-conj-ac*)

lemma *pure-conj-left'*: ((λs. *P' s* $\wedge$ *Q'*) $\wedge^*$ *Q*) = (λs. *Q'* $\wedge$ (*P'* $\wedge^*$ *Q*) *s*)
by (*subst conj-comms*, *subst pure-conj-left*, *simp*)

lemmas *pure-conj* = *pure-conj-right pure-conj-right' pure-conj-left pure-conj-left'*

declare *pure-conj* [*simp add*]

lemma *sep-conj-sep-conj-sep-impl-sep-conj*:
(*P* $\wedge^*$ *R*) *s* $\Longrightarrow$ (*P* $\wedge^*$ (*Q* $\longrightarrow^*$ (*Q* $\wedge^*$ *R*))) *s*
by (*erule* (*1*) *sep-conj-impl*, *erule sep-conj-sep-impl*, *simp add*: *sep-conj-ac*)

lemma *sep-map'-conjE1-exc*:

$[\![(P \wedge^* Q)\ s; \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow_g v)\ s]\!] \Longrightarrow (i \hookrightarrow_g v)\ s$
by (*subst sep-map′-unfold-exc*, *erule sep-conj-impl*, *simp*+)

lemma *sep-map′-conjE2-exc*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ Q\ s \Longrightarrow (i \hookrightarrow_g v)\ s]\!] \Longrightarrow (i \hookrightarrow_g v)\ s$
by (*subst* (*asm*) *sep-conj-com*, *erule sep-map′-conjE1-exc*, *simp*)

lemma *sep-map′-any-conjE1-exc*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow_g -)\ s]\!] \Longrightarrow (i \hookrightarrow_g -)\ s$
by (*subst sep-map′-any-unfold-exc*, *erule sep-conj-impl*, *simp*+)

lemma *sep-map′-any-conjE2-exc*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ Q\ s \Longrightarrow (i \hookrightarrow_g -)\ s]\!] \Longrightarrow (i \hookrightarrow_g -)\ s$
by (*subst* (*asm*) *sep-conj-com*, *erule sep-map′-any-conjE1-exc*, *simp*)

lemma *sep-conj-mapD-exc*:
$((i \mapsto_g v) \wedge^* P)\ s \Longrightarrow (i \hookrightarrow_g v)\ s \wedge ((i \mapsto_g -) \wedge^* P)\ s$
by (*force simp*: *sep-conj-ac intro*: *sep-conj-impl sep-map′-conjE2-exc*)

lemma *sep-impl-conj-sameD*:
$[\![(P \longrightarrow^* P \wedge^* Q)\ s;$ *dom-exact* $P;$ *non-empty* $P;$ *dom* $s \subseteq$ *UNIV* $-$ *dom-eps* P
$]\!]$
$\Longrightarrow Q\ s$
apply(*drule sep-implD*)
apply(*clarsimp simp*: *non-empty-def*)
apply(*rename-tac* s')
apply(*drule-tac* $x{=}s'$ **in** *spec*)
apply(*erule impE*)
apply(*fastforce simp*: *map-disj-def dom-eps*)
apply(*drule sep-conjD*, *clarsimp*)
apply(*clarsimp simp*: *map-disj-def*)
apply(*subst* (*asm*) *map-add-comm*)
apply(*clarsimp simp*: *dom-eps*)
apply *fast*
apply(*subst* (*asm*) *map-add-comm*, *fast*)
apply(*drule map-disj-add-eq-dom-right-eq*)
apply(*simp add*: *dom-eps*)
apply(*clarsimp simp*: *dom-eps map-disj-def*)
apply *fast*
apply(*simp add*: *map-disj-def*)
apply *clarsimp*
done

lemma *sep-impl-conj-sameI*:
$Q\ s \Longrightarrow (P \longrightarrow^* P \wedge^* Q)\ s$
by (*fastforce intro*: *sep-implI sep-conjI simp*: *map-disj-com*)

end

theory *SepCode*
imports
Separation-UMM
Simpl.Vcg
begin

definition
singleton-t :: *'a::c-type ptr* ⇒ *'a* ⇒ *heap-state*
where
singleton-t p v ≡ *lift-state* (*heap-update p v* (λ*x*. *0*), (*ptr-retyp p empty-htd*))

definition
tagd :: *'a ptr-guard* ⇒ *'a::c-type ptr* ⇒ *heap-assert* (**infix** ‹⊢$_s$› *100*)
where
g ⊢$_s$ *p* ≡ λ*s*. *s*,*g* ⊨$_s$ *p* ∧ *dom s* = *s-footprint p*

definition
field-footprint :: *'a::c-type ptr* ⇒ *qualified-field-name* ⇒ (*addr* × *s-heap-index*) *set*
where
field-footprint p f ≡
s-footprint-untyped (*ptr-val p* + *of-nat* (*field-offset TYPE*(*'a*) *f*))
(*export-uinfo* (*field-typ TYPE*(*'a*) *f*))

definition
fs-footprint :: *'a::c-type ptr* ⇒ *qualified-field-name set* ⇒ (*addr* × *s-heap-index*) *set*
where
fs-footprint p F ≡ ⋃{*field-footprint p f* | *f*. *f* ∈ *F*}

definition *fields* :: *'a::c-type itself* ⇒ *qualified-field-name set* **where**
fields t ≡ {*f*. *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* ≠ *None*}

definition
mfs-sep-map :: *'a::c-type ptr* ⇒ *'a ptr-guard* ⇒ *qualified-field-name set* ⇒ *'a* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix mfs-sep-map*››- ↦$_{-}^{-}$ -)› [*56*,*0*,*0*,*51*] *56*)
where
p ↦$_g^F$ *v* ≡ λ*s*. *lift-typ-heap g* (*singleton-t p v* ++ *s*) *p* = *Some v* ∧
F ⊆ *fields TYPE*(*'a*) ∧
dom s = *s-footprint p* − *fs-footprint p F* ∧ *wf-heap-val s*

notation (*input*)
mfs-sep-map (‹(‹*open-block notation*=‹*mixfix mfs-sep-map*››- ↦$_{-}^{-}$ -)› [*56*,*0*,*1000*,*51*] *56*)

definition
disjoint-fn :: *qualified-field-name* ⇒ *qualified-field-name set* ⇒ *bool*
where
disjoint-fn f F ≡ ∀*f'*∈*F*. ¬ *f* ≤ *f'* ∧ ¬ *f'* ≤ *f*

definition
sep-cut' :: *addr* ⇒ *nat* ⇒ (*s-addr*,'*b*) *map-assert*
where
sep-cut' p n ≡ λ*s*. *dom s* = {(*x*,*y*). *x* ∈ {*p*..+*n*}}

definition
sep-cut :: *addr* ⇒ *addr-bitsize word* ⇒ (*s-addr*,'*b*) *map-assert*
where
sep-cut x y ≡ *sep-cut' x* (*unat y*)

—-

lemma *heap-list-h-eq*:
⟦ *x* ∈ {*p*..+*q*}; *q* < *addr-card*; *heap-list h q p* = *heap-list h' q p* ⟧ ⟹ *h x* = *h' x*
proof (*induct q arbitrary*: *p*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*) **thus** *?case* **by** (*force dest*: *intvl-neq-start*)
qed

lemma *s-footprint-intvl*:
(*a*, *SIndexVal*) ∈ *s-footprint p* = (*a* ∈ {*ptr-val* (*p*::'*a*::*c-type ptr*)..+*size-of TYPE*('*a*)})
apply(*clarsimp simp*: *s-footprint-def s-footprint-untyped-def*)
apply(*rule iffI*, *clarsimp*)
apply(*rule intvlI*)
apply(*simp add*: *size-of-def*)
apply(*drule intvlD*, *clarsimp*)
apply(*simp add*: *size-of-def*)
apply *fast*
done

lemma *singleton-t-dom* [*simp*]:
dom (*singleton-t p* (*v*::'*a*::*mem-type*)) = *s-footprint p*
supply *unsigned-of-nat* [*simp del*]
apply(*rule equalityI*; *clarsimp simp*: *singleton-t-def lift-state-def s-footprint-intvl*
split: *s-heap-index.splits if-split-asm option.splits*)
apply(*rule ccontr*)
apply(*simp add*: *ptr-retyp-None*)
subgoal for *a b y x2 x2a*
apply(*cases a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)})
apply(*simp add*: *ptr-retyp-footprint list-map-eq split*: *if-split-asm*)
apply(*drule intvlD*, *clarsimp*)
apply(*rule s-footprintI*)
apply(*subst* (*asm*) *word-unat.eq-norm*)
apply(*subst* (*asm*) *mod-less*)

apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply(*rule max-size*)
apply(*simp add*: *map-le-def*)
apply *assumption*
apply(*simp add*: *ptr-retyp-None*)
done
apply(*rule conjI*; *clarsimp*)
apply (*simp add*: *ptr-retyp-d-empty s-footprintD*)
apply(*frule s-footprintD2*)
apply(*frule s-footprintD*)
apply(*simp add*: *ptr-retyp-footprint*)
done

lemma *heap-update-merge*:
assumes *val*: *d*,*g* ⊨$_t$ *p*
shows *lift-state* ((*heap-update p* (*v*::'*a*::*mem-type*) *h*),*d*)
= *lift-state* (*h*,*d*) ++ *singleton p v h d* (**is** *?x* = *?y*)
proof (*rule ext*, *cases*)
fix *x*
assume *c*: *x* ∈ *dom* (*singleton p v h d*)
with *val*
have *lift-state* (*heap-update-list* (*ptr-val p*)
(*to-bytes v* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))) *h*,
d) *x* =
singleton p v h d x
by (*auto simp*: *heap-list-update-to-bytes singleton-def lift-state-def heap-update-def*
singleton-dom
split: *option.splits s-heap-index.splits*)
with *c* **show** *?x x* = *?y x* **by** (*force simp*: *heap-update-def dest*: *domD*)
next
fix *x*
assume *nc*: *x* ∉ *dom* (*singleton p v h d*)
with *val* **show** *?x x* = *?y x*
apply(*cases x*)
apply(*clarsimp simp*: *lift-state-def heap-update-def map-add-def*
split: *option.splits s-heap-index.splits*)
apply(*safe*; *clarsimp*)
by (*metis heap-list-length heap-update-nmem-same len nc s-footprint-intvl singleton-dom*)
qed

lemma *tagd-dom-exc*:
(*g* ⊢$_s$ *p*) *s* ⟹ *dom s* = *s-footprint p*
by (*clarsimp simp*: *tagd-def*)

lemma *tagd-dom-p-exc*:
(*g* ⊢$_s$ *p*) *s* ⟹ (*ptr-val* (*p*::'*a*::*mem-type ptr*),*SIndexVal*) ∈ *dom s*

by (*drule tagd-dom-exc*) *clarsimp*

lemma *tagd-g-exc*:
(*g* ⊢$_s$ *p* ∧* *P*) *s* ⟹ *g p*
by (*drule sep-conjD*, *force simp*: *tagd-def elim*: *s-valid-g*)

lemma *sep-map-tagd-exc*:
(*p* ↦$_g$ (*v*::'*a*::*mem-type*)) *s* ⟹ (*g* ⊢$_s$ *p*) *s*
by (*clarsimp simp*: *sep-map-def tagd-def lift-typ-heap-s-valid*)

lemma *sep-map-any-tagd-exc*:
(*p* ↦$_g$ −) *s* ⟹ (*g* ⊢$_s$ (*p*::'*a*::*mem-type ptr*)) *s*
by (*clarsimp dest*!: *sep-map-anyD-exc*, *erule sep-map-tagd-exc*)

lemma *ptr-retyp-tagd-exc*:
g (*p*::'*a*::*mem-type ptr*) ⟹
(*g* ⊢$_s$ *p*) (*lift-state* (*h*, *ptr-retyp p empty-htd*))
supply *unsigned-of-nat* [*simp del*]
apply(*simp add*: *tagd-def ptr-retyp-s-valid lift-state-dom*)
apply(*rule equalityI*;
clarsimp simp: *lift-state-def split*: *s-heap-index.splits if-split-asm option.splits*)
subgoal for *a*
apply(*cases a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)})
apply(*drule intvlD*, *clarsimp*)
apply(*rule s-footprintI2*, *simp*)
apply(*subst* (*asm*) *ptr-retyp-None*; *simp*)
done
subgoal for *a b y x2 x2a*
apply(*cases a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)})
apply(*subst* (*asm*) *ptr-retyp-footprint*)
apply *simp*
apply(*drule intvlD*, *clarsimp*)
apply(*subst* (*asm*)*word-unat.eq-norm*)
apply(*subst* (*asm*) *mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply *simp*
apply(*subst* (*asm*) *list-map-eq*)
apply(*clarsimp split*: *if-split-asm*)
apply(*erule* (*1*) *s-footprintI*)
apply(*simp add*: *ptr-retyp-None*)
done
subgoal for *a b*
apply(*rule conjI*; *clarsimp*)
apply(*cases a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)})
apply(*simp add*: *ptr-retyp-footprint*)
apply(*drule s-footprintD*)
apply *simp*
apply(*cases a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)})

apply(*subst* (*asm*) *ptr-retyp-footprint*)
apply *simp*
apply(*drule intvlD*, *clarsimp*)
apply(*subst* (*asm*)*word-unat.eq-norm*)
apply(*subst* (*asm*) *mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply *simp*
apply(*subst* (*asm*) *list-map-eq*)
apply(*clarsimp split*: *if-split-asm*)
apply(*drule s-footprintD2*)
apply *simp*
apply(*subst* (*asm*) *unat-of-nat*)
apply(*subst* (*asm*) *mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*, *simp*)
apply *simp*
apply(*fastforce dest*: *s-footprintD*)
done
done

lemma *singleton-dom-proj-d* [*simp*]:
(*g* ⊢$_s$ *p*) *s* ⟹ *dom* (*singleton p* (*v*::'*a*::*mem-type*) *h* (*proj-d s*)) = *dom s*
by (*clarsimp simp*: *tagd-def singleton-dom s-valid-def*)

lemma *singleton-d-restrict-eq*:
restrict-s d (*s-footprint p*) = *restrict-s d'* (*s-footprint p*)
⟹ *singleton p v h d* = *singleton p* (*v*::'*a*::*mem-type*) *h d'*
apply(*clarsimp simp*: *singleton-def*)
apply(*rule ext*)
subgoal for *x*
apply(*cases x* ∈ *s-footprint p*; *simp*)
apply(*cases x*, *clarsimp*)
subgoal for *a b*
apply(*drule fun-cong* [**where** *x*=*a*])
apply(*clarsimp simp*: *s-footprint-restrict lift-state-def*
split: *s-heap-index.splits if-split-asm option.splits*)
apply(*rule conjI*, *clarsimp simp*: *restrict-s-def*)
apply(*clarsimp simp*: *restrict-s-def*)
subgoal for *x'*
apply(*drule fun-cong* [**where** *x*=*x'*])
apply *auto*
done
done
done
done

lemma *sep-heap-update'-exc*:
assumes *sep*: (*g* ⊢$_s$ *p* ∧* (*p* ↦$_g$ *v* ⟶* *P*)) (*lift-state* (*h*,*d*))

shows P (*lift-state* (*heap-update* p (v::'a::*mem-type*) h,d))
proof −
from *sep* **obtain** s_0 s_1 **where** *disj*: $s_0 \perp s_1$ **and**
merge: *lift-state* (h,d) = s_1 ++ s_0 **and**
l: ($g \vdash_s p$) s_0 **and** r: ($p \mapsto_g v \longrightarrow^* P$) s_1 **by** (*force dest*: *sep-conjD*)
moreover from *this* **have** $s_1 \perp$ *singleton* p v h (*proj-d* s_0)
by (*fastforce simp*: *map-disj-def*)
moreover from l **have** g p **by** (*force simp*: *tagd-def elim*: *s-valid-g*)
moreover from *merge* l **have** *lift-state* (h,d),$g \models_s p$
by (*force simp*: *tagd-def intro*: *s-valid-heap-merge-right*)
hence d,$g \models_t p$ **by** (*simp add*: *h-t-s-valid*)
moreover from l **have** s_0 ++ *singleton* p v h (*proj-d* s_0) = *singleton* p v h (*proj-d* s_0)
by (*force simp*: *map-add-dom-eq singleton-dom dest*: *tagd-dom-exc*)
moreover from l *merge* **have** s_1 ++ *singleton* p v h (*proj-d* s_0) = s_1 ++ s_0 ++ *singleton* p v h d
apply(*clarsimp simp*: *tagd-def*)
apply(*rule ext*)
subgoal for x
apply(*cases* x, *clarsimp simp*: *restrict-map-def*)
subgoal for a b
apply(*simp add*: *s-valid-def*)
apply(*drule singleton-dom* [**where** v=v **and** h=h])
apply(*drule fun-cong* [**where** x=(a,b)])
apply(*cases* (a,b) ∈ *s-footprint* p)
subgoal premises *prems*
proof −
have (s_1 ++ *singleton* p v h (*proj-d* s_0)) (a, b) = *singleton* p v h d (a, b)
using *prems*
by(*force simp*: *lift-state-def map-add-def singleton-def proj-d-def split*: *option.splits s-heap-index.splits if-split-asm*)
then show *?thesis* **using** *prems*
apply(*clarsimp simp*: *map-add-def s-valid-def split*: *option.splits*)
apply *force*
apply (*fastforce intro*: *sym*)
done
qed
apply(*auto simp*: *map-add-def singleton-def split*: *option.splits*)
done
done
done
ultimately show *?thesis*
by (*fastforce dest*: *sep-implD simp*: *sep-map-singleton tagd-def s-valid-def heap-update-merge*)
qed

lemma *sep-heap-update-exc*:
⟦ ($p \mapsto_g$ − ∧* ($p \mapsto_g v \longrightarrow^* P$)) (*lift-state* ($h$,$d$)) ⟧ ⟹
P (*lift-state* (*heap-update* p (v::'a::*mem-type*) h,d))
by (*force intro*: *sep-heap-update'-exc dest*: *sep-map-anyD-exc sep-map-tagd-exc*

elim: *sep-conj-impl*)

lemma *sep-heap-update-global'-exc*:
(*g* ⊢$_s$ *p* ∧* *R*) (*lift-state* (*h*,*d*)) ⟹
((*p* ↦$_g$ *v*) ∧* *R*) (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
by (*rule sep-heap-update'-exc*, *erule sep-conj-sep-conj-sep-impl-sep-conj*)

lemma *sep-heap-update-global-exc*:
(*p* ↦$_g$ − ∧* *R*) (*lift-state* (*h*,*d*)) ⟹
((*p* ↦$_g$ *v*) ∧* *R*) (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
by (*fast intro*: *sep-heap-update-global'-exc sep-conj-impl sep-map-any-tagd-exc*)

lemma *sep-heap-update-global-exc2*:
(*p* ↦$_g$ *u* ∧* *R*) (*lift-state* (*h*,*d*)) ⟹
((*p* ↦$_g$ *v*) ∧* *R*) (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
by (*fastforce intro*: *sep-heap-update-global-exc simp*: *sep-map-any-def sep-conj-exists*)

lemma *heap-update-mem-same-point*:
⟦ *q* ∈ {*p*..+*length v*}; *length v* < *addr-card* ⟧ ⟹
heap-update-list p v h q = *v* ! *unat* (*q* − *p*)
apply(*induct v arbitrary*: *p h*; *clarsimp*)
subgoal for *x1 v p h*
apply(*cases p*=*q*)
apply (*simp add*: *heap-update-list-same* [**where** *k*=*1*, *simplified*])
apply(*drule meta-spec* [**where** *x*=*p*+*1*])
apply(*drule meta-spec*, *drule meta-mp*)
apply(*fastforce dest*: *intvl-neq-start*)
by (*smt* (*verit*, *ccfv-SIG*) *Suc-lessD add-diff-cancel-left' diff-add-cancel diff-diff-eq*

diff-zero heap-update-mem-same intvl-neq-start nth-Cons' plus-1-eq-Suc
unat-eq-zero
unat-minus-one)
done

lemma *heap-update-list-value*:
length v < *addr-card* ⟹
heap-update-list p v h q = (*if q* ∈ {*p*..+*length v*} *then v*!*unat* (*q*−*p*) *else h q*)
by (*auto simp*: *heap-update-nmem-same heap-update-mem-same-point*
split: *if-split*)

lemma *heap-update-list-value'*:
length xs < *addr-card* ⟹
heap-update-list ptr xs hp x = (*if unat* (*x* − *ptr*) < *length xs then xs* ! *unat* (*x*
− *ptr*) *else hp x*)
supply *unsigned-of-nat* [*simp del*]
apply (*simp only*: *heap-update-list-value addr-card-def card-word*)
apply (*rule if-cong*; *simp*)
apply (*rule iffI*)
apply (*drule intvlD*, *clarsimp simp add*: *unat-of-nat*)

apply (*simp add*: *intvl-def unat-arith-simps*(*4*) *unat-of-nat split*: *if-split-asm*)
apply (*rule exI* [**where** *x*=*unat x − unat ptr*], *simp*)
apply (*rule exI* [**where** *x*=*unat x + 2^addr-bitsize − unat ptr*])
using *unat-lt2p* [*of ptr*]
apply (*simp add*: *unat-arith-simps unat-of-nat*)
done

lemma *heap-list-h-eq2*:
(⋀*x*. *x* ∈ {*p*..+*n*} ⟹ *h x* = *h*′ *x*) ⟹ *heap-list h n p* = *heap-list h*′ *n p*
apply(*induct n arbitrary*: *p*; *clarsimp*)
apply(*rule conjI*)
apply(*fastforce intro*: *intvl-self*)
apply(*fastforce intro*: *intvl-plus-sub-Suc*)
done

lemma *map-td-f-eq*′:
(*f*=*g*) ⟶ (*map-td f h t* = *map-td g h t*)
(*f*=*g*) ⟶ (*map-td-struct f h st* = *map-td-struct g h st*)
(*f*=*g*) ⟶ (*map-td-list f h ts* = *map-td-list g h ts*)
(*f*=*g*) ⟶ (*map-td-tuple f h x* = *map-td-tuple g h x*)
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *map-td-f-eq*:
f=*g* ⟹ *map-td f t* = *map-td g t*
by (*erule arg-cong*)

lemma *sep-map*′*-lift-exc*:
(*p* ↪$_g$ (*v*::′*a*::*mem-type*)) (*lift-state* (*h*,*d*)) ⟹ *lift h p* = *v*
by (*frule sep-map*′*-lift-typ-heapD*, *simp add*: *lift-t lift-t-lift*)

lemma *sep-map-lift-wp-exc*:
∃ *v*. (*p* ↦$_g$ *v* ∧* (*p* ↦$_g$ *v* ⟶* *P v*)) (*lift-state* (*h*,*d*))
⟹ *P* (*lift h* (*p*::′*a*::*mem-type ptr*)) (*lift-state* (*h*,*d*))
apply *clarsimp*
apply(*subst sep-map*′*-lift-exc*)
apply(*fastforce simp*: *sep-map*′*-def elim*: *sep-conj-impl*)
subgoal for *v*
apply(*rule sep-conj-impl-same* [**where** *P*=*p* ↦$_g$ *v* **and** *Q*=*P v*])
apply(*erule* (*2*) *sep-conj-impl*)
done
done

lemma *sep-map-lift-exc*:
((*p*::′*a*::*mem-type ptr*) ↦$_g$ −) (*lift-state* (*h*,*d*)) ⟹
(*p* ↦$_g$ *lift h p*) (*lift-state* (*h*,*d*))
by (*clarsimp simp*: *sep-map-any-def*)
(*frule sep-map-sep-map*′*-exc*, *drule sep-map*′*-lift-exc*, *simp*)

lemma *sep-map*′*-lift-rev-exc*:

⟦ *lift h p* = (*v*::'*a*::*mem-type*); (*p* ↪$_g$ −) (*lift-state* (*h*,*d*)) ⟧ ⟹
(*p* ↪$_g$ *v*) (*lift-state* (*h*,*d*))
by (*clarsimp simp*: *sep-map'-any-def*)
(*frule sep-map'-lift-exc*, *simp*)

lemma *sep-lift-exists-exc*:
fixes *p* :: '*a*::*mem-type ptr*
assumes *ex*: ((λ*s*. ∃ *v*. (*p* ↪$_g$ *v*) *s* ∧ *P v s*) ∧* *Q*) (*lift-state* (*h*,*d*))
shows (*P* (*lift h p*) ∧* *Q*) (*lift-state* (*h*,*d*))
proof −
from *ex* **obtain** *v* **where** ((λ*s*. (*p* ↪$_g$ *v*) *s* ∧ *P v s*) ∧* *Q*)
(*lift-state* (*h*,*d*))
by (*subst* (*asm*) *sep-conj-exists*, *clarsimp*)
thus *?thesis*
by (*force simp*: *sep-map'-lift-exc sep-conj-ac*
dest: *sep-map'-conjE2-exc dest*!: *sep-conj-conj*)
qed

lemma *merge-dom*:
x ∈ *dom s* ⟹ (*t* ++ *s*) *x* = *s x*
by (*force simp*: *map-add-def*)

lemma *merge-dom2*:
x ∉ *dom s* ⟹ (*t* ++ *s*) *x* = *t x*
by (*force simp*: *map-add-def split*: *option.splits*)

lemma *fs-footprint-empty* [*simp*]:
fs-footprint p {} = {}
by (*auto simp*: *fs-footprint-def*)

lemma *fs-footprint-un*:
fs-footprint p (*insert f F*) = *fs-footprint p* {*f*} ∪ *fs-footprint p F*
by (*auto simp*: *fs-footprint-def*)

lemma *proj-d-restrict-map-le*:
snd (*proj-d* (*s* |‘ *X*) *x*) ⊆$_m$ *snd* (*proj-d s x*)
by(*clarsimp simp*: *map-le-def proj-d-def restrict-map-def*
split: *option.splits if-split-asm*)

lemma *SIndexVal-conj-setcomp-simp* [*simp*]:
{*x*. *snd x* = *SIndexVal* ∧ *x* ∉ *s-footprint-untyped p t*}
= {(*x*,*SIndexVal*) | *x*. *x* ∉ {*p*..+*size-td t*}}
by (*force dest*: *intvlD intro*: *intvlI simp*: *s-footprint-untyped-def*)

lemma *heap-list-s-restrict-same*:
{(*x*,*SIndexVal*) | *x*. *x* ∈ {*p*..+*n*}} ⊆ *X* ⟹ *heap-list-s* (*s* |‘ *X*) *n p* = *heap-list-s s n p*
apply(*induct n arbitrary*: *p*; *clarsimp simp*: *heap-list-s-def*)

apply(*rule conjI*)
apply(*fastforce intro*: *intvl-self simp*: *proj-h-def restrict-map-def*)
apply(*fastforce intro*: *intvl-plus-sub-Suc*)
done

lemma *heap-list-s-restrict-fs-footprint*:
field-lookup (*typ-info-t TYPE*('*a*::*c-type*)) *f 0* = *Some* (*t*,*n*) ⟹
heap-list-s (*s* |‘ *fs-footprint p* {*f*}) (*size-td t*) &(*p*→*f*)
= *heap-list-s s* (*size-td t*) &((*p*::'*a ptr*)→*f*)
apply(*simp add*: *fs-footprint-def field-footprint-def field-offset-def*)
apply(*subst heap-list-s-restrict-same*)
apply(*fastforce simp*: *s-footprint-untyped-def field-size-def field-lvalue-def field-offset-def*
field-ti-def field-typ-def field-typ-untyped-def dest: *intvlD*)
apply *simp*
done

lemma *heap-list-proj-h-disj*:
{(*x*,*SIndexVal*) | *x*. *x* ∈ {*p*..+*n*}} ∩ *dom* s_1 = {} ⟹
heap-list (*proj-h* (s_0 ++ s_1)) *n p* = *heap-list* (*proj-h* s_0) *n p*
apply(*induct n arbitrary*: *p*; *clarsimp*)
apply(*rule conjI*)
apply(*fastforce simp*: *proj-h-def intro*: *intvl-self split*: *option.splits*)
apply(*fastforce intro*: *intvl-plus-sub-Suc*)
done

lemma *heap-list-proj-h-sub*:
{(*x*,*SIndexVal*) | *x*. *x* ∈ {*p*..+*n*}} ⊆ *dom* s_1 ⟹
heap-list (*proj-h* (s_0 ++ s_1)) *n p* = *heap-list* (*proj-h* s_1) *n p*
apply(*induct n arbitrary*: *p*; *clarsimp*)
apply(*rule conjI*, *fastforce simp*: *proj-h-def intro*: *intvl-self split*: *option.splits*)
subgoal premises *prems* **for** *n p*
proof −
have {*p* + *1*..+*n*} ⊆ {*p*..+*Suc n*} **using** *prems* **by** (*fastforce intro*: *intvl-plus-sub-Suc*)
then show *?thesis* **using** *prems* **by** *fast*
qed
done

lemma *heap-list-s-map-add-super-update-bs*:
⟦ {*x*. (*x*,*SIndexVal*) ∈ *dom* s_1} = {*p*+*of-nat k*..+*z*}; *k* + *z* ≤ *n*; *n* < *addr-card* ⟧
⟹ *heap-list-s* (s_0 ++ s_1) *n p* =
super-update-bs (*heap-list-s* s_1 *z* (*p*+*of-nat k*)) (*heap-list-s* s_0 *n p*) *k*
apply(*clarsimp simp*: *super-update-bs-def heap-list-s-def*)
subgoal premises *prems*
proof −
have *heap-list* (*proj-h* (s_0 ++ s_1)) (*k* + *z* + (*n* − (*k*+*z*))) *p* =
take k (*heap-list* (*proj-h* s_0) *n p*) @
heap-list (*proj-h* s_1) *z* (*p* + *of-nat k*) @

drop $(k + z)$ (*heap-list* (*proj-h* s_0) n p)
using *prems*
apply(*subst heap-list-split2*)
apply(*subst heap-list-split2*)
apply *simp*
apply(*rule conjI*)
apply(*subst take-heap-list-le*, *simp*)
apply(*subst heap-list-proj-h-disj*)
using *init-intvl-disj* [*of k z p*]
apply *fastforce*
apply *simp*
apply(*rule conjI*)
apply(*subst heap-list-proj-h-sub*; *fast*)
apply(*simp add*: *drop-heap-list-le*)
apply(*subst heap-list-proj-h-disj*)
using *final-intvl-disj* [*of k z n p*]
apply *fast*
apply *simp*
done
then show *?thesis* **using** *prems* **by** *simp*
qed
done

lemma *s-footprint-untyped-dom-SIndexVal*:
dom s = *s-footprint-untyped p t* ⟹
{*x*. (*x*,*SIndexVal*) ∈ *dom s*} = {*p*..+*size-td t*}
by (*auto simp*: *s-footprint-untyped-def intro*: *intvlI dest*: *intvlD*)

lemma *field-ti-s-sub*:
field-lookup (*export-uinfo* (*typ-info-t TYPE*('*b*::*mem-type*))) *f 0* = *Some* (*a*,*b*) ⟹
s-footprint-untyped &(*p*→*f*) *a* ⊆ *s-footprint* (*p*::'*b ptr*)
apply(*clarsimp simp*: *field-ti-def s-footprint-def s-footprint-untyped-def split*: *option.splits*)
apply(*simp add*: *field-lvalue-def field-offset-def typ-uinfo-t-def*)
subgoal for *x k*
apply(*rule exI* [**where** *x*=*b*+*x*])
apply *simp*
apply(*simp add*: *field-offset-untyped-def*)
apply(*drule td-set-field-lookupD*)
apply(*frule td-set-offset-size*)
apply(*drule typ-slice-td-set* [**where** *k*=*x*])
apply *simp*
apply(*auto simp*: *prefix-def less-eq-list-def*)
done
done

lemma *wf-heap-val-map-add* [*simp*]:
⟦ *wf-heap-val* s_0; *wf-heap-val* s_1 ⟧ ⟹ *wf-heap-val* (s_0 ++ s_1)
unfolding *wf-heap-val-def* **by** *auto*

lemma *of-nat-lt-size-of*:
⟦ (*of-nat x*::*addr*) = *of-nat y* + *of-nat z*; *x* < *size-of TYPE*('*a*::*mem-type*);
y + *z* < *size-of TYPE*('*a*) ⟧ ⟹ *x* = *y*+*z*
by (*metis* (*mono-tags*) *len-of-addr-card less-trans max-size mod-less of-nat-add unat-of-nat*)

lemma *proj-d-map-add*:
snd (*proj-d* s_1 *p*) *n* = *Some k* ⟹ *snd* (*proj-d* (s_0 ++ s_1) *p*) *n* = *Some k*
by (*auto simp*: *proj-d-def split*: *option.splits*)

lemma *proj-d-map-add2*:
fst (*proj-d* s_1 *p*) ⟹ *fst* (*proj-d* (s_0 ++ s_1) *p*)
by (*auto simp*: *proj-d-def split*: *option.splits*)

lemma *heap-list-s-restrict-disj-same*:
dom s ∩ (*UNIV* − *X*) = {} ⟹ *heap-list-s* (*s* |` *X*) *n p* = *heap-list-s s n p*
apply(*induct n arbitrary*: *p*; *clarsimp simp*: *heap-list-s-def*)
apply(*fastforce simp*: *proj-h-def restrict-map-def split*: *option.splits*)
done

lemma *UNIV-minus-inter*:
(*X* − *Y*) ∩ (*X* ∩ (*X* − *Y*) − *Z*) = *X* − (*Y* ∪ *Z*)
by *fast*

lemma *sep-map-mfs-sep-map-empty*:
(*p* ↦$_g$ (*v*::'*a*::*mem-type*)) = (*p* ↦$_g$({}) *v*)
by (*auto simp*: *sep-map-def mfs-sep-map-def map-add-dom-eq*)

lemma *fd-cons-double-update*:
⟦ *fd-cons t*; *length bs* = *length bs'* ⟧ ⟹
update-ti-t t bs (*update-ti-t t bs' v*) = *update-ti-t t bs v*
by (*simp add*: *fd-cons-def Let-def fd-cons-double-update-def fd-cons-desc-def*)

lemma *fd-cons-update-access*:
⟦ *fd-cons t*; *length bs* = *size-td t* ⟧ ⟹
update-ti-t t (*access-ti t v bs*) *v* = *v*
by (*simp add*: *fd-cons-def Let-def fd-cons-update-access-def fd-cons-desc-def*)

lemma *fd-cons-length*:
⟦ *fd-cons t*; *length bs* = *size-td t* ⟧ ⟹
length (*access-ti t v bs*) = *size-td t*
by (*simp add*: *fd-cons-def Let-def fd-cons-desc-def fd-cons-length-def access-ti*$_0$*-def*)

lemma *fd-cons-length-p*:
fd-cons t ⟹ *length* (*access-ti*$_0$ *t v*) = *size-td t*
by (*simp add*: *fd-cons-length access-ti*$_0$*-def*)

lemma *fd-cons-update-normalise*:

⟦ *fd-cons t*; *length bs* = *size-td t* ⟧ ⟹
update-ti-t t ((*norm-desc* (*field-desc t*) (*size-td t*)) *bs*) *v* = *update-ti-t t bs v*
by (*fastforce simp*: *fd-cons-def Let-def fd-cons-desc-def fd-cons-update-normalise-def*
dest: *fd-cons-update-normalise*)

lemma *field-footprint-SIndexVal*:
field-lookup (*typ-info-t TYPE*('*a*::*c-type*)) *f 0* = *Some* (*t*, *n*) ⟹
{*x*. (*x*, *SIndexVal*) ∈ *field-footprint* (*p*::'*a ptr*) *f*} =
{*ptr-val p* + *of-nat n*..+*size-td t*}
by (*auto simp*: *field-footprint-def s-footprint-untyped-def field-typ-def field-typ-untyped-def*
intro: *intvlI dest*: *intvlD*)

lemma *fs-footprint-subset*:
F ⊆ *fields TYPE*('*a*::*mem-type*) ⟹ *fs-footprint* (*p*::'*a ptr*) *F* ⊆ *s-footprint p*
unfolding *fs-footprint-def field-footprint-def*
apply(*clarsimp simp*: *fields-def*)
apply(*drule* (*1*) *subsetD*, *clarsimp*)
apply(*frule field-lookup-export-uinfo-Some*)
apply(*drule field-ti-s-sub*)
apply(*unfold field-lvalue-def*)[*1*]
apply(*subst* (*asm*) *field-lookup-offset-eq*, *assumption*)
apply(*fastforce simp*: *field-typ-def field-typ-untyped-def*)
done

lemma *length-heap-list-s* [*simp*]:
length (*heap-list-s s n p*) = *n*
by (*clarsimp simp*: *heap-list-s-def*)

lemma *heap-list-proj-h-restrict*:
{*p*..+*n*} ⊆ {*x*. (*x*,*SIndexVal*) ∈ *X*} ⟹ *heap-list* (*proj-h* (*s* |` *X*)) *n p* = *heap-list*
(*proj-h s*) *n p*
apply(*induct n arbitrary*: *p*; *clarsimp*)
apply(*rule conjI*)
apply(*fastforce simp*: *proj-h-restrict intro*: *intvl-self*)
apply(*fastforce intro*: *intvl-plus-sub-Suc*)
done

lemma *heap-list-proj-h-lift-state*:
{*p*..+*n*} ⊆ {*x*. *fst* (*d x*)} ⟹ *heap-list* (*proj-h* (*lift-state* (*h*,*d*))) *n p* = *heap-list h*
n p
by (*fastforce intro*: *heap-list-h-eq2 simp*: *proj-h-lift-state*)

lemma *heap-list-rpbs*:
heap-list (λ*x*. *0*) *n p* = *replicate n 0*
by (*induct n arbitrary*: *p*) *auto*

lemma *field-access-take-drop*:
fixes
t::('*a*,'*b*) *typ-info* **and**

```
  st::('a,'b) typ-info-struct and
  ts::('a,'b) typ-info-tuple list and
  x::('a,'b) typ-info-tuple
 shows
 ∀ s m n f. field-lookup t f m = Some (s,n) ⟶ wf-fd t ⟶
     take (size-td s) (drop (n − m) (access-ti₀ t v)) =
       access-ti₀ s v
 ∀ s m n f. field-lookup-struct st f m = Some (s,n) ⟶ wf-fd-struct st ⟶
     take (size-td s) (drop (n − m) (access-ti-struct st v (replicate (size-td-struct
st) 0))) =
       access-ti₀ s v
 ∀ s m n f. field-lookup-list ts f m = Some (s,n) ⟶ wf-fd-list ts ⟶
    take (size-td s) (drop (n − m) (access-ti-list ts v (replicate (size-td-list ts) 0)))
=
       access-ti₀ s v
 ∀ s m n f. field-lookup-tuple x f m = Some (s,n) ⟶ wf-fd-tuple x ⟶
     take (size-td s) (drop (n − m) (access-ti-tuple x v (replicate (size-td-tuple x)
0))) =
       access-ti₀ s v
   apply(induct t and st and ts and x, all ‹clarsimp simp: access-ti₀-def›)
  apply(fastforce dest: wf-fd-cons-structD
    simp: fd-cons-struct-def fd-cons-desc-def fd-cons-length-def)
  apply(drule wf-fd-cons-tupleD)
  apply(clarsimp simp: fd-cons-tuple-def fd-cons-desc-def fd-cons-length-def)
  apply(clarsimp split: option.splits)
   apply(subst drop-all)
  apply(fastforce dest: field-lookup-offset-le simp: fd-cons-length-def split-DTuple-all)
   apply simp
  apply (metis (no-types, opaque-lifting) add.commute add-le-imp-le-diff diff-is-0-eq'

    diff-zero field-lookup-offset-le(2) fl2 size-td-tuple-dt-fst)
  apply (metis (no-types, lifting) Nat.diff-diff-right add-leD2 append.right-neutral

    diff-is-0-eq length-0-conv length-take nat-min-simps(1)
    td-set-offset-size' td-set-tuple-field-lookup-tupleD zero-le)
 apply fastforce
 done

lemma field-access-take-dropD:
 ⟦ field-lookup t f 0 = Some (s,n); wf-lf (lf-set t []); wf-desc t ⟧ ⟹
    take (size-td s) (drop n (access-ti₀ t v)) = access-ti₀ s v
 using field-access-take-drop(1) [of t v] by (fastforce dest: wf-fdp-fdD wf-lf-fdp)

lemma singleton-t-field:
 field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some (t, n) ⟹
    heap-list-s (singleton-t p v |` fs-footprint p {f}) (size-td t) (ptr-val p + of-nat
n) =
    access-ti₀ t v
 apply(clarsimp simp: heap-list-s-def singleton-def singleton-t-def)
```

apply(*subst heap-list-proj-h-restrict*)
apply(*fastforce simp*: *fields-def fs-footprint-def*
intro!: *fs-footprint-subset*
dest: *field-footprint-SIndexVal*)
apply(*subst heap-list-proj-h-lift-state*)
apply(*frule field-tag-sub*[**where** *p*=*p*])
apply(*fastforce simp*: *field-lvalue-def dest*: *ptr-retyp-footprint*[**where** *d*=*empty-htd*])
apply(*clarsimp simp*: *access-ti*$_0$*-def heap-update-def*)
apply(*subst heap-list-update-list*; *simp?*)
apply(*simp add*: *size-of-def*)
apply(*erule field-lookup-offset-size*)
apply(*fastforce simp*: *access-ti*$_0$*-def to-bytes-def heap-list-rpbs size-of-def*
dest: *field-access-take-dropD*
elim: *field-lookup-offset-size*)
done

lemma *field-lookup-fd-consD*:
field-lookup (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*t*,*n*) ⟹ *fd-cons t*
by (*erule fd-consistentD*) *simp*

lemma *s-valid-map-add*:
⟦ *s*,*g* ⊨$_s$ *p*; *t*,*g'* ⊨$_s$ *p* ⟧ ⟹ (*s* ++ *t* |` *X*),*g* ⊨$_s$ *p*
by (*clarsimp simp*: *map-le-def s-valid-def h-t-valid-def valid-footprint-def Let-def*
proj-d-map-add-snd proj-d-restrict-map-snd proj-d-map-add-fst
proj-d-restrict-map-fst)

lemma *singleton-t-s-valid*:
g p ⟹ *singleton-t p* (*v*::'*a*::*mem-type*),*g* ⊨$_s$ *p*
by (*fastforce simp*: *singleton-t-def h-t-s-valid elim*: *ptr-retyp-h-t-valid*)

lemma *sep-map-mfs-sep-map*:
⟦ (*p* ↦$_g{}^F$ *v*) *s*; *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*) ⟧ ⟹
(*p* ↦$_g{}^{(\{f\}\cup F)}$ (*v*::'*a*::*mem-type*)) (*s* |` (*dom s* − *fs-footprint p* {*f*}))
apply(*clarsimp simp*: *mfs-sep-map-def*)
apply(*rule conjI*)
prefer *2*
apply(*fastforce simp*: *fs-footprint-un*[**where** *F*=*F*] *fields-def*)
apply(*clarsimp simp*: *lift-typ-heap-if split*: *if-split-asm*)
apply(*rule conjI*, *clarsimp*)
subgoal premises *prems*
proof −
have (*singleton-t p v* ++
s |` (*s-footprint p* − *fs-footprint p F* − *fs-footprint p* {*f*})) =
(*singleton-t p v* ++ *s*) ++ (*singleton-t p v* |` *fs-footprint p* {*f*})
using *prems*
apply(*simp add*: *map-add-restrict-sub*)
done
then show *?thesis*
using *prems*

```
    apply clarsimp
    apply(subst heap-list-s-map-add-super-update-bs)
     apply clarsimp
     apply(subgoal-tac fs-footprint p {f} ⊆ s-footprint p)
      apply(subgoal-tac {x. (x, SIndexVal) ∈ fs-footprint p {f}} =
                        {ptr-val p + of-nat n..+size-td t})
       apply fast
      apply(clarsimp simp: fs-footprint-def)
      apply(erule field-footprint-SIndexVal)
     apply(rule fs-footprint-subset)
     apply(clarsimp simp: fields-def)
    apply(clarsimp simp: size-of-def)
    apply(subst ac-simps)
    apply(rule td-set-offset-size)
    apply(erule td-set-field-lookupD)
   apply simp
   apply(clarsimp simp: from-bytes-def)
   apply(frule-tac v=heap-list-s (singleton-t p v |` fs-footprint p {f}) (size-td t)
                                 (ptr-val p + of-nat n) and
     bs=heap-list-s (singleton-t p v ++ s) (size-of TYPE('a)) (ptr-val p) and
     w=undefined in fi-fu-consistentD; simp)
    apply(simp add: size-of-def)
   apply(simp add: singleton-t-field)
  apply(fastforce simp: access-ti0-def fd-cons-update-access dest!: field-lookup-fd-consD)
   done
 qed
 subgoal premises prems
 proof −
  have (singleton-t p v ++
                s |` (s-footprint p − fs-footprint p F − fs-footprint p {f})) =
               (singleton-t p v ++ s) ++ (singleton-t p v |` fs-footprint p {f})
   using prems by (simp add: map-add-restrict-sub)
  then show ?thesis using prems
   by(fastforce intro: s-valid-map-add singleton-t-s-valid ptr-retyp-h-t-valid)
 qed
 done

lemma disjoint-fn-disjoint:
 ⟦ disjoint-fn f F; F ⊆ fields TYPE('a::mem-type);
    field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n) ⟧ ⟹
    fs-footprint (p::'a ptr) F ∩ field-footprint p f = {}
  apply(clarsimp simp: fs-footprint-def field-footprint-def s-footprint-untyped-def
field-typ-def
                     field-typ-untyped-def fields-def)
 apply safe
 apply (drule (1) subsetD, clarsimp,
       drule (1) fa-fu-lookup-disj-interD;
           force intro: intvlI simp: max-size[unfolded size-of-def] disj-fn-def dis-
joint-fn-def)
```

apply (*drule* (*1*) *subsetD*, *clarsimp*,
drule (*1*) *fa-fu-lookup-disj-interD*;
force intro: *intvlI simp*: *max-size*[*unfolded size-of-def*] *disj-fn-def disjoint-fn-def*)
done

lemma *sep-map-mfs-sep-map2*:
⟦*field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*, *n*);
disjoint-fn f F; *guard-mono g g'*;
export-uinfo s = *typ-uinfo-t TYPE*('*b*); ((*p*::'*a ptr*) ↦$_g{}^F$ *v*) *x*⟧
⟹ (*Ptr* &(*p*→*f*) ↦$_g{}'$ ((*from-bytes* (*access-ti*$_0$ *s v*))::'*b*::*mem-type*))
(*x* |` *field-footprint p f*)
apply(*clarsimp simp*: *mfs-sep-map-def sep-map-def*)
apply(*rule conjI*)
subgoal premises *prems*
proof −
have *field-footprint p f* = *s-footprint* ((*Ptr* &(*p*→*f*))::'*b ptr*)
using *prems*
by(*clarsimp simp*: *field-footprint-def s-footprint-def field-lvalue-def typ-uinfo-t-def field-typ-def field-typ-untyped-def*)
then show *?thesis*
using *prems*
apply *simp*
apply(*frule lift-typ-heap-mono*, *assumption*+)
apply(*clarsimp simp*: *lift-typ-heap-if split*: *if-split-asm*)
apply(*rule conjI*, *fastforce simp*: *heap-list-s-heap-merge-right*[**where** *p*=&(*p*→*f*)])
apply(*erule s-valid-heap-merge-right2*)
apply *simp*
apply(*frule* (*2*) *disjoint-fn-disjoint*[**where** *p*=*p*])
apply(*subgoal-tac fs-footprint p* {*f*} ⊆ *s-footprint p*)
apply(*fastforce simp*: *fs-footprint-def*)
apply(*rule fs-footprint-subset*)
apply(*fastforce simp*: *fields-def*)
done
qed
apply(*clarsimp simp*: *field-footprint-def field-lvalue-def s-footprint-def field-typ-def field-typ-untyped-def*)
subgoal premises *prems*
proof −
have {*f*} ⊆ *fields TYPE*('*a*)
using *prems* **by**(*clarsimp simp*: *fields-def*)
then show *?thesis*
using *prems*
apply −
apply(*drule fs-footprint-subset*[**where** *F*={*f*} **and** *p*=*p*])
apply(*subst* (*asm*) (*2*) *fs-footprint-def*)
apply(*clarsimp simp*: *field-footprint-def s-footprint-def field-typ-def field-typ-untyped-def*)
apply(*subgoal-tac fs-footprint p F* ∩
s-footprint-untyped (*ptr-val p* + *of-nat n*) (*typ-uinfo-t TYPE*('*b*))

```
= {})
      apply blast
     apply(drule disjoint-fn-disjoint[where p=p], assumption+)
     apply(simp add: field-footprint-def field-typ-def field-typ-untyped-def)
     done

  qed
  done

lemma export-size-of:
  export-uinfo t = typ-uinfo-t TYPE('a) ⟹ size-of TYPE('a::c-type) = size-td t
  by (fastforce simp: size-of-def simp flip: typ-uinfo-size dest: sym)

lemma sep-map-field-unfold:
  ⟦ field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n);
      disjoint-fn f F; guard-mono g g';
      export-uinfo t = typ-uinfo-t TYPE('b) ⟧ ⟹
      (p ↦_g^F v) = (p ↦_g^(−{f}∪F) (v::'a::mem-type) ∧*
          Ptr (&(p→f)) ↦_g' ((from-bytes (access-ti_0 t v))::'b::mem-type))
  supply unsigned-of-nat [simp del]
  apply(rule ext)
  apply(rule iffI)
  subgoal for x
    apply(rule sep-conjI [where  s_0=x |` (dom x − fs-footprint p {f}) and
          s_1=x |` fs-footprint p {f}] )
      apply(erule (1) sep-map-mfs-sep-map)
     apply(clarsimp simp: fs-footprint-def)
     apply(erule (4) sep-map-mfs-sep-map2)
    apply(clarsimp simp: map-disj-def)
    apply fast
   apply clarsimp
   done

  apply(drule sep-conjD, clarsimp)
  apply(clarsimp simp: mfs-sep-map-def sep-map-def)
  apply(rule conjI)
   apply(subst map-ac-simps)
   apply(subst map-add-com, solves ‹simp add: map-ac-simps›)
   apply(subst map-add-assoc)
   apply(clarsimp simp: lift-typ-heap-if split: if-split-asm)
   apply(rule conjI, clarsimp)
    apply(subst heap-list-s-map-add-super-update-bs)
      apply(subst s-footprint-untyped-dom-SIndexVal)
       apply(fastforce simp: s-footprint-def)
      apply(fastforce simp: field-lvalue-def)
     apply(drule field-lookup-offset-size)
     apply(drule export-size-of)
     apply(simp add: size-of-def)
    apply simp
```

apply(*clarsimp simp*: *from-bytes-def*)
subgoal for s_0 s_1
apply(*frule fi-fu-consistentD* [**where** *v*=*heap-list-s* s_1 (*size-td* (*typ-info-t TYPE*('*b*)))
(*ptr-val p* + *of-nat n*) **and**
bs=*heap-list-s* (*singleton-t p v* ++ s_0) (*size-of TYPE*('*a*)) (*ptr-val p*) **and**
w=*undefined*]; *simp*)
apply(*simp add*: *size-of-def*)
apply(*drule export-size-of*, *simp add*: *size-of-def*)
apply(*subst fd-cons-update-normalise* [*symmetric*])
apply(*erule field-lookup-fd-consD*)
apply *simp*
apply(*drule export-size-of*, *simp add*: *size-of-def*)
apply(*simp add*: *norm-desc-def*)
apply(*drule arg-cong* [**where** *f*=*access-ti*$_0$ (*typ-info-t TYPE*('*b*))])
apply(*drule arg-cong* [**where** *f*=λ*bs*. *update-ti-t t bs v*])
apply(*subst* (*asm*) *wf-fd-norm-tuD* [*symmetric*]; *simp?*)
apply(*simp add*: *size-of-def*)
apply(*subst* (*asm*) *wf-fd-norm-tuD* [*symmetric*])
apply *simp*
apply(*subst fd-cons-length-p*)
apply(*erule field-lookup-fd-consD*)
apply(*drule export-size-of*, *simp add*: *size-of-def*)
apply(*subgoal-tac export-uinfo* (*typ-info-t TYPE*('*b*)) = *typ-uinfo-t TYPE*('*b*))
prefer *2*
apply(*simp add*: *typ-uinfo-t-def*)
apply *simp*
apply(*drule sym*, *simp*)
apply(*subst* (*asm*) *wf-fd-norm-tuD*)
apply(*erule wf-fd-field-lookupD*, *simp*)
apply *simp*
apply(*drule sym*, *drule export-size-of*)
apply(*simp add*: *size-of-def*)
apply(*simp add*: *access-ti*$_0$*-def*)
apply(*clarsimp simp*: *field-lvalue-def*)
apply(*simp add*: *size-of-def*)
apply(*subst wf-fd-norm-tuD*)
apply(*erule wf-fd-field-lookupD*, *simp*)
apply(*subst fd-cons-length*; *simp?*)
apply(*erule field-lookup-fd-consD*)
apply(*subgoal-tac update-ti-t t* (*norm-desc* (*field-desc t*) (*size-td t*)
(*access-ti t v* (*replicate* (*size-td t*) *0*))) *v* = *v*)
apply(*simp add*: *norm-desc-def access-ti*$_0$*-def*)
apply(*subst fd-cons-update-normalise*)
apply(*erule field-lookup-fd-consD*)
apply(*subst fd-cons-length*; *simp?*)
apply(*erule field-lookup-fd-consD*)
apply(*subst fd-cons-update-access*; *simp?*)
apply(*erule field-lookup-fd-consD*)
done

prefer *2*
apply(*subst fs-footprint-un*)
apply(*subst fs-footprint-def*)
apply(*clarsimp simp*: *field-footprint-def s-footprint-def field-lvalue-def field-typ-def*)
apply(*drule disjoint-fn-disjoint* [**where** *p*=*p*]; *assumption?*)
apply(*clarsimp simp*: *field-footprint-def s-footprint-def field-lvalue-def field-typ-def*)
apply(*subgoal-tac* $\{f\} \subseteq$ *fields TYPE*('*a*))
apply(*drule fs-footprint-subset*[**where** *F*=$\{f\}$ **and** *p* = *p*])
apply(*clarsimp simp*: *s-footprint-def*)
apply(*fastforce simp*: *fs-footprint-def field-footprint-def s-footprint-def field-typ-def field-typ-untyped-def field-lvalue-def*)
apply(*clarsimp simp*: *fields-def*)
apply(*clarsimp simp*: *s-valid-def h-t-valid-def valid-footprint-def Let-def*)

subgoal for s_0 s_1 *y*
apply(*standard*, *clarsimp simp*: *map-le-def*)
subgoal for *a*
apply(*subst proj-d-map-add-snd*[**where** *s*=*a* ++ *b* **for** *a b*])
apply(*clarsimp split*: *if-split-asm*)
apply(*frule s-footprintD2*)
apply(*drule s-footprintD*)
apply(*drule spec* [**where** *x*=*y*])
apply *clarsimp*
apply(*drule bspec* [**where** *x*=*a*])
apply *clarsimp*
apply(*drule intvlD*, *clarsimp simp*: *field-lvalue-def*)
subgoal for *k*
apply(*drule spec* [**where** *x*=*k*])
apply(*clarsimp simp add*: *size-of-def*)
apply(*drule bspec* [**where** *x*=*a*])
apply *clarsimp*
apply(*subst* (*asm*) *unat-of-nat*)
apply(*subst* (*asm*) *mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply(*simp add*: *max-size*[*unfolded size-of-def*])
apply *simp*
apply(*simp add*: *ac-simps*)
apply(*rotate-tac* −*1*)
apply(*drule sym*)
apply *simp*
apply(*drule sym*[**where** *s*=*Some s* **for** *s*])
apply *simp*
apply(*drule field-lookup-export-uinfo-Some*)
apply(*drule td-set-field-lookupD*)
apply(*frule typ-slice-td-set* [**where** *k*=*k*])
apply *simp*
apply *simp*

apply(*simp add*: *typ-uinfo-t-def*)
apply(*subgoal-tac* *y=n+k*)
apply(*simp add*: *strict-prefix-def*)
apply(*subst* (*asm*) *unat-of-nat*)
apply(*subst* (*asm*) *mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply(*simp add*: *max-size*[*unfolded size-of-def*])
apply (*clarsimp simp*: *prefix-eq-nth*)
apply(*drule arg-cong* [**where** *f=unat*])
apply(*rotate-tac* *−1*)
apply(*subst* (*asm*) *unat-of-nat*)
apply(*subst* (*asm*) *mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply(*simp add*: *max-size*[*unfolded size-of-def*])
apply(*subst* (*asm*) *Abs-fnat-hom-add*)
apply(*subst* (*asm*) *unat-of-nat*)
apply(*subst* (*asm*) *mod-less*)
apply(*subst len-of-addr-card*)
apply(*drule td-set-offset-size*)
apply(*rule le-less-trans* [**where** *y=size-td* (*typ-info-t TYPE*(*'a*))] , *simp*)
apply(*simp add*: *max-size*[*unfolded size-of-def*])
apply *simp*
done
done

apply(*subst proj-d-map-add-fst*)
apply(*fastforce simp*: *size-of-def field-lvalue-def ac-simps*
dest: *intvlD s-footprintD*)
done
done

lemma *disjoint-fn-empty* [*simp*]:
disjoint-fn f {}
by (*simp add*: *disjoint-fn-def*)

lemma *sep-map-field-map'*:
⟦ ((*p*::*'a*::*mem-type ptr*) $\mapsto_g$ *v*) *s*;
field-lookup (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*d*,*n*); *export-uinfo d* = *typ-uinfo-t TYPE*(*'b*);
guard-mono g g' ⟧ ⟹
((*Ptr* (&(*p*→*f*))::*'b*::*mem-type ptr*) $\hookrightarrow_g$*' from-bytes* (*access-ti*$_0$ *d v*)) *s*
by (*fastforce dest*: *sep-map-g elim*: *sep-conj-impl*
simp: *sep-map-mfs-sep-map-empty sep-map-field-unfold sep-map'-def sep-conj-ac*)

lemma *fd-cons-access-update-p*:
⟦ *fd-cons t*; *length bs* = *size-td t* ⟧ ⟹

access-ti$_0$ *t* (*update-ti-t t bs v*) = *access-ti*$_0$ *t* (*update-ti-t t bs w*)
by (*simp add*: *fd-cons-def Let-def fd-cons-access-update-def fd-cons-desc-def access-ti*$_0$*-def*)

lemma *length-to-bytes-p* [*simp*]:
length (*to-bytes-p* (*v*::'*a*)) = *size-of TYPE*('*a*::*mem-type*)
by (*simp add*: *to-bytes-p-def*)

lemma *inv-p* [*simp*]:
from-bytes (*to-bytes-p v*) = (*v*::'*a*::*mem-type*)
by (*simp add*: *to-bytes-p-def*)

lemma *singleton-SIndexVal*:
x ∈ {*ptr-val p*..+*size-of TYPE*('*a*)} ⟹
singleton-t p (*v*::'*a*::*mem-type*) (*x*,*SIndexVal*) = *Some* (*SValue* (*to-bytes-p v* ! *unat* (*x* − *ptr-val p*)))
by (*clarsimp simp*: *singleton-def singleton-t-def lift-state-def heap-update-def heap-update-mem-same-point to-bytes-p-def heap-list-rpbs ptr-retyp-d-eq-fst*)

lemma *access-ti*$_0$:
access-ti s v (*replicate* (*size-td s*) *0*) = *access-ti*$_0$ *s v*
by (*simp add*: *access-ti*$_0$*-def*)

lemma *fd-cons-mem-type* [*simp*]:
fd-cons (*typ-info-t TYPE*('*a*::*mem-type*))
by (*rule wf-fd-consD*) *simp*

lemma *norm-tu-rpbs*:
wf-fd t ⟹ *norm-tu* (*export-uinfo t*) (*access-ti*$_0$ *t v*) = *access-ti*$_0$ *t v*
apply(*frule wf-fd-consD*)
apply(*simp add*: *wf-fd-norm-tuD fd-cons-length-p*)
apply(*subst fd-cons-access-update-p* [**where** *w*=*v*]; (*simp add*: *fd-cons-length-p*)*?*)
apply(*fastforce simp*: *access-ti*$_0$*-def fd-cons-update-access*)
done

lemma *heap-list-s-singleton-t-field-update*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*, *n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
heap-list-s (*singleton-t p* (*update-ti-t s* (*to-bytes-p w*) *v*)) (*size-td s*)
(*ptr-val* (*p*::'*a*::*mem-type ptr*) + *of-nat n*) =
to-bytes-p (*w*::'*b*::*mem-type*)
apply(*clarsimp simp*: *singleton-t-def singleton-def*)
apply(*subst heap-list-s-heap-list-dom*)
apply *clarsimp*
apply(*frule field-tag-sub* [**where** *p*=*p*])
apply(*clarsimp simp*: *field-lvalue-def*)
apply(*drule* (*1*) *subsetD*)
apply(*drule intvlD* [**where** *n*=*size-of TYPE*('*a*)], *clarsimp*)

apply(*erule s-footprintI2*)
apply(*simp add*: *heap-update-def*)
apply(*subst heap-list-update-list*; *simp?*)
apply(*drule field-lookup-offset-size*)
apply(*simp add*: *size-of-def*)
apply(*frule field-access-take-dropD* [**where** *v*=(*update-ti-t s* (*to-bytes-p w*) *v*)]; *simp?*)
apply(*simp add*: *access-ti*$_0$*-def to-bytes-def heap-list-rpbs size-of-def to-bytes-p-def*)
apply(*simp add*: *access-ti*$_0$)
apply(*subst fd-cons-access-update-p* [**where** *w*=*undefined*])
apply(*erule field-lookup-fd-consD*)
apply(*subst fd-cons-length-p*)
apply *simp*
apply(*drule export-size-of*, *simp add*: *size-of-def*)
apply(*subst wf-fd-norm-tuD* [*symmetric*])
apply(*erule wf-fd-field-lookupD*)
apply *simp*
apply(*fastforce simp*: *fd-cons-length-p size-of-def dest*: *export-size-of*)
apply(*simp add*: *typ-uinfo-t-def norm-tu-rpbs*)
done

lemma *field-access-update-nth-disj*:
fixes
t::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
ts::('*a*,'*b*) *typ-info-tuple list* **and**
y::('*a*,'*b*) *typ-info-tuple*
shows
$\forall$ *m f s n x bs bs'*. *field-lookup t f m* = *Some* (*s*,*n*) $\longrightarrow$ *x* < *size-td t* $\longrightarrow$
(*x* < *n* − *m* $\vee$ *x* $\geq$ (*n* − *m*) + *size-td s*) $\longrightarrow$ *wf-fd t* $\longrightarrow$ *length bs* = *size-td s* $\longrightarrow$ *length bs'* = *size-td t* $\longrightarrow$
access-ti t (*update-ti-t s bs v*) *bs'* ! *x*
= *access-ti t v bs'* ! *x*
$\forall$ *m f s n x bs bs'*. *field-lookup-struct st f m* = *Some* (*s*,*n*) $\longrightarrow$ *x* < *size-td-struct st* $\longrightarrow$
(*x* < *n* − *m* $\vee$ *x* $\geq$ (*n* − *m*) + *size-td s*) $\longrightarrow$ *wf-fd-struct st* $\longrightarrow$*length bs* = *size-td s* $\longrightarrow$ *length bs'* = *size-td-struct st* $\longrightarrow$
access-ti-struct st (*update-ti-t s bs v*) *bs'* ! *x*
= *access-ti-struct st v bs'* ! *x*
$\forall$ *m f s n x bs bs'*. *field-lookup-list ts f m* = *Some* (*s*,*n*) $\longrightarrow$ *x* < *size-td-list ts* $\longrightarrow$
(*x* < *n* − *m* $\vee$ *x* $\geq$ (*n* − *m*) + *size-td s*) $\longrightarrow$ *wf-fd-list ts* $\longrightarrow$*length bs* = *size-td s* $\longrightarrow$ *length bs'* = *size-td-list ts* $\longrightarrow$
access-ti-list ts (*update-ti-t s bs v*) *bs'* ! *x*
= *access-ti-list ts v bs'* ! *x*
$\forall$ *m f s n x bs bs'*. *field-lookup-tuple y f m* = *Some* (*s*,*n*) $\longrightarrow$ *x* < *size-td-tuple y* $\longrightarrow$
(*x* < *n* − *m* $\vee$ *x* $\geq$ (*n* − *m*) + *size-td s*) $\longrightarrow$ *wf-fd-tuple y* $\longrightarrow$*length bs* = *size-td s* $\longrightarrow$ *length bs'* = *size-td-tuple y* $\longrightarrow$
access-ti-tuple y (*update-ti-t s bs v*) *bs'* ! *x*

```
        = access-ti-tuple y v bs' ! x
proof(induct t and st and ts and y)
  case (TypDesc nat typ-struct list)
  then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc dt-tuple list)
  then show ?case
    apply clarify
    subgoal for m f s n x bs bs'
      apply(clarsimp split: if-split-asm)
      apply(clarsimp split: option.splits)

       apply(rotate-tac -3)
       apply(drule spec [where x=m + size-td (dt-fst dt-tuple)])
       apply(drule spec [where x=f])
       apply(drule spec [where x=s])
       apply(rotate-tac -1)
       apply(drule spec [where x=n])

       apply clarsimp
       apply(rotate-tac -1)
       apply(drule-tac spec [where x=x - size-td-tuple dt-tuple])
       apply(frule field-lookup-fa-fu-rhs-listD)
        apply simp
       apply assumption
       apply(clarsimp simp: fa-fu-ind-def)
     apply(subgoal-tac access-ti-tuple dt-tuple (update-ti-t s bs v) (take (size-td-tuple
dt-tuple) bs') =
                    access-ti-tuple dt-tuple v (take (size-td-tuple dt-tuple) bs'))
       prefer 2
       apply (fastforce simp: min-def)
      apply(clarsimp simp: nth-append)
      apply(subgoal-tac length
              (access-ti-tuple dt-tuple v
                (take (size-td-tuple dt-tuple) bs')) = size-td-tuple dt-tuple)
       apply simp
       prefer 2
       apply(drule wf-fd-cons-tupleD)
       apply(clarsimp simp: fd-cons-tuple-def fd-cons-desc-def fd-cons-length-def)
      apply(erule impE)
```

```
      apply simp
     apply(cases dt-tuple, simp+)
    subgoal for a b c

     apply(drule spec [where x=bs])
     apply(drule spec [where x=drop (size-td a) bs'])
     apply clarsimp
     apply(frule field-lookup-offset-le)
     apply clarsimp
     apply(drule td-set-list-field-lookup-listD)
     apply(drule td-set-list-offset-size-m)
     apply clarsimp
     apply(erule disjE)
      apply arith
     apply arith
     done
    apply(frule field-lookup-fa-fu-rhs-tupleD, simp)
     apply assumption
    apply(clarsimp simp: fa-fu-ind-def)
    apply(subgoal-tac length (access-ti-tuple dt-tuple (update-ti-t s bs v)
                        (take (size-td-tuple dt-tuple) bs')) = size-td-tuple dt-tuple)
     apply(subgoal-tac length (access-ti-tuple dt-tuple v
                         (take (size-td-tuple dt-tuple) bs')) = size-td-tuple dt-tuple)
      apply(clarsimp simp: nth-append)
      apply(drule spec [where x=m])
      apply(drule spec [where x=f])
      apply(drule spec [where x=s])
      apply(drule spec [where x=n])
      apply clarsimp
      apply(drule spec [where x=x])
      apply clarsimp
      apply(drule spec [where x=bs])
      apply(drule spec [where x=take (size-td-tuple dt-tuple) bs'])
      apply(clarsimp simp: min-def split: if-split-asm)
     apply(drule wf-fd-cons-tupleD)
     apply(clarsimp simp: fd-cons-tuple-def fd-cons-desc-def fd-cons-length-def)
    apply(drule wf-fd-cons-tupleD)
    apply(clarsimp simp: fd-cons-tuple-def fd-cons-desc-def fd-cons-length-def)
    done
   done
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case by auto
qed


lemma field-access-update-nth-disjD:
  ⟦ field-lookup t f m = Some (s,n); x < size-td t;
     (x < n − m ∨ x ≥ (n − m) + size-td s);  wf-fd t;
```

length bs = size-td s; *length bs′ = size-td t* ⟧ ⟹
access-ti t (*update-ti-t s bs v*) *bs′* ! *x*
= *access-ti t v bs′* ! *x*
by (*simp add*: *field-access-update-nth-disj*)

lemma *intvl-cut*:
⟦ (*x*::*addr*) ∈ {*p*..+*m*}; *x* ∉ {*p*+*of-nat k*..+*n*}; *m* < *addr-card* ⟧ ⟹
unat (*x* − *p*) < *k* ∨ *k* + *n* ≤ *unat* (*x* − *p*)
supply *unsigned-of-nat* [*simp del*]
apply(*drule intvlD*, *clarsimp*)
apply(*subst unat-of-nat*, *subst mod-less*, *subst len-of-addr-card*)
apply(*erule* (*1*) *less-trans*)
apply(*subst* (*asm*) *unat-of-nat*, *subst* (*asm*) *mod-less*, *subst len-of-addr-card*)
apply(*erule* (*1*) *less-trans*)
apply(*rule ccontr*)
apply(*subgoal-tac* ∃ *z*. *ka* = *k* + *z*)
apply(*force simp flip*: *add.assoc intro*: *intvlI*)
apply *arith*
done

lemma *singleton-t-mask-out*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*,*n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*);
K = (*UNIV* − *s-footprint-untyped* &(*p*→*f*) (*export-uinfo s*)) ⟧ ⟹
singleton-t p (*update-ti-t s* (*to-bytes-p* (*w*::'*b*::*mem-type*)) (*v*::'*a*)) |‘ *K* =
singleton-t p v |‘ *K*
supply *unsigned-of-nat* [*simp del*]
supply *max-size*[*unfolded size-of-def*, *simp*]
apply(*rule ext*)
apply(*clarsimp simp*: *restrict-map-def singleton-t-def singleton-def lift-state-def*
heap-update-def to-bytes-def access-ti$_0$ *heap-list-rpbs size-of-def*
split: *s-heap-index.splits*)
apply(*subst heap-update-mem-same-point*)
apply(*fastforce simp*: *fd-cons-length-p ptr-retyp-None size-of-def intro*: *ccontr*)
apply(*simp add*: *fd-cons-length-p*)
apply(*subst heap-update-mem-same-point*)
apply(*fastforce simp*: *fd-cons-length-p ptr-retyp-None size-of-def intro*: *ccontr*)
apply(*simp add*: *fd-cons-length-p*)
apply(*simp add*: *access-ti*$_0$*-def*)
apply(*rule field-access-update-nth-disjD*; *simp?*)
apply(*subst* (*asm*) *ptr-retyp-d-eq-fst*)
apply(*clarsimp simp*: *empty-htd-def split*: *if-split-asm*)
apply(*drule intvlD*, *clarsimp*)
apply(*subst unat-of-nat*)
apply(*subst mod-less*)
apply(*subst len-of-addr-card*)
apply(*erule less-trans*)
apply *simp*
apply(*simp add*: *size-of-def*)

apply(*clarsimp simp*: *empty-htd-def ptr-retyp-d-eq-fst split*: *if-split-asm*)
apply(*drule-tac k=of-nat n* **and** *n=size-td s* **in** *intvl-cut*; *simp?*)
apply(*fastforce dest*: *intvlD export-size-of*
simp: *size-of-def s-footprint-untyped-def field-lvalue-def*)
apply(*drule export-size-of*, *simp add*: *size-of-def*)
done

lemma *singleton-t-SIndexTyp*:
singleton-t p v (*x*,*SIndexTyp n*) = *singleton-t p undefined* (*x*,*SIndexTyp n*)
by (*auto simp*: *singleton-t-def singleton-def restrict-map-def lift-state-def*)

lemma *proj-d-singleton-t*:
proj-d (*singleton-t p* (*v*::'*a*::*mem-type*) ++ *x*) = *proj-d* (*singleton-t p undefined* ++ *x*)
apply(*rule ext*)
apply(*clarsimp simp*: *proj-d-def*)
apply(*safe*)
apply(*subgoal-tac dom* (*singleton-t p undefined*) = *dom* (*singleton-t p v*), *blast*, *simp*)+
apply(*rule ext*)
apply(*clarsimp split*: *option.splits*)
apply(*safe*; *clarsimp?*)
apply(*subgoal-tac dom* (*singleton-t p undefined*) = *dom* (*singleton-t p v*), *blast*, *simp*)+
apply(*subst* (*asm*) *singleton-t-SIndexTyp*)
apply *simp*
done

lemma *from-bytes-heap-list-s-update*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*, *n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*::*mem-type*);
dom x = *s-footprint p* − *fs-footprint p F*; *f* ∈ *F* ⟧ ⟹
from-bytes (*heap-list-s* (*singleton-t p* (*update-ti-t s* (*to-bytes-p* (*w*::'*b*)) (*v*::'*a*)) ++ *x*)
(*size-of TYPE*('*a*)) (*ptr-val p*)) =
update-ti-t s (*to-bytes-p w*)
(*from-bytes* (*heap-list-s* (*singleton-t p v* ++ *x*) (*size-of TYPE*('*a*)) (*ptr-val p*)))
apply(*subst map-add-restrict-UNIV* [**where** *X=s-footprint-untyped* (&(*p*→*f*)) (*export-uinfo s*) **and**
h=singleton-t p v])
apply(*force simp*: *fs-footprint-def field-footprint-def field-lvalue-def*
field-typ-def field-typ-untyped-def)
apply *simp*
apply(*subst heap-list-s-map-add-super-update-bs* [**where** *k=n* **and** *z=size-td s*]; *simp?*)
apply(*rule equalityI*)
apply(*fastforce dest*: *s-footprintD export-size-of intro*: *intvlI*

```
                simp: s-footprint-untyped-def field-lvalue-def size-of-def)
   apply clarsimp
   apply(rule conjI)
    apply(frule field-tag-sub)
    apply(clarsimp simp: field-lvalue-def)
    apply(drule (1) subsetD)
    apply(fastforce elim: s-footprintI2 dest: intvlD)
   apply(fastforce dest: intvlD export-size-of
                 simp: size-of-def s-footprint-untyped-def field-lvalue-def)
  apply(fastforce dest: field-lookup-offset-size simp: size-of-def)
 apply(clarsimp simp: from-bytes-def)
 apply(frule-tac v=heap-list-s (singleton-t p (update-ti-t s (to-bytes-p w) v) |`
                                s-footprint-untyped &(p→f) (export-uinfo s))
                               (size-td s) (ptr-val p + of-nat n) and
               bs=heap-list-s (singleton-t p (update-ti-t s (to-bytes-p w) v) |`
                                (UNIV - s-footprint-untyped &(p→f) (export-uinfo
s)) ++
                               singleton-t p v |`
                                s-footprint-untyped &(p→f) (typ-uinfo-t TYPE('b))
++ x)
                              (size-of TYPE('a)) (ptr-val p) and
               w=undefined in fi-fu-consistentD; simp add: size-of-def)
 apply(subst heap-list-s-restrict)
  apply(fastforce dest: intvlD export-size-of
                 simp: size-of-def field-lvalue-def s-footprint-untyped-def)
 apply(simp add: heap-list-s-singleton-t-field-update)
 apply(subst singleton-t-mask-out; assumption?)
  apply simp
 apply(subst map-add-restrict-comp-left)
 apply simp
 done

lemma mfs-sep-map-field-update:
 ⟦ field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n); f ∈ F;
     export-uinfo s = typ-uinfo-t TYPE('b) ⟧ ⟹
  (p ↦_g^F update-ti-t s (to-bytes-p (w::'b::mem-type)) v) =
  (p ↦_g^F update-ti-t s (to-bytes-p (u::'b::mem-type)) (v::'a::mem-type))
 apply(rule ext)
 apply(clarsimp simp: mfs-sep-map-def lift-typ-heap-if s-valid-def)
 apply safe
   apply(simp add: from-bytes-heap-list-s-update)
   apply(drule-tac f=update-ti-t s (to-bytes-p u) in arg-cong)
   apply(simp add: fd-cons-double-update field-lookup-fd-consD)
  apply(simp add: from-bytes-heap-list-s-update)
  apply(drule-tac f=update-ti-t s (to-bytes-p w) in arg-cong)
  apply(simp add: fd-cons-double-update field-lookup-fd-consD)
 apply(subst (asm) proj-d-singleton-t)
 apply(subst (asm) proj-d-singleton-t[where v=update-ti-t s (to-bytes-p u) v])
 apply simp
```

apply(*subst* (*asm*) *proj-d-singleton-t*)
apply(*subst* (*asm*) *proj-d-singleton-t*[**where** *v*=*update-ti-t s* (*to-bytes-p u*) *v*])
apply *simp*
done

lemma *mfs-sep-map-field-update-v*:
⟦*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*); *f* ∈ *F*;
disjoint-fn f (*F* − {*f*}); *guard-mono g g*′;
export-uinfo t = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
p ↦$_g{}^F$ *update-ti-t t* (*to-bytes-p* (*w*::'*b*::*mem-type*)) (*v*::'*a*::*mem-type*) = *p* ↦$_g{}^F$ *v*
apply(*subst mfs-sep-map-field-update* [**where** *u*=*from-bytes* (*access-ti*$_0$ *t v*)];
simp?)
apply(*simp add*: *to-bytes-p-def to-bytes-def from-bytes-def access-ti*$_0$ *size-of-def*)
apply(*subst wf-fd-norm-tuD* [*symmetric*], *simp*)
apply(*fastforce dest*: *fd-cons-length-p export-size-of field-lookup-fd-consD simp*:
size-of-def)
apply(*rotate-tac* −*1*)
apply(*drule sym*)
apply(*simp add*: *typ-uinfo-t-def*)
apply(*subgoal-tac norm-tu* (*typ-uinfo-t TYPE*('*b*)) = *norm-bytes TYPE*('*b*))
prefer *2*
apply(*simp add*: *norm-bytes-def typ-uinfo-t-def*)
apply(*clarsimp simp*: *norm-bytes-def*
wf-fd-norm-tuD wf-fd-field-lookupD fd-cons-length-p field-lookup-fd-consD)
apply(*subst fd-cons-access-update-p* [**where** *w*=*v*])
apply(*erule field-lookup-fd-consD*)
apply(*simp add*: *fd-cons-length-p field-lookup-fd-consD*)
apply(*simp add*: *access-ti*$_0$*-def fd-cons-update-access field-lookup-fd-consD*)
done

lemma *sep-map-field-fold*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*);
f ∈ *F*; *disjoint-fn f* (*F* − {*f*}); *guard-mono g g*′;
export-uinfo t = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
(*p* ↦$_g{}^F$ (*v*::'*a*::*mem-type*) ∧*
Ptr &(*p*→*f*) ↦$_{g'}$ (*w*::'*b*::*mem-type*))
= *p* ↦$_g{}^($*F* − {*f*}) (*update-ti-t t* (*to-bytes-p w*) *v*)
apply(*simp add*: *sep-map-field-unfold*)
apply(*subst fd-cons-access-update-p* [**where** *w*=*undefined*])
apply(*erule field-lookup-fd-consD*)
apply(*fastforce dest*: *export-size-of simp*: *size-of-def*)
apply(*subst wf-fd-norm-tuD* [*symmetric*])
apply(*simp add*: *wf-fd-field-lookupD*)
apply(*fastforce dest*: *export-size-of simp*: *size-of-def*)
apply(*subgoal-tac norm-tu* (*typ-uinfo-t TYPE*('*b*)) = *norm-bytes TYPE*('*b*))
apply(*simp add*: *sep-conj-ac norm mfs-sep-map-field-update-v insert-absorb*)
apply(*simp add*: *norm-bytes-def typ-uinfo-t-def*)
done

lemma *norm-bytes*:
length bs = size-of TYPE('a) $\Longrightarrow$
to-bytes-p ((from-bytes bs)::'a) = norm-bytes TYPE('a::mem-type) bs
by (*simp add*: *norm-bytes-def wf-fd-norm-tuD size-of-def to-bytes-p-def from-bytes-def to-bytes-def access-ti$_0$-def*)

lemma *sep-heap-update-global-super-fl*:
⟦ *(p* $\mapsto_g$ *u* $\wedge^*$ *R) (lift-state (h,d))*;
field-lookup (typ-info-t TYPE('b::mem-type)) f 0 = Some (t,n);
export-uinfo t = (typ-uinfo-t TYPE('a)) ⟧ $\Longrightarrow$
((p $\mapsto_g$ *update-ti-t t (to-bytes-p v) u)* $\wedge^*$ *R)*
(lift-state (heap-update (Ptr &(p→f)) (v::'a::mem-type) h,d))
apply(*subst sep-map-mfs-sep-map-empty*)
apply(*simp add*: *sep-map-field-unfold* [**where** *g'=*λ*x. True*] *guard-mono-def*)
apply(*subst fd-cons-access-update-p* [**where** *w=undefined*])
apply(*erule field-lookup-fd-consD*)
apply *simp*
apply(*simp add*: *export-size-of*)
apply(*subst wf-fd-norm-tuD* [*symmetric*])
apply(*erule wf-fd-field-lookupD*, *simp*)
apply(*simp add*: *export-size-of*)
apply(*subgoal-tac norm-tu (typ-uinfo-t TYPE('a)) = norm-bytes TYPE('a)*)
prefer *2*
apply(*simp add*: *norm-bytes-def typ-uinfo-t-def*)
apply(*simp add*: *norm sep-conj-ac*)
apply(*subst sep-conj-com*)
apply(*simp add*: *sep-conj-assoc*)
apply(*rule sep-heap-update-global-exc2* [**where** *u=from-bytes (access-ti$_0$ t u)*])
apply(*simp add*: *sep-conj-ac*)
apply(*subst sep-conj-com*)
apply(*simp add*: *sep-map-field-fold guard-mono-def*)
apply(*subst sep-map-mfs-sep-map-empty* [*symmetric*])
apply(*simp add*: *fd-cons-double-update field-lookup-fd-consD*)
apply(*simp add*: *norm-bytes fd-cons-length-p field-lookup-fd-consD export-size-of*)
apply(*simp add*: *norm-bytes-def typ-uinfo-t-def*)
apply(*rotate-tac −1*)
apply(*drule sym*)
apply(*simp add*: *wf-fd-norm-tuD wf-fd-field-lookupD fd-cons-length-p field-lookup-fd-consD*)
apply(*subst fd-cons-access-update-p* [**where** *w=u*])
apply(*erule field-lookup-fd-consD*)
apply(*simp add*: *fd-cons-length-p field-lookup-fd-consD*)
apply(*simp add*: *access-ti$_0$-def fd-cons-update-access field-lookup-fd-consD sep-conj-com*)
done

lemma *sep-cut'-dom*:
sep-cut' x y s $\Longrightarrow$ *dom s =* {(*a*,*b*). *a* $\in$ {*x*..+*y*}}
by (*simp add*: *sep-cut'-def*)

lemma *dom-exact-sep-cut'*:

dom-exact (*sep-cut′ x y*)
by (*force intro*!: *dom-exactI dest*!: *sep-cut′-dom*)

lemma *dom-lift-state-dom-s* [*simp*]:
dom (*lift-state* (*h*,*d*)) = *dom-s d*
by (*force simp*: *lift-state-def dom-s-def split*: *s-heap-index.splits if-split-asm option.splits*)

lemma *dom-ptr-retyp-empty-htd* [*simp*]:
dom (*lift-state* (*h*,*ptr-retyp* (*p*::′*a*::*mem-type ptr*) *empty-htd*)) = *s-footprint p*
by *simp*

lemma *ptr-retyp-sep-cut′-exc*:
fixes *p*::′*a*::*mem-type ptr*
assumes *sc*: (*sep-cut′* (*ptr-val p*) (*size-of TYPE*(′*a*)) ∧* *P*) (*lift-state* (*h*,*d*)) **and** *g p*
shows (*g* ⊢*s p* ∧* *sep-true* ∧* *P*) (*lift-state* (*h*,(*ptr-retyp p d*)))
proof −
from *sc*
obtain s_0 **and** s_1 **where** $s_0 \perp s_1$ **and** *lift-state* (*h*,*d*) = s_1 ++ s_0 **and**
P s_1 **and** *d*: *dom* s_0 = {(*a*,*b*). *a* ∈ {*ptr-val p*..+*size-of TYPE*(′*a*)}}
by (*fast dest*: *sep-conjD sep-cut′-dom*)
moreover from *this*
have *lift-state* (*h*, *ptr-retyp p d*) = s_1 ++ *lift-state* (*h*, *ptr-retyp p d*) |‘ *dom* s_0
apply −
apply(*rule ext*)
subgoal for *x*
apply (*cases x* ∈ *dom* s_0)
apply(*cases x* ∈ *dom* s_1)
apply(*fastforce simp*: *map-disj-def*)
apply(*subst map-add-com*)
apply(*fastforce simp*: *map-disj-def*)
apply(*clarsimp simp*: *map-add-def split*: *option.splits*)
apply(*cases x*, *clarsimp*)
apply(*clarsimp simp*: *lift-state-ptr-retyp-d merge-dom2*)
done
done
moreover have *g p* **by** *fact*
with *d* **have** (*g* ⊢*s p* ∧* *sep-true*) (*lift-state* (*h*, *ptr-retyp p d*) |‘ *dom* s_0)
apply(*clarsimp simp*: *lift-state-ptr-retyp-restrict sep-conj-ac intro*: *ptr-retyp-tagd-exc*)
apply(*rule sep-conjI* [**where** s_0=*lift-state* (*h*,*d*) |‘ ({(*a*, *b*). *a* ∈ {*ptr-val p*..+*size-of TYPE*(′*a*)}} − *s-footprint p*)])
apply (*simp add*: *sep-conj-ac*)
apply(*erule ptr-retyp-tagd-exc* [**where** *h*=*h*])
apply(*fastforce simp*: *map-disj-def*)
apply(*subst map-add-com*[**where** h_0=*lift-state* (*h*, *ptr-retyp p empty-htd*)])
apply (*simp add*: *map-disj-def*)
apply *fast*
apply(*rule ext*)

apply(*clarsimp simp*: *map-add-def split*: *option.splits*)
by (*metis* (*mono-tags*) *Diff-iff dom-ptr-retyp-empty-htd non-dom-eval-eq restrict-in-dom restrict-out*)
ultimately
show *?thesis*
by (*subst sep-conj-assoc* [*symmetric*])
(*rule sep-conjI* [**where** s_0=(*lift-state* (*h*,*ptr-retyp p d*))|‘*dom* s_0 **and** s_1=s_1],
auto simp: *map-disj-def*)
qed

lemma *sep-cut-dom*:
sep-cut x y s $\Longrightarrow$ *dom s* = {(*a*,*b*). *a* $\in$ {*x*..+*unat y*}}
by (*force simp*: *sep-cut-def dest*: *sep-cut′-dom*)

lemma *sep-cut-0* [*simp*]:
sep-cut p 0 = $\square$
by (*auto simp*: *sep-cut′-def sep-cut-def sep-emp-def None-com split-def*)

lemma *heap-merge-restrict-dom-un*:
dom s = *P* $\cup$ *Q* $\Longrightarrow$ (*s*|‘*P*) ++ (*s*|‘*Q*) = *s*
by (*force simp*: *map-add-def restrict-map-def split*: *option.splits*)

lemma *sep-cut-split*:
assumes *sc*: *sep-cut p y s* **and** *le*: *x* $\leq$ *y*
shows (*sep-cut p x* $\wedge^*$ *sep-cut* (*p* + *x*) (*y* − *x*)) *s*
proof (*rule sep-conjI* [**where** s_0=*s*|‘{(*a*,*b*). *a* $\in$ {*p*..+*unat x*}} **and**
s_1=*s*|‘({(*a*,*b*). *a* $\in$ {*p*..+*unat y*}} − {(*a*,*b*). *a* $\in$ {*p*..+*unat x*}})])
from *sc le* **show** *sep-cut p x* (*s* |‘ {(*a*,*b*). *a* $\in$ {*p*..+*unat x*}})
by (*force simp*: *sep-cut-def sep-cut′-def word-le-nat-alt*
dest: *intvl-start-le*)
next
from *sc le*
show *sep-cut* (*p* + *x*) (*y* − *x*) (*s* |‘ ({(*a*,*b*). *a* $\in$ {*p*..+*unat y*}} − {(*a*,*b*). *a* $\in$ {*p*..+*unat x*}}))
by (*force simp*: *sep-cut-def sep-cut′-def intvl-sub-eq*)
next
show *s* |‘ {(*a*,*b*). *a* $\in$ {*p*..+*unat x*}} $\perp$ *s* |‘ ({(*a*,*b*). *a* $\in$ {*p*..+*unat y*}} − {(*a*,*b*). *a* $\in$ {*p*..+*unat x*}})
by (*force simp*: *map-disj-def*)
next
from *sc le*
show *s* = *s* |‘ ({(*a*,*b*). *a* $\in$ {*p*..+*unat y*}} − {(*a*,*b*). *a* $\in$ {*p*..+*unat x*}}) ++ *s* |‘ {(*a*,*b*). *a* $\in$ {*p*..+*unat x*}}
by (*simp add*: *sep-cut-def sep-cut′-def*, *subst heap-merge-restrict-dom-un*)
(*auto simp*: *word-le-nat-alt dest*: *intvl-start-le*)
qed

lemma *tagd-ptr-safe-exc*:
(*g* $\vdash_s$ *p* $\wedge^*$ *sep-true*) (*lift-state* (*h*,*d*)) $\Longrightarrow$ *ptr-safe p d*

apply(*clarsimp simp*: *ptr-safe-def sep-conj-ac sep-conj-def*, *drule tagd-dom-exc*)
apply(*drule-tac x*=(*a*,*b*) **in** *fun-cong*)
apply(*force simp*: *map-ac-simps lift-state-def sep-conj-ac dom-s-def merge-dom*
split: *option.splits s-heap-index.splits if-split-asm*)

done

lemma *sep-map'-ptr-safe-exc*:
(*p* $\hookrightarrow_g$ (*v*::'*a*::*mem-type*)) (*lift-state* (*h*,*d*)) $\Longrightarrow$ *ptr-safe p d*
by (*force simp*: *sep-map'-def intro*: *sep-conj-impl tagd-ptr-safe-exc*
dest: *sep-map-tagd-exc*)

end

theory *SepInv*
imports *SepCode*
begin

definition *inv-footprint* :: '*a*::*c-type ptr* $\Rightarrow$ *heap-assert* **where**
inv-footprint p $\equiv$
$\lambda s.$ *dom s* = {(*x*,*y*). *x* $\in$ {*ptr-val p*..+*size-of TYPE*('*a*)}} $-$ *s-footprint p*

Like in Separation.thy, these arrows are defined using bsub and esub but have an *input* syntax abbreviation with just sub. See original comment there for justification.

definition
sep-map-inv :: '*a*::*c-type ptr* $\Rightarrow$ '*a ptr-guard* $\Rightarrow$ '*a* $\Rightarrow$ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map-inv*››- $\mapsto^i$_ -)› [*56*,*0*,*51*] *56*)
where
p $\mapsto^i_g$ *v* $\equiv$ *p* $\mapsto_g$ *v* $\wedge^*$ *inv-footprint p*

notation (*input*)
sep-map-inv (‹(‹*open-block notation*=‹*mixfix sep-map-inv*››- $\mapsto^i$_ -)› [*56*,*1000*,*51*] *56*)

definition
sep-map-any-inv :: '*a* ::*c-type ptr* $\Rightarrow$ '*a ptr-guard* $\Rightarrow$ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map-any-inv*››- $\mapsto^i$_ −)› [*56*,*0*] *56*)
where
p $\mapsto^i_g$ − $\equiv$ *p* $\mapsto_g$ − $\wedge^*$ *inv-footprint p*

notation (*input*)
sep-map-any-inv (‹(‹*open-block notation*=‹*mixfix sep-map-any-inv*››- $\mapsto^i$_ −)› [*56*,*0*] *56*)

definition
sep-map′-inv :: *'a*::*c-type ptr* ⇒ *'a ptr-guard* ⇒ *'a* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map′-inv*››- ↪^{i}_ -)› [*56*,*0*,*51*] *56*)
where
p ↪^{i}*g v* ≡ *p* ↪*g v* ∧* *inv-footprint p*

notation (*input*)
sep-map′-inv (‹(‹*open-block notation*=‹*mixfix sep-map′-inv*››- ↪^{i}_ -)› [*56*,*1000*,*51*] *56*)

definition
sep-map′-any-inv :: *'a*::*c-type ptr* ⇒ *'a ptr-guard* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map′-any-inv*››- ↪^{i}_ −)› [*56*,*0*] *56*)
where
p ↪^{i}*g* − ≡ *p* ↪*g* − ∧* *inv-footprint p*

notation (*input*)
sep-map′-any-inv (‹(‹*open-block notation*=‹*mixfix sep-map′-any-inv*››- ↪^{i}_ −)› [*56*,*0*] *56*)

definition
tagd-inv :: *'a ptr-guard* ⇒ *'a*::*c-type ptr* ⇒ *heap-assert* (**infix** ‹⊢$_s{}^i$› *100*)
where
g ⊢$_s{}^i$ *p* ≡ *g* ⊢*s p* ∧* *inv-footprint p*

—-

lemma *sep-map′-g*:
(*p* ↪^i_g *v*) *s* ⟹ *g p*
unfolding *sep-map′-inv-def* **by** (*fastforce dest*: *sep-conjD sep-map′-g-exc*)

lemma *sep-map′-unfold*:
(*p* ↪^i_g *v*) = ((*p* ↪^i_g *v*) ∧* *sep-true*)
by (*simp add*: *sep-map′-inv-def sep-map′-def sep-conj-ac*)

lemma *sep-map′-any-unfold*:
(*i* ↪^i_g −) = ((*i* ↪^i_g −) ∧* *sep-true*)
apply(*rule ext*, *simp add*: *sep-map′-any-inv-def sep-map′-any-def sep-conj-ac*)
apply(*rule iffI*)
apply(*subst sep-conj-com*)
apply(*subst sep-conj-assoc*)+
apply(*erule* (*1*) *sep-conj-impl*)
apply(*clarsimp simp*: *sep-conj-ac*)
apply(*subst* (*asm*) *sep-map′-unfold-exc*, *subst sep-conj-com*)
apply(*subst sep-conj-exists*, *fast*)
apply(*subst* (*asm*) *sep-conj-com*)
apply(*subst* (*asm*) *sep-conj-assoc*)+
apply(*erule* (*1*) *sep-conj-impl*)
apply(*subst sep-map′-unfold-exc*)

apply(*subst* (*asm*) *sep-conj-exists*, *fast*)
done

lemma *sep-map′-conjE1*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow^i_g v)\ s]\!] \Longrightarrow (i \hookrightarrow^i_g v)\ s$
by (*subst sep-map′-unfold*, *erule sep-conj-impl*, *simp*+)

lemma *sep-map′-conjE2*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ Q\ s \Longrightarrow (i \hookrightarrow^i_g v)\ s]\!] \Longrightarrow (i \hookrightarrow^i_g v)\ s$
by (*subst* (*asm*) *sep-conj-com*, *erule sep-map′-conjE1*, *simp*)

lemma *sep-map′-any-conjE1*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow^i_g -)\ s]\!] \Longrightarrow (i \hookrightarrow^i_g -)\ s$
by (*subst sep-map′-any-unfold*, *erule sep-conj-impl*, *simp*+)

lemma *sep-map′-any-conjE2*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ Q\ s \Longrightarrow (i \hookrightarrow^i_g -)\ s]\!] \Longrightarrow (i \hookrightarrow^i_g -)\ s$
by (*subst* (*asm*) *sep-conj-com*, *erule sep-map′-any-conjE1*, *simp*)

lemma *sep-map-any-old*:
$(p \mapsto^i_g -) = (\lambda s.\ \exists v.\ (p \mapsto^i_g v)\ s)$
by (*simp add*: *sep-map-inv-def sep-map-any-inv-def sep-map-any-def sep-conj-ac sep-conj-exists*)

lemma *sep-map′-old*:
$(p \hookrightarrow^i_g v) = ((p \mapsto^i_g v) \wedge^* sep\text{-}true)$
by (*simp add*: *sep-map′-inv-def sep-map-inv-def sep-map′-def sep-conj-ac*)

lemma *sep-map′-any-old*:
$(p \hookrightarrow^i_g -) = (\lambda s.\ \exists v.\ (p \hookrightarrow^i_g v)\ s)$
by (*simp add*: *sep-map′-inv-def sep-map′-any-inv-def sep-map′-any-def sep-conj-exists*)

lemma *sep-map-sep-map′* [*simp*]:
$(p \mapsto^i_g v)\ s \Longrightarrow (p \hookrightarrow^i_g v)\ s$
unfolding *sep-map-inv-def sep-map′-inv-def sep-map′-def*
apply(*simp add*: *sep-conj-ac*)
apply(*subst sep-conj-com*)
apply(*simp add*: *sep-conj-assoc sep-conj-impl sep-conj-sep-true*)
done

lemmas *guardI* = *sep-map′-g*[*OF sep-map-sep-map′*]

lemma *sep-map-anyI* [*simp*]:
$(p \mapsto^i_g v)\ s \Longrightarrow (p \mapsto^i_g -)\ s$
by (*fastforce simp*: *sep-map-any-inv-def sep-map-inv-def sep-map-any-def sep-conj-ac*
elim: *sep-conj-impl*)

lemma *sep-map-anyD*:
$(p \mapsto^i_g -)\ s \Longrightarrow \exists v.\ (p \mapsto^i_g v)\ s$

apply(*simp add*: *sep-map-any-def sep-map-any-inv-def sep-map-inv-def sep-conj-ac*)
apply(*subst* (*asm*) *sep-conj-com*)
apply(*clarsimp simp*: *sep-conj-exists sep-conj-ac*)
done

lemma *sep-conj-mapD*:
$((i \mapsto^i_g\ v) \wedge^*\ P)\ s \Longrightarrow (i \hookrightarrow^i_g\ v)\ s \wedge ((i \mapsto^i_g\ -) \wedge^*\ P)\ s$
by (*simp add*: *sep-conj-impl sep-map'-conjE2 sep-conj-ac*)

lemma *sep-map'-ptr-safe*:
$(p \hookrightarrow^i_g (v::'a::mem\text{-}type))\ (lift\text{-}state\ (h,d)) \Longrightarrow ptr\text{-}safe\ p\ d$
unfolding *sep-map'-inv-def*
apply(*rule sep-map'-ptr-safe-exc*)
apply(*subst sep-map'-unfold-exc*)
apply(*fastforce elim*: *sep-conj-impl*)
done

lemmas *sep-map-ptr-safe* = *sep-map'-ptr-safe*[*OF sep-map-sep-map'*]

lemma *sep-map-any-ptr-safe*:
fixes $p::'a::mem\text{-}type\ ptr$
shows $(p \mapsto^i_g -)\ (lift\text{-}state\ (h,\ d)) \Longrightarrow ptr\text{-}safe\ p\ d$
by (*blast dest*: *sep-map-anyD intro*: *sep-map-ptr-safe*)

lemma *sep-heap-update'*:
$(g \vdash_s{}^i\ p \wedge^*\ (p \mapsto^i_g\ v \longrightarrow^*\ P))\ (lift\text{-}state\ (h,d)) \Longrightarrow$
$\quad P\ (lift\text{-}state\ (heap\text{-}update\ p\ (v::'a::mem\text{-}type)\ h,d))$
apply(*rule sep-heap-update'-exc* [**where** *g*=*g*])
apply(*unfold tagd-inv-def*)
apply(*subst* (*asm*) *sep-conj-assoc*)+
apply(*erule* (*1*) *sep-conj-impl*)
apply(*subst* (*asm*) *sep-map-inv-def*)
apply(*simp add*: *sep-conj-ac*)
apply(*drule sep-conjD*, *clarsimp*)
apply(*rule sep-implI*, *clarsimp*)
apply(*drule sep-implD*)
apply(*drule-tac* $x{=}s_0$ ++ s' **in** *spec*)
apply(*simp add*: *map-disj-com map-add-disj*)
apply(*clarsimp simp*: *map-disj-com*)
apply(*erule notE*)
apply(*erule* (*1*) *sep-conjI*)
apply(*simp add*: *map-disj-com*)
apply(*subst map-add-com*; *simp*)
done

lemma *tagd-g*:
$(g \vdash_s{}^i\ p \wedge^*\ P)\ s \Longrightarrow g\ p$
by (*auto simp*: *tagd-inv-def tagd-def dest*!: *sep-conjD elim*: *s-valid-g*)

lemma *tagd-ptr-safe*:
(*g* ⊢$_s{}^i$ *p* ∧* *sep-true*) (*lift-state* (*h*,*d*)) ⟹ *ptr-safe p d*
apply(*rule tagd-ptr-safe-exc*)
apply(*unfold tagd-inv-def*)
apply(*subst* (*asm*) *sep-conj-assoc*)
apply(*erule* (*1*) *sep-conj-impl*)
apply *simp*
done

lemma *sep-map-tagd*:
(*p* ↦^i_g (*v*::'*a*::*mem-type*)) *s* ⟹ (*g* ⊢$_s{}^i$ *p*) *s*
apply(*unfold sep-map-inv-def tagd-inv-def*)
apply(*erule sep-conj-impl*)
apply(*erule sep-map-tagd-exc*)
apply *assumption*
done

lemma *sep-map-any-tagd*:
(*p* ↦^i_g −) *s* ⟹ (*g* ⊢$_s{}^i$ (*p*::'*a*::*mem-type ptr*)) *s*
by (*clarsimp dest*!: *sep-map-anyD*, *erule sep-map-tagd*)

lemma *sep-heap-update*:
⟦ (*p* ↦^i_g − ∧* (*p* ↦^i_g *v* ⟶* *P*)) (*lift-state* (*h*,*d*)) ⟧ ⟹
P (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
by (*force intro*: *sep-heap-update'* *dest*: *sep-map-anyD sep-map-tagd*
elim: *sep-conj-impl*)

lemma *sep-heap-update-global'*:
(*g* ⊢$_s{}^i$ *p* ∧* *R*) (*lift-state* (*h*,*d*)) ⟹
((*p* ↦^i_g *v*) ∧* *R*) (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
by (*rule sep-heap-update'*, *erule sep-conj-sep-conj-sep-impl-sep-conj*)

lemma *sep-heap-update-global*:
(*p* ↦^i_g − ∧* *R*) (*lift-state* (*h*,*d*)) ⟹
((*p* ↦^i_g *v*) ∧* *R*) (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
by (*fast intro*: *sep-heap-update-global' sep-conj-impl sep-map-any-tagd*)

lemma *sep-heap-update-global-super-fl-inv*:
⟦ (*p* ↦^i_g *u* ∧* *R*) (*lift-state* (*h*,*d*));
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = (*typ-uinfo-t TYPE*('*a*)) ⟧ ⟹
((*p* ↦^i_g *update-ti-t t* (*to-bytes-p v*) *u*) ∧* *R*)
(*lift-state* (*heap-update* (*Ptr* &(*p*→*f*)) (*v*::'*a*::*mem-type*) *h*,*d*))
apply(*unfold sep-map-inv-def*)
apply(*simp only*: *sep-conj-assoc*)
apply(*erule* (*2*) *sep-heap-update-global-super-fl*)
done

lemma *sep-map'-inv*:

```
  (p ↪i_g v) s ⟹ (p ↪_g v) s
  apply(unfold sep-map'-inv-def)
  apply(subst sep-map'-unfold-exc)
  apply(erule (1) sep-conj-impl, simp)
  done

lemma sep-map'-lift:
  (p ↪i_g (v::'a::mem-type)) (lift-state (h,d)) ⟹ lift h p = v
  apply(drule sep-map'-inv)
  apply(erule sep-map'-lift-exc)
  done

lemma sep-map-lift:
  ((p::'a::mem-type ptr) ↦i_g −) (lift-state (h,d)) ⟹
        (p ↦i_g lift h p) (lift-state (h,d))
  apply(frule sep-map-anyD)
  apply clarsimp
  apply(frule sep-map-sep-map')
  apply(drule sep-map'-lift)
  apply simp
  done

lemma sep-map-lift-wp:
  ⟦ ∃ v. (p ↦i_g v ∧* (p ↦i_g v ⟶* P v)) (lift-state (h,d)) ⟧
      ⟹ P (lift h (p::'a::mem-type ptr)) (lift-state (h,d))
  apply clarsimp
  subgoal for v
    apply(subst sep-map'-lift [where g=g and d=d])
     apply(subst sep-map'-inv-def)
     apply(subst sep-map'-def)
     apply(subst sep-conj-assoc)+
     apply(subst sep-conj-com[where P=sep-true])
     apply(subst sep-conj-assoc [symmetric])
     apply(erule sep-conj-impl)
      apply(simp add: sep-map-inv-def)
     apply simp
    apply(rule sep-conj-impl-same [where P=p ↦i_g v and Q=P v])
    apply(unfold sep-map-inv-def)
    apply(erule (2) sep-conj-impl)
    done
  done

lemma sep-map'-anyI [simp]:
  (p ↪i_g v) s ⟹ (p ↪i_g −) s
  apply(unfold sep-map'-inv-def sep-map'-any-inv-def)
  apply(erule sep-conj-impl)
   apply(erule sep-map'-anyI-exc)
  apply assumption
  done
```

lemma *sep-map'-anyD*:
(*p* ↪^i_g −) *s* ⟹ ∃ *v*. (*p* ↪^i_g *v*) *s*
unfolding *sep-map'-inv-def sep-map'-any-inv-def sep-map'-any-def*
by (*clarsimp simp*: *sep-conj-exists*)

lemma *sep-map'-lift-rev*:
⟦ *lift h p* = (*v*::'*a*::*mem-type*); (*p* ↪^i_g −) (*lift-state* (*h*,*d*)) ⟧ ⟹
(*p* ↪^i_g *v*) (*lift-state* (*h*,*d*))
by (*fastforce dest*: *sep-map'-anyD simp*: *sep-map'-lift*)

lemma *sep-map'-any-g*:
(*p* ↪^i_g −) *s* ⟹ *g p*
by (*blast dest*: *sep-map'-anyD intro*: *sep-map'-g*)

lemma *any-guardI*:
(*p* ↦^i_g −) *s* ⟹ *g p*
by (*drule sep-map-anyD*) (*blast intro*: *guardI*)

lemma *sep-map-sep-map-any*:
(*p* ↦^i_g *v*) *s* ⟹ (*p* ↦^i_g −) *s*
by (*rule sep-map-anyI*)

lemma *sep-lift-exists*:
fixes *p* :: '*a*::*mem-type ptr*
assumes *ex*: ((λ*s*. ∃ *v*. (*p* ↪^i_g *v*) *s* ∧ *P v s*) ∧* *Q*) (*lift-state* (*h*,*d*))
shows (*P* (*lift h p*) ∧* *Q*) (*lift-state* (*h*,*d*))
proof −
from *ex* **obtain** *v* **where** ((λ*s*. (*p* ↪^i_g *v*) *s* ∧ *P v s*) ∧* *Q*)
(*lift-state* (*h*,*d*))
by (*subst* (*asm*) *sep-conj-exists*, *clarsimp*)
thus *?thesis*
by (*force simp*: *sep-map'-lift sep-conj-ac*
dest: *sep-map'-conjE2 dest*!: *sep-conj-conj*)
qed

lemma *sep-map-dom*:
(*p* ↦^i_g (*v*::'*a*::*c-type*)) *s* ⟹ *dom s* = {(*a*,*b*). *a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)}}
unfolding *sep-map-inv-def*
by (*drule sep-conjD*, *clarsimp*)
(*auto dest*!: *sep-map-dom-exc elim*: *s-footprintD simp*: *inv-footprint-def*)

lemma *sep-map'-dom*:
(*p* ↪^i_g (*v*::'*a*::*mem-type*)) *s* ⟹ (*ptr-val p*,*SIndexVal*) ∈ *dom s*
unfolding *sep-map'-inv-def*
by (*drule sep-conjD*, *clarsimp*) (*drule sep-map'-dom-exc*, *clarsimp*)

lemma *sep-map'-inj*:

⟦ $(p \hookrightarrow^i{}_g\ (v::'a::c\text{-}type))\ s$; $(p \hookrightarrow^i{}_h\ v')\ s$ ⟧ ⟹ $v=v'$
by (*drule sep-map'-inv*)+ (*drule* (*2*) *sep-map'-inj-exc*)

lemma *ptr-retyp-sep-cut'*:
fixes *p*::'*a*::*mem-type ptr*
assumes *sc*: (*sep-cut'* (*ptr-val p*) (*size-of TYPE*('*a*)) ∧* *P*)
(*lift-state* (*h*,*d*)) **and** *g p*
shows (*g* ⊢$_s{}^i$ *p* ∧* *P*) (*lift-state* (*h*,(*ptr-retyp p d*)))
proof −
from *sc*
obtain s_0 **and** s_1
where $s_0 \perp s_1$ **and** *lift-state* (*h*,*d*) = s_1 ++ s_0
and *P* s_1 **and** *d*: *dom* s_0 = {(*a*,*b*). *a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)}}
and *k*: *dom* s_0 ⊆ *dom-s d*
by (*auto dest*!: *sep-conjD sep-cut'-dom simp*: *dom-lift-state-dom-s* [**where** *h*=*h*,*symmetric*])
moreover from *this*
have *lift-state* (*h*, *ptr-retyp p d*) = s_1 ++ *lift-state* (*h*, *ptr-retyp p d*) |` (*dom* s_0)
apply −
apply(*rule ext*)
subgoal for *x*
apply(*cases x* ∈ *dom* s_0)
apply(*cases x* ∈ *dom* s_1)
apply(*fastforce simp*: *map-disj-def*)
apply(*subst map-add-com*)
apply(*fastforce simp*: *map-disj-def*)
apply(*clarsimp simp*: *map-add-def split*: *option.splits*)
apply(*cases x*, *clarsimp*)
apply(*clarsimp simp*: *lift-state-ptr-retyp-d merge-dom2*)
done
done
moreover have *g p* **by** *fact*
with *d k* **have** (*g* ⊢$_s{}^i$ *p*) (*lift-state* (*h*, *ptr-retyp p d*) |` *dom* s_0)
apply(*clarsimp simp*: *lift-state-ptr-retyp-restrict sep-conj-ac tagd-inv-def*)
apply(*rule sep-conjI* [**where** s_0=*lift-state* (*h*,*d*) |` ({(*a*, *b*). *a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)}} − *s-footprint p*)])
apply(*fastforce simp*: *inv-footprint-def*)
apply(*erule ptr-retyp-tagd-exc*[**where** *h*=*h*])
apply(*fastforce simp*: *map-disj-def*)
apply(*subst map-add-comm*[*of lift-state* (*h*, *ptr-retyp p empty-htd*)])
apply *force*
apply(*rule ext*)
apply(*clarsimp simp*: *map-add-def split*: *option.splits*)
by (*metis* (*mono-tags*) *Diff-iff dom-ptr-retyp-empty-htd non-dom-eval-eq restrict-in-dom restrict-out*)
ultimately show *?thesis*
by (*metis restrict-map-on-disj sep-conjI*)
qed

lemma *ptr-retyp-sep-cut'-wp*:
⟦ (*sep-cut'* (*ptr-val p*) (*size-of TYPE*('*a*)) ∧* (*g* ⊢$_s^i$ *p* ⟶* *P*))
(*lift-state* (*h*,*d*)); *g* (*p*::'*a*::*mem-type ptr*) ⟧
⟹ *P* (*lift-state* (*h*,(*ptr-retyp p d*)))
by (*rule sep-conj-impl-same* [**where** *P*=*g* ⊢$_s^i$ *p* **and** *Q*=*P*]) (*simp add*: *ptr-retyp-sep-cut'*)

lemma *tagd-dom*:
(*g* ⊢$_s^i$ *p*) *s* ⟹ *dom s* = {(*a*,*b*). *a* ∈ {*ptr-val* (*p*::'*a*::*c-type ptr*)..+*size-of TYPE*('*a*)}}
unfolding *tagd-inv-def*
by (*drule sep-conjD*, *clarsimp*)
(*auto simp*: *inv-footprint-def dest*!: *tagd-dom-exc elim*: *s-footprintD*)

lemma *tagd-dom-p*:
(*g* ⊢$_s^i$ *p*) *s* ⟹ (*ptr-val* (*p*::'*a*::*mem-type ptr*),*SIndexVal*) ∈ *dom s*
by (*drule tagd-dom*) *clarsimp*

end

theory *SepTactic*
imports *SepInv*
begin

11.24 *sep-point-tac*

definition *sep-conj-extract* :: *heap-assert* ⇒ *heap-assert* **where**
sep-conj-extract ≡ *id*

definition *sep-points* :: *heap-assert* ⇒ *heap-assert* **where**
sep-points P ≡ *P* ∧* *sep-true*

lemma *sep-conj-extract1D*:
(*P* ∧* *Q*) *s* ⟹ *sep-conj-extract* (*P* ∧* *Q*) *s*
by (*simp add*: *sep-conj-extract-def*)

lemma *sep-conj-extract2D*:
sep-conj-extract P s ⟹ *P s* ∧ (*sep-conj-extract P* ∧* *sep-true*) *s*
by (*auto simp*: *sep-conj-extract-def elim*: *sep-conj-sep-true*)

lemma *sep-conj-extract-assoc*:
sep-conj-extract ((*P* ∧* *Q*) ∧* *R*) = *sep-conj-extract* (*P* ∧* (*Q* ∧* *R*))
by (*simp add*: *sep-conj-ac*)

lemma *sep-conj-extract-decomposeD*:
(*sep-conj-extract* (*P* ∧* *Q*) ∧* *sep-true*) *s* ⟹ *sep-points P s* ∧
(*sep-conj-extract Q* ∧* *sep-true*) *s*

apply (*rule conjI*)
apply(*simp add*: *sep-conj-extract-def sep-points-def sep-conj-ac*)
apply(*erule* (*1*) *sep-conj-impl*, *simp*)
apply(*simp add*: *sep-conj-extract-def sep-conj-ac*)
apply(*subst* (*asm*) *sep-conj-assoc* [*symmetric*])
apply(*subst* (*asm*) *sep-conj-com*)
apply(*subst* (*asm*) *sep-conj-assoc*)
apply(*erule* (*1*) *sep-conj-impl*)
apply *simp*
done

lemma *sep-conj-extract-decomposeD2*:
(*sep-conj-extract P* $\wedge^*$ *sep-true*) *s* $\Longrightarrow$ *sep-points P s*
by (*simp add*: *sep-conj-extract-def sep-points-def*)

lemma *sep-point-mapD*:
sep-points ($p \mapsto^i_g v$) *s* $\Longrightarrow$ ($p \hookrightarrow^i_g v$) *s*
by (*simp add*: *sep-points-def sep-map-inv-def sep-map'-inv-def sep-map'-def*
sep-conj-ac)

lemma *sep-point-map-excD*:
sep-points ($p \mapsto_g v$) *s* $\Longrightarrow$ ($p \hookrightarrow_g v$) *s*
by (*simp add*: *sep-points-def sep-map'-def*)

lemma *sep-point-otherD*:
sep-points P s $\Longrightarrow$ *True*
by (*rule TrueI*)

ML ‹

val sep-point-ds = [@{*thm sep-point-mapD*}, @{*thm sep-point-map-excD*}]

fun sep-point-tacs ds ctxt = [
REPEAT1 (*dresolve-tac ctxt* [@{*thm sep-conj-extract1D*}] *1*),
REPEAT1 (*dresolve-tac ctxt* [@{*thm sep-conj-extract2D*}] *1*),
clarify-tac ctxt 1,
TRY (*full-simp-tac* (*put-simpset HOL-ss ctxt addsimps*
[@{*thm sep-conj-extract-assoc*}]) *1*),
REPEAT (*dresolve-tac ctxt* [@{*thm sep-conj-extract-decomposeD*},
@{*thm sep-conj-extract-decomposeD2*}] *1 THEN*
clarify-tac ctxt 1),
TRY (*full-simp-tac ctxt 1*),
REPEAT (*dresolve-tac ctxt* (*sep-point-ds*@*ds*) *1*),
REPEAT (*dresolve-tac ctxt* [@{*thm sep-point-otherD*}] *1*),
TRY (*full-simp-tac ctxt 1*)
]

fun sep-point-tac ds ctxt = *EVERY* (*sep-point-tacs ds ctxt*)
›

method-setup *sep-point-tac* =
Attrib.thms >> (fn thms => Method.SIMPLE-METHOD o sep-point-tac thms)
extract points−to facts from separation assertions in assumptions

lemma $(P \wedge^* p \mapsto_g v \wedge^* Q)\ s \Longrightarrow (p \hookrightarrow_g v)\ s$
by *sep-point-tac*

11.25 *sep-exists-tac*

ML‹
fun sep-exists-tac ctxt = full-simp-tac
(put-simpset HOL-ss ctxt addsimps [@{thm sep-conj-exists1}, @{thm sep-conj-exists2}])
›

method-setup *sep-exists-tac* =
Scan.succeed (Method.SIMPLE-METHOD' o sep-exists-tac)
push existentials inside separation assertions to the outside

lemma $(P \wedge^* (\lambda s.\ (\exists x.\ Q\ x\ s)))\ s \Longrightarrow \exists x.\ (P \wedge^* Q\ x)\ s$
by *sep-exists-tac*

11.26 *sep-select-tac*

definition *sep-mark* :: *heap-assert* ⇒ *heap-assert* **where**
sep-mark ≡ *id*

definition *sep-mark2* :: *heap-assert* ⇒ *heap-assert* **where**
sep-mark2 ≡ *id*

lemma *sep-mark2-id*:
sep-mark2 P = P
by (*simp add*: *sep-mark2-def*)

lemma *sep-markI*:
$(\square \wedge^* sep\text{-}mark\ (P \wedge^* Q))\ s \Longrightarrow (P \wedge^* Q)\ s$
by (*simp add*: *sep-mark-def sep-conj-empty*)

lemma *sep-mark-match*:
$(R \wedge^* sep\text{-}mark2\ P \wedge^* Q)\ s \Longrightarrow (R \wedge^* sep\text{-}mark\ (P \wedge^* Q))\ s$
by (*simp add*: *sep-mark-def sep-mark2-def*)

lemma *sep-mark-match2*:
$(R \wedge^* sep\text{-}mark2\ P)\ s \Longrightarrow (R \wedge^* sep\text{-}mark\ P)\ s$
by (*simp add*: *sep-mark-def sep-mark2-def*)

lemma *sep-mark-mismatch*:
$((R \wedge^* P) \wedge^* sep\text{-}mark\ Q)\ s \Longrightarrow (R \wedge^* sep\text{-}mark\ (P \wedge^* Q))\ s$

by (*simp add*: *sep-mark-def sep-conj-ac*)

lemma *sep-mark-mismatch2*:
(*R* ∧* *P*) *s* ⟹ (*R* ∧* *sep-mark P*) *s*
by (*simp add*: *sep-mark-def*)

lemma *sep-emp-rem*:
P s ⟹ (□ ∧* *P*) *s*
by (*simp add*: *sep-conj-empty*)

lemma *sep-mark2-left*:
(*P* ∧* (*sep-mark2 Q* ∧* *R*)) = (*sep-mark2 Q* ∧* (*P* ∧* *R*))
by (*rule sep-conj-left-com*)

lemma *sep-mark2-left2*:
(*P* ∧* *sep-mark2 Q*) = (*sep-mark2 Q* ∧* *P*)
by (*rule sep-conj-com*)

ML ‹

```
(* Replace all occurrences of an underscore that does not have an alphanumeric
   on either side of it *)
local
  val dummy-char = # ;
  fun get-next (x::-) = x
    | get-next [] = dummy-char;
  fun nth-id n = SEPTAC^(Int.toString n);

  fun ok-char c = not (Char.isAlphaNum c);

  fun can-replace (prev,cur,last) =
        (ok-char prev) andalso (cur = #-) andalso (ok-char last);

  fun replace-usc n prev-char (x::xs) =
      let val (xs', ys) = replace-usc (n+1) x xs
      in
        if can-replace (prev-char,x,get-next xs)
        then (nth-id n::xs', nth-id n::ys)
        else (String.str x::xs', ys)
      end
    | replace-usc - - [] = ([],[])
in
  val rusc = apfst String.concat o replace-usc 0 dummy-char o String.explode
end;

(* some tests *)
rusc (- +_p -) ↦ -;
rusc - ↦_ -;
rusc -;
```

```
rusc - O-o -
›

ML‹

fun sep-select-tacs s ctxt =
  let val (str, vars) = rusc s
      val subst = [((Lexicon.read-indexname P, Position.none), str)]
      val fixes = map (fn v => (Binding.name v, NONE, Mixfix.NoSyn)) vars
  in
  [
    TRY (simp-tac (put-simpset HOL-ss ctxt addsimps [@{thm sep-conj-assoc}])
1),
    resolve-tac ctxt [@{thm sep-markI}] 1,
     REPEAT (Rule-Insts.res-inst-tac ctxt subst fixes @{thm sep-mark-match} 1
ORELSE
      (Rule-Insts.res-inst-tac ctxt subst fixes @{thm sep-mark-match2} 1 ORELSE
      resolve-tac ctxt [@{thm sep-mark-mismatch}] 1 ORELSE
      resolve-tac ctxt [@{thm sep-mark-mismatch2}] 1)),
     TRY (simp-tac (put-simpset HOL-ss ctxt addsimps [@{thm sep-conj-assoc}])
1),
    resolve-tac ctxt [@{thm sep-emp-rem}] 1,
    TRY (simp-tac (put-simpset HOL-ss ctxt addsimps [@{thm sep-mark2-left},
      @{thm sep-mark2-left2}]) 1),
    TRY (simp-tac (put-simpset HOL-ss ctxt addsimps [@{thm sep-mark2-id}]) 1)
  ]
  end

fun sep-select-tac s ctxt = SELECT-GOAL (EVERY (sep-select-tacs s ctxt))

fun lift-parser p (ctxt,ts) =
  let val (r, ts') = p ts
  in (r, (ctxt,ts')) end
›

method-setup sep-select-tac =
  ‹lift-parser Args.name >> (fn s => Method.SIMPLE-METHOD' o sep-select-tac
s)›
  takes a target conjunct in the goal and reorders it to the front

lemma
  ⋀R x f n. ((P::heap-assert) ∧* fac x n ∧* R (f x)) s
  apply(sep-select-tac fac - -)
  apply(sep-select-tac R -)
  apply(sep-select-tac fac x -)
  apply(sep-select-tac R -)
  oops

lemma
```

```
  ((P::heap-assert) ∧* fac x n) s
  apply(sep-select-tac fac - -)
  oops

lemma
  ((P::heap-assert) ∧* long-name) s
  apply(sep-select-tac long-name)
  oops



lemma sep-heap-update'-hrs:
  (c-guard ⊢s^i p ∧* (p ↦^i c-guard v ⟶* P)) (lift-state s) ⟹
     P (lift-state (hrs-mem-update (heap-update p (v::'a::mem-type)) s))
  by (cases s)
     (simp add: hrs-mem-update-def, erule sep-heap-update')

lemma sep-map-lift-wp-hrs:
  ⟦ ∃ v. (p ↦^i c-guard v ∧* (p ↦^i c-guard v ⟶* P v)) (lift-state s) ⟧
     ⟹ P (lift (hrs-mem s) (p::'a::mem-type ptr)) (lift-state s)
  by (cases s)
     (simp add: hrs-mem-def, erule sep-map-lift-wp)

lemma ptr-retyp-sep-cut'-wp-hrs:
  ⟦ (sep-cut' (ptr-val p) (size-of TYPE('a)) ∧* (c-guard ⊢s^i p ⟶* P))
      (lift-state s); c-guard (p::'a::mem-type ptr) ⟧
     ⟹ P (lift-state (hrs-htd-update (ptr-retyp p) s))
  by (cases s)
     (simp add: hrs-htd-update-def, erule (1) ptr-retyp-sep-cut'-wp)

ML ‹

fun destruct bs (Bound n)      = Free (List.nth (bs,n))
  | destruct bs (Abs (s,ty,t)) = destruct ((s,ty)::bs) t
  | destruct bs (l$r)          = destruct bs l $ destruct bs r
  | destruct - x               = x

fun schems (Abs (-,-,t)) = schems t
  | schems (l$r)         = schems l + schems r
  | schems (Var -)       = 1
  | schems -             = 0

fun hrs-mem-update (Const (@{const-name HeapRawState.hrs-mem-update},-)$-$s)
= SOME s
  | hrs-mem-update -                                    = NONE

fun hrs-htd-update (Const (@{const-name HeapRawState.hrs-htd-update},-)$-$s) =
SOME s
  | hrs-htd-update -                                    = NONE
```

```
fun lift-state-arg -   -  (Const (@{const-name TypHeap.lift-state},-)$s) = SOME s
  | lift-state-arg thy nc (l$r)                                  =
       (case lift-state-arg thy nc r of
         SOME s => if nc orelse hrs-mem-update s <> NONE orelse
                      hrs-htd-update s <> NONE then SOME s
                      else lift-state-arg thy nc l
        | NONE   => lift-state-arg thy nc l)
  | lift-state-arg thy nc (Abs (-,-,t))                          =
       lift-state-arg thy nc t
  | lift-state-arg - - - =
       NONE (*error (Sign.string-of-term thy s)*)

fun term-of-thm thm = Thm.term-of (Thm.cprop-of thm)

fun lift-tac ctxt s old-schems =
  Rule-Insts.res-inst-tac ctxt [((Lexicon.read-indexname s,Position.none),s)] [] @{thm
sep-map-lift-wp-hrs} 1 THEN
  (fn thm => if schems (term-of-thm thm) = old-schems then all-tac thm
             else no-tac thm)

fun heap-update-tac ctxt s =
  Rule-Insts.res-inst-tac ctxt [((Lexicon.read-indexname s,Position.none),s)] [] @{thm
sep-heap-update'-hrs} 1

fun ptr-retyp-tac ctxt s =
  Rule-Insts.res-inst-tac ctxt [((Lexicon.read-indexname s,Position.none),s)] [] @{thm
ptr-retyp-sep-cut'-wp-hrs} 1

fun sep-wp-step-tac' ctxt lift-only s old-schems = let
  fun prt s = (
    case s of
       Free (a, -) => a
     | - => raise TERM (Not a free variable,[s]))
  val state = prt s
  val step = case hrs-mem-update s of
               SOME s => heap-update-tac ctxt (prt s)
             | NONE => case hrs-htd-update s of
                         SOME s => ptr-retyp-tac ctxt (prt s)
                       | NONE => no-tac
in
  TRY (lift-tac ctxt state old-schems) THEN (if lift-only then all-tac else step)
end

fun sep-wp-step-tac ctxt lift-only thm = let
  val t = term-of-thm thm
  val thy = Proof-Context.theory-of ctxt
  val old-schems = schems t
  val s = lift-state-arg thy lift-only (destruct [] t)
```

in
(*case s of*
SOME state => sep-wp-step-tac' ctxt lift-only state old-schems
| *NONE => no-tac*) *thm*
end

fun sep-wp-tac ctxt = (*REPEAT* (*sep-wp-step-tac ctxt false*)) *THEN sep-wp-step-tac ctxt true*

›

method-setup *sep-wp-tac* =
Scan.succeed (*Method.SIMPLE-METHOD o sep-wp-tac*)
apply WP separation logic rules to goal

lemma
((λ*z*. *lift* (*hrs-mem s*) (*p*::(*32 word*) *ptr*) + *1* = *2*) ∧* *P*) (*lift-state s*)
apply *sep-wp-tac*
oops

end

theory *SepFrame*
imports *SepTactic*
begin

class *heap-state-type'*

instance *heap-state-type'* ⊆ *type* **..**

consts
hst-mem :: *'a*::*heap-state-type'* ⇒ *heap-mem*
hst-mem-update :: (*heap-mem* ⇒ *heap-mem*) ⇒ *'a*::*heap-state-type'* ⇒ *'a*
hst-htd :: *'a*::*heap-state-type'* ⇒ *heap-typ-desc*
hst-htd-update :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'a*::*heap-state-type'* ⇒ *'a*

class *heap-state-type* = *heap-state-type'* +
assumes *hst-htd-htd-update* [*simp*]: *hst-htd* (*hst-htd-update d s*) = *d* (*hst-htd s*)
assumes *hst-mem-mem-update* [*simp*]: *hst-mem* (*hst-mem-update h s*) = *h* (*hst-mem s*)
assumes *hst-htd-mem-update* [*simp*]: *hst-htd* (*hst-mem-update h s*) = *hst-htd s*
assumes *hst-mem-htd-update* [*simp*]: *hst-mem* (*hst-htd-update d s*) = *hst-mem s*

translations
$s(\!|\ \mathit{hst\text{-}mem} := x\ |\!) <= \mathit{CONST\ hst\text{-}mem\text{-}update}\ (\mathit{K\text{-}record}\ x)\ s$
$s(\!|\ \mathit{hst\text{-}htd} := x\ |\!) <= \mathit{CONST\ hst\text{-}htd\text{-}update}\ (\mathit{K\text{-}record}\ x)\ s$

definition *lift-hst* :: $'a$::*heap-state-type'* $\Rightarrow$ *heap-state* **where**
lift-hst $s \equiv$ *lift-state* (*hst-mem* s,*hst-htd* s)

definition *point-eq-mod* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow bool$ **where**
point-eq-mod $f\ g\ X \equiv \forall x.\ x \notin X \longrightarrow f\ x = g\ x$

definition *exec-fatal* :: $('s,'b,'c)\ com \Rightarrow ('s,'b,'c)\ body \Rightarrow 's \Rightarrow bool$ **where**
exec-fatal $C\ \Gamma\ s \equiv (\exists f.\ \Gamma \vdash \langle C,Normal\ s\rangle \Rightarrow Fault\ f) \vee (\Gamma \vdash \langle C,Normal\ s\rangle \Rightarrow Stuck)$

definition *restrict-htd* :: $'s$::*heap-state-type'* $\Rightarrow$ *s-addr set* $\Rightarrow$ $'s$ **where**
restrict-htd $s\ X \equiv s(\!|\ \mathit{hst\text{-}htd} := \mathit{restrict\text{-}s}\ (\mathit{hst\text{-}htd}\ s)\ X\ |\!)$

definition *restrict-safe-OK* ::
$'s \Rightarrow 's \Rightarrow ('s \Rightarrow ('s,'c)\ xstate) \Rightarrow$ *s-addr set* $\Rightarrow$ ($'s$::*heap-state-type'*,$'b$,$'c$) *com* $\Rightarrow$
($'s$,$'b$,$'c$) *body* $\Rightarrow$ *bool* **where**
restrict-safe-OK $s\ t\ f\ X\ C\ \Gamma \equiv$
$\Gamma \vdash \langle C,(Normal\ (\mathit{restrict\text{-}htd}\ s\ X))\rangle \Rightarrow f\ (\mathit{restrict\text{-}htd}\ t\ X) \wedge$
point-eq-mod (*lift-state* (*hst-mem* t,*hst-htd* t)) (*lift-state* (*hst-mem* s,*hst-htd* s)) X

definition *restrict-safe* ::
$'s \Rightarrow ('s,'c)\ xstate \Rightarrow$ ($'s$::*heap-state-type'*,$'b$,$'c$) *com* $\Rightarrow$ ($'s$,$'b$,$'c$) *body* $\Rightarrow$ *bool* **where**
restrict-safe $s\ t\ C\ \Gamma \equiv$
$\forall X.$ (*case* t *of*
Normal $t' \Rightarrow$ *restrict-safe-OK* $s\ t'$ *Normal* $X\ C\ \Gamma$
| *Abrupt* $t' \Rightarrow$ *restrict-safe-OK* $s\ t'$ *Abrupt* $X\ C\ \Gamma$
| - $\Rightarrow$ *False*) $\vee$
exec-fatal $C\ \Gamma$ (*restrict-htd* $s\ X$)

definition *mem-safe* :: ($'s$::{*heap-state-type'*,*type*},$'b$,$'c$) *com* $\Rightarrow$ ($'s$,$'b$,$'c$) *body* $\Rightarrow$ *bool* **where**
mem-safe $C\ \Gamma \equiv \forall s\ t.\ \Gamma \vdash \langle C,Normal\ s\rangle \Rightarrow t \longrightarrow$ *restrict-safe* $s\ t\ C\ \Gamma$

definition *point-eq-mod-safe* ::
$'s$::{*heap-state-type'*,*type*} *set* $\Rightarrow ('s \Rightarrow 's) \Rightarrow ('s \Rightarrow$ *s-addr* $\Rightarrow 'c) \Rightarrow$ *bool* **where**
point-eq-mod-safe $P\ f\ g \equiv \forall s\ X.$ *restrict-htd* $s\ X \in P \longrightarrow$ *point-eq-mod* $(g\ (f\ s))\ (g\ s)\ X$

definition *comm-restrict* :: ($'s$::{*heap-state-type'*,*type*} $\Rightarrow$ $'s$) $\Rightarrow$ $'s \Rightarrow$ *s-addr set* $\Rightarrow$ *bool* **where**
comm-restrict $f\ s\ X \equiv f\ (\mathit{restrict\text{-}htd}\ s\ X) = \mathit{restrict\text{-}htd}\ (f\ s)\ X$

definition *comm-restrict-safe* :: $'s$ *set* $\Rightarrow$ ($'s$::{*heap-state-type'*,*type*} $\Rightarrow$ $'s$) $\Rightarrow$ *bool*

where
comm-restrict-safe P f ≡ ∀ *s X*. *restrict-htd s X* ∈ *P* ⟶ *comm-restrict f s X*

definition *htd-ind* :: (*'a*::{*heap-state-type'*,*type*} ⇒ *'b*) ⇒ *bool* **where**
htd-ind f ≡ ∀ *x s*. *f s* = *f* (*hst-htd-update x s*)

definition *mono-guard* :: *'s*::{*heap-state-type'*,*type*} *set* ⇒ *bool* **where**
mono-guard P ≡ ∀ *s X*. *restrict-htd s X* ∈ *P* ⟶ *s* ∈ *P*

definition *expr-htd-ind* :: *'s*::{*heap-state-type'*,*type*} *set* ⇒ *bool* **where**
expr-htd-ind P ≡ ∀ *d s*. *s*(| *hst-htd* := *d* |) ∈ *P* = (*s* ∈ *P*)

primrec *intra-safe* :: (*'s*::*heap-state-type'*,*'b*,*'c*) *com* ⇒ *bool*
where
intra-safe Skip = *True*
| *intra-safe* (*Basic f*) = (*comm-restrict-safe UNIV f* ∧
point-eq-mod-safe UNIV f (λ*s*. *lift-state* (*hst-mem s*,*hst-htd s*)))
| *intra-safe* (*Spec r*) = (∀ Γ. *mem-safe* (*Spec r*) (Γ :: (*'s*,*'b*,*'c*) *body*))
| *intra-safe* (*Seq C D*) = (*intra-safe C* ∧ *intra-safe D*)
| *intra-safe* (*Cond P C D*) = (*expr-htd-ind P* ∧ *intra-safe C* ∧ *intra-safe D*)
| *intra-safe* (*While P C*) = (*expr-htd-ind P* ∧ *intra-safe C*)
| *intra-safe* (*Call p*) = *True*
| *intra-safe* (*DynCom f*) = (*htd-ind f* ∧ (∀ *s*. *intra-safe* (*f s*)))
| *intra-safe* (*Guard f P C*) = (*mono-guard P* ∧ (*case C of*
Basic g ⇒ *comm-restrict-safe P g* ∧
point-eq-mod-safe P g (λ*s*. *lift-state* (*hst-mem s*,*hst-htd s*))
| - ⇒ *intra-safe C*))
| *intra-safe Throw* = *True*
| *intra-safe* (*Catch C D*) = (*intra-safe C* ∧ *intra-safe D*)

instance *state-ext* :: (*heap-state-type'*,*type*) *heap-state-type'* **..**

overloading
hs-mem-state ≡ *hst-mem*
hs-mem-update-state ≡ *hst-mem-update*
hs-htd-state ≡ *hst-htd*
hs-htd-update-state ≡ *hst-htd-update*
begin
definition *hs-mem-state* [*simp*]: *hs-mem-state* ≡ *hst-mem* ∘ *globals*
definition *hs-mem-update-state* [*simp*]: *hs-mem-update-state* ≡ *globals-update* ∘
hst-mem-update
definition *hs-htd-state*[*simp*]: *hs-htd-state* ≡ *hst-htd* ∘ *globals*
definition *hs-htd-update-state* [*simp*]: *hs-htd-update-state* ≡ *globals-update* ∘ *hst-htd-update*
end

instance *state-ext* :: (*heap-state-type*,*type*) *heap-state-type*
by *intro-classes auto*

primrec

intra-deps :: (*'s*,*'b*,*'c*) *com* ⇒ *'b set*
where
intra-deps Skip = {}
| *intra-deps* (*Basic f*) = {}
| *intra-deps* (*Spec r*) = {}
| *intra-deps* (*Seq C D*) = (*intra-deps C* ∪ *intra-deps D*)
| *intra-deps* (*Cond P C D*) = (*intra-deps C* ∪ *intra-deps D*)
| *intra-deps* (*While P C*) = *intra-deps C*
| *intra-deps* (*Call p*) = {*p*}
| *intra-deps* (*DynCom f*) = ⋃{*intra-deps* (*f s*) | *s*. *True*}
| *intra-deps* (*Guard f P C*) = *intra-deps C*
| *intra-deps Throw* = {}
| *intra-deps* (*Catch C D*) = (*intra-deps C* ∪ *intra-deps D*)

inductive-set
proc-deps :: (*'s*,*'b*,*'c*) *com* ⇒ (*'s*,*'b*,*'c*) *body* ⇒ *'b set*
for *C* :: (*'s*,*'b*,*'c*) *com*
and Γ :: (*'s*,*'b*,*'c*) *body*
where
x ∈ *intra-deps C* ⟹ *x* ∈ *proc-deps C* Γ
| ⟦ *x* ∈ *proc-deps C* Γ; Γ *x* = *Some D*; *y* ∈ *intra-deps D* ⟧ ⟹ *y* ∈ *proc-deps C* Γ

—-

lemma *point-eq-mod-refl* [*simp*]:
point-eq-mod f f X
by (*simp add*: *point-eq-mod-def*)

lemma *point-eq-mod-subs*:
⟦ *point-eq-mod f g Y*; *Y* ⊆ *X* ⟧ ⟹ *point-eq-mod f g X*
by (*force simp*: *point-eq-mod-def*)

lemma *point-eq-mod-trans*:
⟦ *point-eq-mod x y X*; *point-eq-mod y z X* ⟧ ⟹ *point-eq-mod x z X*
by (*force simp*: *point-eq-mod-def*)

lemma *mem-safe-NormalD*:
⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Normal t*; *mem-safe C* Γ;
¬ *exec-fatal C* Γ (*restrict-htd s X*) ⟧ ⟹
(Γ ⊢ ⟨*C*,(*Normal* (*restrict-htd s X*))⟩ ⇒ *Normal* (*restrict-htd t X*) ∧
point-eq-mod (*lift-state* (*hst-mem t*,*hst-htd t*))
(*lift-state* (*hst-mem s*,*hst-htd s*)) *X*)
by (*force simp*: *mem-safe-def restrict-safe-def restrict-safe-OK-def*)

lemma *mem-safe-AbruptD*:
⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Abrupt t*; *mem-safe C* Γ;
¬ *exec-fatal C* Γ (*restrict-htd s X*) ⟧ ⟹
(Γ ⊢ ⟨*C*,(*Normal* (*restrict-htd s X*))⟩ ⇒ *Abrupt* (*restrict-htd t X*) ∧
point-eq-mod (*lift-state* (*hst-mem t*,*hst-htd t*))

```
      (lift-state (hst-mem s,hst-htd s)) X)
  by (force simp: mem-safe-def restrict-safe-def restrict-safe-OK-def)

lemma mem-safe-FaultD:
  ⟦ Γ ⊢ ⟨C,Normal s⟩ ⇒ Fault f; mem-safe C Γ ⟧ ⟹
      exec-fatal C Γ (restrict-htd s X)
  by (force simp: mem-safe-def restrict-safe-def)

lemma mem-safe-StuckD:
  ⟦ Γ ⊢ ⟨C,Normal s⟩ ⇒ Stuck; mem-safe C Γ ⟧ ⟹
      exec-fatal C Γ (restrict-htd s X)
  by (force simp: mem-safe-def restrict-safe-def)

lemma lift-state-d-restrict [simp]:
  lift-state (h,(restrict-s d X)) = lift-state (h,d) |` X
 by (auto simp: lift-state-def restrict-map-def restrict-s-def split: s-heap-index.splits)

lemma dom-merge-restrict [simp]:
  (x ++ y) |` dom y = y
  by (rule map-add-restrict-dom-right)

lemma dom-compl-restrict [simp]:
  x |` (UNIV − dom x) = Map.empty
  by (force simp: restrict-map-def)

lemma lift-state-point-eq-mod:
  ⟦ point-eq-mod (lift-state (h,d)) (lift-state (h',d')) X ⟧ ⟹
      lift-state (h,d) |` (UNIV − X) =
          lift-state (h',d') |` (UNIV − X)
  by (auto simp: point-eq-mod-def restrict-map-def)

lemma htd-'-update-ind [simp]:
  htd-ind f ⟹ f (hst-htd-update x s) = f s
  by (simp add: htd-ind-def)

lemma sep-frame':
  assumes orig-spec: ∀ s. Γ ⊢ ⦃s. P (f ´(λx. x)) (lift-hst ´(λx. x))⦄
                                C
                                ⦃Q (g s ´(λx. x)) (lift-hst ´(λx. x))⦄
      and hi-f: htd-ind f and hi-g: htd-ind g
      and hi-g': ∀ s. htd-ind (g s)
      and safe: mem-safe (C::('s::heap-state-type,'b,'c) com) Γ
  shows ∀ s. Γ ⊢ ⦃s. (P (f ´(λx. x)) ∧* R (h ´(λx. x))) (lift-hst ´(λx. x))⦄
              C
              ⦃(Q (g s ´(λx. x)) ∧* R (h s)) (lift-hst ´(λx. x))⦄
  apply (standard, rule hoare-complete, simp only: valid-def, clarify)
proof −
  fix ta x
  assume ev: Γ⊢ ⟨C,Normal x⟩ ⇒ ta and
```

```
    pre: (P (f x) ∧* R (h x)) (lift-hst x)
  then obtain s0 and s1 where pre-P: P (f x) s0 and pre-R: R (h x) s1 and
    disj: s0 ⊥ s1 and m: lift-hst x = s1 ++ s0
   by (clarsimp simp: sep-conj-def map-ac-simps)
  with orig-spec hi-f have nofault: ¬ exec-fatal C Γ
    (restrict-htd x (dom s0))
   by (force simp: exec-fatal-def image-def lift-hst-def cvalid-def valid-def
                   restrict-htd-def
             dest: hoare-sound)
  show ta ∈ Normal ` {t. (Q (g x t) ∧* R (h x)) (lift-hst t)}
  proof (cases ta)
   case (Normal s)
   moreover from this ev safe nofault have ev': Γ ⊢
      ⟨C,Normal (x⦇ hst-htd := (restrict-s (hst-htd x) (dom s0)) ⦈)⟩ ⇒
        Normal (s⦇ hst-htd := (restrict-s (hst-htd s) (dom s0)) ⦈) and
      point-eq-mod (lift-state (hst-mem s,hst-htd s))
        (lift-state (hst-mem x,hst-htd x)) (dom s0)
     by (auto simp: restrict-htd-def dest: mem-safe-NormalD)
   moreover from this m disj have s1 = lift-hst s |` (UNIV − dom s0)
     apply(clarsimp simp: lift-hst-def)
     apply(subst lift-state-point-eq-mod)
      apply(fastforce dest: sym)
     apply(simp add: lift-hst-def lift-state-point-eq-mod map-add-restrict)
     apply(subst restrict-map-subdom, auto dest: map-disjD)
     done
   ultimately show ?thesis using orig-spec hi-f hi-g hi-g' pre-P pre-R m
     by (force simp: cvalid-def valid-def image-def lift-hst-def
                     map-disj-def
               intro: sep-conjI dest: hoare-sound)
  next
   case (Abrupt s) with ev safe nofault orig-spec pre-P hi-f m show ?thesis
     apply simp
     apply (drule spec)
     apply (drule hoare-sound)
     apply (drule-tac X=dom s0 in mem-safe-AbruptD)
     by (assumption+, force simp: valid-def cvalid-def lift-hst-def restrict-htd-def)
  next
   case (Fault f) with ev safe nofault show ?thesis
     by (force dest: mem-safe-FaultD)
  next
   case Stuck with ev safe nofault show ?thesis
     by (force dest: mem-safe-StuckD)
  qed
qed

lemma sep-frame:
  ⟦ k = (λs. (hst-mem s,hst-htd s));
    ∀ s. Γ ⊢ ⦃s. P (f ´(λx. x)) (lift-state (k ´(λx. x)))⦄
           C
```

$\{\!|Q\ (g\ s\ \acute{}(\lambda x.\ x))\ (\mathit{lift\text{-}state}\ (k\ \acute{}(\lambda x.\ x)))|\!\}$;
htd-ind f; *htd-ind g*; $\forall\, s.$ *htd-ind* $(g\ s)$;
mem-safe $(C::('s::\mathit{heap\text{-}state\text{-}type}, 'b, 'c)\ \mathit{com})\ \Gamma\ ]\!] \Longrightarrow$
$\forall\, s.\ \Gamma \vdash \{\!|s.\ (P\ (f\ \acute{}(\lambda x.\ x))\ \wedge^*\ R\ (h\ \acute{}(\lambda x.\ x)))\ (\mathit{lift\text{-}state}\ (k\ \acute{}(\lambda x.\ x)))|\!\}$
C
$\{\!|(Q\ (g\ s\ \acute{}(\lambda x.\ x))\ \wedge^*\ R\ (h\ s))\ (\mathit{lift\text{-}state}\ (k\ \acute{}(\lambda x.\ x)))|\!\}$
apply(*simp only*:)
apply(*fold lift-hst-def*)
apply(*erule* (*4*) *sep-frame'*)
done

lemma *point-eq-mod-safe* [*simp*]:
[\![*point-eq-mod-safe P f g*; *restrict-htd s X* $\in$ *P*; $x \notin X$]\!] $\Longrightarrow$
$g\ (f\ s)\ x = (g\ s)\ x$
apply(*simp add*: *point-eq-mod-safe-def point-eq-mod-def*)
apply(*cases x*, *force*)
done

lemma *comm-restrict-safe* [*simp*]:
[\![*comm-restrict-safe P f*; *restrict-htd s X* $\in$ *P*]\!] $\Longrightarrow$
restrict-htd $(f\ s\ )\ X = f\ (\mathit{restrict\text{-}htd}\ s\ X)$
by (*simp add*: *comm-restrict-safe-def comm-restrict-def*)

lemma *mono-guardD*:
[\![*mono-guard P*; *restrict-htd s X* $\in$ *P*]\!] $\Longrightarrow$ $s \in P$
by (*unfold mono-guard-def*, *fast*)

lemma *expr-htd-ind*:
expr-htd-ind P $\Longrightarrow$ *restrict-htd s X* $\in$ $P = (s \in P)$
by (*simp add*: *expr-htd-ind-def restrict-htd-def*)

lemmas *exec-other-intros* = *exec.intros*(*1−3*) *exec.intros*(*5−14*) *exec.intros*(*16−17*) *exec.intros*(*19−*)

lemma *exec-fatal-Seq*:
exec-fatal C Γ *s* $\Longrightarrow$ *exec-fatal* $(C;;D)\ \Gamma\ s$
by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*)

lemma *exec-fatal-Seq2*:
[\![$\Gamma \vdash \langle C, \mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ t$; *exec-fatal D* Γ *t*]\!] $\Longrightarrow$ *exec-fatal* $(C;;D)\ \Gamma\ s$
by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*)

lemma *exec-fatal-Cond*:
exec-fatal $(\mathit{Cond}\ P\ C\ D)\ \Gamma\ s = (\mathit{if}\ s \in P\ \mathit{then}\ \mathit{exec\text{-}fatal}\ C\ \Gamma\ s\ \mathit{else}$
exec-fatal D Γ *s*)
by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*
elim: *exec-Normal-elim-cases*)

lemma *exec-fatal-While*:
  ⟦ *exec-fatal C Γ s*; *s* ∈ *P* ⟧ ⟹ *exec-fatal* (*While P C*) *Γ s*
  by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*
        elim: *exec-Normal-elim-cases*)

lemma *exec-fatal-While2*:
  ⟦ *exec-fatal* (*While P C*) *Γ t*; *Γ* ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Normal t*; *s* ∈ *P* ⟧ ⟹
    exec-fatal (*While P C*) *Γ s*
  by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*
        elim: *exec-Normal-elim-cases*)

lemma *exec-fatal-Call*:
  ⟦ *Γ p* = *Some C*; *exec-fatal C Γ s* ⟧ ⟹ *exec-fatal* (*Call p*) *Γ s*
  by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*)

lemma *exec-fatal-DynCom*:
  exec-fatal (*f s*) *Γ s* ⟹ *exec-fatal* (*DynCom f*) *Γ s*
  by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*)

lemma *exec-fatal-Guard*:
  exec-fatal (*Guard f P C*) *Γ s* = (*s* ∈ *P* ⟶ *exec-fatal C Γ s*)
proof (*cases s* ∈ *P*)
  case *True* **thus** *?thesis*
    by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*
          elim: *exec-Normal-elim-cases*)
next
  case *False* **thus** *?thesis*
    by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*)
qed

lemma *restrict-safe-Guard*:
  assumes *restrict*: *restrict-safe s t C Γ*
  shows *restrict-safe s t* (*Guard f P C*) *Γ*
proof (*cases t*)
  case (*Normal s*) **with** *restrict* **show** *?thesis*
    by (*force simp*: *restrict-safe-def restrict-safe-OK-def exec-fatal-Guard*
          intro: *exec-other-intros*)
next
  case (*Abrupt s*) **with** *restrict* **show** *?thesis*
    by (*force simp*: *restrict-safe-def restrict-safe-OK-def exec-fatal-Guard*
          intro: *exec-other-intros*)
next
  case (*Fault f*) **with** *restrict* **show** *?thesis*
    by (*force simp*: *restrict-safe-def exec-fatal-Guard*)
next
  case *Stuck* **with** *restrict* **show** *?thesis*
    by (*force simp*: *restrict-safe-def exec-fatal-Guard*)
qed

lemma *restrict-safe-Guard2*:
  ⟦ *s* ∉ *P*; *mono-guard P* ⟧ ⟹ *restrict-safe s* (*Fault f*) (*Guard f P C*) Γ
  by (*force simp*: *restrict-safe-def exec-fatal-def intro*: *exec-other-intros*
        dest: *mono-guardD*)

lemma *exec-fatal-Catch*:
  exec-fatal C Γ *s* ⟹ *exec-fatal* (*TRY C CATCH D END*) Γ *s*
  by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*)

lemma *exec-fatal-Catch2*:
  ⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Abrupt t*; *exec-fatal D* Γ *t* ⟧ ⟹
    exec-fatal (*TRY C CATCH D END*) Γ *s*
  by (*force simp*: *exec-fatal-def intro*: *exec-other-intros*)

lemma *intra-safe-restrict* [*rule-format*]:
  assumes *safe-env*: ⋀*n C*. Γ *n* = *Some C* ⟹ *intra-safe C* **and**
    exec: Γ ⊢ ⟨*C*,*s*⟩ ⇒ *t*
  shows ∀ *s'*. *s* = *Normal s'* ⟶ *intra-safe C* ⟶ *restrict-safe s' t C* Γ
using *exec*
proof *induct*
  case (*Skip s*) **thus** *?case*
    by (*force simp*: *restrict-safe-def restrict-safe-OK-def intro*: *exec-other-intros*)
next
  case (*Guard s' P C t f*) **show** *?case*
  proof (*cases* ∃ *g*. *C* = *Basic g*)
    case *False* **with** *Guard* **show** *?thesis*
      by − (*clarsimp*, *split com.splits*, *auto dest*: *restrict-safe-Guard*)
  next
    case *True* **with** *Guard* **show** *?thesis*
      by (*cases t*) (*force simp*: *restrict-safe-def restrict-safe-OK-def*
                point-eq-mod-safe-def exec-fatal-Guard
              intro: *exec-other-intros*
              elim: *exec-Normal-elim-cases*,
            (*fast elim*: *exec-Normal-elim-cases*)+)
  qed
next
  case (*GuardFault C f P s*) **thus** *?case*
    by (*force dest*: *restrict-safe-Guard2*)
next
  case (*FaultProp C f*) **thus** *?case* **by** *simp*
next
  case (*Basic f s*) **thus** *?case*
    by (*force simp*: *restrict-safe-def restrict-safe-OK-def point-eq-mod-safe-def*
          intro: *exec-other-intros*)
next
  case (*Spec r s t*) **thus** *?case*
    by (*fastforce simp*: *mem-safe-def intro*: *exec.Spec*)
next

```
  case (SpecStuck r s) thus ?case
    by (simp add: exec.SpecStuck mem-safe-StuckD restrict-safe-def)
next
  case (Seq C s sa D ta) show ?case
  proof (cases sa)
    case (Normal s') with Seq show ?thesis
      by (cases ta)
        (clarsimp simp: restrict-safe-def restrict-safe-OK-def,
         (drule-tac x=X in spec)+, auto intro: exec-other-intros point-eq-mod-trans
                                               exec-fatal-Seq exec-fatal-Seq2)+
  next
    case (Abrupt s') with Seq show ?thesis
      by (force simp: restrict-safe-def restrict-safe-OK-def
                intro: exec-other-intros dest: exec-fatal-Seq
                elim: exec-Normal-elim-cases)
  next
    case (Fault f) with Seq show ?thesis
      by (force simp: restrict-safe-def dest: exec-fatal-Seq
                elim: exec-Normal-elim-cases)
  next
    case Stuck with Seq show ?thesis
      by (force simp: restrict-safe-def dest: exec-fatal-Seq
                elim: exec-Normal-elim-cases)
  qed
next
  case (CondTrue s P C t D) thus ?case
    by (cases t)
      (auto simp: restrict-safe-def restrict-safe-OK-def exec-fatal-Cond
           intro: exec-other-intros dest: expr-htd-ind split: if-split-asm)
next
  case (CondFalse s P C t D) thus ?case
    by (cases t)
      (auto simp: restrict-safe-def restrict-safe-OK-def exec-fatal-Cond
           intro: exec-other-intros dest: expr-htd-ind split: if-split-asm)
next
  case (WhileTrue P C s s' t) show ?case
  proof (cases s')
    case (Normal sa) with WhileTrue show ?thesis
      by (cases t)
        (clarsimp simp: restrict-safe-def restrict-safe-OK-def,
         (drule-tac x=X in spec)+, auto simp: expr-htd-ind intro: exec-other-intros
              point-eq-mod-trans exec-fatal-While exec-fatal-While2)+
  next
    case (Abrupt sa) with WhileTrue show ?thesis
      by (force simp: restrict-safe-def restrict-safe-OK-def expr-htd-ind
                intro: exec-other-intros exec-fatal-While
                elim: exec-Normal-elim-cases)
  next
    case (Fault f) with WhileTrue show ?thesis
```

```
    by (force simp: restrict-safe-def expr-htd-ind intro: exec-fatal-While)
  next
    case Stuck with WhileTrue show ?thesis
      by (force simp: restrict-safe-def expr-htd-ind intro: exec-fatal-While)
  qed
next
  case (WhileFalse P C s) thus ?case
    by (force simp: restrict-safe-def restrict-safe-OK-def expr-htd-ind
          intro: exec-other-intros)
next
  case (Call C p s t) with safe-env show ?case
    by (cases t)
       (auto simp: restrict-safe-def restrict-safe-OK-def
            intro: exec-fatal-Call exec-other-intros)
next
  case (CallUndefined p s) thus ?case
    by (force simp: restrict-safe-def exec-fatal-def intro: exec-other-intros)

next
  case (StuckProp C) thus ?case by simp
next
  case (DynCom f s t) thus ?case
    by (cases t)
       (auto simp: restrict-safe-def restrict-safe-OK-def
                   restrict-htd-def
            intro!: exec-other-intros exec-fatal-DynCom)
next
  case (Throw s) thus ?case
    by (force simp: restrict-safe-def restrict-safe-OK-def intro: exec-other-intros)
next
  case (AbruptProp C s) thus ?case by simp
next
  case (CatchMatch C D s s' t) thus ?case
    by (cases t)
       (clarsimp simp: restrict-safe-def, drule-tac x=X in spec,
        auto simp: restrict-safe-OK-def intro: exec-other-intros point-eq-mod-trans
            dest: exec-fatal-Catch exec-fatal-Catch2)+
next
  case (CatchMiss C s t D) thus ?case
    by (cases t)
       (clarsimp simp: restrict-safe-def, drule-tac x=X in spec,
        auto simp: restrict-safe-OK-def intro: exec-other-intros
            dest: exec-fatal-Catch)+
qed

lemma intra-mem-safe:
  ⟦ ⋀n C. Γ n = Some C ⟹ intra-safe C; intra-safe C ⟧ ⟹ mem-safe C Γ
  by (force  simp: mem-safe-def intro: intra-safe-restrict)
```

lemma *point-eq-mod-safe-triv*:
(⋀*s*. *g* (*f s*) = *g s*) ⟹ *point-eq-mod-safe P f g*
by (*simp add*: *point-eq-mod-safe-def point-eq-mod-def*)

lemma *comm-restrict-safe-triv*:
(⋀*s X*. *f* (*s*(| *hst-htd* := *restrict-s* (*hst-htd s*) *X* |)) =
(*f s*)(| *hst-htd* := *restrict-s* (*hst-htd* (*f s*)) *X* |)) ⟹ *comm-restrict-safe P f*
by (*force simp*: *comm-restrict-safe-def comm-restrict-def restrict-htd-def*)

lemma *mono-guard-UNIV* [*simp*]:
mono-guard UNIV
by (*force simp*: *mono-guard-def*)

lemma *mono-guard-triv*:
(⋀*s X*. *s*(| *hst-htd* := *X* |) ∈ *g* ⟹ *s* ∈ *g*) ⟹ *mono-guard g*
by (*unfold mono-guard-def*, *unfold restrict-htd-def*, *fast*)

lemma *mono-guard-triv2*:
(⋀*s X*. *s*(| *hst-htd* := *X* |) ∈ *g* = ((*s*::′*a*::*heap-state-type*′) ∈ *g*)) ⟹
mono-guard g
by (*unfold mono-guard-def*, *unfold restrict-htd-def*, *fast*)

lemma *dom-restrict-s*:
x ∈ *dom-s* (*restrict-s d X*) ⟹ *x* ∈ *dom-s d* ∧ *x* ∈ *X*
by (*auto simp*: *restrict-s-def dom-s-def split*: *if-split-asm*)

lemma *mono-guard-ptr-safe*:
⟦ ⋀*s*. *d s* = *hst-htd* (*s*::′*a*::*heap-state-type*); *htd-ind p* ⟧ ⟹
mono-guard {*s*. *ptr-safe* (*p s*) (*d s*)}
by (*auto simp*: *mono-guard-def ptr-safe-def restrict-htd-def dest*: *subsetD dom-restrict-s*)

lemma *point-eq-mod-safe-ptr-safe-update*:
⟦ *d* = (*hst-htd*::′*a*::*heap-state-type* ⇒ *heap-typ-desc*);
m = (λ*s*. *hst-mem-update* (*heap-update* (*p s*) ((*v s*)::′*b*::*mem-type*)) *s*);
h = *hst-mem*; *k* = (λ*s*. *lift-state* (*h s*,*d s*)); *htd-ind p* ⟧ ⟹
point-eq-mod-safe {*s*. *ptr-safe* (*p s*) (*d s*)} *m k*
apply (*clarsimp simp*: *point-eq-mod-safe-def point-eq-mod-def ptr-safe-def heap-update-def*
restrict-htd-def lift-state-def
intro!: *heap-update-nmem-same*
split: *s-heap-index.splits*)
apply(*subgoal-tac* (*a*,*SIndexVal*) ∈ *s-footprint* (*p s*))
apply(*drule* (*1*) *subsetD*)
apply(*drule dom-restrict-s*, *clarsimp*)
apply(*drule intvlD*, *clarsimp*)
apply(*erule s-footprintI2*)
done

lemma *field-ti-s-sub-typ*:
field-lookup (*export-uinfo* (*typ-info-t TYPE*(′*b*::*mem-type*))) *f* 0 = *Some* (*typ-uinfo-t*

```
TYPE('a),b) ⟹
    s-footprint ((Ptr &(p→f))::'a::mem-type ptr) ⊆ s-footprint (p::'b ptr)
  by (drule field-ti-s-sub) (simp add: s-footprint-def)

lemma ptr-safe-mono:
  ⟦ ptr-safe (p::'a::mem-type ptr) d; field-lookup (typ-info-t TYPE('a)) f 0
      = Some (t,n); export-uinfo t = typ-uinfo-t TYPE('b) ⟧ ⟹
      ptr-safe ((Ptr &(p→f))::'b::mem-type ptr) d
  unfolding ptr-safe-def
  by (drule field-lookup-export-uinfo-Some) (auto dest: field-ti-s-sub-typ)

lemma point-eq-mod-safe-ptr-safe-update-fl:
  ⟦ d = (hst-htd::'a::heap-state-type ⇒ heap-typ-desc);
     m = (λs. hst-mem-update (heap-update (Ptr &((p s)→f)) ((v s)::'b::mem-type))
s);
      h = hst-mem; k = (λs. lift-state (h s,d s)); htd-ind p;
      field-lookup (typ-info-t TYPE('c)) f 0 = Some (t,n);
      export-uinfo t = typ-uinfo-t TYPE('b) ⟧ ⟹
      point-eq-mod-safe {s. ptr-safe ((p::'a ⇒ 'c::mem-type ptr) s) (d s)} m k
  apply(drule (3) point-eq-mod-safe-ptr-safe-update)
   apply(fastforce simp: htd-ind-def)
  apply(fastforce simp: point-eq-mod-safe-def intro!: ptr-safe-mono)
  done

context
begin

private method m =
  (clarsimp simp: ptr-retyp-d-eq-snd ptr-retyp-footprint list-map-eq,
   erule notE,
   drule intvlD, clarsimp,
   (rule s-footprintI; assumption?),
   subst (asm) unat-of-nat,
   (subst (asm) mod-less; assumption?),
   subst len-of-addr-card,
   erule less-trans,
   simp)

lemma point-eq-mod-safe-ptr-safe-tag:
  ⟦ d = (hst-htd::'a::heap-state-type ⇒ heap-typ-desc); h = hst-mem;
     m = (λs. hst-htd-update (ptr-retyp (p s)) s);
     k = (λs. lift-state (h s,d s));
     htd-ind p ⟧ ⟹
   point-eq-mod-safe {s. ptr-safe ((p s)::'b::mem-type ptr) (d s)} m k
  supply if-split-asm[split]
  supply unsigned-of-nat [simp del]
  apply(clarsimp simp: point-eq-mod-safe-def point-eq-mod-def ptr-safe-def)
  apply(subgoal-tac (a,b) ∉ s-footprint (p (restrict-htd s X)))
   prefer 2
```

apply(*fastforce simp*: *restrict-htd-def dest*: *dom-restrict-s*)
apply(*clarsimp simp*: *restrict-htd-def lift-state-def split*: *s-heap-index.split option.splits*)
apply (*safe*; *m?*)
apply(*fastforce simp*: *ptr-retyp-d-eq-fst dest*!: *intvlD dest*: *s-footprintI2*)
apply(*fastforce simp*: *ptr-retyp-d-eq-fst*)
apply(*subst* (*asm*) *ptr-retyp-d-eq-snd*, *clarsimp*)
done

end

lemma *comm-restrict-safe-ptr-safe-tag*:
fixes *d*::*'a*::*heap-state-type* ⇒ *heap-typ-desc*
assumes
fun-d: *d* = *hst-htd* **and**
fun-upd: *m* = (λ*s*. *hst-htd-update* (*ptr-retyp* (*p s*)) *s*) **and**
ind: *htd-ind p* **and**
upd: ⋀*d d'* (*s*::*'a*).
hst-htd-update (*d s*) (*hst-htd-update* (*d' s*) *s*) = *hst-htd-update* ((*d s*) ∘ (*d' s*)) *s*
shows *comm-restrict-safe* {*s*. *ptr-safe* ((*p s*)::*'b*::*mem-type ptr*) (*d s*)} *m*
proof −
{
fix *s X*
assume *ptr-safe* (*p* (*restrict-htd s X*)) (*d* (*restrict-htd s X*))
moreover from *ind*
have *p*: *p* (*restrict-htd s X*) = *p s*
by (*simp add*: *restrict-htd-def*)
ultimately
have *ptr-retyp* (*p s*) (*restrict-s* (*hst-htd s*) *X*) = *restrict-s* (*ptr-retyp* (*p s*) (*hst-htd s*)) *X*
using *fun-d*
supply *unsigned-of-nat* [*simp del*]
apply −
apply(*rule ext*)
apply(*clarsimp simp*: *point-eq-mod-safe-def point-eq-mod-def ptr-safe-def restrict-htd-def*)
subgoal for *x*
apply(*cases x* ∉ {*ptr-val* (*p s*)..+*size-of TYPE*(*'b*)})
apply(*clarsimp simp*: *ptr-retyp-d restrict-map-def restrict-s-def*)
apply(*subst ptr-retyp-d*; *simp*)
apply(*clarsimp simp*: *ptr-retyp-footprint restrict-map-def restrict-s-def*)
apply(*subst ptr-retyp-footprint*, *simp*)
apply(*rule conjI*)
apply(*subgoal-tac* (*x*,*SIndexVal*) ∈ *s-footprint* (*p s*))
apply(*fastforce simp*: *dom-s-def*)
apply(*fastforce dest*: *intvlD elim*: *s-footprintI2*)
apply(*rule ext*)
apply(*clarsimp simp*: *map-add-def list-map-eq*)

apply(*subgoal-tac* (*x*,*SIndexTyp y*) ∈ *s-footprint* (*p s*))
apply(*fastforce simp*: *dom-s-def split*: *if-split-asm*)
apply(*drule intvlD*, *clarsimp*)
apply(*rule s-footprintI*; *assumption?*)
apply(*metis len-of-addr-card less-trans max-size mod-less word-unat.eq-norm*)
done
done
hence ((*ptr-retyp* (*p s*) ∘ (λ*x* -. *x*) (*restrict-s* (*hst-htd s*) *X*)::*heap-typ-desc* ⇒ *heap-typ-desc*) =
(λ*x* -. *x*) (*restrict-s* (*ptr-retyp* (*p s*) (*hst-htd s*)) *X*))
by − (*rule ext*, *simp*)
moreover from *upd* **have** *hst-htd-update* (*ptr-retyp* (*p s*))
(*hst-htd-update* ((λ*x* -. *x*) (*restrict-s* (*hst-htd s*) *X*)) *s*) =
hst-htd-update (((*ptr-retyp* (*p s*)) ∘ ((λ*x* -. *x*) (*restrict-s* (*hst-htd s*) *X*)))) *s* **.**
moreover from *upd*
have
hst-htd-update ((λ*x* -. *x*) (*restrict-s* (*ptr-retyp* (*p s*) (*hst-htd s*)) *X*))
(*hst-htd-update* (*ptr-retyp* (*p s*)) *s*) =
hst-htd-update (((λ*x* -. *x*) (*restrict-s* ((*ptr-retyp* (*p s*) (*hst-htd s*))) *X*)) ∘ (*ptr-retyp* (*p s*))) *s* **.**
ultimately have *m* (*restrict-htd s X*) = *restrict-htd* (*m s*) *X* **using** *fun-d fun-upd upd p*
by (*simp add*: *restrict-htd-def o-def*)
}
thus *?thesis*
by (*simp only*: *comm-restrict-safe-def comm-restrict-def*, *auto*)
qed

lemmas *intra-sc* = *hrs-comm comp-def hrs-htd-update-htd-update*
point-eq-mod-safe-triv comm-restrict-safe-triv mono-guard-triv2
mono-guard-ptr-safe point-eq-mod-safe-ptr-safe-update
point-eq-mod-safe-ptr-safe-tag comm-restrict-safe-ptr-safe-tag
point-eq-mod-safe-ptr-safe-update-fl

declare *expr-htd-ind-def* [*iff*]
declare *htd-ind-def* [*iff*]

lemma *proc-deps-Skip* [*simp*]:
proc-deps Skip Γ = {}
by (*force elim*: *proc-deps.induct*)

lemma *proc-deps-Basic* [*simp*]:
proc-deps (*Basic f*) Γ = {}
by (*force elim*: *proc-deps.induct*)

lemma *proc-deps-Spec* [*simp*]:
proc-deps (*Spec r*) Γ = {}
by (*force elim*: *proc-deps.induct*)

lemma *proc-deps-Seq* [*simp*]:
proc-deps (*Seq C D*) Γ = *proc-deps C* Γ ∪ *proc-deps D* Γ
proof
show *proc-deps* (*C*;; *D*) Γ ⊆ *proc-deps C* Γ ∪ *proc-deps D* Γ
by − (*standard*, *erule proc-deps.induct*, *auto intro*: *proc-deps.intros*)
next
show *proc-deps C* Γ ∪ *proc-deps D* Γ ⊆ *proc-deps* (*C*;; *D*) Γ
by *auto* (*erule proc-deps.induct*, *auto intro*: *proc-deps.intros*)+
qed

lemma *proc-deps-Cond* [*simp*]:
proc-deps (*Cond P C D*) Γ = *proc-deps C* Γ ∪ *proc-deps D* Γ
proof
show *proc-deps* (*Cond P C D*) Γ ⊆ *proc-deps C* Γ ∪ *proc-deps D* Γ
by (*standard*, *erule proc-deps.induct*, *auto intro*: *proc-deps.intros*)
next
show *proc-deps C* Γ ∪ *proc-deps D* Γ ⊆ *proc-deps* (*Cond P C D*) Γ
by *auto* (*erule proc-deps.induct*, *auto intro*: *proc-deps.intros*)+
qed

lemma *proc-deps-While* [*simp*]:
proc-deps (*While P C*) Γ = *proc-deps C* Γ
by *auto* (*erule proc-deps.induct*, *auto intro*: *proc-deps.intros*)+

lemma *proc-deps-Guard* [*simp*]:
proc-deps (*Guard f P C*) Γ = *proc-deps C* Γ
by *auto* (*erule proc-deps.induct*, *auto intro*: *proc-deps.intros*)+

lemma *proc-deps-Throw* [*simp*]:
proc-deps Throw Γ = {}
by (*force elim*: *proc-deps.induct*)

lemma *proc-deps-Catch* [*simp*]:
proc-deps (*Catch C D*) Γ = *proc-deps C* Γ ∪ *proc-deps D* Γ
proof
show *proc-deps* (*Catch C D*) Γ ⊆ *proc-deps C* Γ ∪ *proc-deps D* Γ
by (*standard*, *erule proc-deps.induct*, *auto intro*: *proc-deps.intros*)
next
show *proc-deps C* Γ ∪ *proc-deps D* Γ ⊆ *proc-deps* (*Catch C D*) Γ
by *auto* (*erule proc-deps.induct*, *auto intro*: *proc-deps.intros*)+
qed

lemma *proc-deps-Call* [*simp*]:
proc-deps (*Call p*) Γ = {*p*} ∪ (*case* Γ *p of Some C* ⇒
proc-deps C (Γ(*p* := *None*)) | - ⇒ {}) (**is** *?X* = *?Y* ∪ *?Z*)
proof
note *proc-deps.intros*[*intro*]
show *?X* ⊆ *?Y* ∪ *?Z*
apply (*rule subsetI*, *erule proc-deps.induct*, *fastforce*)

```
    subgoal for x' x D y
      apply (cases x=p)
       apply (auto split: option.splits)
      done
    done
next
  show ?Y ∪ ?Z ⊆ ?X
    apply (clarsimp, standard)
  proof −
    show p ∈ ?X by (force intro: proc-deps.intros)
  next
    show ?Z ⊆ ?X
      by (split option.splits, standard, force intro: proc-deps.intros)
        (clarify, erule proc-deps.induct;
         force intro: proc-deps.intros split: if-split-asm)
  qed
qed

lemma proc-deps-DynCom [simp]:
  proc-deps (DynCom f) Γ = ⋃{proc-deps (f s) Γ | s. True}
  by (rule equalityI; clarsimp; erule proc-deps.induct; force intro: proc-deps.intros)

lemma proc-deps-restrict:
  proc-deps C Γ ⊆ proc-deps C (Γ(p := None)) ∪ proc-deps (Call p) Γ
proof
  fix xa
  assume mem: xa ∈ proc-deps C Γ
  hence ∀ p. xa ∈ proc-deps C (Γ(p := None)) ∪ proc-deps (Call p) Γ (is ?X)
  using mem
  proof induct
    fix x
    assume x ∈ intra-deps C
    thus ∀ p. x ∈ proc-deps C (Γ(p := None)) ∪ proc-deps (Call p) Γ
      by (force intro: proc-deps.intros)
  next
    fix D x y
    assume x:
      x ∈ proc-deps C Γ
     x ∈ proc-deps C Γ ⟹ ∀ p. x ∈ proc-deps C (Γ(p := None)) ∪ proc-deps (Call
p) Γ
      Γ x = Some D
      y ∈ intra-deps D
      y ∈ proc-deps C Γ
    show ∀ p. y ∈ proc-deps C (Γ(p := None)) ∪ proc-deps (Call p) Γ
    proof clarify
      fix p
      assume y: y ∉ proc-deps (Call p) Γ
      show y ∈ proc-deps C (Γ(p := None))
      proof (cases x=p)
```

```
      case True with x y show ?thesis
        by (force intro: proc-deps.intros)
    next
      case False with x y show ?thesis
        by (clarsimp, drule-tac x=p in spec)
           (auto intro: proc-deps.intros split: option.splits)
    qed
  qed
 qed
 thus xa ∈ proc-deps C (Γ(p := None)) ∪ proc-deps (Call p) Γ by simp
qed

lemma exec-restrict:
 assumes exec: Γ' ⊢ ⟨C,s⟩ ⇒ t
 shows ⋀Γ X. ⟦ Γ' = Γ |` X; proc-deps C Γ ⊆ X ⟧ ⟹ Γ ⊢ ⟨C,s⟩ ⇒ t
using exec
proof induct
 case (Call p C s t)
 thus ?case
   using proc-deps-restrict [of C Γ p] by (force intro: exec-other-intros)
qed (force intro: exec-other-intros)+

lemma exec-restrict2:
 assumes exec: Γ ⊢ ⟨C,s⟩ ⇒ t
 shows ⋀X. proc-deps C Γ ⊆ X ⟹ Γ |` X ⊢ ⟨C,s⟩ ⇒ t
using exec
proof induct
 case (Call p C s t) thus ?case
 using proc-deps-restrict [of C Γ p]
   by (auto intro!: exec-other-intros split: option.splits)
next
 case (DynCom f s t) thus ?case
   by − (rule exec.intros, simp, blast)
qed (auto intro: exec-other-intros)

lemma exec-restrict-eq:
 Γ |` proc-deps C Γ ⊢ ⟨C,s⟩ ⇒ t = Γ ⊢ ⟨C,s⟩ ⇒ t
 by (fast intro: exec-restrict exec-restrict2)

lemma mem-safe-restrict:
 mem-safe C Γ = mem-safe C (Γ |` proc-deps C Γ)
 by (auto simp: mem-safe-def restrict-safe-def restrict-safe-OK-def
                exec-restrict-eq exec-fatal-def
          split: xstate.splits)

end


theory StructSupport
```

imports *SepCode SepInv*
begin

lemma *field-lookup-list-Some2*:
fn ∉ *dt-snd* ‘ *set ts* ⟹
field-lookup-list (*ts*@[*DTuple t fn d*]) (*fn # fs*) *m* = *field-lookup t fs* (*m* + *size-td-list ts*)
apply(*induct ts arbitrary*: *m*; *clarsimp split*: *option.splits*)
subgoal for *a list m*
apply(*safe*; *cases a*; *clarsimp simp*: *ac-simps split*: *if-split-asm*)
done
done

lemma *field-lookup-list-mismatch*:
f ≠ *fn* ⟹ *field-lookup-list* (*ts*@[*DTuple t f d*]) (*fn # fs*) *m* =
field-lookup-list ts (*fn # fs*) *m*
by (*induct ts arbitrary*: *m*, *auto split*: *option.split*)

lemma *fnl-set*:
set (*CompoundCTypes.field-names-list* (*TypDesc algn* (*TypAggregate xs*) *tn*)) = *dt-snd* ‘ *set xs*
by (*auto simp*: *CompoundCTypes.field-names-list-def*)

lemma *fnl-extend-ti*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *aggregate tag* ⟧ ⟹
field-lookup (*extend-ti tag t algn fn d*) (*f # fs*) *m* =
(*if f*=*fn then field-lookup t fs* (*size-td tag*+*m*) *else field-lookup tag* (*f # fs*) *m*)
apply(*cases tag*)
subgoal for *x1 typ-struct xs*
apply(*cases typ-struct*; *simp*)
apply(*simp add*: *ac-simps field-lookup-list-Some2 fnl-set*)
apply(*clarsimp simp*: *field-lookup-list-append split*: *option.splits*)
done
done

lemma *fnl-extend-ti-mismatch*:
f ≠ *fn* ⟹ *field-lookup* (*extend-ti tag t algn fn d*) (*f # fs*) *m* = *field-lookup tag* (*f # fs*) *m*
apply (*cases tag*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*)
apply *clarsimp*
apply (*simp add*: *field-lookup-list-mismatch*)
done
done

lemma *fl-ti-pad-combine*:
⟦ *hd f* ≠ *CHR* ''!''; *aggregate tag* ⟧ ⟹
field-lookup (*ti-pad-combine n tag*) (*f*#*fs*) *m* = *field-lookup tag* (*f*#*fs*) *m*

by (*auto simp*: *ti-pad-combine-def Let-def fnl-extend-ti foldl-append-nmem*)

lemma *fl-ti-typ-combine*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *aggregate tag* ⟧ ⟹
field-lookup (*ti-typ-combine* (*t-b*::′*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*f*#*fs*) *m* =
(*if f*=*fn*
then field-lookup (*adjust-ti* (*typ-info-t TYPE*(′*b*)) *f-ab f-upd-ab*) *fs* (*size-td tag*+*m*)
else field-lookup tag (*f* # *fs*) *m*)
by (*simp add*: *ti-typ-combine-def Let-def fnl-extend-ti*)

lemma *fl-ti-typ-combine-match*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *aggregate tag* ⟧ ⟹
field-lookup (*ti-typ-combine* (*t-b*::′*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*fn*#*fs*) *m* =
field-lookup (*adjust-ti* (*typ-info-t TYPE*(′*b*)) *f-ab f-upd-ab*) *fs* (*size-td tag*+*m*)
by (*simp add*: *fl-ti-typ-combine*)

lemma *fl-ti-typ-combine-mismatch*:
⟦ *f* ≠ *fn* ⟧ ⟹
field-lookup (*ti-typ-combine* (*t-b*::′*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*f*#*fs*) *m* =
field-lookup tag (*f* # *fs*) *m*
unfolding *ti-typ-combine-def Let-def* **by** (*erule fnl-extend-ti-mismatch*)

lemma *fl-ti-typ-pad-combine*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *hd f* ≠ *CHR* ′′!′′; *hd fn* ≠ *CHR* ′′!′′;
aggregate tag ⟧ ⟹
field-lookup (*ti-typ-pad-combine* (*t-b*::′*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*f*#*fs*) *m* =
(*if f*=*fn*
then field-lookup (*adjust-ti* (*typ-info-t TYPE*(′*b*)) *f-ab f-upd-ab*) *fs*
(*padup* (*max* (2 ^ *algn*) (*align-of TYPE*(′*b*)))
(*size-td tag*) + *size-td tag*+*m*)
else field-lookup tag (*f* # *fs*) *m*)
unfolding *ti-typ-pad-combine-def Let-def*
by (*subst fl-ti-typ-combine*; *clarsimp*) (*simp add*: *fl-ti-pad-combine size-td-ti-pad-combine*)

lemma *fl-ti-typ-pad-combine-match*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *hd fn* ≠ *CHR* ′′!′′;
aggregate tag ⟧ ⟹
field-lookup (*ti-typ-pad-combine* (*t-b*::′*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*fn*#*fs*) *m* =
field-lookup (*adjust-ti* (*typ-info-t TYPE*(′*b*)) *f-ab f-upd-ab*) *fs*
(*padup* (*max* (2 ^ *algn*) (*align-of TYPE*(′*b*)))
(*size-td tag*) + *size-td tag*+*m*)

by (*simp add*: *fl-ti-typ-pad-combine*)

lemma *fl-ti-typ-pad-combine-mismatch*:
⟦ *hd f* ≠ *CHR* ''!''; *aggregate tag*; *f* ≠ *fn* ⟧ ⟹
field-lookup (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*f*#*fs*) *m* =
field-lookup tag (*f* # *fs*) *m*
apply(*unfold ti-typ-pad-combine-def Let-def*)
apply(*subst fl-ti-typ-combine-mismatch*)
apply *assumption*
apply(*simp add*: *fl-ti-pad-combine size-td-ti-pad-combine*)
done

lemma *field-lookup-map-align-cons*: *field-lookup* (*map-align g t*) (*f*#*fs*) *m* = *field-lookup t* (*f*#*fs*) *m*
by (*cases t*) *clarsimp*

lemma *fl-final-pad*:
⟦ *hd f* ≠ *CHR* ''!''; *aggregate tag* ⟧ ⟹
field-lookup (*final-pad algn tag*) (*f*#*fs*) *m* = *field-lookup tag* (*f*#*fs*) *m*
by (*clarsimp simp*: *final-pad-def Let-def fl-ti-pad-combine field-lookup-map-align-cons*)

lemma *field-lookup-adjust-ti2'* [*rule-format*]:
∀ *fn m s n*. *field-lookup ti fn m* = *Some* (*s*,*n*) ⟶
(*field-lookup* (*adjust-ti ti f g*) *fn m* = *Some* (*adjust-ti s f g*,*n*))
∀ *fn m s n*. *field-lookup-struct st fn m* = *Some* (*s*,*n*) ⟶
field-lookup-struct (*map-td-struct* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *st*) *fn m* = *Some* (*adjust-ti s f g*,*n*)
∀ *fn m s n*. *field-lookup-list ts fn m* = *Some* (*s*,*n*) ⟶
field-lookup-list (*map-td-list* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *ts*) *fn m* = *Some* (*adjust-ti s f g*,*n*)
∀ *fn m s n*. *field-lookup-tuple x fn m* = *Some* (*s*,*n*) ⟶
field-lookup-tuple (*map-td-tuple* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *x*) *fn m* = *Some* (*adjust-ti s f g*,*n*)
apply(*induct ti* **and** *st* **and** *ts* **and** *x*, *all* ‹*clarsimp*›)
apply(*clarsimp simp*: *adjust-ti-def*)
apply(*clarsimp split*: *option.splits*)
apply(*fastforce simp*: *split-DTuple-all simp flip*: *adjust-ti-def split*: *if-split-asm*
dest: *field-lookup-adjust-ti*)
apply (*clarsimp simp flip*: *adjust-ti-def*)
done

lemma *field-lookup-adjust-ti2*:
field-lookup t fn m = *Some* (*s*,*n*) ⟹
field-lookup (*adjust-ti t f g*) *fn m* = *Some* (*adjust-ti s f g*,*n*)
by (*simp add*: *field-lookup-adjust-ti2'*)

lemma *fl-update*:

field-lookup (*adjust-ti ti f g*) *fs m* =
(*case-option None* (λ(*t*,*n*). *Some* (*adjust-ti t f g*,*n*)) (*field-lookup ti fs m*))
apply(*clarsimp split*: *option.splits*)
apply *safe*
apply(*rule ccontr*, *clarsimp*)
apply(*drule field-lookup-adjust-ti*, *clarsimp*)
apply(*erule field-lookup-adjust-ti2*)
done

lemmas *fl-simps* = *fl-final-pad fl-ti-pad-combine*
fl-ti-typ-combine-match fl-ti-typ-combine-mismatch
fl-ti-typ-pad-combine-match fl-ti-typ-pad-combine-mismatch

lemma *access-ti-props-simps* [*simp*]:
∀ *g x*. *access-ti* (*adjust-ti* (*tag*::′*a xtyp-info*) (*f*::′*b* ⇒ ′*a*) *g*) *x* = *access-ti tag* (*f x*)
∀ *g x*. *access-ti-struct* (*map-td-struct* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) (*st*::′*a xtyp-info-struct*)) *x* = *access-ti-struct st* (*f x*)
∀ *g x*. *access-ti-list* (*map-td-list* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) (*ts*::′*a xtyp-info-tuple list*)) *x* = *access-ti-list ts* (*f x*)
∀ *g x*. *access-ti-tuple* (*map-td-tuple* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) (*k*::′*a xtyp-info-tuple*)) *x* = *access-ti-tuple k* (*f x*)
unfolding *adjust-ti-def*
by (*induct tag* **and** *st* **and** *ts* **and** *k*) (*auto simp*: *update-desc-def*)

lemma *field-norm-blah*:
⟦ ∀ *u v*. *f* (*g u v*) = *u*; *fd-cons-access-update d n* ⟧ ⟹
field-norm n algn (*update-desc f g d*) = *field-norm n algn d*
by (*auto simp*: *update-desc-def field-norm-def fd-cons-access-update-def*)

lemma *map-td-ext′*:
wf-fd t ∧ (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟶ *map-td f h t* = *map-td g h t*
wf-fd-struct st ∧ (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟶ *map-td-struct f h st* = *map-td-struct g h st*
wf-fd-list ts ∧ (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟶ *map-td-list f h ts* = *map-td-list g h ts*
wf-fd-tuple x ∧ (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟶ *map-td-tuple f h x* = *map-td-tuple g h x*
by (*induct t* **and** *st* **and** *ts* **and** *x*)
(*auto simp*: *fd-cons-struct-def fd-cons-access-update-def fd-cons-desc-def*)

lemma *map-td-extI*:
⟦ *wf-fd t*; (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟧
⟹ *map-td f h t* = *map-td g h t*
by (*simp add*: *map-td-ext′*)

lemma *comp-unit*: (λ-. ()) ∘ *f* = (λ-. ())

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

lemma *export-tag-adjust-ti2*:
⟦ ∀ *u v*. *f* (*g u v*) = *u*; *wf-lf* (*lf-set t* []); *wf-desc t* ⟧ ⟹
export-uinfo (*adjust-ti t f g*) = (*export-uinfo t*)
unfolding *export-uinfo-def adjust-ti-def map-td-map comp-unit*
by (*fastforce simp*: *field-norm-blah intro*: *wf-fdp-fdD map-td-extI elim*: *wf-lf-fdp*)

lemma *field-names-list*:
field-names-list (*xs*@*ys*) *t* = *field-names-list xs t* @ *field-names-list ys t*
by (*induct xs*) *auto*

lemma *field-names-extend-ti*:
typ-name t ≠ *typ-name ti* ⟹
field-names (*extend-ti ti xi algn fn d*) *t* = *field-names ti t* @ (*map* (λ*fs*. *fn*#*fs*)
(*field-names xi t*))
apply (*cases ti*)
subgoal for *x1 typ-struct xs*
by (*cases typ-struct*; *fastforce simp*: *field-names-list*)
done

lemma *field-names-ti-pad-combine*:
⟦ *typ-name t* ≠ *typ-name ti*; *hd* (*typ-name t*) ≠ *CHR* ''!'' ⟧ ⟹
field-names (*ti-pad-combine n ti*) *t* = *field-names ti t*
by (*clarsimp simp*: *ti-pad-combine-def Let-def field-names-extend-ti export-uinfo-def*
size-map-td)

lemma *export-uinfo-map-align-commute*: *export-uinfo* (*map-align f t*) = *map-align*
f (*export-uinfo t*)
by (*cases t*) *auto*

lemma *field-names-map-align*: *typ-name t* ≠ *typ-name ti* ⟹ *field-names* (*map-align*
f ti) *t* = *field-names ti t*
apply (*cases ti*)
apply *auto*
done

lemma *typ-name-map-align* [*simp*]: *typ-name* (*map-align f t*) = *typ-name t*
by (*cases t*) *auto*

lemma *typ-name-ti-pad-combine* [*simp*]:
typ-name (*ti-pad-combine n ti*) = *typ-name ti*
by (*cases ti*) (*simp add*: *ti-pad-combine-def Let-def*)

lemma *field-names-final-pad*:
⟦ *typ-name t* ≠ *typ-name ti*; *hd* (*typ-name t*) ≠ *CHR* ''!'' ⟧ ⟹
field-names (*final-pad a ti*) *t* = *field-names ti t*
by (*clarsimp simp*: *final-pad-def Let-def field-names-ti-pad-combine field-names-map-align*)

lemma *field-names-adjust-ti*:
assumes *fg-cons f g*
shows
wf-fd ti $\longrightarrow$
field-names (*adjust-ti* (*ti*::'*a xtyp-info*) *f g*) *t* = *field-names ti t*
wf-fd-struct st $\longrightarrow$
field-names-struct ((*map-td-struct* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*)
(*st*::'*a xtyp-info-struct*))) *t* =
field-names-struct st t
wf-fd-list ts $\longrightarrow$
field-names-list (*map-td-list* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*)
(*ts*::'*a xtyp-info-tuple list*)) *t* =
field-names-list ts t
wf-fd-tuple x $\longrightarrow$
field-names-tuple (*map-td-tuple* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*)
(*x*::'*a xtyp-info-tuple*)) *t* =
field-names-tuple x t **using** *assms*
by (*induct ti* **and** *st* **and** *ts* **and** *x*) (*auto simp*: *adjust-ti-def export-tag-adjust-ti*)

lemma *field-names-ti-typ-combine*:
⟦ *typ-name t* $\neq$ *typ-name ti*; *fg-cons f g* ⟧ $\Longrightarrow$
field-names (*ti-typ-combine* (*t-b*::'*b*::*mem-type itself*) *f g algn fn ti*) *t* =
field-names ti t @ *map* ((#) *fn*) (*field-names* (*typ-info-t TYPE*('*b*)) *t*)
by (*clarsimp simp*: *ti-typ-combine-def Let-def field-names-adjust-ti field-names-extend-ti*
export-uinfo-def size-map-td)

lemma *size-empty-typ-info* [*simp*]:
size (*empty-typ-info algn tn*) = *2*
by (*simp add*: *empty-typ-info-def*)

lemma *list-size-char*:
size-list (λ*c*. *0*) *xs* = *length xs*
by (*induct xs*) *auto*

lemma *size-ti-extend-ti* [*simp*]:
aggregate ti $\Longrightarrow$ *size* (*extend-ti ti t algn fn d*) = *size ti* + *size t* + *2*
apply (*cases ti*)
subgoal for *x1 typ-struct xs*
by (*cases typ-struct*, *auto simp*: *list-size-char*)
done

lemma *typ-name-empty-typ-info* [*simp*]:
typ-name (*empty-typ-info algn tn*) = *tn*
by (*clarsimp simp*: *empty-typ-info-def*)

lemma *typ-name-extend-ti* [*simp*]:
typ-name (*extend-ti ti t algn fn d*) = *typ-name ti*

by (*cases ti*, *simp*)

lemma *typ-name-ti-typ-combine* [*simp*]:
typ-name (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *f g algn fn ti*) = *typ-name ti*
by (*clarsimp simp*: *ti-typ-combine-def Let-def*)

lemma *typ-name-ti-typ-pad-combine* [*simp*]:
typ-name (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *f g algn fn ti*) = *typ-name ti*
by (*clarsimp simp*: *ti-typ-pad-combine-def Let-def*)

lemma *typ-name-ti-final-pad* [*simp*]:
typ-name (*final-pad algn ti*) = *typ-name ti*
by (*clarsimp simp*: *final-pad-def Let-def*)

lemma *typ-name-map-td* [*simp*]:
typ-name (*map-td f g td*) = *typ-name td*
by (*cases td*, *simp*)

lemma *field-names-ti-typ-pad-combine*:
⟦ *typ-name t* ≠ *typ-name ti*; *fg-cons f g*; *aggregate ti*; *hd* (*typ-name t*) ≠ *CHR* ''!'' ⟧ ⟹
field-names (*ti-typ-pad-combine* (*t-b*::'*b*::*mem-type itself*) *f g algn fn ti*) *t* =
field-names ti t @ *map* ((#) *fn*) (*field-names* (*typ-info-t TYPE*('*b*)) *t*)
by (*auto simp*: *ti-typ-pad-combine-def Let-def field-names-ti-typ-combine field-names-ti-pad-combine*)

lemma *field-names-empty-typ-info*:
typ-name t ≠ *tn* ⟹ *field-names* (*empty-typ-info algn tn*) *t* = []
by(*clarsimp simp*: *empty-typ-info-def*)

lemma *sep-heap-update-global-super-fl'*:
⟦ (*p* ↦$_g$ *u* ∧* *R*) (*lift-state* (*h*,*d*));
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = (*typ-uinfo-t TYPE*('*a*));
w = *update-ti-t t* (*to-bytes-p v*) *u* ⟧ ⟹
((*p* ↦$_g$ *w*) ∧* *R*)
(*lift-state* (*heap-update* (*Ptr* &(*p*→*f*)) (*v*::'*a*::*mem-type*) *h*,*d*))
by (*auto dest*: *sep-heap-update-global-super-fl*)

lemma *sep-heap-update-global-super-fl'-inv*:
⟦ (*p* ↦^i_g *u* ∧* *R*) (*lift-state* (*h*,*d*));
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = (*typ-uinfo-t TYPE*('*a*));
w = *update-ti-t t* (*to-bytes-p v*) *u*⟧ ⟹
((*p* ↦^i_g *w*) ∧* *R*) (*lift-state* (*heap-update* (*Ptr* &(*p*→*f*)) (*v*::'*a*::*mem-type*) *h*,*d*))
unfolding *sep-map-inv-def*
by (*simp only*:*sep-conj-assoc*) (*erule* (2) *sep-heap-update-global-super-fl*)

lemma *sep-map'-field-map'*:

⟦ ((p::'b::mem-type ptr) ↪$_g$ v) s; field-lookup (typ-info-t TYPE('b)) f 0
= Some (d,n); export-uinfo d = typ-uinfo-t TYPE('a);
guard-mono g h ⟧ ⟹
((Ptr (&(p→f))::'a::mem-type ptr) ↪$_h$ from-bytes (access-ti$_0$ d v)) s
by (subst sep-map'-unfold-exc, subst (asm) sep-map'-def)
(fastforce simp: sep-map'-def elim: sep-conj-impl sep-map-field-map')

lemma sep-map'-field-map:
⟦ ((p::'b::mem-type ptr) ↪$_g$ v) s; field-lookup (typ-info-t TYPE('b)) f 0
= Some (d,n); export-uinfo d = typ-uinfo-t TYPE('a);
guard-mono g h; w=from-bytes (access-ti$_0$ d v) ⟧ ⟹
((Ptr (&(p→f))::'a::mem-type ptr) ↪$_h$ w) s
by (simp add: sep-map'-field-map')

lemma inter-sub:
⟦ Y ⊆ X; Y = Z ⟧ ⟹ X ∩ Y = Z
by fast

lemma sep-map'-field-map-inv:
⟦ ((p::'b::mem-type ptr) ↪^i_g v) s; field-lookup (typ-info-t TYPE('b)) f 0 = Some
(d,n);
export-uinfo d = typ-uinfo-t TYPE('a); guard-mono g h; w=from-bytes (access-ti$_0$
d v) ⟧ ⟹
((Ptr &(p→f)::'a::mem-type ptr) ↪^i_h w) s
apply(unfold sep-map'-inv-def)
apply(drule sep-conjD, clarsimp simp: sep-conj-ac)
subgoal for s_0 s_1
apply(subst sep-map'-unfold-exc)
apply(subst sep-conj-assoc [symmetric])
apply(rule sep-conjI [**where** s_0=(s_1 ++ s_0) |` {(x,y) | x y. x ∈ {&(p→f)..+size-td
d}} **and**
s_1=(s_1 ++ s_0) |` (dom (s_1 ++ s_0) − {(x,y) | x y. x ∈ {&(p→f)..+size-td
d}})
])
apply(subst sep-conj-com)
apply(rule sep-conjI [**where** s_0=(s_1 ++ s_0) |` s-footprint ((Ptr &(p→f))::'a::mem-type
ptr) **and**
s_1=(s_1 ++ s_0) |` ({(x, y) |x y. x ∈ {&(p→f)..+size-td d}} −
s-footprint ((Ptr &(p→f))::'a::mem-type ptr))])
apply(drule (3) sep-map'-field-map')
apply(clarsimp simp: sep-conj-ac sep-map'-def sep-conj-def)
subgoal for s_0' s_1'
apply(rule exI [**where** x=s_0' |` s-footprint ((Ptr &(p→f))::'a::mem-type ptr)])
apply(rule exI [**where** x=s_1'])
apply(clarsimp simp: sep-conj-ac)
apply(rule conjI)
apply(fastforce simp: map-disj-def sep-conj-ac)
apply(subst map-add-com[**where** h_0=a ++ b **for** a b])
apply(fastforce simp: map-disj-def sep-conj-ac)

apply(*subst map-add-assoc*)
apply(*simp add*: *map-add-restrict*)
apply(*frule sep-map-dom-exc*)
apply(*rotate-tac −1*)
apply(*drule sym*)
apply(*thin-tac* $s = x$ **for** x)
apply(*fastforce simp*: *restrict-map-disj-dom-empty map-ac-simps sep-conj-ac*)
done
apply(*clarsimp simp*: *inv-footprint-def sep-conj-ac*)
apply(*rule inter-sub*)
apply(*clarsimp simp*: *sep-conj-ac*)
apply(*frule-tac p=p* **in** *field-tag-sub*)
apply(*drule* (*1*) *subsetD*)
apply(*clarsimp simp*: *sep-conj-ac*)
apply(*clarsimp simp*: *sep-conj-ac sep-map′-def sep-conj-def*)
apply(*drule sep-map-dom-exc*)
subgoal for x y $s_0{}'$ $s_1{}'$
apply(*subgoal-tac* $s_1{}'$ $(x,y) \neq$ *None*)
apply(*clarsimp simp*: *sep-conj-ac*)
apply(*subst map-add-comm*)
apply(*fastforce simp*: *map-disj-def sep-conj-ac*)
apply *simp*
apply(*clarsimp simp*: *sep-conj-ac*, *fast*)
done
apply(*fastforce dest*: *export-size-of*)
apply(*fastforce simp*: *map-disj-def*)
apply *clarsimp*
apply(*subst subset-map-restrict-sub-add*; *simp?*)
apply(*fastforce intro*!: *intvlI dest*: *export-size-of*
simp: *size-of-def s-footprint-def s-footprint-untyped-def*)
apply *simp*
apply(*fastforce simp*: *map-disj-def*)
apply (*metis* (*lifting*) *map-add-com map-add-restrict-comp-right-dom map-le-iff-map-add-commute*
restrict-map-sub-add restrict-map-sub-disj)
done
done

lemma *guard-mono-True* [*simp*]:
guard-mono f (*λx*. *True*)
by (*simp add*: *guard-mono-def*)

lemma *access-ti*$_0$*-to-bytes* [*simp*]:
access-ti$_0$ (*typ-info-t TYPE*(*′a*::*c-type*)) = (*to-bytes-p*::*′a* ⇒ *byte list*)
by (*auto simp*: *to-bytes-p-def to-bytes-def access-ti*$_0$*-def size-of-def*)

lemma *update-ti-s-from-bytes*:
length bs = *size-of TYPE*(*′a*) ⟹
update-ti-t (*typ-info-t TYPE*(*′a*::*mem-type*)) *bs x* = *from-bytes bs*
by (*simp add*: *from-bytes-def upd*)

lemma *access-ti$_0$-update-ti* [*simp*]:
access-ti$_0$ (*adjust-ti ti f g*) = *access-ti$_0$ ti* ∘ *f*
by (*auto simp*: *access-ti$_0$-def*)

lemma *update-ti-s-adjust-ti*:
⟦ *length bs* = *size-td ti*; *fg-cons f g* ⟧ ⟹
update-ti-t (*adjust-ti ti f g*) *bs v* = *g* (*update-ti-t ti bs* (*f v*)) *v*
by (*rule update-ti-t-adjust-ti*)

lemma *update-ti-s-adjust-ti-to-bytes-p* [*simp*]:
fg-cons f g ⟹
update-ti-t (*adjust-ti* (*typ-info-t TYPE*('*a*)) *f g*) (*to-bytes-p* (*v*::'*a*::*mem-type*)) *w*
= *g v w*
apply(*simp add*: *update-ti-t-adjust-ti to-bytes-p-def to-bytes-def*)
apply(*subst upd-rf*; *simp add*: *size-of-def fd-cons-length*)
apply(*subst fd-cons-update-access*; *simp*)
done

definition
pad-typ-name ≡ ''!*pad-typ*''

primrec
td-names :: ('*a*,'*b*) *typ-desc* ⇒ (*char list*) *set* **and**
td-names-struct :: ('*a*,'*b*) *typ-struct* ⇒ (*char list*) *set* **and**
td-names-list :: ('*a*,'*b*) *typ-tuple list* ⇒ (*char list*) *set* **and**
td-names-tuple :: ('*a*,'*b*) *typ-tuple* ⇒ (*char list*) *set*
where
tnm0: *td-names* (*TypDesc algn st nm*) = ({*nm*} ∪ *td-names-struct st*) − {*pad-typ-name*}

| *tnm1*: *td-names-struct* (*TypScalar n algn d*) = {}
| *tnm2*: *td-names-struct* (*TypAggregate xs*) = *td-names-list xs*

| *tnm3*: *td-names-list* [] = {}
| *tnm4*: *td-names-list* (*x*#*xs*) = *td-names-tuple x* ∪ *td-names-list xs*

| *tnm5*: *td-names-tuple* (*DTuple t nm d*) = *td-names t*

lemma *td-set-td-names*:
⋀(*tp* :: ('*a*,'*b*) *typ-desc*) *n m*. ⟦(*tp*, *n*) ∈ *td-set tp*' *m*; *typ-name tp* ≠ *pad-typ-name*
⟧ ⟹
typ-name tp ∈ *td-names tp*' **and**
⋀(*tp* :: ('*a*,'*b*) *typ-desc*) *n m*. ⟦(*tp*, *n*) ∈ *td-set-struct tps m*; *typ-name tp* ≠
pad-typ-name ⟧ ⟹
typ-name tp ∈ *td-names-struct tps* **and**
⋀(*tp* :: ('*a*,'*b*) *typ-desc*) *n m*. ⟦(*tp*, *n*) ∈ *td-set-list tpl m*; *typ-name tp* ≠ *pad-typ-name*
⟧ ⟹

typ-name tp ∈ *td-names-list tpl* **and**
⋀(*tp* :: ('*a*,'*b*) *typ-desc*) *n m*. ⟦(*tp*, *n*) ∈ *td-set-tuple tpr m*; *typ-name tp* ≠ *pad-typ-name* ⟧⟹
typ-name tp ∈ *td-names-tuple tpr*
by (*induct tp'* **and** *tps* **and** *tpl* **and** *tpr*) *auto*

lemma *td-names-map-td* [*simp*]:
td-names (*map-td f g tp*) = *td-names tp*
td-names-struct (*map-td-struct f g tps*) = *td-names-struct tps*
td-names-list (*map-td-list f g tpl*) = *td-names-list tpl*
td-names-tuple (*map-td-tuple f g tpr*) = *td-names-tuple tpr*
by (*induct tp* **and** *tps* **and** *tpl* **and** *tpr*) *simp-all*

lemma *td-names-list-append* [*simp*]:
td-names-list (*a* @ *b*) = *td-names-list a* ∪ *td-names-list b*
by (*induct a*; *simp add*: *Un-assoc*)

lemma *pad-typ-name-td-names*:
A − {*pad-typ-name*} ∪ *td-names tp* = (*A* ∪ *td-names tp*) − {*pad-typ-name*}
by (*cases tp*) *fastforce*

lemma *td-names-extend-ti* [*simp*]:
shows *td-names* (*extend-ti tp tp' algn ls d*) = *td-names tp* ∪ *td-names tp'* **and**
td-names-struct (*extend-ti-struct tps tp' ls d*) = *td-names-struct tps* ∪ *td-names tp'*
by (*induct tp* **and** *tps*) (*simp-all add*: *pad-typ-name-td-names*)

lemma *td-names-pad-combine* [*simp*]:
td-names (*ti-pad-combine m td*) = *td-names td*
unfolding *ti-pad-combine-def*
by (*simp add*: *Let-def pad-typ-name-def*)

lemma *td-names-map-align* [*simp*]:
td-names (*map-align f td*) = *td-names td*
by (*cases td*) *simp*

lemma *td-names-final-pad* [*simp*]:
td-names (*final-pad algn td*) = *td-names td*
unfolding *final-pad-def*
by (*simp add*: *Let-def*)

lemma *td-names-adjust-ti* [*simp*]:
td-names (*adjust-ti td f u*) = *td-names td*
unfolding *adjust-ti-def*
by (*simp add*: *Let-def*)

lemma *td-names-typ-combine* [*simp*]:
fixes *its* :: ('*a* :: *c-type*) *itself*

shows *td-names* (*ti-typ-combine its f u algn fn td*) = (*td-names* (*typ-info-t TYPE*('*a*)) ∪ *td-names td*)
unfolding *ti-typ-combine-def*
by (*simp add*: *Let-def Un-ac*)

lemma *td-names-typ-pad-combine* [*simp*]:
fixes *its* :: ('*a* :: *c-type*) *itself*
shows *td-names* (*ti-typ-pad-combine its f u algn nm td*) = *td-names* (*typ-info-t TYPE*('*a*)) ∪ *td-names td*
unfolding *ti-typ-pad-combine-def*
by (*simp add*: *Let-def*)

lemma *td-names-empty-typ-info* [*simp*]:
shows *td-names* (*empty-typ-info algn nm*) = {*nm*} − {*pad-typ-name*}
unfolding *empty-typ-info-def*
by (*simp add*: *Let-def*)

lemma *td-names-ptr* [*simp*]:
td-names (*typ-info-t TYPE*(('*a* :: *c-type*) *ptr*)) = {*typ-name-itself TYPE*('*a*) @ ''+*ptr*''}
by (*simp add*: *pad-typ-name-def*)

lemma *td-names-word8* [*simp*]:
fixes *x* :: *byte itself*
shows *td-names* (*typ-info-t x*) = {''*word00010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-word8'* [*simp*]:
shows *td-names* (*typ-info-t TYPE*(*8 word*)) = {''*word00010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-signed-word8* [*simp*]:
shows *td-names* (*typ-info-t TYPE*(*8 signed word*)) = {''*word00010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-word16* [*simp*]:
shows *td-names* (*typ-info-t TYPE*(*16 word*)) = {''*word000010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-signed-word16* [*simp*]:
shows *td-names* (*typ-info-t TYPE*(*16 signed word*)) = {''*word000010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-word32* [*simp*]:
td-names (*typ-info-t TYPE*(*32 word*)) = {''*word0000010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-signed-word32* [*simp*]:

td-names (*typ-info-t TYPE*(*32 signed word*)) = {''*word0000010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-word64* [*simp*]:
td-names (*typ-info-t TYPE*(*64 word*)) = {''*word00000010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-signed-word64* [*simp*]:
td-names (*typ-info-t TYPE*(*64 signed word*)) = {''*word00000010*''}
by (*simp add*: *pad-typ-name-def nat-to-bin-string.simps*)

lemma *td-names-export-uinfo* [*simp*]:
td-names (*export-uinfo td*) = *td-names td*
unfolding *export-uinfo-def*
by *simp*

lemma *typ-name-export-uinfo* [*simp*]:
typ-name (*export-uinfo td*) = *typ-name td*
unfolding *export-uinfo-def*
by *simp*

lemma *replicate-Suc-append*:
replicate (*Suc n*) *x* = *replicate n x* @ [*x*]
by (*induct n*; *simp*)

lemma *list-eq-subset*:
xs = *ys* ⟹ *set ys* ⊆ *set xs* **by** *simp*

lemma *td-names-array-tag-n*:
td-names ((*array-tag-n n*) :: (('*a*::*c-type*,'*b*::*finite*) *array xtyp-info*)) =
{*typ-name* (*typ-info-t TYPE*('*a*)) @ ''*-array-*'' @ *nat-to-bin-string* (*card* (*UNIV* :: '*b set*))} ∪
(*if n* = *0 then* {} *else td-names* (*typ-info-t TYPE*('*a*)))
apply (*induct n*; *simp add*: *array-tag-n.simps pad-typ-name-def*)
apply (*subst Diff-triv*; *clarsimp simp*: *typ-uinfo-t-def*)
apply (*fastforce dest*: *list-eq-subset*)
done

lemma *td-names-array* [*simp*]:
td-names (*typ-info-t TYPE*(('*a* :: *c-type*)['*b* :: *finite*])) =
{*typ-name* (*typ-info-t TYPE*('*a*)) @ ''*-array-*'' @ *nat-to-bin-string* (*card* (*UNIV* :: '*b set*))} ∪
td-names (*typ-info-t TYPE*('*a*))
by (*simp add*: *typ-info-array array-tag-def td-names-array-tag-n*)

lemma *tag-disj-via-td-name*:
assumes *ta*: *typ-name* (*typ-info-t TYPE*('*a* :: *c-type*)) ≠ *pad-typ-name*
and *tb*: *typ-name* (*typ-info-t TYPE*('*b* :: *c-type*)) ≠ *pad-typ-name*
and *tina*: *typ-name* (*typ-info-t TYPE*('*a* :: *c-type*)) ∉ *td-names* (*typ-info-t*

$TYPE('b :: c\text{-}type)$)
and *tinb*: *typ-name* (*typ-info-t* $TYPE('b :: c\text{-}type)$) $\notin$ *td-names* (*typ-info-t* $TYPE('a :: c\text{-}type)$)
shows *typ-uinfo-t* $TYPE('a :: c\text{-}type)$ $\perp_t$ *typ-uinfo-t* $TYPE('b :: c\text{-}type)$
by (*auto simp add*: *typ-simps dest*: *td-set-td-names simp*: *ta tb tina tinb*)

lemma *lift-t-hrs-mem-update-fld*:
fixes *val* :: $'b$ :: *mem-type* **and** *ptr* :: $'a$ :: *mem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t* $TYPE('a)$) *f* 0 $\equiv$
Some (*adjust-ti* (*typ-info-t* $TYPE('b)$) *xf* (*xfu* $\circ$ (λx -. *x*)), m')
and *xf-xfu*: *fg-cons xf* (*xfu* $\circ$ (λx -. *x*))
and *cl*: *lift-t g hp ptr* = *Some z*
shows (*lift-t g* (*hrs-mem-update* (*heap-update* (*Ptr* &(*ptr*→*f*)) *val*) *hp*)) =
(*lift-t g hp*)(*ptr* $\mapsto$ *xfu* (λ-. *val*) *z*)
(**is** *?LHS* = *?RHS*)
proof −
let *?ati* = *adjust-ti* (*typ-info-t* $TYPE('b)$) *xf* (*xfu* $\circ$ (λx -. *x*))
have *eui*: *typ-uinfo-t* $TYPE('b)$ = *export-uinfo* (*?ati*) **using** *xf-xfu*
by (*simp add*: *typ-uinfo-t-def export-tag-adjust-ti*)

have *cl'*: *lift-t g* (*fst hp*, *snd hp*) *ptr* = *Some z* **using** *cl* **by** *simp*

have *?LHS* = *super-field-update-t* (*Ptr* &(*ptr*→*f*)) *val* (*lift-t g* (*fst hp*, *snd hp*))
unfolding *hrs-mem-update-def split-def*
proof (*rule lift-t-super-field-update* [*OF h-t-valid-sub*])
from *cl'* **show** *snd hp*, *g* $\models_t$ *ptr* **by** (*rule lift-t-h-t-valid*)

show *fti*: *field-ti* $TYPE('a)$ *f* = *Some ?ati*
by (*simp add*: *field-ti-def fl*)

moreover show *export-uinfo* (*?ati*) = *typ-uinfo-t* $TYPE('b)$
by (*rule eui* [*symmetric*])

ultimately show $TYPE('b) \leq_\tau TYPE('a)$ **by** (*rule field-ti-sub-typ*)
qed

also
have ... = (*lift-t g hp*)(*ptr* $\mapsto$ *update-ti-t* (*adjust-ti* (*typ-info-t* $TYPE('b)$) *xf* (*xfu* $\circ$ (λx -. *x*)))
(*to-bytes-p val*) *z*)
by (*simp add*: *cl eui fl super-field-update-lookup*)

also have ... = *?RHS* **using** *xf-xfu*
by (*simp add*: *update-ti-t-adjust-ti update-ti-s-from-bytes*)

finally show *?thesis* .
qed

declare *pad-typ-name-def* [*simp*]

lemma *typ-name-array-tag-n*:
typ-name (*array-tag-n n* :: (*'a* ::*c-type* [*'b* :: *finite*]) *xtyp-info*) =
typ-name (*typ-info-t TYPE*(*'a*)) @ *''-array-''* @ *nat-to-bin-string* (*card* (*UNIV* :: *'b set*))
by (*induct n*; *clarsimp simp*: *array-tag-n.simps typ-uinfo-t-def*)

lemma *typ-name-array* [*simp*]:
typ-name (*typ-info-t TYPE*(*'a*::*c-type*[*'b* :: *finite*])) =
typ-name (*typ-info-t TYPE*(*'a*)) @ *''-array-''* @ *nat-to-bin-string* (*card* (*UNIV* :: *'b set*))
by (*simp add*: *typ-info-array array-tag-def typ-name-array-tag-n*)

definition
size-align-td :: (*'a*,*'b*) *typ-desc* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
size-align-td t s a ≡ *size-td t* = *s* ∧ *align-td t* = *a*

lemma *size-align-td-size-td*:
size-align-td t s a ⟹ *size-td t* = *s*
unfolding *size-align-td-def* **by** *simp*

lemma *size-align-td-align-td*:
size-align-td t s a ⟹ *align-td t* = *a*
unfolding *size-align-td-def* **by** *simp*

lemma *size-align-td-empty-typ-infoI*:
size-align-td (*empty-typ-info algn l*) *0 algn*
unfolding *size-align-td-def* **by** *simp*

lemma *size-align-td-ti-typ-pad-combineI*:
fixes *t* :: (*'a* :: *c-type*) *itself*
shows ⟦ *size-align-td t' s' a'*; *aggregate t'*
; *s'* + *size-td* (*typ-info-t TYPE*(*'a*)) + *padup* (*2* ^ (*max algn* (*align-td* (*typ-info-t TYPE*(*'a*))))) *s'* = *s*
; *max a'* (*max algn* (*align-td* (*typ-info-t TYPE*(*'a*)))) = *a*
⟧
⟹ *size-align-td* (*ti-typ-pad-combine t fa fu algn nm t'*) *s a*
unfolding *size-align-td-def*
by (*simp add*: *size-td-lt-ti-typ-pad-combine*)

lemma *size-align-td-ti-final-padI*:
fixes *t* :: (*'a* :: *c-type*) *xtyp-info*
shows ⟦ *size-align-td t' s' a'*; *aggregate t'*; *s'* + *padup* (*2* ^ *max algn a'*) *s'* = *s*; *max algn a'* = *a* ⟧
⟹ *size-align-td* (*final-pad algn t'*) *s a*
unfolding *size-align-td-def*
by (*simp add*: *size-td-lt-final-pad align-of-final-pad*)

lemma *field-lookup-empty-typ-infoI*:
field-lookup (*empty-typ-info algn l*) [*fld*] *m* = *None*
unfolding *empty-typ-info-def*
by *simp*

lemma *field-lookup-ti-typ-pad-combine-matchI*:
⟦ *n # nm* ∉ *set* (*CompoundCTypes.field-names-list t'*); *aggregate t'*; *size-align-td t' s a*
; *padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*b*))) *s* + *s* = *m'*; *n* ≠ *CHR* ''!'' ⟧
⟹ *field-lookup* (*ti-typ-pad-combine* (*t* :: '*b itself*) *fa fu algn* (*n#nm*) *t'*) [(*n#nm*)] *0* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b* :: *c-type*)) *fa fu*, *m'*)
by (*simp add*: *fl-simps size-align-td-def*)

lemma *field-lookup-ti-typ-pad-combine-nomatchI*:
⟦ *aggregate t'*; *f* ≠ *CHR* ''!''; *f # fld* ≠ *nm* ; *field-lookup t'* [*f # fld*] *0* = *R* ⟧
⟹ *field-lookup* (*ti-typ-pad-combine t fa fu algn nm t'*) [*f # fld*] *0* = *R*
by (*simp add*: *fl-simps*)

lemma *field-lookup-final-padI*:
⟦ *aggregate t'*; *f* ≠ *CHR* ''!''; *field-lookup t'* [*f # fld*] *0* = *R* ⟧ ⟹ *field-lookup* (*final-pad algn t'*) [*f # fld*] *0* = *R*
by (*simp add*: *fl-simps*)

lemmas *field-lookup-ti-intros* =
field-lookup-ti-typ-pad-combine-matchI field-lookup-ti-typ-pad-combine-nomatchI
field-lookup-final-padI field-lookup-empty-typ-infoI

lemma *notin-field-names-list-empty-typ-info*:
fld ∉ *set* (*CompoundCTypes.field-names-list* (*empty-typ-info algn l*))
by *simp*

lemma *notin-field-names-list-ti-typ-pad-combine*:
⟦ *f#fld* ∉ *set* (*CompoundCTypes.field-names-list t'*); *f#fld* ≠ *nm*; *f* ≠ *CHR* ''!'' ⟧
⟹ *f#fld* ∉ *set* (*CompoundCTypes.field-names-list* (*ti-typ-pad-combine t fa fu algn nm t'*))
by *simp*

lemma *fold-typ-uinfo-t*: *export-uinfo* (*typ-info-t TYPE*('*a*::*c-type*)) = *typ-uinfo-t TYPE*('*a*)
by (*simp add*: *typ-uinfo-t-def*)

end

theory *ArrayAssertion*

imports
ArchArraysMemInstance
StructSupport

begin

lemma *array-tag-n-eq*:
(*array-tag-n n* :: (*'a* :: *c-type*[*'b* :: *finite*]) *xtyp-info*) =
TypDesc (*align-td* (*typ-uinfo-t TYPE*(*'a*))) (*TypAggregate*
(*map* (λ*n*. *DTuple* (*adjust-ti* (*typ-info-t TYPE*(*'a*)) (λ*x*. *index x n*)
(λ*x f*. *Arrays.update f n x*)) (*replicate n CHR ''1''*)
(|*field-access* = *xto-bytes* ∘ (λ*x*. *index x n*),
field-update = (λ*x f*. *Arrays.update f n x*) ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*(*'a*::*c-type*)|)) [*0*..<*n*]))
(*typ-name* (*typ-uinfo-t TYPE*(*'a*)) @ *''-array-''* @ *nat-to-bin-string* (*card* (*UNIV* :: *'b* :: *finite set*)))
apply (*induct n*)
apply (*simp add*: *typ-info-array array-tag-def eval-nat-numeral*
array-tag-n.simps empty-typ-info-def align-of-def align-td-uinfo)
apply (*simp add*: *typ-info-array array-tag-def eval-nat-numeral array-tag-n.simps empty-typ-info-def*)
apply (*simp add*: *ti-typ-combine-def Let-def comp-def align-td-uinfo*)
done

lemma *typ-info-array'*:
typ-info-t TYPE (*'a* :: *c-type*[*'b* :: *finite*]) =
TypDesc (*align-td* (*typ-uinfo-t TYPE*(*'a*))) (*TypAggregate*
(*map* (λ*n*. *DTuple* (*adjust-ti* (*typ-info-t TYPE*(*'a*)) (λ*x*. *index x n*)
(λ*x f*. *Arrays.update f n x*)) (*replicate n CHR ''1''*)
(|*field-access* = *xto-bytes* ∘ (λ*x*. *index x n*),
field-update = (λ*x f*. *Arrays.update f n x*) ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*(*'a*::*c-type*)|)) [*0*..<(*card* (*UNIV* :: *'b* :: *finite set*))]))
(*typ-name* (*typ-uinfo-t TYPE*(*'a*)) @ *''-array-''* @ *nat-to-bin-string* (*card* (*UNIV* :: *'b* :: *finite set*)))
by (*simp add*: *typ-info-array array-tag-def array-tag-n-eq*)

definition
uinfo-array-tag-n-m (*v* :: *'a itself*) *n m* = *TypDesc*
(*align-td* (*typ-uinfo-t TYPE*(*'a*)))
(*TypAggregate* (*map* (λ*i*. *DTuple* (*typ-uinfo-t TYPE*(*'a*)) (*replicate i CHR ''1''*) ()) [*0* ..< *n*]))
(*typ-name* (*typ-uinfo-t TYPE*(*'a* :: *c-type*)) @ *''-array-''* @ *nat-to-bin-string m*)

lemma *map-td-list-map*:
map-td-list f g = *map* (*map-td-tuple f g*)

apply (*rule ext*)
subgoal for *x* **by** (*induct x*) *simp-all*
done

lemma *uinfo-array-tag-n-m-eq*:
n ≤ *CARD*('*b*)
⟹ *export-uinfo* (*array-tag-n n* :: ((*'a* :: *wf-type*)['*b* :: *finite*]) *xtyp-info*)
= *uinfo-array-tag-n-m TYPE* (*'a*) *n* (*CARD*('*b*))
apply (*clarsimp simp*: *uinfo-array-tag-n-m-def array-tag-n-eq map-td-list-map o-def adjust-ti-def map-td-map typ-uinfo-t-def export-uinfo-def*)
apply (*fastforce intro*: *map-td-extI simp*: *field-norm-blah*)
done

lemma *typ-uinfo-array-tag-n-m-eq*:
typ-uinfo-t TYPE ((*'a* :: *wf-type*)['*b* :: *finite*])
= *uinfo-array-tag-n-m TYPE* (*'a*) (*CARD*('*b*)) (*CARD*('*b*))
by (*simp add*: *typ-uinfo-t-def typ-info-array array-tag-def uinfo-array-tag-n-m-eq*)

Alternative to *h-t-valid* for arrays. This allows reasoning about arrays of variable width.

definition *h-t-array-valid* :: *heap-typ-desc* ⇒ (*'a* :: *c-type*) *ptr* ⇒ *nat* ⇒ *bool* **where**
h-t-array-valid htd ptr n = *valid-footprint htd* (*ptr-val ptr*) (*uinfo-array-tag-n-m TYPE* (*'a*) *n n*)

Assertion that pointer p is within an array that continues for at least n more elements.

definition
array-assertion (*p* :: (*'a* :: *c-type*) *ptr*) *n htd*
= (∃ *q i j*. *h-t-array-valid htd q j*
∧ *p* = *CTypesDefs.ptr-add q* (*int i*) ∧ *i* < *j* ∧ *i* + *n* ≤ *j*)

lemma *array-assertion-shrink-right*:
array-assertion p n htd ⟹ *n'* ≤ *n* ⟹ *array-assertion p n' htd*
by (*fastforce simp*: *array-assertion-def*)

lemma *array-assertion-shrink-leftD*:
array-assertion p n htd ⟹ *j* < *n* ⟹ *array-assertion* (*CTypesDefs.ptr-add p* (*int j*)) (*n* − *j*) *htd*
apply (*clarsimp simp*: *array-assertion-def*)
subgoal for *q i*
apply (*rule exI*, *rule exI*[**where** *x*=*i* + *j*], *rule exI*, *erule conjI*)
apply (*simp add*: *CTypesDefs.ptr-add-def field-simps*)
done
done

lemma *array-assertion-shrink-leftI*:
array-assertion (*CTypesDefs.ptr-add p* (− (*int j*))) (*n* + *j*) *htd*

$\Longrightarrow n \neq 0 \Longrightarrow$ *array-assertion p n htd*
apply (*drule-tac j=j* **in** *array-assertion-shrink-leftD*, *simp*)
apply (*simp add*: *CTypesDefs.ptr-add-def*)
done

lemma *h-t-array-valid*:
h-t-valid htd gd (*p* :: ((*'a* :: *wf-type*)[*'b* :: *finite*]) *ptr*)
$\Longrightarrow$ *h-t-array-valid htd* (*ptr-coerce p* :: *'a ptr*) (*CARD*(*'b*))
by (*clarsimp simp*: *h-t-valid-def h-t-array-valid-def typ-uinfo-array-tag-n-m-eq*)

lemma *array-ptr-valid-array-assertionD*:
h-t-valid htd gd (*p* :: ((*'a* :: *wf-type*)[*'b* :: *finite*]) *ptr*)
$\Longrightarrow$ *array-assertion* (*ptr-coerce p* :: *'a ptr*) (*CARD*(*'b*)) *htd*
apply (*clarsimp simp*: *array-assertion-def dest*!: *h-t-array-valid*)
apply (*fastforce intro*: *exI*[**where** *x=0*])
done

lemma *array-ptr-valid-array-assertionI*:
h-t-valid htd gd (*q* :: ((*'a* :: *wf-type*)[*'b* :: *finite*]) *ptr*)
$\Longrightarrow$ *q* = *ptr-coerce p*
$\Longrightarrow$ *n* $\leq$ *CARD*(*'b*)
$\Longrightarrow$ *array-assertion* (*p* :: *'a ptr*) *n htd*
by (*auto dest*: *array-ptr-valid-array-assertionD*
simp: *array-assertion-shrink-right*)

Derived from *array-assertion*, an appropriate assertion for performing a pointer addition, or for dereferencing a pointer addition (the strong case).

In either case, there must be an array connecting the two ends of the pointer transition, with the caveat that the last address can be just out of the array if the pointer is not dereferenced, thus the strong/weak distinction.

If the pointer doesn't actually move, nothing is learned.

definition *ptr-add-assertion* :: (*'a* :: *c-type*) *ptr* $\Rightarrow$ *int* $\Rightarrow$ *bool* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *bool* **where**
ptr-add-assertion ptr offs strong htd $\equiv$
offs = *0* $\vee$
array-assertion (*if offs* < *0 then CTypesDefs.ptr-add ptr offs else ptr*)
(*if offs* < *0 then nat* (− *offs*) *else if strong then Suc* (*nat offs*) *else nat offs*)
htd

lemma *ptr-add-assertion-positive*:
offs $\geq$ *0* $\Longrightarrow$ *ptr-add-assertion ptr offs strong htd*
= (*offs* = *0* $\vee$ *array-assertion ptr* (*case strong of True* $\Rightarrow$ *Suc* (*nat offs*)
| *False* $\Rightarrow$ *nat offs*) *htd*)
by (*simp add*: *ptr-add-assertion-def*)

lemma *ptr-add-assertion-negative*:
offs < *0* $\Longrightarrow$ *ptr-add-assertion ptr offs strong htd*
= *array-assertion* (*CTypesDefs.ptr-add ptr offs*) (*nat* (− *offs*)) *htd*

by (*simp add*: *ptr-add-assertion-def*)

lemma *ptr-add-assertion-uint*[*simp*]:
ptr-add-assertion ptr (*uint offs*) *strong htd*
= (*offs* = *0* ∨ *array-assertion ptr*
(*case strong of True* ⇒ *Suc* (*unat offs*) | *False* ⇒ *unat offs*) *htd*)
by (*simp add*: *ptr-add-assertion-positive uint-0-iff*
split: *bool.split*)

Ignore char and void pointers. The C standard specifies that arithmetic on char and void pointers doesn't create any special checks.

definition *ptr-add-assertion'* :: (*'a* :: *c-type*) *ptr* ⇒ *int* ⇒ *bool* ⇒ *heap-typ-desc* ⇒ *bool* **where**
ptr-add-assertion' ptr offs strong htd =
(*typ-uinfo-t TYPE*(*'a*) = *typ-uinfo-t TYPE*(*word8*)
∨ *typ-uinfo-t TYPE* (*'a*) = *typ-uinfo-t TYPE*(*unit*)
∨ *ptr-add-assertion ptr offs strong htd*)

lemma *td-diff-from-typ-name*:
typ-name td ≠ *typ-name td'* ⟹ *td* ≠ *td'*
by *clarsimp*

lemma *typ-name-void*:
typ-name (*typ-uinfo-t TYPE*(*unit*)) = *''unit''*
by (*simp add*: *typ-uinfo-t-def*)

lemmas *ptr-add-assertion'* = *ptr-add-assertion'-def td-diff-from-typ-name typ-name-void*

Mechanism for retyping a range of memory to a non-constant array size.

definition *ptr-arr-retyps* :: *nat* ⇒ (*'a* :: *c-type*) *ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
ptr-arr-retyps n p ≡
htd-update-list (*ptr-val p*)
(*map* (λ*i*. *list-map* (*typ-slice-t* (*uinfo-array-tag-n-m TYPE*(*'a*) *n n*) *i*))
[*0*..<*n* ∗ *size-of TYPE*(*'a*)])

lemma *ptr-arr-retyps-to-retyp*:
n = *CARD*(*'b* :: *finite*)
⟹ *ptr-arr-retyps n* (*p* :: (*'c* :: *wf-type*) *ptr*) = *ptr-retyp* (*ptr-coerce p* :: (*'c*[*'b*]) *ptr*)
by (*auto simp*: *ptr-arr-retyps-def ptr-retyp-def typ-slices-def typ-uinfo-array-tag-n-m-eq*)

definition
array-ptr-index :: ((*'a* :: *c-type*)[*'b* :: *finite*]) *ptr* ⇒ *bool* ⇒ *nat* ⇒ *'a ptr*
where
array-ptr-index p coerce n = *CTypesDefs.ptr-add* (*ptr-coerce p*)
(*if coerce* ∧ *n* ≥ *CARD* (*'b*) *then 0 else of-nat n*)

lemmas *array-ptr-index-simps*
  = *array-ptr-index-def*[**where** *coerce=False*, *simplified*]
    array-ptr-index-def[**where** *coerce=True*, *simplified*]

lemma *field-lookup-list-Some-again*:
  dt-snd (*xs* ! *i*) = *f*
  ⟹ *i* < *length xs*
  ⟹ *f* ∉ *dt-snd* ‘ *set* ((*take i xs*))
  ⟹ *field-lookup-list xs* [*f*] *n*
    = *Some* (*dt-fst* (*xs* ! *i*), *n* + *sum-list* (*map* (*size-td o dt-fst*) (*take i xs*)))
  apply (*induct xs arbitrary*: *i n*, *simp-all*)
  subgoal for *x1 xs i*
   apply (*cases x1*, *simp*)
   apply (*cases i*, *auto split*: *if-split*)
   done
  done

lemma *field-lookup-array*:
  n < *CARD*('*b*) ⟹ *field-lookup* (*typ-info-t TYPE*((*'a* :: *c-type*)[*'b* :: *finite*]))
   [*replicate n* (*CHR ''1''*)] *i* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*a*))
     (λ*x*. *x*.[*n*]) (λ*x f*. *Arrays.update f n x*), *i* + *n* ∗ *size-of TYPE* ('*a*))
  apply (*simp add*: *typ-info-array array-tag-def array-tag-n-eq*)
  apply (*subst field-lookup-list-Some-again*[**where** *i*=*n*],
   auto simp add: *take-map o-def sum-list-triv size-of-def*)
  done

lemma *field-ti-array*:
  n < *CARD*('*b*) ⟹
   field-ti (*TYPE*((*'a* :: *c-type*)[*'b* :: *finite*])) [*replicate n* (*CHR ''1''*)] =
    Some (*adjust-ti* (*typ-info-t TYPE*('*a*)) (λ*x*. *x*.[*n*]) (λ*x f*. *Arrays.update f n x*))
  by (*simp add*: *field-ti-def field-lookup-array*)

lemma *fg-cons-array* [*simp*]:
  n < *card* (*UNIV* :: *'b* :: *finite set*) ⟹
   fg-cons (λ*x*. *index x n*) (λ*x f*. *Arrays.update* (*f* :: *'a*['*b*]) *n x*)
  unfolding *fg-cons-def* **by** *simp*

lemma *ptr-val-array-field-lvalue-0*:
  fixes *p*:: ((*'a* :: *c-type*)[*'b* :: *finite*]) *ptr*
  shows &(*p*→[[]]) = *ptr-val p*
proof −
  note *fl* = *field-lookup-array* [**where** *n=0* **and** *i=0* **and** *'a='a* **and** *'b='b*, *simplified*]
  note *fl-exp* = *field-lookup-export-uinfo-Some* [*OF fl*]
  show *?thesis*
   unfolding *field-lvalue-def field-offset-def field-offset-untyped-def typ-uinfo-t-def*
   apply (*subst fl-exp*)

apply *simp*
done
qed

end

theory *CProof*
imports
umm-heap/SepFrame
Simpl.Vcg
umm-heap/StructSupport
umm-heap/ArrayAssertion
AutoCorres-Utils
begin

ML-file *General.ML*
ML-file *SourcePos.ML*
ML-file *SourceFile.ML*
ML-file *Region.ML*
ML-file *Binaryset.ML*
ML-file *Feedback.ML*
ML-file *basics.ML*
ML-file *MString.ML*

ML-file *TargetNumbers−sig.ML*
ML-file *./umm-heap/ARM/TargetNumbers-ARM.ML*
ML-file *./umm-heap/ARM64/TargetNumbers-ARM64.ML*
ML-file *./umm-heap/ARM-HYP/TargetNumbers-ARM-HYP.ML*
ML-file *./umm-heap/RISCV64/TargetNumbers-RISCV64.ML*
ML-file *./umm-heap/X64/TargetNumbers-X64.ML*
ML-file *./umm-heap/TargetNumbers.ML*

ML-file *RegionExtras.ML*
ML-file *Absyn−CType.ML*
ML-file *Absyn−Expr.ML*
ML-file *Absyn−StmtDecl.ML*
ML-file *Absyn.ML*
ML-file *Absyn−Serial.ML*
ML-file *../lib/ml−helpers/StringExtras.ML*
ML-file *name-generation.ML*

setup ‹
Hoare.add-foldcongsimps [@{*thm update-update*}, @{*thm o-def*}]
›

syntax
-quote :: *'b* => (*'a* => *'b*) (‹(‹*notation*=‹*mixfix quote*››[.[-].])› [*0*] *1000*)

syntax
-heap :: *'b* ⇒ (*'a* ⇒ *'b*)
-heap-state :: *'a* (‹ζ›)
-heap-stateOld :: (*'a* ⇒ *'b*) ⇒ *'b* (‹(‹*open-block notation*=‹*mixfix heap-state old*›› ¯ζ)› [*100*] *100*)

-derefCur :: (*'a* ⇒ *'b*) ⇒ *'b* (‹(‹*open-block notation*=‹*prefix deref*›› ⋆-)› [*100*] *100*)
-derefOld :: *'a* ⇒ (*'a* ⇒ *'b*) ⇒ *'b* (‹(‹*open-block notation*=‹*prefix deref old*›› ¯⋆-)› [*100*,*100*] *100*)

translations
{|*b*|} => *CONST Collect* (*-quote* (*-heap b*))

definition *sep-app* :: (*heap-state* ⇒ *bool*) ⇒ *heap-state* ⇒ *bool* **where**
sep-app P s ≡ *P s*

definition *hrs-id* :: *heap-raw-state* ⇒ *heap-raw-state* **where**
hrs-id ≡ *id*

declare *hrs-id-def* [*simp add*]

parse-translation ‹
let
fun ac x = *Syntax.const -antiquoteCur* $ *Syntax.const x*
fun aco x y = *Syntax.const y* $ (*Syntax.const globals* $ *x*)
fun hd a = *a NameGeneration.global-heap-var*
fun d a = *Syntax.const hrs-htd* $ *hd a*
fun repl (*Abs* (*s*,*T*,*t*)) = *Abs* (*s*,*T*,*repl t*)
| *repl* (*Const* (*-h-t-valid*,-)$*x*) = *Syntax.const h-t-valid* $ *d ac* $ *Syntax.const c-guard* $ *x*
| *repl* (*Const* (*-derefCur*,-)$*x*) = *Syntax.const the* $
(*Syntax.const lift-t* $ *hd ac* $ *x*)
| *repl* (*Const* (*-derefOld*,-)$*x*$*y*) = *Syntax.const the* $
(*Syntax.const lift-t* $ *hd* (*aco x*) $ *y*)
| *repl* (*Const* (*-heap-state*,-)) = *Syntax.const hrs-id* $ *hd ac*
| *repl* (*Const* (*-heap-stateOld*,-)$*x*) = *Syntax.const hrs-id* $ *hd* (*aco x*)
| *repl* (*x*$*y*) = *repl x* $ *repl y*
| *repl x* = *x*
fun heap-assert-tr [*b*] = *repl b*
| *heap-assert-tr ts* = *raise TERM* (*heap-assert-tr*, *ts*);
in [(*-heap*,*K heap-assert-tr*)] *end*
›

parse-translation ‹
let
  fun ac x = Syntax.const -antiquoteCur $ Syntax.const x
  fun aco x y = Syntax.const y $ (Syntax.const globals $ x)
  fun hd a = Syntax.const lift-state $ (a NameGeneration.global-heap-var)
  fun st2 (Abs (s,T,t)) n = Abs (s,T,st2 t (n+1))
    | st2 (Bound k) n = Bound (if k < n then k else k + 1)
    | st2 (x$y) n = st2 x n $ st2 y n
    | st2 x - = x
  fun st1 (Abs (s,T,t)) n = Abs (s,T,st1 t (n+1))
    | st1 (Bound k) n = Bound (if k < n then k else k + 1)
    | st1 (Const (sep-empty,-)) n = Syntax.const sep-empty $ Bound n
    | st1 (Const (sep-map,-)xy) n = Syntax.const sep-map $
       (st2 x n) $ (st2 y n) $ Bound n
    | st1 (Const (sep-map′,-)xy) n = Syntax.const sep-map′ $
       (st2 x n) $ (st2 y n) $ Bound n
    | st1 (Const (sep-conj,-)xy) n = Syntax.const sep-conj $
       (nst2 x n) $ (nst2 y n) $ Bound n
    | st1 (Const (sep-impl,-)xy) n = Syntax.const sep-impl $
       (nst2 x n) $ (nst2 y n) $ Bound n
    | st1 (x$y) n = st1 x n $ st1 y n
    | st1 x - = x
  and new-heap t = Abs (s,dummyT,st1 t 0)
  and nst2 x n = new-heap (st2 x n);
  fun sep-tr [t] = Syntax.const sep-app $ (∗new-heap ∗) t $ hd ac
    | sep-tr ts = raise TERM (sep-tr, ts);
in [(-sep-assert,K sep-tr)] end
›

lemma *c-null-guard*:
  c-null-guard (p::'a::mem-type ptr) ⟹ p ≠ NULL
  by (*fastforce simp*: *c-null-guard-def intro*: *intvl-self*)

lemma (**in** *mem-type*) *c-guard-no-wrap*:
  fixes *p* :: *'a ptr*
  assumes *cgrd*: *c-guard p*
  shows *ptr-val p ≤ ptr-val p + of-nat (size-of TYPE('a) − 1)*
  using *cgrd* **unfolding** *c-guard-def c-null-guard-def*
  apply −
  apply (*erule conjE*)
  apply (*erule contrapos-np*)
  apply (*simp add*: *intvl-def*)
  apply (*drule word-wrap-of-natD*)
  apply (*erule exE*)
  apply (*rule exI*)
  apply (*simp add*: *nat-le-Suc-less size-of-def wf-size-desc-gt*(*1*))

done

lemma *word-le-unat-bound*:
fixes *a*::*'a* ::*len word*
assumes $a \leq a + b$
shows *unat a* + *unat b* < *2* ^ *LENGTH*(*'a*)
using *assms no-olen-add-nat* **by** *blast*

lemma (**in** *mem-type*) *c-guard-no-wrap'*:
fixes *p* :: *'a ptr*
assumes *cgrd*: *c-guard p*
shows *unat* (*ptr-val p*) + *size-of TYPE*(*'a*) ≤ *addr-card*
proof −

have *szb*: *size-of TYPE*(*'a*) < *addr-card*
by (*simp add*: *local.max-size*)

have *not-null*: *0* < *size-of TYPE*(*'a*)
by (*simp add*: *size-of-def wf-size-desc-gt*(*1*))

have *sz-le*: *size-of TYPE*(*'a*) − *Suc 0* < *2* ^ *LENGTH*(*addr-bitsize*)
using *len-of-addr-card less-imp-diff-less szb* **by** *simp*

with *szb*
have *eq*: *unat* (*word-of-nat* (*size-of TYPE*(*'a*) − *1*)::*addr-bitsize word*) = *size-of* (*TYPE*(*'a*)) − *1*
apply (*subst unat-of-nat-eq*)
apply (*simp-all*)
done
from *word-le-unat-bound* [*OF c-guard-no-wrap* [*OF cgrd*], *simplified eq*]
show *?thesis*
by (*simp add*: *addr-card*)
qed

definition
c-fnptr-guard-def: *c-fnptr-guard* (*fnptr*::*unit ptr*) ≡ *ptr-val fnptr* ≠ *0*

lemma *c-fnptr-guard-NULL* [*simp*]: *c-fnptr-guard NULL* = *False*
by (*simp add*: *c-fnptr-guard-def*)

lemma *c-guardD*:
c-guard (*p*::*'a*::*mem-type ptr*) ⟹ *ptr-aligned p* ∧ *p* ≠ *NULL*
by (*clarsimp simp*: *c-guard-def c-null-guard*)

lemma *c-guard-ptr-aligned*:
c-guard p ⟹ *ptr-aligned p*

by (*simp add*: *c-guard-def*)

lemma *c-guard-NULL*:
c-guard (*p*::*'a*::*mem-type ptr*) ⟹ *p* ≠ *NULL*
by (*simp add*: *c-guardD*)

lemma *c-guard-NULL-simp* [*simp*]:
¬ *c-guard* (*NULL*::*'a*::*mem-type ptr*)
by (*force dest*: *c-guard-NULL*)

lemma *c-guard-mono*:
guard-mono (*c-guard*::*'a*::*mem-type ptr-guard*) (*c-guard*::*'b*::*mem-type ptr-guard*)
apply(*clarsimp simp*: *guard-mono-def c-guard-def*)
subgoal premises *prems* **for** *n f p*
proof −
have *guard-mono* (*ptr-aligned*::*'a*::*mem-type ptr-guard*) (*ptr-aligned*::*'b*::*mem-type ptr-guard*)
using *prems* **by** − (*rule ptr-aligned-mono*)
then show *?thesis*
using *prems*
apply −
apply(*clarsimp simp*: *guard-mono-def ptr-aligned-def c-null-guard-def typ-uinfo-t-def*)
apply(*frule field-lookup-export-uinfo-Some-rev*)
apply *clarsimp*
apply(*drule field-tag-sub* [**where** *p*=*p*])
apply(*clarsimp simp*: *field-lvalue-def field-offset-def field-offset-untyped-def typ-uinfo-t-def*)
apply(*metis* (*mono-tags*, *opaque-lifting*) *export-size-of subsetD typ-uinfo-t-def*)
done
qed
done

lemma *c-guard-NULL-fl*:
⟦ *c-guard* (*p*::*'a*::*mem-type ptr*); *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = *typ-uinfo-t TYPE*(*'b*::*mem-type*) ⟧
⟹ *0* < &(*p*→*f*)
using *c-guard-mono*
apply(*clarsimp simp*: *guard-mono-def*)
apply((*erule allE*)+, *erule impE*)
apply(*fastforce dest*: *field-lookup-export-uinfo-Some simp*: *typ-uinfo-t-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add*: *field-lvalue-def field-offset-def field-offset-untyped-def typ-uinfo-t-def*)
apply(*clarsimp simp*: *c-guard-def*)
apply(*drule c-null-guard*)+
apply(*clarsimp simp*: *word-neq-0-conv*)
done

lemma *c-guard-ptr-aligned-fl*:

⟦ *c-guard* (*p*::'*a*::*mem-type ptr*); *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = *typ-uinfo-t TYPE*('*b*::*mem-type*) ⟧ ⟹
ptr-aligned ((*Ptr* &(*p*→*f*))::'*b ptr*)
using *c-guard-mono*
apply(*clarsimp simp*: *guard-mono-def*)
apply((*erule allE*)+, *erule impE*)
apply(*fastforce dest*: *field-lookup-export-uinfo-Some simp*: *typ-uinfo-t-def*)
apply(*drule field-lookup-export-uinfo-Some*)
apply(*simp add*: *c-guard-def field-lvalue-def field-offset-def field-offset-untyped-def typ-uinfo-t-def*)
done

syntax
-sep-map :: '*a*::*c-type ptr* ⇒ '*a* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*infix sep-map*››- ↦ -)› [*56*,*51*] *56*)
-sep-map-any :: '*a*::*c-type ptr* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map-any*››- ↦ −)› [*56*] *56*)
-sep-map' :: '*a*::*c-type ptr* ⇒ '*a* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*infix sep-map'*››- ↪ -)› [*56*,*51*] *56*)
-sep-map'-any :: '*a*::*c-type ptr* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map'*››- ↪ −)› [*56*] *56*)
-tagd :: '*a*::*c-type ptr* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix tagd*››⊢$_s$ -)› [*99*] *100*)

translations
p ↦ *v* == *p* ↦^{i}(*CONST c-guard*) *v*
p ↦ − == *p* ↦^{i}(*CONST c-guard*) −
p ↪ *v* == *p* ↪^{i}(*CONST c-guard*) *v*
p ↪ − == *p* ↪^{i}(*CONST c-guard*) −
⊢$_s$ *p* == *CONST c-guard* ⊢$_s{}^i$ *p*

term *x* ↦ *y*
term (*x* ↦ *y* ∧* *y* ↦ *z*) *s*
term (*x* ↦ *y* ∧* *y* ↦ *z*) ∧* *x* ↦ *y*
term ⊢$_s$ *p*

lemma *sep-map-NULL* [*simp*]:
NULL ↦ (*v*::'*a*::*mem-type*) = *sep-false*
by (*force simp*: *sep-map-def sep-map-inv-def c-guard-def*
dest: *lift-typ-heap-g sep-conjD c-null-guard*)

lemma *sep-map'-NULL-simp* [*simp*]:
NULL ↪ (*v*::'*a*::*mem-type*) = *sep-false*
apply(*simp add*: *sep-map'-def sep-map'-inv-def sep-conj-ac*)
apply(*subst sep-conj-com*, *subst sep-map-inv-def* [*symmetric*])

apply *simp*
done

lemma *sep-map'-ptr-aligned*:
(*p* ↪ *v*) *s* ⟹ *ptr-aligned p*
by (*rule c-guard-ptr-aligned*) (*erule sep-map'-g*)

lemma *sep-map'-NULL*:
(*p* ↪ (*v*::'*a*::*mem-type*)) *s* ⟹ *p* ≠ *NULL*
by (*rule c-guard-NULL*) (*erule sep-map'-g*)

lemma *tagd-sep-false* [*simp*]:
⊢$_s$ (*NULL*::'*a*::*mem-type ptr*) = *sep-false*
by (*auto simp*: *tagd-inv-def tagd-def dest*!: *sep-conjD s-valid-g*)

syntax (**output**)
-Deref :: '*b* ⇒ '*b* (‹(‹*open-block notation*=‹*prefix Deref*››*-*)› [*1000*] *1000*)
-AssignH :: '*b* => '*b* => ('*a*,'*p*,'*f*) *com*
(‹(‹*indent*=*2 notation*=‹*mixfix AssignH*››*-* :==/ *-*)› [*30*, *30*] *23*)

print-translation ‹
let
fun deref (*Const* (*-antiquoteCur*,*-*)$*Const* (*h*,*-*)) *p* =
if h=*NameGeneration.global-heap then Syntax.const -Deref* $ *p else*
raise Match
| *deref h p* = *raise Match*
fun lift-tr [*h*,*p*] = *deref h p*
| *lift-tr ts* = *raise Match*
fun deref-update (*Const* (*heap-update*,*-*)$*l*$*r*$(*Const* (*-antiquoteCur*,*-*)$
Const (*h*,*-*))) =
if h=*NameGeneration.global-heap then Syntax.const -AssignH* $ *l* $ *r*
else raise Match
| *deref-update x* = *raise Match*
(∗ *First we need to determine if this is a heap update or normal assign* ∗)
fun deref-assign (*Const* (*-antiquoteCur*,*-*)$*Const* (*h*,*-*)) *r* =
if h=*NameGeneration.global-heap then deref-update r else*
raise Match
| *deref-assign l r* = *raise Match*
fun assign-tr [*l*,*r*] = *deref-assign l r*
| *assign-tr ts* = *raise Match*
in [(*CTypesDefs.lift*,*K lift-tr*),(*-Assign*,*K assign-tr*)] *end*
›

print-translation ‹
let
fun sep-app-tr [*l*,*r*] = *Syntax.const -sep-assert* $ *l*
| *sep-app-tr ts* = *raise Match*
in [(*sep-app*,*K sep-app-tr*)] *end*

›

syntax *-h-t-valid* :: *'a::c-type ptr* ⇒ *bool*
(‹(‹*open-block notation=‹infix h-t-valid›*›⊨$_t$ -)› [*99*] *100*)

abbreviation *lift-t-c* :: *heap-mem* × *heap-typ-desc* ⇒ *'a::c-type typ-heap* **where**
lift-t-c s == *lift-t c-guard s*

syntax *-h-t-valid* :: *heap-typ-desc* ⇒ *'a::c-type ptr* ⇒ *bool*
(‹(‹*open-block notation=‹infix h-t-valid›*›- ⊨$_t$ -)› [*99*,*99*] *100*)
translations
d ⊨$_t$ *p* == *d*,*CONST c-guard* ⊨$_t$ *p*

lemma *h-t-valid-c-guard*:
d ⊨$_t$ *p* ⟹ *c-guard p*
by (*clarsimp simp*: *h-t-valid-def*)

lemma *h-t-valid-NULL* [*simp*]:
¬ *d* ⊨$_t$ (*NULL*::*'a*::*mem-type ptr*)
by (*clarsimp simp*: *h-t-valid-def dest*!: *c-guard-NULL*)

lemma *h-t-valid-ptr-aligned*:
d ⊨$_t$ *p* ⟹ *ptr-aligned p*
by (*auto simp*: *h-t-valid-def dest*: *c-guard-ptr-aligned*)

lemma *lift-t-NULL*:
lift-t-c s (*NULL*::*'a*::*mem-type ptr*) = *None*
by (*cases s*) (*auto simp*: *lift-t-if*)

lemma *lift-t-ptr-aligned* [*simp*]:
lift-t-c s p = *Some v* ⟹ *ptr-aligned p*
by (*auto dest*: *lift-t-g c-guard-ptr-aligned*)

lemmas *c-typ-rewrs* = *typ-rewrs h-t-valid-ptr-aligned lift-t-NULL*

datatype *'gx c-exntype* = *Break* | *Return* | *Continue* | *Goto string* | *Nonlocal 'gx*

definition *is-local x* = (*x* = *Break* ∨ *x* = *Return* ∨ *x* = *Continue* ∨ (∃ *l*. *x* = *Goto l*))

lemma *is-local-simps* [*simp*]:
is-local Break
is-local Return
is-local Continue
is-local (*Goto l*)
¬*is-local* (*Nonlocal x*)
by (*auto simp add*: *is-local-def*)

primrec *the-Nonlocal* **where**
the-Nonlocal (*Nonlocal x*) = *x*

datatype *strictc-errortype* =
Div-0
| *C-Guard*
| *MemorySafety*
| *ShiftError*
| *SideEffects*
| *ArrayBounds*
| *SignedArithmetic*
| *DontReach*
| *GhostStateError*
| *UnspecifiedSyntax*
| *OwnershipError*
| *UndefinedFunction*
| *AdditionalError string*
| *ImpossibleSpec*
| *AssumeError*
| *StackOverflow*

definition *unspecified-syntax-error* :: *string* ⇒ *bool* **where**
unspecified-syntax-error s = *False*

record ('*g*, '*l*, '*e*) *state* = ('*g*, '*l*) *StateSpace.state-locals* +
global-exn-var'-' :: '*e c-exntype*

lemmas *hrs-simps* = *hrs-mem-update-def hrs-mem-def hrs-htd-update-def hrs-htd-def*

lemma *sep-map'-lift-lift*:
fixes *pa* :: '*a* :: *mem-type ptr* **and** *xf* :: '*a* ⇒ '*b* :: *mem-type*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* ≡
Some (*adjust-ti* (*typ-info-t TYPE*('*b*)) *xf* (*xfu* ∘ (λ*x* -. *x*)), *m'*)
and *xf-xfu*: *fg-cons xf* (*xfu* ∘ (λ*x* -. *x*))
shows (*pa* ↪ *v*)(*lift-state h*) ⟹ *CTypesDefs.lift* (*fst h*) (*Ptr* &(*pa*→*f*)) = *xf v*
using *fl xf-xfu*
apply (*clarsimp simp*: *o-def*)
apply (*rule sep-map'-lift* [*OF sep-map'-field-map-inv*, **where** *g1*=*c-guard*]; *simp?*)
apply (*subst* (*asm*) *surjective-pairing*, *assumption*)
apply (*simp add*: *typ-uinfo-t-def export-tag-adjust-ti*)
apply (*rule guard-mono-True*)
apply (*simp add*: *o-def*)
done

lemma *lift-t-update-fld-other*:
fixes *val* :: '*b* :: *mem-type* **and** *ptr* :: '*a* :: *mem-type ptr*

assumes *fl*: *field-ti TYPE('a) f = Some (adjust-ti (typ-info-t TYPE('b)) xf (xfu ∘ (λx -. x)))*
and *xf-xfu*: *fg-cons xf (xfu ∘ (λx -. x))*
and *disj*: *typ-uinfo-t TYPE('c :: mem-type) ⊥t typ-uinfo-t TYPE('b)*
and *cl*: *lift-t c-guard hp ptr = Some z*
shows (*lift-t c-guard (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) = (lift-t c-guard hp :: 'c :: mem-type typ-heap)*
(**is** *?LHS = ?RHS*)
proof −
let *?ati = adjust-ti (typ-info-t TYPE('b)) xf (xfu ∘ (λx -. x))*
have *eui*: *typ-uinfo-t TYPE('b) = export-uinfo (?ati)* **using** *xf-xfu*
by (*simp add*: *typ-uinfo-t-def export-tag-adjust-ti*)

have *cl'*: *lift-t c-guard (fst hp, snd hp) ptr = Some z* **using** *cl* **by** *simp*

show *?thesis* **using** *fl*
apply (*clarsimp simp add*: *hrs-mem-update-def split-def field-ti-def split*: *option.splits*)
by (*metis cl' disj eui fl h-t-valid-sub lift-t-h-t-valid lift-t-heap-update-same prod.collapse*)
qed

lemma *idupdate-compupdate-fold-congE*:
assumes *idu*: *⋀r. upd (λx. accr r) r = r*
assumes *cpu*: *⋀f f' r. upd f (upd f' r) = upd (f o f') r*
and *r*: *r = r'* **and** *v*: *accr r' = v'*
and *f*: *⋀v. v' = v ⟹ f v = f' v*
shows *upd f r = upd f' r'*
proof −
have *upd (f o (λx. accr r')) r' = upd (f' o (λx. accr r')) r'*
by (*simp add*: *o-def f v*)
then show *?thesis*
by (*simp add*: *cpu*[*symmetric*] *idu r*)
qed

lemma *field-lookup-field-ti*:
field-lookup (typ-info-t TYPE('a :: c-type)) f 0 ≡ Some (a, b) ⟹ field-ti TYPE('a) f = Some a
unfolding *field-ti-def* **by** *simp*

definition *lvar-nondet-init* ::
(('a ⇒ 'a) ⇒ (('g, 'l, 'e, 'z) state-scheme ⇒ ('g, 'l, 'e, 'z) state-scheme)
⇒ (('g, 'l, 'e, 'z) state-scheme, 'f, 'x) com **where**
lvar-nondet-init upd ≡ Spec {(s, t). ∃ v. t = (upd (λ-. v)) s}

axiomatization
asm-semantics :: *string ⇒ addr list ⇒ (heap-mem × 'a) ⇒ (addr × (heap-mem × 'a)) set* **and**
asm-fetch :: *'s ⇒ (heap-mem × 'a)* **and**

asm-store :: $('s \Rightarrow 'b) \Rightarrow (heap\text{-}mem \times 'a) \Rightarrow 's \Rightarrow 's$
where
asm-semantics-enabled: $\forall\, iv.\ asm\text{-}semantics\ nm\ addr\ iv \neq \{\}$ **and**
asm-store-eq: $\forall\, x\ s.\ ghost\text{-}data\ (asm\text{-}store\ ghost\text{-}data\ x\ s) = ghost\text{-}data\ s$

definition
asm-spec :: $'a\ itself \Rightarrow ('g \Rightarrow 'b) \Rightarrow bool \Rightarrow string$
$\Rightarrow (addr \Rightarrow ('g, 'l, 'e, 'z)\ state\text{-}scheme \Rightarrow ('g, 'l, 'e, 'z)\ state\text{-}scheme)$
$\Rightarrow (('g, 'l, 'e, 'z)\ state\text{-}scheme \Rightarrow addr\ list)$
$\Rightarrow (('g, 'l, 'e, 'z)\ state\text{-}scheme \times ('g, 'l, 'e, 'z)\ state\text{-}scheme)\ set$
where
asm-spec ti gdata vol spec lhs vs
$= \{(s, s').\ \exists (v', (gl' :: (heap\text{-}mem \times 'a)))$
$\in asm\text{-}semantics\ spec\ (vs\ s)\ (asm\text{-}fetch\ (globals\ s)).$
$s' = lhs\ v'\ (globals\text{-}update\ (asm\text{-}store\ gdata\ gl')\ s)\}$

lemma *asm-spec-enabled*:
$\exists\, s'.\ (s, s') \in asm\text{-}spec\ ti\ gdata\ vol\ spec\ lhs\ vs$
using *asm-semantics-enabled*[*rule-format*, **where** *nm* = *spec*
and *addr*=*vs s* **and** *iv*=*asm-fetch* (*globals s*)]
by (*auto simp add*: *asm-spec-def*)

lemma *asm-specE*:
$[\![\ (s, s') \in asm\text{-}spec\ (ti :: 'a\ itself)\ gdata\ vol\ spec\ lhs\ vs;$
$\bigwedge v'\ gl'.\ [\![\ (v', (gl' :: (heap\text{-}mem \times 'a))) \in asm\text{-}semantics\ spec\ (vs\ s)\ (asm\text{-}fetch\ (globals\ s));$
$s' = lhs\ v'\ (globals\text{-}update\ (asm\text{-}store\ gdata\ gl')\ s);$
$gdata\ (asm\text{-}store\ gdata\ gl'\ (globals\ s)) = gdata\ (globals\ s)\ ]\!]$
$\Longrightarrow P\ ]\!]$
$\Longrightarrow P$
by (*clarsimp simp*: *asm-spec-def asm-store-eq*)

lemmas *state-eqE* = *arg-cong*[**where** *f*=$\lambda s.$ (*globals s*, *state.more s*), *elim-format*]

lemmas *asm-store-eq-helper*
= *arg-cong2*[**where** *f*=(=) **and** *a*=*asm-store f v s*]
arg-cong2[**where** *f*=(=) **and** *c*=*asm-store f v s*] **for** *f v s*

definition *asm-semantics-ok-to-ignore* :: $'a\ itself \Rightarrow bool \Rightarrow string \Rightarrow bool$ **where**
asm-semantics-ok-to-ignore ti volatile specifier
$= (\forall\, xs\ gl.\ snd\ `\ asm\text{-}semantics\ specifier\ xs\ (gl :: (heap\text{-}mem \times 'a)) = \{gl\})$

lemma *heap-list-nth*:
$m < n \Longrightarrow heap\text{-}list\ hp\ n\ p\ !\ m = hp\ (p + of\text{-}nat\ m)$
proof (*induct m arbitrary*: *n p*)
case (*0 n' p'*)
thus *?case* **by** (*cases n'*, *simp-all*)
next
case (*Suc m' n' p'*)

```
  show ?case
  proof (cases n')
    case 0 thus ?thesis using ‹Suc m' < n'› by simp
  next
    case (Suc n'')
    hence m' < n'' using ‹Suc m' < n'› by simp
    thus ?thesis using Suc
      by (simp add: Suc.hyps ac-simps)
  qed
qed

lemma heap-update-list-id':
  heap-list hp n ptr = xs ⟹ heap-update-list ptr xs hp = hp
proof (induct xs arbitrary: ptr n)
  case Nil
  then show ?case by simp
next
  case (Cons x1 xs)
  then show ?case
  by (cases n) (auto simp add: fun-upd-idem)
qed

lemma heap-update-list-concat-fold:
  assumes ptr' = ptr + of-nat (length ys)
  shows heap-update-list ptr' xs (heap-update-list ptr ys s)
    = heap-update-list ptr (ys @ xs) s
  unfolding assms
  apply (induct ys arbitrary: ptr s)
   apply simp
  apply simp
  apply (elim meta-allE)
  apply (erule trans[rotated])
  apply (simp add: field-simps)
  done

lemma heap-update-list-concat-fold-hrs-mem:
  ptr' = ptr + of-nat (length ys) ⟹
   hrs-mem-update (heap-update-list ptr' xs)
        (hrs-mem-update (heap-update-list ptr ys) s)
    = hrs-mem-update (heap-update-list ptr (ys @ xs)) s
  by (simp add: hrs-mem-update-def split-def
                heap-update-list-concat-fold)

lemmas heap-update-list-concat-unfold
    = heap-update-list-concat-fold[OF refl, symmetric]


lemma intvl-nowrap:
  fixes x :: 'a::len word
```

shows $[\![y \neq 0; \mathit{unat}\ y + z \leq 2\ \hat{}\ \mathit{len\text{-}of}\ \mathit{TYPE}('a)]\!] \Longrightarrow x \notin \{x + y\ ..+\ z\}$
apply *clarsimp*
apply (*drule intvlD*)
apply *clarsimp*
apply (*simp add*: *unat-arith-simps*)
apply (*simp split*: *if-split-asm*)
by (*metis add-le-imp-le-left le-unat-uoi less-imp-le-nat not-less*)

lemma *intvl-off-disj*:
fixes x :: *addr*
assumes *ylt*: $y \leq \mathit{off}$
and *zoff*: $z + \mathit{off} < 2\ \hat{}\ \mathit{word\text{-}bits}$
shows $\{x\ ..+\ y\} \cap \{x + \mathit{of\text{-}nat}\ \mathit{off}\ ..+\ z\} = \{\}$
using *ylt zoff*
supply *unsigned-of-nat* [*simp del*]
apply (*cases off = 0*)
apply *simp*
apply (*rule contrapos-pp* [*OF TrueI*])
apply (*drule intvl-inter*)
apply (*erule disjE*)
subgoal premises *prems*
proof −
have $x \notin \{x + \mathit{word\text{-}of\text{-}nat}\ \mathit{off}..+z\}$
apply (*rule intvl-nowrap* [**where** $x = x$ **and** $y = \mathit{of\text{-}nat}\ \mathit{off}$ :: *addr* **and** $z = z$])
using *prems*
apply −
apply (*rule of-nat-neq-0*)
apply *simp*
apply (*unfold word-bits-len-of*)
apply *simp*
apply (*simp add*: *unat-of-nat word-bits-conv*)
done
then show *?thesis* **using** *prems* **by** *simp*
qed
subgoal
apply (*drule intvlD*)
apply *clarsimp*
by (*metis add.commute add-lessD1 le-Suc-ex nat-less-le unat32-eq-of-nat unat64-eq-of-nat*)
done

lemma *intvl-empty2*:
$(\{p\ ..+\ n\} = \{\}) = (n = 0)$
by (*auto simp add*: *intvl-def*)

lemma *heap-update-list-commute*:

$\{p\ ..+\ length\ xs\} \cap \{q\ ..+\ length\ ys\} = \{\}$
$\Longrightarrow$ *heap-update-list p xs* (*heap-update-list q ys hp*)
= *heap-update-list q ys* (*heap-update-list p xs hp*)
apply (*cases length xs* < *addr-card*)
apply (*cases length ys* < *addr-card*)
apply (*rule ext*, *simp add*: *heap-update-list-value*)
apply *blast*
apply (*simp-all add*: *addr-card intvl-overflow intvl-empty2*)
done

lemma *heap-update-commute*:
⟦ $\{$*ptr-val p ..+ size-of TYPE*('*a*)$\} \cap \{$*ptr-val q ..+ size-of TYPE*('*b*)$\} = \{\}$;
wf-fd (*typ-info-t TYPE*('*a*)); *wf-fd* (*typ-info-t TYPE*('*b*)) ⟧
$\Longrightarrow$ *heap-update p v* (*heap-update q* (*u* :: '*b* :: *c-type*) *h*)
= *heap-update q u* (*heap-update p* (*v* :: '*a* :: *c-type*) *h*)
apply (*simp add*: *heap-update-def*)
apply (*simp add*: *heap-update-list-commute heap-list-update-disjoint-same*
to-bytes-def length-fa-ti size-of-def Int-commute)
done

lemma *heap-update-padding-commute*:
⟦ $\{$*ptr-val p ..+ size-of TYPE*('*a*)$\} \cap \{$*ptr-val q ..+ size-of TYPE*('*b*)$\} = \{\}$;
length bs = *size-of TYPE*('*a*); *length bs'* = *size-of TYPE*('*b*);
wf-fd (*typ-info-t TYPE*('*a*)); *wf-fd* (*typ-info-t TYPE*('*b*))⟧
$\Longrightarrow$ *heap-update-padding p v bs* (*heap-update-padding q* (*u* :: '*b* :: *c-type*) *bs'* *h*)
= *heap-update-padding q u bs'* (*heap-update-padding p* (*v* :: '*a* :: *c-type*) *bs h*)
apply (*simp add*: *heap-update-padding-def*)
apply (*simp add*: *heap-update-list-commute heap-list-update-disjoint-same*
to-bytes-def length-fa-ti size-of-def Int-commute)

done

lemma *heap-update-padding-heap-update-commute*:
⟦ $\{$*ptr-val p ..+ size-of TYPE*('*a*)$\} \cap \{$*ptr-val q ..+ size-of TYPE*('*b*)$\} = \{\}$;
length bs = *size-of TYPE*('*a*);
wf-fd (*typ-info-t TYPE*('*a*)); *wf-fd* (*typ-info-t TYPE*('*b*))⟧
$\Longrightarrow$ *heap-update-padding p v bs* (*heap-update q* (*u* :: '*b* :: *c-type*) *h*)
= *heap-update q u* (*heap-update-padding p* (*v* :: '*a* :: *c-type*) *bs h*)
apply (*simp add*: *heap-update-heap-update-padding-conv*)
apply (*subst heap-update-padding-commute*)
apply (*auto simp add*: *heap-list-update-disjoint-same heap-update-padding-def* *length-fa-ti size-of-def to-bytes-def*)
done

lemma *heap-update-heap-update-padding-commute*:
⟦ $\{$*ptr-val p ..+ size-of TYPE*('*a*)$\} \cap \{$*ptr-val q ..+ size-of TYPE*('*b*)$\} = \{\}$;
length bs' = *size-of TYPE*('*b*);

```
      wf-fd (typ-info-t TYPE('a)); wf-fd (typ-info-t TYPE('b))⟧
       ⟹ heap-update p v (heap-update-padding q (u :: 'b :: c-type) bs' h)
            = heap-update-padding q u bs' (heap-update p (v :: 'a :: c-type) h)
  apply (simp add: heap-update-heap-update-padding-conv)
  apply (subst heap-update-padding-commute)
     apply (auto simp add: heap-list-update-disjoint-same heap-update-padding-def
length-fa-ti size-of-def to-bytes-def
      inf-commute )
  done

lemma addr-card-wb:
  addr-card = 2 ^ word-bits
  by (simp add: addr-card-def card-word word-bits-conv)

lemma fold-cong':
  a = b ⟹ xs = ys ⟹ (⋀x. x ∈ set xs =simp=> f x = g x)
    ⟹ fold f xs a = fold g ys b
  unfolding simp-implies-def
  by (metis fold-cong)


lemma arg-cong-meta: x = y ⟹ (f x = f y) ≡ True
  by simp

simproc-setup arg-cong (‹x = y›) = ‹fn phi => fn ctxt => fn ct =>
  let
    val {f, x, y, ...} = @{cterm-match (fo) ‹?f ?x = ?f ?y›} ct
    val eq = infer-instantiate ‹x = x and y = y in cprop ‹x = y›› ctxt
  in
    try (Goal.prove-internal ctxt [] eq) (fn - => asm-full-simp-tac ctxt 1)
    |> Option.map (fn thm => (@{thm arg-cong-meta} OF [thm]))
  end
  handle Pattern.MATCH => NONE | Match => NONE
›

declare [[simproc del: arg-cong]]

lemma fun-cong-meta: f = g ⟹ (f x = g x) ≡ True
  by simp

simproc-setup fun-cong (‹x = y›) = ‹fn phi => fn ctxt => fn ct =>
  let
    val {f, g, x, ...} = @{cterm-match (fo) ‹?f ?x = ?g ?x›} ct
    val ctxt = ctxt delsimps @{thms fun-eq-iff} (* prevent looping *)
    val eq = infer-instantiate ‹f = f and g = g in cprop ‹f = g›› ctxt
  in
    try (Goal.prove-internal ctxt [] eq) (fn - => asm-full-simp-tac ctxt 1)
    |> Option.map (fn thm => (@{thm fun-cong-meta} OF [thm]))
  end
```

handle Pattern.MATCH => NONE | Match => NONE
›

declare [[*simproc del*: *fun-cong*]]

abbreviation
ptr-span :: *'a*::*c-type ptr* ⇒ *addr set* **where**
ptr-span p ≡ {*ptr-val p* ..+ *size-of TYPE*(*'a*)}

lemma *nat-index-bound*:
j ∗ *a* + *k* < *a* ∗ *b* **if** *jk*: *j* < *b k* < *a* **for** *j k* :: *nat*
proof (*rule less-le-trans*)
show *j* ∗ *a* + *k* < (*b* − *1*) ∗ *a* + *a*
using *jk* **by** (*intro add-le-less-mono mult-le-mono1*) *simp*
show (*b* − *1*) ∗ *a* + *a* ≤ *a* ∗ *b*
using *jk* **by** (*simp add*: *diff-mult-distrib*)
qed

lemma *disj-ptr-span-ptr-neq*:
disjnt (*ptr-span* (*p*::*'a*::*mem-type ptr*)) (*ptr-span* (*q*::*'a*::*mem-type ptr*)) ⟹
p ≠ *q*
unfolding *disjnt-def*
by (*metis empty-iff inf.idem mem-type-self*)

lemma *field-lvalue-same-conv'*: &(*p*::*'a*:: *c-type ptr*→*f*) = &(*q*::*'a*:: *c-type ptr*→*f*)
⟷ *p* = *q*
by (*simp add*: *field-lvalue-def*)

11.27 (Partial) Pointer Lenses

11.27.1 *pointer-lense*

named-theorems
read-write-same **and**
read-write-other **and**
write-cong **and**
map-other-commute

locale *pointer-lense* =
fixes *r*::*'s* ⇒ *'a*::*mem-type ptr* ⇒ *'b*
fixes *w*::*'a ptr* ⇒ (*'b* ⇒ *'b*) ⇒ *'s* ⇒ *'s*
assumes *read-write-same*[*read-write-same*]: *r* (*w p f s*) *p* = *f* (*r s p*)
assumes *write-same*: *f* (*r s p*) = (*r s p*) ⟹ *w p f s* = *s*
assumes *write-comp*[*update-compose*, *simp*]: *w p f* (*w p g s*) = *w p* (*f o g*) *s*
assumes *write-other-commute*[*map-other-commute*]: *disjnt* (*ptr-span p*) (*ptr-span*
q) ⟹ *w q g* (*w p f s*) = *w p f* (*w q g s*)
begin

lemma *read-write-other*[*read-write-other*]:

disjnt (*ptr-span p*) (*ptr-span p'*) ⟹ *r* (*w p f s*) *p'* = *r s p'*
apply (*subst write-same*[*symmetric*, *of* λ*x*. *r s p' s p'*])
apply *simp*
apply (*subst write-other-commute*[*symmetric*])
apply *assumption*
apply (*rule read-write-same*)
done

lemma *write-cong*[*write-cong*]: *f* (*r s p*) = *f'* (*r s p*) ⟹ *w p f s* = *w p f' s*
by (*metis K-record-comp read-write-same write-comp write-same*)

lemma *write-read*: *w p* (λ-. *r s p*) *s* = *s*
by (*simp add*: *write-same*)

lemma *write-id*: *w p* (λ*x*. *x*) *s* = *s*
by (*simp add*: *write-same*)

lemma *read-write-pointwise-id*: *r* (*w p* (λ*x*. *x*) *s*) = *r s*
using *write-same* **by** *simp*

end

11.27.2 *partial-pointer-lense*

locale *partial-pointer-lense* = *is-scene m* **for** *m* :: *'a*::*c-type scene* +
fixes *r* :: *'h* ⇒ *'a ptr* ⇒ *'a upd*
fixes *w* :: *'a ptr* ⇒ *'a* ⇒ *'h upd*
assumes *r-w*[*simp*]: *r* (*w p x h*) *p y* = *m x y*
assumes *w-w*[*simp*]: *w p x* (*w p y h*) = *w p x h*
assumes *w-r*[*simp*]: *w p* (*r h p x*) *h* = *h*
assumes *w-m*[*simp*]: *w p* (*m x y*) *h* = *w p x h*
assumes *w-w-disj*: *disjnt* (*ptr-span p*) (*ptr-span q*) ⟹ *w p x* (*w q y h*) = *w q y* (*w p x h*)
begin

lemma *m-r*[*simp*]: *m* (*r h p x*) *y* = *r h p y*
by (*metis r-w w-r*)

lemma *r-w-disj*[*simp*]: *disjnt* (*ptr-span p*) (*ptr-span q*) ⟹ *r* (*w q x h*) *p y* = *r h p y*
by (*metis r-w w-r w-w-disj*)

lemma *r-m*: *r h p* (*m x y*) = *r h p y*
by (*metis m-r right*)

end

lemma *partial-pointer-lenseI*:
fixes *get upd r w*

assumes *lense get upd*
assumes *pointer-lense r w*
shows *partial-pointer-lense*
(λ*a b*. *upd* (λ-. *get a*) *b*)
(λ*h p*. *upd* (λ-. *r h p*))
(λ*p x*. *w p* (λ-. *get x*))
proof −
interpret *lense get upd* **by** *fact*
interpret *pointer-lense r w* **by** *fact*
show *?thesis*
apply *unfold-locales*
subgoal by *simp*
subgoal by (*simp add*: *comp-def*)
subgoal by *simp*
subgoal by (*simp add*: *read-write-same*)
subgoal by (*simp add*: *comp-def*)
subgoal by (*simp add*: *write-same*)
subgoal by *simp*
subgoal by (*simp add*: *write-other-commute disjnt-def*)
done
qed

lemma *pointer-lense-of-lense-fld*:
assumes *lense r w*
shows *pointer-lense* (λ*h p*. *r h* (*PTR*(′*a*) &(*p*→*f*))) (λ*p v*. *w* (*upd-fun* (*PTR*(′*a*) &(*p*→*f*)) *v*))
proof −
interpret *lense r w* **by** *fact*
show *?thesis*
apply *unfold-locales*
apply *simp*
apply (*simp add*: *upd-same upd-fun.upd-same*)
apply *simp*
apply *simp*
apply (*subst upd-fun-commute*)
apply (*simp-all add*: *field-lvalue-same-conv*′ *disj-ptr-span-ptr-neq eq-commute*)
done
qed

lemma *partial-pointer-lenseI-fld*:
fixes *get* :: ′*a*::*mem-type* ⇒ ′*b* **and** *upd r w*
assumes *1*: *lense get upd*
assumes *2*: *lense r w*
shows *partial-pointer-lense*
(λ*a b*. *upd* (λ-. *get a*) *b*)
(λ*h p*. *upd* (λ-. *r h* (*PTR*(′*b*) &(*p*→*f*))))
(λ*p x*. *w* (*upd-fun* (*PTR*(′*b*) &(*p*→*f*)) (λ-. *get x*)))
by (*rule partial-pointer-lenseI*[*OF 1 pointer-lense-of-lense-fld*, *OF 2*])

lemma *partial-pointer-lense-compose*:
assumes *partial-pointer-lense m1 r1 w1*
assumes *partial-pointer-lense m2 r2 w2*
assumes *m*[*simp*]: $\bigwedge a\ b\ c.\ m1\ a\ (m2\ b\ c) = m2\ b\ (m1\ a\ c)$
assumes *w*[*simp*]: $\bigwedge p\ a\ q\ b\ h.\ p = q \vee disjnt\ (ptr\text{-}span\ p)\ (ptr\text{-}span\ q) \Longrightarrow$
w1 p a (*w2 q b h*) = *w2 q b* (*w1 p a h*)
shows *partial-pointer-lense* ($\lambda a\ b.\ m1\ a\ (m2\ a\ b)$)
($\lambda h\ p\ x.\ r1\ h\ p\ (r2\ h\ p\ x)$)
($\lambda p\ x\ h.\ w1\ p\ x\ (w2\ p\ x\ h)$)
proof −
interpret *A*: *partial-pointer-lense m1 r1 w1* **by** *fact*
interpret *B*: *partial-pointer-lense m2 r2 w2* **by** *fact*

have [*simp*]: *m1* (*m2 a b*) *c* = *m1 b c* **for** *a b c*
by (*metis A.idem A.left m*)

have *r1-w2*[*simp*]: *p* = *q* ∨ *disjnt* (*ptr-span p*) (*ptr-span q*) ⟹
r1 (*w2 p x h*) *q y* = *r1 h q y* **for** *p q h x y*
by (*metis* (*no-types*, *lifting*) *A.r-w A.w-r disjnt-comm w*)

have *r1-r2*: *r1 h p* (*r2 h p x*) = *r2 h p* (*r1 h p x*) **for** *h p x*
by (*metis A.m-r B.m-r m*)

show *?thesis*
apply *unfold-locales*
apply (*simp-all add*: *A.w-w-disj B.w-w-disj disjnt-sym*)
apply (*simp add*: *r1-r2*)
apply (*simp flip*: *m*)
done
qed

lemma *partial-pointer-lense-id*:
partial-pointer-lense ($\lambda a.\ id$) ($\lambda h\ p.\ id$) ($\lambda p\ x.\ id$)
by (*simp add*: *partial-pointer-lense-def partial-pointer-lense-axioms-def*)

lemma *partial-pointer-lense-fold*:
fixes *rs* :: (*'h* ⇒ *'a*::*c-type ptr* ⇒ *'a* ⇒ *'a*) *list*
assumes *length ms* = *length rs length ms* = *length ws*
assumes *list-all* ($\lambda(m,\ r,\ w).$ *partial-pointer-lense m r w*) (*zip ms* (*zip rs ws*))
assumes ∀ *a b c. distinct-prop* ($\lambda m1\ m2.$ *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*)) *ms*
assumes ∀ *p a q b h.*
distinct-prop ($\lambda w1\ w2.$ *p* = *q* ∨ *disjnt* (*ptr-span p*) (*ptr-span q*) ⟶
w1 p a (*w2 q b h*) = *w2 q b* (*w1 p a h*)) *ws*
shows *partial-pointer-lense* ($\lambda a\ b.$ *fold* ($\lambda m.\ m\ a$) *ms b*)
($\lambda h\ p\ x.$ *fold* ($\lambda r.\ r\ h\ p$) *rs x*)
($\lambda p\ x\ h.$ *fold* ($\lambda w.\ w\ p\ x$) *ws h*)
using *assms*
proof (*induction ms arbitrary*: *ws rs*)
case (*Cons m ms ws' rs'*)

```
  from Cons.prems(1,2) obtain r rs w ws where [simp]: rs' = r # rs ws' = w
# ws
    by (cases ws'; cases rs'; auto)
  have ppl-ms:
    partial-pointer-lense
      (λa. fold (λm. m a) ms)
      (λh p. fold (λr. r h p) rs)
      (λp x. fold (λw. w p x) ws)
    using Cons.prems by (intro Cons.IH; simp)
  from Cons.prems(3) have ppl-m: partial-pointer-lense m r w by simp
  show ?case
    apply (simp, rule partial-pointer-lense-compose[OF ppl-ms ppl-m])
  proof −
    fix a b c
    from Cons.prems(4) have list-all (λm'. ∀ a b c. m a (m' b c) = m' b (m a c))
ms
      by (simp add: list-all-iff)
    then show fold (λm. m a) ms (m b c) = m b (fold (λm. m a) ms c)
    proof (induction ms arbitrary: c)
      case (Cons m' ms)
      then show ?case by simp metis
    qed simp
  next
    fix p q :: 'a ptr and a b :: 'a and h :: 'h
    assume p = q ∨ disjnt (ptr-span p) (ptr-span q)
    then have q = p ∨ disjnt (ptr-span q) (ptr-span p)
      by (simp add: eq-commute disjnt-comm)
    with Cons.prems(5) have list-all (λw'. ∀ b a h. w q a (w' p b h) = w' p b (w
q a h)) ws
      by (auto simp add: list-all-iff)
    then show fold (λw. w p a) ws (w q b h) = w q b (fold (λw. w p a) ws h)
    proof (induction ws arbitrary: a b h)
      case (Cons w' ws)
      then show ?case
        by simp metis
    qed simp
  qed
qed (simp add: partial-pointer-lense-id)

lemma pointer-lense-of-partial-pointer-lense:
  assumes partial-pointer-lense m r w
  assumes [simp]: ⋀a b. m a b = a
  shows pointer-lense (λh p. r h p x) (λp f h. w p (f (r h p x)) h)
proof −
  interpret partial-pointer-lense m r w by fact
  show ?thesis
    by unfold-locales (simp-all add: w-w-disj disjnt-def Int-commute)
qed
```

lemma *pointer-lense-of-partials*:
fixes *rs* :: (*'h* ⇒ *'a*::*mem-type ptr* ⇒ *'a* ⇒ *'a*) *list*
assumes ∗:
length ms = *length rs length ms* = *length ws*
list-all (λ(*m*, *r*, *w*). *partial-pointer-lense m r w*) (*zip ms* (*zip rs ws*))
and ∗∗:
⋀*a b c*. *distinct-prop* (λ*m1 m2*. *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*)) *ms*
⋀*p a q b h*. *distinct-prop* (λ*w1 w2*. *p* = *q* ∨ *disjnt* (*ptr-span p*) (*ptr-span q*) ⟶
w1 p a (*w2 q b h*) = *w2 q b* (*w1 p a h*)) *ws*
assumes *ms*: ⋀*a b*. *fold* (λ*m*. *m a*) *ms b* = *a*
assumes *R*: ⋀*h p*. *R h p* = *fold* (λ*r*. *r h p*) *rs x*
assumes *W*: ⋀*p f h*. *W p f h* = *fold* (λ*w*. *w p* (*f* (*R h p*))) *ws h*
shows *pointer-lense R W*
unfolding *R*[*abs-def*] *W*[*abs-def*]
apply (*rule pointer-lense-of-partial-pointer-lense*[*OF partial-pointer-lense-fold*, *OF* ∗])
subgoal using ∗∗(*1*) **by** *simp*
subgoal using ∗∗(*2*) **by** *simp*
subgoal by (*rule ms*)
done

end

theory *Padding-Equivalence*
imports
TypHeap
SepCode
CProof
More-Lib
begin

lemma *field-lookup-padding-field-name*:
fixes
t:: (*'a*, *'b*) *typ-info* **and**
st :: (*'a*, *'b*) *typ-info-struct* **and**
ts :: (*'a*, *'b*) *typ-info-tuple list* **and**
x :: (*'a*, *'b*) *typ-info-tuple*
shows
field-lookup t [*f*] *n* = *Some* (*s*, *m*) ⟹ *padding-field-name f* ⟹ *wf-padding t* ⟹
is-padding-tag s
field-lookup-struct st [*f*] *n* = *Some* (*s*, *m*) ⟹ *padding-field-name f* ⟹ *wf-padding-struct st* ⟹
is-padding-tag s
field-lookup-list ts [*f*] *n* = *Some* (*s*, *m*) ⟹ *padding-field-name f* ⟹ *wf-padding-list ts* ⟹
is-padding-tag s
field-lookup-tuple x [*f*] *n* = *Some* (*s*, *m*) ⟹ *padding-field-name f* ⟹ *wf-padding-tuple x* ⟹

is-padding-tag s
by (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *n m* **and** *n m* **and** *n m* **and** *n m*)
(*auto split*: *if-split-asm option.splits*)

lemma *field-lookup-access-ti-take-drop'*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
field-lookup t f m = *Some* (*s*, *m* + *n*) ⟹ *wf-fd t* ⟹ *wf-desc t* ⟹ *wf-size-desc t* ⟹ *length bs* = *size-td t* ⟹
access-ti s v (*take* (*size-td s*) (*drop n bs*)) =
(*take* (*size-td s*) (*drop n* (*access-ti t v bs*)))

field-lookup-struct st f m = *Some* (*s*, *m* + *n*) ⟹ *wf-fd-struct st* ⟹ *wf-desc-struct st* ⟹ *wf-size-desc-struct st* ⟹ *length bs* = *size-td-struct st* ⟹
access-ti s v (*take* (*size-td s*) (*drop n bs*)) =
(*take* (*size-td s*) (*drop n* (*access-ti-struct st v bs*)))

field-lookup-list ts f m = *Some* (*s*, *m* + *n*) ⟹ *wf-fd-list ts* ⟹ *wf-desc-list ts* ⟹ *wf-size-desc-list ts* ⟹ *length bs* = *size-td-list ts* ⟹
access-ti s v (*take* (*size-td s*) (*drop n bs*)) =
(*take* (*size-td s*) (*drop n* (*access-ti-list ts v bs*)))

field-lookup-tuple x f m = *Some* (*s*, *m* + *n*) ⟹ *wf-fd-tuple x* ⟹ *wf-desc-tuple x* ⟹ *wf-size-desc-tuple x* ⟹ *length bs* = *size-td-tuple x* ⟹
access-ti s v (*take* (*size-td s*) (*drop n bs*)) =
(*take* (*size-td s*) (*drop n* (*access-ti-tuple x v bs*)))
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m n s bs v* **and** *f m n s bs v* **and** *f m n s bs v* **and** *f m n s bs v*)
case (*TypDesc algn st nm*)
then show *?case*
by (*auto split*: *if-split-asm*)
(*metis TypDesc.prems*(*2*) *TypDesc.prems*(*5*) *access-ti.simps le-refl length-fa-ti take-all*)
next
term *TypDesc*
case (*TypScalar algn st d*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: *x* = *DTuple d nm y* **by** (*cases x*) *auto*

from *Cons-typ-desc.prems* **obtain**
lbs: *length bs* = *size-td-tuple x* + *size-td-list fs* **and**
wf-fd-x: *wf-fd-tuple* (*DTuple d nm y*) **and**
wf-desc-x: *wf-desc-tuple* (*DTuple d nm y*) **and**
wf-fd-fs: *wf-fd-list fs* **and**
wf-size-desc-x: *wf-size-desc-tuple* (*DTuple d nm y*) **and**
wf-size-desc-fs: *wf-size-desc-list fs* **and**
nm-notin: *nm* ∉ *dt-snd* ‘ *set fs* **and**
wf-desc-fs: *wf-desc-list fs* **and**
commutes: *fu-commutes* (*update-ti-t d*) (*update-ti-list-t fs*)
by (*auto simp add*: *x*)

from *lbs*
have *lbs-drop*: *length* (*drop* (*size-td-tuple x*) *bs*) = *size-td-list fs*
by *simp*

from *lbs*
have *lbs-take*: *length* (*take* (*size-td-tuple x*) *bs*) = *size-td-tuple* (*DTuple d nm y*)
by (*simp add*: *x*)

from *lbs wf-fd-x*
have *eq1*: *length* (*access-ti d v* (*take* (*size-td d*) *bs*)) = *size-td d*
by (*metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x*)

from *Cons-typ-desc.prems* **obtain** *f1 fxs*
where *f*: *f* = *f1*#*fxs*
by (*cases f*) *auto*

show *?case*
proof (*cases f1* = *nm*)
case *True*
show *?thesis*
proof (*cases field-lookup d fxs m*)
case *None*
from *Cons-typ-desc.prems field-lookup-list-None* [*OF nm-notin*]
have *False*
by (*simp add*: *True None f x*)
thus *?thesis* **by** *simp*
next
case (*Some r*)
from *Cons-typ-desc.prems* **have** *r*: *r* = (*s*, *m* + *n*)
by (*simp add*: *x True Some f*)
hence *fl*: *field-lookup-tuple* (*DTuple d nm y*) *f m* = *Some* (*s*, *m* + *n*)
by (*simp add*: *f True Some r*)

from *td-set-wf-size-desc*(*4*)[*OF wf-size-desc-x td-set-tuple-field-lookup-tupleD*, *OF fl*]
have *wf-size-desc s* **.**
from *wf-size-desc-gt*(*1*)[*OF this*]

have *0 < size-td s* **.**

with *td-set-tuple-offset-size-m* [*OF td-set-tuple-field-lookup-tupleD*, *OF fl*]
have *n-le*: *n < size-td d*
by *auto*

have *bound*: *size-td s + (m + n − m) ≤ size-td-tuple* (*DTuple d nm y*) **by** *fact*
from *bound*
have *take-eq*: (*take* (*size-td s*) (*drop n* (*take* (*size-td d*) *bs*))) =
(*take* (*size-td s*) (*drop n bs*))
by (*simp add*: *take-drop*)

from *Cons-typ-desc.hyps*(*1*)[*simplified x*, *OF fl wf-fd-x wf-desc-x wf-size-desc-x lbs-take*, *of v*] *lbs bound*
show *?thesis* **by** (*simp add*: *x True Some r take-eq eq1*)
qed
next
case *False*
with *Cons-typ-desc.prems*
have *fl*: *field-lookup-list fs* (*f1 # fxs*) (*m + size-td d*) = *Some* (*s*, *m + n*)
by (*clarsimp simp add*: *x f False*)
hence *n-bound*: *size-td d ≤ n*
by (*meson field-lookup-offset-le*(*3*) *nat-add-left-cancel-le*)

from *fl*
have *fl*: *field-lookup-list fs* (*f1 # fxs*) (*m + size-td d*) = *Some* (*s*, (*m + size-td d*) + (*n − size-td d*))
by (*metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD*)
from *n-bound*
have *take-eq*: (*take* (*size-td s*) (*drop* (*n − size-td d + size-td d*) *bs*)) =
(*take* (*size-td s*) (*drop n bs*))
by *simp*

from *Cons-typ-desc.hyps*(*2*) [*OF fl wf-fd-fs wf-desc-fs wf-size-desc-fs lbs-drop*, *of v*]
False n-bound
show *?thesis*
by (*simp add*: *x f eq1*)
qed
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **apply** (*cases f*) **by** (*auto split*: *if-split-asm*)
qed

lemma *field-lookup-access-ti-take-drop*:
field-lookup t f 0 = Some (*s*, *n*) ⟹ *wf-fd t* ⟹ *wf-desc t* ⟹ *wf-size-desc t* ⟹

length bs = size-td t ⟹
access-ti s v (take (size-td s) (drop n bs)) =
(take (size-td s) (drop n (access-ti t v bs)))
using *field-lookup-access-ti-take-drop'* [**where** *m=0*] **by** *auto*

lemma *field-lookup-nth-focus'*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
⟦*field-lookup t f m = Some (s, m + n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)

⟦*field-lookup-struct st f m = Some (s, m + n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td-struct st*;
wf-fd-struct st; *wf-desc-struct st*; *wf-size-desc-struct st*⟧ ⟹
access-ti-struct st v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)

⟦*field-lookup-list ts f m = Some (s, m + n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td-list ts*;
wf-fd-list ts; *wf-desc-list ts*; *wf-size-desc-list ts*⟧ ⟹
access-ti-list ts v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)

⟦*field-lookup-tuple x f m = Some (s, m + n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td-tuple x*;
wf-fd-tuple x; *wf-desc-tuple x*; *wf-size-desc-tuple x*⟧ ⟹
access-ti-tuple x v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m n i s bs v* **and** *f m n i s bs v* **and** *f m n i s bs v* **and** *f m n i s bs v*)
case (*TypDesc algn st nm*)
then show *?case* **by** (*auto split*: *if-split-asm*)
next
case (*TypScalar sz algn d*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: *x = DTuple d nm y* **by** (*cases x*) *auto*
from *Cons-typ-desc.prems* **obtain**
lbs: *length bs = size-td-tuple x + size-td-list fs* **and**
wf-fd-x: *wf-fd-tuple* (*DTuple d nm y*) **and**

```
    wf-desc-x: wf-desc-tuple (DTuple d nm y) and
    wf-fd-fs: wf-fd-list fs and
    wf-size-desc-x: wf-size-desc-tuple (DTuple d nm y) and
    wf-size-desc-fs: wf-size-desc-list fs and
    nm-notin:  nm ∉ dt-snd ‘ set fs and
    wf-desc-fs: wf-desc-list fs and
    i-lower: n ≤ i and
    i-upper: i < n + size-td s
    by (auto simp add: x)

  from lbs
  have lbs-drop: length (drop (size-td-tuple x) bs) = size-td-list fs
    by simp

  from lbs
  have lbs-take: length (take (size-td-tuple x) bs) = size-td-tuple (DTuple d nm y)
    by (simp add: x)

  from lbs wf-fd-x
  have eq1: length (access-ti d v (take (size-td d) bs)) = size-td d
    by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x)

  from Cons-typ-desc.prems obtain f1 fxs
    where f: f = f1#fxs
    by (cases f) auto

  thm Cons-typ-desc.hyps [simplified x]

  show ?case
  proof (cases f1 = nm)
    case True
    show ?thesis
    proof (cases field-lookup d fxs m)
      case None
      from Cons-typ-desc.prems field-lookup-list-None [OF nm-notin]
      have False
        by (simp add: True None f x)
      thus ?thesis by simp
    next
      case (Some r)
      from Cons-typ-desc.prems have r: r = (s, m + n)
        by (simp add: x True Some f)
      hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
        by  (simp add: f True Some r)
     from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
      have wf-size-desc s .
      from wf-size-desc-gt(1)[OF this]
      have 0 < size-td s .
```

```
    with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
    have n-le: n < size-td d
      by auto

    have i-in-d: i < size-td d
      using ‹size-td s + (m + n − m) ≤ size-td-tuple (DTuple d nm y)› i-upper
by auto
     have bound: size-td s + (m + n − m) ≤ size-td-tuple (DTuple d nm y) by
fact
    from bound
    have take-eq: (take (size-td s) (drop n (take (size-td d) bs))) =
       (take (size-td s) (drop n bs))
      by (simp add: take-drop)
   from Cons-typ-desc.hyps(1)[simplified x, OF fl i-lower i-upper lbs-take wf-fd-x
wf-desc-x wf-size-desc-x, of v] lbs bound
    show ?thesis
      by (simp add: x True Some r take-eq eq1 nth-append i-in-d)
  qed
 next
  case False
  with Cons-typ-desc.prems
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
    by (clarsimp simp add: x f False)
  hence n-bound: size-td d ≤ n
    by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

  from fl
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, (m + size-td
d) + (n − size-td d))
    by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD)
  from n-bound
  have take-eq: (take (size-td s) (drop (n − size-td d + size-td d) bs)) =
        (take (size-td s) (drop n bs))
    by simp

  have i-lower': n − size-td d ≤ i − size-td d
    using diff-le-mono i-lower by blast

  have i-upper': i − size-td d < n − size-td d + size-td s
    using i-lower i-upper by linarith

  have i-notin-d: ¬ i < size-td d
    by (meson i-lower leD less-le-trans n-bound)

 from Cons-typ-desc.hyps(2) [OF fl i-lower' i-upper' lbs-drop wf-fd-fs wf-desc-fs
wf-size-desc-fs, where v=v]
    False n-bound
```

show *?thesis*
by (*simp add*: *x f eq1 nth-append i-notin-d*)
qed
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **apply** (*cases f*) **by** (*auto split*: *if-split-asm*)
qed

lemma *field-lookup-nth-focus*:
⟦*field-lookup t f 0 = Some* (*s*, *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t v bs ! *i* = *access-ti s v* (*take* (*size-td s*) (*drop n bs*)) ! (*i* − *n*)
using *field-lookup-nth-focus′* [**where** *m=0*] **by** *auto*

lemma *field-lookup-nth-update-focus′*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
⟦*field-lookup t f m = Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti t v bs*) *n*

⟦*field-lookup-struct st f m = Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-struct st*;
wf-fd-struct st; *wf-desc-struct st*; *wf-size-desc-struct st*⟧ ⟹
access-ti-struct st v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti-struct st v bs*) *n*

⟦*field-lookup-list ts f m = Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-list ts*;
wf-fd-list ts; *wf-desc-list ts*; *wf-size-desc-list ts*⟧ ⟹
access-ti-list ts v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti-list ts v bs*) *n*

⟦*field-lookup-tuple x f m = Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-tuple x*;
wf-fd-tuple x; *wf-desc-tuple x*; *wf-size-desc-tuple x*⟧ ⟹
access-ti-tuple x v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti-tuple x v bs*) *n*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m n i s bs v* **and** *f m n i s bs v*

and *f m n i s bs v* **and** *f m n i s bs v*)
case (*TypDesc algn st nm*)
then show *?case*
by (*auto split*: *if-split-asm simp add*: *super-update-bs-def*)
(*metis TypDesc.prems*(*4*) *TypDesc.prems*(*5*) *access-ti.simps le-refl length-fa-ti length-list-update*)
next
case (*TypScalar sz algn d*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: *x* = *DTuple d nm y* **by** (*cases x*) *auto*
from *Cons-typ-desc.prems* **obtain**
lbs: *length bs* = *size-td-tuple x* + *size-td-list fs* **and**
wf-fd-x: *wf-fd-tuple* (*DTuple d nm y*) **and**
wf-desc-x: *wf-desc-tuple* (*DTuple d nm y*) **and**
wf-fd-fs: *wf-fd-list fs* **and**
wf-size-desc-x: *wf-size-desc-tuple* (*DTuple d nm y*) **and**
wf-size-desc-fs: *wf-size-desc-list fs* **and**
nm-notin: *nm* ∉ *dt-snd* ‘ *set fs* **and**
wf-desc-fs: *wf-desc-list fs* **and**
i-lower: *n* ≤ *i* **and**
i-upper: *i* < *n* + *size-td s*
by (*auto simp add*: *x*)

from *lbs*
have *lbs-drop*: *length* (*drop* (*size-td-tuple x*) *bs*) = *size-td-list fs*
by *simp*

from *lbs*
have *lbs-take*: *length* (*take* (*size-td-tuple x*) *bs*) = *size-td-tuple* (*DTuple d nm y*)
by (*simp add*: *x*)

from *lbs wf-fd-x*
have *eq1*: *length* (*access-ti d v* (*take* (*size-td d*) *bs*)) = *size-td d*
by (*metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x*)

from *Cons-typ-desc.prems* **obtain** *f1 fxs*
where *f*: *f* = *f1*#*fxs*
by (*cases f*) *auto*

thm *Cons-typ-desc.hyps* [*simplified x*]

```
  show ?case
  proof (cases f1 = nm)
    case True
    show ?thesis
    proof (cases field-lookup d fxs m)
      case None
      from Cons-typ-desc.prems field-lookup-list-None [OF nm-notin]
      have False
        by (simp add: True None f x)
      thus ?thesis by simp
    next
      case (Some r)
      from Cons-typ-desc.prems have r: r = (s, m + n)
        by (simp add: x True Some f)
      hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
        by  (simp add: f True Some r)
     from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
      have wf-size-desc s .
      from wf-size-desc-gt(1)[OF this]
      have 0 < size-td s .


      with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
      have n-le: n < size-td d
        by auto

      have i-in-d: i < size-td d
        using ‹size-td s + (m + n − m) ≤ size-td-tuple (DTuple d nm y)› i-upper
by auto
       have bound: size-td s + (m + n − m) ≤ size-td-tuple (DTuple d nm y) by
fact
      from bound
      have take-eq: (take (size-td s) (drop n (take (size-td d) bs))) =
         (take (size-td s) (drop n bs))
        by (simp add: take-drop)
      from bound
       have take-upd-eq:  ((take (size-td d) bs)[i := b]) = (take (size-td d) (bs[i :=
b]))
        by (simp add: take-update-swap)

       have take-eq1: take (size-td s) (drop n (take (size-td d) bs)) = take (size-td
s) (drop n bs)
        using take-eq by blast

      have l-take-s: length (take (size-td s) (drop n bs)) = size-td s
        using bound lbs-take by auto

      have sz-s: size-td s ≤ length bs − n
```

```
      using l-take-s by auto

    from fl
    have fd-cons-s: fd-cons s
      using wf-fd-consD wf-fd-field-lookup(4) wf-fd-x by blast
    have l-acc-s: length (access-ti s v ((take (size-td s) (drop n bs))[i − n := b]))
= size-td s
      by (simp add: fd-cons-length [OF fd-cons-s] sz-s eq1)

    have take-eq2: (take (size-td s) (drop n (take (size-td d) bs)))[i − n := b] =
(take (size-td s) (drop n bs))[i − n := b]
      using take-eq1 by presburger

    note hyp = Cons-typ-desc.hyps(1)[simplified x, OF fl i-lower i-upper lbs-take
wf-fd-x wf-desc-x wf-size-desc-x, of v,
        simplified, simplified x, simplified, simplified take-upd-eq]
    from lbs bound
    show ?thesis
      apply (simp add: x True Some r take-eq take-upd-eq eq1 list-update-append
i-in-d)
      apply (simp add: hyp)
      apply (subst super-update-bs-append1)
       apply (simp add: l-acc-s eq1)
      apply (simp add: take-eq2)
      done
  qed
 next
  case False
  with Cons-typ-desc.prems
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
    by (clarsimp simp add: x f False)
  hence n-bound: size-td d ≤ n
    by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

  from fl
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, (m + size-td
d) + (n − size-td d))
    by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD)
  from n-bound
  have take-eq: (take (size-td s) (drop (n − size-td d + size-td d) bs)) =
        (take (size-td s) (drop n bs))
    by simp

  have i-lower': n − size-td d ≤ i − size-td d
    using diff-le-mono i-lower by blast

  have i-upper': i − size-td d < n − size-td d + size-td s
    using i-lower i-upper by linarith
```

```
    have i-notin-d: ¬ i < size-td d
      by (meson i-lower leD less-le-trans n-bound)

    from i-notin-d have take-d-eq: take (size-td d) (bs[i := b]) = take (size-td d)
bs
      by simp
    have drop-shift: (drop (size-td d) bs)[i − size-td d := b] = drop (size-td d) (bs[i
:= b])
      by (metis drop-update-swap i-notin-d le-def)

     note hyp = Cons-typ-desc.hyps(2) [OF fl i-lower′ i-upper′ lbs-drop wf-fd-fs
wf-desc-fs wf-size-desc-fs, where v=v,
        simplified x, simplified, simplified drop-shift]
    from False n-bound
    show ?thesis
      apply (simp add: x f eq1 nth-append i-notin-d)
      apply (subst super-update-bs-append2)
       apply (simp add: eq1)
      apply (simp add: take-d-eq add: eq1)
      apply (simp add: hyp)
      done
  qed
next
  case (DTuple-typ-desc d nm y)
then show ?case apply (cases f) by (auto split: if-split-asm)
qed

lemma field-lookup-nth-update-focus:
shows
⟦field-lookup t f 0 = Some (s, n); n ≤ i; i < n + size-td s; length bs = size-td t;
  wf-fd t; wf-desc t; wf-size-desc t⟧ ⟹
   access-ti t v (bs[i := b]) =
    super-update-bs (access-ti s v ((take (size-td s) (drop n bs))[i − n := b]))
     (access-ti t v bs) n
  using field-lookup-nth-update-focus′ [where m=0] by auto


context mem-type
begin
lemma mem-type-field-lookup-access-ti-take-drop:
  assumes fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)
  assumes lbs : length bs = size-of TYPE('a)
  shows access-ti s v (take (size-td s) (drop n bs)) =
          take (size-td s) (drop n (access-ti (typ-info-t TYPE('a)) v bs))
proof −
  have wf-fd: wf-fd (typ-info-t TYPE('a))
    by (simp add: wf-fdp-fdD wf-lf-fdp)
  have wf-desc: wf-desc (typ-info-t TYPE('a)) by simp
  have wf-size: wf-size-desc (typ-info-t TYPE('a)) by simp
```

from *field-lookup-access-ti-take-drop* [*OF fl wf-fd wf-desc wf-size lbs* [*simplified size-of-def*]]
show *?thesis* **.**
qed

lemma *mem-type-update-ti-super-update-bs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lv*: *length v* = *size-td s*
shows *update-ti s v* (*update-ti* (*typ-info-t TYPE*('*a*)) *bs w*) =
update-ti (*typ-info-t TYPE*('*a*)) (*super-update-bs v bs n*) *w*
proof −
have *wf-fd*: *wf-fd* (*typ-info-t TYPE*('*a*))
by (*simp add*: *wf-fdp-fdD wf-lf-fdp*)
have *lsuper*: *length* (*super-update-bs v bs n*) = *size-td* (*typ-info-t TYPE*('*a*))
by (*metis add.commute field-lookup-offset-size' fl lbs length-super-update-bs local.size-of-def lv*)

from *fi-fu-consistentD* [*OF fl wf-fd lbs* [*simplified size-of-def*] *lv*, *of w*] *lbs*
show *?thesis*
by (*simp add*: *update-ti-t-def lsuper lv size-of-def*)
qed

lemma *mem-type-update-ti-from-bytes-super-update-bs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lv*: *length v* = *size-td s*
shows *update-ti s v* (*from-bytes bs*) = *update-ti* (*typ-info-t TYPE*('*a*)) (*super-update-bs v bs n*) *undefined*
proof −
from *mem-type-update-ti-super-update-bs* [*OF fl lbs lv*, *of undefined*]
show *?thesis*
by (*simp add*: *from-bytes-def update-ti-t-def lbs size-of-def*)
qed

lemma *mem-type-field-lookup-nth-focus*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *i-lower*: *n* ≤ *i*
assumes *i-upper*: *i* < *n* + *size-td s*
assumes *lbs* : *length bs* = *size-of TYPE*('*a*)
shows *access-ti* (*typ-info-t TYPE*('*a*)) *v bs* ! *i* =
access-ti s v (*take* (*size-td s*) (*drop n bs*)) ! (*i* − *n*)
proof −
have *wf-fd*: *wf-fd* (*typ-info-t TYPE*('*a*))
by (*simp add*: *wf-fdp-fdD wf-lf-fdp*)
have *wf-desc*: *wf-desc* (*typ-info-t TYPE*('*a*)) **by** *simp*
have *wf-size*: *wf-size-desc* (*typ-info-t TYPE*('*a*)) **by** *simp*
from *field-lookup-nth-focus* [*OF fl i-lower i-upper lbs* [*simplified size-of-def*] *wf-fd*

wf-desc wf-size]
show *?thesis* **.**
qed

lemma *mem-type-field-lookup-nth-update-focus*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *i-lower*: $n \leq i$
assumes *i-upper*: $i < n +$ *size-td s*
assumes *lbs* : *length bs* = *size-of TYPE*('*a*)
shows
access-ti (*typ-info-t TYPE*('*a*)) *v* (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti* (*typ-info-t TYPE*('*a*)) *v bs*) *n*
proof −
have *wf-fd*: *wf-fd* (*typ-info-t TYPE*('*a*))
by (*simp add*: *wf-fdp-fdD wf-lf-fdp*)
have *wf-desc*: *wf-desc* (*typ-info-t TYPE*('*a*)) **by** *simp*
have *wf-size*: *wf-size-desc* (*typ-info-t TYPE*('*a*)) **by** *simp*
from *field-lookup-nth-update-focus* [*OF fl i-lower i-upper lbs* [*simplified size-of-def*]
wf-fd wf-desc wf-size]
show *?thesis* **.**
qed
end

ML ‹@{*term* *xs*[*i* := *v*]}›

lemma *nth-take-drop-index-shift*:
$n \leq i \Longrightarrow i < m + n \Longrightarrow m + n \leq$ *length xs* $\Longrightarrow$ *take m* (*drop n xs*) ! ($i - n$) =
xs ! *i*
proof (*induct xs arbitrary*: *n i m*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x1 xs*)
then show *?case* **by** *auto*
qed

lemma *super-update-bs-update-index-shift*: $n \leq i \Longrightarrow i - n <$ *length bs* $\Longrightarrow n +$
length bs $\leq$ *length xbs* $\Longrightarrow$
(*super-update-bs bs xbs n*)[*i* := *b*] = *super-update-bs* (*bs*[*i* − *n* := *b*]) *xbs n*
apply (*simp add*: *super-update-bs-def*)
apply (*simp add*: *list-update-append*)
done

lemma *super-update-bs-nth-shift*:
$n \leq i \Longrightarrow i - n <$ *length bs* $\Longrightarrow n +$ *length bs* $\leq$ *length xbs* $\Longrightarrow$ *super-update-bs*

```
bs xbs n ! i = bs ! (i − n)
  apply (simp add: super-update-bs-def)
  apply (simp add: nth-append)
  done

lemma (in mem-type) field-lookup-is-padding-byte-outer-to-inner:
  assumes fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)
  assumes lower-bound-x: n ≤ x
  assumes upper-bound-x: x < n + size-td s
 assumes is-padding: padding-base.is-padding-byte (access-ti (typ-info-t (TYPE('a))))
(update-ti (typ-info-t (TYPE('a)))) (size-td (typ-info-t (TYPE('a)))) x
  shows padding-base.is-padding-byte (access-ti s) (update-ti s) (size-td s) (x − n)
proof (rule padding-base.is-padding-byteI)
  from lower-bound-x upper-bound-x show x − n < size-td s by simp
next
  fix v::'a and bs::byte list
  assume x − n < size-td s
  assume lbs: length bs = size-td s
  from fl have sz: size-td s + n ≤ size-of TYPE('a)
    by (simp add: field-lookup-offset-size' size-of-def)

  obtain pfx sfx xbs where
    xbs: xbs = pfx @ bs @ sfx and
    lpfx: length pfx = n and
    lsfx: length sfx = size-of TYPE('a) − n − size-td s
    by (metis Ex-list-of-length)
  with sz have lxbs: length xbs = size-of TYPE('a)
    by (metis add-leD2 add-le-imp-le-diff lbs le-add-diff-inverse length-append)
  from xbs lpfx lsfx xbs lbs have bs: (take (size-td s) (drop n xbs)) = bs
    by simp

  have bound: x − n < size-td s by fact

  have lacc: length (access-ti (typ-info-t TYPE('a)) v xbs) = size-of TYPE('a)
    by (simp add: length-fa-ti lxbs size-of-def)

  from mem-type-field-lookup-access-ti-take-drop [OF fl lxbs, simplified bs, of v]
  have acc-conv: access-ti s v bs =
        take (size-td s) (drop n (access-ti (typ-info-t TYPE('a)) v xbs)) .
  from padding-base.is-padding-byte-acc-eq [OF is-padding, of xbs v] lxbs
  have access-ti (typ-info-t TYPE('a)) v xbs ! x = xbs ! x
    by (simp add: size-of-def)
  moreover have xbs ! x = bs ! (x − n)
    using xbs lpfx lsfx lxbs bound
    apply (simp add: xbs)
   by (metis bs drop-append-miracle le-def lower-bound-x nth-append nth-take xbs)
  moreover have take (size-td s) (drop n (access-ti (typ-info-t TYPE('a)) v xbs))
! (x − n) =
    access-ti (typ-info-t TYPE('a)) v xbs ! x
```

```
    using lacc lxbs bound lower-bound-x upper-bound-x
    by (metis less-diff-conv2 nth-take-drop-index-shift sz)

  ultimately show access-ti s v bs ! (x − n) = bs ! (x − n)
    by (simp add: acc-conv)
next
  fix v::'a and bs::byte list and b::byte
  assume x − n < size-td s
  assume lbs: length bs = size-td s

  from fl have sz: size-td s + n ≤ size-of TYPE('a)
    by (simp add: local.field-lookup-offset-size local.size-of-def)

  obtain pfx sfx xbs where
    xbs: xbs = pfx @ bs @ sfx and
    lpfx: length pfx = n and
    lsfx: length sfx = size-of TYPE('a) − n − size-td s
    by (metis Ex-list-of-length)

  with sz have lxbs: length xbs = size-of TYPE('a)
    by (metis add-leD2 add-le-imp-le-diff lbs le-add-diff-inverse length-append)
  from xbs lpfx lsfx xbs lbs have bs: (take (size-td s) (drop n xbs)) = bs
    by simp

  have v-upd-conv: (update-ti (typ-info-t TYPE('a)) (to-bytes v xbs) v) = v
   by (simp add: fu-fa length-fa-ti local.size-of-def local.to-bytes-def lxbs update-ti-update-ti-t)

   have lxto: length (to-bytes v xbs) = size-of TYPE('a)
     by (simp add: length-fa-ti lxbs size-of-def to-bytes-def)

  from lxto lbs
  have lsuper: length (super-update-bs bs (to-bytes v xbs) n) = size-of TYPE('a)
    using sz by auto

  have bound: x − n < size-td s by fact

  have xbs-super: xbs = super-update-bs bs xbs n
    by (simp add: lpfx super-update-bs-def xbs)
  from mem-type-update-ti-super-update-bs [OF fl lxto lbs, of v, simplified v-upd-conv]
  have upd-eq1:
    update-ti s bs v =
      update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v xbs) n) v .

  have lbs': length (bs[x − n := b]) = size-td s
    using lbs by auto

  from mem-type-update-ti-super-update-bs [OF fl lxto lbs', of v, simplified v-upd-conv]
  have upd-eq2:
    update-ti s (bs[x − n := b]) v =
```

update-ti (*typ-info-t TYPE*('*a*)) (*super-update-bs* (*bs*[*x* − *n* := *b*]) (*to-bytes v xbs*) *n*) *v* **.**

from *lxto lxbs lbs lower-bound-x upper-bound-x bound sz*
have *super-eq*: (*super-update-bs bs* (*to-bytes v xbs*) *n*)[*x* := *b*] = *super-update-bs* (*bs*[*x* − *n* := *b*]) (*to-bytes v xbs*) *n*
by (*simp add*: *super-update-bs-update-index-shift*)

from *padding-base.is-padding-byte-upd-eq* [*OF is-padding*, *of* (*super-update-bs bs* (*to-bytes v xbs*) *n*) *v b*] *lsuper*
have *upd-pad*: *update-ti* (*typ-info-t TYPE*('*a*)) (*super-update-bs bs* (*to-bytes v xbs*) *n*) *v* =
update-ti (*typ-info-t TYPE*('*a*)) ((*super-update-bs bs* (*to-bytes v xbs*) *n*)[*x* := *b*]) *v*
by (*simp add*: *size-of-def*)
show *update-ti s bs v* = *update-ti s* (*bs*[*x* − *n* := *b*]) *v*
apply (*simp add*: *upd-eq1*)
apply (*simp add*: *upd-pad*)
apply (*simp add*: *upd-eq2*)
apply (*simp add*: *super-eq*)
done
qed

lemma (**in** *mem-type*) *field-lookup-is-padding-byte-inner-to-outer*:
assumes *fl*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*s*, *n*)
assumes *lower-bound-x*: *n* ≤ *x*
assumes *upper-bound-x*: *x* < *n* + *size-td s*
assumes *is-padding*: *padding-base.is-padding-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) (*x* − *n*)
shows *padding-base.is-padding-byte* (*access-ti* (*typ-info-t* (*TYPE*('*a*)))) (*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
proof (*rule padding-base.is-padding-byteI*)
from *lower-bound-x upper-bound-x fl*
show *x* < *size-td* (*typ-info-t TYPE*('*a*))
using *field-lookup-offset-size'* **by** *fastforce*
next
fix *v*::'*a* **and** *bs*::*byte list*
assume *sz*: *x* < *size-td* (*typ-info-t TYPE*('*a*))
assume *lbs*: *length bs* = *size-td* (*typ-info-t TYPE*('*a*))
from *mem-type-field-lookup-nth-focus* [*OF fl lower-bound-x upper-bound-x*, *simplified size-of-def*, *OF lbs*]
have *eq1*:
access-ti (*typ-info-t TYPE*('*a*)) *v bs* ! *x* =
access-ti s v (*take* (*size-td s*) (*drop n bs*)) ! (*x* − *n*) **.**
from *lbs fl*
have *l-take-drop*: *length* (*take* (*size-td s*) (*drop n bs*)) = *size-td s*
using *field-lookup-offset-size'* **by** *fastforce*

from *padding-base.is-padding-byte-acc-eq* [*OF is-padding l-take-drop*, *of v*]

have *eq2*:
access-ti s v (*take* (*size-td s*) (*drop n bs*)) ! (*x* − *n*) =
take (*size-td s*) (*drop n bs*) ! (*x* − *n*) **.**
have *eq3*:
take (*size-td s*) (*drop n bs*) ! (*x* − *n*) = *bs* ! *x*
by (*metis add.commute field-lookup-offset-size′ fl lbs lower-bound-x nth-take-drop-index-shift upper-bound-x*)

show *access-ti* (*typ-info-t TYPE*('*a*)) *v bs* ! *x* = *bs* ! *x*
apply (*simp add*: *eq1*)
apply (*simp add*: *eq2*)
apply (*simp add*: *eq3*)
done
next
fix *v*::'*a* **and** *bs*::*byte list* **and** *b*::*byte*
assume *sz*: *x* < *size-td* (*typ-info-t TYPE*('*a*))
assume *lbs*: *length bs* = *size-td* (*typ-info-t TYPE*('*a*))

have *v-upd-conv*: (*update-ti* (*typ-info-t TYPE*('*a*)) (*to-bytes v bs*) *v*) = *v*
by (*simp add*: *fu-fa lbs length-fa-ti to-bytes-def update-ti-update-ti-t*)

have *lto*: *length* (*to-bytes v bs*) = *size-td* (*typ-info-t TYPE*('*a*))
by (*simp add*: *lbs length-fa-ti to-bytes-def*)

note *upd-focus* = *mem-type-update-ti-super-update-bs* [*OF fl*, *simplified size-of-def*, *OF lto*, **where** *w*=*v*, *simplified v-upd-conv*]

from *lbs lower-bound-x upper-bound-x sz*
have *lbs1*: *length* (*take* (*size-td s*) (*drop n bs*)) = *size-td s*
by (*metis add-le-imp-le-diff field-lookup-offset-size′ fl length-drop length-take min.absorb2*)

from *lbs lower-bound-x upper-bound-x*
have *bs-upd-eq*: (*take* (*size-td s*) (*drop n bs*))[*x* − *n* := *b*] = *take* (*size-td s*) (*drop n* (*bs*[*x* := *b*]))
by (*metis drop-update-swap take-update-swap*)

from *lbs lbs1 lower-bound-x upper-bound-x*
have *lbs2*: *length* (*take* (*size-td s*) (*drop n* (*bs*[*x* := *b*]))) = *size-td s*
by (*metis bs-upd-eq length-list-update*)

from *lbs lbs1*
have *bs-super-conv*: (*super-update-bs* (*take* (*size-td s*) (*drop n bs*)) *bs n*) = *bs*
by (*metis append-take-drop-id super-update-bs-def take-drop-append*)

from *lbs lbs1 lower-bound-x upper-bound-x*
have *bs-upd-super-conv*: (*super-update-bs* (*take* (*size-td s*) (*drop n* (*bs*[*x* := *b*]))) *bs n*) = *bs*[*x* := *b*]

by (*metis add.commute bs-super-conv bs-upd-eq field-lookup-offset-size' fl nat-diff-less super-update-bs-update-index-shift*)

from *mem-type-update-ti-super-update-bs* [*OF fl*, *simplified size-of-def*, *OF lbs lbs1*, *simplified bs-super-conv*, **where** *w=v*]
have *eq1*:
update-ti (*typ-info-t TYPE*('*a*)) *bs v* =
update-ti s (*take* (*size-td s*) (*drop n bs*)) (*update-ti* (*typ-info-t TYPE*('*a*)) *bs v*) **by** *simp*

from *mem-type-update-ti-super-update-bs* [*OF fl*, *simplified size-of-def*, *OF lbs lbs2*, *simplified bs-upd-super-conv*, **where** *w=v*]
have *eq2*:
update-ti s (*take* (*size-td s*) (*drop n* (*bs*[*x* := *b*])))
(*update-ti* (*typ-info-t TYPE*('*a*)) *bs v*) =
update-ti (*typ-info-t TYPE*('*a*)) (*bs*[*x* := *b*]) *v* **.**

thm *upd-focus* [*OF lbs1*]
note *pad-eq*= *padding-base.is-padding-byte-upd-eq* [*OF is-padding*, *OF lbs1*, **where** *b=b*]
show *update-ti* (*typ-info-t TYPE*('*a*)) *bs v* =
update-ti (*typ-info-t TYPE*('*a*)) (*bs*[*x* := *b*]) *v*
apply (*subst eq1*)
apply (*simp add*: *pad-eq*)
apply (*simp add*: *bs-upd-eq*)
apply (*simp add*: *eq2*)
done
qed

lemma (**in** *mem-type*) *field-lookup-is-padding-byte*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lower-bound-x*: $n \leq x$
assumes *upper-bound-x*: $x < n +$ *size-td s*
shows
padding-base.is-padding-byte (*access-ti s*) (*update-ti s*) (*size-td s*) ($x - n$) $\longleftrightarrow$
padding-base.is-padding-byte
(*access-ti* (*typ-info-t* (*TYPE*('*a*)))) (*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
using *field-lookup-is-padding-byte-outer-to-inner* [*OF fl lower-bound-x upper-bound-x*]
field-lookup-is-padding-byte-inner-to-outer [*OF fl lower-bound-x upper-bound-x*]
by *blast*

lemma (**in** *mem-type*) *field-lookup-is-value-byte-outer-to-inner*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lower-bound-x*: $n \leq x$
assumes *upper-bound-x*: $x < n +$ *size-td s*
assumes *is-value*: *padding-base.is-value-byte* (*access-ti* (*typ-info-t* (*TYPE*('*a*)))) (*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*

shows *padding-base.is-value-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) (*x − n*)
proof (*rule padding-base.is-value-byteI*)
from *lower-bound-x upper-bound-x* **show** *x − n < size-td s* **by** *simp*
next
fix *v*::*'a* **and** *bs*::*byte list* **and** *bs'*::*byte list*
assume *x − n < size-td s*
assume *lbs*: *length bs = size-td s*
assume *lbs'*: *length bs' = size-td s*
from *fl* **have** *sz*: *size-td s + n ≤ size-of TYPE*(*'a*)
by (*metis* (*no-types*, *lifting*) *diff-add-zero diff-is-0-eq dual-order.trans field-lookup-offset-size' size-of-def*)

obtain *pfx sfx xbs* **where**
xbs: *xbs = pfx @ bs @ sfx* **and**
lpfx: *length pfx = n* **and**
lsfx: *length sfx = size-of TYPE*(*'a*) *− n − size-td s*
by (*metis Ex-list-of-length*)

with *sz* **have** *lxbs*: *length xbs = size-of TYPE*(*'a*)
by (*metis add-leD2 add-le-imp-le-diff lbs le-add-diff-inverse length-append*)
from *xbs lpfx lsfx xbs lbs* **have** *bs*: (*take* (*size-td s*) (*drop n xbs*)) *= bs*
by *simp*

obtain *pfx' sfx' xbs'* **where**
xbs': *xbs' = pfx' @ bs' @ sfx'* **and**
lpfx': *length pfx' = n* **and**
lsfx': *length sfx' = size-of TYPE*(*'a*) *− n − size-td s*
by (*metis Ex-list-of-length*)

with *sz* **have** *lxbs'*: *length xbs' = size-of TYPE*(*'a*)
by (*metis add-leD2 add-le-imp-le-diff lbs' le-add-diff-inverse length-append*)

from *xbs' lpfx' lsfx' xbs' lbs'* **have** *bs'*: (*take* (*size-td s*) (*drop n xbs'*)) *= bs'*
by *simp*

have *bound*: *x − n < size-td s* **by** *fact*

have *lacc*: *length* (*access-ti* (*typ-info-t TYPE*(*'a*)) *v xbs*) *= size-of TYPE*(*'a*)
by (*simp add*: *fd-cons-length lxbs size-of-def wf-fd-consD*)

have *v-upd-conv*: (*update-ti* (*typ-info-t TYPE*(*'a*)) (*to-bytes v xbs*) *v*) *= v*
by (*simp add*: *fu-fa lacc lxbs size-of-def to-bytes-def update-ti-update-ti-t*)

have *lxto*: *length* (*to-bytes v xbs*) *= size-of TYPE*(*'a*)
by (*simp add*: *lacc to-bytes-def*)

from *lxto lbs*
have *lsuper*: *length* (*super-update-bs bs* (*to-bytes v xbs*) *n*) *= size-of TYPE*(*'a*)
using *sz* **by** *auto*

```
  have bound: x − n < size-td s by fact

  have xbs-super: xbs = super-update-bs bs xbs n
    by (simp add: lpfx super-update-bs-def xbs)

 from mem-type-update-ti-super-update-bs [OF fl lxto lbs, of v, simplified v-upd-conv]
  have upd-eq1:
    update-ti s bs v =
       update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v xbs) n) v .

  have idx-shift1:
    take (size-td s) (drop n
         (access-ti (typ-info-t TYPE('a))
            (update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v xbs) n)
v) xbs'))
        ! (x − n)
       = access-ti (typ-info-t TYPE('a))
            (update-ti (typ-info-t TYPE('a)) (super-update-bs bs (to-bytes v xbs) n)
v) xbs'
        ! x
    by (metis bound fd-cons-length less-diff-conv2 local.wf-fd lower-bound-x lxbs'
        nth-take-drop-index-shift size-of-def sz wf-fd-consD)

  from lxto lbs lower-bound-x upper-bound-x sz
  have super-nth: super-update-bs bs (to-bytes v xbs) n ! x = bs ! (x − n)
    by (simp add: super-update-bs-nth-shift)
 note val-eq = padding-base.is-value-byte-acc-upd-cancel [OF is-value lsuper [simplified
size-of-def] lxbs' [simplified size-of-def]]
  note acc-eq = mem-type-field-lookup-access-ti-take-drop [OF fl lxbs', simplified
bs' ]
  show access-ti s (update-ti s bs v) bs' ! (x − n) = bs ! (x − n)
    apply (simp add: upd-eq1)
    apply (simp add: acc-eq)
    apply (simp add: idx-shift1 )
    apply (simp add: val-eq)
    apply (simp add: super-nth)
    done
next
  fix v::'a and bs::byte list and b::byte
  assume x − n < size-td s
  assume lbs: length bs = size-td s


  from fl have sz: size-td s + n ≤ size-of TYPE('a)
   by (metis (no-types, lifting) diff-add-zero diff-is-0-eq dual-order.trans field-lookup-offset-size'
size-of-def)

  obtain pfx sfx xbs where
```

```
    xbs: xbs = pfx @ bs @ sfx and
    lpfx: length pfx = n and
    lsfx: length sfx = size-of TYPE('a) − n − size-td s
    by (metis Ex-list-of-length)

  with sz have lxbs: length xbs = size-of TYPE('a)
    by (metis add-leD2 add-le-imp-le-diff lbs le-add-diff-inverse length-append)
  from xbs lpfx lsfx xbs lbs have bs: (take (size-td s) (drop n xbs)) = bs
    by simp

  obtain xbs' where
    xbs': xbs' = pfx @ bs [x − n := b] @ sfx by blast
  from lxbs lbs xbs have lxbs': length xbs' = size-of TYPE('a)
    using xbs' by auto
  from xbs' lpfx lxbs' lbs have bs': (take (size-td s) (drop n xbs')) = bs[x − n :=b]
    by simp

  from lbs lower-bound-x upper-bound-x sz lpfx lsfx
  have xbs'-conv: xbs' = xbs[x := b]
    by (simp add: xbs xbs' list-update-append)

  note eq1 = mem-type-field-lookup-access-ti-take-drop [OF fl lxbs, simplified bs, of
v]
  note eq2 = mem-type-field-lookup-access-ti-take-drop [OF fl lxbs', simplified bs',
of v]
  note val-eq = padding-base.is-value-byte-acc-eq [OF is-value lxbs [simplified size-of-def],
where b=b]
  show access-ti s v bs = access-ti s v (bs[x − n := b])
    apply (simp add: eq1)
    apply (simp add: val-eq)
    apply (simp add: eq2)
    apply (simp add: xbs'-conv)
    done
qed

lemma (in mem-type) field-lookup-is-value-byte-inner-to-outer:
  assumes fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)
  assumes lower-bound-x: n ≤ x
  assumes upper-bound-x: x < n + size-td s
  assumes is-value: padding-base.is-value-byte (access-ti s) (update-ti s) (size-td s)
(x − n)
  shows padding-base.is-value-byte (access-ti (typ-info-t (TYPE('a)))) (update-ti
(typ-info-t (TYPE('a)))) (size-td (typ-info-t (TYPE('a)))) x
proof (rule padding-base.is-value-byteI)
  from lower-bound-x upper-bound-x fl
  show x < size-td (typ-info-t TYPE('a))
    using field-lookup-offset-size' by fastforce
next
  fix v::'a and bs::byte list and bs'::byte list
```

assume *sz*: $x < size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a))$
assume *lbs*: $length\ bs = size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a))$
assume *lbs'*: $length\ bs' = size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a))$

from *fl* **have** *sz*: $size\text{-}td\ s + n \leq size\text{-}of\ TYPE('a)$
by (*metis* (*no-types*, *lifting*) *diff-add-zero diff-is-0-eq dual-order.trans field-lookup-offset-size' size-of-def*)

from *sz lbs*
have *super-bs*: $(super\text{-}update\text{-}bs\ (take\ (size\text{-}td\ s)\ (drop\ n\ bs))\ bs\ n) = bs$
by (*metis* (*no-types*, *lifting*) *append.assoc append-len2 append-take-drop-id diff-diff-cancel*
length-drop nat-move-sub-le size-of-def super-update-bs-def take-add)
from *lower-bound-x upper-bound-x lbs sz*
have *l-take-drop*: $length\ (take\ (size\text{-}td\ s)\ (drop\ n\ bs)) = size\text{-}td\ s$
by (*simp add*: *size-of-def*)

from *lower-bound-x upper-bound-x lbs' sz*
have *l-take-drop'*: $length\ (take\ (size\text{-}td\ s)\ (drop\ n\ bs')) = size\text{-}td\ s$
by (*simp add*: *size-of-def*)

from *lbs lower-bound-x upper-bound-x sz*
have *take-drop-eq*: $take\ (size\text{-}td\ s)\ (drop\ n\ bs)\ !\ (x - n) = bs\ !\ x$
by (*simp add*: *size-of-def*)

note *upd-focus* = *mem-type-update-ti-super-update-bs* [*OF fl*, *simplified size-of-def*, *OF lbs l-take-drop*, *simplified super-bs*, *symmetric*]
note *acc-focus* = *mem-type-field-lookup-nth-focus* [*OF fl lower-bound-x upper-bound-x*, *simplified size-of-def*, *OF lbs'*]
note *cancel* = *padding-base.is-value-byte-acc-upd-cancel* [*OF is-value l-take-drop l-take-drop'*]
show $access\text{-}ti\ (typ\text{-}info\text{-}t\ TYPE('a))\ (update\text{-}ti\ (typ\text{-}info\text{-}t\ TYPE('a))\ bs\ v)\ bs'\ !\ x =$
$bs\ !\ x$
apply (*subst upd-focus*)
apply (*simp add*: *acc-focus*)
apply (*simp add*: *cancel*)
apply (*simp add*: *take-drop-eq*)
done
next
fix *v*::*'a* **and** *bs*::*byte list* **and** *b*::*byte*
assume *sz*: $x < size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a))$
assume *lbs*: $length\ bs = size\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a))$

from *fl* **have** *sz*: $size\text{-}td\ s + n \leq size\text{-}of\ TYPE('a)$
by (*metis* (*no-types*, *lifting*) *diff-add-zero diff-is-0-eq dual-order.trans field-lookup-offset-size' size-of-def*)

note *eq1* = *mem-type-field-lookup-access-ti-take-drop* [*OF fl*, *simplified size-of-def*, *OF lbs*]

from *sz lower-bound-x upper-bound-x lbs*
have *l-take-s*: *length* (*take* (*size-td s*) (*drop n bs*)) = *size-td s*
by (*simp add*: *size-of-def*)

from *lbs sz lower-bound-x upper-bound-x*
have *super-eq*:
super-update-bs (*access-ti s v* (*take* (*size-td s*) (*drop n bs*)))
(*access-ti* (*typ-info-t TYPE*('*a*)) *v bs*) *n* = (*access-ti* (*typ-info-t TYPE*('*a*)) *v bs*)
by (*metis append-take-drop-id eq1 l-take-s length-drop length-fa-ti*
length-take local.wf-fd super-update-bs-def take-drop-append)

note *focus1* = *mem-type-field-lookup-nth-update-focus* [*OF fl lower-bound-x upper-bound-x*, *simplified size-of-def*, *OF lbs*]
note *pad-eq* = *padding-base.is-value-byte-acc-eq* [*OF is-value l-take-s*, *symmetric*]
show *access-ti* (*typ-info-t TYPE*('*a*)) *v bs* =
access-ti (*typ-info-t TYPE*('*a*)) *v* (*bs*[*x* := *b*])

apply (*simp add*: *focus1*)
apply (*subst pad-eq*)
apply (*simp add*: *super-eq*)
done
qed

lemma (**in** *mem-type*) *field-lookup-is-value-byte*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lower-bound-x*: *n* ≤ *x*
assumes *upper-bound-x*: *x* < *n* + *size-td s*
shows *padding-base.is-value-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) (*x* − *n*) ⟷
padding-base.is-value-byte (*access-ti* (*typ-info-t* (*TYPE*('*a*))))
(*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
using *field-lookup-is-value-byte-inner-to-outer* [*OF fl lower-bound-x upper-bound-x*]
field-lookup-is-value-byte-outer-to-inner [*OF fl lower-bound-x upper-bound-x*]
by *blast*

thm *padding-base.eq-padding-def*
thm *padding-base.eq-upto-padding-def*

lemma *td-set-component-descs-independent*:
fixes *t*::'*a xtyp-info*

and *st*::*'a xtyp-info-struct*
and *ts*::*'a xtyp-info-tuple list*
and *x*::*'a xtyp-info-tuple*
shows
(*s*, *n*) ∈ *td-set t m* ⟹ *component-descs-independent t* ⟹ *component-descs-independent s*
(*s*, *n*) ∈ *td-set-struct st m* ⟹ *component-descs-independent-struct st* ⟹ *component-descs-independent s*
(*s*, *n*) ∈ *td-set-list ts m* ⟹ *component-descs-independent-list ts* ⟹ *component-descs-independent s*
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *component-descs-independent-tuple x* ⟹ *component-descs-independent s*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *m* **and** *m* **and** *m* **and** *m*)
case (*TypDesc algn st nm*)
then show *?case* **by** *auto*
next
case (*TypScalar sz algn d*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** *auto*
qed

lemma *td-set-wf-component-descs*:
fixes *t*::*'a xtyp-info*
and *st*::*'a xtyp-info-struct*
and *ts*::*'a xtyp-info-tuple list*
and *x*::*'a xtyp-info-tuple*
shows
(*s*, *n*) ∈ *td-set t m* ⟹ *wf-component-descs t* ⟹ *wf-component-descs s*
(*s*, *n*) ∈ *td-set-struct st m* ⟹ *wf-component-descs-struct st* ⟹ *wf-component-descs s*
(*s*, *n*) ∈ *td-set-list ts m* ⟹ *wf-component-descs-list ts* ⟹ *wf-component-descs s*
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *wf-component-descs-tuple x* ⟹ *wf-component-descs s*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *m* **and** *m* **and** *m* **and** *m*)
case (*TypDesc algn st nm*)
then show *?case* **by** *auto*
next
case (*TypScalar sz algn d*)

then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** *auto*
qed

lemma *td-set-field-descs*:
fixes *t*::*'a xtyp-info*
and *st*::*'a xtyp-info-struct*
and *ts*::*'a xtyp-info-tuple list*
and *x*::*'a xtyp-info-tuple*
shows
(*s*, *n*) ∈ *td-set t m* ⟹ *wf-component-descs t* ⟹ *component-desc s* ∈ *insert* (*component-desc t*) (*set* (*field-descs t*))
(*s*, *n*) ∈ *td-set-struct st m* ⟹ *wf-component-descs-struct st* ⟹ *component-desc s* ∈ (*set* (*field-descs-struct st*))
(*s*, *n*) ∈ *td-set-list ts m* ⟹ *wf-component-descs-list ts* ⟹ *component-desc s* ∈ (*set* (*field-descs-list ts*))
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *wf-component-descs-tuple x* ⟹ *component-desc s* ∈ (*set* (*field-descs-tuple x*))
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *m* **and** *m* **and** *m* **and** *m*)
case (*TypDesc algn st nm*)
then show *?case* **by** *auto*
next
case (*TypScalar sz algn d*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** *auto*
qed

lemma *td-set-wf-field-descs*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
(*s*, *n*) ∈ *td-set t m* ⟹ *wf-field-descs* (*set* (*field-descs t*)) ⟹ *wf-field-descs* (*set* (*field-descs s*))
(*s*, *n*) ∈ *td-set-struct st m* ⟹*wf-field-descs* (*set* (*field-descs-struct st*)) ⟹ *wf-field-descs* (*set* (*field-descs s*))
(*s*, *n*) ∈ *td-set-list ts m* ⟹ *wf-field-descs* (*set* (*field-descs-list ts*)) ⟹ *wf-field-descs* (*set* (*field-descs s*))
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *wf-field-descs* (*set* (*field-descs-tuple x*)) ⟹ *wf-field-descs* (*set* (*field-descs s*))
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *m* **and** *m* **and** *m* **and** *m*)
case (*TypDesc algn st nm*)
then show *?case* **by** *auto*
next
case (*TypScalar sz algn d*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** *auto*
qed

context *xmem-type*
begin
lemma *xmem-type-td-set-field-descs*:
(*s*,*n*) ∈ *td-set* (*typ-info-t TYPE*('*a*)) *m* ⟹
component-desc s ∈ *insert* (*component-desc* (*typ-info-t TYPE*('*a*))) (*set* (*field-descs* (*typ-info-t TYPE*('*a*))))
using *td-set-field-descs local.wf-component-descs* **by** *blast*

lemma *field-lookup-component-desc*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
component-desc s ∈ *insert* (*component-desc* (*typ-info-t TYPE*('*a*))) (*set* (*field-descs* (*typ-info-t TYPE*('*a*))))
using *xmem-type-td-set-field-descs td-set-field-lookupD*
by *blast*

lemma *field-lookup-wf-field-desc*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
wf-field-desc (*component-desc s*)
using *field-lookup-component-desc*
by (*meson local.wf-field-descs*′ *wf-field-descs-def*)

lemma *field-lookup-component-descs-independent*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *component-descs-independent s*
using *field-lookup-component-desc* [*OF fl*]
by (*meson fl local.component-descs-independent td-set-component-descs-independent*(*1*)
td-set-field-lookupD)

lemma *field-lookup-wf-component-descs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *wf-component-descs s*
using *field-lookup-component-desc* [*OF fl*] *td-set-wf-component-descs fl*
local.wf-component-descs td-set-field-lookupD **by** *blast*

lemma *field-lookup-wf-field-descs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *wf-field-descs* (*set* (*field-descs s*))
using *td-set-wf-field-descs fl local.wf-field-descs td-set-field-lookupD* **by** *blast*

lemma *field-lookup-field-access-access-ti*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *field-access* (*component-desc s*) = *access-ti s*
using *access-ti-component-desc-compatible*(*1*) [*OF field-lookup-wf-component-descs*
[*OF fl*]
field-lookup-component-descs-independent [*OF fl*] *field-lookup-wf-field-descs*
[*OF fl*]]
by (*simp add*: *fun-eq-iff*)

lemma *field-lookup-field-update-update-ti*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *field-update* (*component-desc s*) = *update-ti s*
using *update-ti-component-desc-compatible*(*1*) [*OF field-lookup-wf-component-descs*
[*OF fl*]
field-lookup-component-descs-independent [*OF fl*] *field-lookup-wf-field-descs*
[*OF fl*]]
by (*simp add*: *fun-eq-iff*)

lemma *field-lookup-field-sz-size-td*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *field-sz* (*component-desc s*) = *size-td s*
using *size-td-field-sz*(*1*) [*OF field-lookup-wf-component-descs* [*OF fl*]]
by *simp*

lemma *field-lookup-component-desc-complement-padding*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
complement-padding (*field-access* (*component-desc s*)) (*field-update* (*component-desc s*)) (*field-sz* (*component-desc s*))
using *field-lookup-component-desc*
by (*meson field-lookup-wf-field-desc padding-lense.axioms*(*1*) *wf-field-desc.axioms*)

lemma *field-lookup-component-desc-complement-padding'*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *complement-padding* (*access-ti s*) (*update-ti s*) (*size-td s*)
using *field-lookup-component-desc-complement-padding*
field-lookup-field-access-access-ti [*OF fl*]
field-lookup-field-update-update-ti [*OF fl*]
field-lookup-field-sz-size-td [*OF fl*]
by (*metis fl*)

lemma *field-lookup-padding-lense*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
padding-lense (*access-ti s*) (*update-ti s*) (*size-td s*)
by (*metis field-lookup-field-access-access-ti field-lookup-field-update-update-ti*
field-lookup-wf-component-descs field-lookup-wf-field-desc size-td-field-sz(*1*) *wf-field-desc-def*)

sublocale *lense*: *padding-lense*
access-ti (*typ-info-t TYPE*('*a*))
update-ti (*typ-info-t TYPE*('*a*))
size-of TYPE('*a*)
using *local.field-access-access-ti local.field-sz-size-of*
local.field-update-update-ti local.xmem-type-wf-field-desc.padding-lense-axioms
by *simp*

end

lemma *eq-padding-access-update-field-cancel*:
assumes *fl*: *field-lookup* (*typ-info-t* (*TYPE*('*a*::*xmem-type*))) *f 0* = *Some* (*s*, *n*)
assumes *lower-bound-x*: *n* ≤ *x*
assumes *upper-bound-x*: *x* < *n* + *size-of TYPE*('*b*)
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*::*xmem-type*)
assumes *lbs*: *length bs* = *size-of TYPE*('*b*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *eq-padding*: *padding-base.eq-padding* (*access-ti s*) (*size-td s*) *bs* (*take* (*size-of TYPE*('*b*)) (*drop n bs'*))
shows *access-ti* (*typ-info-t TYPE*('*a*::*xmem-type*)) (*update-ti s bs v*) *bs'* ! *x* = *bs* ! (*x* − *n*)
proof −
obtain *i* **where** *i*: *i* = *x* − *n* **by** *blast*

from *match fl* **have** *sz-b*: *size-of TYPE('b)* = *size-td s*
using *export-size-of* **by** *blast*

interpret *cps*: *complement-padding access-ti s update-ti s size-td s*
by (*rule field-lookup-component-desc-complement-padding'* [*OF fl*])

from *lower-bound-x upper-bound-x sz-b*
have *x-n-bound*: *x* − *n* < *size-td s*
by *simp*
from *eq-padding*
have *l-take-drop-bs'*: *length* (*take* (*size-td s*) (*drop n bs'*)) = *size-td s*
by (*metis lbs padding-base.eq-padding-length-eq sz-b*)

from *lower-bound-x upper-bound-x fl lbs'*
have *acc-bs'*: *take* (*size-td s*) (*drop n bs'*) ! (*x* − *n*) = *bs'* ! *x*
by (*metis add.commute field-lookup-offset-size nth-take-drop-index-shift size-of-def sz-b*)

from *mem-type-field-lookup-nth-focus* [*OF fl lower-bound-x* [*simplified size-of-def*] *upper-bound-x* [*simplified sz-b*] *lbs'*]
have *access-ti* (*typ-info-t TYPE('a)*) (*update-ti s bs v*) *bs'* ! *x* =
access-ti s (*update-ti s bs v*) (*take* (*size-td s*) (*drop n bs'*)) ! (*x* − *n*) .
also have ... = *bs* ! (*x* − *n*) (**is** *?lhs* = *?rhs*)
proof (*cases padding-base.is-padding-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) (*x* − *n*))
case *True*

from *padding-base.is-padding-byte-acc-eq* [*OF True l-take-drop-bs'*] *acc-bs'*
have *?lhs* = *bs'* ! *x* **by** *simp*
also have *bs'* ! *x* = *bs* ! (*x* − *n*)
using *cps.eq-padding-padding-byte-id* [*OF eq-padding True*] *acc-bs' sz-b*
by *simp*
finally show *?thesis* .
next
case *False*
then have *is-value*: *cps.is-value-byte* (*x* − *n*) **using** *cps.complement x-n-bound*
by *simp*
from *cps.is-value-byte-acc-upd-cancel* [*OF is-value lbs* [*simplified sz-b*] *l-take-drop-bs'*]
show *?thesis* .
qed
finally show *?thesis* .
qed

context *xmem-type*
begin

sublocale *xmem-type-padding*: *complement-padding*
access-ti (*typ-info-t TYPE('a)*)
update-ti (*typ-info-t TYPE('a)*)

size-of TYPE('a)
using *xmem-type-wf-field-desc.complement-padding-axioms*
by (*simp add*: *field-access-access-ti field-update-update-ti field-sz-size-td size-of-def*)
end

lemma *drop-heap-list-le2*:
heap-list h n (x + of-nat k)
= drop k (heap-list h (n + k) x)
by (*simp add*: *drop-heap-list-le*)

lemma *heap-list-take-drop*:
assumes *N-bound*: *unat a + N ≤ 2 ^ len-of TYPE(addr-bitsize)*
shows *n + m ≤ N ⟹ take m (drop n (heap-list hp N a)) =*
heap-list hp m (a + word-of-nat n)
apply (*induct m arbitrary*: *n*)
apply *simp*
apply *simp*
using *N-bound*
by *auto*
(*metis* (*no-types*, *opaque-lifting*) *add.left-commute add.right-neutral add-Suc-right drop-heap-list-le2 heap-list-rec take-drop take-heap-list-le*)

lemma *heap-list-take-drop'*:
assumes *N-bound*: *unat a + N ≤ addr-card*
assumes *bound*: *n + m ≤ N*
shows *take m (drop n (heap-list hp N a)) =*
heap-list hp m (a + word-of-nat n)
proof −
have *addr-card = 2 ^ len-of TYPE(addr-bitsize)*
by (*simp add*: *addr-card*)
from *heap-list-take-drop* [*OF N-bound* [*simplified this*] *bound*] **show** *?thesis* **by** *blast*
qed

experiment
fixes *proj*:: *'a*::*xmem-type ⇒ 'b*::*xmem-type*
fixes *t*::*'a xtyp-info*

assumes *fl*: *field-lookup* (*typ-info-t TYPE('a)*) *f 0 = Some* (*t*, *n*)
and *eu*: *export-uinfo t = typ-uinfo-t TYPE('b)*
assumes *access-comp*: *access-ti t v = access-ti* (*typ-info-t TYPE('b)*) (*proj v*)
assumes *surj*: *surj proj*
begin

lemma *sz*: *size-td t = size-of TYPE('b)*
using *eu*
by (*metis export-uinfo-size size-of-def typ-uinfo-size*)

lemma *padding-base.eq-padding* (*access-ti t*) (*size-td t*) =

padding-base.eq-padding (*access-ti* (*typ-info-t TYPE*('*b*))) (*size-of TYPE*('*b*))
unfolding *padding-base.eq-padding-def*
apply (*rule ext*)
apply (*rule ext*)
apply (*simp add*: *sz*)
apply (*simp add*: *access-comp*)
using *surj*
by (*metis surj-def*)
end

definition *is-padding-byte*::*typ-uinfo* ⇒ *nat* ⇒ *bool* **where**
is-padding-byte t i ≡ *i* < *size-td t* ∧
(∀ *bs b*. *length bs* = *size-td t* ⟶ *norm-tu t* (*bs*[*i* := *b*]) = *norm-tu t bs*)

definition *is-value-byte*::*typ-uinfo* ⇒ *nat* ⇒ *bool* **where**
is-value-byte t i ≡ *i* < *size-td t* ∧
(∃ *bs b*. *length bs* = *size-td t* ∧ *norm-tu t* (*bs*[*i* := *b*]) ≠ *norm-tu t bs*)

lemma *is-padding-byteI*:
assumes *i* < *size-td t*
assumes ⋀*i bs b*. *length bs* = *size-td t* ⟹ *norm-tu t* (*bs*[*i* := *b*]) = *norm-tu t bs*
shows *is-padding-byte t i*
using *assms* **by** (*simp add*: *is-padding-byte-def*)

lemma *complement-tu-padding*:
i < *size-td t* ⟹ *is-padding-byte t i* ≠ *is-value-byte t i*
by (*auto simp add*: *is-padding-byte-def is-value-byte-def*)

definition *eq-padding*::*typ-uinfo* ⇒ *byte list* ⇒ *byte list* ⇒ *bool* **where**
eq-padding t bs bs' ≡ *length bs* = *size-td t* ∧ *length bs'* = *size-td t* ∧
(∀ *i*. *is-padding-byte t i* ⟶ *bs* ! *i* = *bs'* ! *i*)

definition *eq-upto-padding*::*typ-uinfo* ⇒ *byte list* ⇒ *byte list* ⇒ *bool* **where**
eq-upto-padding t bs bs' ≡ *length bs* = *size-td t* ∧ *length bs'* = *size-td t* ∧
(∀ *i*. *is-value-byte t i* ⟶ *bs* ! *i* = *bs'* ! *i*)

lemma *eq-padding-refl*[*simp*]: *length bs* = *size-td t* ⟹ *eq-padding t bs bs*
by (*auto simp add*: *eq-padding-def*)

lemma *eq-upto-padding-refl*[*simp*]: *length bs* = *size-td t* ⟹ *eq-upto-padding t bs bs*
by (*auto simp add*: *eq-upto-padding-def*)

lemma *eq-padding-sym*: *eq-padding t bs bs'* ⟷ *eq-padding t bs' bs*
by (*auto simp add*: *eq-padding-def*)

lemma *eq-padding-symp*: *symp* (*eq-padding t*)

by (*simp add*: *symp-def eq-padding-sym*)

lemma *eq-upto-padding-sym*: *eq-upto-padding t bs bs'* $\longleftrightarrow$ *eq-upto-padding t bs' bs*
by (*auto simp add*: *eq-upto-padding-def*)

lemma *eq-upto-padding-symp*: *symp* (*eq-upto-padding t*)
by (*simp add*: *symp-def eq-upto-padding-sym*)

lemma *eq-padding-trans*: *eq-padding t bs1 bs2* $\Longrightarrow$ *eq-padding t bs2 bs3* $\Longrightarrow$ *eq-padding t bs1 bs3*
by (*auto simp add*: *eq-padding-def*)

lemma *eq-padding-transp*: *transp* (*eq-padding t*)
unfolding *transp-def*
by (*auto intro*: *eq-padding-trans*)

lemma *eq-upto-padding-trans*: *eq-upto-padding t bs1 bs2* $\Longrightarrow$ *eq-upto-padding t bs2 bs3* $\Longrightarrow$ *eq-upto-padding t bs1 bs3*
by (*auto simp add*: *eq-upto-padding-def*)

lemma *eq-upto-padding-transp*: *transp* (*eq-upto-padding t*)
unfolding *transp-def*
by (*auto intro*: *eq-upto-padding-trans*)

lemma *eq-padding-eq-upto-padding-eq*: *eq-padding t bs bs'* $\Longrightarrow$ *eq-upto-padding t bs bs'* $\Longrightarrow$ *bs* = *bs'*
proof −
assume *a1*: *eq-padding t bs bs'*
assume *a2*: *eq-upto-padding t bs bs'*
have *f3*: *length bs'* = *size-td t*
using *a1* **by** (*simp add*: *eq-padding-def*)
have *length bs* = *size-td t*
using *a1 eq-padding-def* **by** *blast*
then show *?thesis*
using *f3 a2 a1* **by** (*metis* (*full-types*) *complement-tu-padding eq-padding-def eq-upto-padding-def nth-equalityI*)
qed

thm *padding-base.is-padding-byte-def*

lemma *is-padding-byte-access-ti'*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows

⟦ *wf-desc t*; *wf-size-desc t*; *wf-field-descs* (*set* (*field-descs t*)); *wf-component-descs t*; *wf-fd t*;
i < *size-td t*; *length bs* = *size-td t*;
∀ *bs b*. *length bs* = *size-td t* ⟶ *norm-tu* (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) = *norm-tu* (*export-uinfo t*) *bs*⟧
⟹ *access-ti t v bs* ! *i* = *bs* ! *i*

⟦ *wf-desc-struct st*; *wf-size-desc-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));
wf-component-descs-struct st; *wf-fd-struct st*; *i* < *size-td-struct st*; *length bs* = *size-td-struct st*;
∀ *bs b*. *length bs* = *size-td-struct st* ⟶ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) = *norm-tu-struct* ((*map-td-struct field-norm* (λ-. ()) *st*)) *bs*⟧
⟹ *access-ti-struct st v bs* ! *i* = *bs* ! *i*

⟦ *wf-desc-list ts*; *wf-size-desc-list ts*; *wf-field-descs* (*set* (*field-descs-list ts*));
wf-component-descs-list ts; *wf-fd-list ts*; *i* < *size-td-list ts*; *length bs* = *size-td-list ts*;
∀ *bs b*. *length bs* = *size-td-list ts* ⟶ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) = *norm-tu-list* ((*map-td-list field-norm* (λ-. ()) *ts*)) *bs*⟧
⟹ *access-ti-list ts v bs* ! *i* = *bs* ! *i*

⟦ *wf-desc-tuple x*; *wf-size-desc-tuple x*; *wf-field-descs* (*set* (*field-descs-tuple x*));
wf-component-descs-tuple x; *wf-fd-tuple x*; *i* < *size-td-tuple x*; *length bs* = *size-td-tuple x*;
∀ *bs b*. *length bs* = *size-td-tuple x* ⟶ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) = *norm-tu-tuple* ((*map-td-tuple field-norm* (λ-. ()) *x*)) *bs*⟧
⟹ *access-ti-tuple x v bs* ! *i* = *bs* ! *i*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *bs i* **and** *bs i* **and** *bs i* **and** *bs i*)
case (*TypDesc algn st nm*)
then show *?case* **by** *auto*
next
case (*TypScalar sz algn d*)
from *TypScalar* **obtain**
wf-d: *wf-field-desc d* **and**
field-sz: *field-sz d* =*sz* **and**
i-bound: *i* < *sz* **and**
lbs: *length bs* = *sz* **and**
padding: ⋀*bs b*. *length bs* = *sz* ⟹
field-norm sz algn d (*bs*[*i* := *b*]) = *field-norm sz algn d bs*
by *simp*

interpret *wf*: *wf-field-desc d* **by** (*rule wf-d*)
show *?case*
proof (*cases wf.is-padding-byte i*)
case *True*
then show *?thesis*
using *field-sz lbs wf.is-padding-byte-acc-id* **by** *auto*
next

```
    case False
    note not-padding = this
    with wf.complement field-sz i-bound have is-value: wf.is-value-byte i by blast
    from is-value padding have False
      by (auto simp add: field-norm-def)
        (metis (mono-tags, opaque-lifting) field-sz i-bound list-update-id not-padding
padding-base.is-value-byte-upd-neq
       wf.access-result-size wf.is-padding-byte-def wf.update-access wf.update-access-unequal)
    then show ?thesis by blast
  qed
next
case (TypAggregate ts)
then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto
  from Cons-typ-desc.prems obtain
    wf-desc-d: wf-desc d and
    wf-fd-x: wf-fd-tuple (DTuple d nm y) and
    wf-fd-d: wf-fd d and
    wf-fd-fs: wf-fd-list fs and
    nm-notin: nm ∉ dt-snd ‘ set fs and
    wf-desc-fs: wf-desc-list fs and
    wf-sz-d: wf-size-desc d and
    wf-sz-fs: wf-size-desc-list fs and
    wf-field-d: wf-field-desc (component-desc d) and
    wf-fields-d: wf-field-descs (set (field-descs d)) and
    wf-fields-fs: wf-field-descs (set (field-descs-list fs)) and
    y: y = component-desc d and
    wf-comp-d: wf-component-descs d and
    wf-comp-fs: wf-component-descs-list fs and
    i-bound:i < size-td d + size-td-list fs and
    lbs: length bs = size-td d + size-td-list fs and
    wf-desc-x: wf-desc-tuple x and
    wf-sz-x: wf-size-desc-tuple x and
    wf-decs-x: wf-field-descs (set (field-descs-tuple x)) and
    wf-comp-x: wf-component-descs-tuple x and

    padding: ⋀bs b . length bs = size-td d + size-td-list fs ⟹
          norm-tu (map-td field-norm (λ-. ()) d)
              (take (size-td d) (bs[i := b])) @
             norm-tu-list (map-td-list field-norm (λ-. ()) fs)
              (drop (size-td d) (bs[i := b])) =
             norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d) bs) @
             norm-tu-list (map-td-list field-norm (λ-. ()) fs)
              (drop (size-td d) bs)
```

by (*auto simp add*: *x*)

have *padding-d*: ∀ *bs*. *length bs* = *size-td d* ⟶
(∀ *b*. *norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*bs*[*i* := *b*]) =
norm-tu (*map-td field-norm* (λ-. ()) *d*) *bs*)
proof (*safe*)
fix *bs*::*byte list* **and** *b*::*byte*
assume *lbs*: *length bs* = *size-td d*
from *lbs* **obtain** *xbs sfx* **where** *xbs*: *xbs* = *bs* @ *sfx* **and** *lxbs*: *length xbs* = *size-td d* + *size-td-list fs*
by (*metis Ex-list-of-length length-append*)

from *lbs*
have *eq1*: (*take* (*size-td d*) ((*bs* @ *sfx*)[*i* := *b*])) = *bs*[*i* := *b*] **by** (*simp add*: *list-update-append*)

from *lbs*
have *lbs'*: *length* (*bs*[*i*:=*b*]) = *size-td d* **by** *simp*

from *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d lbs*] *lbs*
have *lnorm1*:*length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) *bs*) = *size-td d*
by (*simp add*: *export-uinfo-def length-fa-ti wf-fd-d*)

from *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d lbs'*] *lbs'*
have *lnorm2*: *length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*bs*[*i*:=*b*])) = *size-td d*
by (*simp add*: *export-uinfo-def length-fa-ti wf-fd-d*)

from *padding* [*OF lxbs*, *of b*] *xbs lbs*
show *norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*bs*[*i* := *b*]) =
norm-tu (*map-td field-norm* (λ-. ()) *d*) *bs*
apply *simp*
apply (*subst* (*asm*) *list-append-eq-split-conv*)
apply (*simp add*: *eq1 lnorm1 lnorm2*)
apply (*simp add*: *eq1*)
done
qed

from *lbs*
have *lbs-drop*: *length* (*drop* (*size-td d*) *bs*) = *size-td-list fs*
by (*simp add*: *x*)

from *lbs*
have *lbs-take*: *length* (*take* (*size-td d*) *bs*) = *size-td d*
by (*simp add*: *x*)

have *lacc*: *length* (*access-ti d v* (*take* (*size-td d*) *bs*)) = *size-td d*
by (*metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps wf-fd-x x*)

show *?case*

proof (*cases i < size-td d*)
case *True*
note *hyp* = *Cons-typ-desc.hyps*(*1*) [*OF wf-desc-x wf-sz-x wf-decs-x wf-comp-x*, *simplified x*, *OF wf-fd-x*, *simplified*, *OF True lbs-take padding-d*]
from *hyp lbs*
show *?thesis*
by (*simp add*: *x True nth-append lacc*)
next
case *False*
from *False i-bound* **have** *i-bound'*: *i − size-td d* < *size-td-list fs* **by** *simp*
have *padding-fs*: ∀ *bs b*.
length bs = *size-td-list fs* ⟶
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*) (*bs*[*i − size-td d* := *b*]) =
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*) *bs*
proof (*safe*)
fix *bs*::*byte list* **and** *b*::*byte*
assume *lbs*: *length bs* = *size-td-list fs*
from *lbs* **obtain** *xbs pfx* **where** *xbs*: *xbs* = *pfx* @ *bs* **and** *lxbs*: *length xbs* = *size-td d* + *size-td-list fs*
and *lpfx*: *length pfx* = *size-td d*
by (*metis Ex-list-of-length length-append*)

from *lpfx*
have *drop-eq1*: *drop* (*size-td d*) *xbs* = *bs*
by (*simp add*: *xbs*)
from *lpfx False* **have** *drop-eq2*: (*drop* (*size-td d*) (*xbs*[*i* := *b*])) = *bs* [*i − size-td d* := *b*]
by (*auto simp add*: *xbs list-update-append*)
from *lpfx False*
have *take-eq1*: *take* (*size-td d*) (*xbs*[*i* := *b*]) = *pfx*
by (*simp add*: *xbs list-update-append*)
from *lpfx False*
have *take-eq2*: *take* (*size-td d*) *xbs* = *pfx*
by (*simp add*: *xbs*)

have *leq1*: *length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) (*xbs*[*i* := *b*]))) = *size-td d*
apply (*simp add*: *take-eq1*)
using *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d lpfx*] *lpfx*
by (*simp add*: *export-uinfo-def length-fa-ti wf-fd-d*)
have *leq2*: *length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) *xbs*)) = *size-td d*
apply (*simp add*: *take-eq2*)
using *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d lpfx*] *lpfx*
by (*simp add*: *export-uinfo-def length-fa-ti wf-fd-d*)

```
      from padding [OF lxbs, of b, simplified drop-eq1 drop-eq2]
      show norm-tu-list (map-td-list field-norm (λ-. ()) fs) (bs[i − size-td d := b])
=
      norm-tu-list (map-td-list field-norm (λ-. ()) fs) bs
        apply −
        apply (subst (asm) list-append-eq-split-conv)
        apply (simp add: leq1 leq2)
        apply simp
        done
    qed
    from False lbs have drop-eq: drop (size-td d) bs ! (i − size-td d) = bs ! i by
simp
   note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs wf-sz-fs wf-fields-fs wf-comp-fs
wf-fd-fs i-bound′ lbs-drop padding-fs]
    from hyp
    show ?thesis by (simp add: x False nth-append lacc drop-eq)
  qed
next
  case (DTuple-typ-desc d nm y)
  then show ?case by (auto simp add: export-uinfo-def)
qed

lemma is-padding-byte-access-ti:
  assumes wf: wf-desc (t::'a xtyp-info)
  and wf-sz: wf-size-desc t
  and wf-descs: wf-field-descs (set (field-descs t))
  and wf-comp: wf-component-descs t
  and wf-fd: wf-fd t
  and i-bound: i < size-td t
  and lbs: length bs = size-td t
  and is-padding: is-padding-byte (export-uinfo t) i
shows access-ti t v bs ! i = bs ! i
  using is-padding-byte-access-ti′(1) [OF wf wf-sz wf-descs wf-comp wf-fd i-bound
lbs] is-padding
  by (simp add: is-padding-byte-def export-uinfo-def)

context xmem-type
begin
lemma xmem-type-is-padding-byte-access-ti:
  assumes padding: is-padding-byte (typ-uinfo-t TYPE('a)) i
  and i-bound: i < size-of TYPE('a)
  and lbs: length bs = size-of TYPE('a)
shows access-ti (typ-info-t TYPE('a)) v bs ! i = bs ! i
proof −
  have wf: wf-desc (typ-info-t TYPE('a)) by simp
  have wf-sz: wf-size-desc (typ-info-t TYPE('a)) by simp
  have wf-descs: wf-field-descs (set (field-descs (typ-info-t TYPE('a))))
    using local.wf-field-descs by blast
  have wf-comp: wf-component-descs (typ-info-t TYPE('a))
```

by (*simp add*: *local.wf-component-descs*)
have *wf-fd*: *wf-fd* (*typ-info-t TYPE*('*a*))
using *local.wf-desc local.wf-lf wf-fdp-fd*(*1*) *wf-lf-fdp* **by** *blast*
from *is-padding-byte-access-ti* [*OF wf wf-sz wf-descs wf-comp wf-fd*
i-bound [*simplified size-of-def*] *lbs* [*simplified size-of-def*]
padding [*simplified typ-uinfo-t-def*]]
show *?thesis* **.**
qed

end

lemma *is-padding-byte-update-ti-id'*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
⟦ *wf-desc t*; *wf-size-desc t*; *wf-field-descs* (*set* (*field-descs t*)); *wf-component-descs t*; *wf-fd t*;
i < *size-td t*; *length bs* = *size-td t*;
∀ *bs b*. *length bs* = *size-td t* ⟶ *norm-tu* (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) = *norm-tu* (*export-uinfo t*) *bs*⟧
⟹ *update-ti t* (*bs*[*i* := *b*]) *v* = *update-ti t bs v*

⟦ *wf-desc-struct st*; *wf-size-desc-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));
wf-component-descs-struct st; *wf-fd-struct st*; *i* < *size-td-struct st*; *length bs* = *size-td-struct st*;
∀ *bs b*. *length bs* = *size-td-struct st* ⟶ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) = *norm-tu-struct* ((*map-td-struct field-norm* (λ-. ()) *st*)) *bs*⟧
⟹ *update-ti-struct st* (*bs*[*i* := *b*]) *v* = *update-ti-struct st bs v*

⟦ *wf-desc-list ts*; *wf-size-desc-list ts*; *wf-field-descs* (*set* (*field-descs-list ts*));
wf-component-descs-list ts; *wf-fd-list ts*; *i* < *size-td-list ts*; *length bs* = *size-td-list ts*;
∀ *bs b*. *length bs* = *size-td-list ts* ⟶ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) = *norm-tu-list* ((*map-td-list field-norm* (λ-. ()) *ts*)) *bs*⟧
⟹ *update-ti-list ts* (*bs*[*i* := *b*]) *v* = *update-ti-list ts bs v*

⟦ *wf-desc-tuple x*; *wf-size-desc-tuple x*; *wf-field-descs* (*set* (*field-descs-tuple x*));
wf-component-descs-tuple x; *wf-fd-tuple x*; *i* < *size-td-tuple x*; *length bs* = *size-td-tuple x*;
∀ *bs b*. *length bs* = *size-td-tuple x* ⟶ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) = *norm-tu-tuple* ((*map-td-tuple field-norm* (λ-. ()) *x*)) *bs*⟧
⟹ *update-ti-tuple x* (*bs*[*i* := *b*]) *v* = *update-ti-tuple x bs v*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *bs i v* **and** *bs i v* **and** *bs i v* **and**

bs i v)
case (*TypDesc algn st nm*)
then show *?case* **by** *auto*
next
case (*TypScalar sz algn d*)
from *TypScalar* **obtain**
wf-d: *wf-field-desc d* **and**
field-sz: *field-sz d* = *sz* **and**
i-bound: *i* < *sz* **and**
lbs: *length bs* = *sz* **and**
padding: ⋀*bs b*. *length bs* = *sz* ⟹
field-norm sz algn d (*bs*[*i* := *b*]) = *field-norm sz algn d bs*
by *simp*

interpret *wf*: *wf-field-desc d* **by** (*rule wf-d*)
show *?case*
proof (*cases wf.is-padding-byte i*)
case *True*
then show *?thesis*
using *field-sz lbs wf.is-padding-byte-upd-eq* **by** *auto*
next
case *False*
note *not-padding* = *this*
with *wf.complement field-sz i-bound* **have** *is-value*: *wf.is-value-byte i* **by** *blast*
from *is-value padding* **have** *False*
by (*auto simp add*: *field-norm-def*)
(*metis* (*mono-tags*, *opaque-lifting*) *field-sz i-bound list-update-id not-padding*
padding-base.is-value-byte-upd-neq
wf.access-result-size wf.is-padding-byte-def wf.update-access wf.update-access-unequal)
then show *?thesis* **by** *blast*
qed
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: *x* = *DTuple d nm y* **by** (*cases x*) *auto*
from *Cons-typ-desc.prems* **obtain**
wf-desc-d: *wf-desc d* **and**
wf-fd-x: *wf-fd-tuple* (*DTuple d nm y*) **and**
wf-fd-d: *wf-fd d* **and**
wf-fd-fs: *wf-fd-list fs* **and**
nm-notin: *nm* ∉ *dt-snd* ‘ *set fs* **and**
wf-desc-fs: *wf-desc-list fs* **and**
wf-sz-d: *wf-size-desc d* **and**
wf-sz-fs: *wf-size-desc-list fs* **and**

wf-field-d: *wf-field-desc* (*component-desc d*) **and**
wf-fields-d: *wf-field-descs* (*set* (*field-descs d*)) **and**
wf-fields-fs: *wf-field-descs* (*set* (*field-descs-list fs*)) **and**
y: *y* = *component-desc d* **and**
wf-comp-d: *wf-component-descs d* **and**
wf-comp-fs: *wf-component-descs-list fs* **and**
i-bound:*i* < *size-td d* + *size-td-list fs* **and**
lbs: *length bs* = *size-td d* + *size-td-list fs* **and**
wf-desc-x: *wf-desc-tuple x* **and**
wf-sz-x: *wf-size-desc-tuple x* **and**
wf-decs-x: *wf-field-descs* (*set* (*field-descs-tuple x*)) **and**
wf-comp-x: *wf-component-descs-tuple x* **and**

padding: ⋀*bs b* . *length bs* = *size-td d* + *size-td-list fs* ⟹
norm-tu (*map-td field-norm* (λ-. ()) *d*)
(*take* (*size-td d*) (*bs*[*i* := *b*])) @
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*)
(*drop* (*size-td d*) (*bs*[*i* := *b*])) =
norm-tu (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) *bs*) @
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*)
(*drop* (*size-td d*) *bs*)
by (*auto simp add*: *x*)

have *padding-d*: ∀ *bs*. *length bs* = *size-td d* ⟶
(∀ *b*. *norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*bs*[*i* := *b*]) =
norm-tu (*map-td field-norm* (λ-. ()) *d*) *bs*)
proof (*safe*)
fix *bs*::*byte list* **and** *b*::*byte*
assume *lbs*: *length bs* = *size-td d*
from *lbs* **obtain** *xbs sfx* **where** *xbs*: *xbs* = *bs* @ *sfx* **and** *lxbs*: *length xbs* = *size-td d* + *size-td-list fs*
by (*metis Ex-list-of-length length-append*)

from *lbs*
have *eq1*: (*take* (*size-td d*) ((*bs* @ *sfx*)[*i* := *b*])) = *bs*[*i* := *b*] **by** (*simp add*: *list-update-append*)

from *lbs*
have *lbs'*: *length* (*bs*[*i*:=*b*]) = *size-td d* **by** *simp*

from *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d lbs*] *lbs*
have *lnorm1*:*length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) *bs*) = *size-td d*
by (*simp add*: *export-uinfo-def length-fa-ti wf-fd-d*)

from *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d lbs'*] *lbs'*
have *lnorm2*: *length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*bs*[*i*:=*b*])) = *size-td d*
by (*simp add*: *export-uinfo-def length-fa-ti wf-fd-d*)

```
    from padding [OF lxbs, of b] xbs lbs
    show norm-tu (map-td field-norm (λ-. ()) d) (bs[i := b]) =
       norm-tu (map-td field-norm (λ-. ()) d) bs
      apply simp
      apply (subst (asm) list-append-eq-split-conv)
       apply (simp add: eq1 lnorm1 lnorm2)
      apply (simp add: eq1)
      done
  qed


  from lbs
  have lbs-drop: length (drop (size-td d) bs) = size-td-list fs
    by (simp add: x)

  from lbs
  have lbs-take: length (take (size-td d) bs) = size-td d
    by (simp add: x)

  have lacc: length (access-ti d v (take (size-td d) bs)) = size-td d
    by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps wf-fd-x x)

  show ?case

  proof (cases i < size-td d)
    case True
    from True lbs
    have take-eq: (take (size-td d) bs)[i := b] = take (size-td d) (bs[i := b])
      by (simp add: take-update-swap)
    note hyp = Cons-typ-desc.hyps(1) [OF wf-desc-x wf-sz-x wf-decs-x wf-comp-x,
simplified x, OF wf-fd-x, simplified, OF True lbs-take padding-d]
    from hyp lbs
    show ?thesis
      by  (simp add: x True nth-append lacc take-eq)
  next
    case False
    from False i-bound have i-bound': i − size-td d  < size-td-list fs by simp
    have padding-fs: ∀ bs b.
      length bs = size-td-list fs ⟶
      norm-tu-list (map-td-list field-norm (λ-. ()) fs) (bs[i − size-td d := b]) =
      norm-tu-list (map-td-list field-norm (λ-. ()) fs) bs
    proof (safe)
      fix bs::byte list and b::byte
      assume lbs: length bs = size-td-list fs
      from lbs obtain xbs pfx where xbs: xbs = pfx @ bs and lxbs: length xbs =
size-td d + size-td-list fs
       and lpfx: length pfx = size-td d
       by (metis Ex-list-of-length length-append)
```

```
      from lpfx
      have drop-eq1: drop (size-td d) xbs = bs
        by (simp add: xbs)
       from lpfx False have drop-eq2: (drop (size-td d) (xbs[i := b])) = bs [i −
size-td d := b]
        by (auto simp add: xbs list-update-append)
      from lpfx False
      have take-eq1: take (size-td d) (xbs[i := b]) = pfx
        by (simp add: xbs list-update-append)
      from lpfx False
      have take-eq2: take (size-td d) xbs = pfx
        by (simp add: xbs)

       have leq1: length (norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d)
(xbs[i := b]))) = size-td d
        apply (simp add: take-eq1)
        using wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lpfx ] lpfx
        by (simp add: export-uinfo-def length-fa-ti wf-fd-d)
       have leq2: length (norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d)
xbs)) = size-td d
        apply (simp add: take-eq2)
        using wf-fd-norm-tu(1)[rule-format, OF wf-fd-d lpfx ] lpfx
        by (simp add: export-uinfo-def length-fa-ti wf-fd-d)

      from padding [OF lxbs, of b, simplified drop-eq1 drop-eq2]
      show norm-tu-list (map-td-list field-norm (λ-. ()) fs) (bs[i − size-td d := b])
=
      norm-tu-list (map-td-list field-norm (λ-. ()) fs) bs
        apply −
        apply (subst (asm) list-append-eq-split-conv)
        apply (simp add: leq1 leq2)
        apply simp
        done
    qed
    from False lbs have drop-eq: ((drop (size-td d) bs)[i − size-td d := b]) =
          (drop (size-td d) (bs[i := b]))
      by (simp add: drop-update-swap)
    from False lbs have take-eq: take (size-td d) (bs[i := b]) = take (size-td d) bs
by simp
   note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs wf-sz-fs wf-fields-fs wf-comp-fs
wf-fd-fs i-bound' lbs-drop padding-fs]
    from hyp
    show ?thesis by (simp add: x False nth-append lacc drop-eq take-eq)
  qed
next
  case (DTuple-typ-desc d nm y)
  then show ?case by (auto simp add: export-uinfo-def)
qed
```

lemma *is-padding-byte-update-ti-id*:
  assumes *wf*: *wf-desc* (*t*::'*a xtyp-info*)
  and *wf-sz*: *wf-size-desc t*
  and *wf-descs*: *wf-field-descs* (*set* (*field-descs t*))
  and *wf-comp*: *wf-component-descs t*
  and *wf-fd*: *wf-fd t*
  and *i-bound*: *i* < *size-td t*
  and *lbs*: *length bs* = *size-td t*
  and *is-padding*: *is-padding-byte* (*export-uinfo t*) *i*
shows *update-ti t* (*bs*[*i* := *b*]) *v* = *update-ti t bs v*
 using *is-padding-byte-update-ti-id'*(*1*) [*OF wf wf-sz wf-descs wf-comp wf-fd i-bound lbs*] *is-padding*
  by (*simp add*: *is-padding-byte-def export-uinfo-def*)

context *xmem-type*
begin
lemma *xmem-type-is-padding-byte-update-ti-id*:
  assumes *padding*: *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
  and *i-bound*: *i* < *size-of TYPE*('*a*)
  and *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *update-ti* (*typ-info-t TYPE*('*a*)) (*bs*[*i*:=*b*]) *v* = *update-ti* (*typ-info-t TYPE*('*a*)) *bs v*
proof −
  have *wf*: *wf-desc* (*typ-info-t TYPE*('*a*)) **by** *simp*
  have *wf-sz*: *wf-size-desc* (*typ-info-t TYPE*('*a*)) **by** *simp*
  have *wf-descs*: *wf-field-descs* (*set* (*field-descs* (*typ-info-t TYPE*('*a*))))
    using *local.wf-field-descs* **by** *blast*
  have *wf-comp*: *wf-component-descs* (*typ-info-t TYPE*('*a*))
    by (*simp add*: *local.wf-component-descs*)
  have *wf-fd*: *wf-fd* (*typ-info-t TYPE*('*a*))
    using *local.wf-desc local.wf-lf wf-fdp-fd*(*1*) *wf-lf-fdp* **by** *blast*
  from *is-padding-byte-update-ti-id* [*OF wf wf-sz wf-descs wf-comp wf-fd*
      i-bound [*simplified size-of-def*] *lbs* [*simplified size-of-def*]
      padding [*simplified typ-uinfo-t-def*]]
  show *?thesis* **.**
qed
end

lemma *heap-update-fold*:
sz = *size-of TYPE*('*a*) ⟹
  heap-update-list a (*to-bytes* (*v*::'*a*:: *c-type*) (*heap-list h sz a*)) *h* = *heap-update* (*Ptr a*) *v h*
  by (*simp add*:*heap-update-def*)

lemma *heap-update-padding-fold*:
sz = *size-of TYPE*('*a*) ⟹
  heap-update-list a (*to-bytes* (*v*::'*a*:: *c-type*) *bs*) *h* = *heap-update-padding* (*Ptr a*) *v bs h*

by (*simp add*:*heap-update-padding-def*)

context *padding-base*
begin
thm *is-padding-byte-def*
thm *is-value-byte-def*
end

lemma *is-value-byte-access-ti-id'*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
⟦ *wf-desc t*; *wf-size-desc t*; *wf-field-descs* (*set* (*field-descs t*)); *wf-component-descs t*; *wf-fd t*;
i < *size-td t*; *length bs* = *size-td t*;
∃ *bs b*. *length bs* = *size-td t* ∧ *norm-tu* (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) ≠ *norm-tu* (*export-uinfo t*) *bs*⟧
⟹ *access-ti t v* (*bs*[*i* := *b*]) = *access-ti t v bs*

⟦ *wf-desc-struct st*; *wf-size-desc-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));
wf-component-descs-struct st; *wf-fd-struct st*; *i* < *size-td-struct st*; *length bs* = *size-td-struct st*;
∃ *bs b*. *length bs* = *size-td-struct st* ∧ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) ≠ *norm-tu-struct* ((*map-td-struct field-norm* (λ-. ()) *st*)) *bs*⟧
⟹ *access-ti-struct st v* (*bs*[*i* := *b*]) = *access-ti-struct st v bs*

⟦ *wf-desc-list ts*; *wf-size-desc-list ts*; *wf-field-descs* (*set* (*field-descs-list ts*));
wf-component-descs-list ts; *wf-fd-list ts*; *i* < *size-td-list ts*; *length bs* = *size-td-list ts*;
∃ *bs b*. *length bs* = *size-td-list ts* ∧ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) ≠ *norm-tu-list* ((*map-td-list field-norm* (λ-. ()) *ts*)) *bs*⟧
⟹ *access-ti-list ts v* (*bs*[*i* := *b*]) = *access-ti-list ts v bs*

⟦ *wf-desc-tuple x*; *wf-size-desc-tuple x*; *wf-field-descs* (*set* (*field-descs-tuple x*));
wf-component-descs-tuple x; *wf-fd-tuple x*; *i* < *size-td-tuple x*; *length bs* = *size-td-tuple x*;
∃ *bs b*. *length bs* = *size-td-tuple x* ∧ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) ≠ *norm-tu-tuple* ((*map-td-tuple field-norm* (λ-. ()) *x*)) *bs*⟧
⟹ *access-ti-tuple x v* (*bs*[*i* := *b*]) = *access-ti-tuple x v bs*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *bs i* **and** *bs i* **and** *bs i* **and** *bs i*)
case (*TypDesc algn st nm*)
then show *?case* **by** *auto*
next

```
  case (TypScalar sz algn d)
  from TypScalar obtain
    wf-d: wf-field-desc d and
    field-sz: field-sz d =sz and
    i-bound: i < sz and
    lbs: length bs = sz and
    is-val: ∃ bs b. length bs = sz ∧
        field-norm sz algn d (bs[i := b]) ≠ field-norm sz algn d bs
    by simp

  interpret wf: wf-field-desc d by (rule wf-d)
  show ?case
  proof (cases wf.is-padding-byte i)
    case True
    then show ?thesis
      using field-sz lbs wf.is-padding-byte-upd-eq
      by (metis field-norm-def length-list-update is-val)

  next
    case False
    note not-padding = this
    with wf.complement field-sz i-bound have is-value: wf.is-value-byte i by blast
    from is-value is-val
    show ?thesis
      apply simp
      by (metis field-sz lbs padding-base.is-value-byte-acc-eq)
  qed
next
case (TypAggregate ts)
then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto
  from Cons-typ-desc.prems obtain
    wf-desc-d: wf-desc d and
    wf-fd-x: wf-fd-tuple (DTuple d nm y) and
    wf-fd-d: wf-fd d and
    wf-fd-fs: wf-fd-list fs and
    nm-notin: nm ∉ dt-snd ‘ set fs and
    wf-desc-fs: wf-desc-list fs and
    wf-sz-d: wf-size-desc d and
    wf-sz-fs: wf-size-desc-list fs and
    wf-field-d: wf-field-desc (component-desc d) and
    wf-fields-d: wf-field-descs (set (field-descs d)) and
    wf-fields-fs: wf-field-descs (set (field-descs-list fs)) and
    y: y = component-desc d and
```

wf-comp-d: *wf-component-descs d* **and**
wf-comp-fs: *wf-component-descs-list fs* **and**
i-bound:$i <$ *size-td d* + *size-td-list fs* **and**
lbs: *length bs* = *size-td d* + *size-td-list fs* **and**
wf-desc-x: *wf-desc-tuple x* **and**
wf-sz-x: *wf-size-desc-tuple x* **and**
wf-decs-x: *wf-field-descs* (*set* (*field-descs-tuple x*)) **and**
wf-comp-x: *wf-component-descs-tuple x* **and**

is-value: $\exists$ *bs b*. *length bs* = *size-td d* + *size-td-list fs* $\wedge$
norm-tu (*map-td field-norm* (λ-. ()) *d*)
(*take* (*size-td d*) (*bs*[*i* := *b*])) @
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*)
(*drop* (*size-td d*) (*bs*[*i* := *b*])) $\neq$
norm-tu (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) *bs*) @
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*)
(*drop* (*size-td d*) *bs*)
by (*auto simp add*: *x*)

from *is-value* **obtain** *xbs xb* **where**
lxbs: *length xbs* = *size-td d* + *size-td-list fs* **and**
norm-neq: *norm-tu* (*map-td field-norm* (λ-. ()) *d*)
(*take* (*size-td d*) (*xbs*[*i* := *xb*])) @
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*)
(*drop* (*size-td d*) (*xbs*[*i* := *xb*])) $\neq$
norm-tu (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) *xbs*) @
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*)
(*drop* (*size-td d*) *xbs*)
by *blast*

from *lxbs* **have** *l-take1*: *length* (*take* (*size-td d*) (*xbs*[*i* := *xb*])) = *size-td d*
by *simp*
from *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d l-take1*]
have *l-norm-upd-d*: *length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) (*xbs*[*i* := *xb*]))) = *size-td d*
by (*metis access-ti*$_0$*-def export-uinfo-def fd-cons-length-p wf-fd-consD wf-fd-d*)

from *lxbs* **have** *l-take2*: *length* (*take* (*size-td d*) (*xbs*)) = *size-td d*
by *simp*
from *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d l-take2*]
have *l-norm-d*: *length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) *xbs*)) = *size-td d*
by (*metis access-ti*$_0$*-def export-uinfo-def fd-cons-length-p wf-fd-consD wf-fd-d*)

from *lbs*
have *lbs-drop*: *length* (*drop* (*size-td d*) *bs*) = *size-td-list fs*
by (*simp add*: *x*)

from *lbs*

```
  have lbs-take: length (take (size-td d) bs) = size-td d
    by (simp add: x)

  have lacc: length (access-ti d v (take (size-td d) bs)) = size-td d
    by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps wf-fd-x x)

  show ?case

  proof (cases i < size-td d)
    case True
    from True lbs
    have take-eq: (take (size-td d) bs)[i := b] = take (size-td d) (bs[i := b])
      by (simp add: take-update-swap)
    from True
    have drop-eq: drop (size-td d) (xbs[i := xb]) = drop (size-td d) xbs
      by simp
    from lxbs True
    have take-xbs-eq: (take (size-td d) (xbs[i := xb])) = (take (size-td d) xbs)[i :=
xb]
      by (simp add: take-update-swap)
    from norm-neq
    have norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d) (xbs[i := xb])) ≠
           norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d) xbs)
      by (simp add: drop-eq)
    with take-xbs-eq
    have value-d: ∃ bs. length bs = size-td d ∧
       (∃ b. norm-tu (map-td field-norm (λ-. ()) d) (bs[i := b]) ≠
            norm-tu (map-td field-norm (λ-. ()) d) bs)
      apply simp
      using l-take2 by blast
    note hyp = Cons-typ-desc.hyps(1) [OF wf-desc-x wf-sz-x wf-decs-x wf-comp-x,
simplified x, OF wf-fd-x, simplified, OF True lbs-take value-d]
    from hyp lbs
    show ?thesis
      by  (simp add: x True nth-append lacc take-eq)
  next
    case False
    from False i-bound have i-bound': i − size-td d  < size-td-list fs by simp

    from False lxbs
    have take-xbs: take (size-td d) (xbs[i := xb]) = take (size-td d) xbs by simp

    from norm-neq
    have norm-tu-list (map-td-list field-norm (λ-. ()) fs) (drop (size-td d) (xbs[i :=
xb])) ≠
           norm-tu-list (map-td-list field-norm (λ-. ()) fs) (drop (size-td d) xbs)
      by (simp add: take-xbs)
    then
    have value-fs: ∃ bs b.
```

length bs = size-td-list fs ∧
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*) (*bs*[*i* − *size-td d* := *b*]) ≠
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*) *bs*
by (*metis False drop-update-swap lbs lbs-drop length-drop lxbs not-le*)

from *False lbs* **have** *drop-eq*: ((*drop* (*size-td d*) *bs*)[*i* − *size-td d* := *b*]) =
(*drop* (*size-td d*) (*bs*[*i* := *b*]))
by (*simp add*: *drop-update-swap*)
from *False lbs* **have** *take-eq*: *take* (*size-td d*) (*bs*[*i* := *b*]) = *take* (*size-td d*) *bs*
by *simp*
note *hyp* = *Cons-typ-desc.hyps*(*2*) [*OF wf-desc-fs wf-sz-fs wf-fields-fs wf-comp-fs wf-fd-fs i-bound′ lbs-drop value-fs*]
from *hyp*
show *?thesis* **by** (*simp add*: *x False nth-append lacc drop-eq take-eq*)
qed
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** (*auto simp add*: *export-uinfo-def*)
qed

lemma *is-value-byte-access-ti-id*:
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
assumes *wf-descs*: *wf-field-descs* (*set* (*field-descs t*))
assumes *wf-comps*: *wf-component-descs t*
assumes *wf-fd*: *wf-fd t*
assumes *i-bound*: *i* < *size-td t*
assumes *lbs*: *length bs* = *size-td t*
assumes *is-value*: *is-value-byte* (*export-uinfo t*) *i*
shows *access-ti t v* (*bs*[*i* := *b*]) = *access-ti t v bs*
using *is-value-byte-access-ti-id′*(*1*)[*OF wf wf-sz wf-descs wf-comps wf-fd i-bound lbs*, **where** *v*=*v* **and** *b*=*b*] *is-value*
by (*simp add*: *is-value-byte-def export-uinfo-def*)

lemma *is-value-byte-access-ti-update-ti-cancel′*:
fixes *t*::′*a xtyp-info*
and *st*::′*a xtyp-info-struct*
and *ts*::′*a xtyp-info-tuple list*
and *x*::′*a xtyp-info-tuple*
shows
⟦ *wf-desc t*; *wf-size-desc t*; *wf-field-descs* (*set* (*field-descs t*)); *wf-component-descs t*; *wf-fd t*;
i < *size-td t*; *length bs* = *size-td t*; *length bs′* = *size-td t*;
∃ *bs b*. *length bs* = *size-td t* ∧ *norm-tu* (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) ≠ *norm-tu* (*export-uinfo t*) *bs*⟧
⟹ *access-ti t* (*update-ti t bs v*) *bs′* ! *i* = *bs* ! *i*

⟦ *wf-desc-struct st*; *wf-size-desc-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));

wf-component-descs-struct st; *wf-fd-struct st*; $i <$ *size-td-struct st*; *length bs* = *size-td-struct st*; *length bs'* = *size-td-struct st*;
$\exists$ *bs b. length bs* = *size-td-struct st* $\wedge$ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) $\neq$ *norm-tu-struct* ((*map-td-struct field-norm* (λ-. ()) *st*)) *bs*$\rrbracket$
$\Longrightarrow$ *access-ti-struct st* (*update-ti-struct st bs v*) *bs'* ! *i* = *bs* ! *i*

$[\![$ *wf-desc-list ts*; *wf-size-desc-list ts*; *wf-field-descs* (*set* (*field-descs-list ts*));
wf-component-descs-list ts; *wf-fd-list ts*; $i <$ *size-td-list ts*; *length bs* = *size-td-list ts*; *length bs'* = *size-td-list ts*;
$\exists$ *bs b. length bs* = *size-td-list ts* $\wedge$ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) $\neq$ *norm-tu-list* ((*map-td-list field-norm* (λ-. ()) *ts*)) *bs*$\rrbracket$
$\Longrightarrow$ *access-ti-list ts* (*update-ti-list ts bs v*) *bs'* ! *i* = *bs* ! *i*

$[\![$ *wf-desc-tuple x*; *wf-size-desc-tuple x*; *wf-field-descs* (*set* (*field-descs-tuple x*));
wf-component-descs-tuple x; *wf-fd-tuple x*; $i <$ *size-td-tuple x*; *length bs* = *size-td-tuple x*; *length bs'* = *size-td-tuple x*;
$\exists$ *bs b. length bs* = *size-td-tuple x* $\wedge$ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) $\neq$ *norm-tu-tuple* ((*map-td-tuple field-norm* (λ-. ()) *x*)) *bs*$\rrbracket$
$\Longrightarrow$ *access-ti-tuple x* (*update-ti-tuple x bs v*) *bs'* ! *i* = *bs* ! *i*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *bs bs' i v* **and** *bs bs' i v* **and** *bs bs' i v* **and** *bs bs' i v*)
case (*TypDesc algn st nm*)
then show *?case* **by** *auto*
next
case (*TypScalar sz algn d*)
from *TypScalar* **obtain**
wf-d: *wf-field-desc d* **and**
field-sz: *field-sz d* = *sz* **and**
i-bound: *i* < *sz* **and**
lbs: *length bs* = *sz* **and**
lbs': *length bs'* = *sz* **and**
is-val: $\exists$ *bs b. length bs* = *sz* $\wedge$
field-norm sz algn d (*bs*[*i* := *b*]) $\neq$ *field-norm sz algn d bs*
by *simp*

interpret *wf*: *wf-field-desc d* **by** (*rule wf-d*)
show *?case*
proof (*cases wf.is-padding-byte i*)
case *True*
then show *?thesis*
using *field-sz lbs wf.is-padding-byte-upd-eq*
by (*metis field-norm-def length-list-update is-val*)

next
case *False*
note *not-padding* = *this*
with *wf.complement field-sz i-bound* **have** *is-value*: *wf.is-value-byte i* **by** *blast*
from *is-value is-val lbs'*

```
    show ?thesis
      by (simp add: field-sz lbs wf.is-value-byte-acc-upd-cancel)
  qed
next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto

  from Cons-typ-desc.prems obtain
    wf-desc-d: wf-desc d and
    wf-fd-x: wf-fd-tuple (DTuple d nm y) and
    wf-fd-d: wf-fd d and
    wf-fd-fs: wf-fd-list fs and
    nm-notin: nm ∉ dt-snd ‘ set fs and
    wf-desc-fs: wf-desc-list fs and
    wf-sz-d: wf-size-desc d and
    wf-sz-fs: wf-size-desc-list fs and
    wf-field-d: wf-field-desc (component-desc d) and
    wf-fields-d: wf-field-descs (set (field-descs d)) and
    wf-fields-fs: wf-field-descs (set (field-descs-list fs)) and
    y: y = component-desc d and
    wf-comp-d: wf-component-descs d and
    wf-comp-fs: wf-component-descs-list fs and
    i-bound:i < size-td d + size-td-list fs and
    lbs: length bs = size-td d + size-td-list fs and
    lbs': length bs' = size-td d + size-td-list fs and
    wf-desc-x: wf-desc-tuple x and
    wf-sz-x: wf-size-desc-tuple x and
    wf-decs-x: wf-field-descs (set (field-descs-tuple x)) and
    wf-comp-x: wf-component-descs-tuple x and

    is-value: ∃ bs b. length bs = size-td d + size-td-list fs ∧
          norm-tu (map-td field-norm (λ-. ()) d)
              (take (size-td d) (bs[i := b])) @
             norm-tu-list (map-td-list field-norm (λ-. ()) fs)
              (drop (size-td d) (bs[i := b])) ≠
             norm-tu (map-td field-norm (λ-. ()) d) (take (size-td d) bs) @
             norm-tu-list (map-td-list field-norm (λ-. ()) fs)
              (drop (size-td d) bs) and
    commutes: fu-commutes (update-ti-t d) (update-ti-list-t fs)
    by (auto simp add: x)


  from is-value obtain xbs xb where
```

lxbs: *length xbs* = *size-td d* + *size-td-list fs* **and**
norm-neq: *norm-tu* (*map-td field-norm* (λ-. ()) *d*)
(*take* (*size-td d*) (*xbs*[*i* := *xb*])) @
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*)
(*drop* (*size-td d*) (*xbs*[*i* := *xb*])) ≠
norm-tu (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) *xbs*) @
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*)
(*drop* (*size-td d*) *xbs*)
by *blast*

from *lxbs* **have** *l-take1*: *length* (*take* (*size-td d*) (*xbs*[*i* := *xb*])) = *size-td d*
by *simp*
from *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d l-take1*]
have *l-norm-upd-d*: *length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) (*xbs*[*i* := *xb*]))) = *size-td d*
by (*metis access-ti*$_0$*-def export-uinfo-def fd-cons-length-p wf-fd-consD wf-fd-d*)

from *lxbs* **have** *l-take2*: *length* (*take* (*size-td d*) (*xbs*)) = *size-td d*
by *simp*
from *wf-fd-norm-tu*(*1*)[*rule-format*, *OF wf-fd-d l-take2*]
have *l-norm-d*: *length* (*norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) *xbs*)) = *size-td d*
by (*metis access-ti*$_0$*-def export-uinfo-def fd-cons-length-p wf-fd-consD wf-fd-d*)

from *lbs*
have *lbs-drop*: *length* (*drop* (*size-td d*) *bs*) = *size-td-list fs*
by (*simp add*: *x*)

from *lbs′*
have *lbs′-drop*: *length* (*drop* (*size-td d*) *bs′*) = *size-td-list fs*
by (*simp add*: *x*)

from *lbs*
have *lbs-take*: *length* (*take* (*size-td d*) *bs*) = *size-td d*
by (*simp add*: *x*)

from *lbs′*
have *lbs′-take*: *length* (*take* (*size-td d*) *bs′*) = *size-td d*
by (*simp add*: *x*)

have *lacc*: *length* (*access-ti d v* (*take* (*size-td d*) *bs*)) = *size-td d*
by (*metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps wf-fd-x x*)

from *lbs lbs′-take*
have *l-acc-upd*:
length (*access-ti d* (*update-ti d* (*take* (*size-td d*) *bs*)
(*update-ti-list fs* (*drop* (*size-td d*) *bs*) *v*)) (*take* (*size-td d*) *bs′*)) = *size-td d*
by (*meson length-fa-ti wf-fd-d*)

show *?case*

proof (*cases i* < *size-td d*)
case *True*
from *True lbs*
have *take-eq*: (*take* (*size-td d*) *bs*)[*i* := *b*] = *take* (*size-td d*) (*bs*[*i* := *b*])
by (*simp add*: *take-update-swap*)
from *True*
have *drop-eq*: *drop* (*size-td d*) (*xbs*[*i* := *xb*]) = *drop* (*size-td d*) *xbs*
by *simp*
from *lxbs True*
have *take-xbs-eq*: (*take* (*size-td d*) (*xbs*[*i* := *xb*])) = (*take* (*size-td d*) *xbs*)[*i* := *xb*]
by (*simp add*: *take-update-swap*)
from *norm-neq*
have *norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) (*xbs*[*i* := *xb*])) ≠
norm-tu (*map-td field-norm* (λ-. ()) *d*) (*take* (*size-td d*) *xbs*)
by (*simp add*: *drop-eq*)
with *take-xbs-eq*
have *value-d*: ∃ *bs*. *length bs* = *size-td d* ∧
(∃ *b*. *norm-tu* (*map-td field-norm* (λ-. ()) *d*) (*bs*[*i* := *b*]) ≠
norm-tu (*map-td field-norm* (λ-. ()) *d*) *bs*)
apply *simp*
using *l-take2* **by** *blast*

note *hyp* = *Cons-typ-desc.hyps*(*1*) [*OF wf-desc-x wf-sz-x wf-decs-x wf-comp-x*, *simplified x*, *OF wf-fd-x*, *simplified*,
OF True lbs-take lbs'-take value-d, **where** *v* = (*update-ti-list fs* (*drop* (*size-td d*) *bs*) *v*)]
from *lbs lbs'*
show *?thesis*
by (*simp add*: *x True nth-append lacc take-eq l-acc-upd hyp*)
next
case *False*
from *False i-bound* **have** *i-bound'*: *i* − *size-td d* < *size-td-list fs* **by** *simp*

from *False lxbs*
have *take-xbs*: *take* (*size-td d*) (*xbs*[*i* := *xb*]) = *take* (*size-td d*) *xbs* **by** *simp*

from *norm-neq*
have *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *fs*) (*drop* (*size-td d*) (*xbs*[*i* := *xb*])) ≠
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*) (*drop* (*size-td d*) *xbs*)
by (*simp add*: *take-xbs*)
then
have *value-fs*: ∃ *bs b*.
length bs = *size-td-list fs* ∧

norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*) (*bs*[*i* − *size-td d* := *b*]) ≠
norm-tu-list (*map-td-list field-norm* (λ-. ()) *fs*) *bs*
by (*metis False drop-update-swap lbs lbs-drop length-drop lxbs not-le*)

from *commutes* **have** *commute*:
(*update-ti d* (*take* (*size-td d*) *bs*)
(*update-ti-list fs* (*drop* (*size-td d*) *bs*) *v*)) =
(*update-ti-list fs* (*drop* (*size-td d*) *bs*) (*update-ti d* (*take* (*size-td d*) *bs*) *v*))
apply (*simp add*: *update-ti-t-def update-ti-list-t-def fu-commutes-def*)
apply (*erule allE* [**where** *x*=*v*])
apply (*erule allE* [**where** *x*=(*take* (*size-td d*) *bs*)])
apply (*erule allE* [**where** *x*=(*drop* (*size-td d*) *bs*)])
using *lbs-take lbs-drop*
apply *simp*
done

from *False lbs* **have** *drop-eq*: *drop* (*size-td d*) *bs* ! (*i* − *size-td d*) = *bs* ! *i*
by (*simp add*: *drop-update-swap*)

note *hyp* = *Cons-typ-desc.hyps*(*2*) [*OF wf-desc-fs wf-sz-fs wf-fields-fs wf-comp-fs wf-fd-fs i-bound′ lbs-drop lbs′-drop value-fs*]
from *hyp*
show *?thesis*
apply (*simp add*: *x False nth-append lacc drop-eq l-acc-upd*)
apply (*simp add*: *commute*)
done
qed
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** (*auto simp add*: *export-uinfo-def*)
qed

lemma *is-value-byte-access-ti-update-ti-cancel*:
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
assumes *wf-descs*: *wf-field-descs* (*set* (*field-descs t*))
assumes *wf-comps*: *wf-component-descs t*
assumes *wf-fd*: *wf-fd t*
assumes *i-bound*: *i* < *size-td t*
assumes *lbs*: *length bs* = *size-td t*
assumes *lbs′*: *length bs′*= *size-td t*
assumes *is-value*: *is-value-byte* (*export-uinfo t*) *i*
shows *access-ti t* (*update-ti t bs v*) *bs′* ! *i* = *bs* ! *i*
using *is-value-byte-access-ti-update-ti-cancel′*(*1*) [*OF wf wf-sz wf-descs wf-comps wf-fd i-bound lbs lbs′*] *is-value*
by (*simp add*: *export-uinfo-def is-value-byte-def*)

lemma *field-lookup-is-padding-byte-focus′*:

fixes t::$('a, 'b)$ *typ-info*
and st::$('a, 'b)$ *typ-info-struct*
and ts::$('a, 'b)$ *typ-info-tuple list*
and x::$('a, 'b)$ *typ-info-tuple*
shows
$[\![$*field-lookup t f m* = *Some* $(s, m + n)$; $n \leq i$; $i < n +$ *size-td s*; *length bs* = *size-td t*;
wf-desc t; *wf-size-desc t*$]\!] \Longrightarrow$
norm-tu (*map-td field-norm* $(\lambda$-. $())$ *t*) (*bs*$[i := b]$) = *norm-tu* (*map-td field-norm* $(\lambda$-. $())$ *t*) *bs*
$\longleftrightarrow$
norm-tu (*export-uinfo s*) ((*take* (*size-td s*) (*drop n bs*))$[i - n := b]$) =
norm-tu (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*))

$[\![$*field-lookup-struct st f m* = *Some* $(s, m + n)$; $n \leq i$; $i < n +$ *size-td s*; *length bs* = *size-td-struct st*;
wf-desc-struct st; *wf-size-desc-struct st*$]\!] \Longrightarrow$
norm-tu-struct (*map-td-struct field-norm* $(\lambda$-. $())$ *st*) (*bs*$[i := b]$) = *norm-tu-struct* (*map-td-struct field-norm* $(\lambda$-. $())$ *st*) *bs*
$\longleftrightarrow$
norm-tu (*export-uinfo s*) ((*take* (*size-td s*) (*drop n bs*))$[i - n := b]$) =
norm-tu (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*))

$[\![$*field-lookup-list ts f m* = *Some* $(s, m + n)$; $n \leq i$; $i < n +$ *size-td s*; *length bs* = *size-td-list ts*;
wf-desc-list ts; *wf-size-desc-list ts*$]\!] \Longrightarrow$
norm-tu-list (*map-td-list field-norm* $(\lambda$-. $())$ *ts*) (*bs*$[i := b]$) = *norm-tu-list* (*map-td-list field-norm* $(\lambda$-. $())$ *ts*) *bs*
$\longleftrightarrow$
norm-tu (*export-uinfo s*) ((*take* (*size-td s*) (*drop n bs*))$[i - n := b]$) =
norm-tu (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*))

$[\![$*field-lookup-tuple x f m* = *Some* $(s, m + n)$; $n \leq i$; $i < n +$ *size-td s*; *length bs* = *size-td-tuple x*;
wf-desc-tuple x; *wf-size-desc-tuple x*$]\!] \Longrightarrow$
norm-tu-tuple (*map-td-tuple field-norm* $(\lambda$-. $())$ *x*) (*bs*$[i := b]$) = *norm-tu-tuple* (*map-td-tuple field-norm* $(\lambda$-. $())$ *x*) *bs*
$\longleftrightarrow$
norm-tu (*export-uinfo s*) ((*take* (*size-td s*) (*drop n bs*))$[i - n := b]$) =
norm-tu (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*))
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m n i s bs* **and** *f m n i s bs* **and** *f m n i s bs* **and** *f m n i s bs*)
case (*TypDesc algn st nm*)
then show *?case* **by** (*auto split*: *if-split-asm*)
next
case (*TypScalar sz algn d*)
then show *?case* **by** *auto*
next

```
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto
  from Cons-typ-desc.prems obtain
    lbs: length bs = size-td-tuple x + size-td-list fs and
    wf-desc-x: wf-desc-tuple (DTuple d nm y) and
    wf-size-desc-x: wf-size-desc-tuple (DTuple d nm y) and
    wf-size-desc-fs: wf-size-desc-list fs and
    nm-notin:  nm ∉ dt-snd ‘ set fs and
    wf-desc-fs: wf-desc-list fs and
    i-lower: n ≤ i and
    i-upper: i < n + size-td s
    by (auto simp add: x)

  from lbs
  have lbs-drop: length (drop (size-td-tuple x) bs) = size-td-list fs
    by simp

  from lbs
  have lbs-take: length (take (size-td-tuple x) bs) = size-td-tuple (DTuple d nm y)
    by (simp add: x)

  from lbs-take
  have take-eq1: ((take (size-td d) bs)[i := b]) = (take (size-td d) (bs[i := b]))
    by (simp add: take-update-swap)

  from Cons-typ-desc.prems obtain f1 fxs
    where f: f = f1#fxs
    by (cases f) auto

  thm Cons-typ-desc.hyps [simplified x]

  show ?case
  proof (cases f1 = nm)
    case True
    show ?thesis
    proof (cases field-lookup d fxs m)
      case None
      from Cons-typ-desc.prems field-lookup-list-None [OF nm-notin]
      have False
        by (simp add: True None f x)
      thus ?thesis by simp
    next
      case (Some r)
```

```
    from Cons-typ-desc.prems have r: r = (s, m + n)
      by (simp add: x True Some f)
    hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
      by  (simp add: f True Some r)
   from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
    have wf-size-desc s .
    from wf-size-desc-gt(1)[OF this]
    have 0 < size-td s .


    with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
    have n-le: n < size-td d
      by auto

    have i-in-d: i < size-td d
      using ‹size-td s + (m + n − m) ≤ size-td-tuple (DTuple d nm y)› i-upper
by auto
     have bound: size-td s + (m + n − m) ≤ size-td-tuple (DTuple d nm y) by
fact
    from bound

    have take-eq: (take (size-td s) (drop n (take (size-td d) bs))) =
       (take (size-td s) (drop n bs))
      by (simp add: take-drop)
       from  Cons-typ-desc.hyps(1)[simplified x, OF fl i-lower i-upper lbs-take
wf-desc-x wf-size-desc-x] lbs bound
    show ?thesis
      by (simp add: x True Some r take-eq  nth-append i-in-d take-eq1)
  qed
 next
  case False
  with Cons-typ-desc.prems
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
    by (clarsimp simp add: x f False)
  hence n-bound: size-td d ≤ n
    by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

  from fl
  have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, (m + size-td
d) + (n − size-td d))
    by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD)
  from n-bound
  have take-eq: (take (size-td s) (drop (n − size-td d + size-td d) bs)) =
        (take (size-td s) (drop n bs))
    by simp

  have i-lower': n − size-td d ≤ i − size-td d
    using diff-le-mono i-lower by blast
```

```
    have i-upper': i − size-td d < n − size-td d + size-td s
      using i-lower i-upper by linarith

    have i-notin-d: ¬ i < size-td d
      by (meson i-lower leD less-le-trans n-bound)

   from i-notin-d lbs-take have take-eq: take (size-td d) (bs[i := b]) = take (size-td
d) bs
      by simp

    from i-notin-d lbs i-upper not-le have drop-eq:
      ((drop (size-td d) bs)[i − size-td d := b]) = (drop (size-td d) (bs[i := b]))
      by (simp add: drop-update-swap)

   from Cons-typ-desc.hyps(2) [OF fl i-lower' i-upper' lbs-drop  wf-desc-fs wf-size-desc-fs]
      False n-bound
    show ?thesis
      by (simp add: x f nth-append i-notin-d take-eq drop-eq)
  qed
next
  case (DTuple-typ-desc d nm y)
then show ?case apply (cases f) by (auto split: if-split-asm)
qed

lemma field-lookup-is-padding-byte-focus:
assumes fl: field-lookup t f 0 = Some (s, n)
assumes i-lower: n ≤ i
assumes i-upper :i < n + size-td s
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows is-padding-byte (export-uinfo t) i = is-padding-byte (export-uinfo s) (i − n)
proof
  assume is-padding: is-padding-byte (export-uinfo t) i
  show is-padding-byte (export-uinfo s) (i − n)
    apply (simp add: is-padding-byte-def, safe)
  proof −
    from i-upper i-lower show i − n < size-td s by simp
  next
    fix bs::byte list and b::byte
    assume lbs: length bs = size-td s
    from fl have le: n + size-td s ≤ size-td t
      by (metis add.commute field-lookup-offset-size')
    with lbs i-lower i-upper obtain pfx sfx xbs where
      xbs: xbs = pfx @ bs @ sfx and
      lxbs: length xbs = size-td t and
      lpfx: length pfx = n

    by (metis Ex-list-of-length ab-semigroup-add-class.add-ac(1) le-Suc-ex length-append)
```

```
    have take-eq: take (size-td s) (drop n xbs) = bs
      using xbs lxbs lbs lpfx le
      by simp
     from field-lookup-is-padding-byte-focus'(1) [where m = 0, simplified, OF fl
i-lower i-upper lxbs wf wf-sz, where b=b]
    is-padding lxbs
    show norm-tu (export-uinfo s) (bs[i − n := b]) =
          norm-tu (export-uinfo s) bs
      by (simp add: take-eq export-uinfo-def is-padding-byte-def)
  qed
next
  assume is-padding: is-padding-byte (export-uinfo s) (i − n)
  from fl i-upper i-lower
  have i-t: i < size-td t
    using field-lookup-offset-size' by fastforce
  show is-padding-byte (export-uinfo t) i
    apply (simp add: is-padding-byte-def, safe)
  proof −
    from i-t
    show sz: i < size-td t .
  next
    fix bs::byte list and b::byte
    assume lbs: length bs = size-td t
    from fl i-upper i-lower i-t
    have less: i − n < size-td t
      by simp

    from less lbs i-lower i-upper i-t
    have sz-s: size-td s ≤ length (drop n bs)
      by (metis add-le-imp-le-diff field-lookup-offset-size' fl length-drop)

     from field-lookup-is-padding-byte-focus'(1) [where m = 0, simplified, OF fl
i-lower i-upper lbs wf wf-sz, where b=b]
    is-padding sz-s
    show norm-tu (export-uinfo t) (bs[i := b]) =
          norm-tu (export-uinfo t) bs
      by (auto simp add: is-padding-byte-def export-uinfo-def)
  qed
qed

lemma field-lookup-is-value-byte-focus:
assumes fl: field-lookup t f 0 = Some (s, n)
assumes i-lower: n ≤ i
assumes i-upper :i < n + size-td s
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows is-value-byte (export-uinfo t) i = is-value-byte (export-uinfo s) (i − n)
proof −
```

from *fl i-upper i-lower*
have *i-t*: $i < size\text{-}td\ t$
using *field-lookup-offset-size'* **by** *fastforce*
from *i-lower i-upper*
have $i - n < size\text{-}td\ s$ **by** *simp*
with *field-lookup-is-padding-byte-focus* [*OF fl i-lower i-upper wf wf-sz*] *complement-tu-padding i-t*
show *?thesis*
by (*simp add*: *export-uinfo-size*)
qed

lemma *field-lookup-eq-padding-focus*:
assumes *fl*: *field-lookup t f 0* = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-td t*
assumes *lbs'*: *length bs'* = *size-td t*
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
assumes *eq-padding*: *eq-padding* (*export-uinfo t*) *bs bs'*
shows *eq-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
proof −
from *fl lbs* **have** *bound-s*: *size-td s* ≤ *length bs* − *n*
by (*metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size'*)
from *fl lbs'* **have** *bound-s'*: *size-td s* ≤ *length bs'* − *n*
by (*metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size'*)
from *bound-s lbs*
have *l-take*: *length* (*take* (*size-td s*) (*drop n bs*)) = *size-td s*
by *simp*
from *bound-s' lbs'*
have *l-take'*: *length* (*take* (*size-td s*) (*drop n bs'*)) = *size-td s*
by *simp*

show *?thesis*
apply (*simp add*: *eq-padding-def*, *safe*)
proof −
show *size-td s* ≤ *length bs* − *n* **by** (*rule bound-s*)
next
show *size-td s* ≤ *length bs'* − *n* **by** (*rule bound-s'*)
next
fix *i* **assume** *is-padding*: *is-padding-byte* (*export-uinfo s*) *i*
from *is-padding* **have** *i-s*: *i* < *size-td s*
by (*simp add*: *is-padding-byte-def*)
have *n-i*: *n* ≤ *i* + *n* **by** *simp*
from *i-s* **have** *i-n-bound*: *i* + *n* < *n* + *size-td s* **by** *simp*
show *take* (*size-td s*) (*drop n bs*) ! *i* =
take (*size-td s*) (*drop n bs'*) ! *i*

using *l-take l-take' is-padding eq-padding lbs lbs'*

field-lookup-is-padding-byte-focus [*OF fl n-i i-n-bound wf wf-sz*, *simplified*]
by (*auto simp add*: *eq-padding-def add.commute is-padding-byte-def*)
qed
qed

lemma *field-lookup-eq-padding-focus-eq*:
assumes *fl*: *field-lookup t f 0* = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-td t*
assumes *lbs'*: *length bs'* = *size-td t*
assumes *pfx-eq*: $\bigwedge i.\ i < n \Longrightarrow bs\ !\ i = bs'\ !\ i$
assumes *sfx-eq*: $\bigwedge i.\ n + size\text{-}td\ s \leq i \Longrightarrow i < size\text{-}td\ t \Longrightarrow bs\ !\ i = bs'\ !\ i$
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
shows *eq-padding* (*export-uinfo t*) *bs bs'* =
eq-padding (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
proof −
from *fl lbs* **have** *bound-s*: *size-td s* ≤ *length bs* − *n*
by (*metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size'*)
from *fl lbs'* **have** *bound-s'*: *size-td s* ≤ *length bs'* − *n*
by (*metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size'*)
from *bound-s lbs*
have *l-take*: *length* (*take* (*size-td s*) (*drop n bs*)) = *size-td s*
by *simp*
from *bound-s' lbs'*
have *l-take'*: *length* (*take* (*size-td s*) (*drop n bs'*)) = *size-td s*
by *simp*

show *?thesis*
proof
assume *eq-padding*: *eq-padding* (*export-uinfo t*) *bs bs'*
show *eq-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
by (*rule field-lookup-eq-padding-focus* [*OF fl lbs lbs' wf wf-sz eq-padding*])
next
assume *eq-padding*: *eq-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
show *eq-padding* (*export-uinfo t*) *bs bs'*
apply (*simp add*: *eq-padding-def*, *safe*)
proof −
show *length bs* = *size-td t* **by** (*rule lbs*)
next
show *length bs'* = *size-td t* **by** (*rule lbs'*)
next
fix *i*
assume *is-padding*: *is-padding-byte* (*export-uinfo t*) *i*
from *is-padding* **have** *i-t*: *i* < *size-td t*

```
      by (simp add: is-padding-byte-def)
    show bs ! i = bs' ! i
    proof (cases i < n)
      case True
      with pfx-eq show ?thesis by simp
    next
      case False
      hence i-lower: n ≤ i by simp
      show ?thesis
      proof (cases i < n + size-td s)
        case True
          from field-lookup-is-padding-byte-focus [OF fl i-lower True wf wf-sz,
simplified]
          eq-padding is-padding True i-lower l-take l-take'
        show ?thesis
          by (auto simp add: eq-padding-def add.commute is-padding-byte-def)
      next
        case False
        with fl i-t have i-t: i < size-td t by simp
        with sfx-eq i-lower False show ?thesis by simp
      qed
    qed
  qed
 qed
qed

lemma field-lookup-eq-padding-super-update-bs:
  assumes fl: field-lookup t f 0 = Some (s, n)
  assumes lxbs: length xbs = size-td t
assumes lbs: length bs = size-td s
assumes lbs': length bs' = size-td s
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows eq-padding (export-uinfo t) (super-update-bs bs xbs n) (super-update-bs bs'
xbs n) =
      eq-padding (export-uinfo s) bs bs'
proof −
  from lbs lbs' fl
  have pfx-eq: ⋀i. i < n ⟹ (super-update-bs bs xbs n) ! i = (super-update-bs bs'
xbs n) ! i
    by (metis (no-types, opaque-lifting) add-leD2 assms(2) field-lookup-offset-size'
        length-take less-le-trans min-def nth-append super-update-bs-def)
  from lbs lbs' fl
  have sfx-eq: ⋀i. n + size-td s ≤ i ⟹ i < size-td t ⟹ (super-update-bs bs xbs
n) ! i  = (super-update-bs bs' xbs n) ! i
  proof −
    fix i :: nat
    assume a1: n + size-td s ≤ i
    have n ≤ size-td t
```

by (*meson add-leD2 field-lookup-offset-size' fl*)
then show *super-update-bs bs xbs n ! i = super-update-bs bs' xbs n ! i*
using *a1 lxbs lbs lbs'* **by** (*smt* (*verit*) *Many-More.nat-min-simps*(*1*) *append.assoc*
length-append length-take not-less nth-append super-update-bs-def)
qed
from *lxbs lbs*
have *lsuper*: *length* (*super-update-bs bs xbs n*) = *size-td t*
by (*metis add.commute field-lookup-offset-size' fl length-super-update-bs*)
from *lxbs lbs'*
have *lsuper'*: *length* (*super-update-bs bs' xbs n*) = *size-td t*
by (*metis add.commute field-lookup-offset-size' fl length-super-update-bs*)
from *lxbs lbs lsuper*
have *bs*: (*take* (*size-td s*) (*drop n* (*super-update-bs bs xbs n*))) = *bs*
by (*metis* (*no-types*, *lifting*) *append-take-drop-id append-eq-conv-conj*
append-take-drop-id length-take super-update-bs-def)
from *lxbs lbs' lsuper'*
have *bs'*: (*take* (*size-td s*) (*drop n* (*super-update-bs bs' xbs n*))) = *bs'*
by (*metis* (*no-types*, *lifting*) *append-take-drop-id append-eq-conv-conj*
append-take-drop-id length-take super-update-bs-def)
from *field-lookup-eq-padding-focus-eq* [*OF fl lsuper lsuper' pfx-eq sfx-eq wf wf-sz*]
show *?thesis*
by (*simp add*: *bs bs'*)
qed

lemma *field-lookup-eq-upto-padding-focus*:
assumes *fl*: *field-lookup t f 0* = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-td t*
assumes *lbs'*: *length bs'* = *size-td t*
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
assumes *eq-upto-padding*: *eq-upto-padding* (*export-uinfo t*) *bs bs'*
shows *eq-upto-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
proof −
from *fl lbs* **have** *bound-s*: *size-td s* ≤ *length bs* − *n*
by (*metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size'*)
from *fl lbs'* **have** *bound-s'*: *size-td s* ≤ *length bs'* − *n*
by (*metis add.commute add-diff-cancel-left' diff-le-mono field-lookup-offset-size'*)
from *bound-s lbs*
have *l-take*: *length* (*take* (*size-td s*) (*drop n bs*)) = *size-td s*
by *simp*
from *bound-s' lbs'*
have *l-take'*: *length* (*take* (*size-td s*) (*drop n bs'*)) = *size-td s*
by *simp*

show *?thesis*
apply (*simp add*: *eq-upto-padding-def*, *safe*)

```
  proof −
    show size-td s ≤ length bs − n by (rule bound-s)
  next
    show size-td s ≤ length bs′ − n by (rule bound-s′)
  next
    fix i assume is-value: is-value-byte (export-uinfo s) i
    from is-value have i-s: i < size-td s
      by (simp add: is-value-byte-def)
    have n-i: n ≤ i + n by simp
    from i-s have i-n-bound: i + n < n + size-td s by simp
    show take (size-td s) (drop n bs) ! i =
         take (size-td s) (drop n bs′) ! i

      using l-take l-take′ is-value eq-upto-padding lbs lbs′
        field-lookup-is-value-byte-focus [OF fl n-i i-n-bound wf wf-sz, simplified]
      by (auto simp add: eq-upto-padding-def add.commute is-value-byte-def)
  qed
qed


lemma field-lookup-eq-upto-padding-focus-eq:
assumes fl: field-lookup t f 0 = Some (s, n)
assumes lbs: length bs = size-td t
assumes lbs′: length bs′ = size-td t
assumes pfx-eq: ⋀i. i < n ⟹ bs ! i = bs′ ! i
assumes sfx-eq: ⋀i. n + size-td s ≤ i ⟹ i < size-td t ⟹ bs ! i = bs′ ! i
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows eq-upto-padding (export-uinfo t) bs bs′ =
      eq-upto-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td s)
(drop n bs′))
proof −
  from fl lbs have bound-s: size-td s ≤ length bs − n
   by (metis add.commute add-diff-cancel-left′ diff-le-mono field-lookup-offset-size′)
  from fl lbs′ have bound-s′: size-td s ≤ length bs′ − n
   by (metis add.commute add-diff-cancel-left′ diff-le-mono field-lookup-offset-size′)
  from bound-s lbs
  have l-take: length (take (size-td s) (drop n bs)) = size-td s
    by simp
  from bound-s′ lbs′
  have l-take′: length (take (size-td s) (drop n bs′)) = size-td s
    by simp

  show ?thesis
  proof
    assume eq-upto-padding: eq-upto-padding (export-uinfo t) bs bs′
   show eq-upto-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td
s) (drop n bs′))
     by (rule field-lookup-eq-upto-padding-focus [OF fl lbs lbs′ wf wf-sz eq-upto-padding])
```

```
  next
   assume eq-upto-padding: eq-upto-padding (export-uinfo s) (take (size-td s) (drop
n bs)) (take (size-td s) (drop n bs'))
    show eq-upto-padding (export-uinfo t) bs bs'
      apply (simp add: eq-upto-padding-def, safe)
    proof −
      show length bs = size-td t by (rule lbs)
    next
      show length bs' = size-td t by (rule lbs')
    next
      fix i
      assume is-value: is-value-byte (export-uinfo t) i
      from is-value have i-t: i < size-td t
        by (simp add: is-value-byte-def)
      show bs ! i = bs' ! i
      proof (cases i < n)
        case True
        with pfx-eq show ?thesis by simp
      next
        case False
        hence i-lower: n ≤ i by simp
        show ?thesis
        proof (cases i < n + size-td s)
          case True
        from field-lookup-is-value-byte-focus [OF fl i-lower True wf wf-sz, simplified]
            eq-upto-padding is-value True i-lower l-take l-take'
          show ?thesis
            by (auto simp add: eq-upto-padding-def add.commute )
        next
          case False
          with fl i-t have i-t: i < size-td t by simp
          with sfx-eq i-lower False show ?thesis by simp
        qed
      qed
    qed
  qed
qed

lemma field-lookup-eq-upto-padding-super-update-bs:
  assumes fl: field-lookup t f 0 = Some (s, n)
  assumes lxbs: length xbs = size-td t
assumes lbs: length bs = size-td s
assumes lbs': length bs' = size-td s
assumes wf: wf-desc t
assumes wf-sz: wf-size-desc t
shows eq-upto-padding (export-uinfo t) (super-update-bs bs xbs n) (super-update-bs
bs' xbs n) =
      eq-upto-padding (export-uinfo s) bs bs'
proof −
```

from *lbs lbs′ fl*
have *pfx-eq*: $\bigwedge i.\ i < n \Longrightarrow$ (*super-update-bs bs xbs n*) ! *i* = (*super-update-bs bs′ xbs n*) ! *i*
by (*metis* (*no-types*, *opaque-lifting*) *add-leD2 assms*(*2*) *field-lookup-offset-size′ length-take less-le-trans min-def nth-append super-update-bs-def*)
from *lbs lbs′ fl*
have *sfx-eq*: $\bigwedge i.\ n$ + *size-td s* $\leq i \Longrightarrow i <$ *size-td t* $\Longrightarrow$ (*super-update-bs bs xbs n*) ! *i* = (*super-update-bs bs′ xbs n*) ! *i*
proof −
fix *i* :: *nat*
assume *a1*: *n* + *size-td s* ≤ *i*
have *n* ≤ *size-td t*
by (*meson add-leD2 field-lookup-offset-size′ fl*)
then show *super-update-bs bs xbs n* ! *i* = *super-update-bs bs′ xbs n* ! *i*
using *a1 lxbs lbs lbs′* **by** (*smt* (*verit*) *Many-More.nat-min-simps*(*1*) *append.assoc length-append length-take not-less nth-append super-update-bs-def*)
qed
from *lxbs lbs*
have *lsuper*: *length* (*super-update-bs bs xbs n*) = *size-td t*
by (*metis add.commute field-lookup-offset-size′ fl length-super-update-bs*)
from *lxbs lbs′*
have *lsuper′*: *length* (*super-update-bs bs′ xbs n*) = *size-td t*
by (*metis add.commute field-lookup-offset-size′ fl length-super-update-bs*)
from *lxbs lbs lsuper*
have *bs*: (*take* (*size-td s*) (*drop n* (*super-update-bs bs xbs n*))) = *bs*
by (*metis* (*no-types*, *lifting*) *append-take-drop-id append-eq-conv-conj append-take-drop-id length-take super-update-bs-def*)
from *lxbs lbs′ lsuper′*
have *bs′*: (*take* (*size-td s*) (*drop n* (*super-update-bs bs′ xbs n*))) = *bs′*
by (*metis* (*no-types*, *lifting*) *append-take-drop-id append-eq-conv-conj append-take-drop-id length-take super-update-bs-def*)
from *field-lookup-eq-upto-padding-focus-eq* [*OF fl lsuper lsuper′ pfx-eq sfx-eq wf wf-sz*]
show *?thesis*
by (*simp add*: *bs bs′*)
qed

lemma *field-lookup-wf-size-desc-pres*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
wf-size-desc t $\Longrightarrow$ *field-lookup t f n* = *Some* (*s*, *m*) $\Longrightarrow$ *wf-size-desc s*
wf-size-desc-struct st $\Longrightarrow$ *field-lookup-struct st f n* = *Some* (*s*, *m*) $\Longrightarrow$ *wf-size-desc s*
wf-size-desc-list ts $\Longrightarrow$ *field-lookup-list ts f n* = *Some* (*s*, *m*) $\Longrightarrow$ *wf-size-desc s*
wf-size-desc-tuple x $\Longrightarrow$ *field-lookup-tuple x f n* = *Some* (*s*, *m*) $\Longrightarrow$ *wf-size-desc s*

by (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *n s m f* **and** *n s m f* **and** *n s m f* **and** *n s m f*)
(*auto split*: *if-split-asm option.splits*)

lemma *field-lookup-update-ti-super-update-bs-conv*:
fixes *t*::('*a*,'*b*) *typ-info*
and *st*::('*a*,'*b*) *typ-info-struct*
and *ts*::('*a*,'*b*) *typ-info-tuple list*
and *x*::('*a*,'*b*) *typ-info-tuple*
shows
⟦*field-lookup t f m* = *Some* (*s*, *m* + *n*); *length bs* = *size-td s*; *length xbs* = *size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
update-ti s bs v =
update-ti t (*super-update-bs bs* (*access-ti t v xbs*) *n*) *v*

⟦*field-lookup-struct st f m* = *Some* (*s*, *m* + *n*); *length bs* = *size-td s*; *length xbs* = *size-td-struct st*;
wf-fd-struct st; *wf-desc-struct st*; *wf-size-desc-struct st*⟧ ⟹
update-ti s bs v =
update-ti-struct st (*super-update-bs bs* (*access-ti-struct st v xbs*) *n*) *v*

⟦*field-lookup-list ts f m* = *Some* (*s*, *m* + *n*); *length bs* = *size-td s*; *length xbs* = *size-td-list ts*;
wf-fd-list ts; *wf-desc-list ts*; *wf-size-desc-list ts*⟧ ⟹
update-ti s bs v =
update-ti-list ts (*super-update-bs bs* (*access-ti-list ts v xbs*) *n*) *v*

⟦*field-lookup-tuple x f m* = *Some* (*s*, *m* + *n*); *length bs* = *size-td s*; *length xbs* = *size-td-tuple x*;
wf-fd-tuple x; *wf-desc-tuple x*; *wf-size-desc-tuple x*⟧ ⟹
update-ti s bs v =
update-ti-tuple x (*super-update-bs bs* (*access-ti-tuple x v xbs*) *n*) *v*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m n xbs v* **and** *f m n xbs v* **and** *f m n xbs v* **and** *f m n xbs v*)
case (*TypDesc algn st nm*)
then show *?case*
by (*auto split*: *if-split-asm*)
(*metis TypDesc.prems*(*3*) *TypDesc.prems*(*4*) *access-ti.simps length-fa-ti super-update-bs-same-length*)
next
case (*TypScalar algn st d*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*

then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: *x* = *DTuple d nm y* **by** (*cases x*) *auto*
from *Cons-typ-desc.prems* **obtain**
lxbs: *length xbs* = *size-td-tuple x* + *size-td-list fs* **and**
lbs: *length bs* = *size-td s* **and**
wf-fd-x: *wf-fd-tuple* (*DTuple d nm y*) **and**
wf-desc-x: *wf-desc-tuple* (*DTuple d nm y*) **and**
wf-fd-fs: *wf-fd-list fs* **and**
wf-size-desc-x: *wf-size-desc-tuple* (*DTuple d nm y*) **and**
wf-size-desc-fs: *wf-size-desc-list fs* **and**
nm-notin: *nm* ∉ *dt-snd* ‘ *set fs* **and**
wf-desc-fs: *wf-desc-list fs* **and**
commutes: *fu-commutes* (*update-ti-t d*) (*update-ti-list-t fs*)
by (*auto simp add*: *x*)

from *wf-fd-cons-listD* [*OF wf-fd-fs*]
have *fd-cons-fs*: *fd-cons-list fs* **.**

from *wf-fd-x* **have** *fd-cons-d*: *fd-cons d* **by** (*simp add*: *x wf-fd-consD*)

from *lxbs*
have *lxbs-drop*: *length* (*drop* (*size-td-tuple x*) *xbs*) = *size-td-list fs*
by *simp*

from *lxbs*
have *lxbs-take*: *length* (*take* (*size-td-tuple x*) *xbs*) = *size-td-tuple* (*DTuple d nm y*)
by (*simp add*: *x*)

from *lxbs*
have *ldrop-xbs*: *length* (*drop* (*size-td d*) *xbs*) = *size-td-list fs*
by (*simp add*: *x*)

from *lxbs wf-fd-x*
have *eq1*: *length* (*access-ti d v* (*take* (*size-td d*) *xbs*)) = *size-td d*
by (*metis lxbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x*)

from *Cons-typ-desc.prems* **obtain** *f1 fxs*
where *f*: *f* = *f1*#*fxs*
by (*cases f*) *auto*

show *?case*
proof (*cases f1* = *nm*)
case *True*
show *?thesis*
proof (*cases field-lookup d fxs m*)

```
    case None
    from Cons-typ-desc.prems field-lookup-list-None [OF nm-notin]
    have False
      by (simp add: True None f x)
    thus ?thesis by simp
  next
    case (Some r)
    from Cons-typ-desc.prems have r: r = (s, m + n)
      by (simp add: x True Some f)
    hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
      by  (simp add: f True Some r)

    from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
    have wf-size-desc s .
    from wf-size-desc-gt(1)[OF this]
    have 0 < size-td s .


    with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
    have n-le: n < size-td d
      by auto


     have bound: size-td s + (m + n − m) ≤ size-td-tuple (DTuple d nm y) by
fact
    have bound1: n + length bs ≤ size-td d
      by (metis add.commute bound diff-add-inverse lbs size-td-tuple.simps)

    from fd-cons-fs  ldrop-xbs
    have upd-id: (update-ti-list fs (access-ti-list fs v (drop (size-td d) xbs)) v) = v
    apply (clarsimp simp add: fd-cons-list-def fd-cons-desc-def update-ti-list-t-def)
      apply (simp add: fd-cons-update-access-def fd-cons-length-def)
      done

     note hyp = Cons-typ-desc.hyps(1)[simplified x, OF fl lbs lxbs-take wf-fd-x
wf-desc-x wf-size-desc-x, where v =v]
    show ?thesis apply (simp add: x True r hyp)
    by (simp add: super-update-bs-append-drop-first eq1 bound1 super-update-bs-append-take-first
upd-id)
  qed
 next
  case False
  with Cons-typ-desc.prems
  have fl1: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
    by (clarsimp simp add: x f False)
  hence n-bound: size-td d ≤ n
    by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)
```

from *fl1*
have *fl*: *field-lookup-list fs* (*f1* # *fxs*) (*m* + *size-td d*) = *Some* (*s*, (*m* + *size-td d*) + (*n* − *size-td d*))
by (*metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD*)
from *n-bound*
have *take-eq*: (*take* (*size-td s*) (*drop* (*n* − *size-td d* + *size-td d*) *xbs*)) =
(*take* (*size-td s*) (*drop n xbs*))
by *simp*

from *fd-cons-d lxbs-take eq1*
have *upd-d-id*: *update-ti d* (*access-ti d v* (*take* (*size-td d*) *xbs*)) *v* = *v*
apply (*clarsimp simp add*: *fd-cons-def fd-cons-desc-def update-ti-t-def*)
apply (*simp add*: *fd-cons-update-access-def x*)
apply (*erule allE* [**where** *x*= *v*])
apply (*erule allE* [**where** *x*= (*take* (*size-td d*) *xbs*)])
apply (*simp add*: *eq1*)
done

from *ldrop-xbs fd-cons-fs*
have *lacc-fs*: *length* (*access-ti-list fs v* (*drop* (*size-td d*) *xbs*)) = *size-td-list fs*
by (*simp add*: *fd-cons-list-def fd-cons-desc-def fd-cons-length-def*)

from *td-set-field-lookup-rev''*(*3*) [*rule-format*] *fl*
have (*s*, (*m* + *size-td d*) + (*n* − *size-td d*)) ∈ *td-set-list fs* (*m* + *size-td d*)
by *blast*
from *td-set-offset-size''*(*3*)[*rule-format*, *OF this*, *simplified*]
have *bound*: *size-td s* + (*n* − *size-td d*) ≤ *size-td-list fs***.**

from *lacc-fs n-bound bound lbs*
have *lsuper*: *length* (*super-update-bs bs* (*access-ti-list fs v* (*drop* (*size-td d*) *xbs*)) (*n* − *size-td d*)) = *size-td-list fs*
by (*simp add*: *super-update-bs-length*)

from *commutes* **have**
commute: *update-ti d* (*access-ti d v* (*take* (*size-td d*) *xbs*))
(*update-ti-list fs* (*super-update-bs bs* (*access-ti-list fs v* (*drop* (*size-td d*) *xbs*)) (*n* − *size-td d*)) *v*) =
update-ti-list fs (*super-update-bs bs* (*access-ti-list fs v* (*drop* (*size-td d*) *xbs*)) (*n* − *size-td d*))
(*update-ti d* (*access-ti d v* (*take* (*size-td d*) *xbs*)) *v*)
apply (*simp add*: *fu-commutes-def*)
apply (*erule allE* [**where** *x* = *v*])
apply (*erule allE* [**where** *x* = *access-ti d v* (*take* (*size-td d*) *xbs*)])
apply (*erule allE* [**where** *x* = *super-update-bs bs* (*access-ti-list fs v* (*drop* (*size-td d*) *xbs*)) (*n* − *size-td d*)])
apply (*simp add*: *update-ti-list-t-def update-ti-t-def eq1 lsuper*)
done

note *hyp* = *Cons-typ-desc.hyps*(*2*)[*simplified x*, *OF fl lbs lxbs-drop wf-fd-fs wf-desc-fs wf-size-desc-fs*, *simplified x*, *simplified*]
show *?thesis*
apply (*simp add*: *x f eq1*)
apply (*simp add*: *super-update-bs-append-drop-second eq1 n-bound super-update-bs-append-take-second*)
apply (*simp add*: *commute*)
apply (*simp add*: *hyp*)
apply (*simp add*: *upd-d-id*)
done
qed
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** (*auto split*: *if-split-asm*)
qed

lemma *access-ti-super-update-bs-of-wf*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
⟦*field-lookup t f m* = *Some* (*s*, *m* + *n*); *length bs'* = *size-td s*; *length bs* = *size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t (*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs' bs n*) =
super-update-bs (*access-ti s w bs'*) (*access-ti t v bs*) *n*

⟦*field-lookup-struct st f m* = *Some* (*s*, *m* + *n*); *length bs'* = *size-td s*; *length bs* = *size-td-struct st*;
wf-fd-struct st; *wf-desc-struct st*; *wf-size-desc-struct st*⟧ ⟹
access-ti-struct st (*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs' bs n*) =
super-update-bs (*access-ti s w bs'*) (*access-ti-struct st v bs*) *n*

⟦*field-lookup-list ts f m* = *Some* (*s*, *m* + *n*); *length bs'* = *size-td s*; *length bs* = *size-td-list ts*;
wf-fd-list ts; *wf-desc-list ts*; *wf-size-desc-list ts*⟧ ⟹
access-ti-list ts (*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs' bs n*) =
super-update-bs (*access-ti s w bs'*) (*access-ti-list ts v bs*) *n*

⟦*field-lookup-tuple x f m* = *Some* (*s*, *m* + *n*); *length bs'* = *size-td s*; *length bs* = *size-td-tuple x*;
wf-fd-tuple x; *wf-desc-tuple x*; *wf-size-desc-tuple x*⟧ ⟹
access-ti-tuple x (*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs' bs n*) =
super-update-bs (*access-ti s w bs'*) (*access-ti-tuple x v bs*) *n*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m n bs* **and** *f m n bs* **and** *f m n bs* **and** *f m n bs*)
case (*TypDesc algn st nm*)
note *fd* = *wf-fd-field-lookupD*[*OF TypDesc*(*2*, *5*), *THEN wf-fd-consD*]
have *length bs* = *size-td s* ⟹

```
    access-ti s (update-ti s (access-ti0 s w) v) bs = access-ti s w bs for bs
    using fd
   unfolding fd-cons-def fd-cons-desc-def fd-cons-access-update-def fd-cons-update-access-def
   by simp (metis access-ti0 fd fd-cons-length-p length-replicate update-ti-update-ti-t)
  then show ?case
    using TypDesc(2−) length-fa-ti[OF ‹wf-fd (TypDesc algn st nm)›]
    by (auto split: if-splits
            simp: super-update-bs-same-length TypDesc(1))
next
  case (TypScalar sz algn d)
  then show ?case by auto
next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto
  from Cons-typ-desc.prems have
    lbs: length bs = size-td-tuple x + size-td-list fs and
    wf-fd-x: wf-fd-tuple (DTuple d nm y) and
    wf-desc-x: wf-desc-tuple (DTuple d nm y) and
    wf-fd-fs: wf-fd-list fs and
    wf-size-desc-x: wf-size-desc-tuple (DTuple d nm y) and
    wf-size-desc-fs: wf-size-desc-list fs and
    nm-notin: nm ∉ dt-snd ‘ set fs and
    wf-desc-fs: wf-desc-list fs and
    bs': length bs' = size-td s
    by (auto simp add: x)

  { fix bs bs' :: byte list and v
    assume bs: length bs = size-td d and bs': length bs' = size-td-list fs
    with ‹wf-fd-list (x # fs)› have x-fs:
      access-ti-list fs (update-ti-tuple-t x bs v) bs' = access-ti-list fs v bs'
      by (simp add: fa-fu-ind-def x)
    then have access-ti-list fs (update-ti d bs v) bs' = access-ti-list fs v bs'
      by (simp add: update-ti-tuple-t-def bs x) }
  note fs-x = this

  from lbs
  have lbs-drop: length (drop (size-td-tuple x) bs) = size-td-list fs
    by simp

  from lbs
  have lbs-take: length (take (size-td-tuple x) bs) = size-td-tuple (DTuple d nm y)
    by (simp add: x)
```

```
  from lbs wf-fd-x
  have eq1: length (access-ti d v (take (size-td d) bs)) = size-td d
    by (metis lbs-take length-fa-ti size-td-tuple.simps wf-fd-tuple.simps x)

  from Cons-typ-desc.prems obtain f1 fxs
    where f: f = f1#fxs
    by (cases f) auto

  have wf-s: wf-fd s
   using Cons-typ-desc.prems(1) Cons-typ-desc.prems(4) wf-fd-field-lookup(3) by
blast


  let ?v = update-ti s (access-ti_0 s w) v

  have len-acc-w[simp]: length (access-ti_0 s w) = size-td s
    using length-fa-ti[OF wf-s] by (simp add: access-ti_0-def)

  show ?case
  proof (cases f1 = nm)
    case True
    show ?thesis
    proof (cases field-lookup d fxs m)
      case None
      from Cons-typ-desc.prems field-lookup-list-None [OF nm-notin]
      have False
        by (simp add: True None f x)
      thus ?thesis by simp
    next
      case (Some r)
      from Cons-typ-desc.prems have r: r = (s, m + n)
        by (simp add: x True Some f)
      hence fl: field-lookup-tuple (DTuple d nm y) f m = Some (s, m + n)
        by  (simp add: f True Some r)
     from td-set-wf-size-desc(4)[OF wf-size-desc-x td-set-tuple-field-lookup-tupleD,
OF fl]
      have wf-size-desc s .
      from wf-size-desc-gt(1)[OF this]
      have 0 < size-td s .

      with td-set-tuple-offset-size-m [OF td-set-tuple-field-lookup-tupleD, OF fl]
      have n-le: n < size-td d
        by auto

       have bound: size-td s + (m + n - m) ≤ size-td-tuple (DTuple d nm y) by
fact
      from bound
      have take-eq: (take (size-td s) (drop n (take (size-td d) bs))) =
        (take (size-td s) (drop n bs))
```

by (*simp add*: *take-drop*)

have *take-eq1*: *take* (*size-td s*) (*drop n* (*take* (*size-td d*) *bs*)) = *take* (*size-td s*) (*drop n bs*)
using *take-eq* **by** *blast*

have *l-take-s*: *length* (*take* (*size-td s*) (*drop n bs*)) = *size-td s*
using *bound lbs-take* **by** *auto*

have *sz-s*: *size-td s* ≤ *length bs* − *n*
using *l-take-s* **by** *auto*

from *fl*
have *fd-cons-s*: *fd-cons s*
using *wf-fd-consD wf-fd-field-lookup*(*4*) *wf-fd-x* **by** *blast*

have *sz*: *size-td s* + *n* ≤ *size-td d length bs* = *size-td d* + *size-td-list fs*
using *lbs x bound* **by** *auto*
with *bs′* **have** *super-update-bs-simps*[*simp*]:
take (*size-td d*) (*super-update-bs bs′ bs n*) =
super-update-bs bs′ (*take* (*size-td d*) *bs*) *n*
drop (*size-td d*) (*super-update-bs bs′ bs n*) = *drop* (*size-td d*) *bs*
by (*auto simp*: *super-update-bs-def take-drop*)

have *hyp*:
access-ti d (*update-ti s* (*access-ti*$_0$ *s w*) *v*)
(*super-update-bs bs′* (*take* (*size-td d*) *bs*) *n*) =
super-update-bs (*access-ti s w bs′*) (*access-ti d v* (*take* (*size-td d*) *bs*)) *n*
using *Cons-typ-desc.hyps*(*1*) *fl lbs-take wf-fd-x wf-desc-x wf-size-desc-x bs′*
by (*auto simp*: *x*)

have *length-take-bs*: *length* (*take* (*size-td d*) *bs*) = *size-td d*
using *sz* **by** *auto*

have *fl-d-s*: *field-lookup d fxs m* = *Some* (*s*, *m* + *n*)
using *fl* **by** (*simp add*: *True f*)

have *wf-d*: *wf-fd d wf-desc d wf-size-desc d*
using *wf-fd-x wf-desc-x wf-size-desc-x* **by** *auto*
note *length-fa-ti*[*OF wf-d*(*1*), *simp*]

show *?thesis* **using** *lbs bound fd-cons-length*[*OF* ‹*fd-cons s*› *bs′*]
apply (*simp add*: *x True Some r eq1 hyp bs′ super-update-bs-append1*)
apply (*subst field-lookup-update-ti-super-update-bs-conv*(*1*)[*OF fl-d-s len-acc-w length-take-bs wf-d*])
apply (*subst fs-x*)
apply *simp-all*
done
qed

```
  next
    case False
    with Cons-typ-desc.prems
    have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) = Some (s, m + n)
      by (clarsimp simp add: x f False)
    hence n-bound: size-td d ≤ n
      by (meson field-lookup-offset-le(3) nat-add-left-cancel-le)

    have fl: field-lookup-list fs (f1 # fxs) (m + size-td d) =
        Some (s, (m + size-td d) + (n − size-td d))
      by (metis Nat.diff-cancel field-lookup-list-offset2 field-lookup-list-offsetD fl)

    have hyp:
      access-ti-list fs (update-ti s (access-ti$_0$ s w) v)
        (super-update-bs bs′ (drop (size-td d) bs) (n − size-td d)) =
      super-update-bs (access-ti s w bs′)
        (access-ti-list fs v (drop (size-td d) bs)) (n − size-td d)
      using Cons-typ-desc.hyps(2) fl bs′ lbs-drop wf-fd-fs wf-desc-fs wf-size-desc-fs
      by (simp add: x)

    have super-update-bs-simps[simp]:
      take (size-td d) (super-update-bs bs′ bs n) = take (size-td d) bs
      drop (size-td d) (super-update-bs bs′ bs n) =
        super-update-bs bs′ (drop (size-td d) bs) (n − size-td d)
      using n-bound lbs x by (simp-all add: super-update-bs-def drop-take)

    have fl-x: field-lookup-list (x # fs) [nm] 0 = Some (d, 0)
      by (simp add: x)
    have disj: disj-fn [nm] f
      using False f by (auto simp: disj-fn-def)

    from fa-fu-lookup-disj(3)[rule-format, OF fl-x Cons-typ-desc(3) disj ‹wf-fd-list
(x # fs)›]
    have acc-upd: length bs = size-td s ⟹ length bs′ = size-td d ⟹
      access-ti d (update-ti s bs v) bs′ = access-ti d v bs′ for bs bs′ v
      by (simp add: fa-fu-ind-def update-ti-t-def)

    from False n-bound lbs
    show ?thesis
      by (simp add: x f hyp eq1 nth-append super-update-bs-append2 acc-upd)
  qed
next
  case (DTuple-typ-desc d nm y)
  then show ?case apply (cases f) by (auto split: if-split-asm)
qed

lemma heap-update-list-same:
  assumes p-no-overflow: unat p + length bs ≤ addr-card
  assumes same-pfx: ⋀i. i < length bs ⟹ bs ! i = hp (p + word-of-nat i)
```

```
  shows  heap-update-list p bs hp = hp
  using p-no-overflow same-pfx
proof (induct bs arbitrary: p hp)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  from Cons.prems obtain
    no-overflow: Suc (unat p + length bs) ≤ addr-card and
    same: (⋀i. i < Suc (length bs) ⟹ (b # bs) ! i = hp (p + word-of-nat i))
    by clarsimp

  from no-overflow have no-overflow': unat (p + 1) + length bs ≤ addr-card
   by (metis (no-types, opaque-lifting) Suc-leD
       add.commute add-Suc-right add-leD1 comm-monoid-add-class.add-0 unatSuc
unat-0)

  {
    fix i
    assume bound: i < length bs
    from same [where i=Suc i, simplified, OF bound]
    have bs ! i = hp (p + (1 + word-of-nat i)) .
    then
    have bs ! i = hp (p + 1 + word-of-nat i)
      by (metis group-cancel.add1)
  } note same' = this


  from same [where i=0] have hp-eq: hp(p := b) = hp by simp
  note hyp = Cons.hyps [where p=p+1, OF no-overflow' same']
  then show ?case by (simp add: hp-eq)
qed

lemma heap-update-list-same-prefix:
  assumes p-no-overflow: unat p + length pfx + length bs ≤ addr-card
  assumes same-pfx: ⋀i. i < length pfx ⟹ pfx ! i = hp (p + word-of-nat i)
   shows  heap-update-list p (pfx @ bs) hp = heap-update-list (p + word-of-nat
(length pfx)) bs hp
  using p-no-overflow same-pfx
proof (induct pfx arbitrary: p bs hp)
  case Nil
  then show ?case by simp
next
  case (Cons x pfx)
  from Cons.prems obtain
    no-overflow: Suc (unat p + length pfx + length bs) ≤ addr-card and
    same: (⋀i. i < Suc (length pfx) ⟹ (x # pfx) ! i = hp (p + word-of-nat i))
    by clarsimp
```

from *no-overflow* **have** *no-overflow'*: *unat* (*p* + *1*) + *length pfx* + *length bs* ≤ *addr-card*
by (*metis* (*no-types*, *opaque-lifting*) *Suc-leD ab-semigroup-add-class.add-ac*(*1*) *add.commute add-Suc-right add-leD1 comm-monoid-add-class.add-0 unatSuc unat-0*)

{
fix *i*
assume *bound*: *i* < *length pfx*
from *same* [**where** *i*=*Suc i*, *simplified*, *OF bound*]
have *pfx* ! *i* = *hp* (*p* + (*1* + *word-of-nat i*)) **.**
then
have *pfx* ! *i* = *hp* (*p* + *1* + *word-of-nat i*)
by (*metis group-cancel.add1*)
} note *same'* = *this*

have *add-eq*: *p* + *1* + *word-of-nat* (*length pfx*) = *p* + (*1* + *word-of-nat* (*length pfx*))
using *add.assoc* **by** *blast*
from *same* [**where** *i*=*0*] **have** *hp-eq*: *hp*(*p* := *x*) = *hp* **by** *simp*
note *hyp* = *Cons.hyps* [**where** *p*=*p*+*1*, *OF no-overflow' same'*]
then show *?case* **by** (*simp add*: *hp-eq add-eq*)
qed

lemma *heap-update-list-same-suffix*:
assumes *p-no-overflow*: *unat p* + *length sfx* + *length bs* ≤ *addr-card*
assumes *same-sfx*: ⋀*i*. *i* < *length sfx* ⟹ *sfx* ! *i* = *hp* (*p* + *word-of-nat* (*length bs* + *i*))
shows *heap-update-list p* (*bs* @ *sfx*) *hp* = *heap-update-list p bs hp*
using *p-no-overflow same-sfx*
proof (*induct bs arbitrary*: *p sfx hp*)
case *Nil*
then show *?case* **by** (*simp add*: *heap-update-list-same*)
next
case (*Cons b bs*)
from *Cons.prems* **obtain** *no-overflow*: *Suc* (*unat p* + *length sfx* + *length bs*) ≤ *addr-card* **and**
same : ⋀*i*. *i* < *length sfx* ⟹ *sfx* ! *i* = *hp* (*p* + *word-of-nat* (*Suc* (*length bs*) + *i*))
by *clarsimp*

from *no-overflow* **have** *no-overflow'*: *unat* (*p* + *1*) + *length sfx* + *length bs* ≤ *addr-card*
by (*metis* (*no-types*, *opaque-lifting*) *Suc-leD ab-semigroup-add-class.add-ac*(*1*) *add.commute add-Suc-right add-leD1 comm-monoid-add-class.add-0 unatSuc unat-0*)

{

fix *i*
assume *i-bound*: *i* < *length sfx*
from *same* [*OF i-bound*] **have** *prem*: *sfx* ! *i* = *hp* (*p* + *word-of-nat* (*Suc* (*length bs*) + *i*)) **.**
from *i-bound no-overflow*
have *Suc* (*i* + *length bs*) < *addr-card*
by *simp*
then
have *neq*: *p* ≠ *p* + *1* + *word-of-nat* (*length bs* + *i*)
by (*metis* (*no-types*, *opaque-lifting*) *ab-semigroup-add-class.add-ac*(*1*) *add.commute add-cancel-right-right*
len-of-addr-card not-gr-zero of-nat-Suc unat-of-nat-eq unsigned-0 zero-less-Suc)

from *neq* [*symmetric*]
have *hp-eq*: (*hp*(*p* := *b*)) (*p* + *1* + *word-of-nat* (*length bs* + *i*)) = *hp* (*p* + *1* + *word-of-nat* (*length bs* + *i*))
by (*rule fun-upd-other*)
have *add-eq*: (*p* + *1* + *word-of-nat* (*length bs* + *i*)) = (*p* + *word-of-nat* (*Suc* (*length bs*) + *i*))
by *simp*

have *sfx* ! *i* = (*hp*(*p* := *b*)) (*p* + *1* + *word-of-nat* (*length bs* + *i*))
apply (*subst hp-eq*)
apply (*subst prem*)
apply (*simp only*: *add-eq*)
done
} note *same′* = *this*
note *hyp* = *Cons.hyps* [*OF no-overflow′*, **where** *hp*=(*hp*(*p* := *b*)), *OF same′*]
show *?case* **apply** (*simp add*: *hyp*) **by** (*simp add*: *fun-upd-def*)
qed

lemma *heap-update-list-same-prefix-suffix*:
assumes *p-no-overflow*: *unat p* + *length pfx* + *length bs* + *length sfx* ≤ *addr-card*
assumes *same-pfx*: ⋀*i*. *i* < *length pfx* ⟹ *pfx* ! *i* = *hp* (*p* + *word-of-nat i*)
assumes *same-sfx*: ⋀*i*. *i* < *length sfx* ⟹ *sfx* ! *i* = *hp* (*p* + *word-of-nat* (*length pfx* + *length bs* + *i*))
shows *heap-update-list p* (*pfx* @ *bs* @ *sfx*) *hp* = *heap-update-list* (*p* + *word-of-nat* (*length pfx*)) *bs hp*
proof −
from *p-no-overflow*
have *no-overflow-pfx*: *unat p* + *length pfx* + *length* (*bs* @ *sfx*) ≤ *addr-card* **by** *simp*

from *heap-update-list-same-prefix* [**where** *pfx*=*pfx* **and** *bs*=*bs*@*sfx*, *OF no-overflow-pfx same-pfx*]

have *heap-update-list p* (*pfx* @ *bs* @ *sfx*) *hp* =
heap-update-list (*p* + *word-of-nat* (*length pfx*)) (*bs* @ *sfx*) *hp* **.**

also

from *p-no-overflow* **have** *unat p + length pfx ≤ addr-card*
by *simp*
then
have *unat (p + word-of-nat (length pfx)) ≤ unat p + length pfx*
by (*simp add*: *unat-le-helper*)

with *p-no-overflow*
have *no-overflow-sfx*: *unat (p + word-of-nat (length pfx)) + length sfx + length bs ≤ addr-card*
by *simp*

from *same-sfx p-no-overflow* **have** *same-sfx'*: ⋀*i*. *i < length sfx ⟹*
sfx ! i = hp (p + word-of-nat (length pfx) + word-of-nat (length bs + i))
apply *simp*
by (*simp add*: *ab-semigroup-add-class.add-ac*(*1*))
from *heap-update-list-same-suffix* [**where** *p*=*p + word-of-nat (length pfx)* **and** *bs*=*bs* **and** *sfx* =*sfx* **and** *hp*=*hp*,
OF no-overflow-sfx same-sfx']
have *heap-update-list (p + word-of-nat (length pfx)) (bs @ sfx) hp =*
heap-update-list (p + word-of-nat (length pfx)) bs hp **.**
finally show *?thesis* **.**
qed

lemma *heap-update-list-super-update-bs-heap-list*:
assumes *p-no-overflow*: *unat p + m ≤ addr-card*
assumes *n-m*: *n + length bs ≤ m*
shows *heap-update-list (p + word-of-nat n) bs hp = heap-update-list p (super-update-bs bs (heap-list hp m p) n) hp*
proof −
have *heap-list-hp*: ⋀*i*. *i < m ⟹ (heap-list hp m p) ! i = hp (p + word-of-nat i)*
by (*rule heap-list-nth*)

obtain *pfx sfx* **where**
super: *super-update-bs bs (heap-list hp m p) n = pfx @ bs @ sfx* **and**
pfx: *pfx = take n (heap-list hp m p)* **and**
sfx: *sfx = drop (n + length bs) (heap-list hp m p)* **and**
lpfx: *length pfx = n* **and**
lsfx: *length sfx = m − n − length bs*
using *n-m*
by (*simp add*: *super-update-bs-def*)

have *bounds*: *unat p + length pfx + length bs + length sfx ≤ addr-card*
using *lpfx p-no-overflow n-m lsfx*
by *simp*

have *same-pfx*: ⋀*i*. *i < length pfx ⟹ pfx ! i = hp (p + word-of-nat i)*
using *heap-list-hp*

by (*simp add*: *pfx*)

have *same-sfx*: $\bigwedge i.\ i < length\ sfx \Longrightarrow sfx\ !\ i = hp\ (p + word\text{-}of\text{-}nat\ (length\ pfx + length\ bs + i))$
using *heap-list-hp*
by (*simp add*: *sfx lpfx*)
from *heap-update-list-same-prefix-suffix* [*OF bounds same-pfx same-sfx*]
show *?thesis*
by (*simp add*: *super lpfx*)
qed

lemma *append-take-dropI*: $xs = take\ (length\ xs)\ zs \Longrightarrow ys = drop\ (length\ xs)\ zs \Longrightarrow xs\ @\ ys = zs$
by (*metis append-take-drop-id*)

lemma *heap-list-heap-update-list-id*:
assumes *bound*: $n \leq addr\text{-}card$
assumes *lbs*: $length\ bs = n$
shows $(heap\text{-}list\ (heap\text{-}update\text{-}list\ a\ bs\ h)\ n\ a) = bs$
using *bound lbs*
proof (*induct n arbitrary*: *a bs h*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
apply (*cases bs*)
apply *clarsimp*
apply *clarsimp*
by (*metis* (*no-types*, *lifting*) *Suc-le-D Suc-le-eq add-cancel-right-right add-diff-cancel-left'*
fun-upd-same heap-update-list-value' linorder-not-less one-neq-zero plus-1-eq-Suc
unat-minus-abs unsigned-1)
qed

lemma *heap-update-list-nth-conv*:
$length\ bs \leq addr\text{-}card \Longrightarrow$
heap-update-list a bs h a' =
(*if* $(unat\ a' + (if\ a' < a\ then\ addr\text{-}card\ else\ 0) - unat\ a) < length\ bs\ then$
bs ! $(unat\ a' + (if\ a' < a\ then\ addr\text{-}card\ else\ 0) - unat\ a)$
else
h a')
proof (*induct bs arbitrary*: *a h*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons b bs*)
from *Cons.prems* **have** *bound*: $Suc\ (length\ bs) \leq addr\text{-}card$ **by** *simp*

```
  hence bound': length bs ≤ addr-card by simp
  note hyp = Cons.hyps [OF this, of a+1  (h(a := b))]
  show ?case
  proof (cases a' < a)
    case True
    show ?thesis apply (simp add: hyp True)
      using True
     by (smt (verit) Nat.add-diff-assoc One-nat-def Suc-diff-Suc add-diff-cancel-left'
          cancel-comm-monoid-add-class.diff-cancel diff-add diff-diff-left
          diff-right-commute fun-upd-apply le-add-diff len-of-addr-card not-less-eq
          not-less-eq-eq nth-Cons-pos of-nat-add order-less-le trans-less-add2
          un-ui-le unat-0 unat-sub-if' unsigned-greater-eq unsigned-less word-less-def
          word-overflow-unat word-unat.Rep-inverse zero-less-diff)

  next
    case False
    show ?thesis apply (simp add: hyp False)
      using False bound'
      by (smt (verit) Nat.add-diff-assoc One-nat-def Suc-eq-plus1 Suc-le-eq
          add-diff-cancel-left bound cancel-comm-monoid-add-class.diff-cancel
        diff-zero fun-upd-def le-eq-less-or-eq len-of-addr-card linorder-not-le nth-Cons-0
          nth-Cons-Suc order-less-le plus-1-eq-Suc trans-le-add2 unatSuc2 unat-1
          unat-add-lem unat-mono unsigned-0 unsigned-less)
  qed
qed

lemma heap-update-list-overwrite-all-nth-conv:
  assumes lbs: length bs = addr-card
  shows heap-update-list a bs h a' =
        bs ! (unat a' + (if a' < a then addr-card else 0)  − unat a)
proof −
  from lbs have leq: length bs ≤ addr-card by simp
  show ?thesis
    using lbs
    apply (simp add: heap-update-list-nth-conv [OF leq])
   by (metis len-of-addr-card linorder-not-less unat-sub-if' unsigned-less word-less-nat-alt)
qed



lemma heap-update-list-overwrite':
  assumes bound: length bs ≤ addr-card
  assumes len-eq: length bs = length bs'
  shows (heap-update-list a bs (heap-update-list a bs' h)) = (heap-update-list a bs
h)
  apply (rule ext)
  using bound len-eq
  apply (simp add: heap-update-list-nth-conv )
  done
```

lemma *heap-list-tail-addr-card*:
assumes *len*: *length ys* = *addr-card*
shows *heap-update-list p* (*xs* @ *ys*) *h* = *heap-update-list* (*p* + *word-of-nat* (*length xs*)) *ys h*
proof −
from *len* **have** *len'*: *length ys* ≤ *addr-card* **by** *simp*
show *?thesis*
apply (*simp add*: *heap-update-list-concat-unfold*)
apply (*rule ext*)
apply (*simp add*: *heap-update-list-nth-conv* [*OF len'*])
using *len*
by (*metis len-of-addr-card linorder-not-less unat-sub-if' unsigned-less word-less-nat-alt*)
qed

lemma *heap-update-list-overwrite*:
assumes *leq*: *length bs* = *length bs'*
shows *heap-update-list p bs* (*heap-update-list p bs' h*) = *heap-update-list p bs h*
proof (*cases length bs* ≤ *addr-card*)
case *True*
from *heap-update-list-overwrite'* [*OF True leq*] **show** *?thesis* **by** *simp*
next
case *False*
from *False* **obtain** *xs ys* **where** *bs*: *bs*=*xs*@*ys* **and** *lys*: *length ys* = *addr-card*
by (*metis append-take-drop-id diff-diff-cancel length-drop linorder-not-less order-less-le*)
from *False leq* **obtain** *xs' ys'* **where** *bs'*: *bs'*=*xs'*@*ys'* **and** *lys'*: *length ys'* = *addr-card*
by (*metis append-take-drop-id diff-diff-cancel length-drop linorder-not-less order-less-le*)
from *lys* **have** *leq-ys*: *length ys* ≤ *addr-card* **by** *simp*
from *lys'* **have** *leq-ys'*: *length ys'* ≤ *addr-card* **by** *simp*
note *eqs* = *heap-list-tail-addr-card* [*OF lys*, **where** *xs* = *xs*]
heap-list-tail-addr-card [*OF lys'*, **where** *xs* = *xs'*]
heap-update-list-overwrite-all-nth-conv [*OF lys*]
heap-update-list-overwrite-all-nth-conv [*OF lys'*]

show *?thesis*
apply (*rule ext*)
apply (*simp add*: *bs bs' eqs*)
done
qed

context *xmem-type*
begin

lemma *xmem-type-is-value-byte-access-ti-update-ti-cancel*:
assumes *i-bound*: *i* < *size-of TYPE*('*a*)

assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'*= *size-of TYPE*('*a*)
assumes *is-value*: *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
shows *access-ti* (*typ-info-t TYPE*('*a*)) (*update-ti* (*typ-info-t TYPE*('*a*)) *bs v*) *bs'*
! *i* = *bs* ! *i*
using *is-value-byte-access-ti-update-ti-cancel* [*OF - - - - - i-bound* [*simplified size-of-def*]
lbs [*simplified size-of-def*] *lbs'* [*simplified size-of-def*] *is-value* [*simplified typ-uinfo-t-def*]]
by (*simp add*: *local.wf-component-descs local.wf-field-descs wf-fdp-fd*(*1*) *wf-lf-fdp*)

lemma *xmem-type-is-value-byte-access-ti-id*:
assumes *i-bound*: *i* < *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *is-value*: *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
shows *access-ti* (*typ-info-t TYPE*('*a*)) *v* (*bs*[*i*:=*b*]) = *access-ti* (*typ-info-t TYPE*('*a*))
v bs
using *is-value-byte-access-ti-id* [*OF - - - - - i-bound* [*simplified size-of-def*]
lbs [*simplified size-of-def*] *is-value* [*simplified typ-uinfo-t-def*]]
by (*simp add*: *local.wf-component-descs local.wf-field-descs wf-fdp-fd*(*1*) *wf-lf-fdp*)

lemma *is-padding-byte-to-lense*: *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
⟹ *lense.is-padding-byte i*
by (*metis is-padding-byte-def local.lense.complement local.lense.is-padding-byte-def*
local.lense.is-value-byte-update-depends local.xmem-type-is-padding-byte-update-ti-id
size-of-def typ-uinfo-size)

lemma *is-value-byte-to-lense*: *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
⟹ *lense.is-value-byte i*
using *xmem-type-is-padding-byte-access-ti*
xmem-type-is-padding-byte-update-ti-id xmem-type-is-value-byte-access-ti-update-ti-cancel
using *is-value-byte-def local.lense.is-value-byteI local.size-of-def*
xmem-type-is-value-byte-access-ti-id **by** *auto*

lemma *is-padding-byte-from-lense*:
assumes *padding*: *lense.is-padding-byte i*
shows *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
proof −
from *padding* **have** *i-sz*: *i* < *size-of* (*TYPE*('*a*))
using *local.lense.is-padding-byte-def* **by** *blast*
show *?thesis*
proof (*cases is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) *i*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
with *complement-tu-padding* [*OF i-sz* [*simplified size-of-def*, *simplified typ-uinfo-size*
[*symmetric*]]] **show** *?thesis*
using *i-sz is-value-byte-to-lense local.lense.complement padding* **by** *blast*

qed
qed

lemma *is-padding-byte-lense-conv*: *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) *i* = *lense.is-padding-byte i*
using *is-padding-byte-from-lense is-padding-byte-to-lense* **by** *blast*

lemma *is-value-byte-from-lense*:
assumes *is-value*: *lense.is-value-byte i*
shows *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
proof −
from *is-value* **have** *i-sz*: *i* < *size-of* (*TYPE*('*a*))
using *local.lense.is-value-byte-def* **by** *blast*
show *?thesis*
proof (*cases is-value-byte* (*typ-uinfo-t TYPE*('*a*)) *i*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
with *complement-tu-padding* [*OF i-sz* [*simplified size-of-def*, *simplified typ-uinfo-size* [*symmetric*]]] **show** *?thesis*
using *assms i-sz is-padding-byte-to-lense local.lense.complement* **by** *blast*
qed
qed

lemma *is-value-byte-lense-conv*: *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) *i* = *lense.is-value-byte i*
using *is-value-byte-from-lense is-value-byte-to-lense* **by** *blast*

lemma *eq-padding-lense-conv*: *eq-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs*′ = *lense.eq-padding bs bs*′
apply (*simp add*: *lense.eq-padding-is-padding-byte-conv eq-padding-def*)
by (*simp add*: *is-padding-byte-lense-conv size-of-def*)

lemma *eq-upto-padding-lense-conv*: *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs*′
= *lense.eq-upto-padding bs bs*′
apply (*simp add*: *lense.eq-upto-padding-is-value-byte-conv eq-upto-padding-def*)
by (*simp add*: *is-value-byte-lense-conv size-of-def*)

lemma *lense-eq-upto-padding-from-bytes-eq*:
assumes *eq-upto-padding*: *lense.eq-upto-padding bs bs*′
shows ((*from-bytes bs*)::′*a*) = *from-bytes bs*′
using *eq-upto-padding*
by (*metis field-desc.select-convs*(*2*) *field-desc-def from-bytes-def local.field-sz-size-of local.field-update-update-ti local.xmem-type-wf-field-desc.eq-upto-padding-upd padding-base.eq-upto-padding-length-eq update-ti-t-def*)

lemma *eq-upto-padding-from-bytes-eq*:

assumes *eq-upto-padding*: *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs'*
shows ((*from-bytes bs*)::'*a*) = *from-bytes bs'*
using *lense-eq-upto-padding-from-bytes-eq eq-upto-padding-lense-conv eq-upto-padding*
by (*simp add*: *typ-uinfo-t-def*)

lemma *lense-eq-padding-to-bytes-eq*:
assumes *eq-padding*: *lense.eq-padding bs bs'*
shows (*to-bytes* (*v*::'*a*) *bs*) = *to-bytes v bs'*
proof −
from *lense.eq-padding-acc* [*OF eq-padding*]
show *?thesis*
by (*simp add*: *to-bytes-def*)
qed

lemma *eq-padding-to-bytes-eq*:
assumes *eq-padding*: *eq-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs'*
shows (*to-bytes* (*v*::'*a*) *bs*) = *to-bytes v bs'*
using *lense-eq-padding-to-bytes-eq eq-padding-lense-conv eq-padding*
by (*simp add*: *typ-uinfo-t-def*)

lemma *eq-padding-to-bytes*:
length bs = *size-of TYPE*('*a*) ⟹ *eq-padding* (*typ-uinfo-t TYPE*('*a*)) (*to-bytes* (*v*::'*a*) *bs*) *bs*
by (*simp add*: *eq-padding-lense-conv local.lense.field-access-eq-padding1 to-bytes-def*)

lemma *lense-eq-padding-to-bytes*:
length bs = *size-of TYPE*('*a*) ⟹ *lense.eq-padding* (*to-bytes* (*v*::'*a*) *bs*) *bs*
using *eq-padding-to-bytes eq-padding-lense-conv*
by *simp*

lemma *heap-list-heap-update-id*:
fixes *p*::'*a ptr*
shows (*heap-list*
(*heap-update-list* (*ptr-val p*)
(*to-bytes* (*u*::'*a*) (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))) *h*)
(*size-of TYPE*('*a*)) (*ptr-val p*)) =
(*to-bytes u* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)))
proof −
have *bound*: *size-of TYPE*('*a*) ≤ *addr-card*
using *max-size nless-le* **by** *blast*
have *lbs*: *length* (*to-bytes u* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))) = *size-of TYPE*('*a*)
by (*simp add*: *local.lense.access-result-size to-bytes-def*)
note *eq1* = *heap-list-heap-update-list-id* [*OF bound lbs*]
show *?thesis*
by (*simp add*: *eq1*)
qed

lemma *heap-update-collapse*:
fixes *p*::*'a ptr*
shows *heap-update p v* (*heap-update p u h*) = *heap-update p v h*
proof −

have *lv*: *length* (*to-bytes v* (*to-bytes u* (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*)))) =
size-of TYPE(*'a*)
by (*simp add*: *local.lense.access-result-size to-bytes-def*)

have *lu*: *length* (*to-bytes u* (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*))) = *size-of TYPE*(*'a*)
by (*simp add*: *local.lense.access-result-size to-bytes-def*)
have *leq*: *length* (*to-bytes v* (*to-bytes u* (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*)))) =
length (*to-bytes u* (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*)))
by (*simp add*: *lv lu*)
have *eq*:
(*to-bytes v* (*to-bytes u* (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*)))) =
(*to-bytes v* (*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val p*)))
by (*metis eq-padding-lense-conv eq-padding-to-bytes-eq heap-list-length*
local.lense.field-access-eq-padding1 to-bytes-def)
show *?thesis*
apply (*simp add*: *heap-update-def*)
apply (*simp add*: *heap-list-heap-update-id* [*OF*])
apply (*subst heap-update-list-overwrite* [*OF leq*])
apply (*simp add*: *eq*)
done
qed

lemma *heap-update-padding-collapse*:
fixes *p*::*'a ptr*
assumes *lbs*: *length bs* = *size-of TYPE*(*'a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*(*'a*)
shows *heap-update-padding p v bs* (*heap-update-padding p u bs' h*) = *heap-update-padding p v bs h*
apply (*simp add*: *heap-update-padding-def*)
using *lbs lbs' heap-update-list-overwrite*
by *auto*

lemma *heap-update-padding-heap-update-collapse*:
fixes *p*::*'a ptr*
assumes *lbs*: *length bs* = *size-of TYPE*(*'a*)
shows *heap-update-padding p v bs* (*heap-update p u h*) = *heap-update-padding p v bs h*
by (*simp add*: *heap-update-heap-update-padding-conv heap-update-padding-collapse* [*OF lbs*])

lemma *to-bytes-from-bytes-id*:

length bs = size-of TYPE('a) ⟹ to-bytes ((from-bytes bs)::'a) bs = bs
by (*metis* (*no-types*, *lifting*) *field-desc.select-convs*(*2*) *field-desc-def from-bytes-def lense-eq-padding-to-bytes*
local.field-sz-size-of local.field-sz-size-td local.lense.eq-upto-paddingI
local.lense.padding-eq-complement local.lense.update-access local.upd
padding-base.eq-padding-length1 to-bytes-def update-ti-update-ti-t)

lemma *to-bytes-heap-list-id*:
to-bytes ((from-bytes (heap-list h (size-of TYPE('a)) a))::'a)
(heap-list h (size-of TYPE('a)) a) =
heap-list h (size-of TYPE('a)) a
by (*simp add*: *to-bytes-from-bytes-id*)

lemma *heap-update-id*:
fixes *p*::'*a ptr*
shows *heap-update p (h-val h p) h = h*
by (*simp add*: *heap-update-def h-val-def to-bytes-heap-list-id heap-update-list-id'*)

context
fixes *s f n*
assumes *fl*: *field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)*
begin
interpretation *flense*: *padding-lense access-ti s update-ti s size-td s*
using *field-lookup-padding-lense* [*OF fl*] **.**

private lemma *n-t*: *n < size-of TYPE('a)*
using *fl*
by (*metis add-diff-cancel-right' add-leD2 cancel-comm-monoid-add-class.diff-cancel field-lookup-offset-size'*
field-lookup-wf-size-desc-gt
local.field-sz-size-of local.field-sz-size-td local.wf-size-desc nat-less-le not-add-less2)

private lemma *wf-desc-s*: *wf-desc s*
using *fl field-lookup-wf-desc-pres*(*1*) *local.wf-desc* **by** *blast*

private lemma *wf-size-desc-s*: *wf-size-desc s*
using *fl field-lookup-wf-size-desc-pres*(*1*) *local.wf-size-desc* **by** *blast*

private lemma *wf-field-descs-s*: *wf-field-descs (set (field-descs s))*
using *fl local.field-lookup-wf-field-descs* **by** *blast*

private lemma *wf-component-descs-s*: *wf-component-descs s*
using *fl local.field-lookup-wf-component-descs* **by** *blast*

private lemma *wf-fd-s*: *wf-fd s*
using *fl local.wf-desc local.wf-lf wf-fd-field-lookupD wf-fdp-fd*(*1*) *wf-lf-fdp*
by *blast*

lemma *xmem-type-field-lookup-is-padding-byte-focus*:
assumes *i-lower*: $n \leq i$
assumes *i-upper* :$i < n + size\text{-}td\ s$
shows *is-padding-byte* (*typ-uinfo-t* $TYPE('a)$) i = *is-padding-byte* (*export-uinfo* s) $(i - n)$
using *field-lookup-is-padding-byte-focus* [*OF fl i-lower i-upper*]
by (*simp add*: *typ-uinfo-t-def*)

lemma *xmem-type-field-lookup-is-padding-byte-focus-rev1*:
assumes *is-padding*: *is-padding-byte* (*export-uinfo s*) *i*
shows *is-padding-byte* (*typ-uinfo-t* $TYPE('a)$) $(i + n)$
proof −
from *is-padding* **have** *i-s*: $i < size\text{-}td\ s$
using *is-padding-byte-def* **by** *simp*
have *lower*: $n \leq i + n$ **by** *simp*
from *i-s* **have** *upper*: $i + n < n + size\text{-}td\ s$ **by** *simp*

from *xmem-type-field-lookup-is-padding-byte-focus* [*OF lower upper*] *is-padding*
show *?thesis* **by** *simp*
qed

lemma *xmem-type-field-lookup-is-padding-byte-focus-rev2*:
assumes *is-padding*: *is-padding-byte* (*typ-uinfo-t* $TYPE('a)$) $(i + n)$
assumes *i-bound*: $i < size\text{-}td\ s$
shows *is-padding-byte* (*export-uinfo s*) *i*
proof −
have *lower*: $n \leq i + n$ **by** *simp*
from *i-bound* **have** *upper*: $i + n < n + size\text{-}td\ s$ **by** *simp*
from *xmem-type-field-lookup-is-padding-byte-focus* [*OF lower upper*] *is-padding*
show *?thesis*
by *simp*
qed

lemma *xmem-type-field-lookup-lense-is-padding-byte-focus-rev1*:
assumes *is-padding*: *flense.is-padding-byte i*
shows *lense.is-padding-byte* $(i + n)$
proof −
from *is-padding* **have** *i-s*: $i < size\text{-}td\ s$
using *flense.is-padding-byte-def* **by** *simp*
have *lower*: $n \leq i + n$ **by** *simp*
from *i-s* **have** *upper*: $i + n < n + size\text{-}td\ s$ **by** *simp*

from *field-lookup-is-padding-byte-inner-to-outer* [*OF fl lower upper*] *is-padding*
show *?thesis* **by** (*simp add*: *size-of-def*)
qed

lemma *xmem-type-field-lookup-lense-is-padding-byte-focus-rev2*:

assumes *is-padding*: *lense.is-padding-byte* $(i + n)$
assumes *i-bound*: $i <$ *size-td s*
shows *flense.is-padding-byte* i
proof −
have *lower*: $n \leq i + n$ **by** *simp*
from *i-bound* **have** *upper*: $i + n < n +$ *size-td s* **by** *simp*
from *field-lookup-is-padding-byte-outer-to-inner* [*OF fl lower upper*] *is-padding*
show *?thesis*
by (*simp add*: *size-of-def*)
qed

lemma *xmem-type-field-lookup-is-value-byte-focus*:
assumes *i-lower*: $n \leq i$
assumes *i-upper* :$i < n +$ *size-td s*
shows *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) $i =$ *is-value-byte* (*export-uinfo s*) (i − n)
using *field-lookup-is-value-byte-focus* [*OF fl i-lower i-upper*]
by (*simp add*: *typ-uinfo-t-def*)

lemma *xmem-type-field-lookup-is-value-byte-focus-rev1*:
assumes *is-value*: *is-value-byte* (*export-uinfo s*) i
shows *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) $(i + n)$
proof −
from *is-value* **have** *i-s*: $i <$ *size-td s*
using *is-value-byte-def* **by** *simp*
have *lower*: $n \leq i + n$ **by** *simp*
from *i-s* **have** *upper*: $i + n < n +$ *size-td s* **by** *simp*

from *xmem-type-field-lookup-is-value-byte-focus* [*OF lower upper*] *is-value*
show *?thesis* **by** *simp*
qed

lemma *xmem-type-field-lookup-is-value-byte-focus-rev2*:
assumes *is-value*: *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) $(i + n)$
assumes *i-bound*: $i <$ *size-td s*
shows *is-value-byte* (*export-uinfo s*) i
proof −
have *lower*: $n \leq i + n$ **by** *simp*
from *i-bound* **have** *upper*: $i + n < n +$ *size-td s* **by** *simp*
from *xmem-type-field-lookup-is-value-byte-focus* [*OF lower upper*] *is-value*
show *?thesis*
by *simp*
qed

lemma *xmem-type-field-lookup-lense-is-value-byte-focus-rev1*:
assumes *is-value*: *flense.is-value-byte i*
shows *lense.is-value-byte* $(i + n)$
proof −

```
  from is-value have i-s: i < size-td s
    using flense.is-value-byte-def by simp
  have lower: n ≤ i + n by simp
  from i-s have upper: i + n < n + size-td s by simp

  from field-lookup-is-value-byte-inner-to-outer [OF fl lower upper] is-value
  show ?thesis by (simp add: size-of-def)
qed

lemma xmem-type-field-lookup-lense-is-value-byte-focus-rev2:
  assumes is-value: lense.is-value-byte (i + n)
  assumes i-bound: i < size-td s
  shows flense.is-value-byte i
  proof −
    have lower: n ≤ i + n by simp
    from i-bound have upper: i + n < n + size-td s by simp
    from field-lookup-is-value-byte-outer-to-inner [OF fl lower upper] is-value
    show ?thesis
      by (simp add: size-of-def)
  qed


lemma field-lookup-is-padding-byte-access-ti:
  assumes i-bound: i < size-td s
  assumes lbs: length bs = size-td s
  assumes is-padding: is-padding-byte (export-uinfo s) i
  shows access-ti s v bs ! i = bs ! i
  using is-padding-byte-access-ti [OF wf-desc-s wf-size-desc-s wf-field-descs-s
      wf-component-descs-s wf-fd-s i-bound lbs is-padding] .

lemma field-lookup-is-padding-byte-update-ti-id:
  assumes i-bound: i < size-td s
  assumes lbs: length bs = size-td s
  assumes is-padding: is-padding-byte (export-uinfo s) i
  shows update-ti s (bs[i := b]) v = update-ti s bs v
  using is-padding-byte-update-ti-id [OF wf-desc-s wf-size-desc-s wf-field-descs-s
      wf-component-descs-s wf-fd-s i-bound lbs is-padding] .

lemma field-lookup-is-value-byte-access-ti-update-ti-cancel:
  assumes i-bound: i < size-td s
  assumes lbs: length bs = size-td s
  assumes lbs': length bs' = size-td s
  assumes is-value: is-value-byte (export-uinfo s) i
  shows access-ti s (update-ti s bs v) bs' ! i = bs ! i
using is-value-byte-access-ti-update-ti-cancel [OF wf-desc-s wf-size-desc-s wf-field-descs-s
      wf-component-descs-s wf-fd-s i-bound lbs lbs' is-value] .

lemma field-lookup-is-value-byte-access-ti-id:
  assumes i-bound: i < size-td s
```

assumes *lbs*: *length bs* = *size-td s*
assumes *is-value*: *is-value-byte* (*export-uinfo s*) *i*
shows *access-ti s v* (*bs*[*i* := *b*]) = *access-ti s v bs*
using *is-value-byte-access-ti-id* [*OF wf-desc-s wf-size-desc-s wf-field-descs-s wf-component-descs-s wf-fd-s i-bound lbs is-value*] **.**

lemma *field-lookup-is-padding-byte-to-lense*:
assumes *is-padding*: *is-padding-byte* (*export-uinfo s*) *i*
shows *flense.is-padding-byte i*
proof (*rule flense.is-padding-byteI*)
from *is-padding* **show** *i* < *size-td s*
using *is-padding-byte-def* **by** *simp*
next
fix *v*::′*a* **and** *bs*::*byte list*
assume *i-bound*: *i* < *size-td s*
assume *lbs*: *length bs* = *size-td s*
show *access-ti s v bs* ! *i* = *bs* ! *i*
by (*rule field-lookup-is-padding-byte-access-ti* [*OF i-bound lbs is-padding*])
next
fix *v*::′*a* **and** *bs*::*byte list* **and** *b*::*byte*
assume *i-bound*: *i* < *size-td s*
assume *lbs*: *length bs* = *size-td s*
show *update-ti s bs v* = *update-ti s* (*bs*[*i* := *b*]) *v*
using *field-lookup-is-padding-byte-update-ti-id* [*OF i-bound lbs is-padding*] **by** *simp*
qed

lemma *field-lookup-is-padding-byte-from-lense*:
assumes *is-padding*: *flense.is-padding-byte i*
shows *is-padding-byte* (*export-uinfo s*) *i*
proof −
from *is-padding* **have** *i-bound*: *i* < *size-td s* **by** (*simp add*: *flense.is-padding-byte-def*)
from *xmem-type-field-lookup-lense-is-padding-byte-focus-rev1* [*OF is-padding*]
have *lense.is-padding-byte* (*i* + *n*)**.**
from *is-padding-byte-from-lense* [*OF this*]
have *is-padding-byte* (*typ-uinfo-t TYPE*(′*a*)) (*i* + *n*)**.**
with *xmem-type-field-lookup-is-padding-byte-focus* [**where** *i*=*i* + *n*] *i-bound*
show *?thesis* **by** *simp*
qed

lemma *field-lookup-is-padding-byte-lense-conv*:
is-padding-byte (*export-uinfo s*) *i* ⟷ *flense.is-padding-byte i*
using *field-lookup-is-padding-byte-from-lense field-lookup-is-padding-byte-to-lense*
by *blast*

lemma *field-lookup-is-value-byte-from-lense*:
assumes *is-value*: *flense.is-value-byte i*
shows *is-value-byte* (*export-uinfo s*) *i*
proof −

from *is-value* **have** *i-bound*: $i < size\text{-}td\ s$ **by** (*simp add*: *flense.is-value-byte-def*)
from *xmem-type-field-lookup-lense-is-value-byte-focus-rev1* [*OF is-value*]
have *lense.is-value-byte* $(i + n)$**.**
from *is-value-byte-from-lense* [*OF this*]
have *is-value-byte* (*typ-uinfo-t* $TYPE('a)$) $(i + n)$**.**
with *xmem-type-field-lookup-is-value-byte-focus* [**where** $i{=}i + n$] *i-bound*
show *?thesis* **by** *simp*
qed

lemma *field-lookup-is-value-byte-to-lense*:
assumes *is-value*: *is-value-byte* (*export-uinfo s*) *i*
shows *flense.is-value-byte i*
proof (*rule flense.is-value-byteI*)
from *is-value* **show** $i < size\text{-}td\ s$
using *is-value-byte-def* **by** *simp*
next
fix $v{::}'a$ **and** *bs*::*byte list* **and** bs'::*byte list*
assume *i-bound*: $i < size\text{-}td\ s$
assume *lbs*: $length\ bs = size\text{-}td\ s$
assume *lbs'*: $length\ bs' = size\text{-}td\ s$
show *access-ti s* (*update-ti s bs v*) $bs'\ !\ i = bs\ !\ i$
by (*rule field-lookup-is-value-byte-access-ti-update-ti-cancel* [*OF i-bound lbs lbs' is-value*])
next
fix $v{::}'a$ **and** *bs*::*byte list* **and** *b*::*byte*
assume *i-bound*: $i < size\text{-}td\ s$
assume *lbs*: $length\ bs = size\text{-}td\ s$
show *access-ti s v bs* $=$ *access-ti s v* $(bs[i := b])$
using *field-lookup-is-value-byte-access-ti-id* [*OF i-bound lbs is-value*] **by** *simp*
qed

lemma *field-lookup-is-value-byte-lense-conv*:
is-value-byte (*export-uinfo s*) $i \longleftrightarrow$ *flense.is-value-byte i*
using *field-lookup-is-value-byte-from-lense field-lookup-is-value-byte-to-lense*
by *blast*

lemma *field-lookup-eq-padding-lense-conv*: *eq-padding* (*export-uinfo s*) *bs* $bs' \longleftrightarrow$ *flense.eq-padding bs* bs'
using *field-lookup-is-padding-byte-lense-conv*
by (*simp add*: *eq-padding-def flense.eq-padding-is-padding-byte-conv*)

lemma *field-lookup-eq-upto-padding-lense-conv*: *eq-upto-padding* (*export-uinfo s*) *bs* $bs' \longleftrightarrow$ *flense.eq-upto-padding bs* bs'
using *field-lookup-is-value-byte-lense-conv*
by (*simp add*: *eq-upto-padding-def flense.eq-upto-padding-is-value-byte-conv*)

lemma *xmem-type-field-lookup-eq-padding-focus*:
assumes *lbs*: $length\ bs = size\text{-}of\ TYPE('a)$
assumes *lbs'*: $length\ bs' = size\text{-}of\ TYPE('a)$

assumes *eq-padding*: *eq-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs'*
shows *eq-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
using *field-lookup-eq-padding-focus* [*OF fl lbs* [*simplified size-of-def*] *lbs'*[*simplified size-of-def*] - -
eq-padding [*simplified typ-uinfo-t-def*]]
by (*simp add*: *typ-uinfo-t-def size-of-def*)

lemma *xmem-type-field-lookup-eq-padding-focus-eq*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *pfx-eq*: ⋀*i*. *i* < *n* ⟹ *bs* ! *i* = *bs'* ! *i*
assumes *sfx-eq*: ⋀*i*. *n* + *size-td s* ≤ *i* ⟹ *i* < *size-of TYPE*('*a*) ⟹ *bs* ! *i* = *bs'* ! *i*
shows *eq-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs'* =
eq-padding (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
using *field-lookup-eq-padding-focus-eq* [*OF fl lbs* [*simplified size-of-def*] *lbs'*[*simplified size-of-def*] *pfx-eq sfx-eq*]
by (*simp add*: *typ-uinfo-t-def size-of-def*)

lemma *xmem-type-field-lookup-lense-eq-padding-focus*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *eq-padding*: *lense.eq-padding bs bs'*
shows *flense.eq-padding* (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
using *xmem-type-field-lookup-eq-padding-focus* [*OF lbs lbs'*] *eq-padding*
eq-padding-lense-conv field-lookup-eq-padding-lense-conv
by *simp*

lemma *xmem-type-field-lookup-lense-eq-padding-focus-eq*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *pfx-eq*: ⋀*i*. *i* < *n* ⟹ *bs* ! *i* = *bs'* ! *i*
assumes *sfx-eq*: ⋀*i*. *n* + *size-td s* ≤ *i* ⟹ *i* < *size-of TYPE*('*a*) ⟹ *bs* ! *i* = *bs'* ! *i*
shows *lense.eq-padding bs bs'* =
flense.eq-padding (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
using *xmem-type-field-lookup-eq-padding-focus-eq* [*OF lbs lbs' pfx-eq sfx-eq*]
eq-padding-lense-conv field-lookup-eq-padding-lense-conv
by *simp*

lemma *xmem-type-field-lookup-eq-upto-padding-focus*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *eq-upto-padding*: *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs'*
shows *eq-upto-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
using *field-lookup-eq-upto-padding-focus* [*OF fl lbs* [*simplified size-of-def*] *lbs'*[*simplified*

size-of-def] - -
eq-upto-padding [*simplified typ-uinfo-t-def*]]
by (*simp add*: *typ-uinfo-t-def size-of-def*)

lemma *xmem-type-field-lookup-eq-upto-padding-focus-eq*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *pfx-eq*: $\bigwedge i.\ i < n \Longrightarrow bs\ !\ i = bs'\ !\ i$
assumes *sfx-eq*: $\bigwedge i.\ n + size\text{-}td\ s \leq i \Longrightarrow i < size\text{-}of\ TYPE('a) \Longrightarrow bs\ !\ i = bs'\ !\ i$
shows *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs'* =
eq-upto-padding (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
using *field-lookup-eq-upto-padding-focus-eq* [*OF fl lbs* [*simplified size-of-def*] *lbs'*[*simplified size-of-def*] *pfx-eq sfx-eq*]
by (*simp add*: *typ-uinfo-t-def size-of-def*)

lemma *xmem-type-field-lookup-lense-eq-upto-padding-focus*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *eq-upto-padding*: *lense.eq-upto-padding bs bs'*
shows *flense.eq-upto-padding* (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
using *xmem-type-field-lookup-eq-upto-padding-focus* [*OF lbs lbs'*] *eq-upto-padding*
eq-upto-padding-lense-conv field-lookup-eq-upto-padding-lense-conv
by *simp*

lemma *xmem-type-field-lookup-lense-eq-upto-padding-focus-eq*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *pfx-eq*: $\bigwedge i.\ i < n \Longrightarrow bs\ !\ i = bs'\ !\ i$
assumes *sfx-eq*: $\bigwedge i.\ n + size\text{-}td\ s \leq i \Longrightarrow i < size\text{-}of\ TYPE('a) \Longrightarrow bs\ !\ i = bs'\ !\ i$
shows *lense.eq-upto-padding bs bs'* =
flense.eq-upto-padding (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
using *xmem-type-field-lookup-eq-upto-padding-focus-eq* [*OF lbs lbs' pfx-eq sfx-eq*]
eq-upto-padding-lense-conv field-lookup-eq-upto-padding-lense-conv
by *simp*

lemma *xmem-type-field-lookup-eq-padding-super-update-bs*:
assumes *lxbs*: *length xbs* = *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *eq-padding* (*typ-uinfo-t TYPE*('*a*)) (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*) $\longleftrightarrow$
eq-padding (*export-uinfo s*) *bs bs'*
using *field-lookup-eq-padding-super-update-bs* [*OF fl lxbs* [*simplified size-of-def*] *lbs lbs'*]

by (*simp add*: *typ-uinfo-t-def*)

lemma *xmem-type-field-lookup-lense-eq-padding-super-update-bs*:
assumes *lxbs*: *length xbs* = *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *lense.eq-padding* (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*)
⟷
flense.eq-padding bs bs'
using *xmem-type-field-lookup-eq-padding-super-update-bs* [*OF lxbs lbs lbs'*]
eq-padding-lense-conv field-lookup-eq-padding-lense-conv
by *simp*

lemma *xmem-type-field-lookup-eq-upto-padding-super-update-bs*:
assumes *lxbs*: *length xbs* = *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*) ⟷
eq-upto-padding (*export-uinfo s*) *bs bs'*
using *field-lookup-eq-upto-padding-super-update-bs* [*OF fl lxbs* [*simplified size-of-def*] *lbs lbs'*]
by (*simp add*: *typ-uinfo-t-def*)

lemma *xmem-type-field-lookup-lense-eq-upto-padding-super-update-bs*:
assumes *lxbs*: *length xbs* = *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *lense.eq-upto-padding* (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*) ⟷
flense.eq-upto-padding bs bs'
using *xmem-type-field-lookup-eq-upto-padding-super-update-bs* [*OF lxbs lbs lbs'*]
eq-upto-padding-lense-conv field-lookup-eq-upto-padding-lense-conv
by *simp*

lemma *access-ti-update-ti-lense-eq-upto-padding*:
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *flense.eq-upto-padding* (*access-ti s* (*update-ti s bs v*) *bs'*) *bs*
by (*simp add*: *flense.value-byte-to-eq-upto-padding-eq lbs lbs' length-fa-ti*
flense.is-value-byte-acc-upd-cancel wf-fd-s)

lemma *access-ti-update-ti-eq-upto-padding*:
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *eq-upto-padding* (*export-uinfo s*) (*access-ti s* (*update-ti s bs v*) *bs'*) *bs*
using *access-ti-update-ti-lense-eq-upto-padding* [*OF lbs lbs'*]
field-lookup-eq-upto-padding-lense-conv
by *simp*

lemma *access-ti-update-ti-lense-eq-padding*:
assumes *flense.eq-padding bs bs'*
shows *access-ti s* (*update-ti s bs v*) *bs'* = *bs*
by (*metis* (*no-types*, *lifting*) *access-ti-update-ti-lense-eq-upto-padding assms flense.field-access-eq-padding1*
flense.padding-eq-complement padding-base.eq-padding-length1
flense.eq-padding-length2 flense.eq-padding-trans)

lemma *access-ti-update-ti-eq-padding*:
assumes *eq-padding* (*export-uinfo s*) *bs bs'*
shows *access-ti s* (*update-ti s bs v*) *bs'* = *bs*
using *access-ti-update-ti-lense-eq-padding assms field-lookup-eq-padding-lense-conv*
by *simp*

lemma
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*::*xmem-type*)
assumes *lbs*: *length bs* = *size-td s*
shows *access-ti s v bs* = *to-bytes* ((*from-bytes* (*access-ti s v bs*))::'*b*) *bs*
proof −

from *match*
have *sz-acc-s*: *length* (*access-ti s v bs*) = *size-td* (*typ-info-t TYPE*('*b*))
by (*metis export-size-of flense.access-result-size fold-typ-uinfo-t*)
show *?thesis*
by (*smt* (*verit*, *best*) *field-lookup-eq-padding-lense-conv flense.field-access-eq-padding1*
lbs match sz-acc-s xmem-type-class.eq-padding-lense-conv xmem-type-class.field-sz-size-of
xmem-type-class.field-sz-size-td xmem-type-class.lense-eq-padding-to-bytes-eq
xmem-type-class.to-bytes-from-bytes-id)
qed

context
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*::*xmem-type*)
begin

lemma *field-lookup-lense-eq-padding-fieldtyp-conv*:
flense.eq-padding bs bs' = *xmem-type-class.lense.eq-padding TYPE*('*b*) *bs bs'*
using *field-lookup-eq-padding-lense-conv match xmem-type-class.eq-padding-lense-conv*
by *auto*

lemma *field-lookup-lense-eq-upto-padding-fieldtyp-conv*:
flense.eq-upto-padding bs bs' = *xmem-type-class.lense.eq-upto-padding TYPE*('*b*)
bs bs'
using *field-lookup-eq-upto-padding-lense-conv match xmem-type-class.eq-upto-padding-lense-conv*
by *auto*

lemma *field-lookup-is-value-byte-fieldtyp-conv*:
flense.is-value-byte i = *xmem-type-class.lense.is-value-byte TYPE*('*b*) *i*
using *field-lookup-is-value-byte-lense-conv match xmem-type-class.is-value-byte-lense-conv*
by *auto*

lemma *field-lookup-is-padding-byte-fieldtyp-conv*:
flense.is-padding-byte i = xmem-type-class.lense.is-padding-byte TYPE('b) i
using *field-lookup-is-padding-byte-lense-conv match xmem-type-class.is-padding-byte-lense-conv*
by *auto*

lemma *field-lookup-access-ti-to-bytes-field-conv*:
assumes *eq-upto-padding*: *eq-upto-padding (export-uinfo s) (access-ti s v bs) vs*
assumes *eq-padding*: *eq-padding (export-uinfo s) bs bs'*
shows *access-ti s v bs = to-bytes ((from-bytes vs)::'b) bs'*
proof −
from *eq-upto-padding* **have** *flense-eq-upto*: *flense.eq-upto-padding (access-ti s v bs) vs*
by (*simp add*: *field-lookup-eq-upto-padding-lense-conv*)
hence *blense-eq-upto*: *xmem-type-class.lense.eq-upto-padding TYPE('b) (access-ti s v bs) vs*
by (*simp add*: *field-lookup-lense-eq-upto-padding-fieldtyp-conv*)

from *eq-padding* **have** *flense-eq*: *flense.eq-padding bs bs'*
by (*simp add*: *field-lookup-eq-padding-lense-conv*)
hence *blense-eq*: *xmem-type-class.lense.eq-padding TYPE('b) bs bs'*
by (*simp add*: *field-lookup-lense-eq-padding-fieldtyp-conv*)

show *?thesis*
using *flense-eq flense-eq-upto blense-eq blense-eq-upto*

apply (*simp add*: *to-bytes-def from-bytes-def c-type-class.to-bytes-def c-type-class.from-bytes-def*)
by (*smt* (*verit*, *ccfv-threshold*) *c-type-class.to-bytes-def field-lookup-lense-eq-padding-fieldtyp-conv flense.eq-padding-length1 flense.field-access-eq-padding1 padding-base.eq-upto-padding-length2 update-ti-s-from-bytes xmem-type-class.lense.eq-padding-acc xmem-type-class.lense-eq-upto-padding-from-b xmem-type-class.to-bytes-from-bytes-id*)
qed

lemma *field-lookup-access-ti-eq-upto-padding*:
length bs = size-td s ⟹
eq-upto-padding (export-uinfo s) (access-ti s v bs) (access-ti s v bs')
by (*metis access-ti-update-ti-eq-upto-padding flense.access-result-size flense.update-access*)

lemma *field-lookup-access-ti-eq-padding-value*:
length bs = size-td s ⟹ eq-padding (export-uinfo s) (access-ti s v bs) (access-ti s v' bs)
by (*meson eq-padding-sym eq-padding-trans field-lookup-eq-padding-lense-conv flense.field-access-eq-padding1*)

lemma *field-lookup-access-ti-eq-padding-bytes*:
length bs = size-td s ⟹ eq-padding (export-uinfo s) (access-ti s v bs) bs
using *field-lookup-eq-padding-lense-conv flense.field-access-eq-padding1* **by** *blast*

lemma *field-lookup-access-ti-to-bytes-field-conv'*:

assumes *eq-upto-padding*: *eq-upto-padding* (*export-uinfo s*) (*access-ti s v bs*) *vs*
assumes *lbs*: *length bs* = *size-td s*
shows *access-ti s v bs* = *to-bytes* ((*from-bytes vs*)::*'b*) *bs*
using *field-lookup-access-ti-to-bytes-field-conv* [*OF eq-upto-padding eq-padding-refl*] *lbs*
by *simp*

lemma *field-lookup-update-ti-from-bytes-field-conv*:
fixes *v*::*'a* **and** *vf*::*'b*
assumes *lbs*: *length bs* = *size-td s*
assumes *lxbs*: *length xbs* = *size-of TYPE*(*'a*)
shows *update-ti* (*typ-info-t TYPE*(*'b*)) *bs vf* =
(*from-bytes* (*access-ti s* (*update-ti s bs v*) *xbs*))
proof −
from *lbs lxbs*
have *eq-upto-padding* (*export-uinfo s*) (*access-ti s* (*update-ti s bs v*) *xbs*) *bs*
by (*metis access-ti-update-ti-eq-upto-padding access-ti-update-ti-lense-eq-padding flense.access-result-size flense.eq-padding-refl flense.update-access*)

hence *from-bytes* (*access-ti s* (*update-ti s bs v*) *xbs*) = (*from-bytes bs*::*'b*)
by (*simp add*: *match xmem-type-class.eq-upto-padding-from-bytes-eq*)
moreover have *update-ti* (*typ-info-t TYPE*(*'b*)) *bs vf* = *from-bytes bs*
using *lbs*
by (*metis* (*mono-tags*, *lifting*) *c-type-class.typ-uinfo-size export-size-of export-uinfo-size match update-ti-s-from-bytes update-ti-update-ti-t*)
ultimately show *?thesis* **by** *simp*
qed

lemma *xmem-type-field-lookup-update-ti-super-update-bs-conv*:
fixes *v*::*'a*
assumes *lbs*: *length bs* = *size-td s*
assumes *lxbs*: *length xbs* = *size-of TYPE*(*'a*)
shows *update-ti s bs v* =
update-ti (*typ-info-t TYPE*(*'a*)) (*super-update-bs bs* (*access-ti* (*typ-info-t TYPE*(*'a*)) *v xbs*) *n*) *v*
using *field-lookup-update-ti-super-update-bs-conv*(*1*) [**where** *m*=*0*, *simplified*, *OF fl lbs lxbs* [*simplified size-of-def*], **where** *v*=*v*]
by *simp*

lemma *heap-update-list-field-to-root*:
fixes *p*::*'a ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *size-td s*
shows *heap-update-list* (&(*p*→*f*)) *bs hp* =
heap-update-list (*ptr-val p*) (*super-update-bs bs* (*heap-list hp* (*size-of TYPE*(*'a*)) (*ptr-val p*)) *n*) *hp*

proof −
from *fl*
have *off*: *word-of-nat* (*field-offset TYPE*('*a*) *f*) = *word-of-nat n*
by (*simp*)
from *c-guard-no-wrap'* [*OF cgrd*]
have *no-overflow*: *unat* (*ptr-val p*) + *size-of TYPE*('*a*) ≤ *addr-card* **.**
from *lbs fl* **have** *n-bound*: *n* + *length bs* ≤ *size-of TYPE*('*a*)
by (*metis add.commute field-lookup-offset-size size-of-def*)

show *?thesis*
apply (*simp add*: *field-lvalue-def off*)
apply (*rule heap-update-list-super-update-bs-heap-list* [*OF no-overflow n-bound*])
done
qed

lemma *heap-list-field-to-root*:
fixes *p*::'*a ptr*
shows *heap-list hp* (*size-td s*) &(*p*→*f*) =
take (*size-td s*) ((*drop n*) (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val* (*p*::'*a ptr*))))
proof −
from *fl* **have** *off*: (*field-offset TYPE*('*a*) *f*) = *n* **by** *simp*
from *fl*
have *n-bound*: *n* ≤ *size-of TYPE*('*a*)
by (*meson n-t nat-less-le*)
from *fl*
have *s-bound*: *size-td s* ≤ *size-of TYPE*('*a*) − *n*
by (*metis field-lookup-offset-size nat-move-sub-le size-of-def*)

show *?thesis*
by (*simp add*: *field-lvalue-def off drop-heap-list-le* [*OF n-bound*] *take-heap-list-le* [*OF s-bound*])
qed

lemma *heap-list-field-super-update-bs-root-conv*:
fixes *p*::'*a ptr*
shows *super-update-bs* (*heap-list hp* (*size-td s*) (&(*p*→*f*))) (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)) *n* =
(*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*))
proof −

have *l*: *length* (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)) = *size-of TYPE*('*a*)
by *simp*

from *l fl* **have** *l-take*: (*length* (*take n* (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)))) = *n*
by (*simp add*: *le-less n-t*)

```
  from l fl n-t
  have l-s-take: length ((take (size-td s) (drop n (heap-list hp (size-of TYPE('a))
(ptr-val p))))) = size-td s
    by (metis heap-list-field-to-root heap-list-length)

  have com: (n + size-td s) = (size-td s + n)
    by simp
  show ?thesis
    apply (simp add: super-update-bs-def)
    apply (subst heap-list-field-to-root)
    apply (rule  append-take-dropI)
     apply (simp only: l-take)
    apply (rule  append-take-dropI)
     apply (simp only: l-s-take l-take)
    apply (simp only: l-s-take l-take)
    apply (simp)
    apply (subst com)
    apply (rule refl)
    done

qed

lemma heap-update-field-root-conv:
  fixes p::'a ptr
  assumes cgrd: c-guard p
  shows heap-update (PTR('b) &(p→f)) v hp =
          heap-update p (update-ti s (to-bytes v (heap-list hp (size-of TYPE('b))
(&(p→f)))) (h-val hp p)) hp
unfolding heap-update-def c-type-class.heap-update-def ptr-val-def
proof −
 show heap-update-list &(p→f) (to-bytes v (heap-list hp (size-of TYPE('b)) &(p→f)))
hp =
    heap-update-list (ptr-val p)
     (to-bytes
       (update-ti s (to-bytes v (heap-list hp (size-of TYPE('b)) &(p→f))) (h-val hp
p))
       (heap-list hp (size-of TYPE('a)) (ptr-val p)))
     hp (is ?lhs = ?rhs)
  proof −
    define bs where bs = to-bytes v (heap-list hp (size-of TYPE('b)) (&(p→f)))
    have lbs: length bs = size-td s
      unfolding bs-def
      by (simp add: export-size-of match)

    from heap-update-list-field-to-root [OF cgrd lbs]
    have eq1:
      heap-update-list (&(p→f)) bs hp =
            heap-update-list (ptr-val p) (super-update-bs bs (heap-list hp (size-of
```

TYPE('*a*)) (*ptr-val p*)) *n*) *hp*
by *simp*

have *lhl*: *length* (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)) = *size-of TYPE*('*a*)
by *simp*
from *xmem-type-field-lookup-update-ti-super-update-bs-conv* [*OF lbs this*, **where** *v*=*h-val hp p*]
have *eq2*:
update-ti s bs (*h-val hp p*) =
update-ti (*typ-info-t TYPE*('*a*))
(*super-update-bs bs*
(*access-ti* (*typ-info-t TYPE*('*a*)) (*h-val hp p*) (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)))
n)
(*h-val hp p*) **.**

have *eq3*: *access-ti* (*typ-info-t TYPE*('*a*)) (*h-val hp p*) (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)) =
heap-list hp (*size-of TYPE*('*a*)) (*ptr-val p*)
apply (*simp add*: *h-val-def from-bytes-def update-ti-t-def size-of-def*)
using *lhl*
by (*metis local.field-sz-size-of local.field-sz-size-td local.lense.access-result-size local.lense.eq-padding-neq-is-value-byte local.lense.field-access-eq-padding1 local.lense.is-value-byte-acc-upd-cancel nth-equalityI*)

have *lsuper*: *length* (*super-update-bs bs* (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)) *n*) = *size-of TYPE*('*a*)
using *flense.eq-padding-refl lbs lhl padding-base.eq-padding-length1 xmem-type-field-lookup-lense-eq-padding-super-update-bs* **by** *blast*

thm *field-lookup-eq-padding-super-update-bs*
have *eq-padding-bs*: *eq-padding* (*export-uinfo s*) *bs* (*heap-list hp* (*size-td s*) (&(*p*→*f*)))
apply (*simp add*: *bs-def to-bytes-def c-type-class.to-bytes-def*)
by (*metis* (*mono-tags*, *lifting*) *c-type-class.to-bytes-def export-size-of heap-list-length match xmem-type-class.eq-padding-to-bytes*)

have *l-take-drop*: *length* (*take* (*size-td s*) (*drop n* (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)))) = *size-td s*
by (*metis heap-list-field-to-root heap-list-length*)
thm *heap-list-field-super-update-bs-root-conv*

from *xmem-type-field-lookup-eq-padding-super-update-bs* [*OF lhl lbs*,
where *bs*′ = *take* (*size-td s*) (*drop n* (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*))), *OF l-take-drop*,
simplified heap-list-field-to-root [*symmetric*], *simplified heap-list-field-super-update-bs-root-conv*]

```
        eq-padding-bs
  have eq-padding (typ-uinfo-t TYPE('a)) (super-update-bs bs (heap-list hp (size-of
TYPE('a)) (ptr-val p)) n)
          (heap-list hp (size-of TYPE('a)) (ptr-val p))
    by (simp )

    hence eq-padding: lense.eq-padding (super-update-bs bs (heap-list hp (size-of
TYPE('a)) (ptr-val p)) n)
          (heap-list hp (size-of TYPE('a)) (ptr-val p))
    by (simp add:  eq-padding-lense-conv)

   from eq-padding
   have eq4:
     (to-bytes
        (update-ti (typ-info-t TYPE('a))
         (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) (h-val
hp p))
         (heap-list hp (size-of TYPE('a)) (ptr-val p))) =
      (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n)
     apply (simp add: to-bytes-def)
   by (smt (verit) lhl local.lense.access-result-size local.lense.eq-padding-neq-is-value-byte
        local.lense.field-access-eq-padding1 local.lense.is-value-byte-acc-upd-cancel
lsuper nth-equalityI)

   show ?thesis
     unfolding bs-def [symmetric]
     apply (simp add: eq1)
     apply (simp add: eq2)
     apply (simp add: eq3)
     apply (simp add: eq4)
     done
  qed
qed

lemma heap-update-padding-field-root-conv:
  fixes p::'a ptr
  assumes cgrd: c-guard p
  assumes lbs: length bs = size-of TYPE ('b)
  shows heap-update-padding (PTR('b) &(p→f)) v bs hp =
        heap-update-padding p (update-ti s (to-bytes v bs) (h-val hp p))
         (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) hp
unfolding heap-update-padding-def c-type-class.heap-update-padding-def ptr-val-def
proof −
  show heap-update-list &(p→f) (to-bytes v bs) hp =
    heap-update-list (ptr-val p)
     (to-bytes
       (update-ti s (to-bytes v bs) (h-val hp p))
       (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n))
     hp (is ?lhs = ?rhs)
```

proof −
define *bs′* **where** *bs′ = to-bytes v bs*
have *lbs′*: *length bs′ = size-td s*
unfolding *bs′-def*
by (*simp add*: *export-size-of lbs match*)

from *heap-update-list-field-to-root* [*OF cgrd lbs′*]
have *eq1*:
heap-update-list (*&(p→f)*) *bs′ hp* =
heap-update-list (*ptr-val p*) (*super-update-bs bs′* (*heap-list hp* (*size-of TYPE('a)*) (*ptr-val p*)) *n*) *hp*
by *simp*

have *lhl*: *length* (*heap-list hp* (*size-of TYPE('a)*) (*ptr-val p*)) = *size-of TYPE('a)*
by *simp*
from *xmem-type-field-lookup-update-ti-super-update-bs-conv* [*OF lbs′ this*, **where** *v=h-val hp p*]
have *eq2*:
update-ti s bs′ (*h-val hp p*) =
update-ti (*typ-info-t TYPE('a)*)
(*super-update-bs bs′*
(*access-ti* (*typ-info-t TYPE('a)*) (*h-val hp p*) (*heap-list hp* (*size-of TYPE('a)*) (*ptr-val p*)))
n)
(*h-val hp p*) **.**

have *eq3*: *access-ti* (*typ-info-t TYPE('a)*) (*h-val hp p*) (*heap-list hp* (*size-of TYPE('a)*) (*ptr-val p*)) =
heap-list hp (*size-of TYPE('a)*) (*ptr-val p*)
apply (*simp add*: *h-val-def from-bytes-def update-ti-t-def size-of-def*)
using *lhl*
by (*metis local.field-sz-size-of local.field-sz-size-td local.lense.access-result-size local.lense.eq-padding-neq-is-value-byte local.lense.field-access-eq-padding1 local.lense.is-value-byte-acc-upd-cancel nth-equalityI*)

have *lsuper*: *length* (*super-update-bs bs′* (*heap-list hp* (*size-of TYPE('a)*) (*ptr-val p*)) *n*) = *size-of TYPE('a)*
using *flense.eq-padding-refl lbs′ lhl padding-base.eq-padding-length1 xmem-type-field-lookup-lense-eq-padding-super-update-bs* **by** *blast*

thm *field-lookup-eq-padding-super-update-bs*

have *l-take-drop*: *length* (*take* (*size-td s*) (*drop n* (*heap-list hp* (*size-of TYPE('a)*) (*ptr-val p*)))) = *size-td s*
by (*metis heap-list-field-to-root heap-list-length*)
have *eq4*:
(*to-bytes*

```
        (update-ti (typ-info-t TYPE('a))
         (super-update-bs bs' (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) (h-val
hp p))
        (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n)) =
       (super-update-bs bs' (heap-list hp (size-of TYPE('a)) (ptr-val p)) n)
      apply (simp add: to-bytes-def)
     by (smt (verit, ccfv-threshold) bs'-def export-size-of field-lookup-lense-eq-padding-fieldtyp-conv
       lbs lhl local.lense.access-result-size local.lense.complement local.lense.eq-padding-acc
local.lense.is-padding-byte-def local.lense.is-value-byte-def lsuper match mem-type-class.mem-type-simps(2)
nth-equalityI wf-type-class.len xmem-type-class.lense-eq-padding-to-bytes xmem-type-field-lookup-lense-eq-paddi

    show ?thesis
      unfolding bs'-def [symmetric]
      apply (simp add: eq1)
      apply (simp add: eq2)
      apply (simp add: eq3)
      apply (simp add: eq4)
      done
  qed
qed

lemma heap-update-field-root-conv':
  fixes p::'a ptr
  assumes cgrd: c-guard p
  shows heap-update (PTR('b) &(p→f)) v hp =
        heap-update p (update-ti s (to-bytes-p v) (h-val hp p)) hp
proof −
   have eq-upto-padding (export-uinfo s) (to-bytes-p v) (to-bytes v (heap-list hp
(size-of TYPE('b)) (&(p→f))))
   apply (simp add: to-bytes-def c-type-class.to-bytes-def to-bytes-p-def c-type-class.to-bytes-p-def)
   by (simp add: field-lookup-eq-upto-padding-lense-conv field-lookup-lense-eq-upto-padding-fieldtyp-conv
       xmem-type-class.lense.field-access-eq-upto-padding)
  hence update-ti s (to-bytes-p v) (h-val hp p) =
         update-ti s (to-bytes v (heap-list hp (size-of TYPE('b)) (&(p→f)))) (h-val
hp p)
   by (simp add: field-lookup-eq-upto-padding-lense-conv padding-base.eq-upto-padding-def)
  with heap-update-field-root-conv [OF cgrd] show ?thesis by simp
qed


lemma heap-update-padding-field-root-conv':
  fixes p::'a ptr
  assumes cgrd: c-guard p
  assumes lbs: length bs = size-of TYPE ('b)
  shows heap-update-padding (PTR('b) &(p→f)) v bs hp =
        heap-update-padding p (update-ti s (to-bytes-p v) (h-val hp p))
          (super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) hp
  proof −
  have eq-upto-padding (export-uinfo s) (to-bytes-p v) (to-bytes v bs)
```

apply (*simp add*: *to-bytes-def c-type-class.to-bytes-def to-bytes-p-def c-type-class.to-bytes-p-def*)
by (*simp add*: *field-lookup-eq-upto-padding-lense-conv field-lookup-lense-eq-upto-padding-fieldtyp-conv xmem-type-class.lense.field-access-eq-upto-padding*)
hence *update-ti s* (*to-bytes-p v*) (*h-val hp p*) =
update-ti s (*to-bytes v bs*) (*h-val hp p*)
by (*simp add*: *field-lookup-eq-upto-padding-lense-conv padding-base.eq-upto-padding-def*)
with *heap-update-padding-field-root-conv* [*OF cgrd lbs*] **show** *?thesis* **by** *simp*
qed

end
end

lemma *heap-update-field-root-conv''*:
fixes *p*::'*a ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
shows *heap-update* (*PTR*('*b*) &(*p*→*f*)) *v hp* =
heap-update p (*fld-update* (λ-. *v*) (*h-val hp p*)) *hp*
proof −
have *match*: *export-uinfo* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *fld* (*fld-update* ∘ (λ*x* -. *x*))) = *typ-uinfo-t TYPE*('*b*)
apply (*subst c-type-class.typ-uinfo-t-def*)
apply (*rule export-tag-adjust-ti*(*1*))
apply (*rule fg-cons*)
apply (*simp*)
done
from *field-desc-adjust-ti*(*1*) [*rule-format*, *OF fg-cons*, *of* (*typ-info-t TYPE*('*b*))]
have *upd-eq*:
update-ti-t (*adjust-ti* (*typ-info-t TYPE*('*b*)) *fld* (*fld-update* ∘ (λ*x* -. *x*))) *bs v* =
fld-update (λ-. *update-ti-t* (*typ-info-t TYPE*('*b*)) *bs* (*fld v*)) *v* **for** *bs v*
by (*simp add*: *update-desc-def*)
from *this* [*of* (*to-bytes-p v*) *h-val hp p*]
have *val-conv*:
(*update-ti* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *fld* (*fld-update* ∘ (λ*x* -. *x*))) (*to-bytes-p v*) (*h-val hp p*)) =
(*fld-update* (λ-. *v*) (*h-val hp p*))
apply (*simp add*: *update-ti-t-def size-of-def c-type-class.size-of-def*)
by (*metis* (*mono-tags*, *opaque-lifting*) *inv-p length-to-bytes-p update-ti-s-from-bytes update-ti-t-def*)

note *heap-conv* = *heap-update-field-root-conv'* [*OF fl match cgrd*, *of v hp*]
show *?thesis*
apply (*simp add*: *heap-conv*)
apply (*simp add*: *val-conv*)
done
qed

lemma *heap-update-field-root-conv-pointless'*:
fixes *p*::*'a ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
shows *heap-update* (*PTR*(*'b*) &(*p*→*f*)) *v* =
(λ*hp*. *heap-update p* (*fld-update* (λ-. *v*) (*h-val hp p*)) *hp*)
using *heap-update-field-root-conv''* [*OF fl fg-cons cgrd*]
by (*simp add*: *fun-eq-iff*)

lemma *heap-update-padding-field-root-conv''*:
fixes *p*::*'a ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *size-of TYPE* (*'b*)
shows *heap-update-padding* (*PTR*(*'b*) &(*p*→*f*)) *v bs hp* =
heap-update-padding p (*fld-update* (λ-. *v*) (*h-val hp p*))
(*super-update-bs bs* (*heap-list hp* (*size-of TYPE*(*'a*)) (*ptr-val p*)) *n*) *hp*
proof −
have *match*: *export-uinfo* (*adjust-ti* (*typ-info-t TYPE*(*'b*)) *fld* (*fld-update* ∘ (λ*x* -. *x*))) = *typ-uinfo-t TYPE*(*'b*)
apply (*subst c-type-class.typ-uinfo-t-def*)
apply (*rule export-tag-adjust-ti*(*1*))
apply (*rule fg-cons*)
apply (*simp*)
done
from *field-desc-adjust-ti*(*1*) [*rule-format*, *OF fg-cons*, *of* (*typ-info-t TYPE*(*'b*))]
have *upd-eq*:
update-ti-t (*adjust-ti* (*typ-info-t TYPE*(*'b*)) *fld* (*fld-update* ∘ (λ*x* -. *x*))) *bs v* =
fld-update (λ-. *update-ti-t* (*typ-info-t TYPE*(*'b*)) *bs* (*fld v*)) *v* **for** *bs v*
by (*simp add*: *update-desc-def*)
from *this* [*of* (*to-bytes-p v*) *h-val hp p*]
have *val-conv*:
(*update-ti* (*adjust-ti* (*typ-info-t TYPE*(*'b*)) *fld* (*fld-update* ∘ (λ*x* -. *x*))) (*to-bytes-p v*) (*h-val hp p*)) =
(*fld-update* (λ-. *v*) (*h-val hp p*))
apply (*simp add*: *update-ti-t-def size-of-def c-type-class.size-of-def*)
by (*metis* (*mono-tags*, *opaque-lifting*) *inv-p length-to-bytes-p update-ti-s-from-bytes update-ti-t-def*)

note *heap-conv* = *heap-update-padding-field-root-conv'* [*OF fl match cgrd lbs*, *of v hp*]
show *?thesis*
apply (*simp add*: *heap-conv*)
apply (*simp add*: *val-conv*)

done
qed

lemma *heap-update-field-root-conv'''*:
fixes *p*::*'a ptr*
assumes *fl*: *field-ti TYPE('a) f = Some s*
assumes *cgrd*: *c-guard p*
assumes *match*: *export-uinfo s = typ-uinfo-t TYPE('b::xmem-type)*
shows *heap-update (PTR('b) &(p→f)) v hp =*
heap-update p (update-ti s (to-bytes-p v) (h-val hp p)) hp
proof −
from *fl* **obtain** *n* **where** *fl'*: *field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)*
by (*auto simp add*: *field-ti-def split*: *option.splits*)

from *heap-update-field-root-conv' [OF fl' match cgrd, of v hp]*
show *?thesis*
by (*simp*)
qed

lemma *heap-update-padding-field-root-conv'''*:
fixes *p*::*'a ptr*
assumes *fl*: *field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n)*
assumes *cgrd*: *c-guard p*
assumes *match*: *export-uinfo s = typ-uinfo-t TYPE('b::xmem-type)*
assumes *lbs*: *length bs = size-of TYPE ('b)*
shows *heap-update-padding (PTR('b) &(p→f)) v bs hp =*
heap-update-padding p (update-ti s (to-bytes-p v) (h-val hp p))
(super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) hp
proof −
from *heap-update-padding-field-root-conv' [OF fl match cgrd lbs, of v hp]*
show *?thesis*
by (*simp*)
qed

end

lemma *length-array-to-bytes*:
fixes *arr*::(*'a*::*array-outer-max-size*['*b*::*array-max-count*])
shows *length (to-bytes arr (heap-list h (CARD('b) ∗ size-of TYPE('a)) (ptr-val p))) =*
size-of TYPE('a) ∗ CARD('b)
by (*simp add*: *lense.access-result-size*)

lemma *take-drop-append-first*: *m + n ≤ length xs ⟹ take n (drop m (xs @ ys)) = take n (drop m xs)*
by *simp*

```
lemma size-td-list-array:
size-td-list
    (map (λn. DTuple
              (adjust-ti (typ-info-t TYPE('a::xmem-type)) (λx. x.[n])
                (λx f. Arrays.update f n x))
              (replicate n CHR ''1'')
              (|field-access = xto-bytes ∘ (λx. x.[n]),
                field-update = (λx f. Arrays.update f n x) ∘ xfrom-bytes,
                field-sz = size-of TYPE('a)|))
      [0..<n]) = n * size-of TYPE('a)
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case by (simp add: size-of-def)
qed

lemma length-access-ti-list-array:
  fixes arr::('a::array-outer-max-size['b::array-max-count])
  assumes lxbs: length xbs = n * size-of TYPE('a)
  shows
 length (access-ti-list
        (map (λn. DTuple
                  (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n])
                    (λx f. Arrays.update f n x))
                  (replicate n CHR ''1'')
                  (|field-access = xto-bytes ∘ (λx. x.[n]),
                    field-update = (λx f. Arrays.update f n x) ∘ xfrom-bytes,
                    field-sz = size-of TYPE('a)|))
          [0..<n])
        arr xbs) = n * size-of TYPE('a)
  using lxbs
proof (induct n arbitrary: xbs)
  case 0
  then show ?case by simp
next
  case (Suc n)

  have lxbs': length xbs =
    size-td-list
     (map (λn. DTuple
              (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n])
                (λx f. Arrays.update f n x))
              (replicate n CHR ''1'')
              (|field-access = λa. xto-bytes (a.[n]),
                field-update = λa b. Arrays.update b n (xfrom-bytes a),
                field-sz = size-td (typ-info-t TYPE('a))|))
```

```
      [0..<n] @
     [DTuple (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n]) (λx f. Arrays.update f n
x))
       (replicate n CHR ''1'')
       (⦇field-access = λa. xto-bytes (a.[n]),
          field-update = λa b. Arrays.update b n (xfrom-bytes a),
          field-sz = size-td (typ-info-t TYPE('a))⦈)]
   apply simp
   apply (subst size-td-list-array [simplified size-of-def comp-def])
   using Suc.prems
   apply (simp add: size-of-def)
   done

  from Suc.prems
  have ltake: length (take (n * size-td (typ-info-t TYPE('a))) xbs) =
                n * size-td (typ-info-t TYPE('a))
    by (auto simp add: size-of-def)

  have llast: length
     (access-ti (typ-info-t TYPE('a)) (arr.[n])
       (take (size-td (typ-info-t TYPE('a)))
         (drop
           (size-td-list
             (map (λn. DTuple
                       (adjust-ti (typ-info-t TYPE('a)) (λx. x.[n])
                         (λx f. Arrays.update f n x))
                       (replicate n CHR ''1'')
                       (⦇field-access = λa. xto-bytes (a.[n]),
                          field-update = λa b. Arrays.update b n (xfrom-bytes a),
                          field-sz = size-td (typ-info-t TYPE('a))⦈)
               [0..<n]))
           xbs))) =
    size-td (typ-info-t TYPE('a))
    by (simp add: lense.access-result-size size-of-def)

  show ?case apply (simp add: size-of-def)
    apply (subst access-ti-append [OF lxbs'])
    apply (subst size-td-list-array [simplified size-of-def comp-def])
    apply simp
    apply (subst Suc.hyps [simplified size-of-def comp-def, OF ltake])
    apply (simp add: llast)
    done
qed

lemma take-drop-take: m + k ≤ n ⟹ n ≤ length xbs ⟹ take m (drop k (take
n xbs)) = take m (drop k xbs)
proof (induct xbs arbitrary: m k n)
  case Nil
  then show ?case by simp
```

```
next
  case (Cons x xbs)
  then show ?case by (simp add: take-drop)
qed

lemma access-ti-array-index:
  fixes arr::('a::array-outer-max-size['b::array-max-count])
  assumes bound: i < n
  assumes lxbs: length xbs = n * size-of TYPE('a)
  assumes bs: bs = take (size-of TYPE('a)) (drop (i * (size-of TYPE('a))) xbs)
  shows
    access-ti (typ-info-t TYPE('a)) (arr.[i])
       (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) xbs)) =
     take (size-of TYPE('a))
       (drop (i * size-of TYPE('a)) (access-ti (array-tag-n n) arr xbs))
  using bound lxbs bs
proof (induct n arbitrary: i xbs bs)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems obtain
    i-bound: i < Suc n and
    lxbs: length xbs = size-of TYPE('a) + n * size-of TYPE('a) and
    bs: bs = take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) xbs)
    by clarsimp


  show ?case
  proof (cases i = n)
    case True
    show ?thesis
      apply (simp add: array-tag-n-eq)
      apply (subst access-ti-append)
      apply (simp add: size-td-list-array)
       apply (simp add: size-of-def lxbs)
      apply (subst drop-append-miracle)
       apply (subst length-access-ti-list-array)
        apply (subst size-td-list-array)
       using lxbs
        apply (simp add: True)
       apply (simp add: True)
      apply (simp add: True size-td-list-array)
      apply (simp add: size-of-def)
      by (simp add: lense.access-result-size size-of-def)
  next
    case False
    from False i-bound have i-bound': i < n by simp
     from lxbs have lxbs': length (take (n * size-of TYPE('a)) xbs) = n * size-of
```

```
TYPE('a) by simp
    from bs i-bound' lxbs' lxbs False
    have bs':
      bs = take (size-of TYPE('a))
            (drop (i * size-of TYPE('a)) (take (n * size-of TYPE('a)) xbs))

      apply (simp)
      apply (subst take-drop-take)
      using less-le-mult-nat sz-nzero apply blast
       apply simp
      apply simp
      done
    show ?thesis
      apply (simp add: array-tag-n-eq)
      apply (subst access-ti-append)
      apply (simp add: size-td-list-array)
       apply (simp add: size-of-def lxbs)
      apply (subst take-drop-append-first)
       apply (subst length-access-ti-list-array)
        apply (subst size-td-list-array)
      using lxbs apply (simp add: )
      using lxbs i-bound False
     apply (metis add.commute less-le-mult-nat mem-type-simps(3) not-less-less-Suc-eq)
      apply (subst size-td-list-array)
      apply (simp add: bs [symmetric])

      using Suc.hyps [where xbs=(take (n * size-of TYPE('a)) xbs), OF i-bound'
lxbs' bs', simplified bs'[symmetric]]
      apply (simp)
      apply (subst array-tag-n-eq)
      apply simp
      done
  qed
qed



lemma access-ti-array-index':
  fixes arr::('a::array-outer-max-size['b::array-max-count])
  assumes bound: i < CARD('b)
  assumes lbs: length xbs = (CARD('b) * size-of TYPE('a))
  assumes bs: bs = take (size-of TYPE('a)) (drop (i * (size-of TYPE('a))) xbs)
  shows
   access-ti (typ-info-t TYPE('a)) (arr.[i]) bs =
    take (size-of TYPE('a))
      (drop (i * size-of TYPE('a))
        (access-ti (typ-info-t TYPE('a['b])) arr xbs))
  apply (simp add: typ-info-array array-tag-def)
   using access-ti-array-index [OF bound lbs bs, where arr=arr, simplified bs
```

[*symmetric*]]
apply *simp*
done

lemma *fold-index-shift*: *fold f* [*n*..<*n* + *m*] = *fold* (λ*i*. *f* (*n* + *i*)) [*0*..<*m*]
proof (*induct m arbitrary*: *n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
thus *?case* **by** *simp*
qed

lemma *fold-Suc-index-shift*: *fold f* [*1*..<*Suc n*] = *fold* (λ*i*. *f* (*Suc i*)) [*0*..<*n*]
using *fold-index-shift* [**where** *n*=*1* **and** *m*=*n* **and** *f* = *f*]
by *simp*

lemma *sum-list-index-shift*: *sum-list* (*map f* [*n*..<*n*+*m*]) = *sum-list* (*map* (λ*i*. *f* (*n* + *i*)) [*0*..<*m*])
apply (*induct m*)
apply *auto*
done

lemma *sum-list-Suc-index-shift*: *sum-list* (*map f* [*1*..<*Suc n*]) = *sum-list* (*map* (λ*i*. *f* (*Suc i*)) [*0*..<*n*])
using *sum-list-index-shift* [**where** *n*=*1* **and** *m*=*n* **and** *f* = *f*]
by *simp*

lemma *upt-Suc-snoc*: [*0*..<*Suc n*] = [*0*..<*n*] @ [*n*]
by (*induct n*) *auto*

lemma *sum-list-le-prefix*:
fixes *sz*:: *nat* ⇒ *nat*
assumes *lower*: *n* ≤ *m*
assumes *le*: *m* ≤ *k*
shows *sum-list* (*map sz* [*n*..<*m*]) ≤ *sum-list* (*map sz* [*n*..<*k*])
apply (*subst upt-add-eq-append'* [*OF lower le*])
apply *simp*
done

lemma *intvl-off-disj'*:
fixes *x* :: *addr*
assumes *ylt*: *y* ≤ *off*
and *zoff*: *unat x* + *off* + *z* ≤ *addr-card*
shows {*x* ..+ *y*} ∩ {*x* + *of-nat off* ..+ *z*} = {}
proof (*cases unat x* + *off* + *z* = *addr-card*)
case *True*
then show *?thesis*

```
    using ylt zoff
    apply (simp add: intvl-def)
    apply (safe, clarsimp)
  proof −
    fix k :: nat and ka :: nat
    assume a1: unat x + off + z = addr-card
    assume a2: y ≤ off
    assume a3: k < y
   assume a4: (word-of-nat k::addr-bitsize word) = word-of-nat off + word-of-nat
ka
    assume ka < z
    then have ka + off < addr-card
      using a1 by linarith
    then show False
     using a4 a3 a2 by (metis (no-types) add.commute add-lessD1 len-of-addr-card
less-le-trans nat-less-le unat-of-nat-eq word-arith-nat-add)
  qed
next
  case False
  with zoff have le: off + z < addr-card by simp
  show ?thesis
    apply (rule intvl-off-disj [OF ylt])
    using le by (simp add: addr-card-wb)
qed

lemma heap-update-list-padding-fold-partition:
  fixes   v:: nat ⇒ byte list ⇒ byte list
    and  sz:: nat ⇒ nat
    and off:: nat ⇒ nat
  assumes no-overflow: unat a + length bs ≤ addr-card
  assumes partition: bs = concat (map (λi. take (sz i) (drop (off i) bs)) [0..<m])
  assumes lbs: length bs = sum-list (map sz [0..<m])
   assumes partition-pbs: pbs = concat (map (λi. take (sz i) (drop (off i) pbs))
[0..<m])
  assumes lpbs: length pbs = sum-list (map sz [0..<m])
  assumes off-sz: ⋀i. i < m ⟹ off i = sum-list (map sz [0..<i])
  assumes val: ⋀i. i < m ⟹
                  v i (take (sz i) (drop (off i) pbs)) =
                    (take (sz i) (drop (off i) bs))
  shows
   heap-update-list a bs h =
      fold (λi h. heap-update-list
                    (a + word-of-nat (off i))
                    (v i (take (sz i) (drop (off i) pbs)))
                    h)
          [0..<m] h
  using no-overflow partition lbs partition-pbs lpbs off-sz val
 proof (induct m arbitrary: a bs pbs h )
  case 0
```

```
  then show ?case by simp
next
  case (Suc m)
  from Suc.prems obtain
    no-overflow: unat a + length bs ≤ addr-card and
    bs-partition:  bs = concat (map (λi. take (sz i) (drop (off i) bs)) [0..<Suc m])
and
    pbs-partition:  pbs = concat (map (λi. take (sz i) (drop (off i) pbs)) [0..<Suc
m]) and
    lbs: length bs = sum-list (map sz [0..<Suc m]) and
    lpbs: length pbs = sum-list (map sz [0..<Suc m]) and
    off-sz: ⋀i. i < Suc m ⟹ off i = sum-list (map sz [0..<i]) and
    val: ⋀i. i < Suc m ⟹
            v i (take (sz i) (drop (off i) pbs)) =
            take (sz i) (drop (off i) bs)
    by clarsimp
  have unroll: [0..<Suc m] = [0..<m] @ [m]
    by (rule upt-Suc-snoc)

  define bs' where bs' = concat (map (λi. take (sz i) (drop (off i) bs)) [0..<m])
  define bsm where bsm = take (sz m) (drop (off m) bs)

 define pbs' where pbs' = concat (map (λi. take (sz i) (drop (off i) pbs)) [0..<m])
  define pbsm where pbsm = take (sz m) (drop (off m) pbs)

  from bs-partition have bs: bs = bs' @ bsm
    unfolding bsm-def bs'-def
    apply (subst (asm) unroll)
    apply simp
    done

  from pbs-partition have pbs: pbs = pbs' @ pbsm
    unfolding pbsm-def pbs'-def
    apply (subst (asm) unroll)
    apply simp
    done

  from lbs have lbs1: length bs = sum-list (map sz [0..<m]) + sz m
    apply (subst (asm) unroll)
    apply simp
    done

  from lpbs have lpbs1: length pbs = sum-list (map sz [0..<m]) + sz m
    apply (subst (asm) unroll)
    apply simp
    done

  have lbsm: length bsm = sz m
    apply (simp add: bsm-def)
```

```
  using lbs1 off-sz [of m]
  apply simp
  done

 have lpbsm: length pbsm = sz m
  apply (simp add: pbsm-def)
  using lpbs1 off-sz [of m]
  apply simp
  done

 from lbsm lbs1 have lbs': length bs' = sum-list (map sz [0..<m])
  apply (subst (asm) bs)
  apply simp
  done

 from lpbsm lpbs1 have lpbs': length pbs' = sum-list (map sz [0..<m])
  apply (subst (asm) pbs)
  apply simp
  done

 from off-sz have off-sz': (⋀i. i < m ⟹ off i = sum-list (map sz [0..<i])) by
auto

 {
  fix i
  assume bound: i < m
  hence bound': i < Suc m by simp
  from val [OF this]
  have v-i: v i (take (sz i) (drop (off i) pbs)) =
            take (sz i) (drop (off i) bs) .

  from off-sz [OF bound'] have off-i: off i = sum-list (map sz [0..<i]).
  from off-sz [of m] have off-m: off m = sum-list (map sz [0..<m]) by simp

  from bound have Suc i ≤ m by simp
  from sum-list-le-prefix [where n=0, OF - this]
  have ∗: sum-list (map sz [0..<Suc i]) ≤ sum-list (map sz [0..<m]) by simp
  then have bound-bs': off i + sz i ≤ length bs' using off-i lbs' by simp
  from ∗ have bound-pbs': off i + sz i ≤ length pbs' using off-i lpbs' by simp

  have take-eq1: take (sz i) (drop (off i) bs) = take (sz i) (drop (off i) bs')
   apply (subst bs)
   apply (subst take-drop-append-first [OF bound-bs'])
   apply simp
   done
  have take-eq2: take (sz i) (drop (off i) pbs) = take (sz i) (drop (off i) pbs')
   apply (subst pbs)
   apply (subst take-drop-append-first [OF bound-pbs'])
   apply simp
```

```
    done

  from v-i
  have v i (take (sz i) (drop (off i) pbs')) =
      take (sz i) (drop (off i) bs')
    by (simp add: take-eq1 take-eq2)
 } note val' = this

 from no-overflow
 have no-overflow': unat a + length bs' ≤ addr-card
  by (simp add: bs)
 {
  fix i
  assume bound: i < m
  hence bound': i < Suc m by simp

  from bound have Suc i ≤ m by simp
  from sum-list-le-prefix [where n=0, OF - this]
   have sum-le: sum-list (map sz [0..<Suc i]) ≤ sum-list (map sz [0..<m]) by
simp
  from off-sz [OF bound'] have off-i: off i = sum-list (map sz [0..<i]).
  have take (sz i) (drop (off i) bs) = take (sz i) (drop (off i) bs')
    apply (subst bs)
    using lbs' sum-le off-i
    apply simp
    done
 } note take-drop-eq = this

 from take-drop-eq
 have map-eq: map (λi. take (sz i) (drop (off i) bs)) [0..<m] =
              map (λi. take (sz i) (drop (off i) bs')) [0..<m]
  by simp

 {
  fix i
  assume bound: i < m
  hence bound': i < Suc m by simp

  from bound have Suc i ≤ m by simp
  from sum-list-le-prefix [where n=0, OF - this]
   have sum-le: sum-list (map sz [0..<Suc i]) ≤ sum-list (map sz [0..<m]) by
simp
  from off-sz [OF bound'] have off-i: off i = sum-list (map sz [0..<i]).
  have take (sz i) (drop (off i) pbs) = take (sz i) (drop (off i) pbs')
    apply (subst pbs)
    using lpbs' sum-le off-i
    apply simp
    done
 } note take-drop-eq' = this
```

from *take-drop-eq′*
have *map-eq′*: *map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *pbs*)) [*0*..<*m*] =
map (λ*i*. *take* (*sz i*) (*drop* (*off i*) *pbs′*)) [*0*..<*m*]
by *simp*

from *bs-partition* **have** *bs′-partition*: *bs′* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *bs′*)) [*0*..<*m*])
apply (*subst map-eq* [*symmetric*])
apply (*simp add*: *bs′-def*)
done

from *pbs-partition* **have** *pbs′-partition*: *pbs′* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *pbs′*)) [*0*..<*m*])
apply (*subst map-eq′* [*symmetric*])
apply (*simp add*: *pbs′-def*)
done

from *lbs′ off-sz* [*of m*]
have *off-m*: *off m* = *length bs′*
by *simp*

have *vm-eq*: (*v m* (*take* (*sz m*) (*drop* (*length bs′*) *pbs′*) @
take (*sz m* − (*length pbs′* − *length bs′*)) (*drop* (*length bs′* − *length pbs′*) *pbsm*))) = *bsm*
by (*metis bsm-def drop-append length-drop lessI off-m pbs take-append val*)

have *fold-pbs′*:
(*fold*
(λ*i*. *heap-update-list* (*a* + *word-of-nat* (*off i*)) (*v i* (*take* (*sz i*) (*drop* (*off i*) *pbs*))))
[*0*..<*m*] *h*) =
(*fold*
(λ*i*. *heap-update-list* (*a* + *word-of-nat* (*off i*)) (*v i* (*take* (*sz i*) (*drop* (*off i*) *pbs′*))))
[*0*..<*m*] *h*)
apply (*rule fold-cong*)
using *take-drop-eq′* **by** *auto*

note *hyp* = *Suc.hyps* [**where** *a*=*a* **and** *bs*=*bs′* **and** *pbs*=*pbs′* **and** *h*=*h*, *OF no-overflow′ bs′-partition lbs′ pbs′-partition lpbs′ off-sz′ val′*]
show *?case*
apply (*subst unroll*)
apply (*subst fold-append*)
apply (*simp only*: *comp-def*)
apply (*subst bs*)
apply (*subst fold-pbs′*)

apply (*subst pbs*)
apply (*subst heap-update-list-concat-unfold*)
apply (*subst hyp* [*symmetric*])
apply *assumption*
apply (*simp add*: *off-m vm-eq*)
done
qed

lemma *heap-update-list-fold-partition*:
fixes *v*:: *nat* ⇒ *byte list* ⇒ *byte list*
and *sz*:: *nat* ⇒ *nat*
and *off*:: *nat* ⇒ *nat*
assumes *no-overflow*: *unat a* + *length bs* ≤ *addr-card*
assumes *partition*: *bs* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *bs*)) [*0*..<*m*])
assumes *lbs*: *length bs* = *sum-list* (*map sz* [*0*..<*m*])
assumes *off-sz*: ⋀*i*. *i* < *m* ⟹ *off i* = *sum-list* (*map sz* [*0*..<*i*])
assumes *val*: ⋀*i*. *i* < *m* ⟹
v i (*take* (*sz i*) (*drop* (*off i*) (*heap-list h* (*length bs*) *a*))) =
(*take* (*sz i*) (*drop* (*off i*) *bs*))
shows
heap-update-list a bs h =
fold (λ*i h*. *heap-update-list*
(*a* + *word-of-nat* (*off i*))
(*v i* ((*heap-list h* (*sz i*) (*a* + *word-of-nat* (*off i*)))))
h)
[*0*..<*m*] *h*
using *no-overflow partition lbs off-sz val*
proof (*induct m arbitrary*: *a bs h*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
from *Suc.prems* **obtain**
no-overflow: *unat a* + *length bs* ≤ *addr-card* **and**
bs-partition: *bs* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *bs*)) [*0*..<*Suc m*])
and
lbs: *length bs* = *sum-list* (*map sz* [*0*..<*Suc m*]) **and**
off-sz: ⋀*i*. *i* < *Suc m* ⟹ *off i* = *sum-list* (*map sz* [*0*..<*i*]) **and**
val: ⋀*i*. *i* < *Suc m* ⟹
v i (*take* (*sz i*) (*drop* (*off i*) (*heap-list h* (*length bs*) *a*))) =
take (*sz i*) (*drop* (*off i*) *bs*)
by *clarsimp*
have *unroll*: [*0*..<*Suc m*] = [*0*..<*m*] @ [*m*]
by (*rule upt-Suc-snoc*)

define *bs'* **where** *bs'* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *bs*)) [*0*..<*m*])
define *bsm* **where** *bsm* = *take* (*sz m*) (*drop* (*off m*) *bs*)

from *bs-partition* **have** *bs*: *bs* = *bs'* @ *bsm*

unfolding *bsm-def bs'-def*
apply (*subst* (*asm*) *unroll*)
apply *simp*
done

from *lbs* **have** *lbs1*: *length bs* = *sum-list* (*map sz* [*0..<m*]) + *sz m*
apply (*subst* (*asm*) *unroll*)
apply *simp*
done

have *lbsm*: *length bsm* = *sz m*
apply (*simp add*: *bsm-def*)
using *lbs1 off-sz* [*of m*]
apply *simp*
done

with *lbs1* **have** *lbs'*: *length bs'* = *sum-list* (*map sz* [*0..<m*])
apply (*subst* (*asm*) *bs*)
apply *simp*
done

from *off-sz* **have** *off-sz'*: (⋀*i*. *i* < *m* ⟹ *off i* = *sum-list* (*map sz* [*0..<i*])) **by** *auto*

{
fix *i*
assume *bound*: *i* < *m*
hence *bound'*: *i* < *Suc m* **by** *simp*
from *val* [*OF this*]
have *v-i*: *v i* (*take* (*sz i*) (*drop* (*off i*) (*heap-list h* (*length bs*) *a*))) =
take (*sz i*) (*drop* (*off i*) *bs*) **.**

from *off-sz* [*OF bound'*] **have** *off-i*: *off i* = *sum-list* (*map sz* [*0..<i*])**.**
from *off-sz* [*of m*] **have** *off-m*: *off m* = *sum-list* (*map sz* [*0..<m*]) **by** *simp*

from *bound* **have** *Suc i* ≤ *m* **by** *simp*
from *sum-list-le-prefix* [**where** *n=0*, *OF - this*]
have *sum-list* (*map sz* [*0..<Suc i*]) ≤ *sum-list* (*map sz* [*0..<m*]) **by** *simp*
then have *bound-bs'*: *off i* + *sz i* ≤ *length bs'* **using** *off-i lbs'* **by** *simp*

have *take-eq1*: *take* (*sz i*) (*drop* (*off i*) *bs*) = *take* (*sz i*) (*drop* (*off i*) *bs'*)
apply (*subst bs*)
apply (*subst take-drop-append-first* [*OF bound-bs'*])
apply *simp*
done
have *take-eq2*:
take (*sz i*) (*drop* (*off i*) (*heap-list h* (*length bs*) *a*)) =
take (*sz i*) (*drop* (*off i*) (*heap-list h* (*length bs'*) *a*))

```
    apply (subst lbs1)
    apply (subst lbs')
    apply (subst heap-list-split2)
    using bound-bs' [simplified lbs']
    apply simp
    done
  from v-i
  have v i (take (sz i) (drop (off i) (heap-list h (length bs') a))) =
      take (sz i) (drop (off i) bs')
    by (simp add: take-eq1 take-eq2)
 } note val' = this



 from no-overflow
 have no-overflow': unat a + length bs' ≤ addr-card
  by (simp add: bs)
 {
  fix i
  assume bound: i < m
  hence bound': i < Suc m by simp

  from bound have Suc i ≤ m by simp
  from sum-list-le-prefix [where n=0, OF - this]
   have sum-le: sum-list (map sz [0..<Suc i]) ≤ sum-list (map sz [0..<m]) by
simp
  from off-sz [OF bound'] have off-i: off i = sum-list (map sz [0..<i]).
  have take (sz i) (drop (off i) bs) = take (sz i) (drop (off i) bs')
    apply (subst bs)
    using lbs' sum-le off-i
    apply simp
    done
 } note take-drop-eq = this

 from take-drop-eq
 have map-eq: map (λi. take (sz i) (drop (off i) bs)) [0..<m] =
            map (λi. take (sz i) (drop (off i) bs')) [0..<m]
  by simp


 from bs-partition have bs'-partition: bs' = concat (map (λi. take (sz i) (drop
(off i) bs')) [0..<m])
  apply (subst map-eq [symmetric])
  apply (simp add: bs'-def)
  done

 from lbs' off-sz [of m]
 have off-m: off m = length bs'
  by simp
```

```
  have vm-eq: v m (heap-list (heap-update-list a bs' h) (sz m) (a + word-of-nat
(length bs'))) = bsm
  proof −
    from val [of m]
     have v m (take (sz m) (drop (off m) (heap-list h (length bs) a))) = bsm by
(simp add: bsm-def)
    moreover
    have disj: {a..+length bs'} ∩ {a + word-of-nat (length bs')..+sz m} = {}
      apply (simp add: lbs')
      apply (rule intvl-off-disj')
      apply simp
      using no-overflow [simplified lbs1]
      apply (simp)
      done

    have heap-list (heap-update-list a bs' h) (sz m) (a + word-of-nat (length bs'))
=
          take (sz m) (drop (off m) (heap-list h (length bs) a))
      apply (subst heap-list-update-disjoint-same [OF disj])
      apply (subst lbs1 )
      apply (subst lbs' )
      apply (subst heap-list-take-drop' [OF no-overflow [simplified lbs1]])
       apply (simp add:  off-m [simplified lbs'])
      apply (simp add:  off-m [simplified lbs'])
      done

    ultimately show ?thesis by simp
  qed

  show ?case
    apply (subst unroll)
    apply (subst fold-append)
    apply (simp only: comp-def)
    apply (subst bs)
    apply (subst heap-update-list-concat-unfold)
     apply (subst Suc.hyps [where a=a and bs=bs' and h=h, OF no-overflow'
bs'-partition lbs' off-sz' val', symmetric])
    apply assumption
    apply (simp add: off-m vm-eq)
    done
qed

lemma heap-list-map-partition:
  fixes  sz:: nat ⇒ nat
    and off:: nat ⇒ nat
  assumes no-overflow: unat a + n ≤ addr-card
  assumes lbs: n = sum-list (map sz [0..<m])
  assumes off-sz: ⋀i. i < m ⟹ off i = sum-list (map sz [0..<i])
```

```
  shows
   heap-list h n a =
      concat (map (λi. heap-list h (sz i) (a + word-of-nat (off i))) [0..<m])
  using no-overflow lbs off-sz
proof (induct m arbitrary: n a)
  case 0
  then show ?case by simp
next
  case (Suc m)
  from Suc.prems obtain
    no-overflow: unat a + n ≤ addr-card and
    lbs: n = sum-list (map sz [0..<Suc m]) and
    off-sz: ⋀i. i < Suc m ⟹ off i = sum-list (map sz [0..<i])
    by clarsimp
  have unroll: [0..<Suc m] = [0..<m] @ [m]
    by (rule upt-Suc-snoc)

  from lbs have lbs1: n = sum-list (map sz [0..<m]) + sz m
    apply (subst (asm) unroll)
    apply simp
    done

  from no-overflow lbs1
  have no-overflow': unat a + (n − sz m) ≤ addr-card
    by simp
  from lbs1
  have lbs1': n − sz m = sum-list (map sz [0..<m])
    by simp
  from lbs1 lbs1'
  have sz-m: (n − (n − sz m)) = sz m
    by simp

  from heap-list-split [where k=n − sz m and n = n and h=h and x=a]
  have split: heap-list h n a = heap-list h (n − sz m) a @ heap-list h (sz m) (a +
word-of-nat (n − sz m))
    by (simp add: sz-m)

  have hyp: heap-list h (n − sz m) a = concat (map (λi. heap-list h (sz i) (a +
word-of-nat (off i))) [0..<m])
    apply (rule  Suc.hyps [OF no-overflow' lbs1'])
    using off-sz
    by auto

  from off-sz [of m] lbs1
  have off-m: (n − sz m) = off m
    by simp
  show ?case
    apply (subst split)
    apply (subst unroll)
```

```
    apply (simp add: hyp )
    apply (simp add: off-m)
    done
qed

lemma array-partition:
assumes lbs: length bs = m * n
shows concat (map (λi. take n (drop (i * n) bs)) [0..<m]) = bs
  using lbs
proof (induct m arbitrary: bs n)
  case 0
  then show ?case by simp
next
  case (Suc m)
  have lbs: length bs = Suc m * n by fact
  hence lbs: length bs = (m * n) + n
    by simp
  define bs' where bs' = take (m * n) bs
  define bs-last where bs-last = drop (m * n) bs
  from lbs have bs: bs = bs' @ bs-last
    by (simp add: bs'-def bs-last-def)

  from lbs bs
  have lbs': length bs' = m * n
     by (metis add-diff-cancel-left' add-diff-cancel-right' bs-last-def eq-concat-lenD
length-drop)

  from lbs bs lbs' have lbs-last:  length bs-last = n
    by auto

  {
    fix i
    assume bound: i < m
    have  take n (drop (i * n) bs) =  take n (drop (i * n) bs')
      apply (subst bs)
      using lbs' lbs-last bound
      apply simp
      by (metis Suc-leI mult-Suc mult-le-mono order-refl)
  }
  then
  have map-eq: (map (λi. take n (drop (i * n) bs)) [0..<m]) = (map (λi. take n
(drop (i * n) bs')) [0..<m])
    by simp

  note hyp = Suc.hyps [OF lbs']
  show ?case
    apply simp
    apply (simp add: map-eq hyp )
    apply (simp add: bs-last-def [symmetric])
```

```
    using lbs-last
    apply (simp add: bs)
    done
qed

lemma sum-list-const-fun: sum-list (map (λ-. n::nat) [0..<m]) = m * n
  by (induct m) auto

lemmas export-uinfo-adjust-ti [simp] =  export-tag-adjust-ti(1)[rule-format]


lemma heap-update-list-array:
  fixes arr:: ('a::array-outer-max-size['b::array-max-count])
  fixes p:: ('a['b]) ptr
  assumes cgrd: c-guard p
  shows
heap-update-list (ptr-val p) (to-bytes arr (heap-list h (size-of TYPE('a['b])) (ptr-val
p))) h =
    fold
      (λi h. heap-update-list (ptr-val (array-ptr-index p False i))
              (to-bytes (arr.[i])
               (heap-list h (size-of TYPE('a)) (ptr-val (array-ptr-index p False i))))
              h)
      [0..<CARD('b)] h
proof −
  define xbs where
    xbs= (to-bytes arr (heap-list h (size-of TYPE('a['b])) (ptr-val p)))
  define sz::nat ⇒ nat where
    sz = (λ-. size-of TYPE('a))
  define off::nat ⇒ nat where
    off = (λi. i * size-of TYPE('a))
  define v::nat ⇒ byte list ⇒ byte list where
    v = (λi bs. (to-bytes (arr.[i]) bs))

  have lxbs: length xbs = CARD('b) * size-of TYPE('a)
    by (simp add: xbs-def)

  from c-guard-no-wrap' [OF cgrd] lxbs
  have no-overflow: unat (ptr-val p) + length xbs ≤ addr-card
    by simp

 have partition: xbs = concat (map (λi. take (sz i) (drop (off i) xbs)) [0..<CARD('b)])

    apply (simp add: sz-def off-def)
    apply (rule array-partition [symmetric, OF lxbs])
    done

  have sum-list (map sz [0..<CARD('b)]) = CARD('b) * size-of TYPE('a)
    apply (simp add: sz-def)
```

apply (*rule sum-list-const-fun*)
done

with *lxbs* **have** *lxbs-sum*: *length xbs* = *sum-list* (*map sz* [*0..<CARD('b)*]) **by** *simp*

have *off-sz*: (⋀*i*. *i* < *CARD('b)* ⟹ *off i* = *sum-list* (*map sz* [*0..<i*]))
unfolding *off-def sz-def* **using** *sum-list-const-fun*
by *auto*

{
fix *i*
assume *i-bound*: *i* < *CARD('b)*
have *eq-padding*:
eq-padding (*typ-uinfo-t TYPE('a['b])*)
xbs
(*heap-list h* (*size-of TYPE('a['b])*) (*ptr-val p*))
using *eq-padding-to-bytes heap-list-length xbs-def* **by** *blast*

from *field-lookup-array* [*OF i-bound*, **where** *i=0*, *simplified*]
have *fl*: *field-lookup* (*typ-info-t TYPE('a['b])*) [*replicate i CHR ''1''*] *0* =
Some (*adjust-ti* (*typ-info-t TYPE('a)*) (λ*x*. *x*.[*i*]) (λ*x f*. *Arrays.update f i x*), *i* ∗ *size-of TYPE('a)*) **.**

have *exp*:
(*export-uinfo*
(*adjust-ti* (*typ-info-t TYPE('a)*) (λ*x*::*'a['b]*. (*x*.[*i*])) (λ*x f*. *Arrays.update f i x*))) =
typ-uinfo-t TYPE('a)
apply (*subst export-uinfo-adjust-ti*)
apply (*rule fg-cons-array* [*OF i-bound*])
apply *simp*
apply (*simp add*: *typ-uinfo-t-def*)
done

from *xmem-type-field-lookup-eq-padding-focus* [*OF fl - - eq-padding*]
have
eq-padding (*typ-uinfo-t TYPE('a)*)
(*take* (*size-of TYPE('a)*) (*drop* (*i* ∗ *size-of TYPE('a)*) *xbs*))
(*take* (*size-of TYPE('a)*) (*drop* (*i* ∗ *size-of TYPE('a)*) (*heap-list h* (*size-of TYPE('a['b])*) (*ptr-val p*))))
using *lxbs*
by *simp* (*simp add*: *exp size-of-def*)

hence *eq1*:
to-bytes (*arr*.[*i*]) (*take* (*size-of TYPE('a)*) (*drop* (*i* ∗ *size-of TYPE('a)*) *xbs*))
=
to-bytes (*arr*.[*i*]) (*take* (*size-of TYPE('a)*) (*drop* (*i* ∗ *size-of TYPE('a)*)

```
(heap-list h (size-of TYPE('a['b])) (ptr-val p))))
      by (meson eq-padding-to-bytes-eq)
    have v i (take (sz i) (drop (off i) (heap-list h (length xbs) (ptr-val p)))) =
        take (sz i) (drop (off i) xbs)
      unfolding v-def lxbs sz-def off-def
      apply (simp add: xbs-def)

        using access-ti-array-index' [OF i-bound lxbs, where arr=arr, OF refl,
simplified xbs-def, simplified to-bytes-def [symmetric]]
      apply (simp only: xbs-def [symmetric])
      apply (subst (asm) eq1)
      apply simp
      using eq-padding eq-padding-to-bytes-eq by fastforce
  } note v-padding = this

  {
    fix i
    assume i-bound: i < CARD('b)
    have ptr-val p + word-of-nat i * word-of-nat (size-of TYPE('a))
              =
          ptr-val (PTR-COERCE('a['b] → 'a) p +p int i)
      by (simp add: CTypesDefs.ptr-add-def)
  } note deref-conv = this
  note partition = heap-update-list-fold-partition [where a=ptr-val p and bs=xbs
and sz=sz and off=off and v=v
      and h=h and
   m=CARD('b), OF no-overflow partition lxbs-sum off-sz v-padding, simplified]
  show ?thesis
    apply (simp only: xbs-def [symmetric])
    apply (subst partition)
    apply (rule fold-cong)
      apply simp
     apply simp
    apply (simp add: v-def sz-def off-def array-ptr-index-def deref-conv)
    done
qed

lemma heap-update-array:
  fixes arr:: ('a::array-outer-max-size['b::array-max-count])
  fixes p:: ('a['b]) ptr
  assumes cgrd: c-guard p
  shows
heap-update p arr h =
    fold
      (λi h. heap-update (array-ptr-index p False i) (arr.[i]) h)
      [0..<CARD('b)] h
  using heap-update-list-array [OF cgrd]
  by (simp only: heap-update-def)
```

lemma *heap-update-array-pointless*:
fixes *arr*:: (*'a*::*array-outer-max-size*[*'b*::*array-max-count*])
fixes *p*:: (*'a*[*'b*]) *ptr*
assumes *cgrd*: *c-guard p*
shows
heap-update p arr =
fold
(λ*i h*. *heap-update* (*array-ptr-index p False i*) (*arr*.[*i*]) *h*)
[*0*..<*CARD*(*'b*)]
using *heap-update-array* [*OF cgrd*]
by (*simp add*: *fun-eq-iff*)

lemma *heap-update-padding-list-array*:
fixes *arr*:: (*'a*::*array-outer-max-size*[*'b*::*array-max-count*])
fixes *p*:: (*'a*[*'b*]) *ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *CARD*(*'b*) ∗ *size-of TYPE*(*'a*)
shows
heap-update-list (*ptr-val p*) (*to-bytes arr bs*) *h* =
fold
(λ*i h*. *heap-update-list* (*ptr-val* (*array-ptr-index p False i*))
(*to-bytes* (*arr*.[*i*])
(*take* (*size-of TYPE*(*'a*)) (*drop* (*i* ∗ *size-of TYPE*(*'a*)) *bs*)))
h)
[*0*..<*CARD*(*'b*)] *h*
proof −
define *xbs* **where**
xbs= (*to-bytes arr bs*)
define *sz*::*nat* ⇒ *nat* **where**
sz = (λ-. *size-of TYPE*(*'a*))
define *off*::*nat* ⇒ *nat* **where**
off = (λ*i*. *i* ∗ *size-of TYPE*(*'a*))
define *v*::*nat* ⇒ *byte list* ⇒ *byte list* **where**
v = (λ*i bs*. (*to-bytes* (*arr*.[*i*]) *bs*))
have *lxbs*: *length xbs* = *CARD*(*'b*) ∗ *size-of TYPE*(*'a*)
by (*simp add*: *xbs-def lbs*)

from *c-guard-no-wrap'* [*OF cgrd*] *lxbs*
have *no-overflow*: *unat* (*ptr-val p*) + *length xbs* ≤ *addr-card*
by *simp*

have *partition*: *xbs* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *xbs*)) [*0*..<*CARD*(*'b*)])

apply (*simp add*: *sz-def off-def*)
apply (*rule array-partition* [*symmetric*, *OF lxbs*])
done

have *partition-bs*: *bs* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *bs*)) [*0*..<*CARD*('*b*)])

apply (*simp add*: *sz-def off-def*)
apply (*rule array-partition* [*symmetric*, *OF lbs*])
done

have ∗: *sum-list* (*map sz* [*0*..<*CARD*('*b*)]) = *CARD*('*b*) ∗ *size-of TYPE*('*a*)
apply (*simp add*: *sz-def*)
apply (*rule sum-list-const-fun*)
done

with *lxbs* **have** *lxbs-sum*: *length xbs* = *sum-list* (*map sz* [*0*..<*CARD*('*b*)]) **by** *simp*

from *lbs* ∗ **have** *lbs-sum*: *length bs* = *sum-list* (*map sz* [*0*..<*CARD*('*b*)]) **by** *simp*
have *off-sz*: (⋀*i*. *i* < *CARD*('*b*) ⟹ *off i* = *sum-list* (*map sz* [*0*..<*i*]))
unfolding *off-def sz-def* **using** *sum-list-const-fun*
by *auto*

have *val-eq*: ⋀*i*. *i* < *CARD*('*b*) ⟹ *v i* (*take* (*sz i*) (*drop* (*off i*) *bs*)) = *take* (*sz i*) (*drop* (*off i*) *xbs*)
unfolding *off-def sz-def v-def xbs-def*
apply (*simp add*: *to-bytes-def*)
apply (*subst access-ti-array-index'* [**where** *xbs*=*bs*, *OF* - *lbs*])
apply *auto*
done

have *val-eq'*:
⋀*i*. *i* < *CARD*('*b*) ⟹ (*v i* (*take* (*sz i*) (*drop* (*off i*) *bs*))) =
(*to-bytes* (*arr*.[*i*]) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*)))
unfolding *off-def sz-def v-def*
by *simp*

have *idx-eq*: ⋀*i*. *i* < *CARD*('*b*) ⟹
(*ptr-val p* + *word-of-nat* (*off i*)) = (*ptr-val* (*array-ptr-index p False i*))
unfolding *off-def array-ptr-index-def*
by (*auto simp add*: *c-type-class.ptr-add-def*)

note *partition* = *heap-update-list-padding-fold-partition* [**where** *a*=*ptr-val p* **and** *bs*= *xbs* **and** *pbs* = *bs* **and** *sz*=*sz* **and** *off*=*off* **and** *v*=*v* **and** *h*=*h* **and** *m*=*CARD*('*b*), *OF no-overflow partition lxbs-sum partition-bs lbs-sum off-sz val-eq*]
show *?thesis*
apply (*simp only*: *xbs-def* [*symmetric*])
apply (*subst partition*)
apply (*assumption*)
apply (*rule fold-cong*)

apply (*auto simp add*: *val-eq' idx-eq*)
done
qed

lemma *heap-update-padding-array*:
fixes *arr*:: ('*a*::*array-outer-max-size*['*b*::*array-max-count*])
fixes *p*:: ('*a*['*b*]) *ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *CARD*('*b*) ∗ *size-of TYPE*('*a*)
shows
heap-update-padding p arr bs h =
fold
(λ*i h*. *heap-update-padding* (*array-ptr-index p False i*) (*arr*.[*i*]) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*)) *h*)
[*0*..<*CARD*('*b*)] *h*
using *heap-update-padding-list-array* [*OF cgrd lbs*]
by (*simp only*: *heap-update-padding-def*)

lemma *heap-update-padding-array-pointless*:
fixes *arr*:: ('*a*::*array-outer-max-size*['*b*::*array-max-count*])
fixes *p*:: ('*a*['*b*]) *ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *CARD*('*b*) ∗ *size-of TYPE*('*a*)
shows
heap-update-padding p arr bs =
fold
(λ*i h*. *heap-update-padding* (*array-ptr-index p False i*) (*arr*.[*i*]) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*)) *h*)
[*0*..<*CARD*('*b*)]
using *heap-update-padding-array* [*OF cgrd lbs*]
by (*simp add*: *fun-eq-iff*)

lemma *c-guard-array-ptr-index*:
fixes *p*:: ((*'a* :: *mem-type*)['*b* :: *finite*]) *ptr*
assumes *cgrd*: *c-guard p*
assumes *bound*: *n* < *CARD*('*b*)
shows *c-guard* (*array-ptr-index p coerce n*)
using *cgrd bound*
apply (*simp add*: *array-ptr-index-def c-guard-def*)
apply (*clarsimp*, *safe*)
subgoal
apply (*simp add*: *ptr-aligned-def*)
by (*metis ptr-aligned-def ptr-aligned-plus ptr-val-ptr-coerce*)
subgoal
supply *fun-upd-apply*[*simp del*]
apply (*simp add*: *c-null-guard-def*)
apply (*simp add*: *CTypesDefs.ptr-add-def field-simps*)
by (*auto simp add*: *intvl-def*)
(*metis* (*no-types*, *opaque-lifting*) *Abs-fnat-hom-mult add.commute group-cancel.add1*

```
nat-index-bound word-of-nat-plus)
  done

lemma heap-update-array-update:
  assumes n: n < CARD('b :: array-max-count)
   assumes size: CARD('b) * size-of TYPE('a :: array-outer-max-size) < 2 ^
addr-bitsize
  assumes cgrd: c-guard p
  shows heap-update p (Arrays.update (arr :: 'a['b]) n v) hp
      = heap-update (array-ptr-index p False n) v (heap-update p arr hp)
 proof −

    have P: ⋀x k. ⟦ x < CARD('b); k < size-of TYPE('a) ⟧
        ⟹ unat (of-nat x * of-nat (size-of TYPE('a)) + (of-nat k :: addr))
              = x * size-of TYPE('a) + k
    using size
    supply unsigned-of-nat [simp del]
    apply (cases size-of TYPE('a), simp-all)
    apply (cases CARD('b), simp-all)
    apply (subst unat-add-lem[THEN iffD1])
     apply (simp add: unat-word-ariths unat-of-nat less-Suc-eq-le)
    subgoal premises prems for x k nat nata
    proof −
      have Suc x * size-of TYPE('a) < 2 ^ addr-bitsize
        using prems
        apply simp
        apply (erule order-le-less-trans[rotated], simp add: add-mono)
        done
      then show ?thesis using prems by simp
    qed
    apply (subst unat-mult-lem[THEN iffD1])
     apply (simp add: unat-of-nat unat-add-lem[THEN iffD1])
     apply (rule order-less-le-trans, erule order-le-less-trans[rotated],
           rule add-mono, simp+)
      apply (simp add: less-Suc-eq-le trans-le-add2)
     apply simp
    apply (simp add: unat-of-nat unat-add-lem[THEN iffD1])
    done

  let ?key-upd = heap-update (array-ptr-index p False n) v
  note commute = fold-commute-apply[where h=?key-upd
      and xs=[Suc n ..< CARD('b)], where g=f' and f=f' for f']

  from cgrd n
  have cgrd': c-guard (array-ptr-index p False n)
    by (rule c-guard-array-ptr-index)

  show ?thesis using n
    apply (simp add: heap-update-array [OF cgrd] split-upt-on-n[OF n]
```

```
                )
    apply (subst commute)
    subgoal for x
      apply (rule ext, simp)
      apply (rule heap-update-commute, simp-all add: ptr-add-def)
    apply (simp add: array-ptr-index-def CTypesDefs.ptr-add-def intvl-def Suc-le-eq)
      apply (rule set-eqI, clarsimp)
      apply (drule word-unat.Rep-inject[THEN iffD2])
      apply (clarsimp simp: P nat-eq-add-iff1)
      apply (cases x, simp-all add: less-Suc-eq-le Suc-diff-le)
      done
    subgoal
      apply (subst heap-update-collapse)
      apply (simp cong: fold-cong')
      done
    done
qed


lemma heap-update-array-element'':
  fixes p' :: (('a ::array-outer-max-size)['b::array-max-count]) ptr
  fixes p :: ('a :: array-outer-max-size) ptr
  fixes hp w
  assumes p: p = array-ptr-index p' False n
  assumes n: n < CARD('b)
  assumes cgrd: c-guard p'
  assumes size: CARD('b) * size-of TYPE('a) < 2 ^ addr-bitsize
  shows heap-update p' (Arrays.update (h-val hp p') n w) hp
      = heap-update p w hp
  apply (subst heap-update-array-update[OF n size cgrd])
  apply (simp add: heap-update-id p)
  done

lemmas heap-update-array-element'
    = heap-update-array-element''[simplified array-ptr-index-simps]

lemma (in array-outer-max-size) array-count-size:
  CARD('b :: array-max-count) * size-of TYPE('a ) < 2 ^ addr-bitsize
  using array-outer-max-size-ax[] array-max-count-ax[where 'a='b]
  apply (clarsimp dest!: nat-le-Suc-less-imp)
  apply (drule(1) mult-mono, simp+)
  done

lemmas heap-update-array-element
    = heap-update-array-element'[OF refl - - array-count-size]

primrec
  field-names-u :: typ-uinfo ⇒ typ-uinfo ⇒
      (qualified-field-name) list and
```

field-names-struct-u :: *typ-uinfo-struct* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-list-u :: *typ-uinfo-tuple list* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-tuple-u :: *typ-uinfo-tuple* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list*
where
tufs0: *field-names-u* (*TypDesc algn st nm*) *t* = (*if t*= (*TypDesc algn st nm*) *then*
[[]] *else field-names-struct-u st t*)

| *tufs1*: *field-names-struct-u* (*TypScalar m algn d*) *t* = []
| *tufs2*: *field-names-struct-u* (*TypAggregate xs*) *t* = *field-names-list-u xs t*

| *tufs3*: *field-names-list-u* [] *t* = []
| *tufs4*: *field-names-list-u* (*x*#*xs*) *t* = *field-names-tuple-u x t*@*field-names-list-u xs t*

| *tufs5*: *field-names-tuple-u* (*DTuple s f d*) *t* = *map* (λ*fs*. *f*#*fs*) (*field-names-u s t*)

lemma *field-names-u-field-names-export-uinfo-conv*:
fixes *t* :: ('*a*, '*b*) *typ-info* **and**
st :: ('*a*, '*b*) *typ-info-struct* **and**
ts :: ('*a*, '*b*) *typ-info-tuple list* **and**
x :: ('*a*, '*b*) *typ-info-tuple*
shows
field-names-u (*export-uinfo t*) *s* = *field-names t s*
field-names-struct-u (*map-td-struct field-norm* (λ-. ()) *st*) *s* = *field-names-struct st s*
field-names-list-u (*map-td-list field-norm* (λ-. ()) *ts*) *s* = *field-names-list ts s*
field-names-tuple-u (*map-td-tuple field-norm* (λ-. ()) *x*) *s* = *field-names-tuple x s*
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*auto simp add*: *export-uinfo-def*)

primrec
all-field-names :: ('*a*, '*b*) *typ-desc* ⇒
(*qualified-field-name*) *list* **and**
all-field-names-struct :: ('*a*, '*b*) *typ-struct* ⇒
(*qualified-field-name*) *list* **and**
all-field-names-list :: (('*a*, '*b*) *typ-desc*, *field-name*, '*b*) *dt-tuple list* ⇒
(*qualified-field-name*) *list* **and**
all-field-names-tuple :: (('*a*, '*b*) *typ-desc*, *field-name*, '*b*) *dt-tuple* ⇒
(*qualified-field-name*) *list*
where
afs0: *all-field-names* (*TypDesc algn st nm*) =
[[]] @ *all-field-names-struct st*

| *afs1*: *all-field-names-struct* (*TypScalar m algn d*) = []
| *afs2*: *all-field-names-struct* (*TypAggregate xs*) = *all-field-names-list xs*

| *afs3*: *all-field-names-list* [] = []
| *afs4*: *all-field-names-list* (*x*#*xs*) = *all-field-names-tuple x* @ *all-field-names-list xs*

| *afs5*: *all-field-names-tuple* (*DTuple s f d*) = *map* (λ*fs*. *f*#*fs*) (*all-field-names s*)

lemma *field-lookup-all-field-names*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
field-lookup t f m = *Some* (*s*, *n*) ⟹ *f* ∈ *set* (*all-field-names t*) **and**
field-lookup-struct st f m = *Some* (*s*, *n*) ⟹ *f* ∈ *set* (*all-field-names-struct st*)
and
field-lookup-list ts f m = *Some* (*s*, *n*) ⟹ *f* ∈ *set* (*all-field-names-list ts*) **and**
field-lookup-tuple x f m = *Some* (*s*, *n*) ⟹ *f* ∈ *set* (*all-field-names-tuple x*)
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m n s* **and** *f m n s* **and** *f m n s* **and** *f m n s*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*auto split*: *if-split-asm*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **by** (*auto split*: *option.splits*)
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** (*auto split*: *if-split-asm*)
(*metis imageI list.exhaust-sel*)
qed

lemma *field-names-subset-all-field-names*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
set (*field-names-u t s*) ⊆ *set* (*all-field-names t*)
set (*field-names-struct-u st s*) ⊆ *set* (*all-field-names-struct st*)
set (*field-names-list-u ts s*) ⊆ *set* (*all-field-names-list ts*)

set (*field-names-tuple-u x s*) ⊆ *set* (*all-field-names-tuple x*)
by (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *s* **and** *s* **and** *s* **and** *s*) *auto*

lemma *empty-all-field-names*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
[] ∈ *set* (*all-field-names t*)
[] ∉ *set* (*all-field-names-struct st*)
[] ∉ *set* (*all-field-names-list ts*)
[] ∉ *set* (*all-field-names-tuple x*)
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *empty-field-names-u*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
True
[] ∉ *set* (*field-names-struct-u st s*)
[] ∉ *set* (*field-names-list-u ts s*)
[] ∉ *set* (*field-names-tuple-u x s*)
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *non-empty-field-names-u*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
field-names-u t s ≠ [] ⟹ ∃ *n*. (*s*, *n*) ∈ *td-set t m*
(*field-names-struct-u st s*) ≠ [] ⟹ ∃ *n*. (*s*, *n*) ∈ *td-set-struct st m*
(*field-names-list-u ts s*) ≠ [] ⟹ ∃ *n*. (*s*, *n*) ∈ *td-set-list ts m*
(*field-names-tuple-u x s*) ≠ [] ⟹ ∃ *n*. (*s*, *n*) ∈ *td-set-tuple x m*
by (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *s m* **and** *s m* **and** *s m* **and** *s m*)
fastforce+

lemma *td-set-size*:
(*s*, *n*) ∈ *td-set t m* ⟹ *size s* ≤ *size t*
(*s*, *n*) ∈ *td-set-struct st m* ⟹ *size s* ≤ *size st*
(*s*, *n*) ∈ *td-set-list ts m* ⟹ ∃ *t* ∈ *set ts*. *size s* ≤ *size* (*dt-fst t*)
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *size s* ≤ *size* (*dt-fst x*)
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *s n m* **and** *s n m* **and** *s n m* **and**

s n m)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *fastforce*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** (*auto simp add*: *less-Suc-eq less-imp-le td-set-list-size-lte*)
next
case *Nil-typ-desc*
then show *?case* **by** *simp*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** *auto*
qed

lemma *td-set-field-names-nonempty*:
(*s*, *n*) ∈ *td-set t m* ⟹ *field-names t* (*export-uinfo s*) ≠ []
(*s*, *n*) ∈ *td-set-struct st m* ⟹ *field-names-struct st* (*export-uinfo s*) ≠ []
(*s*, *n*) ∈ *td-set-list ts m* ⟹ *field-names-list ts* (*export-uinfo s*) ≠ []
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *field-names-tuple x* (*export-uinfo s*) ≠ []
by (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *n m* **and** *n m* **and** *n m* **and** *n m*)
auto

lemma *sub-typ-field-names-nonempty*:
assumes *s-t*: *s* ≤ *t*
shows *field-names t* (*export-uinfo s*) ≠ []
proof −
from *s-t* **obtain** *n* **where** (*s*, *n*) ∈ *td-set t 0*
by (*auto simp add*: *typ-tag-le-def*)
from *td-set-field-names-nonempty* (*1*) [*OF this*] **show** *?thesis*.
qed

lemma *sub-typ-export-uinfo-mono*:
assumes *s-t*: *s* ≤ *t*
shows *export-uinfo s* ≤ *export-uinfo t*
using *s-t*
by (*meson td-set-export-uinfoD typ-tag-le-def*)

lemma *descriptor-not-in-self*: (*TypDesc algn st nm*, *n*) ∉ *td-set-struct st m*
proof
assume (*TypDesc algn st nm*, *n*) ∈ *td-set-struct st m*
from *td-set-size*(*2*) [*OF this*] **show** *False* **by** *simp*
qed

lemma *field-names-struct-descriptor-empty*: *field-names-struct-u st* (*TypDesc algn st nm*) = []
using *descriptor-not-in-self non-empty-field-names-u*(*2*)
by *blast*

lemma *all-field-names-exists-field-names-u*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
f ∈ *set* (*all-field-names t*) ⟹ ∃ *s*. *f* ∈ *set* (*field-names-u t s*)
f ∈ *set* (*all-field-names-struct st*) ⟹ ∃ *s*. *f* ∈ *set* (*field-names-struct-u st s*)
f ∈ *set* (*all-field-names-list ts*) ⟹ ∃ *s*. *f* ∈ *set* (*field-names-list-u ts s*)
f ∈ *set* (*all-field-names-tuple x*) ⟹ ∃ *s*. *f* ∈ *set* (*field-names-tuple-u x s*)
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f* **and** *f* **and** *f* **and** *f*)
case (*TypDesc algn st nm*)
then show *?case*
proof (*cases f* = [])
case *True*
with *TypDesc* **show** *?thesis* **by** *auto*
next
case *False*
with *TypDesc* **obtain** *s* **where** *s*: *f* ∈ *set* (*field-names-struct-u st s*) **by** *auto*
with *field-names-struct-descriptor-empty* **have** *s* ≠ *TypDesc algn st nm* **by** *auto*
with *s False TypDesc* **show** *?thesis* **by** *auto*
qed
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case*
by *auto*
(*meson empty-all-field-names*(*1*) *imageI*)
qed

theorem *all-field-names-union-field-names-u-conv*: *set* (*all-field-names t*) = (⋃ *s*. *set* (*field-names-u t s*))

proof
show *set* (*all-field-names t*) ⊆ (⋃ *s*. *set* (*field-names-u t s*))
using *all-field-names-exists-field-names-u*
by *auto*
next
show (⋃ *s*. *set* (*field-names-u t s*)) ⊆ *set* (*all-field-names t*)
using *field-names-subset-all-field-names*(*1*)
by *auto*
qed

corollary *all-field-names-union-field-names-export-uinfo-conv*:
set (*all-field-names* (*export-uinfo t*)) = (⋃ *s*. *set* (*field-names t s*))
proof −
have (⋃ *s*. *set* (*field-names t s*)) = (⋃ *s*. *set* (*field-names-u* (*export-uinfo t*) *s*))
by (*simp add*: *field-names-u-field-names-export-uinfo-conv*)
with *all-field-names-union-field-names-u-conv* **show** *?thesis* **by** *metis*
qed

lemma *filter-same-eq*: (⋀*x*. *x* ∈ *set xs* ⟹ *P x* = *Q x*) ⟹ *filter P xs* = *filter Q xs*
by (*induct xs*) *auto*

lemma *field-lookup-tuple-hd-notin*: *n* ≠ *dt-snd x* ⟹ *field-lookup-tuple x* (*n* # *ns*) *m* = *None*
by (*cases x*) *auto*

lemma *field-lookup-list-hd-notin*: *n* ∉ *dt-snd* ‘ *set xs* ⟹ *field-lookup-list xs* (*n* # *ns*) *m* = *None*
by (*induct xs arbitrary*: *m*)
(*auto split*: *option.splits simp add*: *field-lookup-tuple-hd-notin*)

lemma *list-append-eq-split*: *xs1* = *xs2* ⟹ *ys1* = *ys2* ⟹ (*xs1* @ *ys1*) = (*xs2* @ *ys2*)
by *simp*

lemma *field-names-u-filter-all-field-names-conv*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
wf-desc t ⟹
(*field-names-u t s*) = *filter* (λ*f*. ∃ *n*. *field-lookup t f 0* = *Some* (*s*, *n*)) (*all-field-names t*)
wf-desc-struct st ⟹
field-names-struct-u st s = *filter* (λ*f*. ∃ *n*. *field-lookup-struct st f 0* = *Some* (*s*, *n*)) (*all-field-names-struct st*)
wf-desc-list ts ⟹
field-names-list-u ts s = *filter* (λ*f*. ∃ *n*. *field-lookup-list ts f 0* = *Some* (*s*, *n*))

```
(all-field-names-list ts)
 wf-desc-tuple x ⟹
   field-names-tuple-u x s = filter (λf. ∃ n. field-lookup-tuple x f 0 = Some (s, n))
(all-field-names-tuple x)
proof (induct t and st and ts and x arbitrary: s and s and s and s)
 case (TypDesc algn st nm)
  then show ?case
    apply (clarsimp simp add: empty-all-field-names, safe)
     apply (smt (verit, best) descriptor-not-in-self empty-all-field-names(2)
        filter-False td-set-struct-field-lookup-structD)
    by (metis (no-types, opaque-lifting) field-lookup-struct-empty option.simps(3))
next
  case (TypScalar n algn x)
  then show ?case by auto
next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x xs)
  from Cons-typ-desc.prems obtain
    wf-x: wf-desc-tuple x and
    wf-xs: wf-desc-list xs and
    distinct: dt-snd x ∉ dt-snd ‘ set xs by clarsimp
  have eq1:
    filter (λf. ∃ n. field-lookup-tuple x f 0 = Some (s, n)) (all-field-names-tuple x)
=
    filter
     (λf. ∃ n. (case field-lookup-tuple x f 0 of
                 None ⇒ field-lookup-list xs f (0 + size-td (dt-fst x))
                 | Some x ⇒ Some x) =
                Some (s, n)) (all-field-names-tuple x)
    apply (rule filter-same-eq)
    apply (cases x)
    subgoal for nm t' -
      using distinct
      apply (auto split: option.splits simp add: field-lookup-list-hd-notin)
      done
    done
  have eq2: filter (λf. ∃ n. field-lookup-list xs f 0 = Some (s, n))
     (all-field-names-list xs) =
    filter
     (λf. ∃ n. (case field-lookup-tuple x f 0 of
                 None ⇒ field-lookup-list xs f (0 + size-td (dt-fst x))
                 | Some x ⇒ Some x) =
                Some (s, n))
     (all-field-names-list xs)
```

apply (*rule filter-same-eq*)
apply (*cases x*)
subgoal for *nm t′* -
using *distinct Cons-typ-desc.hyps*(*2*)[*OF wf-xs*]
by (*smt* (*verit*) *all-field-names-exists-field-names-u*(*3*) *dt-prj-simps*(*2*)
field-lookup-list-None field-lookup-list-offsetD field-lookup-offset′(*3*)
fl5 hd-Cons-tl mem-Collect-eq option.case(*1*) *option.simps*(*3*) *set-filter*)
done

show *?case*
by (*simp add*: *Cons-typ-desc.hyps*(*1*)[*OF wf-x*] *Cons-typ-desc.hyps*(*2*)[*OF wf-xs*] *eq1 eq2*)
next
case (*DTuple-typ-desc t′ nm* -)
then show *?case*
by (*auto simp add*: *filter-map intro*: *filter-same-eq*)
qed

lemma *all-field-names-export-uinfo′*:
fixes *t*:: (*′a*, *′b*) *typ-info*
and *st*:: (*′a*, *′b*) *typ-info-struct*
and *ts*::(*′a*, *′b*) *typ-info-tuple list*
and *x*::(*′a*, *′b*) *typ-info-tuple*
shows
all-field-names (*map-td field-norm* (*λ*-. ()) *t*) = *all-field-names t*
all-field-names-struct (*map-td-struct field-norm* (*λ*-. ()) *st*) = *all-field-names-struct st*
all-field-names-list (*map-td-list field-norm* (*λ*-. ()) *ts*) = *all-field-names-list ts*
all-field-names-tuple (*map-td-tuple field-norm* (*λ*-. ()) *x*) = *all-field-names-tuple x*
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *all-field-names-export-uinfo*:
all-field-names (*export-uinfo t*) = *all-field-names t*
by (*simp add*: *export-uinfo-def all-field-names-export-uinfo′*)

lemma *inj* (#)
by (*metis inj-onI list.inject*)

lemma *distinct-map-cons*: *distinct xs* ⟹ *distinct* (*map* ((#) *ys*) *xs*)
by (*induct xs*) *auto*

lemma *all-field-names-list-conv*: *all-field-names-list xs* =
concat (*map* (*λx*. *map* ((#) (*dt-snd x*)) ((*all-field-names o dt-fst*) *x*)) *xs*)
apply (*induct xs*)
apply *simp*
subgoal for *x1 xs*
apply (*cases x1*)
apply *auto*

done
done

lemma *distinct-all-field-names*:
fixes t :: ('a, 'b) *typ-info* **and**
st :: ('a, 'b) *typ-info-struct* **and**
ts :: ('a, 'b) *typ-info-tuple list* **and**
x :: ('a, 'b) *typ-info-tuple*
shows
wf-desc $t \Longrightarrow$ *distinct* (*all-field-names* t) **and**
wf-desc-struct $st \Longrightarrow$ *distinct* (*all-field-names-struct* st) **and**
wf-desc-list $ts \Longrightarrow$ *distinct* (*all-field-names-list* ts) **and**
wf-desc-tuple $x \Longrightarrow$ *distinct* (*all-field-names-tuple* x)
proof (*induct* t **and** st **and** ts **and** x)
case (*TypDesc algn st nm*)
then show *?case*
by *auto* (*metis all-field-names-export-uinfo'*(*2*) *empty-all-field-names*(*2*))
next
case (*TypScalar n algn x*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x xs*)
then show *?case* **apply** (*cases x*) **by** (*auto simp add*: *all-field-names-list-conv*)
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case*
by (*auto simp add*: *distinct-map-cons*)
qed

lemma *td-set-field-names-u-nonempty*:
$(s, n) \in$ *td-set* $t\ m \Longrightarrow$ *field-names-u* $t\ s \neq []$
$(s, n) \in$ *td-set-struct* $st\ m \Longrightarrow$ *field-names-struct-u* $st\ s \neq []$
$(s, n) \in$ *td-set-list* $ts\ m \Longrightarrow$ *field-names-list-u* $ts\ s \neq []$
$(s, n) \in$ *td-set-tuple* $x\ m \Longrightarrow$ *field-names-tuple-u* $x\ s \neq []$
by (*induct* t **and** st **and** ts **and** x *arbitrary*: $n\ m$ **and** $n\ m$ **and** $n\ m$ **and** $n\ m$)
auto

lemma *field-lookup-export-uinfo-Some-rev*:
field-lookup (*export-uinfo t*) $f\ n =$ *Some* $(s, k) \Longrightarrow \exists\, s'.$ *field-lookup* $t\ f\ n =$ *Some* $(s', k) \wedge s =$ *export-uinfo* s'
by (*simp add*: *export-uinfo-def field-lookup-map map-td-flr-def split*: *option.splits*

prod.splits)

lemma *wf-desc-export-uinfo-pres*:
fixes *t* :: ('*a*, '*b*) *typ-info* **and**
st :: ('*a*, '*b*) *typ-info-struct* **and**
ts :: ('*a*, '*b*) *typ-info-tuple list* **and**
x :: ('*a*, '*b*) *typ-info-tuple*
shows
wf-desc t ⟹ *wf-desc* (*export-uinfo t*)
wf-desc-struct st ⟹ *wf-desc-struct* (*map-td-struct field-norm* (λ-. ()) *st*)
wf-desc-list ts ⟹ *wf-desc-list* (*map-td-list field-norm* (λ-. ()) *ts*)
wf-desc-tuple x ⟹ *wf-desc-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*)
apply (*induct t* **and** *st* **and** *ts* **and** *x*)
by (*auto simp add*: *export-uinfo-def*)
(*metis imageI mat4 wf-desc-list.wfd4 wf-desc-map*(*3*))

primrec
toplevel-field-names :: ('*a*, '*b*) *typ-desc* ⇒
(*field-name*) *list* **and**
toplevel-field-names-struct :: ('*a*, '*b*) *typ-struct* ⇒
(*field-name*) *list* **and**
toplevel-field-names-list :: ('*a*, '*b*) *typ-tuple list* ⇒
(*field-name*) *list* **and**
toplevel-field-names-tuple :: ('*a*, '*b*) *typ-tuple* ⇒
(*field-name*) *list*
where
toplevel-field-names (*TypDesc algn st nm*) = *toplevel-field-names-struct st*

| *toplevel-field-names-struct* (*TypScalar m algn d*) = []
| *toplevel-field-names-struct* (*TypAggregate xs*) = *toplevel-field-names-list xs*

| *toplevel-field-names-list* [] = []
| *toplevel-field-names-list* (*x*#*xs*) = *toplevel-field-names-tuple x* @ *toplevel-field-names-list xs*

| *toplevel-field-names-tuple* (*DTuple s f d*) = [*f*]

lemma *all-field-names-root*: ∃ *xs*. *all-field-names t* = [[]] @ *xs*
by (*cases t*) *simp*

lemma *toplevel-field-names-all-field-names*:
fixes *t*:: ('*a*, '*b*) *typ-desc*
and *st*:: ('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
f ∈ *set* (*toplevel-field-names t*) ⟹ [*f*] ∈ *set* (*all-field-names t*)

$f \in set\ (toplevel\text{-}field\text{-}names\text{-}struct\ st) \Longrightarrow [f] \in set\ (all\text{-}field\text{-}names\text{-}struct\ st)$
$f \in set\ (toplevel\text{-}field\text{-}names\text{-}list\ ts) \Longrightarrow [f] \in set\ (all\text{-}field\text{-}names\text{-}list\ ts)$
$f \in set\ (toplevel\text{-}field\text{-}names\text{-}tuple\ x) \Longrightarrow [f] \in set\ (all\text{-}field\text{-}names\text{-}tuple\ x)$
apply (*induct t* **and** *st* **and** *ts* **and** *x*)
by *auto*
(*metis all-field-names-root append-Cons imageI list.set-intros*(*1*))

lemma *append-eq-same-prefixI*: $ys = zs \Longrightarrow xs\ @\ ys = xs\ @\ zs$
by *simp*

lemma *toplevel-field-names-field-lookup*:
fixes *t*:: ('*a*, '*b*) *typ-info*
and *st*:: ('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
$f \in set\ (toplevel\text{-}field\text{-}names\ t) \Longrightarrow wf\text{-}desc\ t \Longrightarrow$
$\exists\, s\ n.\ field\text{-}lookup\ t\ [f]\ m = Some\ (s,\ m + n)$

$f \in set\ (toplevel\text{-}field\text{-}names\text{-}struct\ st) \Longrightarrow wf\text{-}desc\text{-}struct\ st \Longrightarrow$
$\exists\, s\ n.\ field\text{-}lookup\text{-}struct\ st\ [f]\ m = Some\ (s,\ m + n)$

$f \in set\ (toplevel\text{-}field\text{-}names\text{-}list\ ts) \Longrightarrow wf\text{-}desc\text{-}list\ ts \Longrightarrow$
$\exists\, s\ n.\ field\text{-}lookup\text{-}list\ ts\ [f]\ m = Some\ (s,\ m + n)$

$f \in set\ (toplevel\text{-}field\text{-}names\text{-}tuple\ x) \Longrightarrow wf\text{-}desc\text{-}tuple\ x \Longrightarrow$
$\exists\, s\ n.\ field\text{-}lookup\text{-}tuple\ x\ [f]\ m = Some\ (s,\ m + n)$
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m* **and** *f m* **and** *f m* **and** *f m*)
case (*TypDesc algn st nm*)
then show *?case* **by** *simp*
next
case (*TypScalar sz algn d*)
then show *?case* **by** *simp*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: $x = DTuple\ d\ nm\ y$ **by** (*cases x*) *auto*
then show *?case* **using** *Cons-typ-desc*
by *auto*
(*metis* (*no-types*, *opaque-lifting*) *add.assoc*)
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** *auto*
qed

lemma *partition-toplevel-field-names*:
fixes *t*:: ('*a*, '*b*) *typ-info*
and *st*:: ('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
length bs = *size-td t* ⟹ *aggregate t* ⟹ *wf-desc t* ⟹
concat (*map* (λ*f*. *take* (*size-td* (*fst* (*the* (*field-lookup t* [*f*] *n*))))
(*drop* (*snd* (*the* (*field-lookup t* [*f*] *n*)) − *n*) *bs*))
(*toplevel-field-names t*)) = *bs*

length bs = *size-td-struct st* ⟹ *aggregate-struct st* ⟹ *wf-desc-struct st* ⟹
concat (*map* (λ*f*. *take* (*size-td* (*fst* (*the* (*field-lookup-struct st* [*f*] *n*))))
(*drop* (*snd* (*the* (*field-lookup-struct st* [*f*] *n*)) − *n*) *bs*))
(*toplevel-field-names-struct st*)) = *bs*

length bs = *size-td-list ts* ⟹ *wf-desc-list ts* ⟹
concat (*map* (λ*f*. *take* (*size-td* (*fst* (*the* (*field-lookup-list ts* [*f*] *n*))))
(*drop* (*snd* (*the* (*field-lookup-list ts* [*f*] *n*)) − *n*) *bs*))
(*toplevel-field-names-list ts*)) = *bs*

length bs = *size-td-tuple x* ⟹ *wf-desc-tuple x* ⟹
concat (*map* (λ*f*. *take* (*size-td* (*fst* (*the* (*field-lookup-tuple x* [*f*] *n*))))
(*drop* (*snd* (*the* (*field-lookup-tuple x* [*f*] *n*)) − *n*) *bs*))
(*toplevel-field-names-tuple x*)) = *bs*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *bs n* **and** *bs n* **and** *bs n* **and** *bs n*)
case (*TypDesc algn st nm*)
then show *?case* **by** *simp*
next
case (*TypScalar sz algn d*)
then show *?case* **by** *simp*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: *x* = *DTuple d nm y* **by** (*cases x*) *auto*
from *Cons-typ-desc.prems* **obtain**
lbs: *length bs* = *size-td-tuple x* + *size-td-list fs* **and**
wf-desc-x: *wf-desc-tuple* (*DTuple d nm y*) **and**
nm-notin: *nm* ∉ *dt-snd* ' *set fs* **and**
wf-desc-fs: *wf-desc-list fs*

by (*auto simp add*: *x*)

```
  from nm-notin have nm-notin': nm ∉ set (toplevel-field-names-list fs)
  proof (induct fs)
    case Nil
    then show ?case by simp
  next
    case (Cons f1 fs)
    then show ?case by (cases f1) auto
  qed

  from lbs have l-take: length (take (size-td d) bs) = size-td d
    by (simp add: x)

  from lbs have l-drop: length (drop (size-td d) bs) = size-td-list fs
    by (simp add: x)


  from lbs have bs-take-drop: bs = take (size-td d) bs @ drop (size-td d) bs
    by (simp add: x)
  note hyp = Cons-typ-desc.hyps(2) [OF l-drop wf-desc-fs, where n= (n + size-td
d)]

  show ?case
    apply (simp add: x)
    apply (simp cong: if-cong option.case-cong)
    apply (subst (3) bs-take-drop)
    apply (rule append-eq-same-prefixI)
    apply (subst hyp [symmetric])
    apply (rule arg-cong [where f = concat])
    apply (rule map-cong [OF refl])
    using nm-notin'
    apply auto
    thm toplevel-field-names-field-lookup(3)[OF - wf-desc-fs]
    apply (frule toplevel-field-names-field-lookup(3) [OF - wf-desc-fs, where m =
(n + size-td d) ])
    by auto
     (metis add.commute)
next
  case (DTuple-typ-desc d nm y)
  then show ?case by auto
qed


lemma toplevel-field-names-export-uinfo':
  fixes t:: ('a, 'b) typ-info
  and st:: ('a, 'b) typ-info-struct
  and ts::('a, 'b) typ-info-tuple list
  and x::('a, 'b) typ-info-tuple
shows
```

toplevel-field-names (*map-td field-norm* (λ-. ()) *t*) = *toplevel-field-names t*
toplevel-field-names-struct (*map-td-struct field-norm* (λ-. ()) *st*) = *toplevel-field-names-struct st*
toplevel-field-names-list (*map-td-list field-norm* (λ-. ()) *ts*) = *toplevel-field-names-list ts*
toplevel-field-names-tuple (*map-td-tuple field-norm* (λ-. ()) *x*) = *toplevel-field-names-tuple x*
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *toplevel-field-names-export-uinfo*:
toplevel-field-names (*export-uinfo t*) = *toplevel-field-names t*
using *toplevel-field-names-export-uinfo'* **by** (*simp add*: *export-uinfo-def*)

lemma (**in** *xmem-type*) *xmem-type-partition-toplevel-field-names*:
assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
assumes *lbs*: *length bs* = *size-of* (*TYPE*('*a*))
shows *concat* (*map* (λ*f*. *take* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*))))
(*drop* (*snd* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*))) *bs*))
(*toplevel-field-names* (*typ-info-t TYPE*('*a*)))) = *bs*
using *partition-toplevel-field-names*(*1*) [*OF lbs* [*simplified size-of-def*] *aggregate wf-desc*, **where** *n*=*0*]
by *simp*

lemma *toplevel-field-names-sum-list-size*:
fixes *t*:: ('*a*, '*b*) *typ-info*
and *st*:: ('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
aggregate t ⟹ *wf-desc t* ⟹
sum-list (*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup t* [*f*] *n*)))) (*toplevel-field-names t*)) =
size-td t

aggregate-struct st ⟹ *wf-desc-struct st* ⟹
sum-list (*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup-struct st* [*f*] *n*)))) (*toplevel-field-names-struct st*)) =
size-td-struct st

wf-desc-list ts ⟹
sum-list (*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup-list ts* [*f*] *n*)))) (*toplevel-field-names-list ts*)) =
size-td-list ts

wf-desc t ⟹
sum-list (*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup-tuple x* [*f*] *n*)))) (*toplevel-field-names-tuple x*)) =
size-td-tuple x

proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *n* **and** *n* **and** *n* **and** *n*)
case (*TypDesc algn st nm*)
then show *?case* **by** *simp*
next
case (*TypScalar sz algn d*)
then show *?case* **by** *simp*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: *x* = *DTuple d nm y* **by** (*cases x*) *auto*

from *Cons-typ-desc.prems* **obtain**
wf-desc-x: *wf-desc-tuple* (*DTuple d nm y*) **and**
nm-notin: *nm* ∉ *dt-snd* ‘ *set fs* **and**
wf-desc-fs: *wf-desc-list fs*
by (*auto simp add*: *x*)

from *nm-notin* **have** *nm-notin'*: *nm* ∉ *set* (*toplevel-field-names-list fs*)
proof (*induct fs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons f1 fs*)
then show *?case* **by** (*cases f1*) *auto*
qed

note *hyp* = *Cons-typ-desc.hyps*(*2*) [*OF wf-desc-fs*, **where** *n*= (*n* + *size-td d*)]

show *?case*
apply (*simp add*: *x*)
apply (*simp cong*: *if-cong option.case-cong*)
using *nm-notin' hyp*
by (*smt* (*verit*, *best*) *map-eq-conv option.simps*(*4*))
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** *auto*
qed

lemma *toplevel-field-names-sum-list-offset*:
fixes *t*:: (*'a*, *'b*) *typ-info*
and *st*:: (*'a*, *'b*) *typ-info-struct*
and *ts*::(*'a*, *'b*) *typ-info-tuple list*

and x::($'a$, $'b$) *typ-info-tuple*
shows
aggregate t $\Longrightarrow$ *wf-desc t* $\Longrightarrow$ $i <$ *length* (*toplevel-field-names t*) $\Longrightarrow$
sum-list (*map* (λi. *size-td* (*fst* (*the* (*field-lookup t* [(*toplevel-field-names t*) ! *i*] *n*)))) [*0*..<*i*]) =
(*snd* (*the* (*field-lookup t* [(*toplevel-field-names t*) ! *i*] *n*)) − *n*)

aggregate-struct st $\Longrightarrow$ *wf-desc-struct st* $\Longrightarrow$ $i <$ *length* (*toplevel-field-names-struct st*) $\Longrightarrow$
sum-list (*map* (λi. *size-td* (*fst* (*the* (*field-lookup-struct st* [(*toplevel-field-names-struct st*) ! *i*] *n*)))) [*0*..<*i*]) =
(*snd* (*the* (*field-lookup-struct st* [(*toplevel-field-names-struct st*) ! *i*] *n*)) − *n*)

wf-desc-list ts $\Longrightarrow$ $i <$ *length* (*toplevel-field-names-list ts*) $\Longrightarrow$
sum-list (*map* (λi. *size-td* (*fst* (*the* (*field-lookup-list ts* [(*toplevel-field-names-list ts*) ! *i*] *n*)))) [*0*..<*i*]) =
(*snd* (*the* (*field-lookup-list ts* [(*toplevel-field-names-list ts*) ! *i*] *n*)) − *n*)

wf-desc-tuple x $\Longrightarrow$ $i <$ *length* (*toplevel-field-names-tuple x*) $\Longrightarrow$
sum-list (*map* (λi. *size-td* (*fst* (*the* (*field-lookup-tuple x* [(*toplevel-field-names-tuple x*) ! *i*] *n*)))) [*0*..<*i*]) =
(*snd* (*the* (*field-lookup-tuple x* [(*toplevel-field-names-tuple x*) ! *i*] *n*)) − *n*)

proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *n i* **and** *n i* **and** *n i* **and** *n i*)
case (*TypDesc algn st nm*)
then show *?case* **by** *simp*
next
case (*TypScalar sz algn d*)
then show *?case* **by** *simp*
next
case (*TypAggregate ts*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc x fs*)
obtain *d nm y* **where** *x*: $x = DTuple\ d\ nm\ y$ **by** (*cases x*) *auto*

from *Cons-typ-desc.prems* **obtain**
wf-desc-x: *wf-desc-tuple* (*DTuple d nm y*) **and**
nm-notin: $nm \notin$ *dt-snd* ' *set fs* **and**
wf-desc-fs: *wf-desc-list fs* **and**
i-bound: $i <$ *Suc* (*length* (*toplevel-field-names-list fs*))
by (*auto simp add*: *x*)

from *nm-notin* **have** *nm-notin'*: $nm \notin$ *set* (*toplevel-field-names-list fs*)
proof (*induct fs*)

```
  case Nil
  then show ?case by simp
next
  case (Cons f1 fs)
  then show ?case by (cases f1) auto
qed

note hyp = Cons-typ-desc.hyps(2) [OF wf-desc-fs, where n= (n + size-td d)]

from nm-notin' i-bound have nm-eq-conv: nm = (nm # toplevel-field-names-list
fs) ! i ⟷ i = 0
  apply (cases i)
   apply simp
  apply (simp)
  using nth-mem by blast


show ?case
proof (cases i)
  case 0
  then show ?thesis by (simp add: x)
next
  case (Suc i')

  with i-bound have i'-bound: i' < length (toplevel-field-names-list fs) by simp
  have split: [0..<Suc i'] = 0 # [1..<Suc i']
    by (simp add: upt-rec)

  have sum-eq:
    (∑ i←[0..<i']. size-td
                (fst (the (case if nm = toplevel-field-names-list fs ! i
                                then Some (d, n) else None of
                           None ⇒
                             field-lookup-list fs
                              [toplevel-field-names-list fs ! i] (n + size-td d)
                           | Some x2 ⇒ Some x2)))) =
     (∑ i←[0..<i']. size-td
                (fst (the (field-lookup-list fs [toplevel-field-names-list fs ! i]
                           (n + size-td d)))))
    apply (rule arg-cong [where f=sum-list])
    apply (rule map-cong [OF refl])
    using nm-notin' i'-bound
    apply auto
    done

  show ?thesis
    apply (simp add: x)
    apply (clarsimp cong: if-cong option.case-cong simp add: nm-eq-conv)
    apply (subst Suc)
```

apply (*subst split*)
apply (*simp only*: *list.map sum-list.Cons*)
apply (*subst sum-list-Suc-index-shift*)
apply (*simp add*: *Suc*)
apply (*simp cong*: *if-cong option.case-cong*)
apply (*subst sum-eq*)
apply (*subst hyp* [*OF i′-bound*])
using *toplevel-field-names-field-lookup*(*3*)
[**where** *f*=*toplevel-field-names-list fs* ! *i′* **and** *ts* = *fs* **and** *m*=(*n* + *size-td d*), *OF - wf-desc-fs*] *i′-bound*
apply *clarsimp*
done
qed
next
case (*DTuple-typ-desc d nm y*)
then show *?case* **by** *auto*
qed

lemma *sum-list-upt-map-nth-conv*: *sum-list* (*map* (λ*i*. *g* (*xs* ! *i*)) [*0*..<*length xs*]) = *sum-list* (*map g xs*)
proof −
from *map-nth* **have** *eq1*: *map* ((!) *xs*) [*0*..<*length xs*] = *xs* **.**
from *map-map* [**where** *f*=*g* **and** *g*=((!) *xs*)]
have *eq2*: *map g* (*map* ((!) *xs*) [*0*..<*length xs*]) = *map* (*g* ∘ (!) *xs*) [*0*..<*length xs*] **.**
from *eq1 eq2*
show *?thesis*
by (*simp add*: *comp-def*)
qed

lemma *toplevel-field-names-empty-typ-info*: *toplevel-field-names* (*empty-typ-info algn tn*) = []
by (*simp add*: *empty-typ-info-def*)

lemma *toplevel-field-names-no-padding-empty-typ-info*:
filter (*Not o padding-field-name*) (*toplevel-field-names* (*empty-typ-info algn tn*)) = []
by (*simp add*: *empty-typ-info-def*)

lemma *toplevel-field-names-list-append* [*simp*]:
toplevel-field-names-list (*xs* @ *ys*) = *toplevel-field-names-list xs* @ *toplevel-field-names-list ys*
by (*induct xs*) *auto*

lemma *toplevel-field-names-extend-ti*:
fixes
t :: *′a xtyp-info* **and**
st :: *′a xtyp-info-struct* **and**

ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
shows
toplevel-field-names (*extend-ti t s n fn d*) = *toplevel-field-names t* @ [*fn*]
toplevel-field-names-struct (*extend-ti-struct st s fn d*) = *toplevel-field-names-struct st* @ [*fn*]
toplevel-field-names-list ts = *toplevel-field-names-list ts*
toplevel-field-names-tuple x = *toplevel-field-names-tuple x*
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *toplevel-field-names-adjust-ti'*:
fixes
t :: *'a xtyp-info* **and**
st :: *'a xtyp-info-struct* **and**
ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
shows
toplevel-field-names (*map-td* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *t*) =
toplevel-field-names t
toplevel-field-names-struct (*map-td-struct* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *st*) =
toplevel-field-names-struct st
toplevel-field-names-list (*map-td-list* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *ts*) =
toplevel-field-names-list ts
toplevel-field-names-tuple (*map-td-tuple* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *x*) =
toplevel-field-names-tuple x
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*auto*)

lemma *toplevel-field-names-adjust-ti*:
toplevel-field-names (*adjust-ti t acc upd*) = *toplevel-field-names t*
by (*simp add*: *toplevel-field-names-adjust-ti' adjust-ti-def*)

lemma *padding-field-name-pad*: *padding-field-name* (*foldl* (@) *''!pad-''* *xs*)
by (*auto simp add*: *padding-field-name-def foldl-conv-concat*)

lemma *toplevel-field-names-no-padding-ti-pad-combine*:
filter (*Not o padding-field-name*) (*toplevel-field-names* (*ti-pad-combine n t*)) =
filter (*Not o padding-field-name*) (*toplevel-field-names t*)
by (*simp add*: *ti-pad-combine-def Let-def toplevel-field-names-extend-ti*(*1*) *padding-field-name-pad*)

lemma *toplevel-field-names-ti-pad-combine*:
(*toplevel-field-names* (*ti-pad-combine n t*)) =
toplevel-field-names t @ [*foldl* (@) *''!pad-''* (*CompoundCTypes.field-names-list t*)]
by (*simp add*: *ti-pad-combine-def Let-def toplevel-field-names-extend-ti*(*1*) *padding-field-name-pad*)

lemma *toplevel-field-names-ti-typ-combine*:
toplevel-field-names (*ti-typ-combine t-b acc upd algn fn t*) = *toplevel-field-names t* @ [*fn*]
by (*simp add*: *ti-typ-combine-def toplevel-field-names-extend-ti*)

lemma *toplevel-field-names-no-padding-ti-typ-combine*:
¬ *padding-field-name fn* ⟹
filter (*Not o padding-field-name*) (*toplevel-field-names* (*ti-typ-combine t-b acc upd algn fn t*)) =
filter (*Not o padding-field-name*) (*toplevel-field-names t*) @ [*fn*]
by (*simp add*: *toplevel-field-names-ti-typ-combine toplevel-field-names-extend-ti*)

lemma *toplevel-field-names-no-padding-ti-typ-pad-combine*:
¬ *padding-field-name fn* ⟹
filter (*Not o padding-field-name*) (*toplevel-field-names* (*ti-typ-pad-combine t-b acc upd algn fn t*)) =
filter (*Not o padding-field-name*) (*toplevel-field-names t*) @ [*fn*]
by (*simp add*: *ti-typ-pad-combine-def toplevel-field-names-ti-typ-combine toplevel-field-names-no-padding-ti-pa Let-def*)

lemma *toplevel-field-names-ti-typ-pad-combine*:
toplevel-field-names (*ti-typ-pad-combine* (*t-b*:: *'b itself*) *acc upd algn fn t*) =
toplevel-field-names t @ (
if 0 < padup (*max* (*2 ^ algn*) (*align-of TYPE*('*b*::*c-type*))) (*size-td t*) *then*
[*foldl* (@) *''!pad-''* (*CompoundCTypes.field-names-list t*), *fn*]
else
[*fn*])
by (*simp add*: *ti-typ-pad-combine-def toplevel-field-names-ti-typ-combine toplevel-field-names-no-padding-ti-pa Let-def*
toplevel-field-names-ti-pad-combine)

lemma *toplevel-field-names-map-align*: *toplevel-field-names* (*map-align n t*) = *toplevel-field-names t*
by (*cases t*) (*simp add*: *map-align-def*)

lemma *toplevel-field-names-no-padding-final-pad*:
filter (*Not o padding-field-name*) (*toplevel-field-names* (*final-pad n t*))
= *filter* (*Not o padding-field-name*) (*toplevel-field-names t*)
by (*simp add*: *final-pad-def Let-def toplevel-field-names-map-align toplevel-field-names-no-padding-ti-pad-comb*

lemma *toplevel-field-names-final-pad*:
(*toplevel-field-names* (*final-pad n t*))
=
toplevel-field-names t @ (
if 0 < padup (*2 ^ max n* (*align-td t*)) (*size-td t*) *then*
[*foldl* (@) *''!pad-''* (*CompoundCTypes.field-names-list t*)]

else [])
by (*simp add*: *final-pad-def Let-def toplevel-field-names-map-align toplevel-field-names-ti-pad-combine*)

lemmas *toplevel-field-names-no-padding-combinator-simps* =
toplevel-field-names-no-padding-empty-typ-info
toplevel-field-names-no-padding-final-pad
toplevel-field-names-no-padding-ti-typ-pad-combine
toplevel-field-names-no-padding-ti-typ-combine

lemmas *toplevel-field-names-combinator-simps* =
toplevel-field-names-empty-typ-info
toplevel-field-names-final-pad
toplevel-field-names-ti-typ-pad-combine
toplevel-field-names-ti-typ-combine

lemma *fold-filter-out-id*:
assumes *filter-out-id*: $\bigwedge i\ v.\ i < length\ xs \Longrightarrow \neg\ P\ (xs\ !\ i) \Longrightarrow f\ (xs!i)\ v = v$
shows*fold f xs* = *fold f* (*filter P xs*)
using *filter-out-id*
apply (*induct xs*)
apply *simp*
apply (*clarsimp simp add*: *comp-def fun-eq-iff*, *safe*)
apply (*metis Suc-mono nth-Cons-Suc*)
by (*metis not-less-eq nth-Cons-0 nth-Cons-Suc zero-less-Suc*)

context *xmem-type*
begin

lemma *xmem-type-toplevel-field-names-sum-list-size*:
assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
shows *sum-list*
(*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *n*))))
(*toplevel-field-names* (*typ-info-t TYPE*('*a*)))) =
size-of TYPE('*a*)
using *toplevel-field-names-sum-list-size*(*1*) [*OF aggregate wf-desc*] **by** (*simp add*:
size-of-def)

lemma *xmem-type-toplevel-field-names-sum-list-offset*:
assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
assumes *i-bound*: *i* < *length* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
shows
sum-list (*map* (λ*i*. *size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*))
[(*toplevel-field-names* (*typ-info-t TYPE*('*a*))) ! *i*] *0*))))
[*0*..<*i*]) =
(*snd* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [(*toplevel-field-names* (*typ-info-t*
TYPE('*a*))) ! *i*] *0*)))
using *toplevel-field-names-sum-list-offset*(*1*) [*OF aggregate wf-desc i-bound*, **where**
n=0]
by *simp*

lemma *xmem-type-toplevel-field-names-field-lookup*:
assumes *f*: *f* ∈ *set* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
shows ∃ *s n*. *field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0* = *Some* (*s*, *n*)
using *toplevel-field-names-field-lookup*(*1*) [*OF f wf-desc*, *of 0*] **by** *simp*

lemma (**in** *c-type*) *field-lookup-typ-uinfo-t-Some*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f m* = *Some* (*export-uinfo s*, *n*)
by (*simp add*: *typ-uinfo-t-def field-lookup-export-uinfo-Some*)

lemma *toplevel-field-names-field-lookup-offset-conv*:
assumes *f*: *f* ∈ *set* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
shows *snd* (*the* (*field-lookup* (*typ-uinfo-t TYPE*('*a*)) [*f*] *0*)) =
snd (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*))
using *xmem-type-toplevel-field-names-field-lookup* [*OF f*]
by (*auto simp add*: *field-lookup-typ-uinfo-t-Some*)

lemma *heap-update-list-fold-toplevel-field-names*:
fixes *p*::'*a ptr*
assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
assumes *cgrd*: *c-guard p*
shows
heap-update-list (*ptr-val p*) (*to-bytes x* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)))
h =
fold
(λ*f h*. *heap-update-list* (&(*p*→[*f*]))
(*access-ti*
(*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*)))
x
(*heap-list h* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*]
0)))) (&(*p*→[*f*]))))
h)
(*toplevel-field-names* (*typ-info-t TYPE*('*a*))) *h* (**is** *?LHS* = *fold ?F ?fs h*)
proof −

define *fld*::*nat* ⇒ *field-name* **where**
fld = (λ*i*. *toplevel-field-names* (*typ-info-t TYPE*('*a*)) ! *i*)
define *sz*:: *nat* ⇒ *nat* **where**
sz = (λ*i*. *size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*fld i*] *0*))))
define *off*:: *nat* ⇒ *nat* **where**
off = (λ*i*. *snd* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*fld i*] *0*)))
define *v*:: *nat* ⇒ *byte list* ⇒ *byte list* **where**
v = (λ*i*. (*access-ti*
(*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*fld i*] *0*)))
x))

have *lbs*: *length* (*to-bytes x* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))) = *size-of TYPE*('*a*)
by (*simp add*: *local.lense.access-result-size to-bytes-def*)
{
fix *i*::*nat*
assume *bound*: *i* < *length* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
hence *toplevel-field-names* (*typ-info-t TYPE*('*a*)) ! *i* ∈ *set* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
by *simp*
from *toplevel-field-names-field-lookup-offset-conv* [*OF this*]

have *snd* (*the* (*field-lookup* (*typ-uinfo-t TYPE*('*a*))
[*toplevel-field-names* (*typ-info-t TYPE*('*a*)) ! *i*] *0*)) =
snd (*the* (*field-lookup* (*typ-info-t TYPE*('*a*))
[*toplevel-field-names* (*typ-info-t TYPE*('*a*)) ! *i*] *0*)) .
} note *field-lookup-conv* = *this*

have *fs*: *?fs* = *map fld* [*0*..<*length*(*toplevel-field-names* (*typ-info-t TYPE*('*a*)))]
by (*simp add*: *fld-def map-nth*)

from *c-guard-no-wrap'* [*OF cgrd*]
have *no-overflow*: *unat* (*ptr-val p*) + *size-of TYPE*('*a*) ≤ *addr-card* .
have *fold-conv*:
fold ?F ?fs h =
fold
(λ*i h*. *heap-update-list* (*ptr-val p* + *word-of-nat* (*off i*))
(*v i* (*heap-list h* (*sz i*) (*ptr-val p* + *word-of-nat* (*off i*)))) *h*)
[*0*..<*length* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))] *h*
apply (*subst fs*)
apply (*subst fold-map*)
apply (*rule fold-cong* [*OF refl refl*])
apply (*simp add*: *fld-def off-def sz-def v-def field-lvalue-def field-offset-def field-offset-untyped-def*)
apply (*simp add*: *field-lookup-conv*)
done
note *partition*= *heap-update-list-fold-partition* [**where** *a*= *ptr-val p* **and** *sz*=*sz* **and** *off*=*off* **and** *v*=*v* **and** *h*=*h*
and *bs* = *to-bytes x* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)) **and**
m = *length* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
]

{
fix *i*
assume *i-bound*: *i* < *length* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
from *i-bound*
have *f*: *toplevel-field-names* (*typ-info-t TYPE*('*a*)) ! *i* ∈ *set* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
by *simp*
from *toplevel-field-names-field-lookup*(*1*) [*OF f*, **where** *m*= *0*, *OF wf-desc*]

```
    obtain s n where
      fl: field-lookup (typ-info-t TYPE('a))
            [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0 =
          Some (s, n) by auto

    have lh: length (heap-list h (size-of TYPE('a)) (ptr-val p)) = size-of TYPE('a)
by simp
    have v i (take (sz i)
          (drop (off i)
            (heap-list h
              (length (to-bytes x (heap-list h (size-of TYPE('a)) (ptr-val p))))
              (ptr-val p)))) =
         take (sz i) (drop (off i) (to-bytes x (heap-list h (size-of TYPE('a)) (ptr-val
p))))
      apply (simp add: off-def sz-def v-def fld-def lbs)
      apply (simp add: fl)
      using mem-type-field-lookup-access-ti-take-drop [OF fl lh, where v=x ]
      apply (simp add: to-bytes-def)
      done

  } note v-focus = this
  show ?thesis
    apply (subst fold-conv)
    apply (rule partition)
    subgoal
      using no-overflow
      by (simp add: local.lense.access-result-size to-bytes-def)
    subgoal
     apply (subst xmem-type-partition-toplevel-field-names [OF aggregate lbs, sym-
metric])
      apply (subst fs)
      apply (subst map-map)
      apply (rule arg-cong [where f=concat])
      apply (rule map-cong [OF refl])
      apply (simp add: fld-def off-def sz-def)
      done
    subgoal
      apply (subst lbs)
     apply (subst xmem-type-toplevel-field-names-sum-list-size [OF aggregate, sym-
metric])
      apply (simp add: sz-def)
      apply (subst fs)
      apply (simp add: fld-def comp-def)
      done
    subgoal for i
      apply (simp add: off-def sz-def)
      apply (simp add: fld-def)
      using xmem-type-toplevel-field-names-sum-list-offset [OF aggregate, where i
= i]
```

```
    apply simp
    done
  subgoal for i
    by (rule v-focus)
  done
qed

lemma heap-update-list-padding-fold-toplevel-field-names:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  assumes lbs: length bs = size-of TYPE('a)
  shows
heap-update-list (ptr-val p) (to-bytes x bs) h =
  fold
   (λf h. heap-update-list (&(p→[f]))
          (access-ti
            (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
            x
            (take (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0))))
              (drop ((snd (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))) bs)))
           h)
   (toplevel-field-names (typ-info-t TYPE('a))) h (is ?LHS = fold ?F ?fs h)
proof −

  define fld::nat ⇒ field-name where
    fld = (λi. toplevel-field-names (typ-info-t TYPE('a)) ! i)
  define sz:: nat ⇒ nat where
    sz = (λi. size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0))))
  define off:: nat ⇒ nat where
    off = (λi. snd (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0)))
  define v:: nat ⇒ byte list ⇒ byte list where
    v = (λi. (access-ti
               (fst (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0)))
               x))

  have lbs': length (to-bytes x bs) = size-of TYPE('a)
    by (simp add: local.lense.access-result-size to-bytes-def)
  {
    fix i::nat
    assume bound:  i < length (toplevel-field-names (typ-info-t TYPE('a)))
   hence toplevel-field-names (typ-info-t TYPE('a)) ! i ∈ set (toplevel-field-names
(typ-info-t TYPE('a)))
      by simp
    from toplevel-field-names-field-lookup-offset-conv [OF this]

    have snd (the (field-lookup (typ-uinfo-t TYPE('a))
                [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0)) =
         snd (the (field-lookup (typ-info-t TYPE('a))
```

```
        [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0)) .
  } note field-lookup-conv = this

  have fs: ?fs = map fld [0..<length(toplevel-field-names (typ-info-t TYPE('a)))]
    by (simp add: fld-def map-nth)

  from c-guard-no-wrap' [OF cgrd]
  have no-overflow: unat (ptr-val p) + size-of TYPE('a) ≤ addr-card .
  have fold-conv:
    fold ?F ?fs h =
        fold
          (λi h. heap-update-list (ptr-val p + word-of-nat (off i))
             (v i (take (sz i) (drop (off i) bs))) h)
          [0..<length (toplevel-field-names (typ-info-t TYPE('a)))] h
    apply (subst fs)
    apply (subst fold-map)
    apply (rule fold-cong [OF refl refl])
     apply (simp add: fld-def off-def sz-def v-def field-lvalue-def field-offset-def
field-offset-untyped-def)
    apply (simp add: field-lookup-conv)
    done

  note partition= heap-update-list-padding-fold-partition [where a= ptr-val p and
sz=sz and off=off and v=v and h=h
       and bs = to-bytes x bs and pbs = bs and
        m = length (toplevel-field-names (typ-info-t TYPE('a)))
      ]

  {
    fix i
    assume i-bound: i < length (toplevel-field-names (typ-info-t TYPE('a)))
    from i-bound
   have f: toplevel-field-names (typ-info-t TYPE('a)) ! i ∈ set (toplevel-field-names
(typ-info-t TYPE('a)))
      by simp
    from toplevel-field-names-field-lookup(1)  [OF f, where m= 0, OF wf-desc]
    obtain s n where
      fl: field-lookup (typ-info-t TYPE('a))
            [toplevel-field-names (typ-info-t TYPE('a)) ! i] 0 =
          Some (s, n) by auto

   have lh: length (heap-list h (size-of TYPE('a)) (ptr-val p)) = size-of TYPE('a)
by simp
    have v i (take (sz i) (drop (off i) bs)) =
          take (sz i) (drop (off i) (to-bytes x bs))
      apply (simp add: off-def sz-def v-def fld-def lbs)
      apply (simp add: fl)
      using mem-type-field-lookup-access-ti-take-drop [OF fl lbs, where v=x ]
      apply (simp add: to-bytes-def)
```

done

} note *v-focus = this*
show *?thesis*
apply (*subst fold-conv*)
apply (*rule partition*)
subgoal
using *no-overflow*
by (*simp add*: *local.lense.access-result-size to-bytes-def*)
subgoal
apply (*subst xmem-type-partition-toplevel-field-names* [*OF aggregate lbs′*, *symmetric*])
apply (*subst fs*)
apply (*subst map-map*)
apply (*rule arg-cong* [**where** *f=concat*])
apply (*rule map-cong* [*OF refl*])
apply (*simp add*: *fld-def off-def sz-def*)
done
subgoal
apply (*subst lbs′*)
apply (*subst xmem-type-toplevel-field-names-sum-list-size* [*OF aggregate*, *symmetric*])
apply (*simp add*: *sz-def*)
apply (*subst fs*)
apply (*simp add*: *fld-def comp-def*)
done
subgoal
apply (*subst xmem-type-partition-toplevel-field-names* [*OF aggregate lbs*, *symmetric*])
apply (*subst fs*)
apply (*subst map-map*)
apply (*rule arg-cong* [**where** *f=concat*])
apply (*rule map-cong* [*OF refl*])
apply (*simp add*: *fld-def off-def sz-def*)
done
subgoal
apply (*subst lbs*)
apply (*subst xmem-type-toplevel-field-names-sum-list-size* [*OF aggregate*, *symmetric*])
apply (*simp add*: *sz-def*)
apply (*subst fs*)
apply (*simp add*: *fld-def comp-def*)
done
subgoal for *i*
apply (*simp add*: *off-def sz-def*)
apply (*simp add*: *fld-def*)
using *xmem-type-toplevel-field-names-sum-list-offset* [*OF aggregate*, **where** *i = i*]
apply *simp*

```
    done
  subgoal for i
    by (rule v-focus)
  done
qed

lemma heap-update-fold-toplevel-field-names:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  shows
heap-update p x h =
  fold
   (λf h. heap-update-list (&(p→[f]))
           (access-ti
             (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
             x
               (heap-list h (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f]
0)))) (&(p→[f]))))
            h)
   (toplevel-field-names (typ-info-t TYPE('a))) h (is ?LHS = fold ?F ?fs h)
  using heap-update-list-fold-toplevel-field-names [OF aggregate cgrd]
  by (simp only: heap-update-def)

lemma heap-update-padding-fold-toplevel-field-names:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  assumes lbs: length bs = size-of TYPE('a)
  shows
heap-update-padding p x bs h =
  fold
   (λf h. heap-update-list (&(p→[f]))
           (access-ti
             (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
             x
             (take (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0))))
               (drop ((snd (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))) bs)))
            h)
   (toplevel-field-names (typ-info-t TYPE('a))) h (is ?LHS = fold ?F ?fs h)
  using heap-update-list-padding-fold-toplevel-field-names [OF aggregate cgrd lbs]
  by (simp only: heap-update-padding-def)

lemma heap-update-fold-toplevel-field-names-no-padding:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  shows
heap-update p x h =
```

```
  fold
   (λf h. heap-update-list (&(p→[f]))
           (access-ti
             (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
             x
               (heap-list h (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f]
0)))) (&(p→[f]))))
             h)
   (filter (Not o padding-field-name) (toplevel-field-names (typ-info-t TYPE('a))))
h (is ?LHS = fold ?F ?filter-fs h)
proof −

  { fix i
    fix h
    assume i-bound: i < length (toplevel-field-names (typ-info-t TYPE('a)))
    assume ¬ (Not ∘ padding-field-name)
             (toplevel-field-names (typ-info-t TYPE('a)) ! i)
   hence padding: padding-field-name (toplevel-field-names (typ-info-t TYPE('a))
! i)
      by simp
    from i-bound obtain m s where
       fl: field-lookup (typ-info-t TYPE('a))
             [(toplevel-field-names (typ-info-t TYPE('a)) ! i)] 0 = Some (s, m)
      by (meson nth-mem xmem-type-toplevel-field-names-field-lookup)


    from field-lookup-padding-field-name(1) [OF fl padding wf-padding]
    have is-padding-tag: is-padding-tag s .

    have ?F (toplevel-field-names (typ-info-t TYPE('a)) ! i) h = h
      apply (simp add: fl)
      using is-padding-tag
        by (clarsimp simp add: is-padding-tag-def padding-tag-def padding-desc-def
heap-update-list-id')
  } note pad = this

  have fold ?F ?filter-fs = fold ?F (toplevel-field-names (typ-info-t TYPE('a)))
    apply (rule fold-filter-out-id [symmetric])
    by (rule pad)
  with heap-update-fold-toplevel-field-names [OF aggregate cgrd, where x=x and
h=h]
  show ?thesis
    by simp
qed

lemma heap-list-concat-toplevel-field-names:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
```

```
  shows
   heap-list h (size-of TYPE('a)) (ptr-val p) =
    concat (map (λf. heap-list h (size-td (fst (the (field-lookup (typ-info-t TYPE('a))
[f] 0)))) (&(p→[f])))
        (toplevel-field-names (typ-info-t TYPE('a)))) (is ?LHS = concat (map ?F
?fs))
proof −
  define fld::nat ⇒ field-name where
    fld = (λi. toplevel-field-names (typ-info-t TYPE('a)) ! i)
  define sz:: nat ⇒ nat where
    sz = (λi. size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0))))
  define off:: nat ⇒ nat where
    off = (λi. snd (the (field-lookup (typ-info-t TYPE('a)) [fld i] 0)))

  have fs: ?fs = map fld [0..<length(toplevel-field-names (typ-info-t TYPE('a)))]
    by (simp add: fld-def map-nth)
  from c-guard-no-wrap' [OF cgrd]
  have no-overflow: unat (ptr-val p) + size-of TYPE('a) ≤ addr-card .

  have concat-conv:
   concat (map ?F ?fs) =
    concat (map (λi. (heap-list h (sz i) (ptr-val p + word-of-nat (off i))))
      [0..<length (toplevel-field-names (typ-info-t TYPE('a)))])
    apply (subst fs)
    apply (subst list.map-comp)
    apply (rule arg-cong [where f=concat])
    apply (rule map-cong [OF refl])
   apply (simp add: fld-def off-def sz-def field-lvalue-def field-offset-def field-offset-untyped-def)
    by (simp add: toplevel-field-names-field-lookup-offset-conv)

  note partition =  heap-list-map-partition [where a= ptr-val p, where sz=sz
and off=off and n=size-of TYPE('a) and
    m = length (toplevel-field-names (typ-info-t TYPE('a))), OF no-overflow]
  show ?thesis
    apply (subst concat-conv)
    apply (rule partition)
    subgoal
     apply (subst xmem-type-toplevel-field-names-sum-list-size [OF aggregate, sym-
metric])
      apply (simp add: sz-def)
      apply (subst fs)
      apply (simp add: fld-def comp-def)
      done
    subgoal for i
      apply (simp add: off-def sz-def)
      apply (simp add: fld-def)
      using xmem-type-toplevel-field-names-sum-list-offset [OF aggregate, where i
= i]
      apply simp
```

done
done
qed

lemma *h-val-concat-toplevel-field-names*:
fixes *p*::*'a ptr*
assumes *aggregate*: *aggregate* (*typ-info-t TYPE*(*'a*))
assumes *cgrd*: *c-guard p*
shows
h-val h p =
from-bytes
(*concat* (*map* (*λf*. *heap-list h* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*(*'a*)) [*f*] *0*)))) (*&*(*p→*[*f*])))
(*toplevel-field-names* (*typ-info-t TYPE*(*'a*)))))
using *heap-list-concat-toplevel-field-names* [*OF aggregate cgrd*]
by (*simp add*: *h-val-def*)

end

lemma *set-field-names-u-all-field-names-conv*:
set (*field-names-u* (*typ-uinfo-t TYPE*(*'a*::*mem-type*)) *t*) =
{*f*. *f* ∈ *set* (*all-field-names* (*typ-info-t TYPE*(*'a*))) ∧
(∃ *s*. *field-ti TYPE*(*'a*) *f* = *Some s* ∧ *export-uinfo s* = *t*)}
apply *standard*
subgoal
apply (*clarsimp simp add*: *typ-uinfo-t-def*)
apply (*rule conjI*)
apply (*metis* (*mono-tags*, *lifting*) *all-field-names-export-uinfo field-names-subset-all-field-names*(*1*) *in-mono*)
by (*metis field-lookup-field-ti field-names-Some2*(*1*) *field-names-u-field-names-export-uinfo-conv*(*1*) *wf-desc*)
subgoal
apply *clarsimp*
by (*metis all-field-names-exists-field-names-u*(*1*) *all-field-names-export-uinfo*
field-lookup-field-ti field-names-Some2(*1*) *field-names-u-field-names-export-uinfo-conv*(*1*) *fold-typ-uinfo-t option.inject wf-desc*)
done

lemma *field-names-u-all-field-names-conv*:
field-names-u (*typ-uinfo-t TYPE*(*'a*::*mem-type*)) *t* =
filter (*λf*. (∃ *s*. *field-ti TYPE*(*'a*) *f* = *Some s* ∧ *export-uinfo s* = *t*))
(*all-field-names* (*typ-info-t TYPE*(*'a*)))
using *field-names-u-filter-all-field-names-conv*(*1*)
by (*smt* (*verit*) *all-field-names-exists-field-names-u*(*1*) *all-field-names-export-uinfo field-lookup-field-ti*
field-lookup-uinfo-Some-rev field-ti-def filter-same-eq fold-typ-uinfo-t

mem-Collect-eq option.sel set-field-names-u-all-field-names-conv set-filter wf-desc-typ-tag)

lemma *set-field-names-all-field-names-conv*:
set (*field-names* (*typ-info-t TYPE*('*a*::*mem-type*)) *t*) =
{*f*. *f* ∈ *set* (*all-field-names* (*typ-info-t TYPE*('*a*))) ∧
(∃ *s*. *field-ti TYPE*('*a*) *f* = *Some s* ∧ *export-uinfo s* = *t*)}
using *set-field-names-u-all-field-names-conv* [*of t*]
unfolding *typ-uinfo-t-def*
by (*simp add*: *field-names-u-field-names-export-uinfo-conv*(*1*))

lemma *field-names-all-field-names-conv*:
field-names (*typ-info-t TYPE*('*a*::*mem-type*)) *t* =
filter (λ*f*. ∃ *s*. *field-ti TYPE*('*a*) *f* = *Some s* ∧ *export-uinfo s* = *t*) (*all-field-names*
(*typ-info-t TYPE*('*a*)))
using *field-names-u-all-field-names-conv* [*of t*]
unfolding *typ-uinfo-t-def*
by (*simp add*: *field-names-u-field-names-export-uinfo-conv*(*1*))

lemma *field-lookup-qualified-padding-field-name*:
fixes
t:: ('*a*, '*b*) *typ-info* **and**
st :: ('*a*, '*b*) *typ-info-struct* **and**
ts :: ('*a*, '*b*) *typ-info-tuple list* **and**
x :: ('*a*, '*b*) *typ-info-tuple*
shows
field-lookup t f n = *Some* (*s*, *m*) ⟹ *qualified-padding-field-name f* ⟹ *wf-padding*
t ⟹
is-padding-tag s
field-lookup-struct st f n = *Some* (*s*, *m*) ⟹ *qualified-padding-field-name f* ⟹
wf-padding-struct st ⟹
is-padding-tag s
field-lookup-list ts f n = *Some* (*s*, *m*) ⟹ *qualified-padding-field-name f* ⟹ *wf-padding-list*
ts ⟹
is-padding-tag s
field-lookup-tuple x f n = *Some* (*s*, *m*) ⟹ *qualified-padding-field-name f* ⟹
wf-padding-tuple x ⟹
is-padding-tag s
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f n m* **and** *f n m* **and** *f n m* **and**
f n m)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** (*auto split*: *if-split-asm option.splits*)
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *simp*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*

then show *?case* **by** *simp*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **by** (*auto split*: *if-split-asm option.splits*)
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case*
apply *clarsimp*
by (*metis field-lookup-empty last-tl list.exhaust-sel option.distinct*(*1*)
option.inject prod.inject qualified-pading-field-name-cons qualified-pading-field-name-single)
qed

lemma *all-field-names-empty-typ-info* [*simp*]: *all-field-names* (*empty-typ-info algn n*) = [[]]
by (*simp add*: *empty-typ-info-def*)

lemma *all-field-names-no-padding-empty-typ-info* [*simp*]:
filter (*Not o qualified-padding-field-name*) (*all-field-names* (*empty-typ-info algn n*)) = [[]]
by (*simp*)

lemma *all-field-names-list-append* [*simp*]:
all-field-names-list (*xs* @ *ys*) = *all-field-names-list xs* @ *all-field-names-list ys*
by (*induct xs*) *auto*

lemma *all-field-names-extend-ti*:
fixes
t :: *'a xtyp-info* **and**
st :: *'a xtyp-info-struct* **and**
ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
shows
all-field-names (*extend-ti t s n fn d*) = *all-field-names t* @ (*map* ((#) *fn*) (*all-field-names s*))
all-field-names-struct (*extend-ti-struct st s fn d*) = *all-field-names-struct st* @ (*map* ((#) *fn*) (*all-field-names s*))
all-field-names-list ts = *all-field-names-list ts*
all-field-names-tuple x = *all-field-names-tuple x*
by (*induct t* **and** *st* **and** *ts* **and** *x*) *auto*

lemma *all-field-names-adjust-ti'*:
fixes
t :: *'a xtyp-info* **and**
st :: *'a xtyp-info-struct* **and**
ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
shows
all-field-names (*map-td* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *t*) =

all-field-names t
all-field-names-struct (*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *st*) =
all-field-names-struct st
all-field-names-list (*map-td-list* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *ts*) =
all-field-names-list ts
all-field-names-tuple (*map-td-tuple* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *x*) =
all-field-names-tuple x
by (*induct t* **and** *st* **and** *ts* **and** *x*) (*auto*)

lemma *all-field-names-adjust-ti*[*simp*]:
all-field-names (*adjust-ti t acc upd*) = *all-field-names t*
by (*simp add*: *all-field-names-adjust-ti' adjust-ti-def*)

lemma *all-field-names-no-padding-ti-pad-combine*:
filter (*Not o qualified-padding-field-name*) (*all-field-names* (*ti-pad-combine n t*))
=
filter (*Not o qualified-padding-field-name*) (*all-field-names t*)
by (*simp add*: *ti-pad-combine-def Let-def all-field-names-extend-ti*(*1*) *padding-field-name-pad*)

lemma *all-field-names-ti-typ-combine*:
all-field-names (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *acc upd algn fn t*) =
all-field-names t @ (*map* ((#) *fn*) (*all-field-names* (*typ-info-t TYPE*('*b*))))
by (*simp add*: *ti-typ-combine-def all-field-names-extend-ti*)

lemma *all-field-names-no-padding-ti-typ-combine*:
assumes *not-padding*: ¬ *padding-field-name fn*
shows *filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *acc upd algn fn t*)) =
filter (*Not o qualified-padding-field-name*) (*all-field-names t*) @
(*map* ((#) *fn*) (*filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*typ-info-t TYPE*('*b*)))))
proof −
from *not-padding* **have** *eq*: *Not* ∘ *qualified-padding-field-name* ∘ (#) *fn* = *Not* ∘ *qualified-padding-field-name*
apply −
apply (*rule ext*)
apply (*auto simp add*: *neq-Nil-conv*)
done
from *not-padding* **show** *?thesis*
apply (*simp add*: *all-field-names-ti-typ-combine all-field-names-extend-ti*)
apply (*simp add*: *filter-map eq*)
done
qed

lemma *all-field-names-no-padding-ti-typ-pad-combine*:
assumes *not-padding*: ¬ *padding-field-name fn*

shows *filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *acc upd algn fn t*)) =
filter (*Not o qualified-padding-field-name*) (*all-field-names t*) @
(*map* ((#) *fn*) (*filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*typ-info-t TYPE*('*b*)))))
by (*simp add*: *ti-typ-pad-combine-def Let-def*
all-field-names-no-padding-ti-pad-combine all-field-names-no-padding-ti-typ-combine [*OF not-padding*])

lemma *all-field-names-map-align*[*simp*]: *all-field-names* (*map-align n t*) = *all-field-names t*
by (*cases t*) (*simp add*: *map-align-def*)

lemma *all-field-names-no-padding-final-pad*:
filter (*Not o qualified-padding-field-name*) (*all-field-names* (*final-pad n t*))
= *filter* (*Not o qualified-padding-field-name*) (*all-field-names t*)
by (*simp add*: *final-pad-def Let-def all-field-names-no-padding-ti-pad-combine*)

lemmas *all-field-names-filter-no-padding-combinator-simps* =
all-field-names-no-padding-empty-typ-info
all-field-names-no-padding-final-pad
all-field-names-no-padding-ti-typ-pad-combine
all-field-names-no-padding-ti-typ-combine

lemma *all-field-names-array-tag-n*: *all-field-names* ((*array-tag-n n*)::('*a*::*c-type*,'*b*::*finite*) *array xtyp-info*) =
[] #
concat (*map* (λ*i*. (*map* ((#) (*replicate i CHR* ''*1*'')) (*all-field-names* (*typ-info-t TYPE*('*a*))))) [*0*..<*n*])
apply (*induct n*)
apply (*simp add*: *atn-base*)
apply (*auto simp add*: *atn-rec all-field-names-ti-typ-combine*)
done

lemma *all-field-names-array*:
all-field-names (*typ-info-t TYPE*('*a*::*c-type*['*b*::*finite*])) =
[] #
concat (*map* (λ*i*. (*map* ((#) (*replicate i CHR* ''*1*'')) (*all-field-names* (*typ-info-t TYPE*('*a*))))) [*0*..<*CARD*('*b*)])
by (*simp add*: *typ-info-array array-tag-def all-field-names-array-tag-n*)

lemma *not-padding-field-name-replicate-1*[*simp*]: *padding-field-name* (*replicate n CHR* ''*1*'') = *False*
by (*cases n*) *auto*

lemma *non-empty-not-padding-field-conv*: (*x* ≠ [] ⟶ ¬ *padding-field-name* (*last x*)) ⟷ ¬ *qualified-padding-field-name x*
by (*cases x*) *auto*

named-theorems *all-field-names-no-padding* **and** *set-all-field-names-no-padding*

definition *all-field-names-no-padding* :: ('*a*, '*b*) *typ-desc* ⇒ *qualified-field-name list*
where
all-field-names-no-padding t = *filter* (*Not o qualified-padding-field-name*) (*all-field-names t*)

lemma *all-field-names-no-padding-combinator-simps*:
all-field-names-no-padding (*empty-typ-info algn nm*) = [[]]
all-field-names-no-padding (*final-pad n t*) = *all-field-names-no-padding t*
¬ *padding-field-name fn* ⟹
all-field-names-no-padding (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *acc upd algn fn t*) =
all-field-names-no-padding t @
map ((#) *fn*)
(*all-field-names-no-padding* (*typ-info-t TYPE*('*b*::*c-type*)))
¬ *padding-field-name fn* ⟹
all-field-names-no-padding (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *acc upd algn fn t*) =
all-field-names-no-padding t @
map ((#) *fn*)
(*all-field-names-no-padding* (*typ-info-t TYPE*('*b*::*c-type*)))
by (*simp add*: *all-field-names-no-padding-def all-field-names-filter-no-padding-combinator-simps*)

lemma *all-field-names-filter-no-padding-array*:
filter (*Not o qualified-padding-field-name*) (*all-field-names* (*typ-info-t TYPE*('*a*::*c-type*['*b*::*finite*]))) =
[] #
concat (*map* (λ*i*. (*map* ((#) (*replicate i CHR* ''1'')) (*filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*typ-info-t TYPE*('*a*)))))) [*0*..<*CARD*('*b*)])
apply (*simp add*: *all-field-names-array*)
apply (*simp add*: *filter-concat*)
apply (*simp add*: *comp-def filter-map add*: *non-empty-not-padding-field-conv*)
done

lemma *all-field-names-no-padding-array*[*all-field-names-no-padding*]:
all-field-names-no-padding (*typ-info-t TYPE*('*a*::*c-type*['*b*::*finite*])) =
[] #
concat (*map* (λ*i*. (*map* ((#) (*replicate i CHR* ''1'')) (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*))))) [*0*..<*CARD*('*b*)])
by (*simp add*: *all-field-names-no-padding-def all-field-names-filter-no-padding-array*)

lemma *set-all-field-names-no-padding-array*[*set-all-field-names-no-padding*]:
set (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*::*c-type*['*b*::*finite*]))) =
insert []
(⋃ *x*∈{*0*..<*CARD*('*b*)}.

(#) (*replicate x CHR ''1''*) ‘
set (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*))))

by (*simp add*: *all-field-names-no-padding-array*)

lemma *sub-typ-trans*: $t \leq_\tau s \Longrightarrow s \leq_\tau w \Longrightarrow t \leq_\tau w$
by (*simp add*: *sub-typ-def*)

lemma *sub-typ-trans-rev*: $s \leq_\tau w \Longrightarrow t \leq_\tau s \Longrightarrow t \leq_\tau w$
by (*simp add*: *sub-typ-def*)

lemma *element-typ-le-array-typ*: *typ-uinfo-t TYPE*('*a*::*mem-type*) ≤ *typ-uinfo-t TYPE*('*a*['*b*::*finite*])
proof −
have *0* < *CARD*('*b*) **by** *simp*
from *field-lookup-array* [*OF this*, *of 0*, *simplified*]
have *field-lookup* (*typ-info-t TYPE*('*a*['*b*])) [[]] *0* =
Some (*adjust-ti* (*typ-info-t TYPE*('*a*)) (λ*x*. *x*.[*0*]) (λ*x f*. *Arrays.update f 0 x*), *0*) **.**
from *td-set-field-lookupD* [*OF this*]
have (*adjust-ti* (*typ-info-t TYPE*('*a*)) (λ*x*. *x*.[*0*]) (λ*x f*. *Arrays.update f 0 x*), *0*)
∈ *td-set* (*typ-info-t TYPE*('*a*['*b*])) *0***.**
from *td-set-export-uinfoD* [*OF this*]
show *?thesis*
by (*auto simp add*: *export-uinfo-adjust-ti typ-uinfo-t-def typ-tag-le-def*)
qed

lemma *element-typ-subtyp-array-typ*: *TYPE* ('*a*::*mem-type*) ≤$_\tau$ *TYPE*('*a*['*b*::*finite*])
by (*simp add*: *sub-typ-def element-typ-le-array-typ*)

lemma *field-lookup-sub-typ*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*::*c-type*)) *f 0* = *Some* (*s*, *m*)
assumes *match*: *export-uinfo s* = *export-uinfo* (*typ-info-t TYPE*('*b*::*c-type*))
shows *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
using *match td-set-export-uinfoD* [*OF td-set-field-lookupD* [*OF fl*]]
by (*auto simp add*: *sub-typ-def typ-tag-le-def typ-uinfo-t-def*)

lemma *field-lookup-sub-typ'*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*::*c-type*)) *f 0* ≡ *Some* (*adjust-ti* (*typ-info-t TYPE*('*b*::*mem-type*)) *acc upd*, *n*)
assumes *fg-cons*: *fg-cons acc upd*
shows *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
apply (*rule field-lookup-sub-typ* [*of f*])
apply (*simp add*: *fl*)
using *fg-cons*
by (*simp add*: *export-uinfo-adjust-ti*)

lemma *all-field-names-no-padding-word*[*all-field-names-no-padding*]:
all-field-names-no-padding (*typ-info-t* (*TYPE*('*a*::*len8 word*))) = [[]]
by (*simp add*: *all-field-names-no-padding-def*)

lemma *set-all-field-names-no-padding-word*[*set-all-field-names-no-padding*]:
set (*all-field-names-no-padding* (*typ-info-t* (*TYPE*('*a*::*len8 word*)))) = {[]}
by (*simp add*: *all-field-names-no-padding-def*)

lemma *all-field-names-no-padding-ptr*[*all-field-names-no-padding*]:
all-field-names-no-padding (*typ-info-t* (*TYPE*('*a*::*c-type ptr*))) = [[]]
by (*simp add*: *all-field-names-no-padding-def*)

lemma *set-all-field-names-no-padding-ptr*[*set-all-field-names-no-padding*]:
set (*all-field-names-no-padding* (*typ-info-t* (*TYPE*('*a*::*c-type ptr*)))) = {[]}
by (*simp add*: *all-field-names-no-padding-def*)

definition *field-names-no-padding*::('*a*, '*b*) *typ-info* ⇒ *typ-uinfo* ⇒ *qualified-field-name list*
where *field-names-no-padding t s* = *filter* (*Not o qualified-padding-field-name*) (*field-names t s*)

lemma *set-field-names-no-padding-all-field-names-no-padding-conv*:
set (*field-names-no-padding* (*typ-info-t TYPE*('*a*::*mem-type*)) *t*) =
{*f* ∈ *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*))).
∃ *s n*. *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*) ∧ *export-uinfo s* = *t*}
using *set-field-names-all-field-names-conv* [**where** '*a*='*a* **and** *t*=*t*]
by (*auto simp add*: *field-ti-def all-field-names-no-padding-def field-names-no-padding-def split*: *option.split*)

lemma *field-names-no-padding-all-field-names-no-padding-conv*:
field-names-no-padding (*typ-info-t TYPE*('*a*::*mem-type*)) *t* =
filter (λ*f*. ∃ *s n*. *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*) ∧ *export-uinfo s* = *t*)
(*all-field-names-no-padding* (*typ-info-t TYPE*('*a*)))
using *field-names-all-field-names-conv* [**where** '*a*='*a* **and** *t*=*t*]
by (*auto simp add*: *field-ti-def all-field-names-no-padding-def field-names-no-padding-def intro*: *filter-same-eq split*: *option.split*)

lemma *subset-all-field-names-no-padding-all-field-names*:
set (*all-field-names-no-padding t*) ⊆ *set* (*all-field-names t*)
by (*simp add*: *all-field-names-no-padding-def*)

lemma *all-field-names-typ-uinfo-t-conv*:
all-field-names (*typ-info-t* (*TYPE*('*a*::*c-type*))) = *all-field-names* (*typ-uinfo-t* (*TYPE*('*a*::*c-type*)))
by (*simp add*: *all-field-names-export-uinfo typ-uinfo-t-def*)

lemma *all-field-names-no-padding-typ-uinfo-t-conv*:
all-field-names-no-padding (*typ-info-t* (*TYPE*('*a*::*c-type*))) = *all-field-names-no-padding* (*typ-uinfo-t* (*TYPE*('*a*::*c-type*)))
by (*simp add*: *all-field-names-no-padding-def all-field-names-typ-uinfo-t-conv*)

lemma *update-ti-to-bytes-p*[*simp*]:
update-ti (*typ-info-t TYPE*('*a*::*xmem-type*)) (*to-bytes-p* (*v*::'*a*)) *w* = *v*
apply (*subst field-lookup-update-ti-from-bytes-field-conv*[*OF field-lookup-empty* , **where** *xbs*=*replicate* (*size-of TYPE*('*a*)) *0* **and** *v*=*v*])
unfolding *typ-uinfo-t-def* **apply** (*rule refl*)
apply (*simp add*: *size-of-def*)
apply (*simp add*: *to-bytes-p-def to-bytes-def*)
apply (*subst field-lookup-update-ti-from-bytes-field-conv*[*symmetric*, **where** *vf*=*v*, *OF field-lookup-empty*])
apply (*simp-all add*: *size-of-def typ-uinfo-t-def*)
apply (*simp add*: *to-bytes-p-def to-bytes-def*)
apply (*intro lense.update-access*)
done

context *mem-type*
begin

lemma *mem-type-access-ti-super-update-bs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-td s*
shows *access-ti* (*typ-info-t TYPE*('*a*))
(*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs' bs n*) =
super-update-bs (*access-ti s w bs'*) (*access-ti* (*typ-info-t TYPE*('*a*)) *v bs*) *n*
proof −
have *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *0* + *n*) **using** *fl* **by** *simp*
from *access-ti-super-update-bs-of-wf*(*1*)[*OF this lbs' lbs*[*unfolded size-of-def*]]
show *?thesis* **by** *simp*
qed

end

lemma *update-ti-undefined*[*simp*]:
assumes *NO-MATCH undefined w* **assumes** *bs*: *length bs* = *size-of TYPE*('*a*)
shows *update-ti* (*typ-info-t TYPE*('*a*::*xmem-type*)) *bs w* =
update-ti (*typ-info-t TYPE*('*a*)) *bs undefined*
by (*simp add*: *bs size-of-def upd-rf update-ti-update-ti-t*)

11.28 *heap-upd* and *heap-upd-list*

11.29 *heap-upd*

lemma *heap-upd-id*: *heap-upd* (*p*::'*a*::*xmem-type ptr*) *id* = *id*

by (*simp add*: *heap-upd-def fun-eq-iff xmem-type-class.heap-update-id*)

lemma *heap-upd-const*: *heap-upd p* (*λ-. x*) = *heap-update p x*
by (*simp add*: *heap-upd-def*[*abs-def*])

lemma *heap-upd-comp*: *heap-upd* (*p*::*'a*::*xmem-type ptr*) (*f* ∘ *g*) = *heap-upd p f* ∘ *heap-upd p g*
by (*simp add*: *heap-upd-def fun-eq-iff h-val-heap-update heap-update-collapse*)

lemma *hrs-mem-update-heap-upd*:
hrs-mem-update (*heap-upd p g*) *h* = *hrs-mem-update* (*heap-update p* (*g* (*h-val* (*hrs-mem h*) *p*))) *h*
by (*cases h*) (*simp add*: *hrs-mem-update-def heap-upd-def hrs-mem-def*)

lemma *heap-update-eq-heap-upd-list*:
fixes *p* :: *'a*::*mem-type ptr*
shows *heap-update p x* =
heap-upd-list (*size-of TYPE*(*'a*)) (*ptr-val p*) (*access-ti* (*typ-info-t TYPE*(*'a*)) *x*)
by (*simp add*: *heap-update-def fun-eq-iff heap-upd-list-def to-bytes-def*)

lemma *heap-upd-list-id*[*simp*]: *heap-upd-list n p id* = *id*
by (*simp add*: *heap-upd-list-def fun-eq-iff heap-update-list-id'*)

lemma *heap-upd-list-access-ti-typ-info-t*[*simp*]:
sz = *size-of TYPE*(*'a*) ⟹
heap-upd-list sz p (*access-ti* (*typ-info-t TYPE*(*'a*::*xmem-type*)) *v*) =
heap-update (*PTR*(*'a*) *p*) *v*
by (*simp add*: *heap-update-def heap-upd-list-def fun-eq-iff to-bytes-def*)

lemma *heap-list-heap-upd-list*:
n ≤ *addr-card* ⟹ *length xs* = *n* ⟹ (⋀*xs*. *length xs* = *n* ⟹ *length* (*f xs*) = *n*) ⟹
heap-list (*heap-upd-list n p f h*) *n p* = *f* (*heap-list h n p*)
unfolding *heap-upd-list-def*
by (*subst heap-list-heap-update-list-id*) *simp-all*

lemma *heap-upd-list-comp*:
assumes *n* ≤ *addr-card length xs* = *n*
assumes *f*: ⋀*xs*. *length xs* = *n* ⟹ *length* (*f xs*) = *n*
assumes *g*: ⋀*xs*. *length xs* = *n* ⟹ *length* (*g xs*) = *n*
shows *heap-upd-list n p* (*f* ∘ *g*) = *heap-upd-list n p f* ∘ *heap-upd-list n p g*
using *assms*
by (*simp add*: *fun-eq-iff heap-upd-list-def heap-list-heap-update-list-id*
heap-update-list-overwrite)

lemma *heap-update-list-append*:
fixes *v* :: *word8*
shows *heap-update-list s* (*xs* @ *ys*) *hp* =
heap-update-list (*s* + *of-nat* (*length xs*)) *ys* (*heap-update-list s xs hp*)

proof (*induct xs arbitrary*: *ys rule*: *rev-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*snoc v' vs'*)
show *?case*
apply (*simp add*: *snoc.hyps field-simps*)
apply (*rule arg-cong* [**where** *f* = *heap-update-list* (*1* + (*s* + *of-nat* (*length vs'*))) *ys*])
apply (*rule ext*)
apply *simp*
done
qed

lemma *heap-update-list-super-update-bs*:
length bs + *n* ≤ *length bs'* ⟹ *length bs'* ≤ *addr-card* ⟹
heap-update-list (*p* + *of-nat n*) *bs* (*heap-update-list p bs' h*) =
heap-update-list p (*super-update-bs bs bs' n*) *h*
apply (*subst super-update-bs-take-drop*[*symmetric*, *of n length bs bs'*])
unfolding *super-update-bs-def heap-update-list-append*
apply *simp-all*
apply (*subst heap-update-list-commute*)
subgoal
apply (*subst* (*2*) *add-0-right*[*symmetric*])
unfolding *intvl-disj-offset add.assoc*
apply (*cases length bs* + *n* = *length bs'*)
apply *simp*
apply (*intro intvl-disj-left*)
apply (*simp-all add*: *addr-card-def card-word unat-of-nat-eq*)
done
apply (*subst heap-update-list-overwrite*)
apply *simp-all*
done

lemma *update-ti-adjust-ti*:
fixes *t*::*'a xtyp-info*
and *st*::*'a xtyp-info-struct*
and *ts*::*'a xtyp-info-tuple list*
and *x*::*'a xtyp-info-tuple*
assumes *fg-cons*: *fg-cons f g*
shows
update-ti (*adjust-ti t* (*f*::*'b* ⇒ *'a*) (*g*::*'a* ⇒ *'b* ⇒ *'b*)) *bs v* = *g* (*update-ti t bs* (*f v*)) *v*
update-ti-struct (*map-td-struct* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *st*) *bs v* = *g* (*update-ti-struct st bs* (*f v*)) *v*
update-ti-list (*map-td-list* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *ts*) *bs v* = *g* (*update-ti-list ts bs* (*f v*)) *v*
update-ti-tuple (*map-td-tuple* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *x*) *bs v* = *g* (*update-ti-tuple x bs* (*f v*)) *v*

unfolding *adjust-ti-def*
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *v bs* **and** *v bs* **and** *v bs* **and** *v bs*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** (*auto simp add*: *update-desc-def*)
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **using** *fg-cons* **by** (*auto simp add*: *fg-cons-def*)
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **apply** (*cases dt-tuple*) **using** *fg-cons* **by** (*auto simp add*: *fg-cons-def*)
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** *auto*
qed

lemma *field-ti-field-lookupE*:
⟦ *field-ti TYPE('a :: c-type) f = Some t*; ⋀*n*. ⟦*field-lookup* (*typ-info-t TYPE('a)*) *f 0 = Some* (*t*, *n*) ⟧ ⟹ *P* ⟧ ⟹ *P*
unfolding *field-ti-def*
by (*clarsimp split*: *option.splits*)

lemma *field-ti-append-field-lookup*:
field-ti TYPE('a::wf-type) f = Some u ⟹ *field-lookup u g l = Some* (*v*, *k*) ⟹
field-ti TYPE('a) (f @ g) = Some v
by (*auto simp*: *field-ti-def field-lookup-append split*: *option.splits*)

lemma *field-tiD*:
field-ti TYPE('a::mem-type) f = Some t ⟹
field-lookup (*typ-info-t TYPE('a)*) *f 0 = Some* (*t*, *field-offset TYPE('a) f*)
by (*metis field-lookup-offset-eq field-ti-field-lookupE*)

lemma *wf-fd-field-lookup-mem-type*: *field-lookup* (*typ-info-t*(*TYPE*('*a*::*mem-type*))) *f m = Some* (*s*, *n*) ⟹ *wf-fd s*
apply (*erule wf-fd-field-lookupD*)
by *simp*

lemma *wf-fd-field-ti-mem-type*: *field-ti TYPE('a::mem-type) f = Some s* ⟹ *wf-fd s*
unfolding *field-ti-def*
by (*auto simp add*: *wf-fd-field-lookup-mem-type split*: *option.splits prod.splits*)

lemma *field-lookup-offset-non-zero*:
NO-MATCH 0 m $\Longrightarrow$ *field-lookup t f 0 = Some (t', n)* $\Longrightarrow$ *field-lookup t f m = Some (t', m + n)*
by (*simp add*: *field-lookup-offset'* [**where** *m'=0*])

lemma *field-lookup-append-Some*:
assumes *wf*: *wf-desc t*
shows *field-lookup t (f@g) n = Some (s, m)* $\Longrightarrow$
$\exists$ *w k. field-lookup t f n = Some (w, k)* $\wedge$ *field-lookup w g k = Some (s, m)*
using *wf*
proof (*induct n* $\equiv$ *length (f @ g) arbitrary*: *f g n s m t*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
show *?case*
by (*metis Suc.prems*(*1*) *Suc.prems*(*2*)
field-lookup-prefix-None''(*1*) *field-lookup-prefix-Some''*(*1*) *not-Some-eq-tuple*)
qed

11.30 *merge-ti*

definition *merge-ti* :: ('*a field-desc*, '*b*) *typ-desc* $\Rightarrow$ '*a* $\Rightarrow$ '*a* $\Rightarrow$ '*a* **where**
merge-ti t a b = update-ti t (*access-ti*$_0$ *t a*) *b*

lemma *merge-ti-adjust-ti*[*simp*]:
fg-cons g s $\Longrightarrow$ *merge-ti* (*adjust-ti* (*typ-info-t TYPE*('*a*::*xmem-type*)) *g s*) = (λ*a*. *s* (*g a*))
by (*simp add*: *fun-eq-iff merge-ti-def update-ti-adjust-ti*)

lemma *is-scene-merge-ti*:
assumes *t*: *field-ti TYPE*('*a*::*xmem-type*) *f = Some t* **shows** *is-scene* (*merge-ti t*)
proof −
interpret *padding-lense access-ti t update-ti t size-td t*
using *t*[*THEN field-tiD*] **by** (*rule field-lookup-padding-lense*)

show *?thesis*
proof
show *merge-ti t* (*merge-ti t a b*) *c = merge-ti t a c* **for** *a b c*
by (*simp-all add*: *merge-ti-def access-ti*$_0$*-def*) (*metis access-update update-access*)
qed (*simp-all add*: *merge-ti-def access-ti*$_0$*-def update-access double-update*)
qed

lemma *merge-ti-update-ti-disj*:
assumes $*$: *field-ti TYPE*('*a*::*xmem-type*) *f = Some t field-ti TYPE*('*a*::*xmem-type*) *g = Some u*
and *f-g*: *disj-fn f g*
assumes *bs*: *length bs = size-td u*

shows *merge-ti t x* (*update-ti u bs y*) = *update-ti u bs* (*merge-ti t x y*)
proof −
have *eq*[*simp*]: ⋀*ts us x*. *length ts* = *size-td t* ⟹ *length us* = *size-td u* ⟹
update-ti t ts (*update-ti u us x*) = *update-ti u us* (*update-ti t ts x*)
using *fu-commutes-lookup-disjD*[*OF* ∗[*THEN field-tiD*] *f-g*]
by (*simp add*: *wf-lf-fdp fu-commutes-def update-ti-t-def split*: *if-splits*)
have [*simp*]: *wf-fd t wf-fd u*
using ∗[*THEN wf-fd-field-ti-mem-type*] **by** *auto*
show *?thesis*
by (*simp add*: *merge-ti-def access-ti*$_0$*-def length-fa-ti bs*)
qed

lemma *disjnt-scene-merge-ti*:
assumes ∗: *field-ti TYPE*('*a*) *f* = *Some t field-ti TYPE*('*a*::*xmem-type*) *g* = *Some u*
and *f-g*: *disj-fn f g*
shows *disjnt-scene* (*merge-ti t*) (*merge-ti u*)
apply (*clarsimp simp*: *disjnt-scene-def*)
apply (*subst* (*2 3*) *merge-ti-def*)
apply (*rule merge-ti-update-ti-disj*[*OF assms*])
apply (*simp add*: ∗[*THEN wf-fd-field-ti-mem-type*] *length-fa-ti access-ti*$_0$*-def*)
done

lemma *access-ti-merge-ti-sub*:
assumes *r*: *field-ti TYPE*('*a*::*xmem-type*) *r* = *Some t* **and** *t*: *field-lookup t f 0* = *Some* (*v*, *n*)
and *bs*: *length bs* = *size-td v*
shows *access-ti v* (*merge-ti t x y*) *bs* = *access-ti v x bs*
proof −
let *?xbs* = *super-update-bs bs* (*replicate* (*size-td t*) *0*) *n*
let *?x-y* = *update-ti t* (*access-ti t x* (*replicate* (*size-td t*) *0*)) *y*
have *wf-t*: *wf-fd t wf-desc t wf-size-desc t*
using *wf-fd-field-ti-mem-type*[*OF r*] *r*
field-lookup-wf-desc-pres(*1*)[*of typ-info-t TYPE*('*a*) *r 0*]
field-lookup-wf-size-desc-pres(*1*)[*of typ-info-t TYPE*('*a*) *r 0*]
by (*auto elim*: *field-ti-field-lookupE*)

have *length-xbs*: *length ?xbs* = *size-td t*
using *field-lookup-offset-size'*[*OF t*] **by** (*subst length-super-update-bs*) (*simp-all add*: *bs*)
have *n-t*: *n* ≤ *length* (*replicate* (*size-td t*) *0*)
using *field-lookup-offset-size'*[*OF t*] **by** *auto*
have *eq*: *access-ti v x bs* =
take (*size-td v*) (*drop n* (*access-ti t x* (*super-update-bs bs* (*replicate* (*size-td t*) *0*) *n*)))
for *x*
using *field-lookup-access-ti-take-drop*[*OF t wf-t length-xbs*]
unfolding *take-drop-super-update-bs*[*OF bs n-t*] **.**

```
  from wf-fd-consD[OF wf-t(1)]
  have eq-ac:
    length bs = size-td t ⟹ length bs' = size-td t ⟹
        access-ti t (update-ti t bs v) bs' = access-ti t (update-ti t bs v') bs'
    length bs = size-td t ⟹ update-ti t (access-ti t v bs) v = v
    for bs bs' v v'
    by (auto simp: fd-cons-def fd-cons-desc-def fd-cons-update-access-def
                   fd-cons-access-update-def update-ti-t-def length-fa-ti wf-t)

  have [simp]: access-ti v ?x-y bs = access-ti v x bs unfolding eq
    by (subst eq-ac(1)[of - - - x]) (simp-all add: wf-t length-fa-ti length-xbs eq-ac)
  show ?thesis
    by (simp add: merge-ti-def fd-cons-access-update-def access-ti0-def)
qed

lemma access-ti-merge-ti-disj:
  assumes f: field-ti TYPE('a::xmem-type) f = Some t
  assumes g: field-ti TYPE('a::xmem-type) g = Some u
    and f-g: disj-fn f g
    and bs: length bs = size-td u
  shows access-ti u (merge-ti t x y) bs = access-ti u y bs
  apply (subst is-scene.idem[symmetric, OF is-scene-merge-ti, OF g, of y])
  apply (subst disjnt-sceneD[OF disjnt-scene-merge-ti, OF f g f-g])
  apply (rule access-ti-merge-ti-sub[OF g field-lookup-empty bs])
  done

lemma merge-ti-update-ti-sub:
  assumes r: field-ti TYPE('a::xmem-type) r = Some t
    and t: field-lookup t f 0 = Some (v, n)
    and bs: length bs = size-td v
  shows merge-ti t x (update-ti v bs y) = merge-ti t x y
proof −
  have wf-t: wf-fd t wf-desc t wf-size-desc t
    using wf-fd-field-ti-mem-type[OF r] r
      field-lookup-wf-desc-pres(1)[of typ-info-t TYPE('a) r 0]
      field-lookup-wf-size-desc-pres(1)[of typ-info-t TYPE('a) r 0]
    by (auto elim: field-ti-field-lookupE)

  from wf-fd-consD[OF wf-t(1)]
  have eq-uu:
    length bs = size-td t ⟹ length bs' = size-td t ⟹
      update-ti t bs (update-ti t bs' v) = update-ti t bs v
    for bs bs' v
   by (auto simp: fd-cons-def fd-cons-desc-def fd-cons-double-update-def update-ti-t-def
                  length-fa-ti wf-t split: if-splits)

  from wf-fd-consD[OF wf-t(1)]
  have eq-ua:
    length bs = size-td t ⟹ update-ti t (access-ti t v bs) v = v
```

```
    for bs v
  by (auto simp: fd-cons-def fd-cons-desc-def fd-cons-update-access-def update-ti-t-def
                 length-fa-ti wf-t split: if-splits)

  have n-v-t: n + size-td v ≤ size-td t
    using field-lookup-offset-size'[OF t] by simp
  with fi-fu-consistentD[OF t wf-t(1)] have eq-u-sub:
    length bs' = size-td v ⟹ length bs = size-td t ⟹
      update-ti t (super-update-bs bs' bs n) y =
      update-ti v bs' (update-ti t bs y) for bs' y bs
    by (simp add: update-ti-t-def)

  have update-ti v bs y =
    update-ti v bs (update-ti t (access-ti t y (replicate (size-td t) 0)) y)
    by (subst eq-ua) (simp-all add: bs)
  also have ... = update-ti t (super-update-bs bs (access-ti t y (replicate (size-td
t) 0)) n) y
    by (subst eq-u-sub) (simp-all add: bs length-fa-ti wf-t)
  finally show ?thesis
    using bs n-v-t by (simp add: access-ti0-def length-fa-ti wf-t eq-uu merge-ti-def)
qed

lemma merge-ti-merge-ti-sub1:
  assumes t: field-ti TYPE('a::xmem-type) f = Some t
  assumes u: field-ti TYPE('a) (f @ g) = Some u
  shows merge-ti t a (merge-ti u a b) = merge-ti t a b
proof −
  obtain n m where n: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
    and m: field-lookup (typ-info-t TYPE('a)) (f@g) 0 = Some (u, m)
    using t u by (auto elim!: field-ti-field-lookupE)
  from n field-lookup-append-Some[OF - m] have u': field-lookup t g 0 = Some (u,
m − n)
    by (simp add: CTypes.field-lookup-offset2)
  have [simp]: fd-cons u
    using fd-cons fd-consistentD m by blast
  show ?thesis
    by (subst (2) merge-ti-def)
       (simp add: merge-ti-update-ti-sub[OF t u'] fd-cons-length-p)
qed

lemma merge-ti-merge-ti-sub2:
  assumes t: field-ti TYPE('a::xmem-type) f = Some t
  assumes u: field-ti TYPE('a) (f @ g) = Some u
  shows merge-ti u a (merge-ti t a b) = merge-ti t a b
proof −
  obtain n m where n: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
    and m: field-lookup (typ-info-t TYPE('a)) (f@g) 0 = Some (u, m)
    using t u by (auto elim!: field-ti-field-lookupE)
  from n field-lookup-append-Some[OF - m] have u': field-lookup t g 0 = Some (u,
```

$m - n$)
  by (*simp add*: *CTypes.field-lookup-offset2*)
 have [*simp*]: *fd-cons u*
  using *fd-cons fd-consistentD m* **by** *blast*

 interpret *u*: *is-scene merge-ti u*
  using *u* **by** (*rule is-scene-merge-ti*)

 have *eq*: *access-ti*$_0$ *u a* = *access-ti*$_0$ *u* (*merge-ti t a b*)
  by (*simp add*: *access-ti-merge-ti-sub*[*OF t u'*] *access-ti*$_0$*-def*)
 show *?thesis*
  apply (*subst* (*1*) *merge-ti-def*)
  apply (*subst eq*)
  apply (*subst* (*1*) *merge-ti-def*[*symmetric*])
  apply (*rule u.idem*)
  done
qed

lemma *comm-scene-merge-ti-sub*:
 assumes ∗: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t field-ti TYPE*('*a*) (*f* @ *g*)
= *Some u*
 shows *comm-scene* (*merge-ti t*) (*merge-ti u*)
 by (*clarsimp simp*: *comm-scene-def merge-ti-merge-ti-sub1*[*OF* ∗] *merge-ti-merge-ti-sub2*[*OF*
∗])

lemma *comm-scene-merge-ti*:
  assumes ∗: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t field-ti TYPE*('*a*) *g* =
Some u
 shows *comm-scene* (*merge-ti t*) (*merge-ti u*)
proof *cases*
 assume *disj-fn f g*
 from *disjnt-scene-merge-ti*[*OF* ∗ *this*] **show** *?thesis*
  by *blast*
next
 assume ¬ *disj-fn f g*
 with ∗ **show** *?thesis*
  by (*auto simp*: *disj-fn-def less-eq-list-def prefix-def intro*: *comm-scene-merge-ti-sub*)
qed

11.30.1 *merge-ti-list*

definition *merge-ti-list* **where**
 merge-ti-list ts a = *fold* (λ*t*. *merge-ti t a*) *ts*

lemma *is-scene-merge-ti-list*:
 list-all (λ*u*. ∃*f*. *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some u*) *ts* ⟹
  is-scene (*merge-ti-list ts*)
 unfolding *merge-ti-list-def*
 apply (*intro is-scene-fold'*)

by (*auto simp*: *list-all-iff pairwise-def is-scene-merge-ti*)
(*meson comm-scene-merge-ti*)

lemma *merge-ti-list-nil*[*simp*]: *merge-ti-list* [] = (λ*a*. *id*)
by (*simp add*: *merge-ti-list-def fun-eq-iff*)

lemma *merge-ti-list-cons*[*simp*]:
merge-ti-list (*t* # *ts*) *a* = *merge-ti-list ts a* ∘ *merge-ti t a*
by (*simp add*: *merge-ti-list-def*)

lemma *merge-ti-list-append*[*simp*]:
merge-ti-list (*ts* @ *ts'*) *x y* = *merge-ti-list ts' x* (*merge-ti-list ts x y*)
by (*simp add*: *merge-ti-list-def*)

lemma *access-ti-merge-ti-list*:
assumes *ts*: *list-all* (λ(*f*, *u*). *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some u*) *ts*
distinct-prop disj-fn (*map fst ts*)
and *r-t*: (*r*, *t*) ∈ *set ts* **and** *v*: *field-lookup t f 0* = *Some* (*v*, *n*)
and *bs*: *length bs* = *size-td v*
shows *access-ti v* (*merge-ti-list* (*map snd ts*) *x y*) *bs* = *access-ti v x bs*
using *ts r-t*
proof (*induction ts rule*: *rev-induct*)
case (*snoc r-t ts*)
from *this*(*2*,*3*) **have** ∗:
list-all (λ(*f*, *u*). *field-ti TYPE*('*a*) *f* = *Some u*) *ts*
distinct-prop disj-fn (*map fst ts*)
and *disj*: ∀ *t*∈*set ts*. *disj-fn* (*fst t*) (*fst r-t*)
and ∗∗: *field-ti TYPE*('*a*) (*fst r-t*) = *Some* (*snd r-t*)
by (*auto simp*: *distinct-prop-append*)

show *?case*
proof *cases*
assume [*simp*]: *r-t* = (*r*, *t*)
with ∗∗ **have** *ft-r*: *field-ti TYPE*('*a*) *r* = *Some t* **by** *simp*
show *?thesis*
by (*simp add*: *access-ti-merge-ti-sub*[*OF ft-r v bs*])
next
assume *r-t* ≠ (*r*, *t*)
with *snoc.prems* **have** *r-t*: (*r*, *t*) ∈ *set ts* **by** *auto*
with ∗ **have** *field-ti TYPE*('*a*) *r* = *Some t*
by (*auto simp*: *list-all-iff*)
from *field-ti-append-field-lookup*[*OF this v*]
have *r-f*: *field-ti TYPE*('*a*) (*r* @ *f*) = *Some v* **.**
from *r-t disj*
have *disj-fn* (*fst r-t*) *r* **by** (*auto simp*: *disj-fn-def*)
then have *disj*: *disj-fn* (*fst r-t*) (*r* @ *f*)
by (*rule disj-fn-append-right*)
from *snoc.IH*[*OF* ∗ *r-t*] **show** *?thesis*
by (*simp add*: *access-ti-merge-ti-disj*[*OF* ∗∗ *r-f disj bs*])

qed
qed *simp*

lemma *merge-ti-list-update-ti*:
assumes *ts*: *list-all* (λ(*f*, *u*). *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some u*) *ts* (*f-t*, *t*) ∈ *set ts*
and *disj*: *distinct-prop disj-fn* (*map fst ts*)
and *t*: *field-lookup t f 0* = *Some* (*v*, *n*)
and *bs*: *length bs* = *size-td v*
shows *merge-ti-list* (*map snd ts*) *x* (*update-ti v bs y*) = *merge-ti-list* (*map snd ts*) *x y*
using *ts disj*
proof (*induction ts arbitrary*: *y*)
case (*Cons u*′ *ts*)
then obtain *t*′ *f*′ **where** *ts*[*simp*]: *list-all* (λ(*f*, *u*). *field-ti TYPE*('*a*) *f* = *Some u*) *ts*
and *f*′: *field-ti TYPE*('*a*) *f*′ = *Some t*′
and *t*′: *t* = *t*′ ∨ (*t* ≠ *t*′ ∧ (*f-t*, *t*) ∈ *set ts*) **and** *u*′: *u*′ = (*f*′, *t*′)
and *f*′*-ts*: ∀ *x*∈*set ts*. *disj-fn f*′ (*fst x*)
and [*simp*]: *distinct-prop disj-fn* (*map fst ts*)
by *auto*

from *Cons* **have** *f-t*: *field-ti TYPE*('*a*) *f-t* = *Some t*
by (*auto simp*: *list-all-iff*)

show *?case* **using** *t*′
proof (*elim disjE conjE*)
assume *t-eq*: *t* = *t*′
have *merge-ti t*′ *x* (*update-ti v bs y*) = *merge-ti t*′ *x y*
using *f*′ *t*[*unfolded t-eq*] *bs* **by** (*rule merge-ti-update-ti-sub*)
then show *?thesis*
by (*simp add*: *u*′)
next
assume *ne*: *t* ≠ *t*′ **and** *mem*: (*f-t*, *t*) ∈ *set ts*
with *f*′*-ts* **have** *disj-fn f*′ *f-t* **by** *auto*
then have *disj*: *disj-fn f*′ (*f-t* @ *f*)
by (*rule disj-fn-append-right*)
have *merge-ti t*′ *x* (*update-ti v bs y*) = *update-ti v bs* (*merge-ti t*′ *x y*)
using *f*′ *field-ti-append-field-lookup*[*OF f-t t*] *disj bs* **by** (*rule merge-ti-update-ti-disj*)
then show *?thesis*
by (*simp add*: *Cons mem u*′)
qed
qed *simp*

lemma *heap-update-eq-fold-subfields*:
assumes *ts*: *list-all* (λ(*f*, *u*). *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some u*) *ts*
shows *heap-update p x* =
fold (λ(*f*, *u*). *heap-upd-list* (*size-td u*) &(*p*→*f*) (*access-ti u x*)) *ts* ∘
heap-upd-list (*size-of TYPE*('*a*)) (*ptr-val p*)

```
    (access-ti (typ-info-t TYPE('a)) (merge-ti-list (map snd ts) y x))
proof −
  have snd-ts: list-all (λu. ∃f. field-ti TYPE('a::xmem-type) f = Some u) (map
snd ts)
    using ts unfolding list.pred-map by (rule list.pred-mono-strong) auto
  then interpret merge-ti-list: is-scene merge-ti-list (map snd ts)
    by (rule is-scene-merge-ti-list)

  have heap-upd-list (size-of TYPE('a)) (ptr-val p)
      (access-ti (typ-info-t TYPE('a)) (merge-ti-list (map snd ts) x z)) =
    fold (λ(f, u). heap-upd-list (size-td u) &(p→f) (access-ti u x)) ts ∘
      heap-upd-list (size-of TYPE('a)) (ptr-val p) (access-ti (typ-info-t TYPE('a))
z) for z
    using ts
  proof (induction ts arbitrary: z)
    case (Cons t ts)
    then obtain f u where ts: list-all (λa. case a of (f, u) ⇒ field-ti TYPE('a) f
= Some u) ts
      and f: field-ti TYPE('a) f = Some u and t: t = (f, u)
      by (cases t) auto

    have wf-u: wf-fd u using f by (rule wf-fd-field-ti-mem-type)
    have sz[arith]: size-td u + field-offset TYPE('a) f ≤ size-of TYPE('a)
    using f[THEN field-tiD, THEN field-lookup-offset-size'] by (simp add: size-of-def)

    interpret padding-lense access-ti u update-ti u size-td u
      by (rule field-lookup-padding-lense[OF field-tiD, OF f])
    from field-access-eq-padding1[unfolded eq-padding-def]
    have access-ti-access-ti:
      length bs = size-td u ⟹ access-ti u v (access-ti u w bs) = access-ti u v bs for
v w bs
      by auto

    have heap-upd-list (size-of TYPE('a)) (ptr-val p)
        (access-ti (typ-info-t TYPE('a)) (update-ti u (access-ti0 u x) z)) =
      heap-upd-list (size-td u) &(p→f) (access-ti u x) ∘
      heap-upd-list (size-of TYPE('a)) (ptr-val p) (access-ti (typ-info-t TYPE('a))
z)
      apply (rule ext)
      subgoal for h unfolding heap-upd-list-def
        apply (simp add: field-lvalue-def)
        apply (subst heap-list-update-list)
        subgoal by (simp add: lense.access-result-size)
        subgoal by (simp add: lense.access-result-size)
        apply (subst heap-update-list-super-update-bs)
        subgoal by (simp add: lense.access-result-size length-fa-ti wf-u)
      subgoal using max-size[where 'a='a, arith] by (simp add: lense.access-result-size)
        apply (subst mem-type-field-lookup-access-ti-take-drop[symmetric])
        apply (rule f[THEN field-tiD])
```

```
      apply simp
      apply (subst access-ti-access-ti)
      apply simp
       apply (subst mem-type-access-ti-super-update-bs[symmetric, OF field-tiD,
OF f])
      apply (simp-all add: super-update-bs-take-drop)
      done
    done
  then show ?case
    unfolding fold-Cons t by (simp add: Cons.IH[OF ts] merge-ti-def)
 qed simp
 then show ?thesis
  apply (subst heap-update-eq-heap-upd-list)
  apply (subst merge-ti-list.idem[symmetric])
  apply (subst merge-ti-list.right[symmetric])
  apply (simp del: merge-ti-list.right)
  done
qed

end
```

Chapter 12

More Building Blocks for our C-Language Model

theory *CLanguage*
imports
CProof
Lens
begin

12.1 addr bounds

lemma *addr-card-eq*: *addr-card* = *2^LENGTH*(*addr-bitsize*)
by (*simp add*: *addr-card-def card-word*)

lemma *size-of-bnd*: *size-of TYPE*('*a*::*mem-type*) < *2^LENGTH*(*addr-bitsize*)
by (*rule less-le-trans*[*OF max-size*]) (*simp add*: *addr-card-eq*)

lemma *size-of-mem-type*[*simp*]: *size-of TYPE*('*c*::*mem-type*) ≠ *0*
by *simp*

lemma *addr-card-len-of-conv*: *addr-card* = *2* ^ *len-of TYPE*(*addr-bitsize*)
by (*simp add*: *addr-card*)

lemma *intvl-split*:
⟦ *n* ≥ *a* ⟧ ⟹ { *p* :: ('*a* :: *len*) *word* ..+ *n* } = { *p* ..+ *a* } ∪ { *p* + *of-nat a* ..+ (*n* − *a*)}

apply (*rule set-eqI*, *rule iffI*)
apply (*clarsimp simp*: *intvl-def not-less*)
subgoal for *k*
apply (*rule exI*[**where** *x*=*k*])
apply *clarsimp*
apply (*rule classical*)
apply (*drule-tac x*=*k* − *a* **in** *spec*)

apply (*clarsimp simp*: *not-less*)
apply (*metis diff-less-mono not-less*)
done
subgoal for *x*
apply (*clarsimp simp*: *intvl-def not-less*)
apply (*rule exI*[**where** *x*=*unat* (*x* − *p*)])
apply *clarsimp*
apply (*erule disjE*)
apply *clarsimp*
apply (*metis le-unat-uoi less-or-eq-imp-le not-less order-trans*)
apply *clarsimp*
apply (*metis le-def le-eq-less-or-eq le-unat-uoi less-diff-conv add.commute of-nat-add*)
done
done

12.2 More Heap Typing

primrec
htd-upd :: *addr* ⇒ *typ-slice list* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc*
where
htd-upd p [] *d* = *d*
| *htd-upd p* (*x*#*xs*) *d* = *htd-upd* (*p*+*1*) *xs* (*d*(*p* := (*True*, *x*)))

definition (**in** *c-type*) *ptr-force-type* :: *'a ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc*
where
ptr-force-type p ≡ *htd-upd* (*ptr-val p*) (*typ-slices TYPE*('*a*))

definition *ptr-force-types* :: *'a*::*c-type ptr list* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc*
where
ptr-force-types = *fold ptr-force-type*

definition *ptr-force-free* :: *addr* ⇒ *nat* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
ptr-force-free p b = *ptr-force-types* (*map* (λ*n*. *PTR*(*8 word*) *p* +*p n*) (*map of-nat* [*0*..<*2*^*b*]))

definition *ptr-u* :: *'a*::*c-type ptr* ⇒ (*addr* × *typ-uinfo*) **where**
ptr-u p = (*ptr-val p*, *typ-uinfo-t TYPE*('*a*))

abbreviation *ptr-span-u* ≡ (λ(*a*, *t*). {*a* ..+ *size-td t*})

definition *typ-slices-u* :: *typ-uinfo* ⇒ *typ-slice list* **where**
typ-slices-u t = *map* (λ*n*. *list-map* (*typ-slice-t t n*)) [*0*..<*size-td t*]

definition *ptr-force-type-u* :: *typ-uinfo* ⇒ *addr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc*
where
ptr-force-type-u t a ≡ *htd-upd a* (*typ-slices-u t*)

lemma *heap-update-list-id*: *heap-update-list x* [] = (λ*x*. *x*)
by *auto*

lemma *to-bytes-word8*:
to-bytes (*v* :: *word8*) *xs* = [*v*]
by (*simp add*: *to-bytes-def typ-info-word word-rsplit-same*)

lemma *heap-update-heap-update-list*:
⟦ *ptr-val p* = *q* + (*of-nat* (*length l*)); *Suc* (*length l*) < *addr-card* ⟧ ⟹
heap-update (*p* :: *word8 ptr*) *v* (*heap-update-list q l s*) = (*heap-update-list q* (*l* @ [*v*]) *s*)
by (*metis to-bytes-word8 heap-update-def heap-update-list-concat-fold*)

lemma *htd-upd-empty*[*simp*]: *htd-upd p* [] = *id*
by (*simp add*: *fun-eq-iff*)

lemma *htd-upd-append*:
htd-upd p (*xs* @ *ys*) = *htd-upd* (*p* + *of-nat* (*length xs*)) *ys* ∘ *htd-upd p xs*
by (*induction xs arbitrary*: *p*) (*simp-all add*: *fun-eq-iff ac-simps*)

lemma *htd-upd-singleton*[*simp*]: *htd-upd p* [*x*] = *upd-fun p* (λ*h*. (*True*, *x*))
by (*simp add*: *fun-eq-iff upd-fun-def*)

lemma *intvl-Suc-eq*: {*p* ..+ *Suc n*} = *insert p* {*p* + *1* ..+ *n*}
using *intvl-split*[*of 1 Suc n p*] **by** (*auto simp add*: *intvl-def*)

lemma *htd-upd-disj*: *p* ∉ {*p'* ..+ *length v*} ⟹ *htd-upd p' v h p* = *h p*
by (*induction v arbitrary*: *p' h*)
(*auto simp add*: *intvl-Suc-eq fun-upd-other simp del*: *fun-upd-apply*)

lemma *htd-upd-head*:
xs ≠ [] ⟹ *length xs* ≤ *addr-card* ⟹ *htd-upd p xs s p* = (*True*, *hd xs*)
using *intvl-Suc-nmem''*[*of length xs p*] *htd-upd-disj*[*of p p* + *1 tl xs* -]
by (*cases xs*) (*simp-all add*: *addr-card-eq del*: *intvl-Suc-nmem'*)

lemma *htd-upd-at*:
i < *length xs* ⟹ *length xs* ≤ *addr-card* ⟹ *htd-upd p xs s* (*p* + *of-nat i*) = (*True*, *xs* ! *i*)
proof (*induction i arbitrary*: *p xs s*)
case *0* **with** *htd-upd-head*[*of xs p s*] **show** *?case* **by** (*simp add*: *hd-conv-nth*)
next
case (*Suc n*)
from *Suc.prems* **show** *?case* **by** (*cases xs*) (*simp-all add*: *Suc.IH flip*: *add.assoc*)
qed

lemma *ptr-force-type-disj*:
p ∉ *ptr-span* (*p'* :: 'a::*mem-type ptr*) ⟹ *ptr-force-type p' h p* = *h p*
unfolding *ptr-force-type-def*
by (*intro htd-upd-disj*) *simp-all*

lemma *ptr-force-types-disj*:
fixes *xs* :: *'a*::*mem-type ptr list*
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow i \notin ptr\text{-}span\ x$
shows *ptr-force-types xs h i* = *h i*
by (*use assms* **in** ‹*induction xs rule*: *rev-induct*›)
(*auto simp*: *ptr-force-types-def ptr-force-type-disj*)

12.2.1 Heap type tag and valid simple footprint

datatype *heap-typ-contents* =
HeapType typ-uinfo
| *HeapFootprint*
| *HeapEmpty*

definition
heap-type-tag :: *heap-typ-desc* ⇒ *addr* ⇒ *heap-typ-contents*
where
heap-type-tag d a ≡
(*if fst* (*d a*) = *False* ∨ (∀ *x*. (*snd* (*d a*)) *x* = *None*) ∨ (∀ *x*. (*snd* (*d a*)) *x* ≠ *None*) *then*
HeapEmpty
else
case (*snd* (*d a*)) (*GREATEST x*. *snd* (*d a*) *x* ≠ *None*) *of*
Some (-, *False*) ⇒ *HeapFootprint*
| *Some* (*x*, *True*) ⇒ *HeapType x*
| *None* ⇒ *HeapEmpty*)

definition
valid-simple-footprint :: *heap-typ-desc* ⇒ *addr* ⇒ *typ-uinfo* ⇒ *bool*
where
valid-simple-footprint d x t ≡
heap-type-tag d x = *HeapType t* ∧
(∀ *y*. *y* ∈ {*x* + *1*..+ (*size-td t*)− *Suc 0*} ⟶ *heap-type-tag d y* = *HeapFootprint*)

lemma *valid-simple-footprint-size-td*:
assumes *valid*: *valid-simple-footprint d x t*
shows *size-td t* ≤ *addr-card*
proof (*cases size-td t* ≤ *addr-card*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *valid* **have** *heap-type-tag d x* = *HeapType t*
by (*simp add*: *valid-simple-footprint-def*)

moreover
from *False* **have** $x \in \{x + 1..+ \ (size\text{-}td\ t) - Suc\ 0\}$
apply (*clarsimp simp add*: *intvl-def*)
apply (*rule exI* [**where** $x = addr\text{-}card - Suc\ 0$])
by (*metis* (*mono-tags*, *opaque-lifting*) *One-nat-def Suc-pred'* *diff-less-mono* *diff-zero*
neq0-conv not-less not-less-eq-eq of-nat-1 of-nat-Suc of-nat-addr-card unatSuc

unat-minus-abs zero-diff zero-neq-one)
with *valid* **have** *heap-type-tag d x = HeapFootprint* **by** (*auto simp add*: *valid-simple-footprint-def*)
ultimately have *False* **by** *simp*
thus *?thesis* **..**
qed

lemma *valid-simple-footprintI*:
$[\![$ *heap-type-tag d x = HeapType t*; $\bigwedge y.\ y \in \{x + 1..+(size\text{-}td\ t) - Suc\ 0\} \Longrightarrow$ *heap-type-tag d y = HeapFootprint* $]\!]$
$\Longrightarrow$ *valid-simple-footprint d x t*
by (*clarsimp simp*: *valid-simple-footprint-def*)

lemma *valid-simple-footprintD*:
valid-simple-footprint d x t $\Longrightarrow$ *heap-type-tag d x = HeapType t*
by (*simp add*: *valid-simple-footprint-def*)

lemma *valid-simple-footprintD2*:
$[\![$ *valid-simple-footprint d x t*; $y \in \{x + 1..+(size\text{-}td\ t) - Suc\ 0\}$ $]\!] \Longrightarrow$ *heap-type-tag d y = HeapFootprint*
by (*simp add*: *valid-simple-footprint-def*)

lemma *typ-slices-not-empty*:
typ-slices (x::('a::{*mem-type*} *itself*)) $\neq$ []
apply (*clarsimp simp*: *typ-slices-def*)
done

lemma *last-typ-slice-t*:
(*last* (*typ-slice-t t 0*)) = (*t*, *True*)
apply (*cases t*)
apply *clarsimp*
done

lemma *last-typ-slice-t-non-zero*:
$k \neq 0 \Longrightarrow$ (*last* (*typ-slice-t t k*)) = (*t*, *False*)
apply (*cases t*)
apply *clarsimp*
done

lemma *if-eqI*:
$[\![\ a \Longrightarrow x = z;\ \neg\ a \Longrightarrow y = z\ ]\!] \Longrightarrow$ (*if a then x else y*) = *z*
by *simp*

lemma *heap-type-tag-ptr-retyp*:
snd (*s* (*ptr-val t*)) = *Map.empty* ⟹
heap-type-tag (*ptr-retyp* (*t* :: 'a::*mem-type ptr*) *s*) (*ptr-val t*) = *HeapType* (*typ-uinfo-t TYPE*('a))
apply (*unfold ptr-retyp-def heap-type-tag-def*)
apply (*subst htd-update-list-index*, *fastforce*, *fastforce*)+
apply (*rule if-eqI*)
apply *clarsimp*
apply (*erule disjE*)
apply (*erule-tac x=0* **in** *allE*)
apply *clarsimp*
apply (*erule-tac x=length* (*typ-slice-t* (*typ-uinfo-t TYPE*('a)) *0*) **in** *allE*)
apply (*clarsimp simp*: *list-map-eq*)
apply (*clarsimp simp*: *list-map-eq last-conv-nth* [*simplified*, *symmetric*] *last-typ-slice-t split*: *option.splits if-split-asm prod.splits*)
done

lemma *not-snd-last-typ-slice-t*:
k ≠ *0* ⟹ ¬ *snd* (*last* (*typ-slice-t z k*))
by (*cases z*, *clarsimp*)

lemma *heap-type-tag-ptr-retyp-rest*:
⟦ *snd* (*s* (*ptr-val t* + *k*)) = *Map.empty*; *0* < *k*; *unat k* < *size-td* (*typ-uinfo-t TYPE*('a)) ⟧ ⟹
heap-type-tag (*ptr-retyp* (*t* :: 'a::*mem-type ptr*) *s*) (*ptr-val t* + *k*) = *Heap-Footprint*
apply (*unfold ptr-retyp-def heap-type-tag-def*)
apply (*subst htd-update-list-index*, *simp*, *clarsimp*,
metis intvlI size-of-def word-unat.Rep-inverse)+
apply (*rule if-eqI*)
apply *clarsimp*
apply (*erule disjE*)
apply (*erule-tac x=0* **in** *allE*)
apply (*clarsimp simp*: *size-of-def*)
apply (*erule-tac x=length* (*typ-slice-t* (*typ-uinfo-t TYPE*('a)) (*unat k*)) **in** *allE*)
apply (*clarsimp simp*: *size-of-def list-map-eq*)
apply (*clarsimp simp*: *list-map-eq last-conv-nth* [*simplified*, *symmetric*] *size-of-def split*: *option.splits if-split-asm prod.splits bool.splits*)
apply (*metis surj-pair*)
apply (*subst* (*asm*) (*2*) *surjective-pairing*)
apply (*subst* (*asm*) *not-snd-last-typ-slice-t*)
apply *clarsimp*
apply *unat-arith*
apply *simp*
done

lemma *typ-slices-addr-card* [*simp*]:
length (*typ-slices* (*x*::('a::{*mem-type*} *itself*))) < *addr-card*

apply (*clarsimp simp*: *typ-slices-def*)
done

lemma *unat-less-impl-less*:
unat a < *unat b* ⟹ *a* < *b*
by *unat-arith*

lemma *valid-simple-footprint-ptr-retyp*:
⟦ ∀ *k* < *size-td* (*typ-uinfo-t TYPE*('*a*)). *snd* (*s* (*ptr-val t* + *of-nat k*)) = *Map.empty*;
1 ≤ *size-td* (*typ-uinfo-t TYPE*('*a*));
size-td (*typ-uinfo-t TYPE*('*a*)) < *addr-card* ⟧
⟹ *valid-simple-footprint* (*ptr-retyp* (*t* :: '*a*::*mem-type ptr*) *s*) (*ptr-val t*) (*typ-uinfo-t TYPE*('*a*))
apply (*clarsimp simp*: *valid-simple-footprint-def*)
apply (*rule conjI*)
apply (*subst heap-type-tag-ptr-retyp*)
apply (*erule allE* [**where** *x*=*0*])
apply *clarsimp*
apply *clarsimp*
apply (*clarsimp simp*: *intvl-def*)
subgoal for *k*
apply (*erule-tac x*=*k* + *1* **in** *allE*)
apply (*erule impE*)
apply (*metis One-nat-def less-diff-conv*)
apply (*subst add.assoc*, *subst heap-type-tag-ptr-retyp-rest*)
apply *clarsimp*
apply (*cases 1* + *of-nat k* = (*0* :: *addr*))
apply (*metis add.left-neutral intvlI intvl-Suc-nmem size-of-def*)
apply *unat-arith*
apply *clarsimp*
apply (*metis lt-size-of-unat-simps size-of-def Suc-eq-plus1 One-nat-def less-diff-conv of-nat-Suc*)
apply *simp*
done
done

lemma *heap-type-tag-cong*: *s p* = *s*′ *p* ⟹ *heap-type-tag s p* = *heap-type-tag s*′ *p*
by (*simp add*: *heap-type-tag-def cong*: *if-cong*)

lemma *heap-type-tag*:
assumes *eq*: *h p* = (*f*, *list-map l*)
shows *heap-type-tag h p* =
(*if* ¬ *f* ∨ *l* = [] *then HeapEmpty else*
case last l of (*x*, *b*) ⇒ *if b then HeapType x else HeapFootprint*)
unfolding *heap-type-tag-def eq*
by (*auto simp add*: *list-map-def*)
(*auto simp*: *split-beta*′ *last-conv-nth list-map-eq simp flip*: *list-map-def*)

lemma *valid-simple-footprint-cong-state*:
assumes *t*: *wf-size-desc t*
assumes *eq*: $\bigwedge p'.\ p' \in \{p\ ..+ size\text{-}td\ t\} \Longrightarrow s\ p' = s'\ p'$
shows *valid-simple-footprint s p t* $\longleftrightarrow$ *valid-simple-footprint s' p t*
unfolding *valid-simple-footprint-def*
using *eq wf-size-desc-gt*(*1*)[*OF t*]
using *intvl-split*[*of 1 size-td t p*]
by (*intro arg-cong2*[**where** *f*=$\wedge$] *all-cong arg-cong*[**where** *f*=$\lambda x.\ x$ = -] *heap-type-tag-cong*)
(*auto simp*: *intvl-def*)

lemma *heap-type-tag-ptr-force-type-HeapType*:
fixes *x* :: *'a*::*mem-type ptr*
shows *heap-type-tag* (*ptr-force-type x s*) (*ptr-val x*) = *HeapType* (*typ-uinfo-t TYPE*(*'a*))
by (*subst heap-type-tag*)
(*auto simp*: *ptr-force-type-def htd-upd-head typ-slices-not-empty max-size*[*THEN less-imp-le*]
hd-conv-nth last-typ-slice-t)

lemma *heap-type-tag-ptr-force-type-HeapFootprint*:
fixes *p* :: *'a*::*mem-type ptr*
shows *p'* $\in$ {*ptr-val p + 1 ..+ size-of TYPE*(*'a*) − *Suc 0*} $\Longrightarrow$
heap-type-tag (*ptr-force-type p s*) *p'* = *HeapFootprint*
unfolding *intvl-def*
apply (*clarsimp simp*: *less-diff-conv add.assoc ptr-force-type-def simp flip*: *of-nat-Suc*)
subgoal premises *prems* **for** *k*
using *prems*(*1*)
apply (*subst heap-type-tag*)
apply (*subst htd-upd-at*)
apply (*simp-all add*: *max-size*[*THEN less-imp-le*])
apply (*simp add*: *last-typ-slice-t split-beta' not-snd-last-typ-slice-t*)
done
done

lemma *valid-simple-footprint-ptr-force-type-iff*:
fixes *p* :: *'a*::*mem-type ptr*
assumes *t*: *wf-size-desc t*
shows *valid-simple-footprint* (*ptr-force-type p s*) *a t* $\longleftrightarrow$
(*valid-simple-footprint s a t* $\wedge$ *disjnt* {*a ..+ size-td t*} (*ptr-span p*)) $\vee$
(*t* = *typ-uinfo-t TYPE*(*'a*) $\wedge$ *p* = *Ptr a*)
proof *cases*
assume *disjnt*: *disjnt* {*a ..+ size-td t*} (*ptr-span p*)
moreover have *p* $\neq$ *Ptr a*
using *disjnt*[*unfolded disjnt-iff*, *THEN spec*, *of a*] *t*[*THEN wf-size-desc-gt*(*1*)]
by (*cases p*) (*auto simp*: *intvl-self*)
moreover have *valid-simple-footprint* (*ptr-force-type p s*) *a t* = *valid-simple-footprint s a t*
using *t disjnt*
by (*intro valid-simple-footprint-cong-state ptr-force-type-disj*) (*simp-all add*:

disjnt-iff)
ultimately show *?thesis*
by *simp*
next
assume *ndisjnt*: ¬ *disjnt* {*a* ..+ *size-td t*} (*ptr-span p*)
from *intvl-inter*[*OF this*[*unfolded disjnt-def*]]
consider *a* = *ptr-val p*
| *a* ∈ {*ptr-val p* + *1* ..+ *size-of TYPE*('*a*) − *1*}
| *ptr-val p* ∈ {*a* + *1* ..+ *size-td t* − *Suc 0*} *ptr-val p* ≠ *a*
by (*auto dest*: *intvl-neq-start*)
then show *?thesis*
proof *cases*
case *1*
have *valid-simple-footprint* (*ptr-force-type p s*) (*ptr-val p*) (*typ-uinfo-t TYPE*('*a*))
by (*auto simp*: *valid-simple-footprint-def heap-type-tag-ptr-force-type-HeapType*
heap-type-tag-ptr-force-type-HeapFootprint size-of-tag)
with *1 valid-simple-footprintD*[*of ptr-force-type p s a t*] *ndisjnt*
show *?thesis*
by (*auto simp*: *heap-type-tag-ptr-force-type-HeapType*)
next
case *2* **with** *valid-simple-footprintD*[*of ptr-force-type p s a t*] **show** *?thesis*
by (*auto simp*: *heap-type-tag-ptr-force-type-HeapFootprint ndisjnt*)
next
case *3* **with** *valid-simple-footprintD2*[*of ptr-force-type p s a t ptr-val p*] **show**
?thesis
by (*auto simp add*: *heap-type-tag-ptr-force-type-HeapType ndisjnt*)
qed
qed

lemma *valid-simple-footprint-fold-ptr-force-type-iff*:
fixes *ps* :: '*a*::*mem-type ptr list*
assumes [*simp*]: *wf-size-desc t*
shows *distinct-prop* (λ*p1 p2*. *disjnt* (*ptr-span p1*) (*ptr-span p2*)) *ps* ⟹
valid-simple-footprint (*fold ptr-force-type ps s*) *a t* ⟷
(*valid-simple-footprint s a t* ∧ *disjnt* {*a* ..+ *size-td t*} (⋃*p*∈*set ps*. *ptr-span*
p)) ∨
(*t* = *typ-uinfo-t TYPE*('*a*) ∧ *Ptr a* ∈ *set ps*)
by (*induction ps arbitrary*: *s*)
(*auto simp*: *valid-simple-footprint-ptr-force-type-iff size-of-tag distinct-prop-append*)

12.3 Pointers to local (stack) variables

definition *stack-typ-info t* = (*stack-byte-name* ∉ *td-names t*)

lemma *stack-typ-info-export-uinfo*[*simp*]: *stack-typ-info* (*export-uinfo t*) = *stack-typ-info*
t
by (*simp add*: *stack-typ-info-def*)

lemma *stack-typ-info-td-set*:

assumes *stack-typ*: *stack-typ-info t*
assumes *t'*: $(t', n) \in$ *td-set t 0*
shows *typ-name t'* $\neq$ *stack-byte-name*
proof (*cases typ-name t'* = *pad-typ-name*)
case *True*
then show *?thesis* **by** (*simp add*: *stack-byte-name-def pad-typ-name-def*)
next
case *False*
from *td-set-td-names* [*OF t' False*]
have *typ-name t'* $\in$ *td-names t* **.**
with *stack-typ* **show** *?thesis*
by (*auto simp add*: *stack-typ-info-def*)
qed

lemma *stack-typ-info-td-set-stack-byte*:
assumes *stack-typ*: *stack-typ-info t*
assumes *t'*: $(t', n) \in$ *td-set t 0*
shows *t'* $\neq$ *typ-uinfo-t TYPE*(*stack-byte*)
using *stack-typ-info-td-set* [*OF stack-typ t'*]
apply (*cases t'*)
apply (*simp add*: *typ-uinfo-t-def typ-info-stack-byte*)
done

class *stack-type* = *c-type* +
assumes *stack-typ-info*: *stack-typ-info* (*typ-info-t TYPE*('*a*))
begin
lemma *stack-typ-uinfo*: *stack-typ-info* (*typ-uinfo-t TYPE*('*a*))
using *stack-typ-info*
by (*simp add*: *typ-uinfo-t-def*)

lemma *no-stack-byte* [*simp*]: *typ-uinfo-t TYPE*('*a*) $\neq$ *typ-uinfo-t TYPE*(*stack-byte*)
by (*metis order.refl stack-typ-info-td-set-stack-byte stack-typ-uinfo typ-tag-le-def*)

end

lemma *stack-typ-info-no-stack-byte*:
stack-typ-info t $\Longrightarrow$ *t* $\neq$ *typ-uinfo-t TYPE*(*stack-byte*)
using *stack-typ-info-def typ-tag-le-def stack-typ-info-td-set-stack-byte* **by** *fastforce*

lemma *stack-typ-info-empty-typ-info*:
nm $\neq$ *stack-byte-name* $\Longrightarrow$ *stack-typ-info* (*empty-typ-info algn nm*)
by (*simp add*: *empty-typ-info-def stack-typ-info-def typ-info-stack-byte*)

lemma *td-set-list-to-set*:
$(t, m) \in$ *td-set-list xs n* $\Longrightarrow$ $(\exists\, x\ k.\ x \in$ *set xs* $\wedge\ (t, k) \in$ *td-set* (*dt-fst x*) *0*)
apply (*induct xs arbitrary*: *m n*)

apply *simp*
by (*metis Un-iff dt-tuple.collapse insertCI list.set*(*2*) *td-set-list.simps*(*2*)
td-set-offset-0-conv'' ts5)

lemma *td-names-list-to-set*:
nm ∈ *td-names-list xs* ⟹ (∃ *x*. *x* ∈ *set xs* ∧ *nm* ∈ *td-names* (*dt-fst x*))
apply (*induct xs arbitrary*:)
apply *simp*
by (*metis Un-iff dt-tuple.collapse insert-iff list.set*(*2*) *td-names-list.simps*(*2*) *tnm5*)

lemma *set-to-td-names-list*:
x ∈ *set xs* ⟹ *nm* ∈ *td-names* (*dt-fst x*) ⟹ *nm* ∈ *td-names-list xs*
apply (*induct xs arbitrary*:)
apply *simp*
by (*metis Un-iff dt-tuple.collapse insert-iff list.set*(*2*) *td-names-list.simps*(*2*) *tnm5*)

lemma *stack-typ-info-TypAggregateI*:
assumes *xs*: ⋀*x*. *x* ∈ *set xs* ⟹ *stack-typ-info* (*dt-fst x*)
assumes *nm*: *nm* ≠ *stack-byte-name*
shows *stack-typ-info* (*TypDesc algn* (*TypAggregate* (*xs*)) *nm*)
using *xs nm*
apply (*auto simp add*: *stack-typ-info-def stack-byte-name-def dest*: *td-names-list-to-set*)
done

lemma *TypAggregate-not-stack-byte*:
TypDesc algn (*TypAggregate xs*) *nm* ≠ *typ-uinfo-t TYPE*(*stack-byte*)
by (*auto simp add*: *typ-info-stack-byte typ-uinfo-t-def*)

lemma *stack-typ-info-TypAggregateD*:
assumes *aggr*: *stack-typ-info* (*TypDesc algn* (*TypAggregate* (*xs*)) *nm*)
assumes *x*: *x* ∈ *set xs*
shows *stack-typ-info* (*dt-fst x*)
using *aggr x*
by (*auto simp add*: *stack-typ-info-def stack-byte-name-def dest*:*set-to-td-names-list*)

lemma *stack-typ-info-extend-ti*:
stack-typ-info s ⟹ *stack-typ-info t* ⟹
stack-typ-info (*extend-ti s t algn fn d*)
apply (*cases s*)
by (*simp add*: *stack-typ-info-def*)

lemma *stack-typ-info-ti-pad-combine*:
stack-typ-info t ⟹ *stack-typ-info* (*ti-pad-combine n t*)
apply (*simp add*: *ti-pad-combine-def Let-def*)
apply (*rule stack-typ-info-extend-ti*)
apply *assumption*
apply (*auto simp add*: *stack-typ-info-def typ-uinfo-t-def typ-info-stack-byte*)

done

lemma *stack-typ-info-export-uinfo-adjust-ti'*:
shows *stack-typ-info* (*adjust-ti* (*typ-info-t TYPE*('*b*::*stack-type*)) *acc upd*)
proof −
from *stack-typ-info*
have *stack-typ-info* (*typ-info-t TYPE*('*b*)).
then show *?thesis*
by (*simp add*: *stack-typ-info-def*)
qed

lemma *stack-typ-info-export-uinfo-adjust-ti*:
shows *stack-typ-info* (*typ-info-t* (*TYPE*('*b*))) ⟹ *stack-typ-info* (*adjust-ti* (*typ-info-t TYPE*('*b*::*c-type*)) *acc upd*)
by (*simp add*: *stack-typ-info-def*)

lemma *stack-typ-info-ti-typ-combine'*:
stack-typ-info t ⟹
stack-typ-info (*ti-typ-combine TYPE*('*b*::*stack-type*) *acc upd algn nm t*)
apply (*simp add*: *ti-typ-combine-def*)
apply (*rule stack-typ-info-extend-ti*)
apply *assumption*
apply (*rule stack-typ-info-export-uinfo-adjust-ti'*)
done

lemma *stack-typ-info-ti-typ-combine*:
stack-typ-info (*typ-info-t* (*TYPE*('*b*))) ⟹ *stack-typ-info t* ⟹
stack-typ-info (*ti-typ-combine TYPE*('*b*::*c-type*) *acc upd algn nm t*)
apply (*simp add*: *ti-typ-combine-def*)
apply (*rule stack-typ-info-extend-ti*)
apply *assumption*
apply (*simp add*: *stack-typ-info-export-uinfo-adjust-ti*)
done

lemma *stack-typ-info-ti-typ-pad-combine'*:
stack-typ-info t ⟹
stack-typ-info (*ti-typ-pad-combine TYPE*('*b*::*stack-type*) *acc upd algn nm t*)
by (*auto simp add*: *ti-typ-pad-combine-def Let-def stack-typ-info-ti-typ-combine' stack-typ-info-ti-pad-combine*)

lemma *stack-typ-info-ti-typ-pad-combine*:
stack-typ-info (*typ-info-t* (*TYPE*('*b*))) ⟹ *stack-typ-info t* ⟹
stack-typ-info (*ti-typ-pad-combine TYPE*('*b*::*c-type*) *acc upd algn nm t*)
by (*auto simp add*: *ti-typ-pad-combine-def Let-def stack-typ-info-ti-typ-combine stack-typ-info-ti-pad-combine*)

lemma *stack-typ-info-map-align*:
stack-typ-info t ⟹ *stack-typ-info* (*map-align algn t*)

by (*cases t*) (*auto simp add*: *stack-typ-info-def*)

lemma *stack-typ-info-final-pad*: *stack-typ-info t* $\Longrightarrow$
stack-typ-info (*final-pad algn t*)
by (*auto simp add*: *final-pad-def Let-def stack-typ-info-ti-pad-combine stack-typ-info-map-align*)

lemmas *stack-typ-info-intros* =
stack-typ-info-empty-typ-info
stack-typ-info-ti-typ-pad-combine
stack-typ-info-final-pad
stack-typ-info

named-theorems *stack-typ-info*

definition *valid-root-footprint* :: *heap-typ-desc* $\Rightarrow$ *addr* $\Rightarrow$ *typ-uinfo* $\Rightarrow$ *bool* **where**
valid-root-footprint d x t $\equiv$
let n = *size-td t in*
0 < *n* $\wedge$ ($\forall$ *y*. *y* < *n* $\longrightarrow$
snd (*d* (*x* + *of-nat y*)) = *list-map* (*typ-slice-t t y*) $\wedge$ *fst* (*d* (*x* + *of-nat y*)))

lemma *valid-root-footprint-valid-footprint*: *valid-root-footprint d x t* $\Longrightarrow$ *valid-footprint d x t*
by (*auto simp add*: *valid-root-footprint-def valid-footprint-def Let-def*)

lemma *valid-root-footprint-valid-footprint-dom-conv*:
valid-root-footprint d a t
$\longleftrightarrow$
(*valid-footprint d a t* $\wedge$
($\forall$ *n*. *n* < *size-td t* $\longrightarrow$ *dom* (*snd* (*d* (*a* + *of-nat n*))) = {*0*..<*length* (*typ-slice-t t n*)}))
apply (*clarsimp simp add*: *valid-footprint-def valid-root-footprint-def Let-def*)
apply (*intro iffI conjI*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*metis list-map-eq not-None-eq dom-list-map map-le-same-dom-eq*)
by (*metis dom-list-map map-le-same-dom-eq*)

lemma *valid-root-footprint-dom-typing*:
valid-root-footprint d a t $\Longrightarrow$ *n* < *size-td t* $\Longrightarrow$
dom (*snd* (*d* (*a* + *of-nat n*))) = {*0*..<*length* (*typ-slice-t t n*)}
by (*simp add*: *valid-root-footprint-valid-footprint-dom-conv*)

lemma *valid-root-footprint-typing-conv*:
fixes *p*::*'a*::*c-type ptr*
assumes *valid*: *valid-root-footprint d* (*ptr-val p*) (*typ-uinfo-t TYPE*(*'a*))
assumes *n*: *n* < *size-of* (*TYPE*(*'a*))

shows *d* (*ptr-val p* + *of-nat n*) = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *n*))
using *valid n*
by (*simp add*: *valid-root-footprint-def Let-def size-of-def prod-eq-iff*)

definition
root-ptr-valid :: *heap-typ-desc* ⇒ '*a*::*c-type ptr* ⇒ *bool*
where
root-ptr-valid d p ≡
valid-root-footprint d (*ptr-val* (*p*::'*a ptr*)) (*typ-uinfo-t TYPE*('*a*)) ∧
c-guard p

lemma *root-ptr-valid-c-guard*: *root-ptr-valid d p* ⟹ *c-guard p*
by (*simp add*: *root-ptr-valid-def*)

lemma *root-ptr-valid-typing-conv*:
fixes *p*::'*a*::*c-type ptr*
assumes *valid*: *root-ptr-valid d p*
assumes *n*: *n* < *size-of* (*TYPE*('*a*))
shows *d* (*ptr-val p* + *of-nat n*) = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *n*))
using *valid*
by (*simp add*: *root-ptr-valid-def valid-root-footprint-typing-conv* [*OF* - *n*])

lemma *root-ptr-valid-h-t-valid*: *root-ptr-valid d p* ⟹ *d*, *c-guard* ⊨$_t$ *p*
by (*simp add*: *h-t-valid-def root-ptr-valid-def valid-root-footprint-valid-footprint-dom-conv*)

lemma *valid-root-footprint-cong-state*:
assumes *t*: *wf-size-desc t*
assumes *eq*: ⋀*p'*. *p'* ∈ {*p* ..+*size-td t*} ⟹ *s p'* = *s' p'*
shows *valid-root-footprint s p t* ⟷ *valid-root-footprint s' p t*
unfolding *valid-root-footprint-def Let-def*
using *eq wf-size-desc-gt*(*1*)[*OF t*]
using *intvl-split*[*of 1 size-td t p*]
by (*metis intvlI*)

lemma (**in** *mem-type*) *valid-root-foot-print-ptr-force-type*:
valid-root-footprint
(*ptr-force-type* (*p*::'*a ptr*) *s*) (*ptr-val p*) (*typ-uinfo-t* (*TYPE*('*a*)))
by (*simp add*: *htd-upd-at local.ptr-force-type-def*
local.size-of-fold order-less-imp-le valid-root-footprint-def)

lemma *list-map-greatest-last*: *xs* ≠ [] ⟹ *last xs* = *v* ⟹ *list-map xs* (*GREATEST k*. ∃ *v*. *list-map xs k* = *Some v*) = *Some v*
proof (*induct n* == *length xs arbitrary*: *xs*)
case *0*
then show *?case* **by** *simp*
next

```
  case (Suc n xs)
  show ?case
  proof (cases xs)
    case Nil
    then show ?thesis using Suc by simp
  next
    case (Cons x xs')
    have xs: xs = x#xs' by fact
    show ?thesis
    proof (cases xs')
      case Nil

      have *: (GREATEST k::nat. ∃ v'. [0 ↦ v] k = Some v') = 0
        by (rule Greatest-equality) (auto simp add: fun-upd-def split: if-split-asm)

      from Nil show ?thesis using Suc.prems xs
        by (simp add: list-map-def * fun-upd-def Greatest-equality)
    next
      case (Cons x' xs'')
       from Suc xs Cons obtain n: n = length xs' and non-empty: xs' ≠ [] and
last: last xs' = v
        by simp

      from Suc.hyps (1)  [OF n non-empty last]
     have hyp: list-map xs' (GREATEST x. ∃ v. list-map xs' x = Some v) = Some
v .
       from hyp show ?thesis by (simp add: xs Cons add: list-map-eq split:
if-split-asm)
    qed
  qed
qed


lemma valid-root-footprint-valid-simple-footprint:
  assumes valid-td: size-td t ≤ addr-card
  shows valid-root-footprint d x t ⟹ valid-simple-footprint d x t
 apply (clarsimp simp add: valid-root-footprint-def valid-simple-footprint-def Let-def)
proof −
  assume sz: 0 < size-td t
  assume root[rule-format]: ∀ y < size-td t. snd (d (x + word-of-nat y)) = list-map
(typ-slice-t t y) ∧ fst (d (x + word-of-nat y))
  from root [of 0, simplified] sz obtain
    d-x: d x = (True, list-map (typ-slice-t t 0))
    by (cases d x) auto

  have x-HeapType: heap-type-tag d x = HeapType t
  proof −
    from last-typ-slice-t have last (typ-slice-t t 0) = (t, True) .
```

from *list-map-greatest-last* [*OF typ-slice-t-not-empty this*]
have ∗: *list-map* (*typ-slice-t t 0*) (*GREATEST k.* ∃ *a b. list-map* (*typ-slice-t t 0*) *k* = *Some* (*a*,*b*)) = *Some* (*t*, *True*)
by *auto*
from *d-x* **show** *?thesis*
apply (*clarsimp simp add*: *heap-type-tag-def* ∗, *intro conjI*)
using ∗ **apply** *blast*
by (*metis less-irrefl list-map-eq option.simps*(*3*))
qed

show *heap-type-tag d x* = *HeapType t* ∧ (∀ *y. y* ∈ {*x* + *1*..+*size-td t* − *Suc 0*} ⟶ *heap-type-tag d y* = *HeapFootprint*)
proof
from *x-HeapType*
show *heap-type-tag d x* = *HeapType t* **.**
next
show ∀ *y. y* ∈ {*x* + *1*..+*size-td t* − *Suc 0*} ⟶ *heap-type-tag d y* = *HeapFootprint*
proof (*standard*+)
fix *y*
assume *y-bounds*: *y* ∈ {*x* + *1*..+*size-td t* − *Suc 0*}
with *sz* **obtain** *off* **where** *y-off*: *y* = *x* + *word-of-nat off* **and** *off*: *off* < *size-td t*
by (*meson intvlD intvl-plus-sub-Suc*)
from *root* [*OF off*] *y-off* **have** *d-y*: *d y* = (*True*, *list-map* (*typ-slice-t t off*))
by (*cases d* (*x* + *word-of-nat off*)) *auto*

have *x* ∉ {*x* + *1*..+*size-td t* − *Suc 0*}
using *intvl-Suc-nmem''* [*OF valid-td* [*simplified addr-card-len-of-conv*]] **by** *blast*
with *y-bounds y-off* **have** *off-non-zero*: *off* ≠ *0* **by** (*cases off*) *auto*

from *last-typ-slice-t-non-zero* [*OF off-non-zero*] **have** *last* (*typ-slice-t t off*) = (*t*, *False*) **.**
from *list-map-greatest-last* [*OF typ-slice-t-not-empty this*]
have ∗: *list-map* (*typ-slice-t t off*) (*GREATEST k.* ∃ *a b. list-map* (*typ-slice-t t off*) *k* = *Some* (*a*,*b*)) =
Some (*t*, *False*)
by *auto*
from *d-y* **show** *heap-type-tag d y* = *HeapFootprint*
apply (*clarsimp simp add*: *heap-type-tag-def* ∗, *intro conjI*)
using ∗ **apply** *blast*
by (*metis less-irrefl list-map-eq option.simps*(*3*))
qed
qed
qed

lemma *valid-root-footprint-valid-simple-footprint-typ-uinfo-t*:

assumes *valid-root*: *valid-root-footprint d x* (*typ-uinfo-t TYPE*('*a*::*mem-type*))
shows *valid-simple-footprint d x* (*typ-uinfo-t TYPE*('*a*::*mem-type*))
apply (*rule valid-root-footprint-valid-simple-footprint* [*OF* - *valid-root*])
by (*metis less-imp-le max-size size-of-def typ-uinfo-size*)

lemma *first-in-intvl*:
$b \neq 0 \Longrightarrow a \in \{a\ ..+\ b\}$
by (*force simp*: *intvl-def*)

lemma *list-map-comono*:
assumes *s*: *list-map m* $\subseteq_m$ *list-map n*
shows $m \leq n$
using *s*
proof (*induct m arbitrary*: *n rule*: *rev-induct*)
case *Nil* **thus** *?case* **unfolding** *list-map-def* **by** *simp*
next
case (*snoc x xs*)

from *snoc.prems* **have**
sm: [*length xs* $\mapsto$ *x*] ++ *list-map xs* $\subseteq_m$ *list-map n*
unfolding *list-map-def* **by** *simp*

hence *xsn*: $xs \leq n$
by (*rule snoc.hyps* [*OF map-add-le-mapE*])

have *list-map n* (*length xs*) = *Some x* **using** *sm*
by (*simp add*: *map-le-def list-map-def merge-dom2 set-zip*)

hence *length xs* < *length n* **and** *x* = *n* ! *length xs*
by (*auto simp add*: *list-map-eq split*: *if-split-asm*)

thus *xs* @ [*x*] $\leq n$ **using** *xsn*
by (*simp add*: *append-one-prefix less-eq-list-def*)
qed

lemma *valid-root-footprint-overlap-sub-typ*:
assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: {*x* ..+ *size-td t*} ∩ {*y* ..+ *size-td s*} ≠ {}
shows $s \leq t$
using *valid-root-x overlap valid-y*
by (*auto simp add*: *valid-root-footprint-def valid-footprint-def Let-def typ-tag-le-def*)
(*metis intvlD list-map-comono typ-slice-sub typ-tag-le-def*)

lemma *valid-root-footprint-type-neq*:
⟦ *valid-root-footprint d p s*;
valid-root-footprint d q t;
$s \neq t$ ⟧ ⟹
$p \notin \{q..+ (size\text{-}td\ t)\}$

by (*metis antisym-conv disjoint-iff first-in-intvl neq0-conv valid-root-footprint-def*

valid-root-footprint-overlap-sub-typ valid-root-footprint-valid-footprint)

lemma *valid-root-footprint-ptr-force-type-iff*:
fixes *p* :: *'a::mem-type ptr*
assumes *t*: *wf-size-desc t*
shows *valid-root-footprint* (*ptr-force-type p s*) *a t* ⟷
(*valid-root-footprint s a t* ∧ *disjnt* {*a* ..+ *size-td t*} (*ptr-span p*)) ∨
(*t* = *typ-uinfo-t TYPE*('*a*) ∧ *p* = *Ptr a*)
proof *cases*
assume *disjnt*: *disjnt* {*a* ..+ *size-td t*} (*ptr-span p*)
moreover have *p* ≠ *Ptr a*
using *disjnt*[*unfolded disjnt-iff*, *THEN spec*, *of a*] *t*[*THEN wf-size-desc-gt*(*1*)]
by (*cases p*) (*auto simp*: *intvl-self*)
moreover have *valid-root-footprint* (*ptr-force-type p s*) *a t* = *valid-root-footprint s a t*
using *t disjnt*
by (*intro valid-root-footprint-cong-state ptr-force-type-disj*) (*simp-all add*: *disjnt-iff*)
ultimately show *?thesis*
by *simp*
next
assume *ndisjnt*: ¬ *disjnt* {*a* ..+ *size-td t*} (*ptr-span p*)
show *?thesis*
using *ndisjnt*
by (*metis* (*no-types*) *valid-root-footprint-valid-simple-footprint-typ-uinfo-t disjnt-def*
valid-root-foot-print-ptr-force-type intvl-inter ptr-val.ptr-val-def size-of-tag t
valid-simple-footprint-ptr-force-type-iff valid-root-footprint-type-neq)
qed

lemma *valid-root-footprint-fold-ptr-force-type-iff*:
fixes *ps* :: *'a::mem-type ptr list*
assumes [*simp*]: *wf-size-desc t*
shows *distinct-prop* (λ*p1 p2*. *disjnt* (*ptr-span p1*) (*ptr-span p2*)) *ps* ⟹
valid-root-footprint (*fold ptr-force-type ps s*) *a t* ⟷
(*valid-root-footprint s a t* ∧ *disjnt* {*a* ..+ *size-td t*} (⋃*p*∈*set ps*. *ptr-span p*)) ∨
(*t* = *typ-uinfo-t TYPE*('*a*) ∧ *Ptr a* ∈ *set ps*)
by (*induction ps arbitrary*: *s*)
(*auto simp*: *valid-root-footprint-ptr-force-type-iff size-of-tag distinct-prop-append*)

lemma *valid-simple-footprint-neq*:
assumes *valid-p*: *valid-simple-footprint d p s*
and *valid-q*: *valid-simple-footprint d q t*
and *neq*: *p* ≠ *q*
shows *p* ∉ {*q*..+ (*size-td t*)}

proof −
have *heap-type-p*: *heap-type-tag d p = HeapType s*
apply (*metis valid-p valid-simple-footprint-def*)
done

have *heap-type-q*: *heap-type-tag d q = HeapType t*
apply (*metis valid-q valid-simple-footprint-def*)
done

have *heap-type-q-footprint*:
$\bigwedge x.\ x \in \{(q + 1)..+(\textit{size-td}\ t - \textit{Suc}\ 0)\} \Longrightarrow \textit{heap-type-tag}\ d\ x = \textit{HeapFootprint}$
apply (*insert valid-q*)
apply (*simp add*: *valid-simple-footprint-def*)
done

show *?thesis*
using *heap-type-q-footprint heap-type-p neq*
intvl-neq-start heap-type-q
by (*metis heap-typ-contents.simps*(*2*))
qed

lemma *valid-simple-footprint-type-neq*:
⟦ *valid-simple-footprint d p s*;
valid-simple-footprint d q t;
s ≠ *t* ⟧ ⟹
p ∉ {*q*..+ (*size-td t*)}
apply (*subgoal-tac p* ≠ *q*)
apply (*rule valid-simple-footprint-neq*, *simp-all*)[*1*]
apply (*clarsimp simp*: *valid-simple-footprint-def*)
done

lemma *valid-simple-footprint-neq-disjoint*:
⟦ *valid-simple-footprint d p s*; *valid-simple-footprint d q t*; *p* ≠ *q* ⟧ ⟹
{*p*..+(*size-td s*)} ∩ {*q*..+ (*size-td t*)} = {}
apply (*rule ccontr*)
apply (*fastforce simp*: *valid-simple-footprint-neq dest*!: *intvl-inter*)
done

lemma *valid-simple-footprint-type-neq-disjoint*:
⟦ *valid-simple-footprint d p s*;
valid-simple-footprint d q t;
s ≠ *t* ⟧ ⟹
{*p*..+(*size-td s*)} ∩ {*q*..+ (*size-td t*)} = {}
apply (*subgoal-tac p* ≠ *q*)
apply (*rule valid-simple-footprint-neq-disjoint*, *simp-all*)[*1*]
apply (*clarsimp simp*: *valid-simple-footprint-def*)
done

lemma *valid-simple-footprint-disjnt-or-eq*:
valid-simple-footprint d a1 t1 $\Longrightarrow$ *valid-simple-footprint d a2 t2* $\Longrightarrow$
disjnt $\{a1\ ..+\ size\text{-}td\ t1\}\ \{a2\ ..+\ size\text{-}td\ t2\} \vee (a1 = a2 \wedge t1 = t2)$
using *valid-simple-footprint-type-neq-disjoint*[*of d a1 t1 a2 t2*]
using *valid-simple-footprint-neq-disjoint*[*of d a1 t1 a2 t2*]
by (*auto simp*: *disjnt-def*)

lemma *valid-root-footprint-type-neq-disjoint*:
$[\![$ *valid-root-footprint d p s*;
valid-root-footprint d q t;
$s \neq t\ ]\!] \Longrightarrow$
$\{p..+(size\text{-}td\ s)\} \cap \{q..+\ (size\text{-}td\ t)\} = \{\}$
by (*metis intvl-inter valid-root-footprint-type-neq*)

lemma *valid-root-footprint-neq*:
assumes *valid-p*: *valid-root-footprint d p s*
and *valid-q*: *valid-root-footprint d q t*
and *neq*: $p \neq q$
shows $p \notin \{q..+\ (size\text{-}td\ t)\}$
proof
assume *p*: $p \in \{q..+size\text{-}td\ t\}$
then have $\neg\ t \leq s$
by (*metis* (*no-types*, *opaque-lifting*) *less-irrefl less-le-trans neq valid-footprint-sub2*

valid-p valid-q valid-root-footprint-valid-footprint)
moreover
from *p* **have** $\{p..+size\text{-}td\ s\} \cap \{q..+size\text{-}td\ t\} \neq \{\}$
by (*metis disjoint-iff first-in-intvl less-irrefl valid-p valid-root-footprint-def*)

from *valid-root-footprint-overlap-sub-typ* [*OF valid-p valid-root-footprint-valid-footprint*
[*OF valid-q*] *this*]
have $t \leq s$ **.**
ultimately show *False*
by *blast*
qed

lemma *valid-root-footprint-neq-disjoint*:
$[\![$ *valid-root-footprint d p s*; *valid-root-footprint d q t*; $p \neq q\ ]\!] \Longrightarrow$
$\{p..+(size\text{-}td\ s)\} \cap \{q..+\ (size\text{-}td\ t)\} = \{\}$
by (*metis intvl-inter valid-root-footprint-neq*)

lemma *valid-root-footprint-disjnt-or-eq*:
valid-root-footprint d a1 t1 $\Longrightarrow$ *valid-root-footprint d a2 t2* $\Longrightarrow$
disjnt $\{a1\ ..+\ size\text{-}td\ t1\}\ \{a2\ ..+\ size\text{-}td\ t2\} \vee (a1 = a2 \wedge t1 = t2)$
using *valid-root-footprint-type-neq-disjoint*[*of d a1 t1 a2 t2*]
using *valid-root-footprint-neq-disjoint*[*of d a1 t1 a2 t2*]
by (*auto simp*: *disjnt-def*)

definition *ptr-aligned-u* :: *typ-uinfo* $\Rightarrow$ *addr* $\Rightarrow$ *bool* **where**

ptr-aligned-u t a ≡ 2^(align-td t) dvd unat a

lemma *ptr-aligned-ptr-aligned-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *ptr-aligned p = ptr-aligned-u (typ-uinfo-t TYPE('a)) (ptr-val p)*
by (*simp add*: *ptr-aligned-def ptr-aligned-u-def align-of-def typ-uinfo-t-def*)

definition *c-null-guard-u* :: *typ-uinfo ⇒ addr ⇒ bool* **where**
c-null-guard-u t a ≡ 0 ∉ {a..+size-td t}

lemma *c-null-guard-c-null-guard-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *c-null-guard p = c-null-guard-u (typ-uinfo-t TYPE('a)) (ptr-val p)*
by (*simp add*: *c-null-guard-def c-null-guard-u-def size-of-def*)

definition *c-guard-u* :: *typ-uinfo ⇒ addr ⇒ bool* **where**
c-guard-u t a ≡ ptr-aligned-u t a ∧ c-null-guard-u t a

lemma *c-guard-c-guard-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *c-guard p = c-guard-u (typ-uinfo-t TYPE('a)) (ptr-val p)*
by (*simp add*: *ptr-aligned-ptr-aligned-u-conv c-guard-def c-guard-u-def c-null-guard-c-null-guard-u-conv*)

definition
root-ptr-valid-u :: *typ-uinfo ⇒ heap-typ-desc ⇒ addr ⇒ bool* **where**
root-ptr-valid-u t d a ≡ valid-root-footprint d a t ∧ c-guard-u t a

definition
cvalid-u :: *typ-uinfo ⇒ heap-typ-desc ⇒ addr ⇒ bool* **where**
cvalid-u t d a ≡ valid-footprint d a t ∧ c-guard-u t a

lemma *root-ptr-valid-root-ptr-valid-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *root-ptr-valid d p = root-ptr-valid-u (typ-uinfo-t TYPE('a)) d (ptr-val p)*
by (*simp add*: *root-ptr-valid-def root-ptr-valid-u-def c-guard-c-guard-u-conv*)

lemma *root-ptr-valid-ptr-force-type*:
c-guard p ⟹ root-ptr-valid (ptr-force-type p s) (p::*'a*::*mem-type ptr)*
by (*simp add*: *root-ptr-valid-root-ptr-valid-u-conv root-ptr-valid-u-def c-guard-c-guard-u-conv*
valid-root-foot-print-ptr-force-type)
lemma *cvalid-cvalid-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *d ⊨ₜ p = cvalid-u (typ-uinfo-t TYPE('a)) d (ptr-val p)*
by (*simp add*: *h-t-valid-def cvalid-u-def c-guard-c-guard-u-conv*)

lemma *root-ptr-valid-u-cvalid-u*: *root-ptr-valid-u t d a ⟹ cvalid-u t d a*
by (*simp add*: *root-ptr-valid-u-def cvalid-u-def valid-root-footprint-valid-footprint*)

lemma *fold-root-ptr-valid-u*:

root-ptr-valid-u (*typ-uinfo-t TYPE*('*a*::*c-type*)) *d a* = *root-ptr-valid d* (*PTR* ('*a*) *a*)
by (*simp add*: *root-ptr-valid-root-ptr-valid-u-conv*)

lemma *ptr-force-type-eq-ptr-force-type-u*:
ptr-force-type (*p* :: '*a*::*c-type ptr*) = *ptr-force-type-u* (*typ-uinfo-t TYPE*('*a*)) (*ptr-val p*)
by (*simp add*: *ptr-force-type-def ptr-force-type-u-def typ-slices-def typ-slices-u-def size-of-def*)

lemma *typ-slices-u-length* [*simp*]: *length* (*typ-slices-u t*) = *size-td t*
by (*simp add*: *typ-slices-u-def*)

lemma *typ-slices-u-index* [*simp*]:
n < *size-td t* ⟹ *typ-slices-u t* ! *n* = *list-map* (*typ-slice-t t n*)
by (*simp add*: *typ-slices-u-def*)

lemma *valid-root-footprint-ptr-force-type-u*:
wf-size-desc t ⟹ *size-td t* < *addr-card* ⟹
valid-root-footprint (*ptr-force-type-u t a h*) *a t*
by (*simp add*: *valid-root-footprint-def ptr-force-type-u-def Let-def wf-size-desc-gt htd-upd-at*)

lemma *valid-root-footprint-ptr-force-type-u-size*:
wf-size-desc t ⟹ *size-td t* < *addr-card* ⟹
valid-root-footprint (*ptr-force-type-u t a h*) *a t*
by (*simp add*: *valid-root-footprint-def ptr-force-type-u-def Let-def wf-size-desc-gt htd-upd-at*)

definition
stack-alloc:: *addr set* ⇒ '*a*:: *mem-type itself* ⇒ *heap-typ-desc* ⇒ ('*a ptr* × *heap-typ-desc*) *set* **where**
‹*stack-alloc* $\mathcal{S}$ *T d* = {
(*p*::'*a ptr*, *d'*).
ptr-span p ⊆ $\mathcal{S}$ ∧
typ-uinfo-t (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) ∧
(∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)) ∧
ptr-aligned p ∧
d' = *ptr-force-type p d*
}›

definition
stack-allocs:: *nat* ⇒ *addr set* ⇒ '*a*:: *mem-type itself* ⇒ *heap-typ-desc* ⇒ ('*a ptr* × *heap-typ-desc*) *set* **where**
‹*stack-allocs n* $\mathcal{S}$ *T d* = {
(*p*::'*a ptr*, *d'*).
0 < *n* ∧

$(\forall\, i < n.\ ptr\text{-}span\ (p +_p int\ i) \subseteq \mathcal{S}) \wedge$
$typ\text{-}uinfo\text{-}t\ (TYPE('a)) \neq typ\text{-}uinfo\text{-}t\ (TYPE(stack\text{-}byte)) \wedge$
$(\forall\, a \in \{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a)\}\ .\ root\text{-}ptr\text{-}valid\ d\ (PTR\ (stack\text{-}byte)\ a)) \wedge$
$ptr\text{-}aligned\ p \wedge$
$d' = fold\ (\lambda i.\ ptr\text{-}force\text{-}type\ (p +_p int\ i))\ [0..<n]\ d$
$\}\rangle$

lemma *stack-alloc-stack-allocs-conv*: *stack-alloc = stack-allocs 1*
by (*simp add*: *stack-alloc-def stack-allocs-def fun-eq-iff*)

lemma *htd-update-list-other*:
assumes *bound*: *length xs < addr-card*
assumes *notin*: *x ∉ {p..+length xs}*
shows *htd-update-list p xs d x = d x*
using *bound notin*
proof (*induct xs arbitrary*: *p d x*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x1 xs*)
from *Cons.prems* **obtain**
Suc (length xs) < addr-card **and**
bound': *length xs < addr-card* **and**
x ∉ {p..+Suc (length xs)} **and**
notin': *x ∉ {(p + 1)..+(length xs)}* **and**
neq: *x ≠ p*
apply *clarsimp*
by (*metis One-nat-def add-diff-cancel-left' intvl-plus-sub-Suc intvl-self plus-1-eq-Suc zero-less-Suc*)

note *hyp = Cons.hyps* [*OF bound' notin'*, **where** *d = (d(p := (True, snd (d p) ++ x1)))*]
show *?case* **by** (*simp add*: *hyp fun-upd-other neq*)
qed

lemma *dom-typ-slice-t-stack-byte*:
dom (list-map (typ-slice-t (typ-uinfo-t TYPE(stack-byte)) n)) = {0}
by (*simp add*: *typ-info-stack-byte typ-uinfo-t-def*)

lemma *htd-update-list-same'*:
⟦*0 < unat k*; *unat k ≤ addr-card − length v*⟧ ⟹ *htd-update-list (p + k) v h p = h p*
apply (*insert htd-update-list-same* [**where** *v=v* **and** *p=p* **and** *h=h* **and** *k=unat k*])
apply *clarsimp*
done

lemma *dom-htd-update-list*:

```
 assumes xs-bound: length xs < addr-card
 assumes n-bound: n < length xs
 shows dom (snd (htd-update-list a xs d (a + word-of-nat n))) =
       dom (snd (d (a + word-of-nat n))) ∪ dom (xs ! n)
  using xs-bound n-bound
proof (induct xs arbitrary: n a d)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  from Cons.prems obtain
    Suc (length xs) < addr-card and
    lxs: length xs < addr-card and
    n: n < Suc (length xs)
    by auto
  note hyp =  Cons.hyps [OF lxs]
  show ?case
  proof (cases n)
    case 0
    show ?thesis
      apply (simp add: 0)
    by (smt (verit, ccfv-threshold) One-nat-def add-cancel-left-right add-diff-cancel-left'

        dom-map-add fun-upd-same htd-update-list-same' le-eq-less-or-eq le-iff-add
       less-Suc-eq-0-disj lxs nat-neq-iff snd-conv sup-commute unsigned-1 zero-order(1))
  next
    case (Suc m)
    have conv: a + 1 + word-of-nat m = a + (1 + word-of-nat m)
      by simp

    from Suc n have m-xs: m < length xs by simp
   note hyp = hyp [where a = a + 1 and d = (d(a := (True, snd (d a) ++ x)))
and n = m, OF m-xs, simplified conv]
    show ?thesis
      apply (simp add: Suc)
      apply (simp add: hyp )
    by (smt (verit, best) add-cancel-left-right add-diff-cancel-left' add-leE fun-upd-other

        less-natE linorder-not-less lxs m-xs one-plus-x-zero plus-1-eq-Suc)
  qed
qed

lemma dom-ptr-retyp:
  fixes p::'a::mem-type ptr
  assumes n: n < size-of TYPE('a)
  shows dom (snd (ptr-retyp p d (ptr-val p + of-nat n))) =
        dom (snd (d (ptr-val p + of-nat n))) ∪
        {0..<length (typ-slice-t (typ-uinfo-t TYPE('a)) n)}
proof −
```

```
  have sz-bound: size-of TYPE('a) < addr-card by simp
  hence l-slices: length (typ-slices TYPE('a)) < addr-card by simp
  with sz-bound n have n': n < length (typ-slices TYPE('a)) by simp

  from n'
 have dom-conv: dom (typ-slices TYPE('a) ! n) = {0..<length (typ-slice-t (typ-uinfo-t
TYPE('a)) n)}
    by (simp)
  show ?thesis
    apply (simp add: ptr-retyp-def size-of-def )
   apply (simp add:  dom-htd-update-list [OF l-slices n', where a = ptr-val p and
d = d] dom-conv)
    done
qed

lemma length-typ-slice-t: 0 < length (typ-slice-t t n )
  by (cases t) auto

lemma valid-root-footprint-retyp-stack':
  fixes p::'a::mem-type ptr
 assumes stack: ∀ a ∈ ptr-span p. valid-root-footprint d a (typ-uinfo-t TYPE(stack-byte))
  shows valid-root-footprint(ptr-retyp p d) (ptr-val p) (typ-uinfo-t TYPE('a))
proof −
  {
    fix n
    assume n-bound: n < size-td (typ-info-t TYPE('a))
    have dom (snd (ptr-retyp p d (ptr-val p + word-of-nat n))) =
          {0..<length (typ-slice-t (typ-uinfo-t TYPE('a)) n)}
    proof −
      from n-bound have n-sz: n < size-of (TYPE('a)) by (simp add: size-of-def)
       from n-sz
       have ptr-val p + word-of-nat n ∈ ptr-span p
         by (simp add: intvlI)
      note conv = valid-root-footprint-dom-typing [OF stack [rule-format, OF this],
where n=0, simplified ]
       have dom-eq: dom (snd (d (ptr-val p + word-of-nat n))) = {0}
         by (simp add: conv typ-uinfo-t-def typ-info-stack-byte)

      from length-typ-slice-t have l: 0 < length (typ-slice-t (typ-uinfo-t TYPE('a))
n) by simp
       show ?thesis
         apply (simp add: dom-ptr-retyp [OF n-sz] dom-eq )
         using l atLeastLessThan-iff by blast
    qed
  } note  dom-conv = this
  show ?thesis
   by (simp add: valid-root-footprint-valid-footprint-dom-conv ptr-retyp-valid-footprint
dom-conv)
qed
```

lemma (**in** *mem-type*) *ptr-force-type-valid-footprint*:
valid-footprint (*ptr-force-type p d*) (*ptr-val* (*p*::'*a ptr*)) (*typ-uinfo-t TYPE*('*a*))
using *valid-root-foot-print-ptr-force-type valid-root-footprint-valid-footprint* **by** *blast*

lemma *ptr-force-type-valid-footprint*:
valid-footprint (*ptr-force-type p d*) (*ptr-val* (*p*::'*a*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
using *valid-root-foot-print-ptr-force-type valid-root-footprint-valid-footprint* **by** *blast*

lemma *valid-root-footprint-retyp-stack*:
fixes *p*::'*a*::*mem-type ptr*
assumes *stack*: ∀ *a* ∈ *ptr-span p*. *valid-root-footprint d a* (*typ-uinfo-t TYPE*(*stack-byte*))
shows *valid-root-footprint*(*ptr-force-type p d*) (*ptr-val p*) (*typ-uinfo-t TYPE*('*a*))
proof −
{
fix *n*
assume *n-bound*: *n* < *size-td* (*typ-info-t TYPE*('*a*))
have *dom* (*snd* (*ptr-force-type p d* (*ptr-val p* + *word-of-nat n*))) =
{*0*..<*length* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *n*)}
proof −
from *n-bound* **have** *n-sz*: *n* < *size-of* (*TYPE*('*a*)) **by** (*simp add*: *size-of-def*)
from *n-sz*
have *ptr-val p* + *word-of-nat n* ∈ *ptr-span p*
by (*simp add*: *intvlI*)
note *conv* = *valid-root-footprint-dom-typing* [*OF stack* [*rule-format*, *OF this*],
where *n*=*0*, *simplified*]
have *dom-eq*: *dom* (*snd* (*d* (*ptr-val p* + *word-of-nat n*))) = {*0*}
by (*simp add*: *conv typ-uinfo-t-def typ-info-stack-byte*)

from *length-typ-slice-t* **have** *l*: *0* < *length* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *n*) **by** *simp*
show *?thesis*
by (*simp add*: *n-sz size-of-tag valid-root-foot-print-ptr-force-type valid-root-footprint-dom-typing*)
qed
} note *dom-conv* = *this*
show *?thesis*
by (*simp add*: *valid-root-footprint-valid-footprint-dom-conv ptr-force-type-valid-footprint dom-conv*)
qed

lemma *root-ptr-valid-retyp-stack'*:
fixes *p*::'*a*::*mem-type ptr*
assumes *stack*: ∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
assumes *aligned*: *ptr-aligned p*

```
  shows root-ptr-valid (ptr-retyp p d) p
proof −
  from stack
  have stack': ∀ a ∈ ptr-span p. valid-root-footprint d a (typ-uinfo-t TYPE(stack-byte))
    by (simp add: root-ptr-valid-def)
  from stack aligned have c-guard: c-guard p
    apply (simp add: root-ptr-valid-def c-guard-def c-null-guard-def)
    using first-in-intvl by blast

  from valid-root-footprint-retyp-stack' [OF stack']
  have valid-root-footprint (ptr-retyp p d) (ptr-val p) (typ-uinfo-t TYPE('a)).
  with c-guard show ?thesis
    by (simp add: root-ptr-valid-def)
qed

lemma root-ptr-valid-retyp-stack:
  fixes p::'a::mem-type ptr
  assumes stack: ∀ a ∈ ptr-span p. root-ptr-valid d (PTR (stack-byte) a)
  assumes aligned: ptr-aligned p
  shows root-ptr-valid (ptr-force-type p d) p
proof −
  from stack
  have stack': ∀ a ∈ ptr-span p. valid-root-footprint d a (typ-uinfo-t TYPE(stack-byte))
    by (simp add: root-ptr-valid-def)
  from stack aligned have c-guard: c-guard p
    apply (simp add: root-ptr-valid-def c-guard-def c-null-guard-def)
    using first-in-intvl by blast

  from valid-root-footprint-retyp-stack [OF stack']
  have valid-root-footprint (ptr-force-type p d) (ptr-val p) (typ-uinfo-t TYPE('a)).
  with c-guard show ?thesis
    by (simp add: root-ptr-valid-def)
qed

lemma fold-ptr-retyp-other:
  fixes p::'a::mem-type ptr
  assumes a-notin: a ∉ {ptr-val p ..+ n * size-of TYPE('a)}
  shows (fold (λi. ptr-retyp (p +p int i)) [0..<n] d) a = d a
  using a-notin
proof (induct n arbitrary: d)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems have a-notin: a ∉ {ptr-val p..+ Suc n * size-of TYPE('a)} .
  from a-notin
  have a-notin-n: a ∉ {ptr-val p..+ n * size-of TYPE('a)}
    by (metis add-leD2 intvlD intvlI linorder-not-less mult.commute mult-Suc-right)
  from a-notin
```

```
  have a-notin-Suc: a ∉ ptr-span (p +p int n)
    apply (clarsimp simp add: intvl-def ptr-add-def)
   by (metis (no-types, opaque-lifting) add.commute add-less-cancel-right mult.commute
of-nat-add of-nat-mult)

  have fold-split: (fold (λi. ptr-retyp (p +p int i)) ([0..<Suc n]) d) =
        (fold (λi. ptr-retyp (p +p int i)) ([0..<n] @ [n]) d)
    apply (simp only: Many-More.split-upt-on-n [where n=n])
    apply simp
    done
  show ?case
    apply (simp only: fold-split)
    apply (simp only: fold-append)
    apply (simp)
    apply (subst (2) Suc.hyps [OF a-notin-n, symmetric])
    apply (rule ptr-retyp-d [OF a-notin-Suc])
    done
qed

lemma (in mem-type) ptr-force-type-d:
  x ∉ {ptr-val (p::'a ptr)..+size-of TYPE('a)} ⟹
      ptr-force-type p d x = d x
  by (simp add: htd-upd-disj local.ptr-force-type-def)

lemma fold-ptr-force-type-other:
  fixes p::'a::mem-type ptr
  assumes a-notin: a ∉ {ptr-val p ..+ n * size-of TYPE('a)}
  shows (fold (λi. ptr-force-type (p +p int i)) [0..<n] d) a = d a
  using a-notin
proof (induct n arbitrary: d)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems have a-notin: a ∉ {ptr-val p..+ Suc n * size-of TYPE('a)} .
  from a-notin
  have a-notin-n: a ∉ {ptr-val p..+ n * size-of TYPE('a)}
   by (metis add-leD2 intvlD intvlI linorder-not-less mult.commute mult-Suc-right)
  from a-notin
  have a-notin-Suc: a ∉ ptr-span (p +p int n)
    apply (clarsimp simp add: intvl-def ptr-add-def)
   by (metis (no-types, opaque-lifting) add.commute add-less-cancel-right mult.commute
of-nat-add of-nat-mult)

  have fold-split: (fold (λi. ptr-force-type (p +p int i)) ([0..<Suc n]) d) =
        (fold (λi. ptr-force-type (p +p int i)) ([0..<n] @ [n]) d)
    apply (simp only: Many-More.split-upt-on-n [where n=n])
    apply simp
    done
```

show *?case*
apply (*simp only*: *fold-split*)
apply (*simp only*: *fold-append*)
apply (*simp*)
apply (*subst* (*2*) *Suc.hyps* [*OF a-notin-n*, *symmetric*])
apply (*rule ptr-force-type-d* [*OF a-notin-Suc*])
done
qed

lemma *root-ptr-valid-domain*:
fixes *p*::*'a*::*mem-type ptr*
assumes *root-ptr-valid d p*
assumes $\bigwedge a.\ a \in$ *ptr-span p* $\Longrightarrow$ *d' a* = *d a*
shows *root-ptr-valid d' p*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *root-ptr-valid-def s-footprintD s-footprintI2 size-of-tag valid-root-footprint-def*)

lemma *root-ptr-valid-domain'*:
fixes *p*::*'a*::*mem-type ptr*
assumes $\bigwedge a.\ \ a \in$ *ptr-span p* $\Longrightarrow$ *d' a* = *d a*
shows *root-ptr-valid d' p* = *root-ptr-valid d p*
by (*metis assms root-ptr-valid-domain*)

lemma *root-ptr-valid-range-not-NULL*:
root-ptr-valid htd (*p* :: (*'a* :: *c-type*) *ptr*)
$\Longrightarrow$ *0* $\notin$ {*ptr-val p ..+ size-of TYPE*(*'a*)}
apply (*clarsimp simp*: *root-ptr-valid-def*)
apply (*metis c-guard-def c-null-guard-def*)
done

lemma *intvl-no-overflow-nat'*:
assumes *no-overflow*: *unat a* + *b* $\leq$ *2* $\hat{}$ *len-of TYPE*(*'a*::*len*)
shows (*x* $\in$ {(*a* :: *'a word*) *..+ b*}) = (*unat a* $\leq$ *unat x* $\wedge$ *unat x* $<$ (*unat a* + *b*))
apply (*simp add*: *intvl-def*)
using *no-overflow*
apply (*intro iffI conjI*)
apply (*smt* (*verit*) *add.commute nat-add-left-cancel-less nat-le-iff-add trans-less-add1 unat-of-nat-len word-arith-nat-add*)
apply (*smt* (*verit*) *add.commute nat-add-left-cancel-less nat-le-iff-add trans-less-add1 unat-of-nat-len word-arith-nat-add*)
by (*metis nat-add-left-cancel-less nat-le-iff-add of-nat-add unat-eq-of-nat unat-lt2p*)

lemma *intvl-no-overflow-nat*:
assumes *no-overflow*: *unat a* + *b* $\leq$ *addr-card*
shows (*x* $\in$ {(*a* :: *addr-bitsize word*) *..+ b*}) = (*unat a* $\leq$ *unat x* $\wedge$ *unat x* $<$ (*unat a* + *b*))
using *no-overflow* **unfolding** *addr-card-def* **using** *intvl-no-overflow-nat'* **by** (*metis card-word*)

lemma *intvl-no-overflow-nat-conv*:
assumes *no-overflow*: *unat a + b ≤ addr-card*
shows {(*a :: addr-bitsize word*) ..+ *b*} = {*x*. (*unat a ≤ unat x ∧ unat x < (unat a + b)*)}
using *intvl-no-overflow-nat* [*OF no-overflow*] **by** *blast*

lemma *zero-not-in-intvl-no-overflow*:
0 ∉ {a :: 'a::len word ..+ b} ⟹ unat a + b ≤ 2 ^ len-of TYPE('a)
apply (*rule ccontr*)
apply (*simp add*: *intvl-def not-le*)
apply (*drule-tac x=2 ^ len-of TYPE('a) − unat a* **in** *spec*)
apply (*clarsimp simp*: *not-less*)
by (*smt* (*verit*) *Nat.le-diff-conv2 add.commute add.left-neutral add-uminus-conv-diff*

cancel-comm-monoid-add-class.diff-cancel diff-zero linorder-not-less
nat-less-le of-nat-numeral semiring-1-class.of-nat-power unat-0
unat-lt2p unat-minus' word-arith-nat-add word-exp-length-eq-0)

lemma *root-ptr-valid-last-byte-no-overflow*:
root-ptr-valid htd (*p :: ('a :: c-type) ptr*)
⟹ unat (*ptr-val p*) + *size-of TYPE('a) ≤ 2 ^ len-of TYPE*(*addr-bitsize*)
by (*metis c-guard-def c-null-guard-def root-ptr-valid-def*
zero-not-in-intvl-no-overflow)

lemma *root-ptr-valid-retyps-stack'*:
fixes *p::'a::mem-type ptr*
assumes *stack*: ∀ *a* ∈ {*ptr-val p ..+ n ∗ size-of TYPE('a)*}. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
assumes *aligned*: *ptr-aligned p*
assumes *i-bound*: *i < n*
shows *root-ptr-valid* (*fold* (λ*i*. *ptr-retyp* (*p* +*p int i*)) [*0..<n*] *d*) (*p* +*p int i*)
using *stack i-bound*
proof (*induct n arbitrary*: *d*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)

have *stack*: ∀ *a*∈{*ptr-val p..+ Suc n ∗ size-of TYPE('a)*}. *root-ptr-valid d* (*PTR*(*stack-byte*) *a*) **by** *fact*
from *stack* **have** *stack-n*: ∀ *a*∈{*ptr-val p..+ n ∗ size-of TYPE('a)*}. *root-ptr-valid d* (*PTR*(*stack-byte*) *a*)
by (*metis add.commute intvlD intvlI mult-Suc trans-less-add1*)
from *stack*
have *stack-Suc*: ∀ *a* ∈ *ptr-span* (*p* +*p int n*). *root-ptr-valid d* (*PTR*(*stack-byte*) *a*)
apply (*clarsimp simp add*: *intvl-def ptr-add-def*)
by (*metis* (*no-types*, *opaque-lifting*) *add.assoc add.commute add-less-cancel-right*

of-nat-add of-nat-mult)

from *stack* **have** *no-overflow*: $0 \notin \{ptr\text{-}val\ p..+\ Suc\ n * size\text{-}of\ TYPE('a)\}$
apply (*clarsimp simp add*: *intvl-def*)
by (*metis c-guard-NULL-simp root-ptr-valid-def*)

from *stack* **have** *no-overflow'*: $\forall a \in \{ptr\text{-}val\ p..+\ Suc\ n * size\text{-}of\ TYPE('a)\}.\ 0 \notin ptr\text{-}span\ (PTR(stack\text{-}byte)\ a)$
apply (*clarsimp simp add*: *intvl-def*)
by (*metis c-guard-NULL-simp root-ptr-valid-def*)

have *i-bound*: $i < Suc\ n$ **by** *fact*

have *fold-split*: $(fold\ (\lambda i.\ ptr\text{-}retyp\ (p +_p int\ i))\ ([0..<Suc\ n])\ d) =$
$(fold\ (\lambda i.\ ptr\text{-}retyp\ (p +_p int\ i))\ ([0..<n] @ [n])\ d)$
apply (*simp only*: *Many-More.split-upt-on-n* [**where** n=n])
apply *simp*
done

have *sz-a*: $0 < size\text{-}of\ (TYPE('a))$
by *simp*
hence *sz-bound*: $size\text{-}of\ TYPE(stack\text{-}byte) \leq size\text{-}of\ TYPE('a)$
using *size-of-stack-byte*(*1*) **by** *linarith*

from *stack*
have *bound3*: $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ (TYPE('a)) < addr\text{-}card$
by (*smt* (*verit*, *ccfv-SIG*) *add.commute add.left-neutral add-less-cancel-right c-guard-NULL-simp len-of-addr-card less-le mult.commute mult-Suc-right not-less root-ptr-valid-def stack-n sz-a zero-not-in-intvl-no-overflow*)
from *bound3* **have** *unat-dist1*: $unat\ (ptr\text{-}val\ (p +_p int\ n)) = unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ (TYPE('a))$
by (*smt* (*verit*, *ccfv-threshold*) *Abs-fnat-hom-add Abs-fnat-hom-mult CTypes-Defs.ptr-add-def len-of-addr-card*
of-int-of-nat-eq of-nat-inverse ptr-val.ptr-val-def word-unat.Rep-inverse')
show *?case*
proof (*cases* i=n)
case *True*
show *?thesis*
apply (*simp only*: *fold-split*)
apply (*simp only*: *fold-append*)
apply (*simp add*: *True*)
apply (*rule root-ptr-valid-retyp-stack'*)
subgoal
apply *clarify*
apply (*rule root-ptr-valid-domain*)
apply (*rule stack-Suc* [*rule-format*])
apply *assumption*
apply (*rule fold-ptr-retyp-other*)

apply (*subst* (*asm*) *intvl-no-overflow-nat-conv*)
subgoal
apply (*simp add*: *unat-dist1*)
using *bound3*
by (*metis c-guard-NULL-simp len-of-addr-card root-ptr-valid-def stack-Suc unat-dist1 zero-not-in-intvl-no-overflow*)
apply (*subst* (*asm*) *intvl-no-overflow-nat-conv*)
subgoal
apply *simp*
using *bound3*
by (*metis Suc-leI len-of-addr-card unat-lt2p*)
apply (*subst intvl-no-overflow-nat-conv*)
subgoal
using *bound3*
by (*simp add*: *mult.commute*)
subgoal using *bound3* **by** (*simp add*: *mult.commute unat-dist1*)
done
subgoal
using *aligned ptr-aligned-plus* **by** *blast*
done
next
case *False*
with *i-bound* **have** *i-bound-n*: $i < n$ **by** *simp*
from *no-overflow* **have** *bound1*: *unat* (*ptr-val p* + *word-of-nat n* ∗ *word-of-nat* (*size-of TYPE*('*a*))) < *addr-card*
by (*metis len-of-addr-card unsigned-less*)
from *Suc.hyps* [*OF stack-n i-bound-n*]
have *hyp*: *root-ptr-valid* (*fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*) (*p* +$_p$ *int i*) **.**
from *bound3 i-bound-n* **have** *unat-dist2*: *unat* (*ptr-val* (*p* +$_p$ *int i*)) = *unat* (*ptr-val p*) + *i* ∗ *size-of* (*TYPE*('*a*))
by (*smt* (*verit*, *ccfv-threshold*) *Abs-fnat-hom-add Abs-fnat-hom-mult CTypes-Defs.ptr-add-def add.commute*
add-less-cancel-right len-of-addr-card mult-strict-right-mono nat-less-le of-int-of-nat-eq of-nat-inverse
order-less-le-trans ptr-val.ptr-val-def sz-nzero word-unat.Rep-inverse′)
show *?thesis*
apply (*simp only*: *fold-split*)
apply (*simp only*: *fold-append*)
apply (*simp*)
apply (*rule root-ptr-valid-domain*)
apply (*rule hyp*)
apply (*rule ptr-retyp-d*)

apply (*subst intvl-no-overflow-nat-conv*)
subgoal
using *bound1*
by (*metis c-guard-NULL-simp len-of-addr-card root-ptr-valid-def stack-Suc zero-not-in-intvl-no-overflow*)

```
    subgoal for a
      apply (subst (asm) intvl-no-overflow-nat-conv)
      subgoal using bound3 i-bound-n
        apply (simp add: ptr-add-def)
         by (metis (no-types, lifting) CTypesDefs.ptr-add-def hyp len-of-addr-card
of-int-of-nat-eq
            ptr-val.ptr-val-def root-ptr-valid-last-byte-no-overflow)
      subgoal using i-bound-n
        by (auto simp add: unat-dist1 unat-dist2 dest!: le-less-trans)
       (metis add.commute le-def less-le-mult-nat mem-type-class.mem-type-simps(3))
      done
    done
  qed
qed

lemma root-ptr-valid-retyps-stack:
  fixes p::'a::mem-type ptr
  assumes stack: ∀ a ∈ {ptr-val p ..+ n * size-of TYPE('a)}. root-ptr-valid d (PTR
(stack-byte) a)
  assumes aligned: ptr-aligned p
  assumes i-bound: i < n
  shows root-ptr-valid (fold (λi. ptr-force-type (p +p int i)) [0..<n] d) (p +p int
i)
  using stack i-bound
proof (induct n arbitrary: d)
  case 0
  then show ?case by simp
next
  case (Suc n)

  have stack: ∀ a∈{ptr-val p..+ Suc n * size-of TYPE('a)}. root-ptr-valid d (PTR(stack-byte)
a) by fact
  from stack have stack-n: ∀ a∈{ptr-val p..+ n * size-of TYPE('a)}. root-ptr-valid
d (PTR(stack-byte) a)
    by (metis add.commute intvlD intvlI mult-Suc trans-less-add1)
  from stack
   have stack-Suc: ∀ a ∈ ptr-span (p +p int n). root-ptr-valid d (PTR(stack-byte)
a)
    apply (clarsimp simp add: intvl-def ptr-add-def)
   by (metis (no-types, opaque-lifting) add.assoc add.commute add-less-cancel-right

       of-nat-add of-nat-mult)

  from stack have no-overflow: 0 ∉ {ptr-val p..+ Suc n * size-of TYPE('a)}
    apply (clarsimp simp add: intvl-def)
    by (metis c-guard-NULL-simp root-ptr-valid-def)

  from stack have no-overflow': ∀ a ∈ {ptr-val p..+ Suc n * size-of TYPE('a)}. 0
∉ ptr-span (PTR(stack-byte) a)
```

apply (*clarsimp simp add*: *intvl-def*)
by (*metis c-guard-NULL-simp root-ptr-valid-def*)

have *i-bound*: $i < Suc\ n$ **by** *fact*

have *fold-split*: (*fold* ($\lambda i.$ *ptr-force-type* ($p +_p$ *int* i)) ([$0..<Suc\ n$]) d) =
(*fold* ($\lambda i.$ *ptr-force-type* ($p +_p$ *int* i)) ([$0..<n$] @ [n]) d)
apply (*simp only*: *Many-More.split-upt-on-n* [**where** n=n])
apply *simp*
done

have *sz-a*: $0 <$ *size-of* (*TYPE*('*a*))
by *simp*
hence *sz-bound*: *size-of TYPE*(*stack-byte*) $\leq$ *size-of TYPE*('*a*)
using *size-of-stack-byte*(*1*) **by** *linarith*

from *stack*
have *bound3*: *unat* (*ptr-val p*) + $n *$ *size-of* (*TYPE*('*a*)) $<$ *addr-card*
by (*smt* (*verit*, *ccfv-SIG*) *add.commute add.left-neutral add-less-cancel-right c-guard-NULL-simp len-of-addr-card less-le mult.commute mult-Suc-right not-less root-ptr-valid-def stack-n sz-a zero-not-in-intvl-no-overflow*)
from *bound3* **have** *unat-dist1*: *unat* (*ptr-val* ($p +_p$ *int* n)) = *unat* (*ptr-val p*) + $n *$ *size-of* (*TYPE*('*a*))
by (*smt* (*verit*, *ccfv-threshold*) *Abs-fnat-hom-add Abs-fnat-hom-mult CTypes-Defs.ptr-add-def len-of-addr-card*
of-int-of-nat-eq of-nat-inverse ptr-val.ptr-val-def word-unat.Rep-inverse')
show *?case*
proof (*cases* i=n)
case *True*
show *?thesis*
apply (*simp only*: *fold-split*)
apply (*simp only*: *fold-append*)
apply (*simp add*: *True*)
apply (*rule root-ptr-valid-retyp-stack*)
subgoal
apply *clarify*
apply (*rule root-ptr-valid-domain*)
apply (*rule stack-Suc* [*rule-format*])
apply *assumption*
apply (*rule fold-ptr-force-type-other*)
apply (*subst* (*asm*) *intvl-no-overflow-nat-conv*)
subgoal
apply (*simp add*: *unat-dist1*)
using *bound3*
by (*metis c-guard-NULL-simp len-of-addr-card root-ptr-valid-def stack-Suc unat-dist1 zero-not-in-intvl-no-overflow*)
apply (*subst* (*asm*) *intvl-no-overflow-nat-conv*)
subgoal
apply *simp*

```
        using bound3
        by (metis Suc-leI len-of-addr-card unat-lt2p)
      apply (subst intvl-no-overflow-nat-conv)
      subgoal
        using bound3
        by (simp add: mult.commute)
      subgoal using bound3 by (simp add: mult.commute unat-dist1)
      done
    subgoal
      using aligned ptr-aligned-plus by blast
    done
 next
   case False
   with i-bound have i-bound-n: i < n by simp
   from no-overflow have bound1: unat (ptr-val p + word-of-nat n * word-of-nat
(size-of TYPE('a))) < addr-card
     by (metis len-of-addr-card unsigned-less)
   from Suc.hyps [OF stack-n i-bound-n]
   have hyp: root-ptr-valid (fold (λi. ptr-force-type (p +p int i)) [0..<n] d) (p +p
int i) .
    from bound3 i-bound-n have unat-dist2: unat (ptr-val (p +p int i)) = unat
(ptr-val p) + i * size-of (TYPE('a))
     by (smt (verit, ccfv-threshold) Abs-fnat-hom-add Abs-fnat-hom-mult CTypes-
Defs.ptr-add-def add.commute
          add-less-cancel-right len-of-addr-card mult-strict-right-mono nat-less-le
of-int-of-nat-eq of-nat-inverse
        order-less-le-trans ptr-val.ptr-val-def sz-nzero word-unat.Rep-inverse')
   show ?thesis
     apply (simp only: fold-split)
     apply (simp only: fold-append)
     apply (simp)
     apply (rule root-ptr-valid-domain)
      apply (rule hyp)
     apply (rule ptr-force-type-d)

     apply (subst intvl-no-overflow-nat-conv)
     subgoal
       using bound1
       by (metis c-guard-NULL-simp len-of-addr-card root-ptr-valid-def stack-Suc
zero-not-in-intvl-no-overflow)
     subgoal for a
       apply (subst (asm) intvl-no-overflow-nat-conv)
       subgoal using bound3 i-bound-n
         apply (simp add: ptr-add-def)
         by (metis (no-types, lifting) CTypesDefs.ptr-add-def hyp len-of-addr-card
of-int-of-nat-eq
           ptr-val.ptr-val-def root-ptr-valid-last-byte-no-overflow)
       subgoal using i-bound-n
         by (auto simp add: unat-dist1 unat-dist2 dest!: le-less-trans)
```

(*metis add.commute le-def less-le-mult-nat mem-type-class.mem-type-simps*(*3*))
done
done
qed
qed

definition *stack-free* :: *heap-typ-desc* ⇒ *addr set* **where**
stack-free d = {*a*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)}

lemma *stack-alloc-cases* [*consumes 1*]:
fixes *p*::*'a*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ *T d*
assumes *dest*:
⟦*typ-uinfo-t* (*TYPE*(*'a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*));
ptr-span p ⊆ $\mathcal{S}$;
∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*);
ptr-aligned p; *c-guard p*; *root-ptr-valid d' p*;
d' = *ptr-force-type p d*⟧ ⟹ *P*
shows *P*
using *dest root-ptr-valid-retyp-stack stack-alloc root-ptr-valid-c-guard*
by (*auto simp add*: *stack-alloc-def*)

lemma *stack-allocs-cases* [*consumes 1*]:
fixes *p*::*'a*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ *T d*
assumes *dest*:
⟦*0* < *n*; *0* ∉ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*)}; *unat* (*ptr-val p*) + *n* ∗ *size-of TYPE*(*'a*) ≤ *addr-card*;
typ-uinfo-t (*TYPE*(*'a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*));
∀ *i* < *n*. *ptr-span* (*p* +$_p$ *int i*) ⊆ $\mathcal{S}$;
∀ *a* ∈ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*)}. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*);
{*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*)} ⊆ *stack-free d*;
ptr-aligned p; *c-guard p*; *root-ptr-valid d' p*;
∀ *i* < *n*. *ptr-aligned* (*p* +$_p$ *int i*); ∀ *i* < *n*. *c-guard* (*p* +$_p$ *int i*); ∀ *i* < *n*. *root-ptr-valid d'* (*p* +$_p$ *int i*);
d' = *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*⟧ ⟹ *P*
shows *P*
proof −
from *stack-alloc* **obtain**
bound: *0* < *n* **and**
$\mathcal{S}$: (∀ *i* < *n*. *ptr-span* (*p* +$_p$ *int i*) ⊆ $\mathcal{S}$) **and**
non-stack-byte: *typ-uinfo-t* (*TYPE*(*'a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) **and**
valid-stack-byte: ∀ *a* ∈ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*)} . *root-ptr-valid d* (*PTR* (*stack-byte*) *a*) **and**
stack-free: {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*)} ⊆ *stack-free d* **and**
aligned-p: *ptr-aligned p* **and**
d': *d'* = *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*
by (*auto simp add*: *stack-allocs-def stack-free-def*)

from *valid-stack-byte* **have** *no-overflow*: $0 \notin \{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a)\}$
using *c-guard-NULL-simp root-ptr-valid-c-guard* **by** *blast*
hence *no-overflow'*: $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ TYPE('a) \leq addr\text{-}card$
by (*metis len-of-addr-card zero-not-in-intvl-no-overflow*)
note *valid-retyps* = *root-ptr-valid-retyps-stack* [*OF valid-stack-byte aligned-p*]
show *?thesis*
apply (*rule dest*)
apply (*rule bound*)
apply (*rule no-overflow*)
apply (*rule no-overflow'*)
apply (*rule non-stack-byte*)
apply (*rule* $\mathcal{S}$)
apply (*rule valid-stack-byte*)
apply (*rule stack-free*)
apply (*rule aligned-p*)
subgoal using *valid-retyps* [*OF bound*] *root-ptr-valid-c-guard d'*
by *auto*
subgoal using *valid-retyps* [*OF bound*] *d'*
by *auto*
subgoal using *aligned-p*
by (*auto simp add*: *ptr-aligned-plus*)
subgoal using *valid-retyps root-ptr-valid-c-guard d'*
by *auto*
subgoal using *valid-retyps d'*
by *auto*
apply (*rule d'*)
done
qed

lemma *stack-allocs-no-overflow*:
assumes *stack-alloc*: $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
shows $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ TYPE('a) \leq addr\text{-}card$
using *stack-allocs-cases* [*OF stack-alloc*] **.**

lemma *stack-alloc-ptr-force-type*: $(p,\ d') \in stack\text{-}alloc\ \mathcal{S}\ T\ d \Longrightarrow$
$d' = ptr\text{-}force\text{-}type\ p\ d$
by (*auto simp add*: *stack-alloc-def*)

lemma *stack-allocs-ptr-force-type*: $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ T\ d \Longrightarrow$
$d' = fold\ (\lambda i.\ ptr\text{-}force\text{-}type\ (p +_p int\ i))\ [0..<n]\ d$
by (*auto simp add*: *stack-allocs-def*)

lemma *stack-allocs-no-stack-byte*: $(p::'a::mem\text{-}type\ ptr,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ T\ d$

$\Longrightarrow typ\text{-}uinfo\text{-}t\ (TYPE('a)) \neq typ\text{-}uinfo\text{-}t\ (TYPE(stack\text{-}byte))$
by (*erule stack-allocs-cases*)

lemma *stack-allocs-*$\mathcal{S}$: $(p::'a::mem\text{-}type\ ptr,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ T\ d \Longrightarrow i < n$
$\Longrightarrow ptr\text{-}span\ (p +_p int\ i) \subseteq \mathcal{S}$

by (*erule stack-allocs-cases*) *auto*

lemma *ptr-retyp-split*: *ptr-retyp* (*p*::'*a*::*mem-type ptr*) *d a* =
(*if a* ∈ *ptr-span p then*
(*True*, *snd* (*d a*) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*a* − *ptr-val p*))))
else d a)
by (*auto simp add*: *ptr-retyp-d ptr-retyp-footprint*)

lemma (**in** *mem-type*) *ptr-force-type-footprint*:
x ∈ {*ptr-val p*..+*size-of TYPE*('*a*)} ⟹
ptr-force-type (*p*::'*a ptr*) *d x* =
(*True*,*list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*x* − *ptr-val p*))))
apply(*clarsimp simp*: *ptr-force-type-def*)
by (*metis add-diff-cancel-left*′ *htd-upd-at intvlD less-imp-le local.lt-size-of-unat-simps*

local.max-size local.typ-slices-index local.typ-slices-length word-unat.Rep-inverse)

lemma *ptr-force-type-split*: *ptr-force-type* (*p*::'*a*::*mem-type ptr*) *d a* =
(*if a* ∈ *ptr-span p then*
(*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*a* − *ptr-val p*))))
else d a)
by (*auto simp add*: *ptr-force-type-d ptr-force-type-footprint*)

lemma *in-intvl-Suc*: *x* ∈ {*x*..+*Suc n*}
by (*simp add*: *intvl-self*)

definition *zero-heap*:: *heap-mem* **where**
zero-heap = (λ*a*. *zero-class.zero*)

definition *stack-byte-typing*::*heap-typ-desc* **where**
stack-byte-typing = (λ*a*. *ptr-force-type* (*PTR*(*stack-byte*) *a*) *empty-htd a*)

definition *stack-release*:: '*a*::*mem-type ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**

stack-release p d = *override-on d stack-byte-typing* (*ptr-span p*)

definition *stack-releases*:: *nat* ⇒ '*a*::*mem-type ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
stack-releases n p d = *override-on d stack-byte-typing* {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)}

lemma *stack-release-stack-releases-conv*: *stack-release* = *stack-releases 1*
by (*auto simp add*: *stack-release-def stack-releases-def fun-eq-iff*)

lemma *stack-releases-0* [*simp*]: *stack-releases 0 d* = *id*
by (*rule ext*) (*auto simp add*: *stack-releases-def*)

lemma *stack-release-stack-alloc-inverse*: $(p,\ d') \in \mathit{stack\text{-}alloc}\ \mathcal{S}\ T\ d \Longrightarrow \mathit{stack\text{-}release}\ p\ d' = d$
apply (*rule ext*)
subgoal for *x*
apply (*clarsimp simp add*: *stack-alloc-def stack-release-def stack-byte-typing-def*)
apply (*cases x* ∈ *ptr-span p*)
apply (*simp add*: *root-ptr-valid-def*)
apply (*subst ptr-force-type-footprint*)
apply (*simp add*: *in-intvl-Suc*)
apply (*simp add*: *prod.expand valid-root-footprint-def*)
apply (*simp add*: *ptr-force-type-d*)
done
done

lemma *stack-releases-stack-allocs-inverse*: $(p,\ d') \in \mathit{stack\text{-}allocs}\ n\ \mathcal{S}\ T\ d \Longrightarrow \mathit{stack\text{-}releases}\ n\ p\ d' = d$
apply (*rule ext*)
subgoal for *x*
apply (*clarsimp simp add*: *stack-allocs-def stack-releases-def stack-byte-typing-def*)
apply (*cases x* ∈ {*ptr-val p ..+ n* ∗ *size-of TYPE*('*a*)})
apply (*simp add*: *root-ptr-valid-def*)
apply (*subst ptr-force-type-footprint*)
apply (*simp add*: *in-intvl-Suc*)
apply (*simp add*: *prod.expand valid-root-footprint-def*)
apply (*simp add*: *fold-ptr-force-type-other*)
done
done

lemma *sub-typ-stack-byte*:
$\mathit{TYPE}('b::\mathit{c\text{-}type}) \leq_\tau \mathit{TYPE}(\mathit{stack\text{-}byte}) \Longrightarrow \mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}('b) = \mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}(\mathit{stack\text{-}byte})$
by (*simp add*: *sub-typ-def typ-uinfo-t-def typ-info-stack-byte typ-tag-le-def*)

lemma *root-ptr-valid-not-subtype-disjoint*:
⟦ *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*);
$d \models_t$ (*q*::'*b*::*mem-type ptr*);
$\neg\ \mathit{TYPE}('b) \leq_\tau \mathit{TYPE}('a)$ ⟧ ⟹
ptr-span p ∩ *ptr-span q* = {}
by (*metis h-t-valid-def root-ptr-valid-def size-of-tag sub-typ-def valid-root-footprint-overlap-sub-typ*)

lemma *stack-alloc-disjoint*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: $(p,\ d') \in \mathit{stack\text{-}alloc}\ \mathcal{S}\ (\mathit{TYPE}('a::\mathit{mem\text{-}type}))\ d$
assumes *no-stack*: $\mathit{typ\text{-}uinfo\text{-}t}\ (\mathit{TYPE}('b)) \neq \mathit{typ\text{-}uinfo\text{-}t}\ (\mathit{TYPE}(\mathit{stack\text{-}byte}))$
assumes *typed*: $d \models_t q$
shows *ptr-span p* ∩ *ptr-span q* = {}

proof −
from *stack-alloc* **obtain**
typ-uinfo-t (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) **and**
stack-bytes: ∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
by (*cases rule*: *stack-alloc-cases*) *auto*
from *no-stack* **have** *no-sub-typ*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*(*stack-byte*) **by** (*metis sub-typ-stack-byte*)
{
fix *a*
assume *a*: *a* ∈ *ptr-span p*
have *a* ∉ *ptr-span q*
proof −
from *stack-bytes* [*rule-format*, *OF a*] **have** *root-ptr-valid d* (*PTR* (*stack-byte*) *a*) **.**
from *root-ptr-valid-not-subtype-disjoint* [*OF this typed no-sub-typ*] **show** *?thesis*
by (*simp add*: *disjoint-iff first-in-intvl*)
qed
}
then show *ptr-span p* ∩ *ptr-span q* = {}
by *auto*
qed

lemma *stack-allocs-disjoint*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d*') ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *d* ⊨$_t$ *q*
shows {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {}
proof −
from *stack-alloc* **obtain**
typ-uinfo-t (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) **and**
stack-bytes: ∀ *a* ∈ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)}. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
by (*cases rule*: *stack-allocs-cases*) *auto*
from *no-stack* **have** *no-sub-typ*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*(*stack-byte*) **by** (*metis sub-typ-stack-byte*)
{
fix *a*
assume *a*: *a* ∈ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)}
have *a* ∉ *ptr-span q*
proof −
from *stack-bytes* [*rule-format*, *OF a*] **have** *root-ptr-valid d* (*PTR* (*stack-byte*) *a*) **.**
from *root-ptr-valid-not-subtype-disjoint* [*OF this typed no-sub-typ*] **show** *?thesis*
by (*simp add*: *disjoint-iff first-in-intvl*)
qed
}

then show $\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}$
by *auto*
qed

lemma *stack-allocs-contained*:
assumes *stack-alloc*: $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
assumes *i*: $i < n$
shows *ptr-span* $(p +_p int\ i) \subseteq \{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a)\}$
using *i stack-allocs-no-overflow* [*OF stack-alloc*]
apply (*clarsimp simp add*: *intvl-def ptr-add-def*)
by (*metis* (*no-types*, *opaque-lifting*) *mult.commute nat-index-bound of-nat-add of-nat-mult*)

lemma *stack-allocs-disjoint'*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
assumes *no-stack*: $typ\text{-}uinfo\text{-}t\ (TYPE('b)) \neq typ\text{-}uinfo\text{-}t\ (TYPE(stack\text{-}byte))$
assumes *typed*: $d \models_t q$
assumes *i*: $i < n$
shows $ptr\text{-}span\ (p +_p int\ i) \cap ptr\text{-}span\ q = \{\}$
using *stack-allocs-disjoint* [*OF stack-alloc no-stack typed*] *i stack-allocs-contained* [*OF stack-alloc*]
by *blast*

lemma *ptr-retyp-valid-footprint-disjoint2*:
$[\![valid\text{-}footprint\ (ptr\text{-}retyp\ (q::'b::mem\text{-}type\ ptr)\ d)\ p\ s;\ \{p..+size\text{-}td\ s\} \cap \{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}\ ]\!]$
$\Longrightarrow valid\text{-}footprint\ d\ p\ s$
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply (*drule spec*, *drule* (*1*) *mp*)
apply(*subgoal-tac p + of-nat y* $\in \{p..+size\text{-}td\ s\}$)
apply (*subst* (*asm*) *ptr-retyp-d*)
apply *clarsimp*
apply *fast*
apply (*clarsimp simp add*: *ptr-retyp-d-eq-fst split*: *if-split-asm*)
apply *fast*
apply (*erule intvlI*)
done

lemma *ptr-force-type-valid-footprint-disjoint2*:
$[\![valid\text{-}footprint\ (ptr\text{-}force\text{-}type\ (q::'b::mem\text{-}type\ ptr)\ d)\ p\ s;\ \{p..+size\text{-}td\ s\} \cap \{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}\ ]\!]$
$\Longrightarrow valid\text{-}footprint\ d\ p\ s$
by (*simp add*: *disjoint-iff intvlI ptr-force-type-d valid-footprint-def*)

lemma *ptr-span-no-overflow-split-last-disjoint*:
fixes *p*::'*a*::*mem-type ptr*

assumes *no-overflow*: $0 \notin \{ptr\text{-}val\ p..+Suc\ n * size\text{-}of\ TYPE('a)\}$
shows $ptr\text{-}span\ (p +_p int\ n) \cap \{ptr\text{-}val\ p..+n * size\text{-}of\ TYPE('a)\} = \{\}$
proof −
have *sz-a*: $0 < size\text{-}of\ (TYPE('a))$
by *simp*
from *no-overflow sz-a*
have *bound-Suc*: $unat\ (ptr\text{-}val\ p) + Suc\ n * size\text{-}of\ (TYPE('a)) \leq addr\text{-}card$
by (*metis len-of-addr-card zero-not-in-intvl-no-overflow*)
with *sz-a*
have *bound-n*: $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ (TYPE('a)) < addr\text{-}card$
by (*metis lessI mult-less-cancel2 nat-add-left-cancel-less order-less-le-trans*)
show *?thesis*
apply (*subst intvl-no-overflow-nat-conv*)
subgoal using *bound-Suc*
apply (*simp add*: *ptr-add-def*)
by (*smt* (*verit*, *best*) *Abs-fnat-hom-add Abs-fnat-hom-mult add.commute add.left-commute*
bound-n len-of-addr-card mult.commute of-nat-inverse word-unat.Rep-inverse')
apply (*subst intvl-no-overflow-nat-conv*)
subgoal using *bound-n* **by** *simp*
subgoal using *bound-n bound-Suc*
by *auto* (*smt* (*verit*) *Abs-fnat-hom-add Abs-fnat-hom-mult CTypesDefs.ptr-add-def bound-n*
len-of-addr-card mult-of-nat-commute not-le of-int-of-nat-eq ptr-val.ptr-val-def unat-of-nat-len word-unat.Rep-inverse')
done
qed

lemma *ptr-span-no-overflow-indexes-disjoint*:
fixes $p::'a::mem\text{-}type\ ptr$
assumes *no-overflow*: $0 \notin \{ptr\text{-}val\ p..+\ n * size\text{-}of\ TYPE('a)\}$
assumes *i*: $i < n$
assumes *j*: $j < n$
assumes *neq*: $i \neq j$
shows $ptr\text{-}span\ (p +_p int\ i) \cap ptr\text{-}span\ (p +_p int\ j) = \{\}$
proof −
have *sz-a*: $0 < size\text{-}of\ (TYPE('a))$
by *simp*
from *no-overflow sz-a*
have *bound-n*: $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ (TYPE('a)) \leq addr\text{-}card$
by (*metis len-of-addr-card zero-not-in-intvl-no-overflow*)
from *bound-n i* **have** *bound-i*: $unat\ (ptr\text{-}val\ p) + i * size\text{-}of\ (TYPE('a)) < addr\text{-}card$
by (*metis linorder-not-le mult-le-cancel2 nat-add-left-cancel-le order-less-le-trans sz-nzero*)
from *bound-n j* **have** *bound-j*: $unat\ (ptr\text{-}val\ p) + j * size\text{-}of\ (TYPE('a)) < addr\text{-}card$
by (*metis linorder-not-le mult-le-cancel2 nat-add-left-cancel-le order-less-le-trans sz-nzero*)

```
  show ?thesis
    apply (subst intvl-no-overflow-nat-conv)
    subgoal using bound-i i bound-n
      apply (simp add: ptr-add-def)
      by (smt (verit, ccfv-SIG) len-of-addr-card less-le-mult mult-strict-right-mono
        of-nat-add of-nat-inverse of-nat-le-iff of-nat-less-iff of-nat-mult word-unat.Rep-inverse)
    apply (subst intvl-no-overflow-nat-conv)
    subgoal using bound-j j bound-n
      apply (simp add: ptr-add-def)
      by (smt (verit, ccfv-SIG) len-of-addr-card less-le-mult mult-strict-right-mono
        of-nat-add of-nat-inverse of-nat-le-iff of-nat-less-iff of-nat-mult word-unat.Rep-inverse)
    subgoal using bound-i bound-j i j neq
      by (auto simp add: ptr-add-def)
        (smt (verit, ccfv-SIG) len-of-addr-card less-le-mult mult-strict-right-mono
          of-nat-add of-nat-eq-iff of-nat-inverse of-nat-less-iff of-nat-mult
          order-le-less-trans word-unat.Rep-inverse)
    done
qed

lemma array-to-index-span:
  assumes x-in: x ∈ {ptr-val (p::'a::mem-type ptr)..+ n * size-of TYPE('a)}
  shows ∃ i. i < n ∧ x ∈ ptr-span (p +_p int i)
  using x-in
  apply (clarsimp simp add: intvl-def ptr-add-def)
  subgoal for k
    apply (rule exI [where x=k div size-of TYPE('a)])
    by (metis (no-types, opaque-lifting) less-mult-imp-div-less mod-div-decomp
        mod-less-divisor of-nat-add of-nat-mult sz-nzero)
  done

lemma array-to-index-span-exact:
  assumes x-in: x ∈ {ptr-val (p::'a::mem-type ptr)..+ n * size-of TYPE('a)}
  shows x ∈ ptr-span (p +_p int ((unat (x - ptr-val p)) div size-of TYPE('a)))
  using x-in
  apply (clarsimp simp add: intvl-def ptr-add-def)
  by (metis (no-types, opaque-lifting) mod-div-decomp mod-less-divisor of-nat-add
of-nat-mult
      sz-nzero word-unat.Rep-inverse)

lemma array-index-span-conv:
  {ptr-val (p::'a::mem-type ptr)..+ n * size-of TYPE('a)} = (⋃ i<n. ptr-span (p
+_p int i))
  apply standard
  subgoal
    apply clarsimp
    using array-to-index-span by blast
  subgoal
    apply clarsimp
   by (smt (verit, del-insts) CTypesDefs.ptr-add-def add.assoc intvlD intvlI mult.commute
```

nat-index-bound of-int-of-nat-eq of-nat-add of-nat-mult ptr-val.ptr-val-def)
done

lemma *fold-ptr-retyp-footprint*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n * size-of TYPE('a)*}
assumes *i*: *i* < *n*
assumes *x*: *x* ∈ *ptr-span* (*p* +$_p$ *int i*)
shows *fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [*0*..<*n*] *d x* =
(*True*, *snd* (*d x*) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE('a)*) (*unat* (*x* − *ptr-val* (*p* +$_p$ *int i*)))))
using *no-overflow i*
proof (*induct n arbitrary*: *d*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have *no-overflow*: *0* ∉ {*ptr-val p..+ Suc n * size-of TYPE('a)*} **by** *fact*
hence *no-overflow-n*: *0* ∉ {*ptr-val p..+n * size-of TYPE('a)*}
by (*meson intvl-start-le lessI mult-less-cancel2 order-less-le subsetD sz-nzero*)
from *ptr-span-no-overflow-split-last-disjoint* [*OF no-overflow*]
have *disj*: *ptr-span* (*p* +$_p$ *int n*) ∩ {*ptr-val p..+ n * size-of TYPE('a)*} = {} **.**
have *i*: *i* < *Suc n* **by** *fact*
show *?case*
proof (*cases i* = *n*)
case *True*
with *disj x* **have** *x-notin-n*: *x* ∉ {*ptr-val p..+ n * size-of TYPE('a)*} **by** *auto*
from *True x* **have** *x*: *x* ∈ *ptr-span* (*p* +$_p$ *int n*) **by** *simp*
show *?thesis*
apply *simp*
using *x*
apply (*simp add*: *ptr-retyp-split*)
apply (*subst fold-ptr-retyp-other* [*OF x-notin-n*])
apply (*simp add*: *True*)
done
next
case *False*
with *x ptr-span-no-overflow-indexes-disjoint* [*OF no-overflow i*, *of n*]
have *x-notin-n*: *x* ∉ *ptr-span* (*p* +$_p$ *int n*) **by** *blast*
from *False i* **have** *i* < *n* **by** *simp*
note *hyp* = *Suc.hyps* [*OF no-overflow-n this*]
show *?thesis*
apply *simp*
using *x-notin-n*
apply (*simp add*: *ptr-retyp-split*)
apply (*simp add*: *hyp*)
done

qed
qed

lemma *ptr-retyp-idem*:
apply (*rule ext*)
apply (*clarsimp simp add*: *ptr-retyp-split*)
by (*metis* (*no-types*, *lifting*) *map-add-assoc map-add-dom-eq*)

lemma *fold-ptr-retyp-d-empty*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n* ∗ *size-of TYPE*(*'a*)}
assumes *i*: *i* < *n*
assumes *x*: *x* ∈ *ptr-span* (*p* +*p int i*)
shows *fold* (λ*i*. *ptr-retyp* (*p* +*p int i*)) [*0..*<*n*] *d x* =
(*True*, *snd* (*d x*) ++ *snd* (*ptr-retyp* (*p* +*p int i*) *empty-htd x*))
apply (*simp add*: *fold-ptr-retyp-footprint* [*OF no-overflow i x*])
apply (*simp add*: *ptr-retyp-footprint* [*OF x*])
done

lemma *fold-ptr-retyp-eq-fst*:
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n* ∗ *size-of TYPE*(*'a*)}
shows*fst* (*fold* (λ*i*. *ptr-retyp* (*p* +*p int i*)) [*0..*<*n*] *d x*) =
(*if x* ∈ {*ptr-val* (*p*::*'a*::*mem-type ptr*)*..+ n* ∗ *size-of TYPE*(*'a*)} *then True*
else fst (*d x*))
proof (*cases x* ∈ {*ptr-val* (*p*::*'a*::*mem-type ptr*)*..+ n* ∗ *size-of TYPE*(*'a*)})
case *True*
from *array-to-index-span* [*OF True*]
obtain *i* **where** *i*: *i* < *n* **and** *x-in*: *x* ∈ *ptr-span* (*p* +*p int i*)
by *blast*

from *fold-ptr-retyp-footprint* [*OF no-overflow i x-in*] *True*
show *?thesis* **by** *simp*
next
case *False*
from *fold-ptr-retyp-other* [*OF False*] *False* **show** *?thesis* **by** *simp*
qed

lemma *fold-ptr-retyp-valid-footprint-disjoint2*:
assumes *no-overflow*: *0* ∉ {*ptr-val q..+ n* ∗ *size-of TYPE*(*'b*)}
shows ⟦*valid-footprint* (*fold* (λ*i*. *ptr-retyp* ((*q*::*'b*::*mem-type ptr*) +*p int i*)) [*0..*<*n*]
d) *p s*;
{*p..+size-td s*} ∩ {*ptr-val q ..+ n* ∗ *size-of TYPE*(*'b*)} = {} ⟧
⟹ *valid-footprint d p s*
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply (*drule spec*, *drule* (*1*) *mp*)
apply(*subgoal-tac p* + *of-nat y* ∈ {*p..+size-td s*})
apply (*subst* (*asm*) *fold-ptr-retyp-other*)

apply *clarsimp*
apply *fast*
apply (*clarsimp simp add*: *fold-ptr-retyp-eq-fst* [*OF no-overflow*] *split*: *if-split-asm*)
apply *fast*
apply (*erule intvlI*)
done

lemma *ptr-retyp-disjoint2*:
⟦*ptr-retyp* (*p*::'*a*::*mem-type ptr*) *d*,*g* ⊨$_t$ *q*;
ptr-span p ∩ *ptr-span q* = {} ⟧
⟹ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
apply(*clarsimp simp*: *h-t-valid-def*)
apply(*erule ptr-retyp-valid-footprint-disjoint2*)
apply(*simp add*: *size-of-def*)
apply *fast*
done

lemma *fold-ptr-retyp-disjoint2*:
fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)}
shows ⟦*fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*,*g* ⊨$_t$ *q*;
{*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q*= {} ⟧
⟹ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
apply(*clarsimp simp*: *h-t-valid-def*)
apply(*erule fold-ptr-retyp-valid-footprint-disjoint2* [*OF no-overflow*])
apply(*simp add*: *size-of-def*)
apply *fast*
done

lemma *ptr-retyp-disjoint-iff*:
{*ptr-val p*..+*size-of TYPE*('*a*)} ∩ {*ptr-val q*..+*size-of TYPE*('*b*)} = {}
⟹ *ptr-retyp* (*p*::'*a*::*mem-type ptr*) *d*,*g* ⊨$_t$ *q* = *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
apply *standard*
apply (*erule* (*1*) *ptr-retyp-disjoint2*)
apply (*erule* (*1*) *ptr-retyp-disjoint*)
done

lemma (**in** *mem-type*) *ptr-force-type-valid-footprint-disjoint*:
⟦ *valid-footprint d p s*; {*p*..+*size-td s*} ∩ {*ptr-val q*..+*size-of TYPE*('*a*)} = {} ⟧
⟹ *valid-footprint* (*ptr-force-type* (*q*::'*a ptr*) *d*) *p s*
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply((*subst ptr-force-type-d*; *clarsimp*), *fastforce intro*: *intvlI*)+
done

lemma *ptr-force-type-disjoint*:
⟦ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*); {*ptr-val p*..+*size-of TYPE*('*a*)} ∩
{*ptr-val q*..+*size-of TYPE*('*b*)} = {} ⟧ ⟹
ptr-force-type (*p*::'*a*::*mem-type ptr*) *d*,*g* ⊨$_t$ *q*

apply(*clarsimp simp*: *h-t-valid-def*)
apply(*erule ptr-force-type-valid-footprint-disjoint*)
apply(*fastforce simp*: *size-of-def*)
done

lemma *ptr-force-type-disjoint2*:
⟦*ptr-force-type* (*p*::'*a*::*mem-type ptr*) *d*,*g* ⊨$_t$ *q*;
ptr-span p ∩ *ptr-span q* = {} ⟧
⟹ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
apply(*clarsimp simp*: *h-t-valid-def*)
apply(*erule ptr-force-type-valid-footprint-disjoint2*)
apply(*simp add*: *size-of-def*)
apply *fast*
done

lemma *ptr-force-type-disjoint-iff*:
{*ptr-val p..+size-of TYPE*('*a*)} ∩ {*ptr-val q..+size-of TYPE*('*b*)} = {}
⟹ *ptr-force-type* (*p*::'*a*::*mem-type ptr*) *d*,*g* ⊨$_t$ *q* = *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
apply *standard*
apply (*erule* (*1*) *ptr-force-type-disjoint2*)
apply (*erule* (*1*) *ptr-force-type-disjoint*)
done

lemma *fold-ptr-force-type-valid-footprint-disjoint2*:
assumes *no-overflow*: *0* ∉ {*ptr-val q..+ n* ∗ *size-of TYPE*('*b*)}
shows ⟦*valid-footprint* (*fold* (λ*i*. *ptr-force-type* ((*q*::'*b*::*mem-type ptr*) +$_p$ *int i*))
[*0*..<*n*] *d*) *p s*;
{*p..+size-td s*} ∩ {*ptr-val q ..+ n* ∗ *size-of TYPE*('*b*)} = {} ⟧
⟹ *valid-footprint d p s*
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply (*drule spec*, *drule* (*1*) *mp*)
apply(*subgoal-tac p* + *of-nat y* ∈ {*p..+size-td s*})
apply (*subst* (*asm*) *fold-ptr-force-type-other*)
apply *clarsimp*
apply *fast*
apply (*simp add*: *disjoint-iff fold-ptr-force-type-other*)
apply (*erule intvlI*)
done

lemma *fold-ptr-force-type-disjoint2*:
fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n* ∗ *size-of TYPE*('*a*)}
shows ⟦*fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*,*g* ⊨$_t$ *q*;
{*ptr-val p..+ n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q*= {} ⟧
⟹ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
apply(*clarsimp simp*: *h-t-valid-def*)
apply(*erule fold-ptr-force-type-valid-footprint-disjoint2* [*OF no-overflow*])
apply(*simp add*: *size-of-def*)

apply *fast*
done

lemma *fold-ptr-retyp-valid-footprint-disjoint*:
⟦ *valid-footprint d p s*; {*p*..+*size-td s*} ∩ {*ptr-val q* ..+ *n* ∗ *size-of TYPE*('*b*)} = {} ⟧
⟹ *valid-footprint* (*fold* (λ*i*. *ptr-retyp* ((*q*::'*b*:: *mem-type ptr*) +$_p$ *int i*)) [0..<*n*] *d*) *p s*
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply((*subst fold-ptr-retyp-other*; *clarsimp*), *fastforce intro*: *intvlI*)+
done

lemma *fold-ptr-force-type-valid-footprint-disjoint*:
⟦ *valid-footprint d p s*; {*p*..+*size-td s*} ∩ {*ptr-val q* ..+ *n* ∗ *size-of TYPE*('*b*)} = {} ⟧
⟹ *valid-footprint* (*fold* (λ*i*. *ptr-force-type* ((*q*::'*b*:: *mem-type ptr*) +$_p$ *int i*)) [0..<*n*] *d*) *p s*
apply(*clarsimp simp*: *valid-footprint-def Let-def*)
apply((*subst fold-ptr-force-type-other*; *clarsimp*), *fastforce intro*: *intvlI*)+
done

lemma *fold-ptr-retyp-disjoint*:
fixes *p*::'*a*::*mem-type ptr*
shows ⟦ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*); {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {} ⟧ ⟹
fold (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [0..<*n*] *d*,*g* ⊨$_t$ *q*
apply(*clarsimp simp*: *h-t-valid-def*)
apply(*erule fold-ptr-retyp-valid-footprint-disjoint*)
apply(*fastforce simp*: *size-of-def*)
done

lemma *fold-ptr-force-type-disjoint*:
fixes *p*::'*a*::*mem-type ptr*
shows ⟦ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*); {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {} ⟧ ⟹
fold (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [0..<*n*] *d*,*g* ⊨$_t$ *q*
apply(*clarsimp simp*: *h-t-valid-def*)
apply(*erule fold-ptr-force-type-valid-footprint-disjoint*)
apply(*fastforce simp*: *size-of-def*)
done

lemma *fold-ptr-retyp-disjoint-iff*:
fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)}
shows {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {}
⟹ *fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [0..<*n*] *d*,*g* ⊨$_t$ *q* = *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
apply *standard*
apply (*erule* (*1*) *fold-ptr-retyp-disjoint2* [*OF no-overflow*])

apply (*erule* (*1*) *fold-ptr-retyp-disjoint*)
done

lemma *fold-ptr-force-type-disjoint-iff*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n * size-of TYPE('a)*}
shows {*ptr-val p..+ n * size-of TYPE('a)*} ∩ *ptr-span q* = {}
⟹ *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0..<n*] *d*,*g* ⊨$_t$ *q* = *d*,*g* ⊨$_t$ (*q*::*'b*::*mem-type ptr*)
apply *standard*
apply (*erule* (*1*) *fold-ptr-force-type-disjoint2* [*OF no-overflow*])
apply (*erule* (*1*) *fold-ptr-force-type-disjoint*)
done

lemma *stack-alloc-preserves-typing*:
fixes *q*::*'b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* 𝒮 (*TYPE*(*'a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*(*'b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *d* ⊨$_t$ *q*
shows *d'* ⊨$_t$ *q*
proof −
from *stack-alloc* **obtain**
d': *d'* = *ptr-force-type p d*
by (*cases rule*: *stack-alloc-cases*) *auto*

from *stack-alloc-disjoint* [*OF stack-alloc no-stack typed*]
have *ptr-span p* ∩ *ptr-span q* = {} **.**
from *ptr-force-type-disjoint-iff* [*OF this*, **where** *d*=*d* **and** *g*=*c-guard*] *typed*
show *?thesis*
by (*simp add*: *d'*)
qed

lemma *stack-allocs-preserves-typing*:
fixes *q*::*'b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 (*TYPE*(*'a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*(*'b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *d* ⊨$_t$ *q*
shows *d'* ⊨$_t$ *q*
proof −
from *stack-alloc* **obtain**
d': *d'* = *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0..<n*] *d* **and**
no-overflow: *0* ∉ {*ptr-val p ..+ n * size-of TYPE('a)*}
by (*cases rule*: *stack-allocs-cases*) *auto*

from *stack-allocs-disjoint* [*OF stack-alloc no-stack typed*]
have {*ptr-val p..+n * size-of TYPE('a)*} ∩ *ptr-span q* = {} **.**
from *fold-ptr-force-type-disjoint-iff* [*OF no-overflow this*, **where** *d*=*d* **and** *g*=*c-guard*] *typed* **show** *?thesis*
by (*simp add*: *d'*)

qed

lemma *h-t-valid-valid-footprint*: $d,g \models_t p \Longrightarrow$ *valid-footprint d* (*ptr-val* (*p*::'*a*::*c-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
by (*simp add*: *h-t-valid-def*)

lemma *stack-alloc-preserves-root-ptr-valid*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *root-ptr-valid d q*
shows *root-ptr-valid d' q*
proof −
from *stack-alloc* **obtain**
d': *d'* = *ptr-force-type p d*
by (*cases rule*: *stack-alloc-cases*) *auto*

from *typed* **have** *typed-q*: *d* $\models_t$ *q*
by (*simp add*: *root-ptr-valid-h-t-valid*)

from *stack-alloc-disjoint* [*OF stack-alloc no-stack this*]
have *disj*: *ptr-span p* ∩ *ptr-span q* = {} **.**
from *stack-alloc-preserves-typing* [*OF stack-alloc no-stack typed-q*]
have *typed'*: *d'* $\models_t$ *q* **.**
hence *valid-fp*: *valid-footprint d'* (*ptr-val q*) (*typ-uinfo-t TYPE*('*b*))
by (*simp add*: *h-t-valid-valid-footprint*)

show *?thesis*
apply (*simp add*: *root-ptr-valid-def valid-root-footprint-valid-footprint-dom-conv valid-fp*
h-t-valid-c-guard [*OF typed'*])
apply (*simp add*: *d'*)
by (*metis* (*no-types*, *lifting*) *disj disjoint-iff intvlI ptr-force-type-d root-ptr-valid-def*

size-of-def typ-uinfo-size typed valid-root-footprint-dom-typing)
qed

lemma *stack-allocs-preserves-root-ptr-valid*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *root-ptr-valid d q*
shows *root-ptr-valid d' q*
proof −
from *stack-alloc* **obtain**
d': *d'* = *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d* **and**
no-overflow: *0* ∉ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)}
by (*cases rule*: *stack-allocs-cases*) *auto*

from *typed* **have** *typed-q*: $d \models_t q$
by (*simp add*: *root-ptr-valid-h-t-valid*)

from *stack-allocs-disjoint* [*OF stack-alloc no-stack this*]
have *disj*: $\{ptr\text{-}val\ p..+n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}$.
from *stack-allocs-preserves-typing* [*OF stack-alloc no-stack typed-q*]
have *typed'*: $d' \models_t q$.
hence *valid-fp*: *valid-footprint d' (ptr-val q) (typ-uinfo-t TYPE('b))*
by (*simp add*: *h-t-valid-valid-footprint*)

show *?thesis*
apply (*simp add*: *root-ptr-valid-def valid-root-footprint-valid-footprint-dom-conv valid-fp*
h-t-valid-c-guard [*OF typed'*])
apply (*simp add*: *d'*)
using *disj fold-ptr-force-type-other*
by (*smt* (*verit*) *d' disjoint-iff dom-list-map root-ptr-valid-def s-footprintD s-footprintI2 size-of-def stack-alloc stack-allocs-cases typ-uinfo-size typed valid-root-footprint-def*)
qed

lemma *stack-alloc-root-ptr-valid-new-cases*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: $(p,\ d') \in stack\text{-}alloc\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
assumes *root-ptr-valid d' q*
shows $(ptr\text{-}val\ p = ptr\text{-}val\ q \wedge typ\text{-}uinfo\text{-}t\ (TYPE('b)) = typ\text{-}uinfo\text{-}t\ (TYPE('a)))$
$\vee$ *root-ptr-valid d q*
by (*metis Ptr-ptr-val assms*(*2*) *ptr.inject root-ptr-valid-def stack-alloc stack-alloc-cases*
valid-root-footprint-ptr-force-type-iff wf-size-desc-typ-uinfo-t-simp)

lemma *valid-root-footprints-no-overlap*:
assumes *valid-root-footprint d a1 t1*
assumes *valid-root-footprint d a2 t2*
assumes *t1* $\neq$ *t2*
shows $\{a1\ ..+\ size\text{-}td\ t1\} \cap \{a2\ ..+\ size\text{-}td\ t2\} = \{\}$
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *valid-root-footprint-type-neq-disjoint* **by** *presburger*

lemma *root-ptr-valid-neq-disjoint*:
$[\![$ *root-ptr-valid d* (*p*::'*a*::*c-type ptr*);
root-ptr-valid d (*q*::'*b*::*c-type ptr*);
ptr-val p $\neq$ *ptr-val q* $]\!] \Longrightarrow$
$\{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\} \cap$
$\{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}$
apply (*clarsimp simp only*: *size-of-tag* [*symmetric*])
by (*metis boolean-algebra.conj-ac*(*2*) *h-t-valid-valid-footprint intvl-inter order-antisym-conv root-ptr-valid-def root-ptr-valid-h-t-valid valid-footprint-neq-nmem valid-root-footprint-overlap-sub-typ*)

lemma *root-ptr-valid-same-type-neq-disjoint*:
root-ptr-valid d p $\Longrightarrow$ *root-ptr-valid d q* $\Longrightarrow$ $p \neq q = (ptr\text{-}span\ p \cap ptr\text{-}span\ q = \{\})$
apply *standard*
apply (*simp add*: *root-ptr-valid-neq-disjoint*)
by (*metis empty-iff inf.idem intvlI root-ptr-valid-def size-of-tag valid-root-footprint-def*)

lemma *cvalid-same-type-neq-disjoint*:
$d \models_t p \Longrightarrow d \models_t q \Longrightarrow p \neq q = (ptr\text{-}span\ p \cap ptr\text{-}span\ q = \{\})$
apply *standard*
apply (*simp add*: *h-t-valid-neq-disjoint peer-typ-not-field-of*)
by (*metis disjoint-iff h-t-valid-valid-footprint intvl-self size-of-tag valid-footprint-def*)

lemma *root-ptr-valid-type-neq-disjoint*:
$[\![$ *root-ptr-valid d* ($p::'a::c\text{-}type\ ptr$);
root-ptr-valid d ($q::'b::c\text{-}type\ ptr$);
typ-uinfo-t TYPE($'a$) $\neq$ *typ-uinfo-t TYPE*($'b$) $]\!] \Longrightarrow$
$\{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\} \cap$
$\{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}$
apply (*subgoal-tac ptr-val p* $\neq$ *ptr-val q*)
apply (*rule root-ptr-valid-neq-disjoint*, *auto*)[*1*]
by (*metis disjoint-iff intvlI root-ptr-valid-def sub-tag-antisym valid-footprint-def valid-root-footprint-overlap-sub-typ valid-root-footprint-valid-footprint-dom-conv*)

lemma *valid-root-footprints-cases*:
assumes *valid-root-footprint d a1 t1*
assumes *valid-root-footprint d a2 t2*
shows $(t1 = t2 \wedge a1 = a2) \vee (\{a1\ ..+\ size\text{-}td\ t1\} \cap \{a2\ ..+\ size\text{-}td\ t2\} = \{\})$
using *assms valid-root-footprint-neq-disjoint valid-root-footprint-type-neq-disjoint*
by *blast*

lemma *root-ptr-valid-cases*:
fixes $p::'a::mem\text{-}type\ ptr$
fixes $q::'b::mem\text{-}type\ ptr$
assumes *root-p*: *root-ptr-valid d p*
assumes *root-q*: *root-ptr-valid d q*
shows $(ptr\text{-}val\ p = ptr\text{-}val\ q \wedge typ\text{-}uinfo\text{-}t\ (TYPE('a)) = typ\text{-}uinfo\text{-}t\ (TYPE('b)))$
$\vee$
$(ptr\text{-}span\ p \cap ptr\text{-}span\ q) = \{\}$
using *root-p root-ptr-valid-neq-disjoint root-ptr-valid-type-neq-disjoint root-q* **by** *blast*

lemma *root-ptr-valid-casesE* [*consumes 2*]:
fixes $p::'a::mem\text{-}type\ ptr$
fixes $q::'b::mem\text{-}type\ ptr$
assumes *root-p*: *root-ptr-valid d p*

assumes *root-q*: *root-ptr-valid d q*
assumes *same*: (*ptr-val q* = *ptr-val p* ∧ *typ-uinfo-t* (*TYPE*('*a*)) = *typ-uinfo-t* (*TYPE*('*b*))) ⟹ *P*
assumes *disj*: *ptr-span p* ∩ *ptr-span q* = {} ⟹ *P*
shows *P*
using *root-ptr-valid-cases* [*OF root-p root-q*] *same disj* **by** *auto*

lemma *stack-allocs-root-ptr-valid-new-cases*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 (*TYPE*('*a*::*mem-type*)) *d*
assumes *root-ptr-valid d' q*
shows (∃ *i*<*n*. *ptr-val q* = *ptr-val* (*p* +$_p$ *int i*) ∧ *typ-uinfo-t* (*TYPE*('*b*)) = *typ-uinfo-t* (*TYPE*('*a*))) ∨ *root-ptr-valid d q*
proof (*cases* {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {})
case *True*
with *stack-alloc* **show** *?thesis*
by (*smt* (*verit*, *best*) *assms*(*2*) *disjoint-iff fold-ptr-force-type-other root-ptr-valid-domain' stack-allocs-cases*)
next
case *False*
with *stack-alloc array-to-index-span* **show** *?thesis*
by (*smt* (*verit*) *assms*(*2*) *disjoint-iff root-ptr-valid-neq-disjoint root-ptr-valid-type-neq-disjoint stack-allocs-cases*)
qed

lemma *stack-alloc-root-ptr-valid-same*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* 𝒮 (*TYPE*('*a*::*mem-type*)) *d*
assumes *addr-eq*: *ptr-val p* = *ptr-val q*
assumes *match*: *typ-uinfo-t* (*TYPE*('*b*)) = *typ-uinfo-t* (*TYPE*('*a*))
shows *root-ptr-valid d' q*
proof (*cases root-ptr-valid d' q*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *stack-alloc* **have** *root-ptr-valid d' p* **by** (*rule stack-alloc-cases*)
with *addr-eq match* **show** *?thesis*
apply (*clarsimp simp add*: *root-ptr-valid-def*)
apply (*simp add*: *c-guard-def c-null-guard-def ptr-aligned-def*)
apply (*clarsimp simp add*: *align-of-def align-td-uinfo*[*symmetric*] *size-of-def*)
by (*metis typ-uinfo-size*)
qed

lemma *stack-allocs-root-ptr-valid-same*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 (*TYPE*('*a*::*mem-type*)) *d*
assumes *i*: *i* < *n*
assumes *addr-eq*: *ptr-val q* = *ptr-val* (*p* +$_p$ *int i*)

assumes *match*: *typ-uinfo-t* (*TYPE*('*b*)) = *typ-uinfo-t* (*TYPE*('*a*))
shows *root-ptr-valid d' q*
proof (*cases root-ptr-valid d' q*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *stack-alloc* **have** *root-ptr-valid d'* (*p* +$_p$ *int i*)
apply (*rule stack-allocs-cases*)
using *i*
by *auto*

with *addr-eq match* **show** *?thesis*
apply (*clarsimp simp add*: *root-ptr-valid-def*)
apply (*simp add*: *c-guard-def c-null-guard-def ptr-aligned-def*)
apply (*clarsimp simp add*: *align-of-def align-td-uinfo*[*symmetric*] *size-of-def*)
by (*metis typ-uinfo-size*)
qed

lemma *stack-alloc-root-ptr-valid-other*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *valid-d*: *root-ptr-valid d q*
assumes *non-stack*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
shows *root-ptr-valid d' q*
proof (*cases root-ptr-valid d' q*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *stack-alloc*
show *?thesis*
apply (*rule stack-alloc-cases*)
using *False valid-d non-stack*
using *stack-alloc stack-alloc-preserves-root-ptr-valid* **by** *blast*
qed

lemma *stack-allocs-root-ptr-valid-other*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *valid-d*: *root-ptr-valid d q*
assumes *non-stack*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
shows *root-ptr-valid d' q*
proof (*cases root-ptr-valid d' q*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *stack-alloc*

show *?thesis*
apply (*rule stack-allocs-cases*)
using *False valid-d non-stack*
using *stack-alloc stack-allocs-preserves-root-ptr-valid* **by** *blast*
qed

lemma *stack-alloc-root-ptr-valid-cases*:
fixes *q*::*'b*::*mem-type ptr*
assumes *stack-alloc*: $(p,\ d') \in$ *stack-alloc* $\mathcal{S}$ $(TYPE('a::mem\text{-}type))$ d
assumes *non-stack-byte*: *typ-uinfo-t* $(TYPE('b)) \neq$ *typ-uinfo-t* $(TYPE(stack\text{-}byte))$
shows *root-ptr-valid* d' $q \longleftrightarrow$
(*ptr-val* p = *ptr-val* $q \wedge$ *typ-uinfo-t* $(TYPE('b))$ = *typ-uinfo-t* $(TYPE('a))) \vee$
root-ptr-valid d q

using *stack-alloc non-stack-byte*
stack-alloc-root-ptr-valid-new-cases stack-alloc-root-ptr-valid-other stack-alloc-root-ptr-valid-same
by *blast*

lemma *stack-allocs-root-ptr-valid-cases*:
fixes *q*::*'b*::*mem-type ptr*
assumes *stack-alloc*: $(p,\ d') \in$ *stack-allocs* n $\mathcal{S}$ $(TYPE('a::mem\text{-}type))$ d
assumes *non-stack-byte*: *typ-uinfo-t* $(TYPE('b)) \neq$ *typ-uinfo-t* $(TYPE(stack\text{-}byte))$
shows *root-ptr-valid* d' $q \longleftrightarrow$
$(\exists\, i{<}n.$ *ptr-val* q = *ptr-val* $(p +_p$ *int* $i) \wedge$ *typ-uinfo-t* $(TYPE('b))$ = *typ-uinfo-t* $(TYPE('a))) \vee$
root-ptr-valid d q

using *stack-alloc non-stack-byte*
stack-allocs-root-ptr-valid-new-cases stack-allocs-root-ptr-valid-other stack-allocs-root-ptr-valid-same
by *metis*

lemma *stack-alloc-root-ptr-valid-same-type-cases*:
assumes *stack-alloc*: $(p,\ d') \in$ *stack-alloc* $\mathcal{S}$ $(TYPE('a::mem\text{-}type))$ d
shows *root-ptr-valid* d' $q \longleftrightarrow p = q \vee$ *root-ptr-valid* d q
by (*metis ptr-val-inj stack-alloc stack-alloc-cases stack-alloc-root-ptr-valid-cases*)

lemma *stack-allocs-root-ptr-valid-same-type-cases*:
assumes *stack-alloc*: $(p,\ d') \in$ *stack-allocs* n $\mathcal{S}$ $(TYPE('a::mem\text{-}type))$ d
shows *root-ptr-valid* d' $q \longleftrightarrow (\exists\, i{<}n.\ q = (p +_p$ *int* $i) \vee$ *root-ptr-valid* d $q)$
using *stack-alloc stack-allocs-cases stack-allocs-root-ptr-valid-cases*
by (*metis ptr-val-inj*)

lemma *root-ptr-valid-valid-root-footprint*:
root-ptr-valid d $(p::'a\ ptr) \Longrightarrow$ *valid-root-footprint* d (*ptr-val* p) (*typ-uinfo-t* $TYPE('a::c\text{-}type)$)
by (*simp add*: *root-ptr-valid-def*)

definition

allocated-ptr:: *addr set* ⇒ *'a*::*mem-type itself* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* ⇒ *'a ptr* **where**
‹*allocated-ptr* $\mathcal{S}$ *T d d'* = (*THE p*. (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ *TYPE*(*'a*) *d*)›

definition
allocated-ptrs:: *nat* ⇒ *addr set* ⇒ *'a*::*mem-type itself* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* ⇒ *'a ptr* **where**
‹*allocated-ptrs n* $\mathcal{S}$ *T d d'* = (*THE p*. (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*(*'a*) *d*)›

lemma *allocated-ptr-allocated-ptrs-def*: *allocated-ptr* = *allocated-ptrs 1*
by (*simp add*: *stack-alloc-stack-allocs-conv allocated-ptr-def allocated-ptrs-def fun-eq-iff*)

abbreviation (*input*)
cptr-type :: (*'a* :: *c-type*) *ptr* ⇒ *'a itself*
where
cptr-type p ≡ *TYPE*(*'a*)

lemma *h-t-valid-guard-subst*:
⟦ *d*,*g* ⊨$_t$ *p*; *g' p* ⟧ ⟹ *d*,*g'* ⊨$_t$ *p*
apply(*simp add*:*h-t-valid-def*)
done

lemma *h-t-valid-ptr-retyp-eq*:
¬ *cptr-type p* <$_\tau$ *cptr-type p'* ⟹ *h-t-valid* (*ptr-retyp p td*) *g p'*
= (*if ptr-span* (*p*::*'a*::*mem-type ptr*) ∩ *ptr-span* (*p'*::*'b*::*mem-type ptr*) = {} *then h-t-valid td g p'*
else field-of-t p' p ∧ *g p'*)
apply (*clarsimp simp*: *ptr-retyp-disjoint-iff split*: *if-split*)
apply (*cases g p'*)
apply (*rule iffI*)
apply (*rule ccontr*, *drule h-t-valid-neq-disjoint*, *rule ptr-retyp-h-t-valid*, *simp*+)
apply (*simp add*: *Int-commute*)
apply (*clarsimp simp*: *field-of-t-def field-of-def*)
apply (*drule sub-h-t-valid*[**where** *p*=*p*, *rotated*], *rule ptr-retyp-h-t-valid*, *simp*, *simp*)
apply (*erule*(*1*) *h-t-valid-guard-subst*)
apply (*simp add*: *h-t-valid-def*)
done

lemma *field-of-t-refl*:
field-of-t p p' = (*p* = *p'*)
apply (*safe*, *simp-all add*: *field-of-t-def*)
apply (*simp add*: *field-of-def*)
apply (*drule td-set-size-lte*)
apply (*simp add*: *unat-eq-0*)
done

lemma *ptr-retyp-same-cleared-region*:
fixes *p* :: *'a* :: *mem-type ptr* **and** *p'* :: *'a* :: *mem-type ptr*

assumes *ht*: *ptr-retyp p td*, *g* $\models_t$ *p'*
shows *p* = *p'* ∨ {*ptr-val p*..+ *size-of TYPE*('*a*)} ∩ {*ptr-val p'* ..+ *size-of TYPE*('*a*)} = {}
using *ht*
by (*simp add*: *h-t-valid-ptr-retyp-eq*[**where** *p*=*p* **and** *p'*=*p'*] *field-of-t-refl* *split*: *if-split-asm*)

lemma (**in** *mem-type*) *ptr-force-type-h-t-valid*:
g p ⟹ *ptr-force-type p d*,*g* $\models_t$ (*p*::'*a ptr*)
by (*simp add*: *h-t-valid-def ptr-force-type-valid-footprint*)

lemma *h-t-valid-ptr-force-type-eq*:
¬ *cptr-type p* <$_\tau$ *cptr-type p'* ⟹ *h-t-valid* (*ptr-force-type p td*) *g p'*
= (*if ptr-span* (*p*::'*a*::*mem-type ptr*) ∩ *ptr-span* (*p'*::'*b*::*mem-type ptr*) = {} *then* *h-t-valid td g p'*
else field-of-t p' p ∧ *g p'*)
apply (*clarsimp simp*: *ptr-force-type-disjoint-iff split*: *if-split*)
apply (*cases g p'*)
apply (*rule iffI*)
apply (*rule ccontr*, *drule h-t-valid-neq-disjoint*, *rule ptr-force-type-h-t-valid*, *simp*+)
apply (*simp add*: *Int-commute*)
apply (*clarsimp simp*: *field-of-t-def field-of-def*)
apply (*drule sub-h-t-valid*[**where** *p*=*p*, *rotated*], *rule ptr-force-type-h-t-valid*, *simp*, *simp*)
apply (*erule*(*1*) *h-t-valid-guard-subst*)
apply (*simp add*: *h-t-valid-def*)
done

lemma *ptr-force-type-same-cleared-region*:
fixes *p* :: '*a* :: *mem-type ptr* **and** *p'* :: '*a* :: *mem-type ptr*
assumes *ht*: *ptr-force-type p td*, *g* $\models_t$ *p'*
shows *p* = *p'* ∨ {*ptr-val p*..+ *size-of TYPE*('*a*)} ∩ {*ptr-val p'* ..+ *size-of TYPE*('*a*)} = {}
using *ht*
by (*simp add*: *h-t-valid-ptr-force-type-eq*[**where** *p*=*p* **and** *p'*=*p'*] *field-of-t-refl* *split*: *if-split-asm*)

lemma *stack-alloc-unique*:
assumes *p*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *q*: (*q*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
shows *p* = *q*
proof −
from *p* **obtain** *p-props*:
typ-uinfo-t (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
ptr-aligned p
c-guard p
root-ptr-valid d' p

```
    d' = ptr-force-type p d
    by (cases rule: stack-alloc-cases) auto

  from q obtain q-props:
    typ-uinfo-t (TYPE('a)) ≠ typ-uinfo-t (TYPE(stack-byte))
     ∀ a ∈ ptr-span q. root-ptr-valid d (PTR (stack-byte) a)
    ptr-aligned q
    c-guard q
    root-ptr-valid d' q and
    d' = ptr-force-type q d
    by (cases rule: stack-alloc-cases) auto

  show ?thesis
  proof (cases ptr-val p = ptr-val q)
    case True
    then show ?thesis by simp
  next
    case False
    with p-props q-props show ?thesis
      by (metis disj-ptr-span-ptr-neq[unfolded disjnt-def] ptr-force-type-disjoint2
       ptr-force-type-same-cleared-region q root-ptr-valid-h-t-valid stack-alloc-disjoint)
  qed
qed

lemma stack-allocs-unique:
  assumes p: (p, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
  assumes q: (q, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
  shows p = q
proof −
  from p obtain p-props:

    0 ∉ {ptr-val p ..+ n * size-of TYPE('a)}
    typ-uinfo-t (TYPE('a)) ≠ typ-uinfo-t (TYPE(stack-byte))
    ∀ a ∈ {ptr-val p ..+ n * size-of TYPE('a)}. root-ptr-valid d (PTR (stack-byte)
a)
    ptr-aligned p c-guard p root-ptr-valid d' p
    d' = fold (λi. ptr-force-type (p +p int i)) [0..<n] d
    by (cases rule: stack-allocs-cases) auto

  from q obtain q-props:
    0 < n
    0 ∉ {ptr-val q ..+ n * size-of TYPE('a)}
    typ-uinfo-t (TYPE('a)) ≠ typ-uinfo-t (TYPE(stack-byte))
    ∀ a ∈ {ptr-val q ..+ n * size-of TYPE('a)}. root-ptr-valid d (PTR (stack-byte)
a)
    ptr-aligned q c-guard q root-ptr-valid d' q
    d' = fold (λi. ptr-force-type (q +p int i)) [0..<n] d
    by (cases rule: stack-allocs-cases) auto
```

show *?thesis*
proof (*cases ptr-val p = ptr-val q*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
with *p-props q-props* **show** *?thesis*
using *disj-ptr-span-ptr-neq fold-ptr-retyp-disjoint2 ptr-force-type-same-cleared-region*

q p root-ptr-valid-h-t-valid stack-allocs-disjoint
by (*smt* (*verit*, *ccfv-threshold*) *disjoint-iff fold-ptr-force-type-disjoint2 inf.idem*

inf.order-iff intvl-inter intvl-no-overflow-nat len-of-addr-card mem-type-self

order-antisym-conv root-ptr-valid-cases stack-allocs-contained
stack-allocs-ptr-force-type zero-not-in-intvl-no-overflow)
qed
qed

lemma *stack-alloc-allocated-ptr*:
(*p*, *d'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*) *d* ⟹ *allocated-ptr* 𝒮 *TYPE*('*a*::*mem-type*) *d*
d' = *p*
apply (*simp add*: *allocated-ptr-def*)
apply (*rule theI2*)
apply (*assumption*)
apply (*erule* (*1*) *stack-alloc-unique*)
apply (*erule* (*1*) *stack-alloc-unique*)
done

lemma *stack-allocs-allocated-ptrs*:
(*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*) *d* ⟹ *allocated-ptrs n* 𝒮 *TYPE*('*a*::*mem-type*)
d d' = *p*
apply (*simp add*: *allocated-ptrs-def*)
apply (*rule theI2*)
apply (*assumption*)
apply (*erule* (*1*) *stack-allocs-unique*)
apply (*erule* (*1*) *stack-allocs-unique*)
done

lemma *null-not-stack-free*: *0* ∉ *stack-free d*
by (*simp add*: *root-ptr-valid-def stack-free-def*)

lemma *stack-alloc-stack-subset-stack-free*:
(*p*, *d'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹
ptr-span p ⊆ *stack-free d*
by (*metis mem-Collect-eq stack-alloc-cases stack-free-def subsetI*)

lemma *stack-allocs-stack-subset-stack-free'*:
(*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹ *i* < *n* ⟹
ptr-span (*p* +$_p$ *int i*) ⊆ *stack-free d*
using *stack-allocs-cases*
by (*smt* (*verit*, *best*) *mem-Collect-eq stack-allocs-contained stack-free-def subsetD subsetI*)

lemma *stack-allocs-stack-subset-stack-free*:
(*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹
{*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)} ⊆ *stack-free d*
by (*metis mem-Collect-eq stack-allocs-cases stack-free-def subsetI*)

lemma *stack-alloc-stack-free-mono*:
assumes *sub*: *stack-free d1* ⊆ *stack-free d2*
assumes *alloc-d1*: (*p*, *d1'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*::*mem-type*) *d1*
shows ∃ *d2'*. (*p*, *d2'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*) *d2*
by (*smt* (*verit*, *del-insts*) *Collect-mono-iff alloc-d1 case-prodI mem-Collect-eq stack-alloc-cases*
stack-alloc-def stack-free-def sub)

lemma *stack-allocs-stack-free-mono*:
assumes *sub*: *stack-free d1* ⊆ *stack-free d2*
assumes *alloc-d1*: (*p*, *d1'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d1*
shows ∃ *d2'*. (*p*, *d2'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*) *d2*
using *alloc-d1*
apply (*rule stack-allocs-cases*)
using *sub stack-allocs-stack-subset-stack-free'* [*OF alloc-d1*]
apply (*auto simp add*: *stack-allocs-def stack-free-def*)
done

lemma *stack-alloc-stack-free-eq*:
assumes *sub*: *stack-free d1* = *stack-free d2*
assumes *alloc-d1*: (*p*, *d1'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*::*mem-type*) *d1*
shows ∃ *d2'*. (*p*, *d2'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*) *d2*
using *stack-alloc-stack-free-mono* [*OF* - *alloc-d1*] *sub* **by** *blast*

lemma *stack-allocs-stack-free-eq*:
assumes *sub*: *stack-free d1* = *stack-free d2*
assumes *alloc-d1*: (*p*, *d1'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d1*
shows ∃ *d2'*. (*p*, *d2'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*) *d2*
using *stack-allocs-stack-free-mono* [*OF* - *alloc-d1*] *sub* **by** *blast*

lemma *fresh-ptr-stack-free-disjunct*:
(*p*, *d'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹ *ptr-span p* ∩ *stack-free d'* = {}
by (*smt* (*verit*, *best*) *disjoint-iff mem-Collect-eq mem-type-self ptr-val.ptr-val-def*
root-ptr-valid-type-neq-disjoint stack-alloc-cases stack-free-def)

lemma *fresh-ptrs-stack-free-disjunct'*:
(*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹ *i* < *n* ⟹ *ptr-span* (*p* +$_p$ *int i*) ∩ *stack-free d'* = {}
using *stack-allocs-cases*
by (*smt* (*verit*, *ccfv-threshold*) *disjoint-iff mem-Collect-eq mem-type-self ptr-val.ptr-val-def root-ptr-valid-casesE stack-free-def*)

lemma *fresh-ptrs-stack-free-disjunct*:
(*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹
{*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*) } ∩ *stack-free d'* = {}
apply (*simp add*: *array-index-span-conv*)
using *fresh-ptrs-stack-free-disjunct'*
apply *blast*
done

lemma *stack-allocs-neq*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹ *d* ≠ *d'*
by (*meson basic-trans-rules*(*31*) *disjoint-iff fresh-ptrs-stack-free-disjunct mem-type-self stack-allocs-cases stack-allocs-contained*)

lemma *stack-free-stack-alloc*:
assumes *p*: (*p*, *d'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*::*mem-type*) *d*
shows *stack-free d'* = *stack-free d* − *ptr-span p*
proof −
from *p* **obtain**
not-sb: *typ-uinfo-t* (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) **and**
sb: ∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*) **and**
ptr-aligned p **and**
c-guard: *c-guard p* **and**
valid-d': *root-ptr-valid d' p* **and**
d': *d'* = *ptr-force-type p d*
by (*cases rule*: *stack-alloc-cases*) *auto*

from *c-guard valid-d' not-sb*
have *not-stack*: ∀ *a* ∈ *ptr-span p*. ¬ *root-ptr-valid d'* (*PTR* (*stack-byte*) *a*)
apply (*simp add*: *d'*)
by (*metis IntI empty-iff mem-type-self ptr-force-type-h-t-valid ptr-val.ptr-val-def root-ptr-valid-not-subtype-disjoint sub-typ-stack-byte*)
with *fresh-ptr-stack-free-disjunct* [*OF p*] *stack-alloc-stack-subset-stack-free* [*OF p*]

show *?thesis*
apply *safe*
subgoal
using *not-sb d' not-stack*
by (*metis mem-Collect-eq p stack-alloc-root-ptr-valid-new-cases stack-free-def*)
subgoal by *auto*
subgoal

```
    using sb d' not-stack
   by (simp add: ptr-force-type-d root-ptr-valid-def stack-free-def valid-root-footprint-def)
   done
qed

lemma stack-free-stack-allocs:
  assumes p: (p, d') ∈ stack-allocs n 𝒮 TYPE('a::mem-type) d
  shows stack-free d' = stack-free d − {ptr-val p ..+ n * size-of TYPE('a)}
proof −
  from p obtain
    not-sb: typ-uinfo-t (TYPE('a)) ≠ typ-uinfo-t (TYPE(stack-byte)) and
   sb: ∀ a ∈ {ptr-val p ..+ n * size-of TYPE('a)}. root-ptr-valid d (PTR (stack-byte)
a) and

    aligned: ∀ i < n. ptr-aligned (p +p int i) and
    c-guard: ∀ i < n. c-guard (p +p int i) and
    root-valid: ∀ i < n. root-ptr-valid d' (p +p int i)and
    d': d' = fold (λi. ptr-force-type (p +p int i)) [0..<n] d
    by (cases rule: stack-allocs-cases) auto

  from c-guard root-valid not-sb
  have not-stack: ∀ a ∈ {ptr-val p ..+ n * size-of TYPE('a)}. ¬ root-ptr-valid d'
(PTR (stack-byte) a)
    apply (simp add: d' array-index-span-conv)
    by (smt (verit, best) disjoint-iff lessThan-iff mem-type-self ptr-val.ptr-val-def
root-ptr-valid-casesE)

  with fresh-ptrs-stack-free-disjunct [OF p] stack-allocs-stack-subset-stack-free [OF
p]
  show ?thesis
    apply safe
    subgoal
      using not-sb d' not-stack
     by (metis mem-Collect-eq p stack-allocs-root-ptr-valid-new-cases stack-free-def)

    subgoal by auto
    subgoal
      using sb d' not-stack
     by (simp add: fold-ptr-force-type-other root-ptr-valid-def stack-free-def valid-root-footprint-def)
    done
qed

lemma stack-release-other: x ∉ ptr-span p ⟹ stack-release p d x = d x
  by (simp add: stack-release-def ptr-retyp-d)

lemma stack-releases-other:
  fixes p::'a::mem-type ptr
  shows x ∉ {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ⟹ stack-releases n
p d x = d x
```

by (*simp add*: *stack-releases-def fold-ptr-retyp-other*)

lemma *in-ptr-span-itself*: $x \in$ *ptr-span* (*PTR*('*a*::*mem-type*) *x*)
by (*metis mem-type-self ptr-val.ptr-val-def*)

lemma *stack-byte-typing-footprint*:
stack-byte-typing x = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*(*stack-byte*))
0))
apply (*simp add*: *stack-release-def stack-byte-typing-def*)
apply (*subst ptr-force-type-footprint*)
apply (*rule in-ptr-span-itself*)
apply *simp*
done

lemma *stack-release-footprint*: $x \in$ *ptr-span p* $\Longrightarrow$
stack-release p d x = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*(*stack-byte*))
0))
apply (*simp add*: *stack-release-def stack-byte-typing-def*)
apply (*subst ptr-force-type-footprint*)
apply (*rule in-ptr-span-itself*)
apply *simp*
done

lemma *stack-releases-footprint*:
fixes *p*::'*a*::*mem-type ptr*
shows $x \in \{$*ptr-val p ..+ n* $*$ *size-of TYPE*('*a*::*mem-type*)$\}$ $\Longrightarrow$
stack-releases n p d x = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*(*stack-byte*))
0))
apply (*simp add*: *stack-releases-def stack-byte-typing-def*)
apply (*subst ptr-force-type-footprint*)
apply (*rule in-ptr-span-itself*)
apply *simp*
done

lemma *stack-byte-typing-valid-root-footprint*:
valid-root-footprint stack-byte-typing x (*typ-uinfo-t TYPE*(*stack-byte*))
using *stack-byte-typing-footprint valid-root-footprint-def* **by** (*fastforce*)

lemma *stack-release-valid-root-footprint*: $x \in$ *ptr-span p* $\Longrightarrow$
valid-root-footprint (*stack-release p d*) *x* (*typ-uinfo-t TYPE*(*stack-byte*))
using *stack-release-footprint valid-root-footprint-def* **by** (*fastforce*)

lemma *stack-releases-valid-root-footprint*:
fixes *p*::'*a*::*mem-type ptr*
shows $x \in \{$*ptr-val p ..+ n* $*$ *size-of TYPE*('*a*::*mem-type*)$\}$ $\Longrightarrow$
valid-root-footprint (*stack-releases n p d*) *x* (*typ-uinfo-t TYPE*(*stack-byte*))
using *stack-releases-footprint valid-root-footprint-def* **by** *fastforce*

lemma *stack-release-root-ptr-valid1*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *root-q*: *root-ptr-valid* (*stack-release p d*) *q*
shows *ptr-span p* ∩ *ptr-span q* = {} ∧ *root-ptr-valid d q*
apply (*rule conjI*)
using *assms*
apply (*smt* (*verit*, *ccfv-threshold*) *disjoint-iff root-ptr-valid-valid-root-footprint size-of-tag*
stack-release-valid-root-footprint valid-root-footprint-type-neq)
by (*smt* (*verit*, *best*) *intvlI non-stack-q root-ptr-valid-def root-q stack-release-other*

stack-release-valid-root-footprint valid-root-footprint-def valid-root-footprint-type-neq)

lemma *stack-releases-root-ptr-valid1*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *root-q*: *root-ptr-valid* (*stack-releases n p d*) *q*
shows {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ∩ *ptr-span q* = {} ∧ *root-ptr-valid d q*
apply (*rule context-conjI*)
subgoal
using *assms*
by (*smt* (*verit*, *best*) *disjoint-iff root-ptr-valid-def size-of-tag stack-releases-valid-root-footprint valid-root-footprint-type-neq*)
subgoal
using *assms*
by (*metis* (*full-types*) *disjoint-iff root-ptr-valid-domain' stack-releases-other*)
done

lemma *stack-release-root-ptr-valid2*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *disj*: *ptr-span p* ∩ *ptr-span q* = {}
assumes *valid-q*: *root-ptr-valid d q*
shows *root-ptr-valid* (*stack-release p d*) *q*
using *assms*
by (*smt* (*verit*, *ccfv-threshold*) *disjoint-iff intvlI root-ptr-valid-def size-of-def*
stack-release-other typ-uinfo-size valid-root-footprint-def)

lemma *stack-releases-root-ptr-valid2*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *disj*: {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ∩ *ptr-span q* = {}

assumes *valid-q*: *root-ptr-valid d q*
shows *root-ptr-valid* (*stack-releases n p d*) *q*
using *assms*
by (*metis* (*full-types*) *disjoint-iff root-ptr-valid-domain' stack-releases-other*)

lemma *stack-release-root-ptr-valid-cases*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid* (*stack-release p d*) *q* ⟷ *ptr-span p* ∩ *ptr-span q* = {} ∧ *root-ptr-valid d q*
using *assms stack-release-root-ptr-valid1 stack-release-root-ptr-valid2* **by** *blast*

lemma *stack-releases-root-ptr-valid-cases*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid* (*stack-releases n p d*) *q* ⟷ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ∩ *ptr-span q* = {} ∧ *root-ptr-valid d q*
using *assms stack-releases-root-ptr-valid1 stack-releases-root-ptr-valid2* **by** *blast*

lemma *stack-release-root-ptr-valid-same-type-cases*:
fixes *p*::*'a*::*mem-type ptr*
assumes *cvalid-p*: *d* ⊨*t p*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid* (*stack-release p d*) *q* ⟷ *p* ≠ *q* ∧ *root-ptr-valid d q*
using *non-stack-p cvalid-p root-ptr-valid-h-t-valid cvalid-same-type-neq-disjoint stack-release-root-ptr-valid-cases*
by *fastforce*

lemma *stack-releases-root-ptr-valid-same-type-cases*:
fixes *p*::*'a*::*mem-type ptr*
assumes *cvalid-p*: ⋀*i*. *i* < *n* ⟹ *d* ⊨*t* (*p* +*p int i*)
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid* (*stack-releases n p d*) *q* ⟷ (∀ *i* < *n*. *q* ≠ *p* +*p* (*int i*)) ∧ *root-ptr-valid d q*
apply *standard*
subgoal
using *non-stack-p cvalid-p cvalid-same-type-neq-disjoint stack-releases-root-ptr-valid-cases array-index-span-conv*
by (*smt* (*verit*) *Abs-fnat-hom-mult CTypesDefs.ptr-add-def disjoint-iff-not-equal ex-in-conv*
in-ptr-span-itself intvlI intvl-empty mult-less-cancel2 mult-zero-left not-gr-zero

of-int-of-nat-eq ptr-add-0-id root-ptr-valid-domain semiring-1-class.of-nat-0 stack-releases-other)

subgoal
using *non-stack-p cvalid-p root-ptr-valid-same-type-neq-disjoint stack-releases-root-ptr-valid-cases array-index-span-conv*
by (*smt* (*verit*, *best*) *array-to-index-span cvalid-same-type-neq-disjoint disjoint-iff root-ptr-valid-h-t-valid*)
done

lemma *ptr-aligned-stack-byte*[*simp*]: *ptr-aligned* (*PTR*(*stack-byte*) *x*)
by (*simp add*: *ptr-aligned-def*)

lemma *c-null-guard-cast-stack-byte*:
x ∈ *ptr-span* (*p*::'*a*::*mem-type ptr*) ⟹ *c-null-guard p* ⟹
c-null-guard (*PTR* (*stack-byte*) *x*)
apply (*clarsimp simp add*: *c-null-guard-def*)
using *intvl-Suc* **by** *blast*

lemma *c-guard-cast-stack-byte*:
x ∈ *ptr-span* (*p*::'*a*::*mem-type ptr*) ⟹ *c-guard p* ⟹
c-guard (*PTR* (*stack-byte*) *x*)
by (*clarsimp simp add*: *c-guard-def c-null-guard-cast-stack-byte*)

lemma *stack-heap-typing-root-ptr-valid-footprint*: *c-guard* (*p*::*stack-byte ptr*) ⟹
root-ptr-valid stack-byte-typing p
by (*simp add*: *root-ptr-valid-def stack-byte-typing-valid-root-footprint c-guard-cast-stack-byte*)

lemma *stack-release-root-ptr-valid-footprint*: *x* ∈ *ptr-span p* ⟹ *c-guard p* ⟹
root-ptr-valid (*stack-release p d*) (*PTR* (*stack-byte*) *x*)
by (*simp add*: *root-ptr-valid-def stack-release-valid-root-footprint c-guard-cast-stack-byte*)

lemma *stack-releases-root-ptr-valid-footprint*:
fixes *p*::'*a*::*mem-type ptr*
shows *x* ∈ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*::*mem-type*)} ⟹ ∀ *i* < *n*. *c-guard* (*p* +$_p$ *int i*) ⟹
root-ptr-valid (*stack-releases n p d*) (*PTR* (*stack-byte*) *x*)
apply (*simp add*: *root-ptr-valid-def stack-releases-valid-root-footprint c-guard-cast-stack-byte*)
using *array-to-index-span c-guard-cast-stack-byte* **by** *blast*

lemma *stack-alloc-other*:
(*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) *d* ⟹ *x* ∉ *ptr-span p* ⟹
d' x = *d x*
using *ptr-force-type-d stack-alloc-cases* **by** *blast*

lemma *stack-allocs-other*:
(*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) *d* ⟹ *x* ∉ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*::*mem-type*)} ⟹
d' x = *d x*
using *fold-ptr-force-type-other stack-allocs-cases* **by** *blast*

lemma *stack-free-stack-release-mono*:
shows *stack-free d* ⊆ *stack-free* (*stack-release p d*)
by (*smt* (*verit*) *Abs-fnat-hom-0 One-nat-def add.right-neutral less-Suc0*
mem-Collect-eq root-ptr-valid-def size-of-stack-byte(*2*)
stack-release-other stack-release-valid-root-footprint stack-free-def
subsetI typ-uinfo-size valid-root-footprint-def)

lemma *stack-free-stack-release-mono′*:
stack-free d1 ⊆ *stack-free d2* ⟹ *stack-free* (*stack-release p d1*) ⊆ *stack-free*
(*stack-release p d2*)
by (*smt* (*verit*) *Collect-mono-iff One-nat-def less-Suc0 root-ptr-valid-def size-of-stack-byte*(*3*)
stack-free-def stack-release-footprint stack-release-other valid-root-footprint-def)

lemma *stack-free-stack-releases-mono*:
shows *stack-free d* ⊆ *stack-free* (*stack-releases n p d*)
by (*smt* (*verit*) *Abs-fnat-hom-0 One-nat-def add.right-neutral less-Suc0*
mem-Collect-eq root-ptr-valid-def size-of-stack-byte(*2*)
stack-releases-other stack-releases-valid-root-footprint stack-free-def
subsetI typ-uinfo-size valid-root-footprint-def)

lemma *stack-free-stack-releases-mono′*:
stack-free d1 ⊆ *stack-free d2* ⟹ *stack-free* (*stack-releases n p d1*) ⊆ *stack-free*
(*stack-releases n p d2*)
apply (*clarsimp simp add*: *stack-releases-def stack-free-def root-ptr-valid-def valid-root-footprint-def*)
by (*smt* (*verit*, *best*) *Collect-mono-iff override-on-apply-in override-on-apply-notin*)

lemma *stack-free-ptr-span-stack-release*:
c-null-guard p ⟹ *ptr-span p* ⊆ *stack-free* (*stack-release p d*)
by (*simp add*: *c-guard-def c-null-guard-cast-stack-byte root-ptr-valid-def*
stack-release-valid-root-footprint stack-free-def subsetI)

lemma *stack-free-ptr-span-stack-releases*:
fixes *p*::*′a*::*mem-type ptr*
shows (⋀*i*. *i* < *n* ⟹ *c-null-guard* (*p* +$_p$ *int i*)) ⟹
{*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*′a*::*mem-type*)} ⊆ *stack-free* (*stack-releases n*
p d)
by (*fastforce simp add*: *c-guard-def c-null-guard-cast-stack-byte root-ptr-valid-def*

stack-releases-valid-root-footprint stack-free-def subsetI array-index-span-conv)

lemma *stack-free-stack-release*:
assumes *c-null-guard*: *c-null-guard p*
shows *stack-free* (*stack-release p d*) = *ptr-span p* ∪ *stack-free d*
proof
show *stack-free* (*stack-release p d*) ⊆ *ptr-span p* ∪ *stack-free d*
proof
fix *x*
assume *x-in*: *x* ∈ *stack-free* (*stack-release p d*)
show *x* ∈ *ptr-span p* ∪ *stack-free d*

```
    proof (cases x ∈ ptr-span p)
      case True
      then show ?thesis
        by simp
    next
      case False
      with c-null-guard x-in show ?thesis
        by (simp add: root-ptr-valid-def stack-release-other stack-free-def
            valid-root-footprint-def)
    qed
  qed
next
  show ptr-span p ∪ stack-free d ⊆ stack-free (stack-release p d)
   using c-null-guard stack-free-ptr-span-stack-release stack-free-stack-release-mono
    by blast
qed

lemma stack-free-stack-releases:
  fixes p::'a::mem-type ptr
  assumes c-null-guard: ⋀i. i < n ⟹ c-null-guard (p +p int i)
 shows stack-free (stack-releases n p d) = {ptr-val p ..+ n * size-of TYPE('a::mem-type)}
∪ stack-free d
proof
 show stack-free (stack-releases n p d) ⊆ {ptr-val p ..+ n * size-of TYPE('a::mem-type)}
∪ stack-free d
  proof
    fix x
    assume x-in: x ∈ stack-free (stack-releases n p d)
    show x ∈ {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ∪ stack-free d
    proof (cases x ∈ {ptr-val p ..+ n * size-of TYPE('a::mem-type)})
      case True
      then show ?thesis
        by simp
    next
      case False
      with c-null-guard x-in show ?thesis
        by (simp add: root-ptr-valid-def stack-releases-other stack-free-def
            valid-root-footprint-def)
    qed
  qed
next
  show {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ∪ stack-free d ⊆ stack-free
(stack-releases n p d)
   using c-null-guard stack-free-ptr-span-stack-releases stack-free-stack-releases-mono
    by blast
qed

definition On-Exit:: ('s, 'p, 'f) com ⇒ ('s, 'p, 'f) com ⇒ ('s, 'p, 'f) com where
On-Exit c cleanup = Seq (Catch c (Seq cleanup Throw)) cleanup
```

locale *heap-typing-state* =
lense htd htd-upd
for
htd:: *'s* ⇒ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s*

locale *heap-mem-state* =
lense hmem hmem-upd
for
hmem:: *'s* ⇒ *heap-mem* **and**
hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s*

locale *heap-state* =
typing: *heap-typing-state htd htd-upd* + *heap*: *heap-mem-state hmem hmem-upd*
for
htd:: *'s* ⇒ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s* **and**
hmem:: *'s* ⇒ *heap-mem* **and**
hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s* +
assumes *heap-commute*: *hmem-upd f* (*htd-upd g s*) = *htd-upd g* (*hmem-upd f s*)

begin
lemma *htd-hmem-upd* [*simp*]: *htd* (*hmem-upd f s*) = *htd s*
by (*metis heap-commute typing.get-upd typing.upd-get*)

lemma *hmem-htd-upd* [*simp*]: *hmem* (*htd-upd f s*) = *hmem s*
by (*metis heap.get-upd heap.upd-get heap-commute*)
end

locale *heap-state-global* =
heap-state htd htd-upd hmem hmem-upd + *lense glob glob-upd*
for
htd:: *'s* ⇒ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s* **and**
hmem:: *'s* ⇒ *heap-mem* **and**
hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s* **and**
glob:: *'s* ⇒ *'a* **and**
glob-upd:: (*'a* ⇒ *'a*) ⇒ *'s* ⇒ *'s* +
assumes *glob-htd-commute*: ⋀*g h*. *glob-upd g* (*htd-upd h s*) = *htd-upd h* (*glob-upd g s*)
assumes *glob-hmem-commute*: ⋀*g h*. *glob-upd g* (*hmem-upd h s*) = *hmem-upd h* (*glob-upd g s*)

locale *heap-raw-state* =
lense t-hrs t-hrs-update
for
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s*

begin
sublocale *heap-state*
λs. (*hrs-htd* (*t-hrs s*)) *λupd.* (*t-hrs-update* (*hrs-htd-update upd*))
λs. (*hrs-mem* (*t-hrs s*)) *λupd.* (*t-hrs-update* (*hrs-mem-update upd*))
apply (*unfold-locales*)
apply (*simp-all add*: *hrs-htd-update hrs-htd-update-comp hrs-mem-update hrs-mem-update-comp*)
apply (*metis hrs-htd-def hrs-htd-update hrs-mem-def hrs-mem-htd-update prod.collapse upd-same*)
apply (*metis hrs-htd-def hrs-htd-mem-update hrs-mem-def hrs-mem-update prod.expand upd-same*)
using *hrs-update-commute* **by** *force*
end

locale *heap-raw-state-global* =
heap-raw-state t-hrs t-hrs-update + *lense glob glob-upd*
for
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s* **and**
glob:: *'s* ⇒ *'a* **and**
glob-upd:: (*'a* ⇒ *'a*) ⇒ *'s* ⇒ *'s* +
assumes *glob-heap-commute*: ⋀*g h. glob-upd g* (*t-hrs-update h s*) = *t-hrs-update h* (*glob-upd g s*)
begin
sublocale *heap-state-global*
λs. (*hrs-htd* (*t-hrs s*)) *λupd.* (*t-hrs-update* (*hrs-htd-update upd*))
λs. (*hrs-mem* (*t-hrs s*)) *λupd.* (*t-hrs-update* (*hrs-mem-update upd*))
glob glob-upd
apply (*unfold-locales*)
using *glob-heap-commute*
apply *simp*
using *glob-heap-commute*
apply *simp*
done
end

locale *stack-heap-state* = *heap-state htd htd-upd hmem hmem-upd*
for
htd:: *'s* ⇒ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s* **and**
hmem:: *'s* ⇒ *heap-mem* **and**
hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s* +
fixes *S*::*addr set*
begin

definition *With-Fresh-Stack-Ptr*:: *nat* ⇒ (*'s* ⇒ *'a list set*) ⇒ ((*'a*::*mem-type ptr*)

$\Rightarrow$ ('s, 'p, strictc-errortype) com) $\Rightarrow$
('s, 'p, strictc-errortype) com **where**
‹*With-Fresh-Stack-Ptr n init c* =
Guard StackOverflow ({*s*. *stack-allocs n* $\mathcal{S}$ *TYPE*('a) (*htd s*) $\neq$ {} $\wedge$ ($\exists$ *vs*. *vs* $\in$ *init s* $\wedge$ *length vs* = *n*)})
(*DynCom* (λ*s0*.
Spec {(*s*, *t*). $\exists$ *p d' vs bs*.
(*p*, *d'*) $\in$ *stack-allocs n* $\mathcal{S}$ *TYPE*('a) (*htd s*) $\wedge$
vs $\in$ *init s* $\wedge$ *length vs* = *n* $\wedge$ *length bs* = *n* $*$ *size-of TYPE*('a) $\wedge$
t = *hmem-upd* (*fold* (λ*i*. *heap-update-padding* (*p* $+_p$ *int i*) (*vs*!*i*) (*take* (*size-of TYPE*('a)) (*drop* (*i* $*$ *size-of TYPE*('a)) *bs*))) [*0*..<*n*])
(*htd-upd* (λ-. *d'*) *s*)};;
DynCom (λ*s1*.
On-Exit
(*c* (*allocated-ptrs n* $\mathcal{S}$ *TYPE*('a) (*htd s0*) (*htd s1*)))
(*Spec* {(*s*, *t*). $\exists$ *bs*. *length bs* = *n* $*$ *size-of TYPE*('a) $\wedge$
t = *hmem-upd* (*heap-update-list* (*ptr-val* (*allocated-ptrs n* $\mathcal{S}$ *TYPE*('a) (*htd s0*) (*htd s1*))) *bs*)
(*htd-upd* (*stack-releases n* ((*allocated-ptrs n* $\mathcal{S}$ *TYPE*('a) (*htd s0*) (*htd s1*)))) *s*)})))))
›

ML ‹

structure With-Fresh-Stack-Ptr =
struct

type data = {*match*: *term* -> {*n*:*term*, *init*:*term*, *c*:*term*, *ct*-: *term*, *instantiate*: {*n*:*term*, *init*:*term*, *c*:*term*} -> *term*},
cterm-match: *cterm* -> {*n*:*cterm*, *init*:*cterm*, *c*:*cterm*, *ct*-: *cterm*, *instantiate*: {*n*:*cterm*, *init*:*cterm*, *c*:*cterm*} -> *cterm*}}

fun map-match f ({*match*, *cterm-match*}:*data*) = {*match* = *f match*, *cterm-match* = *cterm-match*}:*data*
fun map-cterm-match f ({*match*, *cterm-match*}:*data*) = {*match* = *match*, *cterm-match* = *f cterm-match*}:*data*
fun merge-match (*f*, *g*) = *Utils.fast-merge* (*fn* (*f*, *g*) => *Utils.first-match* [*f*, *g*]) (*f*, *g*)

structure Data = *Generic-Data* (
type T = *data*
val empty = {*match* = *fn* - => *raise Match*, *cterm-match* = *fn* - => *raise Match*};
val merge = *Utils.fast-merge* (*fn* ({*match* = *f1*, *cterm-match* = *g1*}, {*match* = *f2*, *cterm-match* = *g2*}) =>
{*match* = *merge-match* (*f1*, *f2*), *cterm-match* = *merge-match* (*g1*, *g2*)});
)

fun match ctxt = #*match* (*Data.get* (*Context.Proof ctxt*))

fun cterm-match ctxt = #cterm-match (Data.get (Context.Proof ctxt))

fun add-match match = Data.map (map-match (Utils.add-match match))
fun add-cterm-match cterm-match = Data.map (map-cterm-match (Utils.add-match cterm-match))

end
›

declaration ‹
fn phi => fn context =>
  let
    fun match t = @{morph-match (fo) ‹With-Fresh-Stack-Ptr ?n ?init ?c›} phi (Context.theory-of context) t
      handle Pattern.MATCH => raise Match
    fun cterm-match ct = @{cterm-morph-match (fo) ‹With-Fresh-Stack-Ptr ?n ?init ?c›} phi ct
      handle Pattern.MATCH => raise Match
  in
    context
    |> With-Fresh-Stack-Ptr.add-match match
    |> With-Fresh-Stack-Ptr.add-cterm-match cterm-match
  end
›

end

locale *stack-heap-raw-state = heap-raw-state t-hrs t-hrs-update*
  for
  t-hrs :: 's ⇒ heap-raw-state **and**
  t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's +
fixes *𝒮::addr set*
begin
  sublocale *stack-heap-state*
    λs. hrs-htd (t-hrs s) λupd. t-hrs-update (hrs-htd-update upd)
    λs. hrs-mem (t-hrs s) λupd. t-hrs-update (hrs-mem-update upd)
    𝒮
    by *unfold-locales*
end

locale *globals-stack-heap-state = stack-heap-state htd htd-upd hmem hmem-upd 𝒮*
  for
    htd:: 's ⇒ heap-typ-desc **and**
    htd-upd:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's ⇒ 's **and**
    hmem:: 's ⇒ heap-mem **and**
    hmem-upd:: (heap-mem ⇒ heap-mem) ⇒ 's ⇒ 's **and**
    𝒮::addr set +

fixes $\mathcal{G}$::*addr set*

locale *globals-stack-heap-raw-state* = *stack-heap-raw-state t-hrs t-hrs-update* $\mathcal{S}$
for
t-hrs :: $'s \Rightarrow$ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ $'s \Rightarrow$ $'s$ **and**
$\mathcal{S}$::*addr set* +
fixes $\mathcal{G}$::*addr set*
begin
sublocale *globals-stack-heap-state*
λs. hrs-htd (*t-hrs s*) *λupd. t-hrs-update* (*hrs-htd-update upd*)
λs. hrs-mem (*t-hrs s*) *λupd. t-hrs-update* (*hrs-mem-update upd*)
$\mathcal{S}$ $\mathcal{G}$
by *unfold-locales*
end

12.4 Misc derived language elements

definition
creturn :: ((*'e c-exntype* $\Rightarrow$ *'e c-exntype*) $\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme* $\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme*)
$\Rightarrow$ ((*'a* $\Rightarrow$ *'a*) $\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme* $\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme*)
$\Rightarrow$ ((*'g*, *'l*, *'e*, *'x*) *state-scheme* $\Rightarrow$ *'a*) $\Rightarrow$ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
creturn rtu xfu v $\equiv$ (*Basic* (*λs. xfu* (*λ-. v s*) *s*);; (*Basic* (*rtu* (*λ-. Return*));; *THROW*))

definition

creturn-void :: ((*'e c-exntype* $\Rightarrow$ *'e c-exntype*) $\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme*
$\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme*) $\Rightarrow$ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
creturn-void rtu $\equiv$ (*Basic* (*rtu* (*λ-. Return*));; *THROW*)

definition
cexit :: ((*'g*, *'l*, *'e*, *'x*) *state-scheme* $\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme*) $\Rightarrow$ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
cexit xfu $\equiv$ (*Basic xfu*;; *THROW*)

definition
cbreak :: ((*'e c-exntype* $\Rightarrow$ *'e c-exntype*) $\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme*
$\Rightarrow$ (*'g*, *'l*, *'e*, *'x*) *state-scheme*) $\Rightarrow$ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
cbreak rtu $\equiv$ (*Basic* (*rtu* (*λ-. Break*));; *THROW*)

definition

ccatchbrk :: (('*g*, '*l*, '*e*, '*x*) *state-scheme* ⇒'*e c-exntype*) ⇒ (('*g*, '*l*, '*e*, '*x*) *state-scheme*,'*p*,'*f*) *com*
where
ccatchbrk rt ≡ *Cond* {*s*. *rt s* = *Break*} *SKIP THROW*

definition
cgoto :: *string* ⇒ (('*e c-exntype* ⇒ '*e c-exntype*) ⇒ ('*g*, '*l*, '*e*, '*x*) *state-scheme*
⇒ ('*g*, '*l*, '*e*, '*x*) *state-scheme*) ⇒ (('*g*, '*l*, '*e*, '*x*) *state-scheme*,'*p*,'*f*) *com*
where
cgoto l rtu ≡ (*Basic* (*rtu* (λ-. *Goto l*));; *THROW*)

definition
ccatchgoto :: *string* ⇒ (('*g*, '*l*, '*e*, '*x*) *state-scheme* ⇒ '*e c-exntype*) ⇒ (('*g*, '*l*, '*e*, '*x*) *state-scheme*,'*p*,'*f*) *com*
where
ccatchgoto l rt ≡ *Cond* {*s*. *rt s* = *Goto l*} *SKIP THROW*

definition
ccatchreturn :: (('*g*, '*l*, '*e*, '*x*) *state-scheme* ⇒ '*e c-exntype*) ⇒ (('*g*, '*l*, '*e*, '*x*) *state-scheme*,'*p*,'*f*) *com*
where
ccatchreturn rt ≡ *Cond* {*s*. *is-local* (*rt s*)} *SKIP THROW*

definition
cchaos :: ('*b* ⇒ '*a* ⇒ '*a*) ⇒ ('*a*,'*c*,'*d*) *com*
where
cchaos upd ≡ *Spec* { (*s0*,*s*) . ∃ *v*. *s* = *upd v s0* }

definition
guarded-spec-body F R = *Guard F* (*fst* ‘ *R*) (*Spec R*)

end

theory *UMM*
imports
Padding-Equivalence
CLanguage
begin

instantiation *word* :: (*len8*) *stack-type*
begin
instance
by *intro-classes* (*simp add*: *typ-info-word stack-typ-info-def stack-byte-name-def*)
end

instantiation *ptr* :: (*c-type*) *stack-type*
begin
instance
by *intro-classes* (*simp add*: *typ-info-ptr stack-typ-info-def stack-byte-name-def*)
end

lemma *list-neq-witness*: $x \in set\ ys \Longrightarrow x \notin set\ xs \Longrightarrow xs \neq ys$
by *blast*

lemma *stack-typ-info-array-tag-n*:
stack-typ-info ((*array-tag-n n*)::('*a*::{*stack-type*},'*b*::*finite*) *array xtyp-info*)
apply (*induct n*)
subgoal
apply (*simp add*: *stack-typ-info-def atn-base stack-byte-name-def*)
apply (*rule list-neq-witness* [**where** *x*=*CHR ''r''*])
apply *simp*
apply *simp*
done
apply (*simp add*: *atn-rec*)
apply (*rule stack-typ-info-ti-typ-combine'*)
apply *assumption*
done

instantiation *array* ::(*stack-type*, *finite*) *stack-type*
begin
instance
by *intro-classes* (*simp add*: *typ-info-array array-tag-def stack-typ-info-array-tag-n*)
end

lemma *max-non-zero-unfold*: *NO-MATCH 0 a* $\Longrightarrow$ *NO-MATCH 0 b* $\Longrightarrow$ *max a b* = (*if a* $\leq$ *b then b else a*)
by (*simp add*: *max-def*)

lemma *eq-comp*:
assumes *eq1*: *field-update* (*component-desc x*) *bs v* $\equiv$ *g*
assumes *eq2*: *field-update* (*component-desc x*) *bs w* $\equiv$ *g*
shows *field-update* (*component-desc x*) *bs v* = *field-update* (*component-desc x*) *bs w*
using *eq1 eq2* **by** *simp*

lemma *word-rcat-single*: *word-rcat* [*x*] = *x*
by (*simp add*: *word-rcat-def bin-rcat-def*)

lemma *length-word-rsplit-8*: *length* ((*word-rsplit* (*x*::*8 word*)) :: *8 word list*) = *1*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-16*: *length* ((*word-rsplit* (*x*::*16 word*)) :: *8 word list*) = *2*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-32*: *length* ((*word-rsplit* (*x*::*32 word*)) :: *8 word list*) = *4*

by (*simp add*: *word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-64*: *length* ((*word-rsplit* (*x*::*64 word*)) :: *8 word list*) = *8*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-128*: *length* ((*word-rsplit* (*x*::*128 word*)) :: *8 word list*) = *16*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)

lemma *length-word-rsplit-signed-8*: *length* ((*word-rsplit* (*x*::*8 signed word*)) :: *8 word list*) = *1*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-signed-16*: *length* ((*word-rsplit* (*x*::*16 signed word*)) :: *8 word list*) = *2*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-signed-32*: *length* ((*word-rsplit* (*x*::*32 signed word*)) :: *8 word list*) = *4*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-signed-64*: *length* ((*word-rsplit* (*x*::*64 signed word*)) :: *8 word list*) = *8*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)
lemma *length-word-rsplit-signed-128*: *length* ((*word-rsplit* (*x*::*128 signed word*)) :: *8 word list*) = *16*
by (*simp add*: *word-rsplit-def bin-rsplit-def*)

lemmas *length-word-rsplit* =
length-word-rsplit-8
length-word-rsplit-16
length-word-rsplit-32
length-word-rsplit-64
length-word-rsplit-128
length-word-rsplit-signed-8
length-word-rsplit-signed-16
length-word-rsplit-signed-32
length-word-rsplit-signed-64
length-word-rsplit-signed-128

lemmas *wf-component-descs-intros* =
wf-component-descs-empty-typ-info
wf-component-descs-final-pad
wf-xfield.wf-component-descs-ti-typ-combine
wf-xfield.wf-component-descs-ti-typ-pad-combine

lemmas *component-descs-independent-intros* =
component-descs-independent-empty-typ-info
component-descs-independent-final-pad
wf-xfield.component-descs-independent-ti-typ-combine
wf-xfield.component-desc-independent-ti-typ-pad-combine

lemmas *wf-field-descs-intros* =

wf-field-descs-empty-typ-info
wf-field-descs-final-pad
wf-xfield.wf-field-descs-ti-typ-combine
wf-xfield.wf-field-descs-ti-typ-pad-combine

lemmas *contained-field-descs-intros* =
contained-field-descs-empty-typ-info
contained-field-descs-final-pad
contained-field-descs-ti-typ-combine
contained-field-descs-ti-typ-pad-combine

lemmas *field-update-simps* =
size-of-def ti-typ-pad-combine-def empty-typ-info-def ti-typ-combine-def ti-pad-combine-def
final-pad-def padup-def Let-def

lemmas *size-td-simps-arr-fl* =
size-td-simps
size-td-array align-td-array max-def

method *wf-xfield-solver* =
(*intro-locales*, *rule wf-field.intro*; *simp add*: *comp-def*)

method *try-wf-xfield-solver* **methods** *m* =
(*match* **conclusion in** *wf-xfield acc upd* **for** *acc upd* ⇒ *wf-xfield-solver* | *m*)

method *wf-component-descs-solver* =
(*try-wf-xfield-solver* ‹(*rule wf-component-descs-intros*)›)+

method *field-desc-independent-solver* =
(*rule field-desc-independent-PAD-expand*,
simp only: *aggregate-typ-combinators-simps set-toplevel-field-descs-combinator-simps*,
(*simp only*: *insert-union-out*)*?*,
rule field-desc-independent-PAD-collapse,
rule field-desc-independent.intro;
fastforce simp add: *fu-commutes-def*)

method *component-descs-independent-solver* =
((*try-wf-xfield-solver* ‹*rule component-descs-independent-intros*›)+;
field-desc-independent-solver)

method *wf-field-descs-solver* =
(*try-wf-xfield-solver* ‹(*rule wf-field-descs-intros*)›)+

method *contained-field-descs-solver* =
(*rule contained-field-descs-intros*)+

lemma *unat-less-helper'*: $x < \textit{of-nat}\ n \equiv \textit{True} \Longrightarrow \textit{unat}\ x < n$
by (*simp add*: *unat-less-helper*)

lemma *unat-less-helper-numeral*:
$x < (\textit{numeral}\ n) \Longrightarrow \textit{unat}\ x < (\textit{numeral}\ n)$
$x < 1 \Longrightarrow \textit{unat}\ x < 1$
by (*simp-all add*: *unat-less-helper*)

lemma *unat-less-helper-numeral'*:
$x < (\textit{numeral}\ n) \equiv \textit{True} \Longrightarrow \textit{unat}\ x < (\textit{numeral}\ n)$
$x < 1 \equiv \textit{True} \Longrightarrow \textit{unat}\ x < 1$
using *unat-less-helper-numeral* **by** *blast+*

lemma *nat-sint-less-helper*:
$i <s \textit{of-nat}\ n \Longrightarrow 0 \leq s\ i \Longrightarrow (\textit{nat}\ (\textit{sint}\ i)) < n$
by (*simp add*: *sint-eq-uint unat-less-helper word-sle-msb-le word-sless-msb-less*)

lemma *nat-sint-less-helper'*:
$i <s \textit{of-nat}\ n \equiv \textit{True} \Longrightarrow 0 \leq s\ i \equiv \textit{True} \Longrightarrow (\textit{nat}\ (\textit{sint}\ i)) < n$
by (*simp add*: *nat-sint-less-helper*)

lemma *nat-sint-less-helper-numeral*:
$i <s (\textit{numeral}\ n) \Longrightarrow 0 \leq s\ i \Longrightarrow \textit{nat}\ (\textit{sint}\ i) < (\textit{numeral}\ n)$
$i <s 1 \Longrightarrow 0 \leq s\ i \Longrightarrow \textit{nat}\ (\textit{sint}\ i) < 1$
subgoal by (*metis nat-uint-eq not-less signed.leD sint-eq-uint*
unat-less-helper-numeral(*1*) *word-msb-0 word-sle-msb-le*)
by (*metis linorder-not-less nat-code*(*2*) *signed.leD signed-0 word-less-1 word-msb-0 word-sle-msb-le zero-less-one*)

lemma *nat-sint-less-helper-numeral'*:
$i <s (\textit{numeral}\ n) \equiv \textit{True} \Longrightarrow 0 \leq s\ i \equiv \textit{True} \Longrightarrow \textit{nat}\ (\textit{sint}\ i) < (\textit{numeral}\ n)$
$i <s 1 \equiv \textit{True} \Longrightarrow 0 \leq s\ i \equiv \textit{True} \Longrightarrow \textit{nat}\ (\textit{sint}\ i) < 1$
using *nat-sint-less-helper-numeral* **by** *blast+*

lemma *sint-ucast-eq-uint'*:
$[\![\ \textit{LENGTH}('a) < \textit{LENGTH}('b)]\!]$
$\Longrightarrow \textit{sint}\ ((\textit{ucast} :: ('a{::}\textit{len word} \Rightarrow 'b{::}\textit{len word}))\ x) = \textit{uint}\ x$
apply (*rule sint-ucast-eq-uint*)
apply (*simp add*: *is-down-def target-size-def source-size-def*)
done

lemma *sint-ucast-signed-eq-uint*:
$\textit{LENGTH}('a) < \textit{LENGTH}('b) \Longrightarrow \textit{sint}\ (\textit{ucast}\ (x :: 'a :: \textit{len word}) :: 'b :: \textit{len signed word}) = \textit{uint}\ x$
apply *transfer*
apply (*clarsimp simp add*: *signed-take-bit-take-bit*)
done

lemma *ucast-unat-sless-helper*:
UCAST('a::len → 'b::len signed) x <s of-nat n ⟹
LENGTH('a::len) < LENGTH('b::len) ⟹ unat x < n
by (*smt* (*verit*, *best*) *Word.of-nat-unat nat-int-comparison*(*2*) *of-nat-numeral sint-ucast-signed-eq-uint unat-lt2p unat-of-nat-eq word-sless-sint-le*)

lemma *ucast-unat-sless-helper'*:
UCAST('a::len → 'b::len signed) x <s of-nat n ≡ True ⟹
LENGTH('a::len) < LENGTH('b::len) ⟹ unat x < n
using *ucast-unat-sless-helper* **by** *blast*

lemma *ucast-unat-sless-helper-numeral-n*:
UCAST('a::len → 'b::len signed) x <s (numeral n) ⟹
LENGTH('a::len) < LENGTH('b::len) ⟹ unat x < (numeral n)
using *ucast-unat-sless-helper*
by (*metis of-nat-numeral*)

lemma *ucast-unat-sless-helper-numeral-1*:
UCAST('a::len → 'b::len signed) x <s 1 ⟹
LENGTH('a::len) < LENGTH('b::len) ⟹ unat x < 1
using *ucast-unat-sless-helper*
by *force*

lemmas *ucast-unat-sless-helper-numeral* =
ucast-unat-sless-helper-numeral-n
ucast-unat-sless-helper-numeral-1

lemma *ucast-unat-sless-helper-numeral'*:
UCAST('a::len → 'b::len signed) x <s (numeral n) ≡ True ⟹
LENGTH('a::len) < LENGTH('b::len) ⟹ unat x < (numeral n)
UCAST('a::len → 'b::len signed) x <s 1 ≡ True ⟹
LENGTH('a::len) < LENGTH('b::len) ⟹ unat x < 1
using *ucast-unat-sless-helper-numeral* **by** *blast+*

lemma *ucast-unat-less-helper*:
UCAST('a::len → 'b::len) x < of-nat n ⟹
LENGTH('a::len) ≤ LENGTH('b::len) ⟹ unat x < n
by (*metis unat-less-helper unat-ucast-up-simp*)

lemma *ucast-unat-less-helper'*:
UCAST('a::len → 'b::len) x < of-nat n ≡ True ⟹
LENGTH('a::len) ≤ LENGTH('b::len) ⟹ unat x < n
using *ucast-unat-less-helper* **by** *blast*

lemma *ucast-unat-less-helper-numeral-n*:
UCAST('a::len → 'b::len) x < (numeral n) ⟹
LENGTH('a::len) ≤ LENGTH('b::len) ⟹ unat x < (numeral n)
using *ucast-unat-less-helper*
by (*metis of-nat-numeral*)

lemma *ucast-unat-less-helper-numeral-1*:
UCAST('a::*len* → 'b::*len*) *x* < *1* ⟹
LENGTH('a::*len*) ≤ *LENGTH*('b::*len*) ⟹ *unat x* < *1*
using *ucast-unat-less-helper*
by *force*

lemmas *ucast-unat-less-helper-numeral* =
ucast-unat-less-helper-numeral-n
ucast-unat-less-helper-numeral-1

lemma *ucast-unat-less-helper-numeral'*:
UCAST('a::*len* → 'b::*len*) *x* < (*numeral n*) ≡ *True* ⟹
LENGTH('a::*len*) ≤ *LENGTH*('b::*len*) ⟹ *unat x* < (*numeral n*)
UCAST('a::*len* → 'b::*len*) *x* < *1* ≡ *True* ⟹
LENGTH('a::*len*) ≤ *LENGTH*('b::*len*) ⟹ *unat x* < *1*
using *ucast-unat-less-helper-numeral* **by** *blast+*

lemma *len-of-less-basic-cases*:
LENGTH(*8*) < *LENGTH*(*16*)
LENGTH(*8*) < *LENGTH*(*32*)
LENGTH(*8*) < *LENGTH*(*64*)
LENGTH(*8*) < *LENGTH*(*128*)
LENGTH(*16*) < *LENGTH*(*32*)
LENGTH(*16*) < *LENGTH*(*64*)
LENGTH(*16*) < *LENGTH*(*128*)
LENGTH(*32*) < *LENGTH*(*64*)
LENGTH(*32*) < *LENGTH*(*128*)
LENGTH(*64*) < *LENGTH*(*128*)
by *simp-all*

lemma *len-of-le-basic-cases*:
LENGTH(*8*) ≤ *LENGTH*(*16*)
LENGTH(*8*) ≤ *LENGTH*(*32*)
LENGTH(*8*) ≤ *LENGTH*(*64*)
LENGTH(*8*) ≤ *LENGTH*(*128*)
LENGTH(*16*) ≤ *LENGTH*(*32*)
LENGTH(*16*) ≤ *LENGTH*(*64*)
LENGTH(*16*) ≤ *LENGTH*(*128*)
LENGTH(*32*) ≤ *LENGTH*(*64*)
LENGTH(*32*) ≤ *LENGTH*(*128*)
LENGTH(*64*) ≤ *LENGTH*(*128*)
by *simp-all*

ML ‹

structure UMM-Tools =
struct
fun tactic-from-method (*m*:*Proof.context* −> *Method.method*) *ctxt thms st* =

```
  (m ctxt thms (ctxt, st)) |> Seq.filter-results |> Seq.map snd

fun tactic-from-src ctxt src =
let
  val (-, tok) = Method.read-closure-input ctxt src
  val m = Method.method ctxt tok
in tactic-from-method m end

val wf-component-descs-tac = tactic-from-src @{context} ‹wf-component-descs-solver›
val component-descs-independent-tac = tactic-from-src @{context} ‹component-descs-independent-solver›
val wf-field-descs-tac = tactic-from-src @{context} ‹wf-field-descs-solver›
val contained-field-descs-tac = tactic-from-src @{context} ‹contained-field-descs-solver›

end

›
```

ML ‹*val UMM-ss = simpset-of* ***context***›

lemma *heap-update-fold-comp-apply* : *heap-update p v (g z)* ≡ (*heap-update p v* ∘ *g*) *z*
  by (*simp add*: *comp-apply*)

named-theorems
  fg-cons-simps **and**
  typ-info-simps **and**
  td-names-simps **and**
  typ-name-simps **and**
  upd-lift-simps **and**
  upd-other-simps **and**
  size-align-simps **and**
  fl-Some-simps **and**
  fl-ti-simps **and**
  sub-typ-simps **and**
  typ-tag-defs **and**
  size-simps **and**
  typ-name-itself **and**
  heap-update-fold-toplevel-fields **and**
  heap-update-fold-toplevel-fields-pointless **and**
  h-val-fields **and** *heap-update-fields* **and**
  h-val-unfold

named-theorems *field-lookup-prems*

declare
  size-of-words[*size-simps*]

size-of-swords[*size-simps*]
size-of-array[*size-simps*]
size-of-stack-byte[*size-simps*]
size-of-ptr[*size-simps*]

lemma *field-of-lookup-info*:
fixes *p*::'*a*:: *mem-type ptr*
assumes *field*: *field-of off* (*typ-uinfo-t TYPE*('*b*::*mem-type*)) (*typ-uinfo-t TYPE*('*a*))
shows ∃*f*. *f* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))
∧
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f* 0 = *Some* (*typ-uinfo-t TYPE*('*b*), *unat off*) ∧
field-of-t (*PTR*('*b*) &(*p*→*f*)) *p* ∧
&(*p*→*f*) = *ptr-val* (*p*::'*a ptr*) + *off* ∧
{&(*p*→*f*)..+*size-td* (*typ-uinfo-t TYPE*('*b*))} ⊆ *ptr-span p*
proof −
have *wf-desc* (*typ-info-t TYPE*('*a*)) **by** (*rule wf-desc*)

from *td-set-field-lookup* [*OF wf-desc-export-uinfo-pres*(*1*) [*OF this*]] *field*
obtain *f* **where** *lookup-u*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f* 0 = *Some* (*typ-uinfo-t TYPE*('*b*), *unat off*)
by (*auto simp add*: *field-of-def typ-uinfo-t-def*)
from *field-lookup-export-uinfo-Some-rev* [*OF lookup-u* [*simplified typ-uinfo-t-def*]]
obtain *s*′ **where** *lookup*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f* 0 = *Some* (*s*′, *unat off*) **and**
exp-s′: *export-uinfo* (*typ-info-t TYPE*('*b*)) = *export-uinfo s*′
by *blast*
from *field-names-SomeD3* [*OF lookup*]
have *f-in*: *f* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))
by (*simp add*: *field-names-u-field-names-export-uinfo-conv* (*1*) [*symmetric*] *typ-uinfo-t-def exp-s*′)

from *lookup*
have *addr*: &(*p*→*f*) = *ptr-val* (*p*::'*a ptr*) + *off*
by (*simp add*: *field-lvalue-def*)
from *field-tag-sub* [*OF lookup*]
have *contained*: {&(*p*→*f*)..+*size-td* (*typ-uinfo-t TYPE*('*b*))} ⊆ *ptr-span p*
by (*simp add*: *exp-s*′ *typ-uinfo-t-def*)

from *field addr*
have *field-of-t* (*PTR*('*b*) &(*p*→*f*)) *p*
by (*simp add*: *field-of-t-def field-of-def*)

with *contained addr lookup-u f-in*
show *?thesis*
by *blast*
qed

lemma *sub-typ-field-names-u-nonempty*:

assumes *s-t*: $s \leq t$
shows *field-names-u t s* $\neq$ []
proof −
from *s-t* **obtain** *n* **where** $(s, n) \in$ *td-set t 0*
by (*auto simp add*: *typ-tag-le-def*)
from *td-set-field-names-u-nonempty* (*1*) [*OF this*] **show** *?thesis*.
qed

definition *TO-SUC* (*n*::*nat*) ≡ *n*

simproc-setup *TO-SUC* (‹*TO-SUC* (*numeral x*)›) = ‹
fn phi => fn ctxt => fn ct =>
SOME (*Simplifier.rewrite* (*ctxt addsimps* @{*thms TO-SUC-def Num.numeral-nat*})
ct)
›
declare [[*simproc del*: *TO-SUC*]]

lemma *array-tag-SUC*:
array-tag (*t*::(*'a*::*c-type*,*'b*::*finite*) *array itself*) = *array-tag-n* (*TO-SUC* (*CARD*(*'b*)))
by (*simp add*: *array-tag-def TO-SUC-def*)

lemma *field-lookup-cons*: *field-lookup t* [*f*] *m* = *Some* (*t'*, *n*) ⟹ *wf-desc t* ⟹
field-lookup t (*f # g # gs*) *m* = *field-lookup t'* (*g*#*gs*) *n*
using *field-lookup-prefix-Some''*(*1*)[*rule-format*, **where** *f* = [*f*] **and** *g* = *g*#*gs*]
by *auto*

lemma *field-lookup-cons'*:
field-lookup (*typ-info-t* (*TYPE*(*'a*::*mem-type*))) [*f*] *m* = *Some* (*t'*, *n*) ⟹
field-lookup (*typ-info-t* (*TYPE*(*'a*::*mem-type*))) (*f # g # gs*) *m* = *field-lookup t'*
(*g*#*gs*) *n*
by (*simp add*: *field-lookup-cons*)

lemma *exists-conj-disj*: *P* ⟹ (∃ *n*. (*P* ∧ *Q n*) ∨ *R n*) = (∃ *n*. *Q n* ∨ *R n*)
by *blast*

lemma *abs-if-eq*:
assumes ⋀*x*. *b x* ⟹ *f1 x* = *f2 x*
assumes ⋀*x*. ¬ *b x* ⟹ *g1 x* = *g2 x*
shows (λ*x*. *if b x then f1 x else g1 x*) = (λ*x*. *if b x then f2 x else g2 x*) ⟷ *True*
using *assms*
by *metis*

lemma *field-lookup-typ-uinfo-t-Some*:
field-lookup (*typ-info-t TYPE*(*'a*::*c-type*)) *f m* = *Some* (*s*, *n*) ⟹
field-lookup (*typ-uinfo-t TYPE*(*'a*)) *f m* = *Some* (*export-uinfo s*, *n*)
by (*simp add*: *typ-uinfo-t-def field-lookup-export-uinfo-Some*)

lemma *adjust-ti-wf-fd-pres'*:
fixes *t*::*'a xtyp-info*

```
  and st::'a xtyp-info-struct
  and ts::'a xtyp-info-tuple list
  and x::'a xtyp-info-tuple
  assumes fg-cons: fg-cons acc upd
shows
 wf-fd t ⟹
    wf-fd (map-td (λn algn. update-desc acc upd) (update-desc acc upd) t)
 wf-fd-struct st ⟹
   wf-fd-struct (map-td-struct (λn algn. update-desc acc upd) (update-desc acc upd)
st)
 wf-fd-list ts ⟹
    wf-fd-list (map-td-list (λn algn. update-desc acc upd) (update-desc acc upd) ts)
 wf-fd-tuple x ⟹
    wf-fd-tuple (map-td-tuple (λn algn. update-desc acc upd) (update-desc acc upd)
x)
proof (induct t and st and ts and x )
case (TypDesc algn st nm)
then show ?case by auto
next
case (TypScalar  algn st d)
  then show ?case
    apply simp
    apply (simp add: fd-cons-struct-def)
    apply (insert fg-cons)
    apply (simp add: fd-cons-desc-def fg-cons-def)
    apply safe
    subgoal
      by (auto simp add: fd-cons-double-update-def update-desc-def)
        presburger
    subgoal
      by (auto simp add: fd-cons-update-access-def update-desc-def)
        presburger
    subgoal
      by (auto simp add: fd-cons-access-update-def update-desc-def)
    subgoal
      by (auto simp add: fd-cons-length-def update-desc-def)
    done
next
  case (TypAggregate ts)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto

  from Cons-typ-desc.prems obtain
    wf-fd-d: wf-fd d and
```

```
    wf-fd-fs: wf-fd-list fs and
    commutes-d-fs: fu-commutes (update-ti-t d) (update-ti-list-t fs) and
    fa-fu-ind-d-fs: fa-fu-ind
     (⦇field-access = access-ti d, field-update = update-ti-t d, field-sz = size-td d⦈)
     (⦇field-access = access-ti-list fs, field-update = update-ti-list-t fs,
       field-sz = size-td-list fs⦈)
     (size-td-list fs) (size-td d) and
    fa-fu-ind-fs-d: fa-fu-ind
     (⦇field-access = access-ti-list fs, field-update = update-ti-list-t fs,
       field-sz = size-td-list fs⦈)
     (⦇field-access = access-ti d, field-update = update-ti-t d, field-sz = size-td d⦈)
     (size-td d) (size-td-list fs)
     by (simp add: x)

   note hyps = Cons-typ-desc.hyps [simplified x, simplified]
   note hyp-d = hyps(1) [OF  wf-fd-d]
   note hyp-fs = hyps(2) [OF wf-fd-fs]
  show ?case
    apply (simp add: x hyp-d hyp-fs)
    apply safe
    subgoal
      using commutes-d-fs fg-cons
      apply (simp add: fu-commutes-def fg-cons-def)
      by auto
     (smt (verit) adjust-ti-def fg-cons field-desc.select-convs(2) field-desc-adjust-ti(3)
field-desc-list-def
           map-td-ext'(3) map-td-extI update-desc-def update-ti-t-adjust-ti wf-fd-d
wf-fd-fs)
    subgoal
      using fa-fu-ind-d-fs fg-cons
      apply (simp add: fa-fu-ind-def fg-cons-def)
      apply clarsimp
      by (smt (verit) ‹fu-commutes (update-ti-t (map-td (λn algn. update-desc acc
upd) (update-desc acc upd) d)) (update-ti-list-t (map-td-list (λn algn. update-desc
acc upd) (update-desc acc upd) fs))›
       fa-fu-v fd-cons-length fd-cons-update-access fu-commutes-def hyp-d map-td-size(1)
wf-fd-consD)
    subgoal
      using fa-fu-ind-fs-d fg-cons
      apply (simp add: fa-fu-ind-def fg-cons-def)
      apply clarsimp
      by (metis adjust-ti-def fg-cons update-ti-t-adjust-ti)
    done
next
  case (DTuple-typ-desc d nm y)
  then show ?case by auto
qed
```

lemma *adjust-ti-wf-fd-pres*[*simp*]: *fg-cons acc upd* $\Longrightarrow$ *wf-fd t* $\Longrightarrow$ *wf-fd* (*adjust-ti t acc upd*)
by (*simp add*: *adjust-ti-wf-fd-pres′ adjust-ti-def*)

lemma *neq-td-names-eq-neq-export-uinfo*: *td-names* (*typ-info-t t*) $\neq$ *td-names* (*typ-info-t s*)
$\Longrightarrow$ *export-uinfo* (*typ-info-t t*) = *export-uinfo* (*typ-info-t s*) $\longleftrightarrow$ *False*
by (*metis td-names-export-uinfo*)

lemma *set-field-names-no-padding-all-field-names-no-padding-conv′*:
set (*field-names-no-padding* (*typ-info-t TYPE*('*a*::*mem-type*)) *t*) =
Set.filter
(λ*f*. ∃ *s n*. *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*) ∧ *export-uinfo s* = *t*)
(*set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*))))
by (*simp add*: *set-field-names-no-padding-all-field-names-no-padding-conv Set.filter-def*)

lemma *field-names-no-padding-all-field-names-no-padding-conv′*:
field-names-no-padding (*typ-info-t TYPE*('*a*::*mem-type*)) *t* =
filter
(λ*f*. ∃ *s n*. *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*) ∧ *export-uinfo s* = *t*)
(*all-field-names-no-padding* (*typ-info-t TYPE*('*a*)))
by (*simp add*: *field-names-no-padding-all-field-names-no-padding-conv Set.filter-def*)

lemma *set-filter-insert*: *Set.filter P* (*insert x S*) =
(*if P x then insert x* (*Set.filter P S*) *else Set.filter P S*)
by (*auto simp add*: *Set.filter-def*)

lemma *set-filter-cons-image*: *Set.filter P* ((#) *x* ‘ *S*) = (#) *x* ‘ *Set.filter* (λ*fs*. *P* (*x*#*fs*)) *S*
by (*auto simp add*: *Set.filter-def*)

lemma *set-filter-Sup*: *Set.filter P* (⋃*x*∈*X*. *S x*) = (⋃*x*∈*X*. *Set.filter P* (*S x*))
by (*auto simp add*: *Set.filter-def*)

lemma *set-filter-empty*: *Set.filter P* {} = {}
by (*auto simp add*: *Set.filter-def*)

lemma *cons-image-Sup*: (#) *x* ‘ (⋃*xs*∈*X*. *S xs*) = (⋃*xs*∈*X*. ((#) *x* ‘ *S xs*))
by (*rule image-UN*)

lemma *set-filter-image-all*:
assumes ⋀*x*. *x* < *n* $\Longrightarrow$ *P* (*f x*)
shows *Set.filter P* (*f* ‘ {*0*..<*n*}) = *f* ‘ {*0*..<*n*}
using *assms*
by *fastforce*

lemma *set-filter-image-none*:

assumes $\bigwedge x.\ x < n \Longrightarrow \neg\ P\ (f\ x)$
shows *Set.filter* $P\ (f\ `\ \{0..<n\}) =\ \{\}$
using *assms*
by *fastforce*

lemma *set-filter-union-distrib*: *Set.filter* $P\ (X \cup Y) =$ *Set.filter* $P\ X \cup$ *Set.filter* $P\ Y$
by (*auto simp add*: *Set.filter-def*)

lemma *sub-typ-refl* [*simp*]: $TYPE('a) \leq_\tau\ TYPE('a{::}c\text{-}type)$
by (*simp add*: *sub-typ-def*)

lemma *not-sub-typ-via-td-name*:
assumes *ta*: *typ-name* (*typ-info-t* $TYPE('a :: c\text{-}type)) \neq$ *pad-typ-name*
and *tina*: *typ-name* (*typ-info-t* $TYPE('a :: c\text{-}type)) \notin$ *td-names* (*typ-info-t* $TYPE('b :: c\text{-}type)$)
shows $\neg\ TYPE('a :: c\text{-}type) \leq_\tau\ TYPE('b :: c\text{-}type)$
using *ta tina*
apply (*clarsimp simp add*: *sub-typ-def typ-tag-le-def*)
apply (*drule td-set-td-names*)
apply (*auto simp add*: *typ-uinfo-t-def*)
done

lemma *nat-to-bin-string-eq-to-nat-eq*:
assumes *eq*: *nat-to-bin-string* $n =$ *nat-to-bin-string* m
shows $n = m$
using *eq*
proof (*induct* n *arbitrary*: m *rule*: *nat-less-induct*)
case (*1* n)
then obtain *eq*: *nat-to-bin-string* $n =$ *nat-to-bin-string* m **and**
hyp: $\bigwedge m\ x.\ m < n \Longrightarrow$ *nat-to-bin-string* $m =$ *nat-to-bin-string* $x \Longrightarrow m = x$
by *auto*

show *?case*
proof (*cases* n)
case *0*
with *eq* **show** *?thesis*
apply(*cases* m)
apply (*simp add*: *ntbs*)
apply (*simp add*: *ntbs*)
by (*metis list.simps*(*3*))
next
case (*Suc* n')
note $n' =$ *this*
show *?thesis*
proof (*cases* m)
case *0*
with *eq* **show** *?thesis*
apply (*simp add*: n' *ntbs*)

```
      by (meson list.simps(3))
    next
      case (Suc m')
      have le: (Suc n' div 2) < n
        by (simp add: n')

      note hyp' = hyp [OF le, of Suc m' div 2]
      from eq show ?thesis
        apply (simp add: n' Suc)
        apply (subst (asm) (1 2) ntbs)
        apply clarsimp
        apply (frule hyp')
        apply (clarsimp split: if-split-asm)
         apply arith+
        done
    qed
  qed
qed

lemma nat-to-bin-string-inj [simp]: nat-to-bin-string n = nat-to-bin-string m ⟷
n = m
  using nat-to-bin-string-eq-to-nat-eq by blast

simproc-setup nat-to-bin-string (‹nat-to-bin-string (numeral x)›) = ‹
fn phi => fn ctxt => fn ct =>
  SOME (Simplifier.rewrite (ctxt addsimps @{thms nat-to-bin-string.simps}) ct)
›

lemma rewrite-solve-prop:
  assumes rew: (PROP P) ≡ Trueprop Q
  assumes solve: PROP Trueprop Q
  shows (PROP P) ≡ Trueprop True
proof −
  from solve have Q = True by simp
  from rew [simplified this]
  show PROP ?thesis
    by simp
qed

lemma trueprop-eq-bool-eq:
  assumes prop-eq: PROP Trueprop P ≡ PROP Trueprop Q
  shows P = Q
proof (cases P)
  case True
  with prop-eq have Q
    by (simp add: True)
  with True show ?thesis by simp
next
  case False
```

with *prop-eq* **have** $Q = False$
by (*cases* Q) *auto*
with *False* **show** *?thesis* **by** *simp*
qed

lemmas *rewrite-solve-prop-eq* = *eq-reflection* [*OF trueprop-eq-bool-eq*, *OF rewrite-solve-prop*]
lemmas *trueprop-eq-bool-meta-eq* = *trueprop-eq-bool-eq* [*THEN eq-reflection*]

lemma *export-uinfo-typ-uinfo-t-match*[*simp*]:
export-uinfo (*typ-info-t* $TYPE('a)$) = *typ-uinfo-t* ($t::'a::c\text{-}type\ itself$) = *True*
by (*simp add*: *typ-uinfo-t-def*)

lemma *export-uinfo-eq-sub-typ-conv*:
export-uinfo (*typ-info-t* $TYPE('a::c\text{-}type)$) = *export-uinfo* (*typ-info-t* $TYPE('b::c\text{-}type)$)
$\longleftrightarrow$
$TYPE('a) \leq_\tau TYPE('b) \wedge TYPE('b) \leq_\tau TYPE\ ('a)$
apply *standard*
apply (*simp add*: *fold-typ-uinfo-t sub-typ-def*)
by (*simp add*: *dual-order.antisym fold-typ-uinfo-t sub-typ-def*)

lemma *typ-uinfo-eq-sub-typ-conv*:
typ-uinfo-t $TYPE('a::c\text{-}type)$ = *export-uinfo* (*typ-info-t* $TYPE('b::c\text{-}type)$)
$\longleftrightarrow$
$TYPE('a) \leq_\tau TYPE('b) \wedge TYPE('b) \leq_\tau TYPE\ ('a)$
export-uinfo (*typ-info-t* $TYPE('a::c\text{-}type)$) = *typ-uinfo-t* $TYPE('b::c\text{-}type)$
$\longleftrightarrow$
$TYPE('a) \leq_\tau TYPE('b) \wedge TYPE('b) \leq_\tau TYPE\ ('a)$
typ-uinfo-t $TYPE('a::c\text{-}type)$ = *typ-uinfo-t* $TYPE('b::c\text{-}type)$
$\longleftrightarrow$
$TYPE('a) \leq_\tau TYPE('b) \wedge TYPE('b) \leq_\tau TYPE\ ('a)$
by (*simp-all add*: *typ-uinfo-t-def export-uinfo-eq-sub-typ-conv*)

lemma *array-typ-subtyp-array-typ*:
assumes *typ-uinfo-t* ($TYPE\ ('a::mem\text{-}type)$) = *typ-uinfo-t* ($TYPE('c::mem\text{-}type)$)
shows *typ-uinfo-t* ($TYPE\ ('a::mem\text{-}type['b::finite])$) = *typ-uinfo-t* ($TYPE('c['b::finite])$)
apply (*simp add*: *typ-uinfo-array-tag-n-m-eq*)
apply (*simp add*: *uinfo-array-tag-n-m-def*)
using *assms*
by *simp*

lemma *le-array-typ-intro*:
$TYPE\ ('a::mem\text{-}type) \leq_\tau TYPE\ ('c::mem\text{-}type) \Longrightarrow$
$TYPE\ ('c::mem\text{-}type) \leq_\tau TYPE\ ('a::mem\text{-}type) \Longrightarrow$
$TYPE('a::mem\text{-}type['b::finite]) \leq_\tau TYPE('c::mem\text{-}type['b::finite])$
using *typ-uinfo-eq-sub-typ-conv array-typ-subtyp-array-typ*
by (*smt* (*verit*))

lemma *sub-typ-signed-unsiged*: $TYPE('a::len8\ signed\ word) \leq_\tau TYPE('a\ word)$
by (*simp add*: *sub-typ-def typ-uinfo-t-signed-word-word-conv*)

lemma *sub-typ-unsigned-signed*: $TYPE('a\ word) \leq_\tau TYPE('a::len8\ signed\ word)$
by (*simp add*: *sub-typ-def typ-uinfo-t-signed-word-word-conv*)

lemma *sub-typ-proper-conv*: $TYPE('a::c\text{-}type) <_\tau TYPE('b::c\text{-}type) \longleftrightarrow$
$typ\text{-}uinfo\text{-}t\ TYPE('a) \neq typ\text{-}uinfo\text{-}t\ TYPE('b) \wedge TYPE('a::c\text{-}type) \leq_\tau TYPE('b::c\text{-}type)$
by (*metis sub-typ-def sub-typ-proper-def typ-tag-lt-def*)

lemma *sub-typ-proper-to-sub-typ*:
$TYPE('a::c\text{-}type) <_\tau TYPE('b::c\text{-}type) \Longrightarrow TYPE('a::c\text{-}type) \leq_\tau TYPE('b::c\text{-}type)$
using *sub-typ-proper-conv* **by** *blast*

lemma *to-bytes-p-zero*: *to-bytes-p* (*c-type-class.zero*::$'a$::*xmem-type*) = *replicate* (*size-of* $TYPE('a)$) *0*
by (*simp add*: *zero-def to-bytes-p-def to-bytes-from-bytes-id*)

lemma *field-lookup-zero*:
assumes *fl*: *field-lookup* (*typ-info-t* $TYPE('a::xmem\text{-}type)$) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t* $TYPE('b::c\text{-}type)$
shows *from-bytes* (*access-ti*$_0$ *t* (*c-type-class.zero*::$'a$)) = (*c-type-class.zero*::$'b$)
proof −
from *field-access-take-dropD* [*OF fl*, *simplified*]
have *eq1*: *take* (*size-td t*) (*drop n* (*to-bytes-p* (*c-type-class.zero*::$'a$))) = *access-ti*$_0$ *t* (*c-type-class.zero*::$'a$) **by** *simp*

from *fl match* **have** *sz*: *size-td t* = *size-of* $TYPE('b)$
by (*simp add*: *export-size-of*)
from *fl* **have** *le*: *size-of* $TYPE('b) \leq$ *size-of* $TYPE('a) - n$
by (*metis export-size-of field-lookup-offset-size fold-typ-uinfo-t nat-move-sub-le sz*)

have *take* (*size-td t*) (*drop n* (*to-bytes-p* (*c-type-class.zero*::$'a$))) = *replicate* (*size-of* $TYPE('b)$) *0*
by (*simp add*: *to-bytes-p-zero sz le*)
with *eq1* **have** *eq2*: *access-ti*$_0$ *t* (*c-type-class.zero*::$'a$) = *replicate* (*size-of* $TYPE('b)$) *0* **by** *simp*
show *?thesis*
apply (*simp add*: *eq2*)
apply (*simp add*: *zero-def*)
done
qed

lemma *field-lookup-zero′*:
assumes *fl*: *field-lookup* (*typ-info-t* $TYPE('a::xmem\text{-}type)$) *f 0* ≡ *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *export-uinfo* (*typ-info-t* $TYPE('b::c\text{-}type)$)
shows *from-bytes* (*access-ti*$_0$ *t* (*c-type-class.zero*::$'a$)) = (*c-type-class.zero*::$'b$)
using *field-lookup-zero* [*simplified typ-uinfo-t-def*, *OF - match*] *fl* **by** *blast*

lemma *array-index-zero*:
  assumes *i-bound*: $i < CARD('b)$
  shows (*c-type-class.zero*::(('*a* :: *array-outer-max-size*)['*b* :: *array-max-count*])).[*i*]
= (*c-type-class.zero*::'*a*)
proof −
  from *field-lookup-array* [*OF i-bound*, **where** *'a='a*]
  have *fl*: *field-lookup* (*typ-info-t TYPE*('*a*['*b*])) [*replicate i CHR ''1''*] *0* =
        Some (*adjust-ti* (*typ-info-t TYPE*('*a*)) (λ*x*. *x*.[*i*]) (λ*x f*. *Arrays.update f i x*), *i* ∗ *size-of TYPE*('*a*))
    by *simp*

  have *export-uinfo* (*adjust-ti* (*typ-info-t TYPE*('*a*)) (λ*x*::'*a*['*b*]. *x*.[*i*]) (λ*x f*. *Arrays.update f i x*)) = *typ-uinfo-t TYPE*('*a*)
    by (*simp add*: *typ-uinfo-t-def i-bound*)

  from *field-lookup-zero* [*OF fl this*]
  show *?thesis*
    by *simp*
qed

named-theorems *zero-simps* **and** *make-zero*
end

Chapter 13

Packed Types (no implicit padding)

theory *PackedTypes*
imports *WordSetup CProof*
begin

13.1 Underlying definitions for the class axioms

field-access / *field-update* is the identity for packed types

definition *fa-fu-idem* :: *'a field-desc* ⇒ *nat* ⇒ *bool* **where**
fa-fu-idem fd n ≡
∀ *bs bs' v*. *length bs* = *n* ⟶ *length bs'* = *n* ⟶ *field-access fd* (*field-update fd bs v*) *bs'* = *bs*

primrec
td-fafu-idem :: (*'a field-desc*,*'b*)*typ-desc* ⇒ *bool* **and**
td-fafu-idem-struct :: (*'a field-desc*,*'b*) *typ-struct* ⇒ *bool* **and**
td-fafu-idem-list :: ((*'a field-desc*,*'b*) *typ-desc*, *char list*,*'b*) *dt-tuple list* ⇒ *bool* **and**
td-fafu-idem-tuple :: ((*'a field-desc*,*'b*) *typ-desc*, *char list*,*'b*) *dt-tuple* ⇒ *bool* **where**
fai0: *td-fafu-idem* (*TypDesc algn ts n*) = *td-fafu-idem-struct ts*

| *fai1*: *td-fafu-idem-struct* (*TypScalar n algn d*) = *fa-fu-idem d n*
| *fai2*: *td-fafu-idem-struct* (*TypAggregate ts*) = *td-fafu-idem-list ts*

| *fai3*: *td-fafu-idem-list* [] = *True*
| *fai4*: *td-fafu-idem-list* (*x*#*xs*) = (*td-fafu-idem-tuple x* ∧ *td-fafu-idem-list xs*)

| *fai5*: *td-fafu-idem-tuple* (*DTuple x n d*) = *td-fafu-idem x*

lemmas *td-fafu-idem-simps* = *fai0 fai1 fai2 fai3 fai4 fai5*

field-access is independent of the underlying bytes

definition *fa-heap-indep* :: *'a field-desc* ⇒ *nat* ⇒ *bool* **where**
fa-heap-indep fd n ≡
∀ *bs bs' v*. *length bs* = *n* ⟶ *length bs'* = *n* ⟶ *field-access fd v bs* = *field-access fd v bs'*

primrec
td-fa-hi :: (*'a field-desc*,*'b*) *typ-desc* ⇒ *bool* **and**
td-fa-hi-struct :: (*'a field-desc*,*'b*) *typ-struct* ⇒ *bool* **and**
td-fa-hi-list :: ((*'a field-desc*,*'b*) *typ-desc*, *char list*,*'b*) *dt-tuple list* ⇒ *bool* **and**
td-fa-hi-tuple :: ((*'a field-desc*,*'b*) *typ-desc*, *char list*,*'b*) *dt-tuple* ⇒ *bool*
where
fahi0: *td-fa-hi* (*TypDesc algn ts n*) = *td-fa-hi-struct ts*

| *fahi1*: *td-fa-hi-struct* (*TypScalar n algn d*) = *fa-heap-indep d n*
| *fahi2*: *td-fa-hi-struct* (*TypAggregate ts*) = *td-fa-hi-list ts*

| *fahi3*: *td-fa-hi-list* [] = *True*
| *fahi4*: *td-fa-hi-list* (*x*#*xs*) = (*td-fa-hi-tuple x* ∧ *td-fa-hi-list xs*)

| *fahi5*: *td-fa-hi-tuple* (*DTuple x n d*) = *td-fa-hi x*

lemmas *td-fa-hi-simps* = *fahi0 fahi1 fahi2 fahi3 fahi4 fahi5*

13.2 Lemmas about *td-fafu-idem*

lemma *field-lookup-td-fafu-idem*:
shows ⋀(*s* :: (*'a field-desc*,*'b*) *typ-desc*) *f m n*.
⟦ *field-lookup t f m* = *Some* (*s*, *n*); *td-fafu-idem t* ⟧ ⟹ *td-fafu-idem s*
and ⋀(*s* :: (*'a field-desc*,*'b*) *typ-desc*) *f m n*.
⟦ *field-lookup-struct st f m* = *Some* (*s*, *n*); *td-fafu-idem-struct st* ⟧ ⟹ *td-fafu-idem s*
and ⋀(*s* :: (*'a field-desc*,*'b*) *typ-desc*) *f m n*.
⟦ *field-lookup-list ts f m* = *Some* (*s*, *n*); *td-fafu-idem-list ts* ⟧ ⟹ *td-fafu-idem s*
and ⋀(*s* :: (*'a field-desc*,*'b*) *typ-desc*) *f m n*.
⟦ *field-lookup-tuple p f m* = *Some* (*s*, *n*); *td-fafu-idem-tuple p* ⟧ ⟹ *td-fafu-idem s*
by (*induct t* **and** *st* **and** *ts* **and** *p*) (*auto split*: *if-split-asm option.splits*)

lemma *field-access-update-same*:
fixes *t* :: (*'a* :: *mem-type field-desc*,*'b*) *typ-desc* **and** *st* :: (*'a field-desc*,*'b*) *typ-struct* **and**
ts:: (*'a field-desc*, *'b*) *typ-tuple list* **and**
p:: (*'a field-desc*, *'b*) *typ-tuple*
shows ⋀(*v* :: *'a*) *bs bs'*. ⟦ *td-fafu-idem t*; *wf-fd t*; *length bs* = *size-td t*; *length bs'* = *size-td t*⟧

$\Longrightarrow$ *access-ti t* (*update-ti t bs v*) *bs'* = *bs*
and $\bigwedge$(*v* :: *'a*) *bs bs'*. ⟦ *td-fafu-idem-struct st*; *wf-fd-struct st*; *length bs* = *size-td-struct st*; *length bs'* = *size-td-struct st* ⟧
$\Longrightarrow$ *access-ti-struct st* (*update-ti-struct st bs v*) *bs'* = *bs*
and $\bigwedge$(*v* :: *'a*) *bs bs'*. ⟦ *td-fafu-idem-list ts*; *wf-fd-list ts*; *length bs* = *size-td-list ts*; *length bs'* = *size-td-list ts*⟧
$\Longrightarrow$ *access-ti-list ts* (*update-ti-list ts bs v*) *bs'* = *bs*
and $\bigwedge$(*v* :: *'a*) *bs bs'*. ⟦ *td-fafu-idem-tuple p*; *wf-fd-tuple p*; *length bs* = *size-td-tuple p*; *length bs'* = *size-td-tuple p*⟧
$\Longrightarrow$ *access-ti-tuple p* (*update-ti-tuple p bs v*) *bs'* = *bs*
proof (*induct t* **and** *st* **and** *ts* **and** *p*)
case *TypScalar* **thus** *?case* **by** (*clarsimp simp*: *fa-fu-idem-def*)
next
case (*Cons-typ-desc p' ts' v bs bs'*)
hence *fu-commutes* (*update-ti-tuple-t p'*) (*update-ti-list-t ts'*) **by** *clarsimp*
moreover
have *update-ti-tuple p'* (*take* (*size-td-tuple p'*) *bs*) = *update-ti-tuple-t p'* (*take* (*size-td-tuple p'*) *bs*)
using *Cons-typ-desc.prems* **by** (*simp add*: *update-ti-tuple-t-def min-ll*)
moreover
have *update-ti-list ts'* (*drop* (*size-td-tuple p'*) *bs*) = *update-ti-list-t ts'* (*drop* (*size-td-tuple p'*) *bs*)
using *Cons-typ-desc.prems* **by** (*simp add*: *update-ti-list-t-def*)
ultimately have *updeq*:
(*update-ti-tuple p'* (*take* (*size-td-tuple p'*) *bs*) (*update-ti-list ts'* (*drop* (*size-td-tuple p'*) *bs*) *v*))
= (*update-ti-list ts'* (*drop* (*size-td-tuple p'*) *bs*) (*update-ti-tuple p'* (*take* (*size-td-tuple p'*) *bs*) *v*))
unfolding *fu-commutes-def* **by** *simp*

show *?case* **using** *Cons-typ-desc.prems*
by (*auto simp add*: *Cons-typ-desc.hyps*) (*simp add*: *updeq Cons-typ-desc.hyps*)
qed *simp+*

lemma *access-ti-tuple-dt-fst*:
access-ti-tuple p v bs = *access-ti* (*dt-fst p*) *v bs*
by (*cases p*, *simp*)

lemma *wf-fd-tuple-dt-fst*:
wf-fd-tuple p = *wf-fd* (*dt-fst p*)
by (*cases p*, *simp*)

lemma *field-lookup-offset2*:
assumes *fl*: (*field-lookup t f* (*m* + *n*) = *Some* (*s*, *q*))
shows *field-lookup t f m* = *Some* (*s*, *q* − *n*)
proof −
from *fl* **have** *le*: *m* + *n* ≤ *q*
by (*rule field-lookup-offset-le*)

hence $q = (m + n) + (q - (m + n))$
by *simp*

hence *field-lookup t f* $(m + n) =$ *Some* $(s, (m + n) + (q - (m + n)))$ **using** *fl*
by *simp*

hence *field-lookup t f m* = *Some* $(s, m + (q - (m + n)))$
by (*rule iffD1* [*OF field-lookup-offset'*(*1*)])

thus *?thesis* **using** *le* **by** *simp*
qed

lemma *field-lookup-offset2-list*:
assumes *fl*: (*field-lookup-list ts f* $(m + n) =$ *Some* (s, q))
shows *field-lookup-list ts f m* = *Some* $(s, q - n)$
proof −
from *fl* **have** *le*: $m + n \leq q$
by (*rule field-lookup-offset-le*)

hence $q = (m + n) + (q - (m + n))$
by *simp*

hence *field-lookup-list ts f* $(m + n) =$ *Some* $(s, (m + n) + (q - (m + n)))$
using *fl* **by** *simp*

hence *field-lookup-list ts f m* = *Some* $(s, m + (q - (m + n)))$
by (*rule iffD1* [*OF field-lookup-offset'*(*3*)])

thus *?thesis* **using** *le* **by** *simp*
qed

lemma *field-lookup-offset2-pair*:
assumes *fl*: (*field-lookup-tuple p f* $(m + n) =$ *Some* (s, q))
shows *field-lookup-tuple p f m* = *Some* $(s, q - n)$
proof −
from *fl* **have** *le*: $m + n \leq q$
by (*rule field-lookup-offset-le*)

hence $q = (m + n) + (q - (m + n))$
by *simp*

hence *field-lookup-tuple p f* $(m + n) =$ *Some* $(s, (m + n) + (q - (m + n)))$
using *fl* **by** *simp*

hence *field-lookup-tuple p f m* = *Some* $(s, m + (q - (m + n)))$
by (*rule iffD1* [*OF field-lookup-offset'*(*4*)])

thus *?thesis* **using** *le* **by** *simp*

qed

lemma *field-access-update-nth-inner*:
shows $\bigwedge f$ $(s :: ('a :: mem\text{-}type\ field\text{-}desc, 'b)\ typ\text{-}desc)$ $n\ x\ v\ bs\ bs'$.
$[\![$ *field-lookup t f 0 = Some* (s, n); $n \leq x$; $x < n +$ *size-td s*; *td-fafu-idem s*; *wf-fd s*; *wf-fd t*;
length bs = size-td s; *length bs′ = size-td t* $]\!]$
$\Longrightarrow$ *access-ti t* $($*update-ti s bs v*$)$ *bs′* ! *x = bs* ! $(x - n)$

and $\bigwedge f$ $(s :: ('a :: mem\text{-}type\ field\text{-}desc, 'b)\ typ\text{-}desc)$ $n\ x\ v\ bs\ bs'$.
$[\![$*field-lookup-struct st f 0 = Some* (s, n); $n \leq x$; $x < n +$ *size-td s*; *td-fafu-idem s*; *wf-fd s*; *wf-fd-struct st*;
length bs = size-td s; *length bs′ = size-td-struct st* $]\!]$
$\Longrightarrow$ *access-ti-struct st* $($*update-ti s bs v*$)$ *bs′* ! *x = bs* ! $(x - n)$

and $\bigwedge f$ $(s :: ('a :: mem\text{-}type\ field\text{-}desc, 'b)\ typ\text{-}desc)$ $n\ x\ v\ bs\ bs'$.
$[\![$*field-lookup-list ts f 0 = Some* (s, n); $n \leq x$; $x < n +$ *size-td s*; *td-fafu-idem s*; *wf-fd s*; *wf-fd-list ts*;
length bs = size-td s; *length bs′ = size-td-list ts*$]\!]$
$\Longrightarrow$ *access-ti-list ts* $($*update-ti s bs v*$)$ *bs′* ! *x = bs* ! $(x - n)$

and $\bigwedge f$ $(s :: ('a :: mem\text{-}type\ field\text{-}desc, 'b)\ typ\text{-}desc)$ $n\ x\ v\ bs\ bs'$.
$[\![$*field-lookup-tuple p f 0 = Some* (s, n); $n \leq x$; $x < n +$ *size-td s*; *td-fafu-idem s*; *wf-fd s*; *wf-fd-tuple p*;
length bs = size-td s; *length bs′ = size-td-tuple p*$]\!]$
$\Longrightarrow$ *access-ti-tuple p* $($*update-ti s bs v*$)$ *bs′* ! *x = bs* ! $(x - n)$
proof $($*induct t* **and** *st* **and** *ts* **and** *p*$)$
case $($*TypDesc align typ-struct ls f s n x v bs bs′*$)$

show *?case*
proof $($*cases f* = []$)$
case *False* **thus** *?thesis* **using** *TypDesc* **by** *clarsimp*
next
case *True*
thus *?thesis* **using** *TypDesc.prems*
by $($*simp add*: *field-access-update-same*$)$
qed
next
case $($*Cons-typ-desc p′ ts′ f s n x v bs bs′*$)$
have *nlex*: $n \leq x$ **and** *xln*: $x < n +$ *size-td s*
and *lbs*: *length bs = size-td s* **and** *lbs′*: *length bs′ = size-td-list* $(p' \# ts')$ **by** *fact+*
from *Cons-typ-desc* **have** *wf*: *wf-fd* $($*dt-fst p′*$)$ **and** *wfts*: *wf-fd-list ts′* **by** $($*cases p′*, *auto*$)$

{
assume *fl*: *field-lookup-list ts′ f* $($*size-td* $($*dt-fst p′*$))$ = *Some* (s, n)

hence *mlt*: *size-td* (*dt-fst p′*) ≤ *n*
by (*rule field-lookup-offset-le*)

from *fl* **have** *fl′*: *field-lookup-list ts′ f 0* = *Some* (*s*, *n* − *size-td* (*dt-fst p′*))
by (*rule field-lookup-offset2-list* [**where** *m* = *0*, *simplified*])

hence *atl*: *access-ti-list ts′* (*update-ti s bs v*) (*drop* (*size-td* (*dt-fst p′*)) *bs′*) ! (*x* − *size-td* (*dt-fst p′*)) = *bs* ! (*x* − *n*)
using *mlt nlex xln lbs lbs′ wf wfts* ‹*td-fafu-idem s*› ‹*wf-fd s*›
by (*simp add*: *Cons-typ-desc.hyps*(*2*) [*OF fl′*] *size-td-tuple-dt-fst*)

from *mlt* **have** *size-td* (*dt-fst p′*) ≤ *x*
by (*rule order-trans*) *fact*

hence *?case* **using** *wf lbs lbs′ atl*
by (*simp add*: *nth-append length-fa-ti access-ti-tuple-dt-fst size-td-tuple-dt-fst*)
}
moreover
{
note *ih* = *Cons-typ-desc.hyps*(*1*)[*simplified access-ti-tuple-dt-fst wf-fd-tuple-dt-fst*]

assume *fl*: *field-lookup-tuple p′ f 0* = *Some* (*s*, *n*)

hence *x* < *size-td* (*dt-fst p′*)
apply (*cases p′*)
apply (*simp split*: *if-split-asm*)
apply (*drule field-lookup-offset-size′*)
apply (*rule order-less-le-trans* [*OF xln*])
apply *simp*
done

hence *?case* **using** *wf lbs lbs′ nlex xln wf wfts* ‹*td-fafu-idem s*› ‹*wf-fd s*›
by (*simp add*: *nth-append length-fa-ti access-ti-tuple-dt-fst size-td-tuple-dt-fst ih*[*OF fl*])
}
ultimately show *?case* **using** ‹*field-lookup-list* (*p′* # *ts′*) *f 0* = *Some* (*s*, *n*)› **by** (*simp split*: *option.splits*)
qed (*clarsimp split*: *if-split-asm*)+

13.2.1 *td-fa-hi*

lemma *fa-heap-indepD*:
⟦ *fa-heap-indep fd n*; *length bs* = *n*; *length bs′* = *n* ⟧ ⟹
field-access fd v bs = *field-access fd v bs′*
unfolding *fa-heap-indep-def*
apply (*drule spec*, *drule spec*, *drule spec*)
apply (*drule* (*1*) *mp*)
apply (*erule* (*1*) *mp*)
done

lemma *td-fa-hi-heap-independence*:
fixes *t*::('*a*::*mem-type*, '*b*) *typ-info* **and**
st::('*a*::*mem-type*,'*b*) *typ-info-struct* **and**
ts::('*a*::*mem-type*, '*b*) *typ-info-tuple list* **and**
p::('*a*::*mem-type*, '*b*) *typ-info-tuple*

shows $\bigwedge$(*v* :: '*a* :: *mem-type*) *h h'*. ⟦ *td-fa-hi t*; *length h* = *size-td t*; *length h'* = *size-td t* ⟧
⟹ *access-ti t v h* = *access-ti t v h'*
and $\bigwedge$(*v* :: '*a* :: *mem-type*) *h h'*. ⟦ *td-fa-hi-struct st*; *length h* = *size-td-struct st*; *length h'* = *size-td-struct st*⟧
⟹ *access-ti-struct st v h* = *access-ti-struct st v h'*
and $\bigwedge$(*v* :: '*a* :: *mem-type*) *h h'*. ⟦ *td-fa-hi-list ts*; *length h* = *size-td-list ts*; *length h'* = *size-td-list ts* ⟧
⟹ *access-ti-list ts v h* = *access-ti-list ts v h'*
and $\bigwedge$(*v* :: '*a* :: *mem-type*) *h h'*. ⟦ *td-fa-hi-tuple p*; *length h* = *size-td-tuple p*; *length h'* = *size-td-tuple p* ⟧
⟹ *access-ti-tuple p v h* = *access-ti-tuple p v h'*
proof (*induct t* **and** *st* **and** *ts* **and** *p*)
case *TypDesc*
from *TypDesc.prems* **show** *?case*
by (*simp*) (*erule* (*2*) *TypDesc.hyps*)
next
case *TypScalar*
from *TypScalar.prems* **show** *?case*
by *simp* (*erule* (*2*) *fa-heap-indepD*)
next
case *TypAggregate*
from *TypAggregate.prems* **show** *?case*
by (*simp*) (*erule* (*2*) *TypAggregate.hyps*)
next
case *Nil-typ-desc* **thus** *?case* **by** *simp*
next
case *Cons-typ-desc*
from *Cons-typ-desc.prems* **show** *?case*
apply *simp*
apply (*erule conjE*)
apply (*rule arg-cong2* [**where** *f* = (@)])
apply (*erule Cons-typ-desc.hyps*; *simp*)
apply (*erule Cons-typ-desc.hyps*; *simp*)
done
next
case *DTuple-typ-desc*
from *DTuple-typ-desc.prems* **show** *?case*
by *simp* (*erule* (*2*) *DTuple-typ-desc.hyps*)
qed

13.3 Simp rules for deriving packed props from the type combinators

13.3.1 *td-fafu-idem*

lemma *td-fafu-idem-map-align* [*simp*]: *td-fafu-idem* (*map-align f t*) = *td-fafu-idem t*
by (*cases t*) *simp*

lemma *td-fafu-idem-final-pad*:
padup (*2* ^ *max algn* (*align-td t*)) (*size-td t*) = *0*
⟹ *td-fafu-idem* (*final-pad algn t*) = *td-fafu-idem t*
unfolding *final-pad-def*
by (*clarsimp simp add*: *padup-def Let-def*)

lemma *td-fafu-idem-ti-typ-pad-combine*:
fixes *t* :: *'a* :: *c-type itself* **and** *s* :: (*'b* :: *c-type*) *xtyp-info*
assumes *pad*: *padup* (*max* (*2* ^ *algn*) (*align-of TYPE*(*'a*))) (*size-td s*) = *0*
shows *td-fafu-idem* (*ti-typ-pad-combine t xf xfu algn nm s*) = *td-fafu-idem* (*ti-typ-combine t xf xfu algn nm s*)
unfolding *ti-typ-pad-combine-def* **using** *pad*
by (*clarsimp simp*: *Let-def*)

lemma *td-fafu-idem-list-append*:
fixes *xs* :: *'a* :: *c-type xtyp-info-tuple list*
shows *td-fafu-idem-list* (*xs* @ *ys*) = (*td-fafu-idem-list xs* ∧ *td-fafu-idem-list ys*)
by (*induct xs*) *simp+*

lemma *td-fafu-idem-extend-ti*:
fixes *t* :: *'a* :: *c-type xtyp-info*
fixes *s* :: *'a* :: *c-type xtyp-info*
assumes *as*: *td-fafu-idem s*
and *at*: *td-fafu-idem t*
shows *td-fafu-idem* (*extend-ti s t algn nm d*) **using** *as at*
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*; *simp add*: *td-fafu-idem-list-append*)
done
done

lemma *fd-cons-access-updateD*:
⟦ *fd-cons-access-update d n*; *length bs* = *n*; *length bs'* = *n*⟧ ⟹
field-access d (*field-update d bs v*) *bs'* = *field-access d* (*field-update d bs v'*) *bs'*
unfolding *fd-cons-access-update-def* **by** *clarsimp*

lemma *fa-fu-idem-update-desc*:
fixes *a* :: *'a field-desc*
assumes *fg*: *fg-cons xf xfu*
and *fd*: *fd-cons-struct* (*TypScalar n n' a*)

shows *fa-fu-idem* (*update-desc xf xfu a*) *n* = *fa-fu-idem a n*
proof
assume *asm*: *fa-fu-idem* (*update-desc xf xfu a*) *n*

let *?fu* = λ*bs*. *if length bs* = *n then field-update a bs else id*
let *?a'* = (| *field-access* = *field-access a*, *field-update* = *?fu*, *field-sz* = *n* |)

show *fa-fu-idem a n*
unfolding *fa-fu-idem-def*
proof (*intro impI conjI allI*)
fix *bs* :: *byte list* **and** *bs'* :: *byte list* **and** *v*
assume *l*: *length bs* = *n* **and** *l'*: *length bs'* = *n*

hence (∀ *v*. *field-access a* (*field-update a bs* (*xf v*)) *bs'* = *bs*)
= (∀ *v*. *field-access a* (*?fu bs* (*xf v*)) *bs'* = *bs*) **by** *simp*

also have ... = (∀ *v*. *field-access a* (*field-update a bs v*) *bs'* = *bs*) **using** *fd*
apply −
apply (*rule iffI*)
apply (*rule allI*)
apply (*subst* (*asm*) *fd-cons-access-updateD* [*OF* - *l l'*, **where** *d* = *?a'*, *simplified*])
apply (*simp add*: *fd-cons-struct-def fd-cons-desc-def*)
apply (*fastforce simp*: *l l'*)
apply (*fastforce simp*: *l l'*)
done

finally show *field-access a* (*field-update a bs v*) *bs'* = *bs* **using** *asm fg l l'*
by (*clarsimp simp add*: *update-desc-def fa-fu-idem-def fg-cons-def*)
qed
next
assume *fa-fu-idem a n*
thus *fa-fu-idem* (*update-desc xf xfu a*) *n*
unfolding *fa-fu-idem-def update-desc-def* **using** *fg*
by (*clarsimp simp add*: *update-desc-def fa-fu-idem-def fg-cons-def*)
qed

lemma *td-fafu-idem-map-td-update-desc*:
assumes *fg*: *fg-cons xf xfu*
shows *wf-fd t* ⟹ *td-fafu-idem* (*map-td* (λ- -. *update-desc xf xfu*) (*update-desc xf xfu*) *t*) = *td-fafu-idem t*
and *wf-fd-struct st* ⟹ *td-fafu-idem-struct* (*map-td-struct* (λ- -. *update-desc xf xfu*) (*update-desc xf xfu*) *st*) = *td-fafu-idem-struct st*
and *wf-fd-list ts* ⟹ *td-fafu-idem-list* (*map-td-list* (λ- -. *update-desc xf xfu*) (*update-desc xf xfu*) *ts*) = *td-fafu-idem-list ts*
and *wf-fd-tuple p* ⟹ *td-fafu-idem-tuple* (*map-td-tuple* (λ- -. *update-desc xf xfu*) (*update-desc xf xfu*) *p*) = *td-fafu-idem-tuple p*
by (*induct t* **and** *st* **and** *ts* **and** *p*) (*auto elim*!: *fa-fu-idem-update-desc* [*OF fg*])

lemmas *td-fafu-idem-adjust-ti = td-fafu-idem-map-td-update-desc(1)[folded adjust-ti-def]*

lemma *td-fafu-idem-ti-typ-combine*:
fixes *s :: 'b :: c-type xtyp-info*
assumes *fg*: *fg-cons xf xfu*
and *tda*: *td-fafu-idem (typ-info-t TYPE('a :: mem-type))*
and *tds*: *td-fafu-idem s*
shows *td-fafu-idem (ti-typ-combine TYPE('a :: mem-type) xf xfu algn nm s)*
unfolding *ti-typ-combine-def* **using** *tda tds*
apply (*clarsimp simp*: *Let-def*)
apply (*cases s*)
subgoal for *x1 typ-struct xs*
apply (*cases typ-struct*)
apply *simp*
apply (*subst td-fafu-idem-adjust-ti* [*OF fg wf-fd*], *assumption*)
apply (*simp add*: *td-fafu-idem-list-append*)
apply (*subst td-fafu-idem-adjust-ti* [*OF fg wf-fd*], *assumption*)
done
done

lemma *td-fafu-idem-ptr*:
td-fafu-idem (typ-info-t TYPE('a :: c-type ptr))
apply (*clarsimp simp add*: *fa-fu-idem-def*)
apply (*subst word-rsplit-rcat-size*)
apply (*clarsimp simp add*: *size-of-def word-size*)
apply *simp*
done

lemma *td-fafu-idem-word*:
td-fafu-idem (typ-info-t TYPE('a :: len8 word))
apply(*clarsimp simp*: *fa-fu-idem-def*)
apply (*subst word-rsplit-rcat-size*)
apply (*insert len8-dv8*)
apply (*clarsimp simp add*: *size-of-def word-size*)
apply (*subst dvd-div-mult-self*; *simp*)
apply *simp*
done

lemma *td-fafu-idem-array-n*:
⟦ *td-fafu-idem (typ-info-t TYPE('a))*; *n ≤ card (UNIV :: 'b set)* ⟧ ⟹
td-fafu-idem (array-tag-n n :: ('a :: mem-type ['b :: finite]) xtyp-info)
by (*induct n*; *simp add*: *array-tag-n.simps empty-typ-info-def*)
(*simp add*: *td-fafu-idem-ti-typ-combine*)

lemma *td-fafu-idem-array*:
td-fafu-idem (typ-info-t TYPE('a)) ⟹ td-fafu-idem (typ-info-t TYPE('a :: mem-type ['b :: finite]))
by (*clarsimp simp*: *typ-info-array array-tag-def fa-fu-idem-def td-fafu-idem-array-n*)

lemma *td-fafu-idem-empty-typ-info*:
td-fafu-idem (*empty-typ-info algn t*)
unfolding *empty-typ-info-def*
by *simp*

13.3.2 *td-fa-hi*

lemma *td-fa-hi-final-pad*:
padup (*2* ^ *max algn* (*align-td t*)) (*size-td t*) = *0*
⟹ *td-fa-hi* (*final-pad algn t*) = *td-fa-hi t*
unfolding *final-pad-def*
by (*cases t*) (*clarsimp simp add*: *padup-def Let-def*)

lemma *td-fa-hi-ti-typ-pad-combine*:
fixes *t* :: *'a* :: *c-type itself* **and** *s* :: *'b* :: *c-type xtyp-info*
assumes *pad*: *padup* (*max* (*2* ^ *align*) (*align-of TYPE*(*'a*))) (*size-td s*) = *0*
shows *td-fa-hi* (*ti-typ-pad-combine t xf xfu algn nm s*) = *td-fa-hi* (*ti-typ-combine t xf xfu algn nm s*)
unfolding *ti-typ-pad-combine-def* **using** *pad*
by (*clarsimp simp*: *Let-def*)

lemma *td-fa-hi-list-append*:
fixes *xs* :: *'a* :: *c-type xtyp-info-tuple list*
shows *td-fa-hi-list* (*xs* @ *ys*) = (*td-fa-hi-list xs* ∧ *td-fa-hi-list ys*)
by (*induct xs*) *simp*+

lemma *td-fa-hi-extend-ti*:
fixes *t* :: *'a* :: *c-type xtyp-info*
assumes *as*: *td-fa-hi s*
and *at*: *td-fa-hi t*
shows *td-fa-hi* (*extend-ti s t algn nm d*) **using** *as at*
apply (*cases s*)
subgoal for *x1 typ-struct xs*
by (*cases typ-struct*; *simp add*: *td-fa-hi-list-append*)
done

lemma *fa-heap-indep-update-desc*:
fixes *a* :: *'a field-desc*
assumes *fg*: *fg-cons xf xfu*
and *fd*: *fd-cons-struct* (*TypScalar n n' a*)
shows *fa-heap-indep* (*update-desc xf xfu a*) *n* = *fa-heap-indep a n*
proof
assume *asm*: *fa-heap-indep* (*update-desc xf xfu a*) *n*

have *xf-xfu*: ⋀*v v'*. *xf* (*xfu v v'*) = *v* **using** *fg*
unfolding *fg-cons-def*
by *simp*

show *fa-heap-indep a n*
unfolding *fa-heap-indep-def*
proof (*intro impI conjI allI*)
fix *bs* :: *byte list* **and** *bs'* :: *byte list* **and** *v*
assume *l*: *length bs* = *n* **and** *l'*: *length bs'* = *n*
with *asm*
have *field-access* (*update-desc xf xfu a*) (*xfu v undefined*) *bs* =
field-access (*update-desc xf xfu a*) (*xfu v undefined*) *bs'*
by (*rule fa-heap-indepD*)

thus *field-access a v bs* = *field-access a v bs'*
unfolding *update-desc-def*
by (*simp add*: *xf-xfu*)
qed
next
assume *asm*: *fa-heap-indep a n*
show *fa-heap-indep* (*update-desc xf xfu a*) *n*
unfolding *fa-heap-indep-def update-desc-def*
apply (*simp*, *intro impI conjI allI*)
using *asm* **by** (*metis fa-heap-indepD*)
qed

lemma *td-fa-hi-map-td-update-desc*:
assumes *fg*: *fg-cons xf xfu*
shows *wf-fd t* ⟹ *td-fa-hi* (*map-td* (λ- -. *update-desc xf xfu*) (*update-desc xs xfu*) *t*) = *td-fa-hi t*
and *wf-fd-struct st* ⟹ *td-fa-hi-struct* (*map-td-struct* (λ- -. *update-desc xf xfu*) (*update-desc xs xfu*) *st*) = *td-fa-hi-struct st*
and *wf-fd-list ts* ⟹ *td-fa-hi-list* (*map-td-list* (λ- -. *update-desc xf xfu*) (*update-desc xs xfu*) *ts*) = *td-fa-hi-list ts*
and *wf-fd-tuple p* ⟹ *td-fa-hi-tuple* (*map-td-tuple* (λ- -. *update-desc xf xfu*) (*update-desc xs xfu*) *p*) = *td-fa-hi-tuple p*
by (*induct t* **and** *st* **and** *ts* **and** *p*) (*auto elim*!: *fa-heap-indep-update-desc* [*OF fg*])

lemma *td-fa-hi-adjust-ti*:
assumes *fg*: *fg-cons xf xfu*
assumes *wf*: *wf-fd t*
shows *td-fa-hi* (*adjust-ti t xf xfu*) = *td-fa-hi t*
using *fg wf*
by (*simp add*: *adjust-ti-def td-fa-hi-map-td-update-desc*)

lemma *td-fa-hi-ti-typ-combine*:
fixes *s* :: *'b* :: *c-type xtyp-info*
assumes *fg*: *fg-cons xf xfu*
and *tda*: *td-fa-hi* (*typ-info-t TYPE*(*'a* :: *mem-type*))
and *tds*: *td-fa-hi s*
shows *td-fa-hi* (*ti-typ-combine TYPE*(*'a* :: *mem-type*) *xf xfu algn nm s*)
unfolding *ti-typ-combine-def Let-def* **using** *tda tds*

apply (*cases s*)
subgoal for *x1 typ-struct xs*
by (*cases typ-struct*; *simp add*: *td-fa-hi-list-append td-fa-hi-adjust-ti*[*OF fg wf-fd*])
done

lemma *td-fa-hi-ptr*:
td-fa-hi (*typ-info-t TYPE*('*a* :: *c-type ptr*))
by (*clarsimp simp add*: *fa-heap-indep-def*)

lemma *td-fa-hi-word*:
td-fa-hi (*typ-info-t TYPE*('*a* :: *len8 word*))
by (*clarsimp simp add*: *fa-heap-indep-def*)

lemma *td-fa-hi-array-n*:
⟦*td-fa-hi* (*typ-info-t TYPE*('*a*)); *n* ≤ *card* (*UNIV* :: '*b set*)⟧ ⟹ *td-fa-hi* (*array-tag-n n* :: ('*a* :: *mem-type* ['*b* :: *finite*]) *xtyp-info*)
by (*induct n*; *simp add*: *array-tag-n.simps empty-typ-info-def td-fa-hi-ti-typ-combine*)

lemma *td-fa-hi-array*:
td-fa-hi (*typ-info-t TYPE*('*a*)) ⟹ *td-fa-hi* (*typ-info-t TYPE*('*a* :: *mem-type* ['*b* :: *finite*]))
by (*clarsimp simp add*: *typ-info-array array-tag-def fa-fu-idem-def td-fa-hi-array-n*)

lemma *td-fa-hi-empty-typ-info*:
td-fa-hi (*empty-typ-info algn t*)
unfolding *empty-typ-info-def*
by *simp*

13.4 The type class and simp sets

Packed types, with no padding, have the defining property that access is invariant under substitution of the underlying heap and access/update is the identity

class *packed-type* = *mem-type* +
assumes *td-fafu-idem*: *td-fafu-idem* (*typ-info-t TYPE*('*a*))
assumes *td-fa-hi*: *td-fa-hi* (*typ-info-t TYPE*('*a*))

lemmas *td-fafu-idem-intro-simps* =
— Axioms
td-fafu-idem
— Combinators
td-fafu-idem-final-pad td-fafu-idem-ti-typ-pad-combine td-fafu-idem-ti-typ-combine td-fafu-idem-empty-typ-info
— Constructors
td-fafu-idem-ptr td-fafu-idem-word td-fafu-idem-array

lemmas *td-fa-hi-intro-simps* =

— Axioms
td-fa-hi
— Combinators
td-fa-hi-final-pad td-fa-hi-ti-typ-pad-combine td-fa-hi-ti-typ-combine td-fa-hi-empty-typ-info
— Constructors
td-fa-hi-ptr td-fa-hi-word td-fa-hi-array

lemma *align-td-wo-align-array′*:
align-td-wo-align (*typ-info-t TYPE*('*a* :: *c-type*['*b* :: *finite*])) = *align-td-wo-align* (*typ-info-t TYPE*('*a*))
by (*simp add*: *typ-info-array array-tag-def align-td-wo-align-array-tag*)

lemma *align-td-array′*:
align-td (*typ-info-t TYPE*('*a* :: *c-type*['*b* :: *finite*])) = *align-td* (*typ-info-t TYPE*('*a*))
by (*simp add*: *typ-info-array array-tag-def align-td-array-tag*)

lemmas *packed-type-intro-simps* =
td-fafu-idem-intro-simps td-fa-hi-intro-simps align-td-wo-align-array′ size-td-simps-3 size-td-array

lemma *access-ti-append′*:
⋀*list*.
access-ti-list (*xs* @ *ys*) *t list* =
access-ti-list xs t (*take* (*size-td-list xs*) *list*) @
access-ti-list ys t (*drop* (*size-td-list xs*) *list*)
proof(*induct xs*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case* **by** (*simp add*: *min-def ac-simps drop-take*)
qed

13.5 Instances

Words (of multiple of 8 size) are packed

instantiation *word* :: (*len8*) *packed-type*
begin
instance
by (*intro-classes*; *rule td-fafu-idem-word td-fa-hi-word*)
end

Pointers are always packed

instantiation *ptr* :: (*c-type*)*packed-type*
begin
instance
by (*intro-classes*; *simp add*: *fa-fu-idem-def word-rsplit-rcat-size word-size fa-heap-indep-def*)
end

Arrays of packed types are in turn packed

class *array-outer-packed* = *packed-type* + *array-outer-max-size*
class *array-inner-packed* = *array-outer-packed* + *array-inner-max-size*

instance *word* :: (*len8*)*array-outer-packed* **..**
instance *word* :: (*len8*)*array-inner-packed* **..**

instance *array* :: (*array-outer-packed*, *array-max-count*) *packed-type*
by (*intro-classes*; *simp add*: *td-fafu-idem-intro-simps td-fa-hi-intro-simps*)

instance *array* :: (*array-inner-packed*, *array-max-count*) *array-outer-packed* **..**

13.6 Theorems about packed types

13.6.1 *td-fa-hi*

lemma *heap-independence*:
⟦*length h* = *size-of TYPE*('*a* :: *packed-type*); *length h'* = *size-of TYPE*('*a*) ⟧
⟹ *access-ti* (*typ-info-t TYPE*('*a*)) *v h* = *access-ti* (*typ-info-t TYPE*('*a*)) *v h'*
by (*rule td-fa-hi-heap-independence*(*1*)[*OF td-fa-hi*], *simp-all add*: *size-of-def*)

theorem *packed-heap-update-collapse*:
fixes *u*::'*a*::*packed-type*
fixes *v*::'*a*
shows *heap-update p v* (*heap-update p u h*) = *heap-update p v h*
unfolding *heap-update-def*
apply(*rule ext*)
subgoal for *x*
apply(*cases x* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)})
apply(*simp add*: *heap-update-mem-same-point*)
apply(*simp add*:*to-bytes-def*)
apply(*subst heap-independence*, *simp*)
prefer *2*
apply(*rule refl*)
apply(*simp*)
apply(*simp add*: *heap-update-nmem-same*)
done
done

lemma *packed-heap-update-collapse-hrs*:
fixes *p* :: '*a* :: *packed-type ptr*
shows *hrs-mem-update* (*heap-update p v*) (*hrs-mem-update* (*heap-update p v'*) *hp*) =
hrs-mem-update (*heap-update p v*) *hp*
unfolding *hrs-mem-update-def*
by (*simp add*: *split-def packed-heap-update-collapse*)

13.6.2 *td-fafu-idem*

lemma *order-leE*:

fixes $x :: 'a :: order$
shows $[\![x \leq y; x = y \Longrightarrow P; x < y \Longrightarrow P]\!] \Longrightarrow P$
by (*auto simp*: *order-le-less*)

lemma *of-nat-mono-maybe-le*:
shows $[\![X < 2 \hat{} \textit{len-of } TYPE('a);\ Y \leq X]\!] \Longrightarrow (\textit{of-nat } Y :: 'a :: \textit{len word}) \leq \textit{of-nat } X$
apply (*erule order-leE*)
apply *simp*
apply (*rule order-less-imp-le*)
apply (*erule* (*1*) *of-nat-mono-maybe*)
done

lemma *intvl-le-lower*:
fixes $x :: 'a :: \textit{len word}$
shows $[\![x \in \{y..+n\}; y \leq y + \textit{of-nat } (n - 1); n < 2 \hat{} \textit{len-of } TYPE('a)]\!] \Longrightarrow y \leq x$
apply (*drule intvlD*)
apply (*elim conjE exE*)
apply (*erule ssubst*)
apply (*erule word-plus-mono-right2*)
apply (*rule of-nat-mono-maybe-le*)
apply *simp*
apply *simp*
done

lemma *intvl-less-upper*:
fixes $x :: 'a :: \textit{len word}$
shows $[\![x \in \{y..+n\}; y \leq y + \textit{of-nat } (n - 1); n < 2 \hat{} \textit{len-of } TYPE('a)]\!] \Longrightarrow x \leq y + \textit{of-nat } (n - 1)$
apply (*drule intvlD*)
apply (*elim conjE exE*)
apply (*erule ssubst*)
apply (*rule word-plus-mono-right*; *assumption?*)
apply (*rule of-nat-mono-maybe-le*; *simp*)
done

lemma *packed-type-access-ti*:
fixes $v :: 'a :: \textit{packed-type}$
assumes *lbs*: $\textit{length } bs = \textit{size-of } TYPE('a)$
shows $\textit{access-ti } (\textit{typ-info-t } TYPE('a))\ v\ bs = \textit{access-ti}_0\ (\textit{typ-info-t } TYPE('a))\ v$
unfolding $\textit{access-ti}_0\textit{-def}$
by (*rule heap-independence*; *simp add*: *lbs size-of-def*)

lemma *c-guard-field-lvalue*:
fixes $p :: 'a :: \textit{mem-type ptr}$
assumes *cg*: *c-guard p*
and *fl*: $\textit{field-lookup } (\textit{typ-info-t } TYPE('a))\ f\ 0 = \textit{Some } (t, n)$
and *eu*: $\textit{export-uinfo } t = \textit{typ-uinfo-t } TYPE('b :: \textit{mem-type})$

shows *c-guard* (*Ptr* &(*p*→*f*) :: *'b* :: *mem-type ptr*)
unfolding *c-guard-def*
proof (*rule conjI*)
from *cg fl eu* **show** *ptr-aligned* (*Ptr* &(*p*→*f*) :: *'b ptr*)
by (*rule c-guard-ptr-aligned-fl*)
next
from *eu* **have** *std*: *size-td t* = *size-of TYPE*(*'b*) **using** *fl*
by (*simp add*: *export-size-of*)

from *cg* **have** *c-null-guard p* **unfolding** *c-guard-def* **..**
thus *c-null-guard* (*Ptr* &(*p*→*f*) :: *'b ptr*) **unfolding** *c-null-guard-def*
apply (*rule contrapos-nn*)
apply (*rule subsetD* [*OF field-tag-sub*, *OF fl*])
apply (*simp add*: *std*)
done
qed

lemma *word-wrap-of-natD*:
fixes *x* :: *'a* :: *len word*
assumes *wraps*: ¬ *x* ≤ *x* + *of-nat n*
shows ∃ *k*. *x* + *of-nat k* = *0* ∧ *k* ≤ *n*
proof −
show *?thesis*
proof (*rule exI* [**where** *x* = *unat* (− *x*)], *intro conjI*)
show *x* + *of-nat* (*unat* (−*x*)) = *0*
by *simp*
next
show *unat* (−*x*) ≤ *n*
by (*metis add.commute no-plus-overflow-neg not-less olen-add-eqv word-unat-less-le wraps*)
qed
qed

theorem *packed-heap-super-field-update*:
fixes *v* :: *'a* :: *packed-type* **and** *p* :: *'b* :: *packed-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*t*, *n*)
and *cgrd*: *c-guard p*
and *eu*: *export-uinfo t* = *typ-uinfo-t TYPE*(*'a*)
shows *heap-update* (*Ptr* &(*p*→*f*)) *v hp* = *heap-update p* (*update-ti t* (*to-bytes-p v*) (*h-val hp p*)) *hp*
unfolding *heap-update-def to-bytes-def*
apply (*simp add*: *packed-type-access-ti*, *rule ext*)
proof −
fix *x*
let *?LHS* = *heap-update-list* &(*p*→*f*) (*to-bytes-p v*) *hp x*
let *?RHS* = *heap-update-list* (*ptr-val p*) (*to-bytes-p* (*update-ti t* (*to-bytes-p v*) (*h-val hp p*))) *hp x*

from *cgrd* **have** *al*: *ptr-val p* ≤ *ptr-val p* + *of-nat* (*size-of TYPE*(*'b*) − *1*) **by**

(*rule c-guard-no-wrap*)

have *szb*: *size-of TYPE*('*b*) < 2 ^ *len-of TYPE*(*addr-bitsize*)
apply (*fold card-word*)
apply (*fold addr-card-def*)
apply (*rule max-size*)
done

have *szt*: *n* + *size-td t* ≤ *size-of TYPE*('*b*)
unfolding *size-of-def*
by (*subst add.commute*, *rule field-lookup-offset-size* [*OF fl*])
moreover have *t0*: *0* < *size-td t* **using** *fl wf-size-desc*
by (*rule field-lookup-wf-size-desc-gt*)
ultimately have *szn*: *n* < *size-of TYPE*('*b*) **by** *simp*
from *szt* **have** *szt1*: *n* + (*size-td t* − *1*) ≤ *size-of TYPE*('*b*)
by *simp*

have *b0*: *0* < *size-of* (*TYPE* ('*b*)) **using** *wf-size-desc*
unfolding *size-of-def*
by (*rule wf-size-desc-gt*)

have *uofn*: *unat* (*of-nat n* :: *addr-bitsize word*) = *n* **using** *szn szb*
by (*metis le-unat-uoi nat-less-le unat-of-nat-len*)

from *eu* **have** *std*: *size-td t* = *size-of TYPE*('*a*) **using** *fl*
by (*simp add*: *export-size-of*)

hence *?LHS* = (*if x* ∈ {&(*p*→*f*)..+*size-td t*} *then* (*to-bytes-p v*) ! *unat* (*x* − &(*p*→*f*)) *else hp x*)
by (*simp add*: *heap-update-mem-same-point heap-update-nmem-same*)
also have ... = *?RHS*
apply (*simp*, *intro impI conjI*)
proof −
assume *xin*: *x* ∈ {&(*p*→*f*)..+*size-td t*}
have *to-bytes-p v* ! *unat* (*x* − &(*p*→*f*)) = *to-bytes-p* (*update-ti t* (*to-bytes-p v*) (*h-val hp p*)) ! *unat* (*x* − *ptr-val p*)
proof (*simp add*: *to-bytes-p-def to-bytes-def*, *subst field-access-update-nth-inner*(*1*)[*OF fl*, *simplified*])

have *c-guard* (*Ptr* &(*p*→*f*) :: '*a ptr*) **using** *cgrd fl eu*
by (*rule c-guard-field-lvalue*)
hence *pft*: &(*p*→*f*) ≤ &(*p*→*f*) + *of-nat* (*size-td t* − *1*)
apply −
apply (*drule c-guard-no-wrap*)
apply (*simp add*: *std*)
done

have *szt'*: *size-td t* < 2 ^ *len-of TYPE*(*addr-bitsize*)
apply (*subst std*)

```
    apply (fold card-word)
    apply (fold addr-card-def)
    apply (rule max-size)
    done

  have ofn: of-nat n ≤ x − ptr-val p
  proof (rule le-minus')
    from xin show ptr-val p + of-nat n ≤ x using pft szt'
      unfolding field-lvalue-def field-lookup-offset-eq [OF fl]
      by (rule intvl-le-lower)
  next
   from szb szn have of-nat n ≤ (of-nat (size-of TYPE('b) − 1) :: addr-bitsize
word)
      apply −
      apply (rule of-nat-mono-maybe-le)
       apply simp-all
      done
    with al show ptr-val p ≤ ptr-val p + of-nat n
      by (rule word-plus-mono-right2)
  qed

  thus nlt: n ≤ unat (x − ptr-val p)
    by (metis uofn word-less-eq-iff-unsigned)

  have x ≤ ptr-val p + (of-nat n + of-nat (size-td t − 1)) using xin pft szt' t0
    unfolding field-lvalue-def field-lookup-offset-eq [OF fl]
    by (metis (no-types) add.assoc intvl-less-upper)
  moreover have x ∈ {ptr-val p..+size-of TYPE('b)} using fl xin
    by (rule subsetD [OF field-tag-sub])
  ultimately have x − ptr-val p ≤ (of-nat n + of-nat (size-td t − 1)) using
al szb
    by (metis add-diff-cancel-left' intvl-le-lower word-diff-ls(4))
  moreover have unat (of-nat n + of-nat (size-td t − 1) :: addr-bitsize word)
= n + size-td t − 1
    using t0 order-le-less-trans [OF szt1 szb]
   by (metis Nat.add-diff-assoc One-nat-def Suc-leI of-nat-add unat-of-nat-len)
  ultimately have unat (x − ptr-val p) ≤ n + size-td t − 1
    by (simp add: word-le-nat-alt)
  thus unat (x − ptr-val p) < n + size-td t using t0
    by simp

  show td-fafu-idem t
    by (rule field-lookup-td-fafu-idem(1)[OF fl td-fafu-idem])

  show wf-fd t
    by (rule wf-fd-field-lookupD [OF fl wf-fd])

  show length (access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a))
0)) = size-td t
```

```
      using wf-fd [where 'a = 'a]
      by (simp add: length-fa-ti size-of-def std)

    show length (replicate (size-of TYPE('b)) 0) = size-td (typ-info-t TYPE('b))
      by (simp add: size-of-def)

    have unat (x − &(p→f)) = unat ((x − ptr-val p) − of-nat n)
      by (simp add: field-lvalue-def field-lookup-offset-eq [OF fl])
    also have ... = unat (x − ptr-val p) − n
      by (metis ofn unat-sub uofn)
    finally have unat (x − &(p→f)) = unat (x − ptr-val p) − n .

   thus access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a)) 0) ! unat
(x − &(p→f)) =
       access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a)) 0) ! (unat
(x − ptr-val p) − n)
      by simp
  qed

  thus to-bytes-p v ! unat (x − &(p→f)) = ?RHS
    apply (subst heap-update-mem-same-point, simp-all)
  proof −
    show x ∈ {ptr-val p..+size-of TYPE('b)} using fl xin
      by (rule subsetD [OF field-tag-sub])
  qed
 next
  assume xni: x ∉ {&(p→f)..+size-td t}
  have ?RHS = (if x ∈ {ptr-val p..+size-of TYPE('b)}
      then (to-bytes-p (update-ti t (to-bytes-p v) (h-val hp p))) ! unat (x − ptr-val
p) else hp x)
    by (simp add: heap-update-mem-same-point heap-update-nmem-same)

  also
  {
    assume xin: x ∈ {ptr-val p..+size-of TYPE('b)}

    hence access-ti (typ-info-t TYPE('b))
     (update-ti-t t (access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a))
0)) (h-val hp p))
      (replicate (size-of TYPE('b)) 0) ! unat (x − ptr-val p) = hp x
    proof (subst field-access-update-nth-disjD [OF fl])
      have x − ptr-val p ≤ of-nat (size-of TYPE('b) − 1)
      proof (rule word-diff-ls(4)[where xa=x and x=x for x, simplified])
        from xin show x ≤ of-nat (size-of TYPE('b) − 1) + ptr-val p using al
szb
          by (subst add.commute, rule intvl-less-upper)
        show ptr-val p ≤ x using xin al szb
          by (rule intvl-le-lower)
      qed
```

```
      thus unx: unat (x − ptr-val p) < size-td (typ-info-t TYPE('b)) using szb
b0
       apply (simp)
       by (metis nat-le-Suc-less add.right-neutral b0 id-apply
         len-of-addr-card max-size neq0-conv of-nat-Suc of-nat-eq-id size-of-def
         unat-of-nat-minus-1 word-less-eq-iff-unsigned)

     show unat (x − ptr-val p) < n − 0 ∨ n − 0 + size-td t ≤ unat (x − ptr-val
p) using xin xni
       unfolding field-lvalue-def field-lookup-offset-eq [OF fl]
       apply −
       apply (erule intvl-cut)
        apply simp
       apply (rule max-size)
       done

      show wf-fd (typ-info-t TYPE('b)) by (rule wf-fd)

     show length (access-ti (typ-info-t TYPE('a)) v (replicate (size-of TYPE('a))
0)) = size-td t
       using wf-fd [where 'a = 'a]
       by (simp add: length-fa-ti size-of-def std)

     show length (replicate (size-of TYPE('b)) 0) = size-td (typ-info-t TYPE('b))
       by (simp add: size-of-def)

       have heap-list hp (size-td (typ-info-t TYPE('b))) (ptr-val p) ! unat (x −
ptr-val p) = hp x
       apply (subst heap-list-nth)
        apply (rule unx)
       apply simp
       done

     thus access-ti (typ-info-t TYPE('b)) (h-val hp p) (replicate (size-of TYPE('b))
0) ! unat (x − ptr-val p) = hp x
       unfolding h-val-def
      by (simp add: from-bytes-def update-ti-t-def size-of-def field-access-update-same(1)[OF
td-fafu-idem wf-fd])
     qed
   }
   hence ... = hp x
    by (simp add: to-bytes-p-def to-bytes-def update-ti-update-ti-t length-fa-ti [OF
wf-fd] std size-of-def)
   finally show hp x = ?RHS by simp
 qed
 finally show ?LHS = ?RHS .
qed
```

13.6.3 Proof automation for packed types

definition *td-packed* :: (*'a*,*'b*) *typ-info* ⇒ *nat* ⇒ *nat* ⇒ *bool*
  where *td-packed t sz al* ⟷
    td-fafu-idem t ∧ *td-fa-hi t* ∧ *aggregate t* ∧ *size-td t* = *sz* ∧ *align-td t* = *al*

lemma *packed-type-class-intro*:
  td-packed (*typ-info-t TYPE*(*'a*::*mem-type*)) *s a*
    ⟹ *OFCLASS*(*'a*::*mem-type*, *packed-type-class*)
  by *standard* (*simp-all add*: *td-packed-def*)

lemma *td-fa-hi-map-align*[*simp*]:*td-fa-hi* (*map-align f t*) = *td-fa-hi t*
  by (*cases t*) *auto*

lemma *td-packed-final-pad*:
  ⟦*td-packed t s a*; *2* ^ (*max algn a*) *dvd s*⟧ ⟹ *td-packed* (*final-pad algn t*) *s* (*max algn a*)
  by (*simp add*: *padup-dvd* [*symmetric*] *td-packed-def final-pad-def*)

lemma *td-packed-final-pad'*:
  assumes *packed-t*: *td-packed t s a*
  assumes *le*: *algn* ≤ *a*
  assumes *dvd*: *2* ^ *a dvd s*
  shows *td-packed* (*final-pad algn t*) *s a*
proof −
  from *le* **have** *max algn a* = *a* **by** *simp*
  from *td-packed-final-pad*[*OF packed-t*, *of algn*, *simplified this*, *OF dvd*]
  show *?thesis* **.**
qed

lemma *td-packed-ti-typ-combine*:
  ⟦ *td-packed* (*td*::*'a*::*c-type xtyp-info*) *s a*;
     align-of TYPE(*'b*::*packed-type*) *dvd s*; *fg-cons xf xfu*; *aggregate td* ⟧
    ⟹ *td-packed* (*ti-typ-combine TYPE*(*'b*) *xf xfu algn nm td*)
                (*s* + *size-td* (*typ-info-t TYPE*(*'b*)))
                (*max a* (*max algn* (*align-td* (*typ-info-t TYPE*(*'b*)))))
  unfolding *td-packed-def*
  apply *safe*
     apply (*rule td-fafu-idem-ti-typ-combine*; *assumption?*)
     apply (*rule td-fafu-idem*)
    apply (*rule td-fa-hi-ti-typ-combine*; *assumption?*)
    apply (*rule td-fa-hi*)
   apply *simp*
  apply (*simp only*: *size-td-lt-ti-typ-combine*)
  apply *simp*
  done

lemma *td-packed-ti-typ-pad-combine*:
  ⟦ *td-packed* (*td*::*'a*::*c-type xtyp-info*) *s a*;
     align-of TYPE(*'b*::*packed-type*) *dvd s*; *algn* ≤ *align-td* (*typ-info-t TYPE*(*'b*));

fg-cons xf xfu; *aggregate td*⟧
⟹ *td-packed* (*ti-typ-pad-combine TYPE*('*b*) *xf xfu algn nm td*)
(*s* + *size-td* (*typ-info-t TYPE*('*b*)))
(*max a* (*align-td* (*typ-info-t TYPE*('*b*))))
apply (*subgoal-tac padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*b*))) (*size-td td*) = *0*)
apply (*simp add*: *ti-typ-pad-combine-def Let-def td-packed-ti-typ-combine*)
apply (*auto simp add*: *padup-dvd td-packed-def packed-type-intro-simps size-td-lt-ti-typ-combine max-2-exp max-absorb2*)
done

lemma *td-packed-ti-typ-combine-array*:
⟦*td-packed* (*td*::'*a*::*c-type xtyp-info*) *s a*;
align-of TYPE('*b*::*packed-type*) *dvd s*; *0* < *CARD*('*n*); *algn* ≤ *align-td* (*typ-info-t TYPE*('*b*)); *fg-cons xf xfu*⟧
⟹ *td-packed*
(*ti-typ-combine TYPE*('*b* ['*n* :: *finite*]) *xf xfu algn nm td*)
(*s* + *size-td* (*typ-info-t TYPE*('*b*)) ∗ *CARD*('*n*))
(*max a* (*align-td* (*typ-info-t TYPE*('*b*))))
apply (*clarsimp simp*: *ti-typ-combine-def td-packed-def*
packed-type-intro-simps td-fafu-idem-extend-ti
td-fa-hi-extend-ti td-fa-hi-adjust-ti
size-td-extend-ti size-of-def
td-fafu-idem-adjust-ti
align-td-array-info max-absorb2)
done

lemma *td-packed-ti-typ-pad-combine-array*:
⟦ *td-packed* (*td*::'*a*::*c-type xtyp-info*) *s a*;
align-of TYPE('*b*::*packed-type*) *dvd s*; *0* < *CARD*('*n*); *algn* ≤ *align-td* (*typ-info-t TYPE*('*b*)); *fg-cons xf xfu* ⟧
⟹ *td-packed* (*ti-typ-pad-combine TYPE*('*b* ['*n* :: *finite*]) *xf xfu algn nm td*)
(*s* + *size-td* (*typ-info-t TYPE*('*b*)) ∗ *CARD*('*n*))
(*max a* (*align-td* (*typ-info-t TYPE*('*b*))))
apply (*subgoal-tac padup* (*align-of TYPE*('*b*['*n*])) (*size-td td*) = *0*)
apply (*clarsimp simp add*: *ti-typ-pad-combine-def Let-def td-packed-ti-typ-combine-array align-td-array-info align-of-def max-2-exp max-absorb2*)
apply (*simp add*: *td-packed-ti-typ-combine-array*)
apply (*simp add*: *align-of-def padup-dvd td-packed-def align-td-array*)
done

lemma *td-packed-empty-typ-info*:
td-packed (*empty-typ-info 0 fn*) *0 0*
apply (*unfold td-packed-def*, *safe*)
apply (*rule td-fafu-idem-empty-typ-info*)
apply (*rule td-fa-hi-empty-typ-info*)
apply (*rule aggregate-empty-typ-info*)
apply (*rule size-td-empty-typ-info*)
apply (*rule align-of-empty-typ-info'*)

done

lemmas *td-packed-intros* =
td-packed-final-pad
td-packed-empty-typ-info
td-packed-ti-typ-combine
td-packed-ti-typ-pad-combine
td-packed-ti-typ-combine-array
td-packed-ti-typ-pad-combine-array

end

13.7 Prettier Printing for Programs

theory *PrettyProgs*
imports *Simpl.Vcg*
begin

syntax (**output**)
-Assign :: *'b => 'b => ('a,'p,'f) com* (‹(*2- :==/ -*)› [*30, 30*] *23*)

-seq::*('s,'p,'f) com ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f) com* (‹-;;//-› [*20, 21*] *20*)

-While-inv :: *'a bexp => 'a assn => bdy => ('a,'p,'f) com*
(‹(*0WHILE* (-)//*INV* (-)//-)› [*25, 0, 81*] *71*)

-Do :: *('a,'p,'f) com ⇒ bdy* (‹*DO*// (-)//*OD*› [*0*] *1000*)

-Cond :: *'a bexp => ('a,'p,'f) com => ('a,'p,'f) com => ('a,'p,'f) com*
(‹(*0IF* - *THEN*// (-)//*ELSE*// (-)//*FI*)› [*0, 0, 0*] *71*)
-Cond-no-else:: *'a bexp => ('a,'p,'f) com => ('a,'p,'f) com*
(‹(*0IF* - *THEN*// (-)//*FI*)› [*0, 0*] *71*)

-Try-Catch:: *('a,'p,'f) com ⇒('a,'p,'f) com ⇒ ('a,'p,'f) com*
(‹(*0TRY*// (-)//*CATCH* -//*END*)› [*0,0*] *71*)

end

theory *StaticFun*
imports
Main
begin

datatype *('a, 'b) Tree = Node 'a 'b ('a, 'b) Tree ('a, 'b) Tree | Leaf*

primrec
lookup-tree :: *('a, 'b) Tree ⇒ ('a ⇒ 'c :: linorder) ⇒ 'a ⇒ 'b option*
where

lookup-tree Leaf fn x = None
| *lookup-tree (Node y v l r) fn x = (if fn x = fn y then Some v*
else if fn x < fn y then lookup-tree l fn x
else lookup-tree r fn x)

definition *optional-strict-range* :: *('a :: linorder) option ⇒ 'a option ⇒ 'a set*
where
optional-strict-range x y = {z. (x = None ∨ the x < z) ∧ (y = None ∨ z < the y)}

lemma *optional-strict-range-split*:
z ∈ optional-strict-range x y
⟹ optional-strict-range x (Some z) = ({..< z} ∩ optional-strict-range x y)
∧ optional-strict-range (Some z) y = ({z <..} ∩ optional-strict-range x y)
by *(auto simp add: optional-strict-range-def)*

lemma *optional-strict-rangeI*:
z ∈ optional-strict-range None None
z < y ⟹ z ∈ optional-strict-range None (Some y)
x < z ⟹ z ∈ optional-strict-range (Some x) None
x < z ⟹ z < y ⟹ z ∈ optional-strict-range (Some x) (Some y)
by *(simp-all add: optional-strict-range-def)*

definition
tree-eq-fun-in-range :: *('a, 'b) Tree ⇒ ('a ⇒ 'c :: linorder) ⇒ ('a ⇀ 'b) ⇒ 'c set ⇒ bool*
where
tree-eq-fun-in-range T ord f S ≡ ∀ x. (ord x ∈ S) ⟶ f x = lookup-tree T ord x

lemma *tree-eq-fun-in-range-from-def*:
⟦ f ≡ lookup-tree T ord ⟧
⟹ tree-eq-fun-in-range T ord f (optional-strict-range None None)
by *(simp add: tree-eq-fun-in-range-def)*

lemma *tree-eq-fun-in-range-split*:
tree-eq-fun-in-range (Node z v l r) ord f (optional-strict-range x y)
⟹ ord z ∈ optional-strict-range x y
⟹ tree-eq-fun-in-range l ord f (optional-strict-range x (Some (ord z)))
∧ f z = Some v
∧ tree-eq-fun-in-range r ord f (optional-strict-range (Some (ord z)) y)
apply *(simp add: tree-eq-fun-in-range-def optional-strict-range-split)*
apply *fastforce*
done

ML ‹

structure StaticFun = struct

(∗ Actually build the tree −− theta (n lg(n)) ∗)

```
fun build-tree' - mk-leaf [] = mk-leaf
  | build-tree' mk-node mk-leaf xs = let
    val len = length xs
    val (ys, zs) = chop (len div 2) xs
  in case zs of [] => error build-tree': impossible
    | ((a, b) :: zs) => mk-node a b (build-tree' mk-node mk-leaf ys)
        (build-tree' mk-node mk-leaf zs)
  end

fun build-tree ord xs = case xs of [] => error build-tree : empty
  | (idx, v) :: - => let
    val idxT = fastype-of idx
    val vT = fastype-of v
    val treeT = Type (@{type-name StaticFun.Tree}, [idxT, vT])
    val mk-leaf = Const (@{const-name StaticFun.Leaf}, treeT)
    val node = Const (@{const-name StaticFun.Node},
        idxT --> vT --> treeT --> treeT --> treeT)
    fun mk-node a b l r = node $ a $ b $ l $ r
    val lookup = Const (@{const-name StaticFun.lookup-tree},
        treeT --> fastype-of ord --> idxT
            --> Type (@{type-name option}, [vT]))
  in
    lookup $ (build-tree' mk-node mk-leaf xs) $ ord
  end

fun define-partial-map-tree name mappings ord ctxt = let
    val tree = build-tree ord mappings
  in Local-Theory.define
    ((name, NoSyn), ((Thm.def-binding name, []), tree)) ctxt
    |> apfst (apsnd snd)
  end

fun prove-partial-map-thms thm ctxt = let
    val init = thm RS @{thm tree-eq-fun-in-range-from-def}
    fun rec-tree thm = case Thm.concl-of thm of
    @{term Trueprop} $ (Const (@{const-name tree-eq-fun-in-range}, -)
        $ (Const (@{const-name Node}, -) $ z $ v $ - $ -) $ - $ - $ -) => let
        val t' = thm RS @{thm tree-eq-fun-in-range-split}
        val solve-simp-tac = SUBGOAL (fn (t, i) =>
            (simp-tac ctxt THEN-ALL-NEW SUBGOAL (fn (t', -) =>
                raise TERM (prove-partial-map-thms: unsolved, [t, t']))) i)
        val r = t' |> (resolve-tac ctxt @{thms optional-strict-rangeI}
            THEN-ALL-NEW solve-simp-tac) 1 |> Seq.hd
        val l = r RS @{thm conjunct1}
        val kr = r RS @{thm conjunct2}
        val k = kr RS @{thm conjunct1}
        val r = kr RS @{thm conjunct2}
      in rec-tree l @ [((z, v), k)] @ rec-tree r end
    | - => []
```

```
  in rec-tree init end

fun define-tree-and-save-thms name names mappings ord exsimps ctxt = let
    val ((tree, def-thm), ctxt) = define-partial-map-tree name mappings ord ctxt
    val thms = prove-partial-map-thms def-thm (ctxt addsimps exsimps)
    val (idents, thms) = map-split I thms
    val - = map (fn ((x, y), (x', y')) => (x aconv x' andalso y aconv y')
        orelse raise TERM (define-tree-and-thms: different, [x, y, x', y']))
        (mappings ~~ idents)
    val (-, ctxt) = Local-Theory.notes
        (map (fn (s, t) => ((Binding.name s, []), [([t], [])]))
        (names ~~ thms)) ctxt
  in (tree, ctxt) end

fun define-tree-and-thms-with-defs name names key-defs opt-values ord ctxt = let
    val data = names ~~ (key-defs ~~ opt-values)
        |> map-filter (fn (-, (-, NONE)) => NONE | (nm, (thm, SOME v))
            => SOME (nm, (fst (Logic.dest-equals (Thm.concl-of thm)), v)))
    val (names, mappings) = map-split I data
  in define-tree-and-save-thms name names mappings ord key-defs ctxt end

end

›
```

end

theory *IndirectCalls*

imports
  PrettyProgs

begin

lemma *hoare-indirect-call-known-proc*:
assumes *spec*: $\Gamma \vdash P$ (*call-exn init q return result-exn c*) Q,A
shows $\Gamma\vdash$ ($\{s.\ p\ s = q\} \cap P$) (*dynCall-exn f UNIV init p return result-exn c*) Q,A
  using *spec*
  apply −
  apply (*rule hoare-complete*, *drule hoare-sound*)
  apply (*clarsimp simp*: *cvalid-def HoarePartialDef.valid-def dynCall-exn-def*)
  apply (*auto elim*: *exec-Normal-elim-cases*)
  done

lemma *hoare-indirect-call-guard*:

assumes *conseq*: $P \subseteq g \cap R$
assumes *spec*: $\Gamma \vdash R$ (*dynCall-exn f UNIV init p return result-exn c*) Q,A
shows $\Gamma\vdash$ P (*dynCall-exn f g init p return result-exn c*) Q,A
using *spec conseq*
apply −
apply (*rule hoare-complete*, *drule hoare-sound*)
apply (*clarsimp simp*: *cvalid-def HoarePartialDef.valid-def dynCall-exn-def maybe-guard-def*)
apply (*cases g* = *UNIV*)
subgoal
by *force*
subgoal
apply (*clarsimp elim*: *exec-Normal-elim-cases*)
by (*meson exec-Normal-elim-cases*(5) *imageI subsetD*)
done

end

theory *ModifiesProofs*
imports *CLanguage*
begin

definition
modifies-inv-refl :: (′*a* ⇒ ′*a set*) ⇒ *bool*
where
modifies-inv-refl P ≡ ∀ *x*. *x* ∈ *P x*

definition
modifies-inv-incl :: (′*a* ⇒ ′*a set*) ⇒ *bool*
where
modifies-inv-incl P ≡ ∀ *x y*. *y* ∈ *P x* ⟶ *P y* ⊆ *P x*

definition
modifies-inv-prop :: (′*a* ⇒ ′*a set*) ⇒ *bool*
where
modifies-inv-prop P ≡ *modifies-inv-refl P* ∧ *modifies-inv-incl P*

lemma *modifies-inv-prop*:
modifies-inv-refl P ⟹ *modifies-inv-incl P* ⟹ *modifies-inv-prop P*
by (*simp add*: *modifies-inv-prop-def*)

named-theorems *modifies-inv-intros*

locale *modifies-assertion* =
fixes *P* :: ′*s* ⇒ ′*s set*
assumes *p*: *modifies-inv-prop P*
begin

lemmas *modifies-inv-prop'* =
p[*unfolded modifies-inv-prop-def modifies-inv-refl-def modifies-inv-incl-def*]

lemma *modifies-inv-prop-lift*:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} (P\ \sigma)\ c\ (P\ \sigma),(P\ \sigma)$
using *modifies-inv-prop'* **by** (*fastforce intro*: *c hoarep.Conseq*)

lemma *modifies-inv-prop-lower*:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} (P\ \sigma)\ c\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
using *modifies-inv-prop'* **by** (*fastforce intro*: *c hoarep.Conseq*)

Note that the C-Parser associates sequential composition to the right. So the first statement is typically already an 'atomic' statement (or at least no further sequential composition) that can be solved. We place it as the second precondition because the modifies-tactic follows the canonical order of tactical reasoning and solves the subgoals from the back. So *c1* is already solved before further decomposing *c2*. This keeps the number of subgoals (and thus the overall goal-state) small.

lemma *modifies-inv-Seq* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c2\ (P\ \sigma),(P\ \sigma)$ $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c1\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c1\ ;;\ c2\ (P\ \sigma),(P\ \sigma)$
by (*intro modifies-inv-prop-lower HoarePartial.SeqSwap*[*OF c*[*THEN modifies-inv-prop-lift*]])

lemma *modifies-inv-Cond* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c1\ (P\ \sigma),(P\ \sigma)$ $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c2\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ Cond\ b\ c1\ c2\ (P\ \sigma),(P\ \sigma)$
by (*auto intro*: *HoarePartial.Cond c*)

lemma *modifies-inv-Guard-strip* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ Guard\ f\ b\ c\ (P\ \sigma),(P\ \sigma)$
by (*rule HoarePartial.GuardStrip*[*OF subset-refl c UNIV-I*])

lemma *modifies-inv-Skip* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ SKIP\ (P\ \sigma),(P\ \sigma)$
using *modifies-inv-prop'* **by** (*auto intro*: *modifies-inv-prop-lift HoarePartial.Skip*)

lemma *modifies-inv-Skip'* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ SKIP\ (P\ \sigma)$
using *modifies-inv-prop'* **by** (*auto intro*: *modifies-inv-prop-lift HoarePartial.Skip*)

lemma *modifies-inv-whileAnno* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$

shows $\Gamma,\Theta\vdash_{/F}\{\sigma\}$ *whileAnno b I V c* $(P\ \sigma),(P\ \sigma)$
apply (*rule HoarePartial.reannotateWhileNoGuard*[**where** $I{=}P\ \sigma$])
by (*intro HoarePartial.While hoarep.Conseq*;
fastforce simp: *modifies-inv-prop′ intro*: *modifies-inv-prop-lift c*)

lemma *modifies-inv-While* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F}\{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}\{\sigma\}$ *While b c* $(P\ \sigma),(P\ \sigma)$
by (*intro modifies-inv-whileAnno*[*unfolded whileAnno-def*] *c*)

lemma *modifies-inv-Throw* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F}\{\sigma\}$ *THROW* $(P\ \sigma),(P\ \sigma)$
using *modifies-inv-prop′* **by** (*auto intro*: *modifies-inv-prop-lift HoarePartial.Throw*)

lemma *modifies-inv-Catch* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F}\{\sigma\}$ *c1* $(P\ \sigma),(P\ \sigma)$
$\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F}\{\sigma\}$ *c2* $(P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}\{\sigma\}$ *TRY c1 CATCH c2 END* $(P\ \sigma),(P\ \sigma)$
by (*intro modifies-inv-prop-lower HoarePartial.Catch*[*OF c*[*THEN modifies-inv-prop-lift*]])

lemma *modifies-inv-Catch-all* [*modifies-inv-intros*]:
assumes *1*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F}\{\sigma\}$ *c1* $(P\ \sigma),(P\ \sigma)$
assumes *2*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F}\{\sigma\}$ *c2* $(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}\{\sigma\}$ *TRY c1 CATCH c2 END* $(P\ \sigma)$
apply (*intro HoarePartial.Catch*[*OF 1*] *hoarep.Conseq*, *clarsimp*)
apply (*metis modifies-inv-prop′ 2 singletonI*)
done

lemma *modifies-inv-switch-Nil* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F}\{\sigma\}$ *switch v* [] $(P\ \sigma),(P\ \sigma)$
by (*auto intro*: *modifies-inv-Skip*)

lemma *modifies-inv-switch-Cons* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F}\{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
$\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F}\{\sigma\}$ *switch p vcs* $(P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}\{\sigma\}$ *switch p* $((v,c)\ \#\ vcs)\ (P\ \sigma),(P\ \sigma)$
by (*auto intro*: *c modifies-inv-Cond*)

end

locale *modifies-state-assertion* = *modifies-assertion P* **for**
P :: $('g,\ 'l,\ 'e,\ 'x)$ *state-scheme* ⇒ $('g,\ 'l,\ 'e,\ 'x)$ *state-scheme set* +
assumes *p*: *modifies-inv-prop P*
begin

lemma *modifies-inv-creturn* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F}\{\sigma\}$ *Basic* $(\lambda s.\ xfu\ (\lambda\text{-}.\ v\ s)\ s)\ (P\ \sigma),(P\ \sigma)$

$\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* (*rtu* (λ-. *Return*)) (P σ),(P σ)
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *creturn rtu xfu v* (P σ),(P σ)
unfolding *creturn-def* **by** (*intro p c modifies-inv-intros*)

lemma *modifies-inv-creturn-void* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* (*rtu* (λ-. *Return*)) (P σ),(P σ)
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *creturn-void rtu* (P σ),(P σ)
unfolding *creturn-void-def* **by** (*intro p c modifies-inv-intros*)

lemma *modifies-inv-cbreak* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* (*rtu* (λ-. *Break*)) (P σ),(P σ)
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *cbreak rtu* (P σ),(P σ)
unfolding *cbreak-def* **by** (*intro p c modifies-inv-intros*)

lemma *modifies-inv-ccatchbrk* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *ccatchbrk rt* (P σ),(P σ)
unfolding *ccatchbrk-def* **by** (*intro p modifies-inv-intros*)

lemma *modifies-inv-cgoto* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma$. $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* (*rtu* (λ-. *Goto l*)) (P σ),(P σ)
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *cgoto l rtu* (P σ),(P σ)
unfolding *cgoto-def* **by** (*intro p c modifies-inv-intros*)

lemma *modifies-inv-ccatchgoto* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *ccatchgoto l rt* (P σ),(P σ)
unfolding *ccatchgoto-def* **by** (*intro p modifies-inv-intros*)

lemma *modifies-inv-ccatchreturn* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *ccatchreturn rt* (P σ),(P σ)
unfolding *ccatchreturn-def* **by** (*intro p modifies-inv-intros*)

end

lemma *On-Exit-wp*:
assumes *cleanup*: $\Gamma,\Theta\vdash_{/F} R$ *cleanup* Q, A
assumes *cleanup-catch*: $\Gamma,\Theta\vdash_{/F} B$ *cleanup* A, A
assumes *c*: $\Gamma,\Theta\vdash_{/F} P$ *c* R, B
shows $\Gamma,\Theta\vdash_{/F} P$ *On-Exit c cleanup* Q, A
unfolding *On-Exit-def*
apply (*rule HoarePartial.SeqSwap* [**where** R=R])
apply (*rule HoarePartial.conseq-no-aux*)
apply (*rule cleanup*)
apply *simp*
apply (*rule HoarePartial.Catch*)
apply (*rule c*)
apply (*rule HoarePartial.SeqSwap*)
apply (*rule HoarePartial.Throw*)

apply (*rule subset-refl*)
apply (*rule cleanup-catch*)
done

lemma *DynCom-fix-pre*: $\forall s \in P.\ \Gamma,\Theta\vdash_{/F} \{s\}\ (c\ s)\ Q,A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P\ (DynCom\ c)\ Q,A$
by (*smt* (*verit*, *best*) *HoarePartial.conseq-exploit-pre Int-insert-left-if1 hoarep.DynCom inf-bot-left singletonD*)

lemma $\Gamma,\Theta\vdash_{/F} \{s.\ (\forall t.\ (s,\ t) \in r \longrightarrow t \in Q) \wedge (\exists t.\ (s,\ t) \in r)\}$ *Spec r Q*,*A*
by (*rule hoarep.Spec*)

lemma *hoarep-Spec-fixed*: $(\exists t.\ (s,\ t) \in r) \Longrightarrow \Gamma,\Theta\vdash_{/F}\{s\}$ *Spec r* $\{t.\ (s,\ t) \in r\}$,*A*
by (*simp add*: *HoarePartial.Spec*)

context *stack-heap-state*
begin

lemma *With-Fresh-Stack-Ptr-tight*:
assumes *c*: $\bigwedge s\ p\ d\ vs\ bs.\ s \in P \Longrightarrow vs \in init\ s \Longrightarrow length\ vs = n \Longrightarrow length\ bs = n * size\text{-}of\ TYPE('a) \Longrightarrow$
$(p,\ d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a::mem\text{-}type)\ (htd\ s) \Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\{hmem\text{-}upd\ (fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (vs!i)\ (take\ (size\text{-}of\ TYPE('a))\ (drop\ (i * size\text{-}of\ TYPE('a))\ bs)))\ [0..<n])\ (htd\text{-}upd\ (\lambda\text{-}.\ d)\ s)\}$
$(c\ p)$
$\{t.\ \forall bs.\ length\ bs = n * size\text{-}of\ TYPE('a) \longrightarrow hmem\text{-}upd\ (heap\text{-}update\text{-}list\ (ptr\text{-}val\ p)\ bs)\ (htd\text{-}upd\ (stack\text{-}releases\ n\ p)\ t) \in Q\}$,
$\{t.\ \forall bs.\ length\ bs = n * size\text{-}of\ TYPE('a) \longrightarrow hmem\text{-}upd\ (heap\text{-}update\text{-}list\ (ptr\text{-}val\ p)\ bs)\ (htd\text{-}upd\ (stack\text{-}releases\ n\ p)\ t) \in A\}$
assumes *no-overflow*: *StackOverflow* $\in F$
shows $\Gamma,\Theta\vdash_{/F} P$ *With-Fresh-Stack-Ptr n init c Q*, *A*
unfolding *With-Fresh-Stack-Ptr-def*
apply (*rule hoarep.Guarantee* [*OF no-overflow*])
apply (*rule DynCom-fix-pre*)
apply (*rule ballI*)
apply *clarsimp*
apply (*rule HoarePartial.Seq*)
apply (*rule hoarep-Spec-fixed*)
apply *clarsimp*
subgoal using *Ex-list-of-length*
by (*metis equals0I old.prod.exhaust*)

apply *clarsimp*
apply (*rule DynCom-fix-pre*)
apply (*rule ballI*)

apply *clarsimp*
apply (*rule On-Exit-wp*)

apply (*rule HoarePartial.Spec*)
apply (*rule subset-refl*)
apply (*rule HoarePartial.Spec*)
apply (*rule subset-refl*)
subgoal premises *prems* **for** *s vs p d' vsa bs*
proof −
have *eq1*: *length vs* = *n* **using** *prems* **by** *simp*
have *eq2*:(*allocated-ptrs n* $\mathcal{S}$ *TYPE*('*a*) (*htd s*) *d'*) = *p* **using** *prems* **by** (*simp add*: *stack-allocs-allocated-ptrs*)
show *?thesis*
apply (*simp add*: *eq1 eq2 Ex-list-of-length*)
apply (*rule HoarePartialDef.conseqPost* [*OF c* [**where** *s1*=*s* **and** *vs1*=*vsa* **and** *d1*=*d'* **and** *p1*=*p*]])
using *prems*
apply *auto*
done
qed
done

lemma *With-Fresh-Stack-Ptr-tight-wp*:
assumes *conseq*: $\bigwedge$*s p d vs bs*. *s* ∈ *P* ⟹ *vs* ∈ *init s* ⟹ *length vs* = *n* ⟹ *length bs* = *n* ∗ *size-of TYPE*('*a*) ⟹
(*p*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) (*htd s*) ⟹
(*hmem-upd* (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*vs*!*i*) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*))) [*0*..<*n*]) (*htd-upd* (λ-. *d*) *s*)) ∈ *R s p d vs*
assumes *c*: $\bigwedge$*s p d vs bs*. *s* ∈ *P* ⟹ *vs* ∈ *init s* ⟹ *length vs* = *n* ⟹ *length bs* = *n* ∗ *size-of TYPE*('*a*) ⟹
(*p*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) (*htd s*) ⟹
(*hmem-upd* (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*vs*!*i*) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*))) [*0*..<*n*]) (*htd-upd* (λ-. *d*) *s*)) ∈ *R s p d vs* ⟹
Γ,Θ⊢$_{/F}$(*R s p d vs*) (*c p*)
{*t*. ∀ *bs*. *length bs* = *n* ∗ *size-of TYPE*('*a*) ⟶ *hmem-upd* (*heap-update-list* (*ptr-val p*) *bs*) (*htd-upd* (*stack-releases n p*) *t*)) ∈ *Q*},
{*t*. ∀ *bs*. *length bs* = *n* ∗ *size-of TYPE*('*a*) ⟶ *hmem-upd* (*heap-update-list* (*ptr-val p*) *bs*) (*htd-upd* (*stack-releases n p*) *t*)) ∈ *A*}
assumes *no-overflow*: *StackOverflow* ∈ *F*
shows Γ,Θ⊢$_{/F}$ *P With-Fresh-Stack-Ptr n init c Q*, *A*
apply (*rule With-Fresh-Stack-Ptr-tight* [*OF - no-overflow*])
subgoal for *s p d vs bs*
apply (*rule HoarePartial.conseq*)
apply (*rule allI*)

```
    apply (rule c[where s1=s and vs1=vs and d1=d])
        apply assumption
       apply assumption
      apply assumption
     apply assumption
    apply assumption
   apply (rule conseq)
       apply assumption
      apply assumption
     apply assumption
    apply assumption
   apply clarsimp
  subgoal premises prems
  proof −
    from prems have eq: length vs = n by simp
    show ?thesis
      apply (simp add: eq)
      apply (rule conseq)
      using prems by auto
  qed
  done
done
```

Caution: this WP-setup was developed to solve the modified clauses. It might not be the best fit for WP-style reasoning in general. Also note that it is currently not invoked by the automatic modifies-proofs triggerd by the C-parser as *With-Fresh-Stack-Ptr* is first decomposed by the (see the rule below).

```
declaration ‹fn phi =>
let
  val rule = Morphism.thm phi @{thm With-Fresh-Stack-Ptr-tight}
  fun tac cont-tac ctxt mode i =
      resolve-tac ctxt [rule] i THEN
      SOLVED-verbose no-overflow ctxt (simp-tac ctxt) (i + 1) THEN
      cont-tac true i
in
   Hoare.add-hoare-tacs [(With-Fresh-Stack-Ptr-tac, tac)]
end
›

end

locale modifies-assertion-stack-heap-state =
  modifies-assertion P + stack-heap-state htd htd-upd hmem hmem-upd 𝒮
  for P::'s ⇒ 's set and
    htd:: 's ⇒ heap-typ-desc and
    htd-upd:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's ⇒ 's and
    hmem:: 's ⇒ heap-mem and
    hmem-upd:: (heap-mem ⇒ heap-mem) ⇒ 's ⇒ 's  and
```

```
    𝒮::addr set +
  assumes hmem-upd-inv: ⋀σ m. hmem-upd m σ ∈ (P σ)
  assumes htd-upd-inv: ⋀σ d. htd-upd d σ ∈ (P σ)
begin

lemma modifies-With-Fresh-Stack-Ptr [modifies-inv-intros]:
  assumes no-overflow: StackOverflow ∈ F
  assumes c: ⋀σ p. Γ,Θ⊢/F {σ} (c (p::'a::mem-type ptr)) (P σ),(P σ)
  shows Γ,Θ⊢/F {σ} With-Fresh-Stack-Ptr n init c (P σ), (P σ)
  apply (rule With-Fresh-Stack-Ptr-tight-wp [where R = λ s p d v. P s])
  subgoal for s p d vs bs
  proof −
    have (htd-upd (λ-. d) s) ∈ P s by (rule htd-upd-inv)
    moreover have hmem-upd (fold (λi. heap-update-padding (p +p int i) (vs!i)
(take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<n]) (htd-upd (λ-.
d) s) ∈ P (htd-upd (λ-. d) s)
      by (rule hmem-upd-inv)
    ultimately
    show hmem-upd (fold (λi. heap-update-padding (p +p int i) (vs!i) (take (size-of
TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<n]) (htd-upd (λ-. d) s) ∈ P s
      using modifies-inv-prop' p by blast
  qed
  subgoal for s p d v
    apply (rule HoarePartial.conseq [where P'=P and Q'=P and A'=P])
   apply (rule allI)
    subgoal for Z
      apply (rule modifies-inv-prop-lift)
      apply (rule c)
      done
    using modifies-inv-prop' hmem-upd-inv htd-upd-inv
    by (smt (verit, best) mem-Collect-eq singletonD subset-eq)
  apply (rule no-overflow)
  done

end



locale modifies-assertion-stack-heap-raw-state =
  modifies-assertion P + stack-heap-raw-state t-hrs t-hrs-update 𝒮
  for P::'s ⇒ 's set and
    t-hrs:: 's ⇒ heap-raw-state and
    t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's and
    𝒮::addr set +
  assumes hrs-upd-inv: ⋀σ m. t-hrs-update m σ ∈ (P σ)
begin

sublocale modifies-assertion-stack-heap-state
  P
```

λs. hrs-htd (t-hrs s) λupd. t-hrs-update (hrs-htd-update upd)
λs. hrs-mem (t-hrs s) λupd. t-hrs-update (hrs-mem-update upd)
S
using *hrs-upd-inv* **by** *unfold-locales auto*

end

end

13.8 Modelling Local Variables

theory *CLocals*
imports *UMM*
HOL−Library.Code-Binary-Nat
ML-Record-Antiquotation
begin

ML ‹
structure Code-Simproc =
struct

val meta-eq-to-prop = @{lemma ‹(PROP P ≡ Trueprop True) ⟹ PROP P› for P by simp}

fun solved-eq eq =
case Thm.prop-of eq of
@{term-pat - ≡ Trueprop True} => true
| - => false

fun code-prove ctxt prop =
let
val code-eq = SOME (Code-Runtime.dynamic-holds-conv ctxt prop) handle ERROR x => (warning x; NONE)
val - = Utils.verbose-msg 4 ctxt (fn - => code-prove: ^ @{make-string} code-eq)
in
code-eq |> Option.mapPartial (fn eq =>
if solved-eq eq then
SOME (meta-eq-to-prop OF [Code-Simp.dynamic-conv ctxt prop])
else NONE)
end

fun fill-default default xs =
let
fun fill ys - [] = rev ys
| fill ys n (zs as ((i, z)::zs')) =
if n = i then
fill (z::ys) (n + 1) zs'

```
        else
          if n < i then fill (default::ys) (n + 1) zs
        else error (fill-default: expecting indexes in ascending order ^@{make-string}
(map fst xs))
  in
    fill [] 0 xs
  end

fun code-simp-prems cond-eq-rule0 ps ctxt ct =
  let
    val cond-eq-rule = Thm.transfer' ctxt cond-eq-rule0
    val - = Utils.verbose-msg 4 ctxt (fn - => code-simp-prems:  ^@{make-string}
(ct, cond-eq-rule))
    val ps = sort int-ord ps
    val lhs = cond-eq-rule |> Thm.cconcl-of |> Thm.dest-equals-lhs
    val eq = Thm.instantiate (Thm.match (lhs, ct)) cond-eq-rule
    val prems = Thm.cprems-of eq
    fun solve i = code-prove ctxt (nth prems i)
    val solved = map-filter solve ps
  in
    if length solved = length ps then
      SOME (eq OF (fill-default Drule.asm-rl (ps ~~ solved)))
    else
      NONE
  end

fun is-eq (@{term-pat Trueprop (- = -)}) = true
  | is-eq (@{term-pat - ≡ -}) = true
  | is-eq - = false

fun positions-of len interval =
  case interval of
    Facts.FromTo (i, j) => (@{assert} (j <= len) ; (i - 1) upto (j - 1))
  | Facts.From i => (@{assert} (i <= len) ; (i - 1) upto (len - 1))
  | Facts.Single i => (@{assert} (i <= len); [i - 1])

fun positions-of-intervals len =
  map (positions-of len) #> flat #> sort-distinct int-ord

fun lhs-pattern ctxt eq =
 let
   val ((-, [eq]), -) = Variable.import false [eq] ctxt
 in
   eq |> Thm.concl-of |> Logic.dest-equals |> fst
 end

structure Data = Generic-Data (
  type T = simproc list Symtab.table
  val empty = Symtab.empty
```

```
  val merge = Symtab.merge (K false))

fun get-simprocs ctxt named-thms =
 let
   val simprocs = Data.get (Context.Proof ctxt)
 in
   Symtab.lookup-list simprocs named-thms
 end

(* piggy back name−space of named theorems *)
fun check context named-thms =
  let
    val ctxt = Context.proof-of context
    val facts = (Proof-Context.facts-of ctxt);
  in
    Facts.check context facts named-thms
  end

fun add-simproc pos named-thms (intervals: Facts.interval list) thm context =
  let
    val ctxt = Context.proof-of context
    val named-thms = if fst (named-thms) =  then  else (check context named-thms
)
    val - = is-eq (Thm.concl-of thm) orelse error (Code-Simproc.add-simproc: con-
clusion is not an equality)
    val - = Thm.has-name-hint thm orelse error (Code-Simproc.add-simproc: theo-
rem has no name)
    val name-hint = Thm-Name.short (Thm.get-name-hint thm)
    val base-name = Long-Name.base-name name-hint
    val thm1 = safe-mk-meta-eq thm
    val nprems = Thm.nprems-of thm1
    val - = nprems > 0 orelse error (Code-Simproc.add-simprocs: theorem has no
premises)
   val positions = if null intervals then 0 upto nprems − 1 else positions-of-intervals
nprems intervals
    val patterns = thm1 |> lhs-pattern ctxt |> single
    val thm2 = Thm.trim-context thm1
  in
    if Local-Theory.level ctxt = 0 then context
    else
      context
      |> Context.map-proof-result (Simplifier.define-simproc
          {name = Binding.make (base-name, pos), passive=false, kind = Simproc,
identifier=[],
           lhss = patterns, proc = fn - => code-simp-prems thm2 positions})
      |−> (fn simproc => Context.map-proof (
            Local-Theory.declaration {pervasive=false, syntax=false, pos = here}
(fn - =>
              (Data.map (Symtab.map-default (named-thms, [simproc]) (cons sim-
```

```
proc))))))
  end
end
›


setup ‹
let
  fun position scan = (Scan.ahead (Scan.one (K true)) >> Token.pos-of) -- scan
>> Library.swap;
in
  Attrib.setup binding ‹code-simproc›
    (Scan.lift (position
    (Scan.optional Parse.name-position (, Position.none) -- Scan.optional Parse.thm-sel
[]) >>
    (fn ((named-thms, intervals), pos) =>
     Thm.declaration-attribute (fn thm => Code-Simproc.add-simproc pos named-thms
intervals thm))))
    solve preconditions with code-simp #>
  ML-Antiquotation.value-embedded binding ‹code-simprocs›
    (Args.context -- Scan.lift Parse.embedded-position >>
      (fn (ctxt, name) => Code-Simproc.get-simprocs ML-context ^
        ML-Syntax.print-string (Code-Simproc.check (Context.Proof ctxt) name)))
end
›

ML ‹
String.str
›
type-synonym locals = nat ⇒ byte list


definition clookup :: nat ⇒ locals ⇒ 'a::mem-type where
  clookup n l = from-bytes (l n)

definition cupdate :: nat ⇒ ('a::mem-type ⇒ 'a) ⇒ locals ⇒ locals where
  cupdate n f l = l(n := to-bytes (f (from-bytes (l (n)))) (replicate (size-of TYPE('a))
0))

lemma clookup-cupdate-same[simp, state-simp]: clookup n (cupdate n f l) = f
(clookup n l)
  by (simp add: clookup-def cupdate-def)

lemma clookup-cupdate-same-cond[code-simproc state-simp]:
  n = 0 ⟹ clookup n (cupdate 0 f l) = f (clookup n l)
  n = 0 ⟹ clookup 0 (cupdate n f l) = f (clookup n l)
  n = 1 ⟹ clookup n (cupdate 1 f l) = f (clookup n l)
  n = Suc 0 ⟹ clookup n (cupdate (Suc 0) f l) = f (clookup n l)
  n = 1 ⟹ clookup 1 (cupdate n f l) = f (clookup n l)
```

$n = Suc\ 0 \Longrightarrow clookup\ (Suc\ 0)\ (cupdate\ n\ f\ l) = f\ (clookup\ n\ l)$
$n = numeral\ m \Longrightarrow clookup\ n\ (cupdate\ (numeral\ m)\ f\ l) = f\ (clookup\ n\ l)$
$n = numeral\ m \Longrightarrow clookup\ (numeral\ m)\ (cupdate\ n\ f\ l) = f\ (clookup\ n\ l)$
by (*auto simp add*: *clookup-def cupdate-def*)

lemma *clookup-refl-cond*[*code-simproc state-simp*]:
$n = m \Longrightarrow clookup\ n\ l = clookup\ m\ l \longleftrightarrow True$
by (*simp-all add*: *clookup-def*)

lemma *clookup-cupdate-other*[*code-simproc state-simp*]: $n \neq m \Longrightarrow clookup\ n\ (cupdate\ m\ f\ l) = (clookup\ n\ l)$
by (*auto simp add*: *clookup-def cupdate-def*)

lemma *cupdate-compose*[*simp*, *state-simp*]: $cupdate\ n\ f\ (cupdate\ n\ g\ l) = cupdate\ n\ (f\ o\ g)\ l$
by (*simp add*: *cupdate-def*)

lemma *cupdate-compose-cond*[*code-simproc state-simp*]:
$n = 0 \Longrightarrow cupdate\ n\ f\ (cupdate\ 0\ g\ l) = cupdate\ n\ (f\ o\ g)\ l$
$n = 0 \Longrightarrow cupdate\ 0\ f\ (cupdate\ n\ g\ l) = cupdate\ n\ (f\ o\ g)\ l$
$n = 1 \Longrightarrow cupdate\ n\ f\ (cupdate\ 1\ g\ l) = cupdate\ n\ (f\ o\ g)\ l$
$n = Suc\ 0 \Longrightarrow cupdate\ n\ f\ (cupdate\ (Suc\ 0)\ g\ l) = cupdate\ n\ (f\ o\ g)\ l$
$n = Suc\ 0 \Longrightarrow cupdate\ (Suc\ 0)\ f\ (cupdate\ n\ g\ l) = cupdate\ n\ (f\ o\ g)\ l$
$n = numeral\ m \Longrightarrow cupdate\ n\ f\ (cupdate\ (numeral\ m)\ g\ l) = cupdate\ n\ (f\ o\ g)\ l$
$n = numeral\ m \Longrightarrow cupdate\ (numeral\ m)\ f\ (cupdate\ n\ g\ l) = cupdate\ n\ (f\ o\ g)\ l$
by (*auto simp add*: *cupdate-def*)

lemma *clookup-cupdate-other-numeral*[*simplified*, *simp*, *state-simp*]:
$clookup\ 0\ (cupdate\ 1\ f\ l) = (clookup\ 0\ l)$
$clookup\ 0\ (cupdate\ (numeral\ m)\ f\ l) = (clookup\ 0\ l)$
$clookup\ 1\ (cupdate\ 0\ f\ l) = (clookup\ 1\ l)$
$numeral\ m \neq (1{::}nat) \Longrightarrow$
$clookup\ 1\ (cupdate\ (numeral\ m)\ f\ l) = (clookup\ 1\ l)$

$clookup\ (numeral\ n)\ (cupdate\ 0\ f\ l) = (clookup\ (numeral\ n)\ l)$
$numeral\ n \neq (1{::}nat) \Longrightarrow$
$clookup\ (numeral\ n)\ (cupdate\ 1\ f\ l) = (clookup\ (numeral\ n)\ l)$

$n \neq m \Longrightarrow$
$clookup\ (numeral\ n)\ (cupdate\ (numeral\ m)\ f\ l) = (clookup\ (numeral\ n)\ l)$
by (*auto simp add*: *clookup-def cupdate-def*)

lemma *cupdate-commute*: $n \neq m \Longrightarrow cupdate\ n\ f\ (cupdate\ m\ g\ l) = cupdate\ m\ g\ (cupdate\ n\ f\ l)$
by (*auto simp add*: *cupdate-def fun-upd-def*)

lemma *cupdate-commute-ordered*[*code-simproc state-simp*]: $n < m \Longrightarrow cupdate\ n\ f\ (cupdate\ m\ g\ l) = cupdate\ m\ g\ (cupdate\ n\ f\ l)$

by (*auto simp add*: *cupdate-def fun-upd-def*)

lemma *cupdate-commute-numeral-simp*[*simplified*, *simp*, *state-simp*]:
cupdate 0 f (*cupdate 1 g l*) = *cupdate 1 g* (*cupdate 0 f l*)
cupdate 0 f (*cupdate* (*numeral m*) *g l*) = *cupdate* (*numeral m*) *g* (*cupdate 0 f l*)

numeral m ≠ (*1*::*nat*) ⟹
cupdate 1 f (*cupdate* (*numeral m*) *g l*) = *cupdate* (*numeral m*) *g* (*cupdate 1 f l*)

n < *m* ⟹ *cupdate* (*numeral n*) *f* (*cupdate* (*numeral m*) *g l*) = *cupdate* (*numeral m*) *g* (*cupdate* (*numeral n*) *f l*)
by (*auto simp add*: *cupdate-def fun-upd-def*)

lemma *const-compose* [*simp*, *state-simp*]:
cupdate n ((λ-. *x*) ∘ *f*) = *cupdate n* (λ-. *x*)
cupdate n (*f* ∘ (λ-. *x*)) = *cupdate n* (λ-. *f x*)
by (*auto simp add*: *comp-def*)

lemma *K-eq-cong*: ((λ-. *x*) = (λ-. *y*)) ⟷ *x* = *y*
by (*simp add*: *fun-eq-iff*)

ML-file ‹*positional-symbol-table.ML*›

named-theorems *locals*

consts *clocals-string-embedding* :: *string* ⇒ *nat*
consts *exit-'*::*nat*

definition *global-exn-var-clocal* = *clocals-string-embedding* ''*global-exn-var*''

bundle *clocals-string-embedding*
begin
notation *clocals-string-embedding* (‹*S*›)
end

ML ‹
structure CLocals =
struct
structure Locals = *Positional-Symbol-Table*(*type key* = *string val ord* = *fast-string-ord*);

@{*record* ‹*datatype entry* = *Entry of* {*typ* : *typ*, *term*: *term*, *def*: *thm*, *kind*: *NameGeneration.var-kind*}›}

fun morph-entry phi {*typ*, *term*, *def*, *kind*} = *make-entry*
{*typ* = *Morphism.typ phi typ*, *term* = *Morphism.term phi term*, *def* = *Morphism.thm phi def*, *kind* = *kind*};

fun entry-ord (*Entry* {*kind* = *kind1*, *typ* = *T1*, *term* = *t1*, *def* = *thm1*}, *Entry*

```
{kind = kind2, typ = T2, term = t2, def = thm2})
  = prod-ord NameGeneration.var-kind-ord (prod-ord Term-Ord.typ-ord (prod-ord
Term-Ord.fast-term-ord Thm.thm-ord))
      ((kind1, (T1, (t1, thm1))), (kind2, ((T2, (t2, thm2)))))

val entry-eq = is-equal o entry-ord

type locals-scope = {locals: entry Locals.symbol-table, scope: Locals.qualifier}

structure Data = Generic-Data (struct
  type T = locals-scope
  val empty = {locals = Locals.empty, scope = []}
  fun merge ({locals=locals1, ...}, {locals=locals2, ...}) =
      {locals = Locals.merge entry-eq (locals1, locals2),
       scope = []}
  end)

fun map-locals f ({locals, scope}:locals-scope) =
  ({locals = f locals, scope = scope}:locals-scope)

fun map-scope f ({locals, scope}:locals-scope) = ({locals = locals, scope = f scope}:locals-scope)


fun locals-map f  = Data.map (map-locals f)
fun scope-map f  = Data.map (map-scope f)
fun enter-scope n = map-scope (fn xs => xs @ [n])
val exit-scope = map-scope (fst o split-last)

fun switch-scope fname =
 Context.proof-map (Data.map (exit-scope #> enter-scope fname))


fun program-name ctxt =
  case #scope (Data.get (Context.Proof ctxt)) of
    (prog-name::-) => prog-name
  | - =>

fun function-name ctxt =
  case #scope (Data.get (Context.Proof ctxt)) of
    [-, fun-name] => fun-name
  | - =>

fun read-function-pointer ctxt s =
  let
    val s = if Long-Name.is-qualified s then s else Long-Name.qualify (program-name
ctxt) s
  in
    Proof-Context.read-const {proper=true, strict=true} ctxt s
  end
```

```
fun mk-lookup-positional T i =
  Const‹clookup T› $ HOLogic.mk-number @{typ nat} i

fun lookup-by-short-name ctxt x =
  let
    val {scope, locals} = Data.get (Context.Proof ctxt)
    val x = NameGeneration.un-varname x
    val (qualifier, base) = Long-Name.explode x |> split-last
    val common-scope = rev scope |> drop (length qualifier) |> rev
   val full-scope = (if null common-scope then scope else common-scope) @ qualifier
    fun proj (name,  (pos, Entry {typ, term,...})) = (name, (typ, term))
    val hits =
        Locals.lookup locals full-scope base |> Option.map (proj o pair x)
  in
    case hits of
      SOME v => v
    | NONE => error (lookup-by-short-name:  ^ quote base ^  not defined in scope
^ @{make-string} full-scope)
  end

fun info-from-term ctxt t =
  let
    val {scope, locals} = Data.get (Context.Proof ctxt)
  in
    case t of
      Const (n, -) =>
        let
          val name = NameGeneration.un-varname (Long-Name.base-name n)
        in Locals.lookup locals scope name |> Option.map (fn (p, x) => (name, (p,
x))) end
    | - => try Utils.dest-nat-or-number t
          |> Option.mapPartial (fn p =>
              Locals.lookup-positional locals scope p
              |> Option.map (fn (name, x) => (name, (p, x))))
  end

fun kind-from-term ctxt t = info-from-term ctxt t
    |> Option.map (fn (-, (-, Entry {kind, ...})) => kind)

fun is-defined ctxt x =
  is-some (try (lookup-by-short-name ctxt) x)

fun gen-mk-lookup ctxt (NameGeneration.Named x) =
     let val (-, (typ, term)) = lookup-by-short-name ctxt x
     in (Const‹clookup typ› $ term, typ) end
  | gen-mk-lookup - (NameGeneration.Positional (i, T)) =
      (Const‹clookup T› $ HOLogic.mk-number @{typ nat} i, T)
```

```
fun get-position ctxt n =
 let
   val {scope, locals} = Data.get (Context.Proof ctxt)
 in Locals.lookup locals scope n |> Option.map fst end

fun get-default-position ctxt n = the-default (~1) (get-position ctxt n)


fun positional-ord ctxt = int-ord o apply2 (get-default-position ctxt)
fun get-locals ctxt  =
  let
    val {scope, locals} = Data.get (Context.Proof ctxt)
    val locals = Locals.get-scope (locals) scope
  in
    locals
    |> Locals.Keytab.dest
    |> map (fn (n, (-, Entry {typ, term, ...})) =>
        (Long-Name.base-name n, (typ, (Const ‹clookup typ› $ term))))
  end

fun mk-lookup ctxt x = fst (gen-mk-lookup ctxt (NameGeneration.Named x))

fun mk-update ctxt x =
  let val (-, (typ, term)) = lookup-by-short-name ctxt x
  in Const ‹cupdate typ› $ term end

fun mk-update-positional T i =
  Const ‹cupdate T› $ HOLogic.mk-number @{typ nat} i

fun get-name ctxt n = try ‹
  let val (n, (typ, -)) = lookup-by-short-name ctxt n
  in (n, typ) end
  catch - => raise Match›

(* Name hints for bound variable names. Special treatment to unprime return variable name *)
fun return-name-hint ret = ret'
  | return-name-hint x = x

fun dest-return-name-hint ret' = ret
  | dest-return-name-hint x = x

fun embedded-name-hint n = @{const clocals-string-embedding} $ Utils.encode-isa-string n

fun name-hint ctxt n =
 if n = NameGeneration.global-exn-var-name then
   @{const global-exn-var-clocal}
 else
```

```
  (case try (lookup-by-short-name ctxt) (return-name-hint n) of
    SOME (-, (-, term)) => term
  | NONE =>
      let
          val - = warning (name-tag: no match for  ^ quote n ^ . Using string
embedding.)
      in
        embedded-name-hint n
      end)

fun name-hints ctxt = map (name-hint ctxt) #> Utils.encode-isa-list @{typ nat}

fun dest-name-hint (Const- ‹clocals-string-embedding for s›) = HOLogic.dest-string
s
  | dest-name-hint (const ‹global-exn-var-clocal›) = NameGeneration.global-exn-var-name
  | dest-name-hint (Const (n, @{typ nat})) = dest-return-name-hint (NameGeneration.un-varname
(Long-Name.base-name n))
  | dest-name-hint - = raise Match

val dest-name-hints = Utils.decode-isa-list #> map dest-name-hint

fun split-scope n =
  case rev (Long-Name.explode n) of
    (base:: fname :: prog-name:: -) => ([prog-name, fname], base)
  | - => raise Match

fun strip-common [] [] = []
  | strip-common (x::xs) (y::ys) = if x = y then strip-common xs ys else (y::ys)
  | strip-common [] ys = ys
  | strip-common - [] = []


val short-names = Attrib.setup-config-bool binding ‹clocals-short-names› (K false);

fun print-tr ctxt (c as (Const (n, T))) =
      let
        val {scope, locals} = Data.get (Context.Proof ctxt)
        val (name-scope, base-name) = split-scope n
        val base-name = NameGeneration.un-varname base-name
        val min-scope = strip-common scope name-scope
        val short-name = base-name
          |> not (Config.get ctxt short-names) ? fold Long-Name.qualify min-scope
       in
        case Locals.lookup locals name-scope base-name of
          SOME (-, Entry {typ, ...}) =>
              Syntax.const syntax-const ‹-constrain› $ Const (short-name, T) $
Syntax-Phases.term-of-typ ctxt typ
        | - => raise Match
      end
```

```
  | print-tr - t = raise Match

fun is-lookup ctxt (Const (@{const-name clookup}, - ) $ -) = true
  | is-lookup ctxt (Const (@{const-syntax clookup}, - ) $ -) = true
  | is-lookup ctxt - = raise Match

fun lookup-tr' ctxt (Const (@{const-name clookup}, - ) $ (c as Const -)) = print-tr
ctxt c
  | lookup-tr' ctxt (Const (@{const-syntax clookup}, - ) $ (c as Const -)) = print-tr
ctxt c
  | lookup-tr' ctxt t = raise Match

fun dest-update-tr' ctxt ((c as (Const (@{const-syntax cupdate}, - ) $  Const - ))
$ v $ s) =
      (c, v, SOME s)
  | dest-update-tr' ctxt ((c as (Const (@{const-name cupdate}, - ) $  Const - )) $
v $ s) =
      (c, v, SOME s)
  | dest-update-tr' ctxt ((c as (Const (@{const-syntax cupdate}, - ) $  Const - )) $
v) =
      (c, v, NONE)
  | dest-update-tr' ctxt ((c as (Const (@{const-name cupdate}, - ) $  Const - )) $
v) =
      (c, v, NONE)
  | dest-update-tr' ctxt t = raise Match

fun update-tr' ctxt (Const (@{const-name cupdate}, - ) $ (c as Const -)) = print-tr
ctxt c
  | update-tr' ctxt (Const (@{const-syntax cupdate}, - ) $ (c as Const -)) = print-tr
ctxt c
  | update-tr' ctxt t = t

fun locals-insert qualifier pos name typ kind def tab =
  let
    val tab' = tab |> Locals.update-new qualifier (name,
    Entry {typ=typ, term = Thm.concl-of def |> Utils.lhs-of-eq, def = Thm.trim-context
def, kind = kind})
     val - = @{assert} ((Locals.lookup tab' qualifier name |> Option.map fst) =
SOME pos)
  in
    tab'
  end

fun add-entry qualifier pos name typ kind def =
 locals-map (locals-insert qualifier pos name typ kind def)

fun add-entry-attr qualifier pos name typ kind =
  Thm.declaration-attribute (add-entry qualifier pos name typ kind)
```

```
fun define-locals qualifier decls thy =
  let
    val - = @{assert} (not (null qualifier))
    val fname = split-last qualifier |> snd
    val lthy = Named-Target.theory-init thy
    fun define (i, (name, typ, kind)) lthy =
      let
        val vname = NameGeneration.ensure-varname name
          val b = Binding.make (vname, here) |> fold-rev (Binding.qualify true)
qualifier
         val attrib = Attrib.internal here (fn - => add-entry-attr qualifier i name
typ kind)
        val rhs = HOLogic.mk-number @{typ nat} i
        val ((t, (s, thm)), lthy) = lthy
          |> Local-Theory.define ((b, Mixfix.NoSyn), ((Binding.suffix-name -def b,
[attrib] @ @{attributes [locals]} ), rhs))
      in
        lthy |> Code.declare-default-eqns [(thm, true)]
 (*|>
          Local-Theory.declaration {pervasive=true, syntax=false} (fn - =>
          Context.mapping
            (Code.declare-default-eqns-global [(thm, true)])
            I)*)
      end
    val lthy = lthy
      |> fold define (tag-list 0 decls)
      |> Bundle.bundle {open-bundle = false} (Binding.make (suffix -scope fname,
here),
          Attrib.internal-declaration here (Morphism.entity (fn - => scope-map (K
(qualifier))))) []
  in
    lthy |> Local-Theory.exit-global
  end

fun symmetric ctxt thm = fst (Thm.apply-attribute Calculation.symmetric thm
(Context.Proof ctxt))

fun gen-unfolded {sym} qualifier ctxt thm =
 let
   val {scope, locals} = Data.get (Context.Proof ctxt)
   val qualifier = if null qualifier then scope else qualifier
   val defs0 = Locals.get-scope locals qualifier |> Locals.Keytab.dest
       |> map (fn (-, (-, Entry {def,...})) => Thm.transfer' ctxt def)
   val defs = defs0 |> sym ? map (symmetric ctxt)
 in
  Simplifier.simplify (Simplifier.put-simpset HOL-basic-ss ctxt addsimps defs) thm
 end
```

```
val unfolded-with = gen-unfolded {sym = false}
val folded-with  = gen-unfolded {sym = true}

val unfolded = unfolded-with []
val folded  = folded-with []

fun check-scope ctxt (qualifier, pos) =
  let
    val {locals, scope} = Data.get (Context.Proof ctxt)
    val exploded = Long-Name.explode qualifier
    val extended = if length exploded = 1 then
                     take 1 scope @ exploded
                   else
                     exploded
  in
    if null extended orelse Locals.defined-scope locals extended then
      extended
    else
     error (undefined scope: ^quote (Long-Name.implode extended) ^Position.here
pos)
  end
end

›

setup ‹
Attrib.setup binding ‹fold-locals›
  (Args.context -- Scan.lift (Scan.optional Parse.name-position (, Position.none))
>>
  (fn (ctxt, (qualifier, pos)) =>
     let
       val exploded = CLocals.check-scope ctxt (qualifier, pos)
     in
       Thm.rule-attribute [] (CLocals.folded-with exploded o Context.proof-of)
     end))
  fold local variables within optional scope qualifier #>

Attrib.setup binding ‹unfold-locals›
  (Args.context -- Scan.lift (Scan.optional Parse.name-position (, Position.none))
>>
  (fn (ctxt, (qualifier, pos)) =>
     let
       val exploded = CLocals.check-scope ctxt (qualifier, pos)
     in
       Thm.rule-attribute [] (CLocals.unfolded-with exploded o Context.proof-of)
     end))
  unfold local variables within optional scope qualifier
›
```

nonterminal *localsupdbinds* **and** *localsupdbind*

syntax
-localsupdbind :: *'a* ⇒ *'a* ⇒ *localsupdbind*
(‹(‹*indent=2 notation=‹infix localsupdbind›*›- :=$_{\mathcal{L}}$/ -)›)
:: *localsupdbind* ⇒ *localsupdbinds* (‹-›)
-localsupdbinds:: *localsupdbind* ⇒ *localsupdbinds* ⇒ *localsupdbinds*
(‹(‹*open-block notation=‹mixfix localsupdbinds*›-,/ -)›)

syntax
-statespace-lookup :: *locals* ⇒ *'name* ⇒ *'c*
(‹(‹*open-block notation=‹infix statespace-lookup*›- · -)› [*60*, *60*] *60*)
-statespace-locals-lookup :: (*'g*, *locals*, *'e*, *'x*) *state-scheme* ⇒ *'name* ⇒ *'c*
(‹(‹*open-block notation=‹infix statespace-locals-lookup*›- ·$_{\mathcal{L}}$ -)› [*60*, *60*] *60*)

-statespace-update :: *locals* ⇒ *'name* ⇒ (*'c* ⇒ *'c*) ⇒ *locals*
-statespace-updates :: *locals* ⇒ *updbinds* ⇒ *locals*
(‹(‹*open-block notation=‹mixfix statespace-updates*›-⟨-⟩)› [*900*, *0*] *900*)

-statespace-locals-update :: (*'g*, *locals*, *'e*, *'x*) *state-scheme* ⇒ *'name* ⇒ (*'c* ⇒ *'c*) ⇒ (*'g*, *locals*, *'e*, *'x*) *state-scheme*
-statespace-locals-updates :: *locals* ⇒ *localsupdbinds* ⇒ *locals*
(‹(‹*open-block notation=‹mixfix statespace-locals-updates*›-⟨-⟩)› [*900*, *0*] *900*)

-statespace-locals-map ::
'name ⇒ (*'c* ⇒ *'c*) ⇒ (*'g*, *locals*, *'e*, *'x*) *state-scheme* ⇒ (*'g*, *locals*, *'e*, *'x*) *state-scheme*
(‹(‹*indent=2 notation=‹infix statespace-locals-map*›-:=$_{\mathcal{L}}$/ -)› [*1000*, *1000*] *1000*)

syntax-consts
-statespace-updates ⇌ *fun-upd* **and**
-statespace-locals-updates ⇌ *locals-update*

translations
-statespace-updates f (*-updbinds b bs*) ==
-statespace-updates (*-statespace-updates f b*) *bs*
s⟨*x* := *y*⟩ == *-statespace-update s x y*

-statespace-locals-updates f (*-localsupdbinds b bs*) ==
-statespace-locals-updates (*-statespace-locals-updates f b*) *bs*
s⟨*x* :=$_{\mathcal{L}}$ *y*⟩ == *-statespace-locals-update s x y*

parse-translation
‹

```
let
fun lookup-tr ctxt [s, x] =
  let
  in
    (case Term-Position.strip-positions x of
      Free (n,-) => CLocals.mk-lookup ctxt n $ s
    | - => raise Match)
  end

fun locals-lookup-tr ctxt [s, x] =
  (case Term-Position.strip-positions x of
    Free (n,-) =>
      CLocals.mk-lookup ctxt n $ (Syntax.const const-name‹locals› $ s)
  | - => raise Match);

fun update-tr ctxt [s, x, v] =
  (case Term-Position.strip-positions x of
     Free (n, -) =>
       let
         val upd = CLocals.mk-update ctxt n
       in upd $ v $ s end
   | - => raise Match);

fun locals-update-tr ctxt [s, x, v] =
  let
    val upd $ n $ v' $ - = update-tr ctxt [s, x, v]

  in
    Syntax.const const-name‹locals-update› $ (upd $ n $ v') $ s
  end


fun locals-map-tr ctxt [x, v] =
  let
     val upd $ s = locals-update-tr ctxt [Bound 0, x, v]
  in
    upd
  end

in

 [(syntax-const‹-statespace-lookup›, lookup-tr),
  (syntax-const‹-statespace-locals-lookup›, locals-lookup-tr),
  (syntax-const‹-statespace-update›, update-tr),
  (syntax-const‹-statespace-locals-update›, locals-update-tr),
  (syntax-const‹-statespace-locals-map›, locals-map-tr)]
end
›
```

```
print-translation ‹
let

  fun dest-number (Const (const-syntax ‹Groups.zero›, -)) = 0
    | dest-number (Const (const-syntax ‹Groups.one›, -)) = 1
   | dest-number (Const (const-syntax ‹numeral›, -) $ n) = Numeral.dest-num-syntax
n

  fun lookup-tr' ctxt (args as [n, s]) =
    let
      val n' = CLocals.print-tr ctxt n
    in
      case s of
        Const (const-syntax ‹locals›, -) $ s'' =>
          Syntax.const syntax-const ‹-statespace-locals-lookup› $ s'' $ n'
      | Const (syntax-const ‹-antiquoteCur›, -) $ Const (get-actuals, -) =>
          Syntax.const syntax-const ‹-antiquoteCur› $ n'
      | - =>  Syntax.const syntax-const ‹-statespace-lookup› $ s $ n'
   end

  fun update-tr' ctxt (args as [n, v, s]) =
    let
      val n' = CLocals.print-tr ctxt n
    in
      Syntax.const syntax-const ‹-statespace-update› $ s $ n' $ v
    end

  fun locals-update-tr' ctxt (args as [upd, s]) =
    let
      val - $ s' $ n' $ v = update-tr' ctxt ((snd (strip-comb upd)) @ [s])
    in
      Syntax.const syntax-const ‹-statespace-locals-update› $ s' $ n' $ v
    end
  | locals-update-tr' ctxt (args as [upd]) =
    let
      val - $ - $ n' $ v = update-tr' ctxt ((snd (strip-comb upd)) @ [Bound 0])
    in
      Syntax.const syntax-const ‹-statespace-locals-map› $ n' $ v
    end

  fun assign-tr' ctxt (args as [(aq as Const (-antiquoteCur, -)) $ (c as Const (locals,
-)), upd]) =
    let
      fun dest-K (Abs (-, -, v)) = v
        | dest-K t = t

      val s = Bound 0
      val - $ - $ n $ v = locals-update-tr' ctxt [upd, s]
    in
```

```
    Syntax.const syntax-const ‹-Assign› $ (aq $ n) $ (dest-K v)
  end
  | assign-tr' ctxt t = raise Match


in
  [(const-syntax ‹clookup›, lookup-tr'),
   (const-syntax ‹cupdate›, update-tr'),
   (const-syntax ‹locals-update›, locals-update-tr'),
   (syntax-const ‹-Assign›, assign-tr')]

end
›


end
```

Chapter 14

Setup Lex / Yacc and Translation from C to Simpl

theory *CTranslationSetup*
imports
UMM
PackedTypes
PrettyProgs
StaticFun
IndirectCalls
ModifiesProofs
HOL−Eisbach.Eisbach
ML-Record-Antiquotation
Option-Scanner
Misc-Antiquotation
MkTermAntiquote
TermPatternAntiquote
CLocals
keywords
cond-sorry-modifies-proofs :: *thy-decl*
and
mllex
mlyacc:: *thy-load*
begin

ML ‹
structure Coerce-Syntax =
struct

val show-types = *Ptr-Syntax.show-ptr-types*

fun coerce-tr′ cnst ctxt typ ts = *if Config.get ctxt show-types then*
case Term.strip-type typ of
([*S*], *T*) =>

```
        list-comb
          (Syntax.const cnst $ Syntax-Phases.term-of-typ ctxt S
                           $ Syntax-Phases.term-of-typ ctxt T
          , ts)
      | - => raise Match
  else raise Match
end
›

definition coerce::'a::mem-type ⇒ 'b::mem-type where
  coerce v = from-bytes (to-bytes-p v)

syntax
  -coerce :: type ⇒ type ⇒ logic
   (‹(‹indent=1 notation=‹mixfix COERCE›› COERCE/(‹indent=1 notation=‹infix
coerce›› '(- → -')))›)
syntax-consts
  -coerce == coerce
translations
  COERCE('a → 'b) => CONST coerce :: ('a ⇒ 'b)
typed-print-translation
  ‹ [(@{const-syntax coerce}, Coerce-Syntax.coerce-tr' @{syntax-const -coerce})] ›

lemma coerce-id[simp]:
  shows coerce v = v
  by (simp add: coerce-def)

lemma coerce-cancel-packed[simp]:
  fixes v::'a::packed-type
  assumes sz-eq: size-of (TYPE('a)) = size-of (TYPE('b))
  shows coerce ((coerce v)::'b::packed-type) = v
  using assms
  apply (simp add: coerce-def)
  by (metis (mono-tags, lifting) access-ti₀-def access-ti₀-to-bytes
      field-access-update-same(1) inv-p length-replicate length-to-bytes-p
      packed-type-intro-simps(1) wf-fd)

definition coerce-map:: ('a::mem-type ⇒ 'a) ⇒ ('b::mem-type ⇒ 'b) where
  coerce-map f v = coerce (f (coerce v))

lemma coerce-map-id[simp]: coerce-map f (coerce v) = f v
  by (simp add: coerce-map-def)

lemma coerce-coerce-map-cancel-packed[simp]:
  fixes f::'a::packed-type ⇒ 'a
  fixes v::'b::packed-type
  assumes sz-eq[simp]: size-of (TYPE('a)) = size-of (TYPE('b))
  shows ((coerce (coerce-map f v))::'a) = f (coerce v)
  by (simp add: coerce-map-def)
```

named-theorems *global-const-defs* **and**
global-const-array-selectors **and**
global-const-non-array-selectors **and**
global-const-selectors

named-theorems *fun-ptr-simps*
named-theorems *fun-ptr-intros*
named-theorems *fun-ptr-distinct*
named-theorems *fun-ptr-subtree*
named-theorems *disjoint-$\mathcal{G}$-$\mathcal{S}$*

We integrate mllex and mlyacc directly into Isabelle:

- We compile the SML files according to the description in `tools/mlyacc/src/FILES`
- We export the necessary signatures and structures to the Isabelle/ML environment.
- As mllex / mlyacc operate directly on files we invoke them on temporary files and redirect stdout / stderr to display the messages within PIDE. We wrap this in Isabelle commands **mllex**, **mlyacc**.

SML-import ‹
infix 1 |> val op |> = Basics.|>
structure File-Stream = File-Stream
val make-path = Path.explode o ML-System.standard-path
›
SML-file ‹*tools/mllex/mllex.ML*›
SML-export ‹*structure LexGen = LexGen*›

SML-file ‹*tools/mlyacc/mlyacclib/MLY-base−sig.ML*›
SML-file ‹*tools/mlyacc/mlyacclib/MLY-stream.ML*›
SML-file ‹*tools/mlyacc/mlyacclib/MLY-lrtable.ML*›
SML-file ‹*tools/mlyacc/mlyacclib/MLY-join.ML*›
SML-file ‹*tools/mlyacc/mlyacclib/MLY-parser2.ML*›
SML-file ‹*tools/mlyacc/src/utils.ML*›
SML-file ‹*tools/mlyacc/src/sigs.ML*›
SML-file ‹*tools/mlyacc/src/hdr.ML*›
SML-file ‹*tools/mlyacc/src/yacc−grm−sig.sml*›
SML-file ‹*tools/mlyacc/src/yacc−grm.sml*›
SML-file ‹*tools/mlyacc/src/yacc.lex.sml*›
SML-file ‹*tools/mlyacc/src/parse.ML*›
SML-file ‹*tools/mlyacc/src/grammar.ML*›
SML-file ‹*tools/mlyacc/src/core.ML*›

SML-file ‹*tools/mlyacc/src/coreutils.ML*›
SML-file ‹*tools/mlyacc/src/graph.ML*›
SML-file ‹*tools/mlyacc/src/look.ML*›
SML-file ‹*tools/mlyacc/src/lalr.ML*›
SML-file ‹*tools/mlyacc/src/mklrtable.ML*›
SML-file ‹*tools/mlyacc/src/mkprstruct.ML*›
SML-file ‹*tools/mlyacc/src/shrink.ML*›
SML-file ‹*tools/mlyacc/src/verbose.ML*›
SML-file ‹*tools/mlyacc/src/absyn−sig.ML*›
SML-file ‹*tools/mlyacc/src/absyn.ML*›
SML-file ‹*tools/mlyacc/src/yacc.ML*›
SML-file ‹*tools/mlyacc/src/link.ML*›

SML-export ‹*signature LR-TABLE = LR-TABLE*›
SML-export ‹*structure LrTable = LrTable*›
SML-export ‹*signature LR-PARSER = LR-PARSER*›
SML-export ‹*structure LrParser = LrParser*›
SML-export ‹*signature TOKEN = TOKEN*›
SML-export ‹*signature PARSER-DATA = PARSER-DATA*›
SML-export ‹*signature PARSE-GEN = PARSE-GEN*›
SML-export ‹*signature LEXER = LEXER*›
SML-export ‹*signature PARSER-DATA = PARSER-DATA*›
SML-export ‹*signature PARSER = PARSER*›
SML-export ‹*signature ARG-PARSER = ARG-PARSER*›
SML-export ‹*structure ParseGen = ParseGen*›

SML-export ‹*functor Join*(*structure Lex* : *LEXER*
structure ParserData: *PARSER-DATA*
structure LrParser : *LR-PARSER*
sharing ParserData.LrTable = LrParser.LrTable
sharing ParserData.Token = LrParser.Token
sharing type Lex.UserDeclarations.svalue = ParserData.svalue
sharing type Lex.UserDeclarations.pos = ParserData.pos
sharing type Lex.UserDeclarations.token = ParserData.Token.token):
PARSER =
Join(*structure Lex = Lex*
structure ParserData = ParserData
structure LrParser = LrParser)›
SML-export ‹*functor JoinWithArg*(*structure Lex* : *ARG-LEXER*
structure ParserData: *PARSER-DATA*
structure LrParser : *LR-PARSER*
sharing ParserData.LrTable = LrParser.LrTable
sharing ParserData.Token = LrParser.Token
sharing type Lex.UserDeclarations.svalue = ParserData.svalue
sharing type Lex.UserDeclarations.pos = ParserData.pos
sharing type Lex.UserDeclarations.token = ParserData.Token.token)
: *ARG-PARSER* =
JoinWithArg(*structure Lex = Lex*
structure ParserData = ParserData

```
      structure LrParser = LrParser)
›

ML ‹
local

val tmp-io = Synchronized.var tmp-io ()

fun system-command cmd =
  if Isabelle-System.bash cmd <> 0 then error (System command failed: ^ cmd)
else ();


fun copy-file qualify-ref src0 dst0 =
  let
    val src = Path.expand src0;
    val dst = Path.expand dst0;
    val target = if File.is-dir dst then Path.append dst (Path.base src) else dst;
  in
    if File.eq (src, target) then
      ()
    else if qualify-ref then
     ignore (system-command (sed 's/ ref / Unsynchronized.ref /g' ^File.bash-path
src ^ > ^File.bash-path target))
    else
       ignore (system-command (cp -f ^ File.bash-path src ^ ^ File.bash-path
target))
  end;

fun file-command qualify-ref tag exts f files thy =
  let
    val (files, thy) = files thy
    val {src-path, lines,...}: Token.file = the-single files
    val filename = Path.file-name src-path
    val full-src-path = Path.append (Resources.master-directory thy) src-path
     val - = Isabelle-System.with-tmp-dir tag (fn tmp-dir => Synchronized.change
tmp-io (fn - =>
      let
        val orig-stdOut = TextIO.getOutstream TextIO.stdOut
        val orig-stdErr = TextIO.getOutstream TextIO.stdErr
        val out-file = (Utils.sanitized-path thy tmp-dir (Path.ext out src-path))
        val err-file = (Utils.sanitized-path thy tmp-dir (Path.ext err src-path))
        val dir = Path.dir out-file
        val - = Isabelle-System.make-directory dir
        val stdOut = TextIO.openOut (File.platform-path out-file)
        val stdErr = TextIO.openOut (File.platform-path err-file)

        val - = TextIO.setOutstream (TextIO.stdOut, TextIO.getOutstream stdOut)
        val - = TextIO.setOutstream (TextIO.stdErr, TextIO.getOutstream stdErr)
```

```
        val tmp-file = Utils.sanitized-path thy tmp-dir src-path

        val - = File.write tmp-file (cat-lines lines)
        val res = Exn.result f (File.platform-path tmp-file)


        val - = TextIO.setOutstream (TextIO.stdOut, orig-stdOut)
        val - = TextIO.setOutstream (TextIO.stdErr, orig-stdErr)
        val - = TextIO.closeOut stdOut
        val - = TextIO.closeOut stdErr
        val lines = filter (fn s => s <> ) (File.read-lines out-file @ File.read-lines
err-file)
        val msg = if null lines then  else :\n ^ (prefix-lines    (cat-lines lines))
      in
        case res of
          Exn.Res () =>
           let
             val result-files = map (fn ext => (Path.ext ext tmp-file,
                    Path.ext ext full-src-path)) exts
             val - = app (fn (src, dst) => copy-file qualify-ref src dst) result-files
                val - = tracing (Markup.markup Markup.keyword1 tag ^   ^ quote
filename ^  succeeded ^ msg)
           in () end
        | Exn.Exn exn =>
          error (Markup.markup Markup.keyword1 tag ^   ^ quote filename ^ failed
^ msg ^
             \n ^ Runtime.exn-message exn)
      end))

  in
    thy
  end
in
val - =
  Outer-Syntax.command
    @{command-keyword mllex} generate lexer
   (Resources.provide-parse-files single >> (Toplevel.theory o (file-command true
mllex [sml] LexGen.lexGen)))
val - =
  Outer-Syntax.command
    @{command-keyword mlyacc} generate parser
   (Resources.provide-parse-files single >> (Toplevel.theory o (file-command true
mlyacc [sig, sml] ParseGen.parseGen)))

end
›
```

primrec *map-of-default* ::
  ('*p* ⇒ '*a*) ⇒ ('*p* ∗ '*a*) *list* ⇒ '*p* ⇒ '*a*
where
  map-of-default d [] *x* = *d x*
| *map-of-default d* (*x* # *xs*) *x*′ = (*if fst x* = *x*′ *then snd x else map-of-default d xs*
x′)

lemma *map-of-default-append*: ‹*map-of-default d* (*xs* @ *ys*) = *map-of-default* (*map-of-default*
d ys) *xs*›
  by (*induction xs arbitrary*: *d ys*) *auto*

lemma *map-of-default-map-of-conv*:
  ‹*map-of-default d xs p* = (*case map-of xs p of Some f* ⇒ *f* | *None* ⇒ *d p*)›
  by (*induction xs*) (*auto simp add*: *fun-upd-same fun-upd-other*)

lemma *map-of-default-fallthrough*:
  p ∉ *set* (*map fst xs*) ⟹ *map-of-default d xs p* = *d p*
  by (*induct xs*) *auto*

lemma *map-of-default-distinct*:
  assumes *distinct* (*map fst xs*)
  shows *list-all* (λ(*p*, *f*). *map-of-default d xs p* = *f*) *xs*
  using *assms*
  apply (*induction xs*)
   apply *simp-all*
  by (*smt* (*verit*, *ccfv-SIG*) *image-iff list.pred-mono-strong prod.case-eq-if*)

lemma *map-of-default-default-conv*:
  assumes *list-all* (λ(*p*, *f*). *d p* = *f*) *xs*
  shows *map-of-default d xs* = *d*
  using *assms*
  by (*induct xs*) *auto*

lemma *map-of-default-monotone-cons*[*partial-function-mono*]:
  assumes *f1* [*partial-function-mono*]: *monotone R X f1*
  assumes [*partial-function-mono*]: *monotone R X* (λ*f*. *map-of-default d* (*xs f*) *p*)
  shows *monotone R X* (λ*f*. *map-of-default d* ((*p1*, *f1 f*)#*xs f*) *p*)
  apply (*simp only*: *map-of-default.simps fst-conv snd-conv cong*: *if-cong*)
  apply (*intro partial-function-mono*)
  done

hide-const (**open**) *StaticFun.Node*

primrec *tree-of* :: '*a list* ⇒ '*a tree*
where
  tree-of [] = *Tip*
| *tree-of* (*x*#*xs*) = *Node* (*Tip*) *x False* (*tree-of xs*)

lemma *set-of-tree-of*: *set-of* (*tree-of xs*) = *set xs*
by (*induct xs*) *auto*

lemma *all-distinct-tree-of*:
assumes *all-distinct* (*tree-of xs*)
shows *distinct xs*
using *assms* **by** (*induct xs*) (*auto simp add*: *set-of-tree-of*)

lemma *all-distinct-tree-of'*:
all-distinct t ⟹ *tree-of xs* ≡ *t* ⟹ *distinct xs*
by (*simp add*: *all-distinct-tree-of*)

lemma *map-of-default-fallthrough'*:
map fst xs ≡ *ps* ⟹ *tree-of ps* ≡ *t* ⟹ *p* ∉ *set-of t* ⟹ *map-of-default d xs p* = *d p*
apply (*rule map-of-default-fallthrough*)
using *set-of-tree-of* [*of map fst xs*]
apply *simp*
done

primrec *list-of* :: *'a tree* ⇒ *'a list*
where
list-of Tip = []
| *list-of* (*Node l x d r*) = *list-of l* @ (*if d then* [] *else* [*x*]) @ *list-of r*

lemma *list-of-tree-of-conv* [*simp*]: *list-of* (*tree-of xs*) = *xs*
by (*induct xs*) *auto*

lemma *set-list-of-set-of-conv*: *set* (*list-of t*) = *set-of t*
by (*induct t*) *auto*

lemma *all-distinct-list-of*:
assumes *all-distinct t*
shows *distinct* (*list-of t*)
using *assms* **by** (*induct t*) (*auto simp add*: *set-list-of-set-of-conv*)

lemma *map-of-default-distinct-lookup-list-all*:
distinct (*map fst xs*) ⟹ *list-all* (λ(*p*, *f*). *map-of-default d xs p* = *f*) *xs*
by (*induct xs*) (*auto simp add*: *case-prod-beta' image-iff list.pred-mono-strong*)

lemma *map-of-default-distinct-lookup-list-all'*:
assumes *ps*: *map fst xs* ≡ *ps*
assumes *t*: *tree-of ps* ≡ *t*
assumes *dist*: *all-distinct t*
shows *list-all* (λ(*p*, *f*). *map-of-default d xs p* = *f*) *xs*
using *all-distinct-tree-of'* [*OF dist t*] *ps*
by (*simp add*: *map-of-default-distinct-lookup-list-all*)

lemma *map-of-default-distinct-lookup-list-all''*:

assumes *t*: *list-of t* ≡ *ps*
assumes *ps*: *map fst xs* ≡ *ps*
assumes *dist*: *all-distinct t*
shows *list-all* (λ(*p*, *f*). *map-of-default d xs p* = *f*) *xs*
by (*metis all-distinct-list-of dist map-of-default-distinct-lookup-list-all ps t*)

lemma *map-of-default-other-lookup-list-all*:
set ps ∩ *set* (*map fst xs*) = {} ⟹ *list-all* (λ*p*. *map-of-default d xs p* = *d p*) *ps*
using *map-of-default-fallthrough* [**where** *d*=*d* **and** *xs*= *xs*]
apply (*induct xs*)
apply (*clarsimp simp add*: *list.pred-True*)
by (*smt* (*verit*, *best*) *IntI ball-empty list.pred-set*)

lemma *delete-Some-subset*: *DistinctTreeProver.delete x t* = *Some t′* ⟹ *set-of t* ⊆ {*x*} ∪ *set-of t′*
by (*induct t arbitrary*: *t′*) (*auto split*: *option.splits if-split-asm*)

lemma *delete-Some-set-of-union*:
assumes *del*: *DistinctTreeProver.delete x t* = *Some t′* **shows** *set-of t* = {*x*} ∪ *set-of t′*
proof −
from *delete-Some-set-of* [*OF del*] **have** *set-of t′* ⊆ *set-of t* .
moreover
from *delete-Some-x-set-of* [*OF del*] **obtain** *x* ∈ *set-of t* *x* ∉ *set-of t′*
by *simp*
ultimately have {*x*} ∪ *set-of t′* ⊆ *set-of t*
by *blast*
with *delete-Some-subset* [*OF del*]
show *?thesis* **by** *blast*
qed

primrec *undeleted* :: *′a tree* ⇒ *bool*
where
undeleted Tip = *True*
| *undeleted* (*Node l y d r*) = (¬ *d* ∧ *undeleted l* ∧ *undeleted r*)

lemma *undeleted-tree-of*[*simp*]: *undeleted* (*tree-of xs*)
by (*induct xs*) *auto*

lemma *subtract-union-subset*:
subtract t$_1$ *t*$_2$ = *Some t* ⟹ *undeleted t*$_1$ ⟹ *set-of t*$_2$ ⊆ *set-of t*$_1$ ∪ *set-of t*
proof (*induct t*$_1$ *arbitrary*: *t*$_2$ *t*)
case *Tip* **thus** *?case* **by** *simp*
next
case (*Node l x b r*)
have *sub*: *subtract* (*Node l x b r*) *t*$_2$ = *Some t* **by** *fact*
from *Node.prems* **obtain** *not-b*: ¬ *b* **and** *undeleted-l*: *undeleted l* **and** *undeleted-r*:

```
  undeleted r
    by simp
  from subtract-Some-set-of [OF sub] have sub-t2: set-of (Node l x b r) ⊆ set-of
t2 .

  show ?case
  proof (cases DistinctTreeProver.delete x t2)
    case (Some t2')
    note del-x-Some = this
    from delete-Some-set-of-union [OF Some]
    have t2'-t2: set-of t2 = {x} ∪ set-of t2' .
    show ?thesis
    proof (cases subtract l t2')
      case (Some t2'')
      note sub-l-Some = this
      from Node.hyps (1) [OF Some undeleted-l]
      have t2''-t2': set-of t2' ⊆ set-of l ∪ set-of t2'' .
      show ?thesis
      proof (cases subtract r t2'')
        case (Some t2''')
        from Node.hyps (2) [OF Some undeleted-r]
        have set-of t2'' ⊆ set-of r ∪ set-of t2''' .
        with Some sub-l-Some del-x-Some sub t2''-t2' t2'-t2 not-b
        show ?thesis
          by auto
      next
        case None
        with del-x-Some sub-l-Some sub
        show ?thesis
          by simp
      qed
    next
      case None
      with del-x-Some sub
      show ?thesis
        by simp
    qed
  next
    case None
    with sub show ?thesis by simp
  qed
qed


lemma subtract-union-eq:
  assumes sub: subtract t1 t2 = Some t
  assumes und: undeleted t1
  shows set-of t2 = set-of t1 ∪ set-of t
proof
```

from *sub und*
show *set-of* $t_2 \subseteq$ *set-of* $t_1 \cup$ *set-of* t
by (*rule subtract-union-subset*)
next
from *subtract-Some-set-of-res* [*OF sub*] **have** *set-of* $t \subseteq$ *set-of* t_2**.**
moreover from *subtract-Some-set-of* [*OF sub*] **have** *set-of* $t_1 \subseteq$ *set-of* t_2 **.**
ultimately
show *set-of* $t_1 \cup$ *set-of* $t \subseteq$ *set-of* t_2 **by** *blast*
qed

lemma *subtract-empty*:
assumes *sub*: *subtract* t_1 t_2 = *Some* t
assumes *und*: *undeleted* t_1
assumes *empty*: *set-of* $t = \{\}$
shows *set-of* t_1 = *set-of* t_2
using *subtract-union-eq* [*OF sub und*] *empty* **by** *simp*

lemma *map-of-default-other-lookup-Ball*:
assumes *ps*: *list-of* $t \equiv$ *ps*
assumes *map-fst*: *map fst xs* $\equiv$ *ps*
assumes *sub*: *subtract t t-all* = *Some t-sub*
shows $\forall p \in$ *set-of t-sub. map-of-default d xs p* = *d p*
by (*metis disjoint-iff map-fst map-of-default-fallthrough ps set-list-of-set-of-conv sub subtract-Some-dist-res*)

lemma *subtract-set-of-exchange-first*:
assumes *sub1*: *subtract* t_1 t = *Some* t'
assumes *sub2*: *subtract* t_2 t = *Some* t''
assumes *und1*: *undeleted* t_1
assumes *und2*: *undeleted* t_2
assumes *seq*: *set-of* t_1 = *set-of* t_2
shows *set-of* t' = *set-of* t''
using *subtract-union-eq* [*OF sub1 und1*] *subtract-union-eq* [*OF sub2 und2*] *seq*
subtract-Some-dist-res [*OF sub1*] *subtract-Some-dist-res* [*OF sub2*]
by *blast*

lemma *TWO*: *Suc* (*Suc 0*) = *2* **by** *arith*

definition
fun-addr-of :: *int* ⇒ *unit ptr* **where**
fun-addr-of i ≡ *Ptr* (*word-of-int i*)

definition
ptr-range :: *'a::c-type ptr* ⇒ *addr set* **where**
ptr-range p ≡ {*ptr-val* (*p::'a ptr*) ..< *ptr-val p* + *word-of-int*(*int*(*size-of* (*TYPE*('*a*))))
}

lemma *guarded-spec-body-wp* [*vcg-hoare*]:
P ⊆ {*s*. (∀ *t*. (*s*,*t*) ∈ *R* ⟶ *t* ∈ *Q*) ∧ (*Ft* ∉ *F* ⟶ (∃ *t*. (*s*,*t*) ∈ *R*))}
⟹ Γ,Θ⊢$_{/F}$ *P* (*guarded-spec-body Ft R*) *Q*, *A*
apply (*simp add*: *guarded-spec-body-def*)
apply (*cases Ft* ∈ *F*)
apply (*erule HoarePartialDef.Guarantee*)
apply (*rule HoarePartialDef.conseqPre*, *rule HoarePartialDef.Spec*)
apply *auto*[*1*]
apply (*rule HoarePartialDef.conseqPre*, *rule HoarePartialDef.Guard*[**where** *P*=*P*])
apply (*rule HoarePartialDef.conseqPre*, *rule HoarePartialDef.Spec*)
apply *auto*[*1*]
apply *simp*
apply (*erule order-trans*)
apply (*auto simp*: *image-def Bex-def*)
done

named-theorems *recursive-records-fold-congs* **and** *recursive-records-split-all-eqs*

ML-file *../lib/ml−helpers/StringExtras.ML*
ML-file *topo-sort.ML*
ML-file *isa-termstypes.ML*

mllex *StrictC.lex*
mlyacc *StrictC.grm*

ML-file *StrictC.grm.sig*
ML-file *StrictC.grm.sml*
ML-file *StrictC.lex.sml*

ML-file *StrictCParser.ML*
ML-file *complit.ML*
ML-file *hp-termstypes.ML*
ML-file *termstypes−sig.ML*
term *word-of-int*
ML-file *termstypes.ML*
ML-file *recursive-records/recursive-record-pp.ML*
ML-file *recursive-records/recursive-record-package.ML*
ML-file *UMM-termstypes.ML*
ML-file *expression-typing.ML*
ML-file *program-analysis.ML*

context

begin
ML-file *cached-theory-simproc.ML*
ML-file *UMM-Proofs.ML*
end

simproc-setup *size-of-bound* (‹*size-of TYPE('a::c-type)* ≤ *n*›) = ‹*K* (*fn ctxt* => *fn ct* =>
  let
    val - = (*case Thm.term-of ct of*
      @{*term-pat size-of ?T* ≤ - } => *if UMM-Proofs.is-ctype T then* () *else raise Match*
     | - => *raise Match*)
   val ctxt' = *ctxt addsimps* (*Named-Theorems.get ctxt* @{*named-theorems size-simps*})
    val eq = *Simplifier.asm-full-rewrite ctxt' ct*
    val - = *Utils.verbose-msg 5 ctxt* (*fn* - => *size-of-bound*: ^ *Thm.string-of-thm ctxt eq*)
    val rhs = *Thm.rhs-of eq* |> *Thm.term-of*
  in
    if rhs aconv ***term*** ‹*True*› *orelse rhs aconv* ***term*** ‹*False*› *then*
      SOME eq
    else
      NONE
  end
  handle Match => *NONE*)
›

named-theorems *enum-defs*

ML-file *heapstatetype.ML*
ML-file *MemoryModelExtras−sig.ML*
ML-file *MemoryModelExtras.ML*
ML-file *calculate-state.ML*
ML-file *syntax-transforms.ML*
ML-file *expression-translation.ML*
ML-file *modifies-proofs.ML*
ML-file *HPInter.ML*
ML-file *stmt-translation.ML*

method-setup *vcg* = ‹*Hoare.vcg* (∗|> *then-shorten-names*∗)›
    Simpl 'vcg' followed by C parser 'shorten-names'

method-setup *vcg-step* = ‹*Hoare.vcg-step* (∗|> *then-shorten-names*∗)›
    Simpl 'vcg-step' followed by C parser 'shorten-names'

declare *typ-info-word* [*simp del*]
declare *typ-info-ptr* [*simp del*]

lemma *valid-call-Spec-eq-subset*:

Γ' *procname* = *Some* (*Spec R*) $\Longrightarrow$
HoarePartialDef.valid Γ' *NF P* (*Call procname*) *Q A* = (*P* $\subseteq$ *fst* ‘ *R* $\wedge$ (*R* $\subseteq$ (− *P*) $\times$ *UNIV* $\cup$ *UNIV* $\times$ *Q*))
apply (*safe*; *simp?*)
subgoal for *x*
apply (*clarsimp simp*: *HoarePartialDef.valid-def*)
apply (*rule ccontr*)
apply (*drule-tac x=Normal x* **in** *spec*, *elim allE*)
apply (*drule mp*, *erule exec.Call*, *rule exec.SpecStuck*)
apply (*auto simp*: *image-def*)[*2*]
done
apply (*clarsimp simp*: *HoarePartialDef.valid-def*)
apply (*elim allE*, *drule mp*, *erule exec.Call*, *erule exec.Spec*)
apply *auto*[*1*]
apply (*clarsimp simp*: *HoarePartialDef.valid-def*)
apply (*erule exec-Normal-elim-cases*, *simp-all*)
apply (*erule exec-Normal-elim-cases*, *auto*)
done

lemma *creturn-wp* [*vcg-hoare*]:
assumes *P* $\subseteq$ {*s*. (*exnupd* (λ-. *Return*)) (*rvupd* (λ-. *v s*) *s*) $\in$ *A*}
shows Γ,$\Theta\vdash_{/F}$ *P creturn exnupd rvupd v Q*, *A*
unfolding *creturn-def*
by *vcg*

lemma *creturn-wp-total* [*vcg-hoare*]:
assumes *P* $\subseteq$ {*s*. (*exnupd* (λ-. *Return*)) (*rvupd* (λ-. *v s*) *s*) $\in$ *A*}
shows Γ,$\Theta\vdash_{t/F}$ *P creturn exnupd rvupd v Q*, *A*
unfolding *creturn-def*
by *vcg*

lemma *creturn-void-wp* [*vcg-hoare*]:
assumes *P* $\subseteq$ {*s*. (*exnupd* (λ-. *Return*)) *s* $\in$ *A*}
shows Γ,$\Theta\vdash_{/F}$ *P creturn-void exnupd Q*, *A*
unfolding *creturn-void-def*
by *vcg*

lemma *creturn-void-wp-total* [*vcg-hoare*]:
assumes *P* $\subseteq$ {*s*. (*exnupd* (λ-. *Return*)) *s* $\in$ *A*}
shows Γ,$\Theta\vdash_{t/F}$ *P creturn-void exnupd Q*, *A*
unfolding *creturn-void-def*
by *vcg*

lemma *cbreak-wp* [*vcg-hoare*]:
assumes *P* $\subseteq$ {*s*. (*exnupd* (λ-. *Break*)) *s* $\in$ *A*}
shows Γ,$\Theta\vdash_{/F}$ *P cbreak exnupd Q*, *A*

unfolding *cbreak-def*
by *vcg*

lemma *cbreak-wp-totoal* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ (\lambda\text{-}.\ Break))\ s \in A\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ cbreak\ exnupd\ Q,\ A$
unfolding *cbreak-def*
by *vcg*

lemma *ccatchbrk-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ s = Break \longrightarrow s \in Q) \wedge$
$(exnupd\ s \neq Break \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{/F} P\ ccatchbrk\ exnupd\ Q,\ A$
unfolding *ccatchbrk-def*
by *vcg*

lemma *ccatchbrk-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ s = Break \longrightarrow s \in Q) \wedge$
$(exnupd\ s \neq Break \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ ccatchbrk\ exnupd\ Q,\ A$
unfolding *ccatchbrk-def*
by *vcg*

lemma *cgoto-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ (\lambda\text{-}.\ Goto\ l))\ s \in A\}$
shows $\Gamma,\Theta\vdash_{/F} P\ cgoto\ l\ exnupd\ Q,\ A$
unfolding *cgoto-def*
by *vcg*

lemma *cgoto-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ (\lambda\text{-}.\ Goto\ l))\ s \in A\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ cgoto\ l\ exnupd\ Q,\ A$
unfolding *cgoto-def*
by *vcg*

lemma *ccatchgoto-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ s = Goto\ l \longrightarrow s \in Q) \wedge$
$(exnupd\ s \neq Goto\ l \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{/F} P\ ccatchgoto\ l\ exnupd\ Q,\ A$
unfolding *ccatchgoto-def*
by *vcg*

lemma *ccatchgoto-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ s = Goto\ l \longrightarrow s \in Q) \wedge$
$(exnupd\ s \neq Goto\ l \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ ccatchgoto\ l\ exnupd\ Q,\ A$
unfolding *ccatchgoto-def*
by *vcg*

lemma *ccatchreturn-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (\textit{is-local}\ (\textit{exnupd}\ s) \longrightarrow s \in Q) \wedge$
$(\neg\ \textit{is-local}\ (\textit{exnupd}\ s) \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{/F} P\ \textit{ccatchreturn}\ \textit{exnupd}\ Q,\ A$
unfolding *ccatchreturn-def*
by *vcg*

lemma *ccatchreturn-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (\textit{is-local}\ (\textit{exnupd}\ s) \longrightarrow s \in Q) \wedge$
$(\neg\ \textit{is-local}\ (\textit{exnupd}\ s) \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ \textit{ccatchreturn}\ \textit{exnupd}\ Q,\ A$
unfolding *ccatchreturn-def*
by *vcg*

lemma *cexit-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ \textit{exnupd}\ s \in A\}$
shows $\Gamma,\Theta\vdash_{/F} P\ \textit{cexit}\ \textit{exnupd}\ Q,\ A$
unfolding *cexit-def*
by *vcg*

lemma *cexit-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ \textit{exnupd}\ s \in A\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ \textit{cexit}\ \textit{exnupd}\ Q,\ A$
unfolding *cexit-def*
by *vcg*

lemma *cchaos-wp* [*vcg-hoare*]:
assumes $P \subseteq\ \{s.\ \forall x.\ (v\ x\ s) \in Q\ \}$
shows $\Gamma,\Theta\vdash_{/F} P\ \textit{cchaos}\ v\ Q,\ A$
unfolding *cchaos-def*
using *assms* **by** (*blast intro*: *HoarePartial.Spec*)

lemma *cchaos-wp-total* [*vcg-hoare*]:
assumes $P \subseteq\ \{s.\ \forall x.\ (v\ x\ s) \in Q\ \}$
shows $\Gamma,\Theta\vdash_{t/F} P\ \textit{cchaos}\ v\ Q,\ A$
unfolding *cchaos-def*
using *assms* **by** (*blast intro*: *HoareTotal.Spec*)

lemma *lvar-nondet-init-wp* [*vcg-hoare*]:
$P \subseteq \{s.\ \forall v.\ (\textit{upd}\ (\lambda\text{-}.\ v))\ s \in Q\} \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ \textit{lvar-nondet-init}\ \textit{upd}\ Q,\ A$
unfolding *lvar-nondet-init-def*
by (*rule HoarePartialDef.conseqPre*, *vcg*, *auto*)

lemma *lvar-nondet-init-wp-total* [*vcg-hoare*]:
$P \subseteq \{s.\ \forall v.\ (\textit{upd}\ (\lambda\text{-}.\ v))\ s \in Q\} \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ \textit{lvar-nondet-init}\ \textit{upd}\ Q,\ A$
unfolding *lvar-nondet-init-def*
by (*rule HoareTotalDef.conseqPre*, *vcg*, *auto*)

lemma *Seq-propagate-precond*:
 $[\![\Gamma,\Theta\vdash_{/F} P\ c_1\ P,A;\ \Gamma,\Theta\vdash_{/F} P\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F} P\ (Seq\ c_1\ c_2)\ Q,A$
 apply (*rule hoarep.Seq*)
  apply *assumption*
 apply *assumption*
 done

lemma *Seq-propagate-precond-total*:
 $[\![\Gamma,\Theta\vdash_{t/F} P\ c_1\ P,A;\ \Gamma,\Theta\vdash_{t/F} P\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ (Seq\ c_1\ c_2)\ Q,A$
 apply (*rule hoaret.Seq*)
  apply *assumption*
 apply *assumption*
 done

lemma *mem-safe-lvar-init* [*simp*,*intro*]:
 assumes *upd*: $\bigwedge g\ v\ s.$ *globals-update g* (*upd* (λ-. *v*) *s*) = *upd* (λ-. *v*) (*globals-update g s*)
 assumes *acc*: $\bigwedge v\ s.$ *globals* (*upd* (λ-. *v*) *s*) = *globals s*
 shows *mem-safe* (*lvar-nondet-init upd*) *x*
 apply (*clarsimp simp*: *mem-safe-def lvar-nondet-init-def*)
 apply (*erule exec.cases*; *clarsimp simp*: *restrict-safe-def*)
 apply (*fastforce simp*: *restrict-safe-OK-def restrict-htd-def upd acc intro*: *exec.Spec*)
 apply (*simp add*: *exec-fatal-def*)
 apply (*fastforce simp*: *exec-fatal-def restrict-htd-def upd acc intro*!: *exec.SpecStuck*)
 done

lemma *intra-safe-lvar-nondet-init* [*simp*]:
 intra-safe (*lvar-nondet-init upd* :: (('*a*::*heap-state-type*',*'l*,*'e*,*'x*) *state-scheme*,*'p*,*'f*) *com*) =
 (∀Γ. *mem-safe* (*lvar-nondet-init upd* :: (('*a*::*heap-state-type*',*'l*,*'e*,*'x*) *state-scheme*,*'p*,*'f*) *com*)
  (Γ :: (('*a*,*'l*,*'e*,*'x*) *state-scheme*,*'p*,*'f*) *body*))
 by (*simp add*: *lvar-nondet-init-def*)

lemma *proc-deps-lvar-nondet-init* [*simp*]:
 proc-deps (*lvar-nondet-init upd*) Γ = {}
 by (*simp add*: *lvar-nondet-init-def*)

declare *word-neq-0-conv*[*simp*]

declare [[*hoare-use-generalise*=*true*]]
end

theory *Array-Selectors*
 imports

CTranslationSetup
keywords *array-selectors* :: *thy-defn*
begin

named-theorems *array-selectors-simps*

lemmas [*array-selectors-simps*] =— numerals
Arrays.arr-fupdate-same
Arrays.arr-fupdate-other
Numeral-Type.card-num0
Numeral-Type.card-num1
Numeral-Type.card-bit0
Numeral-Type.card-bit1
Nat.One-nat-def
Nat.mult-Suc-right
Nat.mult-0-right
Nat.Suc-not-Zero
Num.Suc-numeral
Num.eq-numeral-Suc
Num.Suc-eq-numeral
Num.less-Suc-numeral
Num.mult-num-simps
Num.add-num-simps
Num.pred-numeral-simps
Num.numeral-times-numeral
Num.num.distinct
Num.num.inject
Nat.add-0-right
Num.arith-simps
Num.more-arith-simps
Num.rel-simps
Nat.Zero-not-Suc

lemmas [*array-selectors-simps*] =
comp-apply
fcp-beta

ML-file *array-selectors.ML*

experiment
begin

definition *my-array* ≡ *fupdate 3*
(*apfst* (λ-. *0x30*) *o apsnd* (λ-. *43*))
(*fupdate 2* (*apfst* (λ-. *0x20*) *o apsnd* (λ-. *42*))
(*fupdate 1* (*apfst* (λ-. *0x10*) *o apsnd* (λ-. *41*))
(*fupdate 0* (*apfst* (λ-. *0x0*) *o apsnd* (λ-. *40*))
((*ARRAY* -. (*0*, *0*))::(*32 word* × *nat*)[*4*]))))

lemmas [*array-selectors-simps*] =
apfst-conv
apsnd-conv
— in applications, the update functions are from the recursive record package, therefore the recursive record package simpset is included by default

array-selectors (*no-recursive-record-simpset*)— does not make a difference here
my-array-sels **is** *my-array-def*

lemma *my-array*.[*0*] ≡ (*0*, *40*)
my-array.[*1*] ≡ (*0x10*, *41*)
my-array.[*Suc 0*] ≡ (*0x10*, *41*)
my-array.[*2*] ≡ (*0x20*, *42*)
my-array.[*3*] ≡ (*0x30*, *43*)
by (*rule my-array-sels*)+

end

end

theory *CTranslation*
imports
CTranslationSetup
Array-Selectors
keywords
new-C-include-dir:: *thy-decl*
and
include-C-file
install-C-types
install-C-file :: *thy-load*

begin
ML-file *isar-install.ML*

end

Chapter 15

Misc. Lemmas

theory *TypHeapLib*
imports *CTranslation*
begin

15.1 Abbreviations and helpers

definition *is-an-abbreviation* ≡ *True*

abbreviation
  clift ≡ *lift-t c-guard*

lemma *clift-def*: *is-an-abbreviation* **by** (*simp add*: *is-an-abbreviation-def*)

15.2 Basic operations

15.2.1 clift

lemma *c-guard-clift*:
  clift hp p = *Some x* ⟹ *c-guard p*
  by (*rule lift-t-g*)

lemma *clift-heap-update*:
  fixes *p* :: *'a* :: *mem-type ptr*
  shows *hrs-htd hp* ⊨$_t$ *p* ⟹ *clift* (*hrs-mem-update* (*heap-update p v*) *hp*) = (*clift hp*)(*p* ↦ *v*)
  unfolding *hrs-mem-update-def*
  apply (*cases hp*)
  apply (*simp add*: *split-def hrs-htd-def*)
  apply (*erule lift-t-heap-update*)
  done

lemma *clift-heap-update-same*:
  fixes *p* :: *'a* :: *mem-type ptr*

shows ⟦ *hrs-htd hp* ⊨$_t$ *p*; *typ-uinfo-t TYPE('a)* ⊥$_t$ *typ-uinfo-t TYPE('b)* ⟧
⟹ *clift* (*hrs-mem-update* (*heap-update p v*) *hp*) = (*clift hp* :: *'b* :: *mem-type typ-heap*)
unfolding *hrs-mem-update-def*
apply (*cases hp*)
apply (*simp add*: *split-def hrs-htd-def*)
apply (*erule lift-t-heap-update-same*)
apply *simp*
done

lemmas *clift-heap-update-same-td-name* = *clift-heap-update-same* [*OF - tag-disj-via-td-name*, *unfolded pad-typ-name-def*]

15.2.2 *h-val*

lemmas *h-val-clift* = *lift-t-lift* [**where** *g* = *c-guard*, *unfolded CTypesDefs.lift-def*, *simplified*]
lemma *h-val-clift'*:
clift hp p = *Some v* ⟹ *h-val* (*hrs-mem hp*) *p* = *v*
unfolding *hrs-mem-def* **by** (*cases hp*, *simp add*: *h-val-clift*)

15.2.3 *h-t-valid*

lemma *clift-Some-eq-valid*:
(∃ *v*. *clift hp p* = *Some v*) = (*hrs-htd hp* ⊨$_t$ *p*)
apply (*cases hp*)
apply (*simp add*: *lift-t-if hrs-htd-def*)
done

lemma *h-t-valid-clift-Some-iff*:
(*hrs-htd hp* ⊨$_t$ *p*) = (∃ *v*. *clift hp p* = *Some v*)
apply (*cases hp*)
apply (*simp add*: *lift-t-if hrs-htd-def*)
done

lemma *h-t-valid-clift*:
clift hp p = *Some v* ⟹ *hrs-htd hp* ⊨$_t$ *p*
apply (*cases hp*, *simp add*: *hrs-htd-def*)
apply (*erule lift-t-h-t-valid*)
done

lemma *c-guard-h-t-valid*:
hrs-htd hp ⊨$_t$ *p* ⟹ *c-guard p*
by (*simp add*: *h-t-valid-def*)

15.3 *field-lvalue*

15.3.1 *heap-update*

lemma *heap-update-field*:
⟦*field-ti TYPE('a :: packed-type) f = Some t*; *c-guard p*;
export-uinfo t = export-uinfo (typ-info-t TYPE('b :: packed-type))⟧
⟹ *heap-update (Ptr &(p→f) :: 'b ptr) v hp =*
heap-update p (update-ti t (to-bytes-p v) (h-val hp p)) hp
apply (*erule field-ti-field-lookupE*)
apply (*erule (2) packed-heap-super-field-update* [*unfolded typ-uinfo-t-def*])
done

lemma *heap-update-field'*:
⟦*field-ti TYPE('a :: packed-type) f = Some t*; *c-guard p*;
export-uinfo t = export-uinfo (typ-info-t TYPE('b :: packed-type))⟧
⟹ *heap-update (Ptr &(p→f) :: 'b ptr) v hp =*
heap-update p (update-ti-t t (to-bytes-p v) (h-val hp p)) hp
apply (*erule field-ti-field-lookupE*)
apply (*subst packed-heap-super-field-update* [*unfolded typ-uinfo-t-def*])
apply *assumption+*
apply (*drule export-size-of* [*simplified typ-uinfo-t-def*])
apply (*simp add*: *update-ti-t-def*)
done

lemma *heap-update-field-hrs*:
fixes *p :: 'a :: packed-type ptr* **and** *v :: 'b :: packed-type*
shows ⟦*field-ti TYPE('a) f = Some t*; *c-guard p*;
export-uinfo t = export-uinfo (typ-info-t TYPE('b))⟧
⟹ *hrs-mem-update (heap-update (Ptr &(p→f)) v) hp =*
hrs-mem-update (heap-update p (update-ti-t t (to-bytes-p v) (h-val (hrs-mem hp) p))) hp
unfolding *hrs-mem-update-def*
apply (*simp add*: *split-def*)
apply (*subst heap-update-field*)
apply *assumption*
apply *assumption*
apply *assumption*
apply (*frule export-size-of* [*unfolded typ-uinfo-t-def*])
apply (*simp add*: *update-ti-t-def hrs-mem-def*)
done

lemma *heap-update-field-ext*:
⟦*field-ti TYPE('a :: packed-type) f = Some t*; *c-guard p*;
export-uinfo t = export-uinfo (typ-info-t TYPE('b :: packed-type))⟧
⟹ *heap-update (Ptr &(p→f) :: 'b ptr) =*
(λv hp. heap-update p (update-ti t (to-bytes-p v) (h-val hp p)) hp)
apply (*rule ext*, *rule ext*)
apply (*erule (2) heap-update-field*)
done

15.3.2 *c-guard*

lemma *c-guard-field*:
⟦*c-guard* (*p* :: *'a* :: *mem-type ptr*); *field-ti TYPE*(*'a* :: *mem-type*) *f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*(*'b* :: *mem-type*))⟧
⟹ *c-guard* (*Ptr* &(*p*→*f*) :: *'b* :: *mem-type ptr*)
apply (*erule field-ti-field-lookupE*)
apply (*erule* (*1*) *c-guard-field-lvalue*)
apply (*simp add*: *typ-uinfo-t-def*)
done

15.3.3 clift

lemma *clift-field*:
fixes *v* :: *'a* :: *mem-type* **and** *p* :: *'a* :: *mem-type ptr*
assumes *lf*: *clift hp p* = *Some v*
and *fl*: *field-ti TYPE*(*'a*) *f* = *Some t*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*(*'b* :: *mem-type*))
shows *clift hp* (*Ptr* &(*p*→*f*) :: *'b* :: *mem-type ptr*) = *Some* (*from-bytes* (*access-ti*$_0$ *t v*))
using *lf fl eu*
apply (*clarsimp elim*!: *field-ti-field-lookupE*)
apply (*erule* (*2*) *lift-t-mono* [*unfolded typ-uinfo-t-def*])
apply (*rule c-guard-mono*)
done

Updates

lemma *clift-field-update*:
fixes *val* :: *'b* :: *mem-type* **and** *ptr* :: *'a* :: *mem-type ptr*
assumes *fl*: *field-ti TYPE*(*'a*) *f* = *Some t*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*(*'b*))
and *cl*: *clift hp ptr* = *Some z*
shows (*clift* (*hrs-mem-update* (*heap-update* (*Ptr* &(*ptr*→*f*)) *val*) *hp*)) =
(*clift hp*)(*ptr* ↦ *field-update* (*field-desc t*) (*to-bytes-p val*) *z*)
(**is** *?LHS* = *?RHS*)
proof −
have *cl'*: *clift* (*fst hp*, *snd hp*) *ptr* = *Some z* **using** *cl* **by** *simp*

have *?LHS* = *super-field-update-t* (*Ptr* &(*ptr*→*f*)) *val* (*clift* (*fst hp*, *snd hp*))
unfolding *hrs-mem-update-def split-def*
proof (*rule lift-t-super-field-update* [*OF h-t-valid-sub*, *unfolded typ-uinfo-t-def*])
from *cl'* **show** *snd hp* ⊨$_t$ *ptr* **by** (*rule lift-t-h-t-valid*)
show *TYPE*(*'b*) ≤$_\tau$ *TYPE*(*'a*) **using** *fl eu* **by** (*rule field-ti-sub-typ* [*unfolded typ-uinfo-t-def*])
qed *fact*+

also have ... = *?RHS* **using** *fl eu cl*
apply −
apply (*erule field-ti-field-lookupE*)

apply (*subst super-field-update-lookup* [*unfolded typ-uinfo-t-def*])
apply *assumption*
apply *simp*
apply *simp*
apply *simp*
done

finally show *?thesis* **.**
qed

lemma *clift-field-update-padding*:
fixes *val* :: *'b* :: *mem-type* **and** *ptr* :: *'a* :: *mem-type ptr*
assumes *fl*: *field-ti TYPE('a) f = Some t*
and *eu*: *export-uinfo t = export-uinfo (typ-info-t TYPE('b))*
and *cl*: *clift hp ptr = Some z*
and *lbs*: *length bs = size-of TYPE('b)*
shows (*clift* (*hrs-mem-update* (*heap-update-padding* (*Ptr &(ptr→f)*) *val bs*) *hp*))
=
(*clift hp*)(*ptr* ↦ *field-update* (*field-desc t*) (*to-bytes-p val*) *z*)
(**is** *?LHS = ?RHS*)
proof −
have *cl'*: *clift* (*fst hp*, *snd hp*) *ptr = Some z* **using** *cl* **by** *simp*

have *?LHS = super-field-update-t* (*Ptr &(ptr→f)*) *val* (*clift* (*fst hp*, *snd hp*))
unfolding *hrs-mem-update-def split-def*
proof (*rule lift-t-super-field-update-padding* [*OF h-t-valid-sub - lbs*, *unfolded typ-uinfo-t-def*])
from *cl'* **show** *snd hp* ⊨$_t$ *ptr* **by** (*rule lift-t-h-t-valid*)
show *TYPE('b)* ≤$_\tau$ *TYPE('a)* **using** *fl eu* **by** (*rule field-ti-sub-typ* [*unfolded typ-uinfo-t-def*])
qed *fact+*

also have *... = ?RHS* **using** *fl eu cl*
apply −
apply (*erule field-ti-field-lookupE*)
apply (*subst super-field-update-lookup* [*unfolded typ-uinfo-t-def*])
apply *assumption*
apply *simp*
apply *simp*
apply *simp*
done

finally show *?thesis* **.**
qed

15.3.4 cparent

definition
cparent :: (*'a* :: *c-type*) *ptr* ⇒ *string list* ⇒ (*'b* :: *c-type*) *ptr*
where

cparent p fs ≡ *THE p′*. *p* = *Ptr* &(*p′*→*fs*)

lemma *cparent-field* [*simp*]:
cparent (*Ptr* &(*p*→*fs*)) *fs* = *p*
unfolding *cparent-def*
by (*simp add*: *field-lvalue-def*)

lemma *cparent-def2*:
fixes *p* :: *′b* :: *c-type ptr*
shows *cparent p f* ≡ (*Ptr* (*ptr-val p* − *of-nat* (*field-offset TYPE*(*′a* :: *c-type*) *f*))
:: *′a* :: *c-type ptr*)
(**is** *cparent p f* ≡ *?p′*)
proof −
have *pv*: *p* = *Ptr* &(*?p′*→*f*)
by (*simp add*: *field-lvalue-def field-simps*)
show *cparent p f* ≡ *?p′*
by (*subst pv*) *simp*
qed

lemma *field-cparent* [*simp*]:
fixes *p* :: *′a* :: *c-type ptr*
shows (*Ptr* &(*cparent p f* :: *′b* :: *c-type ptr* →*f*)) = *p*
by (*simp add*: *cparent-def2 field-lvalue-def*)

lemma *clift-cparentE*:
fixes *v* :: *′a* :: *mem-type* **and** *p* :: *′b* :: *mem-type ptr*
assumes *lf*: *clift hp* (*cparent p fs* :: *′a ptr*) = *Some v*
and *fl*: *field-ti TYPE*(*′a*) *fs* = *Some t*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*(*′b*))
shows *clift hp p* = *Some* (*from-bytes* (*access-ti*$_0$ *t v*))
unfolding *cparent-def*
proof −
have *pv*: *p* = *Ptr* &((*Ptr* (*ptr-val p* − *of-nat* (*field-offset TYPE*(*′a*) *fs*)) :: *′a ptr*)→*fs*)
(**is** *p* = *Ptr* &(*?p′*→*fs*)) **by** (*simp add*: *field-lvalue-def field-simps*)
have *cp*: *cparent p fs* = *?p′*
by (*subst pv*) *simp*

from *lf* **have** *lf′*: *clift hp ?p′* = *Some v*
by (*simp add*: *cparent-def2*)

have *clift hp* (*Ptr* &(*?p′*→*fs*) :: *′b ptr*) = *Some* (*from-bytes* (*access-ti*$_0$ *t v*)) **using** *lf′ fl eu*
by (*rule clift-field*)

thus *?thesis*
by (*simp add*: *pv* [*symmetric*])
qed

lemma *heap-update-to-cparent*:
fixes *p* :: *'b* :: *packed-type ptr* **and** *fs* :: *char list list*
defines *cp* ≡ *cparent p fs* :: *'a* :: *packed-type ptr*
assumes *fl*: *field-ti TYPE('a* :: *packed-type) fs* = *Some t*
and *cg*: *c-guard cp*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE('b)*)
shows *heap-update p v hp* = *heap-update cp* (*update-ti t* (*to-bytes-p v*) (*h-val hp cp*)) *hp*
(**is** *?LHS* = *?RHS*)
proof −
have *?LHS* = *heap-update* (*Ptr* &(*cp*→*fs*)) *v hp*
unfolding *cp-def* **by** *simp*

moreover have ... = *?RHS* **using** *fl cg eu*
by (*fastforce intro*: *heap-update-field*)

finally show *?thesis* **.**
qed

lemma *c-guard-cparent*:
⟦ *c-guard* ((*cparent p f*)::*'a*::*mem-type ptr*);
field-ti TYPE('a) f = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE('b)* ⟧ ⟹
c-guard (*p*::*'b*::*mem-type ptr*)
apply(*subst field-cparent*[*symmetric*, **where** *f*=*f*])
apply(*rule c-guard-field*, (*simp add*:*typ-uinfo-t-def*)+)
done

lemma *parent-update-child*:
fixes *p*::*'b*::*packed-type ptr*
shows
⟦ *c-guard* ((*cparent p f*)::*'a*::*packed-type ptr*); *field-ti TYPE('a) f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE('b)*)⟧
⟹ *hrs-mem-update* (*heap-update p v*) *hp* =
hrs-mem-update
(*heap-update* ((*cparent p f*)::*'a ptr*)
(*update-ti-t t* (*to-bytes-p v*) (*h-val* (*hrs-mem hp*) (*cparent p f*))))
hp
apply(*subst field-cparent*[*symmetric*, **where** *f*=*f* **and** *'b*=*'a*])
apply(*rule heap-update-field-hrs*, *simp*+)
done

15.3.5 *h-val*

lemma *h-val-field-clift*:
fixes *pa* :: *'a* :: *mem-type ptr*
assumes *cl*: *clift* (*h*, *d*) *pa* = *Some v*
and *fl*: *field-ti TYPE('a) f* = *Some t*

and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*('*b* :: *mem-type*))
shows *h-val h* (*Ptr* &(*pa*→*f*) :: '*b* :: *mem-type ptr*) = *from-bytes* (*access-ti*$_0$ *t v*)
using *cl fl eu*
apply −
apply (*rule h-val-clift*)
apply (*clarsimp simp*: *field-ti-def split*: *option.splits*)
apply (*erule* (2) *lift-t-mono* [*unfolded typ-uinfo-t-def*])
apply (*rule c-guard-mono*)
done

lemma *h-val-field-clift'*:
fixes *pa* :: '*a* :: *mem-type ptr*
assumes *cl*: *clift hp pa* = *Some v*
and *fl*: *field-ti TYPE*('*a*) *f* = *Some t*
and *eu*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b* :: *mem-type*)
shows *h-val* (*hrs-mem hp*) (*Ptr* &(*pa*→*f*) :: '*b* :: *mem-type ptr*) = *from-bytes* (*access-ti*$_0$ *t v*)
using *cl fl eu*
apply (*cases hp*)
apply (*simp add*: *h-val-field-clift hrs-mem-def typ-uinfo-t-def*)
done

lemma *clift-subtype*:
⟦ *clift hp* ((*cparent p f*)::'*a*::*mem-type ptr*) = *Some v*;
field-ti TYPE('*a*) *f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*('*b*::*mem-type*)) ⟧ ⟹
clift hp (*p*::'*b ptr*) = *Some* (*from-bytes* (*access-ti*$_0$ *t v*))
apply(*subst field-cparent*[*symmetric*, **where** *f*=*f*])
apply(*rule clift-field*)
apply(*simp*)+
done

15.3.6 *h-t-valid*

lemma *h-t-valid-field*:
fixes *p* :: '*a* :: *mem-type ptr*
assumes *htv*: *d* ⊨$_t$ *p*
and *fti*: *field-ti TYPE*('*a* :: *mem-type*) *f* = *Some t*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*('*b* :: *mem-type*))
shows *d* ⊨$_t$ (*Ptr* &(*p*→*f*) :: '*b* :: *mem-type ptr*)
using *htv fti eu*
apply −
apply (*clarsimp simp*: *field-ti-def split*: *option.splits*)
apply (*erule* (1) *h-t-valid-mono* [*rule-format*, *unfolded typ-uinfo-t-def*])
apply (*rule c-guard-mono*)
apply *assumption*
done

lemma *h-t-valid-field'*:
fixes *p*::'*a*::*mem-type ptr*
shows
⟦ *field-ti TYPE*('*a*) *f* = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('*b*);
d,*g* ⊨$_t$ *p*; *g'* ((*Ptr* &(*p*→*f*))::'*b*::*mem-type ptr*) ⟧ ⟹ *d*,*g'* ⊨$_t$ *Ptr* &(*p*→*f*)
apply(*simp add*:*h-t-valid-guard-subst*[*OF h-t-valid-sub*])
done

lemma *h-t-valid-c-guard-field*:
fixes *p*::'*a*::*mem-type ptr*
shows
⟦ *d* ⊨$_t$ *p*;
field-ti TYPE('*a*) *f* = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
d ⊨$_t$ ((*Ptr* &(*p*→*f*))::'*b*::*mem-type ptr*)
apply(*simp add*:*h-t-valid-field c-guard-field h-t-valid-c-guard typ-uinfo-t-def*)
done

lemma *h-t-valid-cparent*:
⟦ *field-ti TYPE*('*a*) *f* = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('*b*);
d,*g* ⊨$_t$ ((*cparent p f*)::'*a*::*mem-type ptr*); *g'* (*p*::'*b*::*mem-type ptr*) ⟧ ⟹
d,*g'* ⊨$_t$ *p*
apply(*subst field-cparent*[*symmetric*, **where** *f*=*f*])
apply(*rule h-t-valid-field'*)
apply(*assumption*)
apply(*simp add*:*typ-uinfo-t-def*)
apply(*assumption*)
apply(*simp*)
done

lemma *h-t-valid-c-guard-cparent*:
fixes *p*::'*b*::*mem-type ptr*
shows
⟦ *d* ⊨$_t$ ((*cparent p f*)::'*a*::*mem-type ptr*);
field-ti TYPE('*a*) *f* = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
d ⊨$_t$ *p*
apply(*rule h-t-valid-cparent*)
apply(*assumption*)
apply(*simp add*:*typ-uinfo-t-def*)
apply(*assumption*)
apply(*rule c-guard-cparent*)
apply(*rule h-t-valid-c-guard*, *assumption*)
apply(*assumption*)
apply(*simp add*:*typ-uinfo-t-def*)
done

lemma *c-guard-array-c-guard*:
c-guard (*ptr-coerce p* :: (*'b* :: *c-type*, *'a* :: *finite*) *array ptr*) ⟹ *c-guard* (*p* :: *'b ptr*)
apply (*clarsimp simp*: *c-guard-def*)
apply (*rule conjI*)
apply (*clarsimp simp*: *ptr-aligned-def align-of-def align-td-array*)
apply (*simp add*: *c-null-guard-def*)
apply (*erule contra-subsetD* [*rotated*])
apply (*rule intvl-start-le*)
apply *simp*
done

lemma *c-guard-array-field*:
assumes *parent-cguard*: *c-guard* (*p* :: *'a* :: *mem-type ptr*)
and *subfield*: *field-ti TYPE*(*'a* :: *mem-type*) *f* = *Some t*
and *type-match*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*((*'b* :: *array-outer-max-size*, *'c* :: *array-max-count*) *array*))
shows *c-guard* (*Ptr* &(*p*→*f*) :: *'b ptr*)
by (*metis c-guard-array-c-guard c-guard-field parent-cguard ptr-coerce.simps subfield type-match*)

instantiation *ptr* :: (*type*) *enum*
begin

definition *enum-ptr* ≡ *map Ptr enum-class.enum*
definition *enum-all-ptr P* ≡ *enum-class.enum-all* (λ*v*. *P* (*Ptr v*))
definition *enum-ex-ptr P* ≡ *enum-class.enum-ex* (λ*v*. *P* (*Ptr v*))

instance
apply (*intro-classes*)
apply (*clarsimp simp*: *enum-ptr-def*)
apply (*metis ptr.exhaust surj-def*)
apply (*clarsimp simp*: *enum-ptr-def distinct-map*)
apply (*metis injI ptr.inject*)
apply (*clarsimp simp*: *enum-all-ptr-def*)
apply (*rule iffI*)
apply (*rule allI*)
apply (*rename-tac x*)
apply (*erule-tac x*=*ptr-val x* **in** *allE*)
apply *force*
apply *force*
apply (*clarsimp simp*: *enum-ex-ptr-def*)
apply (*rule iffI*)
apply *force*
apply *clarsimp*
subgoal for *P x*
apply (*rule exI*[**where** *x*=*ptr-val x*])
apply *clarsimp*
done

done
end

15.3.7 Type Combinators and Padding

lemma *ti-typ-pad-combine-empty-ti*:
fixes *tp* :: *'b* :: *c-type itself*
shows *ti-typ-pad-combine tp lu upd algn fld* (*empty-typ-info algn' n*) =
TypDesc (*max algn'* (*max algn* (*align-td* (*typ-info-t TYPE*('*b*))))))
(*TypAggregate* [*DTuple* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *lu upd*) *fld*
(|*field-access* = *xto-bytes* ∘ *lu*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*b*)|)] *n*
by (*simp add*: *ti-typ-pad-combine-def ti-typ-combine-def empty-typ-info-def Let-def*)

lemma *ti-typ-combine-empty-ti*:
fixes *tp* :: *'b* :: *c-type itself*
shows *ti-typ-combine tp lu upd algn fld* (*empty-typ-info algn' n*) =
TypDesc (*max algn'* (*max algn* (*align-td* (*typ-info-t TYPE*('*b*))))))
(*TypAggregate* [*DTuple* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *lu upd*) *fld*
(|*field-access* = *xto-bytes* ∘ *lu*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*b*)|)] *n*
by (*simp add*: *ti-typ-combine-def empty-typ-info-def Let-def*)

lemma *ti-typ-pad-combine-td*:
fixes *tp* :: *'b* :: *c-type itself*
shows *padup* (*max* (*2* ^ *algn*) (*align-of TYPE*('*b*))) (*size-td-struct st*) = *0* ⟹
ti-typ-pad-combine tp lu upd algn fld (*TypDesc algn' st n*) =
TypDesc (*max algn'* (*max algn* (*align-td* (*typ-info-t TYPE*('*b*))))))
(*extend-ti-struct st* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *lu upd*) *fld*
(|*field-access* = *xto-bytes* ∘ *lu*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*b*)|)) *n*
by (*simp add*: *ti-typ-pad-combine-def ti-typ-combine-def Let-def*)

lemma *ti-typ-combine-td*:
fixes *tp* :: *'b* :: *c-type itself*
shows *padup* (*align-of TYPE*('*b*)) (*size-td-struct st*) = *0* ⟹
ti-typ-combine tp lu upd algn fld (*TypDesc algn' st n*) =
TypDesc (*max algn'* (*max algn* (*align-td* (*typ-info-t TYPE*('*b*))))))
(*extend-ti-struct st* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *lu upd*) *fld*
(|*field-access* = *xto-bytes* ∘ *lu*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*b*)|)) *n*
by (*simp add*: *ti-typ-combine-def Let-def*)

lemma *update-ti-t-pad-combine*:
assumes *std*: *size-td td' mod 2* ^ (*max algn* (*align-td* (*typ-info-t TYPE*('*a* ::

c-type)))) = *0*
shows *update-ti-t* (*ti-typ-pad-combine TYPE*('*a* :: *c-type*) *lu upd algn fld td*') *bs v* =
update-ti-t (*ti-typ-combine TYPE*('*a* :: *c-type*) *lu upd algn fld td*') *bs v*
using *std*
by (*simp add*: *ti-typ-pad-combine-def size-td-simps Let-def max-2-exp*)

15.3.8 The orphanage: miscellaneous lemmas pulled up to (roughly) where they belong.

lemma *uinfo-array-tag-n-m-not-le-typ-name*:
typ-name (*typ-info-t TYPE*('*b*)) @ ''*-array-*'' @ *nat-to-bin-string m*
∉ *td-names* (*typ-info-t TYPE*('*a*))
⟹ ¬ *uinfo-array-tag-n-m TYPE*('*b* :: *c-type*) *n m* ≤ *typ-uinfo-t TYPE*('*a* :: *c-type*)
apply (*clarsimp simp*: *typ-tag-le-def typ-uinfo-t-def*)
apply (*drule td-set-td-names*)
apply (*clarsimp simp*: *uinfo-array-tag-n-m-def typ-uinfo-t-def*)
apply (*drule arg-cong*[**where** *f*=λ*xs*. *set* ''*r*'' ⊆ *set xs*], *simp*)
apply (*simp add*: *uinfo-array-tag-n-m-def typ-uinfo-t-def*)
done

lemma *tag-not-le-via-td-name*:
typ-name (*typ-info-t TYPE*('*a*)) ∉ *td-names* (*typ-info-t TYPE*('*b*))
⟹ *typ-name* (*typ-info-t TYPE*('*a*)) ≠ *pad-typ-name*
⟹ ¬ *typ-uinfo-t TYPE*('*a* :: *c-type*) ≤ *typ-uinfo-t TYPE* ('*b* :: *c-type*)
apply (*clarsimp simp*: *typ-tag-le-def typ-uinfo-t-def*)
apply (*drule td-set-td-names*, *simp*+)
done

lemmas *typ-heap-simps* =
— *c-guard*
c-guard-field
c-guard-h-t-valid
— *h-t-valid*
h-t-valid-field
h-t-valid-clift
— *h-val*
h-val-field-clift'
h-val-clift'
— *clift*
clift-field
clift-field-update
heap-update-field-hrs
heap-update-field'

clift-heap-update
clift-heap-update-same-td-name — Try this last (is expensive)

end

theory *LemmaBucket-C*
imports
More-Lib
WordSetup
TypHeapLib
ArrayAssertion
begin

declare *word-neq-0-conv* [*simp del*]

lemma *Ptr-not-null-pointer-not-zero*: $(Ptr\ p \neq NULL) = (p \neq 0)$
by *simp*

lemma *hrs-mem-f*: $f\ (hrs\text{-}mem\ s) = hrs\text{-}mem\ (hrs\text{-}mem\text{-}update\ f\ s)$
apply (*cases s*)
apply (*clarsimp simp*: *hrs-mem-def hrs-mem-update-def*)
done

lemma *hrs-mem-heap-update*:
$heap\text{-}update\ p\ v\ (hrs\text{-}mem\ s) = hrs\text{-}mem\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ p\ v)\ s)$
apply (*rule hrs-mem-f*)
done

lemma *surj-Ptr* [*simp*]:
surj Ptr
by (*rule surjI* [**where** $f = ptr\text{-}val$], *simp*)

lemma *inj-Ptr* [*simp*]:
inj Ptr
apply (*rule injI*)
apply *simp*
done

lemma *bij-Ptr* :
bij Ptr
by (*simp add*: *bijI*)

lemma *exec-Guard*:
$(G \vdash \langle Guard\ Err\ S\ c, Normal\ s\rangle \Rightarrow s')$
$= (if\ s \in S\ then\ G \vdash \langle c, Normal\ s\rangle \Rightarrow s'$
$else\ s' = Fault\ Err)$
by (*auto split*: *if-split elim*!: *exec-elim-cases intro*: *exec.intros*)

lemma *byte-ptr-guarded:ptr-val* (*x*::*8 word ptr*) ≠ *0* ⟹ *c-guard x*
unfolding *c-guard-def c-null-guard-def ptr-aligned-def*
by (*clarsimp simp*: *intvl-Suc*)

lemma *intvl-aligned-bottom-eq*:
fixes *p* :: *'a*::*len word*
assumes *al1*: *is-aligned x n*
and *al2*: *is-aligned p bits*
and *nb*: ¬ *n* < *bits*
and *off*: *off* ≤ *2* ^ *bits off* ≠ *0*
shows (*x* ∈ {*p* ..+ *off*}) = (*x* = *p*)
proof (*rule iffI*)
assume *x* = *p*
thus *x* ∈ {*p* ..+ *off*} **using** *off*
by (*simp add*: *intvl-self*)
next
assume *x-in-intvl*: *x* ∈ {*p* ..+ *off*}

show *x* = *p*
proof *cases*
assume *wb*: *bits* < *len-of TYPE*(*'a*)

from *x-in-intvl* **obtain** *kp* **where** *xp*: *x* = *p* + *of-nat kp* **and** *kp*: *kp* < *off*
by (*clarsimp dest*!: *intvlD*)

hence *is-aligned* (*p* + *of-nat kp*) *n* **using** *al1* **by** *simp*
hence *2* ^ *n dvd unat* (*p* + *of-nat kp*) **unfolding** *is-aligned-def* **.**
hence *2* ^ *n dvd unat p* + *kp* **using** *kp off wb*
apply −
apply (*subst* (*asm*) *iffD1* [*OF unat-plus-simple*])
apply (*rule is-aligned-no-wrap'* [*OF al2*])
apply (*rule of-nat-power*)
apply *simp-all*[*2*]
apply (*subst* (*asm*) *unat-of-nat*)
apply (*subst* (*asm*) *mod-less*)
apply (*erule order-less-le-trans*)
apply (*erule order-trans*)
apply *simp*
apply *simp*
done

moreover from *al2* **obtain** *q2* **where** *pbits*: *p* = *2* ^ *bits* ∗ *of-nat q2*
and *q2*: *q2* < *2* ^ (*len-of TYPE*(*'a*) − *bits*)
by (*rule is-alignedE*)

moreover from *nb* **obtain** *kn* **where** *nbits*: *n* = *bits* + *kn*
by (*clarsimp simp*: *linorder-not-less le-iff-add*)

```
  ultimately have 2 ^ bits dvd 2 ^ bits * q2 + kp
    apply (simp add: power-add)
    apply (simp add: unat-mult-power-lem [OF q2])
    apply (erule dvd-mult-left)
    done

  hence 2 ^ bits dvd kp by (simp add: dvd-reduce-multiple)
  with kp have kp = 0
    apply -
    apply (erule contrapos-pp)
    apply (simp add: linorder-not-less)
    apply (drule (1) dvd-imp-le)
    apply (erule order-trans [OF off(1)])
    done

  thus ?thesis using xp by simp
  next
    assume wb: ¬ bits < len-of TYPE('a)
    with assms
    show ?thesis by (simp add: is-aligned-mask mask-def power-overflow)
  qed
qed

lemma intvl-mem-weaken: x ∈ {p..+a - n} ⟹ x ∈ {p..+a}
  apply -
  apply (drule intvlD)
  apply clarsimp
  apply (rule intvlI)
  apply simp
  done


lemma upto-intvl-eq:
  fixes x :: 'a::len word
  assumes al: is-aligned x n
  shows {x..+2 ^ n} = {x .. x + 2 ^ n - 1}
proof cases
  assume n < len-of TYPE('a)
  with assms show ?thesis
  unfolding intvl-def
  apply simp
  apply standard
   apply clarsimp
   apply (subgoal-tac of-nat k < (2 :: 'a word) ^ n)
    apply (intro conjI)
     apply (erule (1) is-aligned-no-wrap')
    apply (subst p-assoc-help)
    apply (rule word-plus-mono-right)
     apply (simp add: word-less-sub-1)
```

```
    apply (simp add: field-simps is-aligned-no-overflow)
   apply (simp add: of-nat-power)
  apply clarsimp
  subgoal for xa
    apply (rule exI[where x = unat (xa - x)])
    apply clarsimp
    apply (rule unat-less-power, assumption)
    apply (subst word-less-sub-le [symmetric])
     apply assumption
    apply (rule word-diff-ls'(4))
     apply (simp add: field-simps)
    apply assumption
    done
  done
next
  assume ¬ n < len-of TYPE('a)
  with assms show ?thesis
    apply (simp add: is-aligned-mask mask-def power-overflow intvl-def)
    apply (rule set-eqI)
    apply clarsimp
     by (metis (no-types, opaque-lifting) le-less-trans nat-power-less-imp-less not-le
power-eq-0-iff
   power-zero-numeral unat-lt2p word-unat.Rep-inverse)
qed


lemma upto-intvl-eq':
  fixes x :: 'a :: len word
  shows ⟦ x ≤ x + (of-nat b - 1); b ≠ 0; b ≤ 2 ^ len-of TYPE('a)⟧ ⟹ {x..+b}
= {x .. x + of-nat b - 1}
  unfolding intvl-def
  supply unsigned-of-nat
  apply standard
   apply clarsimp
   apply (subgoal-tac of-nat k ≤ (of-nat (b - 1) :: 'a word))
    apply (intro conjI)
     apply (erule word-random)
     apply simp
    apply (subst field-simps [symmetric], rule word-plus-mono-right)
     apply simp
    apply assumption
   apply (subst More-Word.of-nat-mono-maybe-le [symmetric])
     apply simp
    apply simp
   apply simp
  apply clarsimp
  subgoal for xa
    apply (rule exI[where x = unat (xa - x)])
    apply simp
```

```
    apply (simp add: unat-sub)
    apply (rule nat-diff-less)
     apply (subst (asm) word-le-nat-alt, erule order-le-less-trans)
     apply (subst add-diff-eq[symmetric], subst unat-plus-if')
     apply (simp add: no-olen-add-nat)
     apply (simp add: le-eq-less-or-eq)
     apply (erule disjE)
      apply (subst unat-minus-one)
       apply (erule (1) of-nat-neq-0)
      apply (simp add: unat-of-nat)
     apply (erule ssubst, rule unat-lt2p)
    apply (simp add: word-le-nat-alt)
    done
  done

lemma intvl-aligned-top:
  fixes x :: 'a::len word
  assumes al1: is-aligned x n
  and     al2: is-aligned p bits
  and      nb: n ≤ bits
  and    offn: off < 2 ^ n
  and      wb: bits < len-of TYPE('a)
  shows  (x ∈ {p ..+ 2 ^ bits − off}) = (x ∈ {p ..+ 2 ^ bits})
proof (rule iffI)
  assume x ∈ {p..+2 ^ bits − off}
  thus x ∈ {p..+2 ^ bits} by (rule intvl-mem-weaken)
next
  assume asm: x ∈ {p..+2 ^ bits}

  show x ∈ {p..+2 ^ bits − off}
  proof (cases n = 0)
    case True
    with offn asm show ?thesis by simp
  next
    case False

    from asm have x ∈ {p .. p + 2 ^ bits − 1}
      by (simp add: upto-intvl-eq [OF al2])
    then obtain q where xp: x = p + of-nat (q * 2 ^ n) and qb: q < 2 ^ (bits −
n) using False nb
      by (fastforce dest!: is-aligned-diff[OF al1 al2 wb,simplified field-simps])

    have q * 2 ^ n < 2 ^ bits − off
    proof −
      show ?thesis using offn qb nb
        apply (simp add: less-diff-conv)
        apply (erule (1) nat-add-offset-less)
        apply arith
        done
```

```
    qed

    with xp show ?thesis
      apply −
      apply (erule ssubst)
      apply (erule intvlI)
      done
  qed
qed



lemma heap-update-list-update:
  fixes v :: word8
  shows x ≠ y ⟹ heap-update-list s xs (hp(y := v)) x = heap-update-list s xs hp
x
  apply (induct xs rule: rev-induct)
   apply simp
  apply (simp add: heap-update-list-append cong: if-cong)
  done


lemma heap-update-list-append2:
  length xs + length ys < 2 ^ word-bits ⟹
    heap-update-list s (xs @ ys) hp
      = heap-update-list s xs (heap-update-list (s + of-nat (length xs)) ys hp)
proof (induct xs arbitrary: hp s)
  case Nil
  show ?case by simp
next
  case (Cons v′ vs′)

  have (1 :: addr) + of-nat (length vs′) = of-nat (length (v′ # vs′))
    by simp
  also have ... ≠ 0 using Cons.prems
    apply −
    apply (rule of-nat-neq-0)
    apply simp
    apply (simp add: word-bits-conv)
    done
  finally have neq0: (1 :: addr) + of-nat (length vs′) ≠ 0 .

  have (1 :: addr) + of-nat (length vs′) = of-nat (length (v′ # vs′))
    by simp
  also have unat ... + length ys < 2 ^ word-bits using Cons.prems
    apply (subst unat-of-nat)
    apply (simp add: word-bits-conv)
    done
   finally have lt: unat ((1 :: addr) + of-nat (length vs′)) + length ys < 2 ^
```

word-bits **.**

from *Cons.prems* **have** *length vs′ + length ys < 2 ^ word-bits* **by** *simp*
thus *?case*
apply *simp*
apply (*subst Cons.hyps*, *assumption*)
apply (*rule arg-cong* [**where** *f* = *heap-update-list* (*s + 1*) *vs′*])
apply (*rule ext*)
subgoal for *x*
apply (*cases x = s*)
apply *simp*
apply (*subst heap-update-nmem-same*)
apply (*subst add.assoc*)
apply (*rule intvl-nowrap*[*OF neq0 order-less-imp-le*
[*OF lt*[*unfolded word-bits-def*]]])
apply *simp*
apply (*clarsimp simp*: *heap-update-list-update field-simps*)
done
done
qed

lemma *heap-update-word8*:
heap-update p (*v* :: *word8*) *hp* = *hp*(*ptr-val p* := *v*)
unfolding *heap-update-def* **by** (*simp add*: *to-bytes-word8*)

lemma *index-foldr-update2*:
⟦ *n* ≤ *i*; *i* < *CARD*('*b*::*finite*) ⟧ ⟹ *index* (*foldr* (λ*n arr*. *Arrays.update arr n m*) [*0*..<*n*] (*x* :: ('*a*,'*b*) *array*)) *i* = *index x i*
apply (*induct n arbitrary*: *x*)
apply *simp*
apply *simp*
done

lemma *index-foldr-update*:
⟦ *i* < *n*; *n* ≤ *CARD*('*b*::*finite*) ⟧ ⟹ *index* (*foldr* (λ*n arr*. *Arrays.update arr n m*) [*0*..<*n*] (*x* :: ('*a*,'*b*) *array*)) *i* = *m*
apply (*induct n arbitrary*: *x*)
apply *simp*
apply *simp*
apply (*erule less-SucE*)
apply *simp*
apply *simp*
apply (*subst index-foldr-update2*)
apply *simp*
apply *simp*
apply *simp*
done

lemma *intvl-disjoint1*:

fixes $a :: 'a :: len\ word$
assumes *abc*: $a + of\text{-}nat\ b \leq c$
and *alb*: $a \leq a + of\text{-}nat\ b$
and *cld*: $c \leq c + of\text{-}nat\ d$
and *blt*: $b < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
and *dlt*: $d < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
shows $\{a..+b\} \cap \{c..+d\} = \{\}$
proof (*rule disjointI*, *rule notI*)
fix *x y*
assume *x*: $x \in \{a..+b\}$ **and** *y*: $y \in \{c..+d\}$ **and** *xy*: $x = y$

from *x* **obtain** *kx* **where** $x = a + of\text{-}nat\ kx$ **and** *kx*: $kx < b$
by (*clarsimp dest*!: *intvlD*)

moreover from *y* **obtain** *ky* **where** $y = c + of\text{-}nat\ ky$ **and** *ky*: $ky < d$
by (*clarsimp dest*!: *intvlD*)

ultimately have *ac*: $a + of\text{-}nat\ kx = c + of\text{-}nat\ ky$ **using** *xy* **by** *simp*

have $of\text{-}nat\ kx < (of\text{-}nat\ b :: 'a\ word)$ **using** *blt kx*
by (*rule of-nat-mono-maybe*)
hence $a + of\text{-}nat\ kx < a + of\text{-}nat\ b$ **using** *alb*
by (*rule word-plus-strict-mono-right*)

also have $\ldots \leq c$ **by** (*rule abc*)
also have $\ldots \leq c + of\text{-}nat\ ky$ **using** *cld dlt ky*
by − (*rule word-random* [*OF - iffD1* [*OF More-Word.of-nat-mono-maybe-le*]], *simp*+)
finally show *False* **using** *ac* **by** *simp*
qed

lemma *intvl-disjoint2*:
fixes $a :: 'a :: len\ word$
assumes *abc*: $a + of\text{-}nat\ b \leq c$
and *alb*: $a \leq a + of\text{-}nat\ b$
and *cld*: $c \leq c + of\text{-}nat\ d$
and *blt*: $b < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
and *dlt*: $d < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
shows $\{c..+d\} \cap \{a..+b\} = \{\}$
using *abc alb cld blt dlt*
by (*subst Int-commute*, *rule intvl-disjoint1*)

lemma *typ-slice-t-self*:
$td \in fst\ `\ set\ (typ\text{-}slice\text{-}t\ td\ m)$
apply (*cases td*)
apply (*simp split*: *if-split*)
done

lemma *index-fold-update*:

⟦ *distinct xs*; *set xs* ⊆ {..< *card* (*UNIV* :: 'b *set*)}; *n* < *card* (*UNIV* :: 'b *set*) ⟧
⟹
index (*foldr* (λ*n* (*arr* :: 'a['b :: *finite*]). *Arrays.update arr n* (*f n* (*index arr n*)))
xs v) *n*
= (*if n* ∈ *set xs then f n* (*index v n*) *else index v n*)
apply (*induct xs*)
apply *simp*
subgoal for *x xs* **by** (*cases x* = *n*, *auto*)
done

lemma *hrs-mem-update-cong*:
⟦ ⋀*x*. *f x* = *f' x* ⟧ ⟹ *hrs-mem-update f* = *hrs-mem-update f'*
by (*simp add*: *hrs-mem-update-def*)

lemma *Guard-no-cong*:
⟦ *A*=*A'*; *c*=*c'* ⟧ ⟹ *Guard A P c* = *Guard A' P c'*
by *simp*

lemma *coerce-heap-update-to-heap-updates*:
assumes *n*: *n* = *chunk* ∗ *m* **and** *len*: *length xs* = *n*
shows *heap-update-list x xs*
= (λ*s*. *foldl* (λ*s n*. *heap-update-list* (*x* + (*of-nat n* ∗ *of-nat chunk*))
(*take chunk* (*drop* (*n* ∗ *chunk*) *xs*)) *s*)
s [*0* ..< *m*])
using *len*[*simplified n*]
apply (*induct m arbitrary*: *x xs*)
apply (*rule ext*, *simp*)
apply (*rule ext*)
subgoal for *m x xs s*
apply (*simp only*: *upt-conv-Cons map-Suc-upt*[*symmetric*])
apply (*subgoal-tac* ∃ *ys zs*. *length ys* = *chunk* ∧ *xs* = *ys* @ *zs*)
apply (*clarsimp simp*: *heap-update-list-concat-unfold foldl-map*
field-simps)
apply (*rule exI*[**where** *x*=*take chunk xs*])
apply (*rule exI*[**where** *x*=*drop chunk xs*])
apply *simp*
done
done

lemma *update-ti-list-array'*:
⟦ *update-ti-list-t* (*map f* [*0* ..< *n*]) *xs v* = *y*;
∀ *n*. *size-td-tuple* (*f n*) = *v3*; *length xs* = *v3* ∗ *n*;
∀ *m xs v'*. *length xs* = *v3* ∧ *m* < *n* ⟶
update-ti-tuple-t (*f m*) *xs v'* = *Arrays.update v' m* (*update-ti-t* (*g m*) *xs* (*index*
v' m)) ⟧
⟹ *y* = *foldr* (λ*n arr*. *Arrays.update arr n* (*update-ti-t* (*g n*) (*take v3* (*drop*

(*v3* ∗ *n*) *xs*)) (*index arr n*))) [*0* ..< *n*] *v*
apply (*subgoal-tac* ∀ *ys*. *size-td-list* (*map f ys*) = *v3* ∗ *length ys*)
prefer *2*
apply (*rule allI*)
subgoal for *ys* **by** (*induct ys*) *auto*
apply (*induct n arbitrary*: *xs y v*)
apply *simp*
apply (*simp add*: *access-ti-append*)
apply (*elim meta-allE*, *drule*(*1*) *meta-mp*)
apply *simp*
apply (*rule foldr-cong*, (*rule refl*)+)
apply (*simp add*: *take-drop*)
apply (*subst min.absorb1*)
apply (*fold mult-Suc-right*, *rule mult-le-mono2*)
apply *simp*
apply *simp*
done

lemma *update-ti-list-array*:
⟦ *update-ti-list-t* (*map f* [*0* ..< *n*]) *xs v* = (*y* :: *'a*['*b* :: *finite*]);
∀ *n*. *size-td-tuple* (*f n*) = *v3*; *length xs* = *v3* ∗ *n*;
∀ *m xs v'*. *length xs* = *v3* ∧ *m* < *n* ⟶
update-ti-tuple-t (*f m*) *xs v'* = *Arrays.update v' m* (*update-ti-t* (*g m*) *xs* (*index v' m*));
n ≤ *card* (*UNIV* :: *'b set*) ⟧
⟹ ∀ *m* < *n*. *update-ti-t* (*g m*) (*take v3* (*drop* (*v3* ∗ *m*) *xs*)) (*index v m*) = *index y m*
apply (*subst update-ti-list-array'*[**where** *y*=*y*], *assumption*+)
apply *clarsimp*
apply (*subst index-fold-update*)
apply *clarsimp*+
done

lemma *access-in-array*:
fixes *y* :: (*'a* :: *c-type*)['*b* :: *finite*]
assumes *assms*: *h-val hp x* = *y*
n < *card* (*UNIV* :: *'b set*)
and *subst*: ∀ *xs v*. *length xs* = *size-of TYPE*(*'a*)
⟶ *update-ti-t* (*typ-info-t TYPE*(*'a*)) *xs v* = *f xs*
shows *h-val hp*
(*Ptr* (*ptr-val x* + *of-nat* (*n* ∗ *size-of TYPE*(*'a*)))) = *index y n*
using *assms*
apply (*simp add*: *h-val-def drop-heap-list-le2 del*: *of-nat-mult*)
apply (*subst take-heap-list-le*[*symmetric*, **where** *n*=*card* (*UNIV* :: *'b set*) ∗ *size-of TYPE* (*'a*)])
apply (*fold mult-Suc*, *rule mult-le-mono1*)
apply *simp*
apply (*simp add*: *from-bytes-def typ-info-array'*)
apply (*drule update-ti-list-array*, *simp*+)

apply (*simp add*: *size-of-def*)
apply (*clarsimp simp*: *update-ti-s-adjust-ti*)
apply (*rule refl*)
apply *simp*
apply (*drule spec*, *drule*(*1*) *mp*)
apply (*simp add*: *size-of-def ac-simps drop-take*)
apply (*subgoal-tac length v = size-of TYPE*('*a*) **for** *v*)
apply (*subst subst*, *assumption*)
apply (*subst*(*asm*) *subst*, *assumption*)
apply *simp*
apply (*simp add*: *size-of-def*)
apply (*subst le-diff-conv2*)
apply *simp*
apply (*fold mult-Suc*, *rule mult-le-mono1*)
apply *simp*
done

lemma *access-ti-list-array*:
⟦ ∀ *n*. *size-td-tuple* (*f n*) = *v3*; *length xs* = *v3* ∗ *n*;
∀ *m*. *m* < *n* ∧ *v3* ≤ *length* (*drop* (*v3* ∗ *m*) *xs*)
⟶ *access-ti-tuple* (*f m*) (*FCP g*) (*take v3* (*drop* (*v3* ∗ *m*) *xs*)) = (*h m*)
⟧ ⟹
access-ti-list (*map f* [*0* ..< *n*]) (*FCP g*) *xs*
= *foldl* (@) [] (*map h* [*0* ..< *n*])
apply (*subgoal-tac* ∀ *ys*. *size-td-list* (*map f ys*) = *v3* ∗ *length ys*)
prefer *2*
apply (*rule allI*)
subgoal for *ys* **by** (*induct ys*, *simp*+)
apply (*induct n arbitrary*: *xs*)
apply *simp*
apply (*simp add*: *access-ti-append*)
apply (*erule-tac x*=*take* (*v3* ∗ *n*) *xs* **in** *meta-allE*)
apply *simp*
apply (*frule spec*, *drule mp*, *rule conjI*, *rule lessI*)
apply *simp*
apply *simp*
apply (*erule meta-mp*)
apply (*auto simp add*: *drop-take*)
done

lemma *take-drop-foldl-concat*:
⟦ ⋀*y*. *y* < *m* ⟹ *length* (*f y*) = *n*; *x* < *m* ⟧
⟹ *take n* (*drop* (*x* ∗ *n*) (*foldl* (@) [] (*map f* [*0* ..< *m*]))) = *f x*
apply (*subst split-upt-on-n*, *assumption*)
apply (*simp only*: *foldl-concat-concat map-append*)
apply (*subst drop-append-miracle*)
apply (*induct x*, *simp-all*)[*1*]
apply *simp*
done

lemma *heap-update-Array*:
heap-update (*p* ::('*a*::*packed-type*['*b*::*finite*]) *ptr*) *arr*
= (λ*s*. *foldl* (λ*s n*. *heap-update* (*array-ptr-index p False n*)
(*Arrays.index arr n*) *s*) *s* [*0* ..< *card* (*UNIV* :: '*b set*)])
apply (*rule ext*, *simp add*: *heap-update-def*)
apply (*subst coerce-heap-update-to-heap-updates*
[*OF* - *refl*, **where** *chunk*=*size-of TYPE*('*a*) **and** *m*=*card* (*UNIV* :: '*b set*)])
apply (*simp*)
apply (*rule foldl-cong*[*OF refl refl*])
apply (*simp add*: *array-ptr-index-def CTypesDefs.ptr-add-def*)
subgoal for *s a x*
apply (*rule arg-cong*[**where** *f*=λ*xs*. *heap-update-list p xs s* **for** *p s*])
apply (*simp add*: *to-bytes-def size-of-def*
packed-type-access-ti)
apply (*simp add*: *typ-info-array'*)
apply (*subst fcp-eta*[*symmetric*], *subst access-ti-list-array*)
apply *simp*
apply *simp*
apply (*simp add*: *packed-type-access-ti size-of-def*)
apply *fastforce*
apply (*rule take-drop-foldl-concat*)
apply (*simp add*: *size-of-def*)
apply *simp*
done
done

lemma *from-bytes-Array-element*:
fixes *p* :: ('*a*::*mem-type*['*b*::*finite*]) *ptr*
assumes *less*: *of-nat n* < *card* (*UNIV* :: '*b set*)
assumes *len*: *length bs* = *size-of TYPE*('*a*) ∗ *CARD*('*b*)
shows
index (*from-bytes bs* :: '*a*['*b*]) *n*
= *from-bytes* (*take* (*size-of TYPE*('*a*)) (*drop* (*n* ∗ *size-of TYPE*('*a*)) *bs*))
using *less*
apply (*simp add*: *from-bytes-def size-of-def typ-info-array'*)
apply (*subst update-ti-list-array'*[*OF refl*])
apply *simp*
apply (*simp add*: *len size-of-def*)
apply (*clarsimp simp*: *update-ti-s-adjust-ti*)
apply (*rule refl*)
apply (*simp add*: *split-upt-on-n*[*OF less*])
apply (*rule trans*, *rule foldr-does-nothing-to-xf*[**where** *xf*=λ*s*. *index s n*])
apply *simp*+
apply (*subst foldr-does-nothing-to-xf*[**where** *xf*=λ*s*. *index s n*])
apply *simp*
apply (*simp add*: *mult.commute*)
apply (*frule Suc-leI*)

apply (*drule-tac k=size-of TYPE('a)* **in** *mult-le-mono2*)
apply (*rule upd-rf*)
apply (*simp add*: *size-of-def len mult.commute*)
done

lemma *heap-access-Array-element'*:
fixes *p* :: ('*a*::*mem-type*['*b*::*finite*]) *ptr*
assumes *less*: *of-nat n* < *card* (*UNIV* :: '*b set*)
shows
index (*h-val hp p*) *n*
= *h-val hp* (*array-ptr-index p False n*)
using *less*
apply (*simp add*: *array-ptr-index-def CTypesDefs.ptr-add-def h-val-def*)
apply (*simp add*: *from-bytes-Array-element*)
apply (*simp add*: *drop-heap-list-le take-heap-list-le*)
apply (*subst take-heap-list-le*)
apply (*simp add*: *le-diff-conv2*)
apply (*drule Suc-leI*)
apply (*drule-tac k=size-of TYPE('a)* **in** *mult-le-mono2*)
apply (*simp add*: *mult.commute*)
apply *simp*
done

lemmas *heap-access-Array-element*
= *heap-access-Array-element'*[*simplified array-ptr-index-simps*]

lemma *heap-update-id*:
h-val hp ptr = (*v* :: '*a* :: *packed-type*)
⟹ *heap-update ptr v hp* = *hp*
apply (*simp add*: *h-val-def heap-update-def*)
apply (*rule heap-update-list-id'*[**where** *n=size-of TYPE('a)*])
apply *clarsimp*
apply (*simp add*: *from-bytes-def to-bytes-def update-ti-t-def*
size-of-def field-access-update-same
td-fafu-idem)
done

lemma *heap-update-Array-update*:
assumes *n*: *n* < *CARD*('*b* :: *finite*)
assumes *size*: *CARD*('*b*) ∗ *size-of TYPE*('*a* :: *packed-type*) < 2 ^ *addr-bitsize*
shows *heap-update p* (*Arrays.update* (*arr* :: '*a*['*b*]) *n v*) *hp*
= *heap-update* (*array-ptr-index p False n*) *v* (*heap-update p arr hp*)
proof −

have *P*: ⋀*x k*. ⟦ *x* < *CARD*('*b*); *k* < *size-of TYPE*('*a*) ⟧
⟹ *unat* (*of-nat x* ∗ *of-nat* (*size-of TYPE*('*a*)) + (*of-nat k* :: *addr*))
= *x* ∗ *size-of TYPE*('*a*) + *k*

```
    using size
    supply unsigned-of-nat
    apply (cases size-of TYPE('a), simp-all)
    apply (cases CARD('b), simp-all)
    apply (subst unat-add-lem[THEN iffD1])
     apply (simp add: unat-word-ariths unat-of-nat less-Suc-eq-le)
     apply (subgoal-tac Suc x * size-of TYPE('a) < 2 ^ addr-bitsize, simp-all)
     apply (erule order-le-less-trans[rotated], simp add: add-mono)
    apply (subst unat-mult-lem[THEN iffD1])
     apply (simp add: unat-of-nat unat-add-lem[THEN iffD1])
     apply (rule order-less-le-trans, erule order-le-less-trans[rotated],
           rule add-mono, simp+)
      apply (simp add: less-Suc-eq-le trans-le-add2)
     apply simp
    apply (simp add: unat-of-nat unat-add-lem[THEN iffD1])
    done

  let ?key-upd = heap-update (array-ptr-index p False n) v
  note commute = fold-commute-apply[where h=?key-upd
      and xs=[Suc n ..< CARD('b)], where g=f' and f=f' for f']

  show ?thesis using n
    apply (simp add: heap-update-Array split-upt-on-n[OF n]
                     foldl-conv-fold)
    apply (subst commute)
     apply (simp-all add: packed-heap-update-collapse
                    cong: fold-cong')
    apply (rule ext, simp)
    subgoal for x
      apply (rule heap-update-commute, simp-all add: ptr-add-def)
     apply (simp add: array-ptr-index-def CTypesDefs.ptr-add-def intvl-def Suc-le-eq)
      apply (rule set-eqI, clarsimp)
      apply (drule word-unat.Rep-inject[THEN iffD2])
      apply (clarsimp simp: P nat-eq-add-iff1)
      apply (cases x, simp-all add: less-Suc-eq-le Suc-diff-le)
      done
    done
qed

lemma heap-update-id-Array:
  fixes arr :: ('a :: packed-type)['b :: finite]
  shows arr = h-val hp p
    ⟹ heap-update p arr hp = hp
  apply (simp add: heap-update-Array)
  apply (rule foldl-does-nothing[where s=hp])
  apply (simp add: heap-access-Array-element' heap-update-id)
  done

lemma heap-update-Array-element'':
```

fixes p' :: $(('a$ :: *packed-type*$)['b$::*finite*$])$ *ptr*
fixes p :: $('a$ :: *packed-type*$)$ *ptr*
fixes *hp w*
assumes *p*: $p = $ *array-ptr-index* p' *False n*
assumes *n*: $n < CARD('b)$
assumes *size*: $CARD('b) * $ *size-of* $TYPE('a) < 2 \hat{} $ *addr-bitsize*
shows *heap-update* p' (*Arrays.update* (*h-val hp* p') *n w*) *hp*
$=$ *heap-update p w hp*
apply (*subst heap-update-Array-update*[*OF n size*])
apply (*simp add*: *heap-update-id-Array p*)
done

lemmas *heap-update-Array-element'*
$=$ *heap-update-Array-element''*[*simplified array-ptr-index-simps*]

lemma *array-count-size*:
$CARD('b$ :: *array-max-count*$) * $ *size-of* $TYPE('a$ :: *array-outer-max-size*$) < 2 \hat{}$ *addr-bitsize*
using *array-outer-max-size-ax*[**where** $'a='a$] *array-max-count-ax*[**where** $'a='b$]
apply (*clarsimp dest*!: *nat-le-Suc-less-imp*)
apply (*drule*(*1*) *mult-mono*, *simp*+)
done

lemmas *heap-update-Array-element*
$=$ *heap-update-Array-element'*[*OF refl - array-count-size*]

lemma *typ-slice-list-cut*:
$[\![(\forall x \in$ *set xs*. *size-td* (*dt-fst x*) $= m); m \neq 0; n < ($*length xs* $* m)]\!]$
$\Longrightarrow$ *typ-slice-list xs n* $=$
typ-slice-tuple (*xs* ! (*n div m*)) (*n mod m*)
apply (*induct xs arbitrary*: *n*, *simp-all*)
subgoal for *x1 xs n*
apply (*intro conjI impI*)
apply *simp*
apply (*subgoal-tac* $\exists n'.$ $n = n' + m$)
apply *clarsimp*
apply (*metis* (*no-types*, *opaque-lifting*) *add.commute add.left-neutral cancel-comm-monoid-add-class.diff-cancel*
div-add1-eq div-mult-self-is-m mod-div-trivial modulo-nat-def mult.left-neutral nth-Cons-Suc plus-1-eq-Suc)
apply (*rule exI*[**where** $x=n - m$])
apply *simp*
done
done

lemma *typ-slice-t-array*:
$[\![n < CARD('b); y <$ *size-of* $TYPE('a)]\!]$
$\Longrightarrow$ *typ-slice-t* (*export-uinfo* (*typ-info-t* $TYPE('a))) y \leq$
typ-slice-t (*export-uinfo* (*array-tag* $TYPE('a['b ::$ *finite*$]))))$

```
            (y + size-of TYPE('a :: mem-type) * n)
  apply (simp add: array-tag-def array-tag-n-eq
              split del: if-split)
  apply (rule disjI2)
  apply (subgoal-tac y + (size-of TYPE('a) * n) < CARD('b) * size-of TYPE('a))
   apply (simp add: typ-slice-list-cut[where m=size-of TYPE('a)]
                    map-td-list-map o-def size-of-def
                    sz-nzero[unfolded size-of-def])
   apply (simp flip: export-uinfo-def)
  apply (rule order-less-le-trans[where y=Suc n * size-of TYPE('a)])
   apply (simp add: size-of-def)
  apply (simp only: size-of-def mult-le-mono1)
  done

lemma h-t-valid-Array-element':
  ⟦ htd ⊨t (p :: (('a :: mem-type)['b :: finite]) ptr); coerce ∨ n < CARD('b) ⟧
    ⟹ htd ⊨t array-ptr-index p coerce n for coerce
  apply (clarsimp simp only: h-t-valid-def valid-footprint-def Let-def
      c-guard-def c-null-guard-def)
  apply (subgoal-tac ∃ offs. array-ptr-index p coerce n = CTypesDefs.ptr-add (ptr-coerce
p) (of-nat offs)
        ∧ offs < CARD ('b))
   apply (clarsimp simp: size-td-array size-of-def typ-uinfo-t-def
      typ-info-array array-tag-def)
  subgoal for offs
    apply (intro conjI)
     apply (clarsimp simp: CTypesDefs.ptr-add-def
       field-simps)
     apply (rename-tac y)
     apply (drule-tac x=offs * size-of TYPE('a) + y in spec)
     apply (drule mp)
      apply (rule order-less-le-trans[where y=Suc offs * size-of TYPE('a)])
       apply (simp add: size-of-def)
      apply (simp only: size-of-def mult-le-mono1)
     apply (clarsimp simp: field-simps)
     apply (erule map-le-trans[rotated])
     apply (rule list-map-mono)
     apply (subst mult.commute, rule typ-slice-t-array[unfolded array-tag-def])
      apply assumption
     apply (simp add: size-of-def)
    apply (simp add: ptr-aligned-def align-of-def align-td-array
       array-ptr-index-def
       CTypesDefs.ptr-add-def unat-word-ariths unat-of-nat)
   using align-size-of[where 'a='a] align[where 'a='a]
    apply (simp add: align-of-def size-of-def addr-card-def card-word)
    apply (simp add: dvd-mod)
   apply (thin-tac ∀ x. P x for P)
   apply (clarsimp simp: intvl-def)
   apply (rename-tac k)
```

apply (*drule-tac x=offs * size-of TYPE('a) + k* **in** *spec*)
apply (*drule mp*)
apply (*simp add*: *array-ptr-index-def CTypesDefs.ptr-add-def field-simps*)
apply (*erule notE*)
apply (*rule order-less-le-trans*[**where** *y=Suc offs * size-of TYPE('a)*])
apply (*simp add*: *size-of-def*)
apply (*simp only*: *size-of-def mult-le-mono1*)
done
subgoal by (*auto simp add*: *array-ptr-index-def intro*: *exI*[**where** *x=0*])
done

lemma *h-t-valid-Array-element*:
⟦ *htd* ⊨$_t$ (*p* :: (('*a* :: *mem-type*)['*b* :: *finite*]) *ptr*); *0* ≤ *n*; *n* < *int CARD('b)* ⟧
⟹ *htd* ⊨$_t$ ((*ptr-coerce p* :: '*a ptr*) +$_p$ *n*)
apply (*drule-tac n=nat n* **and** *coerce=False* **in** *h-t-valid-Array-element'*)
apply *simp*
apply (*simp add*: *array-ptr-index-def*)
done

lemma *ptr-safe-Array-element*:
⟦ *ptr-safe* (*p* :: (('*a* :: *mem-type*)['*b* :: *finite*]) *ptr*) *htd*; *coerce* ∨ *n* < *CARD('b)* ⟧
⟹ *ptr-safe* (*array-ptr-index p coerce n*) *htd* **for** *coerce*
apply (*simp add*: *ptr-safe-def*)
apply (*erule order-trans*[*rotated*])
apply (*subgoal-tac* ∃ *offs*. *array-ptr-index p coerce n = CTypesDefs.ptr-add* (*ptr-coerce p*) (*of-nat offs*)
∧ *offs* < *CARD* ('*b*))
prefer *2*
subgoal by (*auto simp*: *array-ptr-index-def intro*: *exI*[**where** *x=0*])[*1*]
apply (*clarsimp simp*: *s-footprint-def s-footprint-untyped-def*
CTypesDefs.ptr-add-def
size-td-array size-of-def)
subgoal for *offs x k*
apply (*rule exI*[**where** *x=offs * size-of TYPE('a) + x*])
apply (*simp add*: *size-of-def*)
apply (*rule conjI*)
apply (*rule order-less-le-trans*[**where** *y=Suc offs * size-of TYPE('a)*])
apply (*simp add*: *size-of-def*)
apply (*simp only*: *size-of-def*)
apply (*rule mult-le-mono1*)
apply *simp*
apply (*thin-tac coerce* ∨ *P* **for** *P*)
apply (*elim disjE exE conjE*, *simp-all add*: *typ-uinfo-t-def*)
apply (*erule order-less-le-trans*)
apply (*rule prefix-length-le*)
apply (*rule order-trans*, *erule typ-slice-t-array*)
apply (*simp add*: *size-of-def*)
apply (*simp add*: *size-of-def field-simps typ-info-array*)
done

done

lemma *from-bytes-eq*:
from-bytes [*x*] = *x*
apply (*clarsimp simp*:*from-bytes-def update-ti-t-def typ-info-word*)
apply (*simp add*:*word-rcat-def*)
done

lemma *bytes-disjoint*:(*x*::('*a*::*c-type*) *ptr*) ≠ *y* ⟹ {*ptr-val x* + *a* ..+ *1*} ∩ {*ptr-val y* + *a* ..+ *1*} = {}
by (*clarsimp simp*:*intvl-def*)

lemma *byte-ptrs-disjoint*:(*x*::('*a*::*c-type*) *ptr*) ≠ *y* ⟹ ∀ *i* < *of-nat* (*size-of TYPE*('*a*)). *ptr-val x* + *i* ≠ *ptr-val y* + *i*
by *force*

lemma *le-step*:⟦(*x*::('*a*::*len*) *word*) < *y* + *1*; *x* ≠ *y*⟧ ⟹ *x* < *y*
by (*metis less-x-plus-1 max-word-max order-less-le*)

lemma *ptr-add-disjoint*:
⟦ *ptr-val y* ∉ {*ptr-val x* ..+ *size-of TYPE*('*a*)};
ptr-val (*x*::('*a*::*c-type*) *ptr*) < *ptr-val* (*y*::('*b*::*c-type*) *ptr*);
a < *of-nat* (*size-of TYPE*('*a*)) ⟧ ⟹
ptr-val x + *a* < *ptr-val y*
apply (*erule swap*)
apply (*rule intvl-inter-le* [**where** *k*=*0* **and** *ka*=*unat* (*ptr-val y* − *ptr-val x*)])
apply *clarsimp*
apply (*metis* (*opaque-lifting*, *mono-tags*) *add-diff-cancel2 add-diff-inverse diff-add-cancel trans-less-add1 unat-less-helper word-le-less-eq word-less-add-right word-of-nat-less word-unat.Rep-inverse*)
apply *simp*
done

lemma *ptr-add-disjoint2*:
⟦ *ptr-val x* ∉ {*ptr-val y* ..+ *size-of TYPE*('*a*)};
ptr-val (*y*::('*b*::*c-type*) *ptr*) < *ptr-val* (*x*::('*a*::*c-type*) *ptr*);
a < *of-nat* (*size-of TYPE*('*a*)) ⟧ ⟹
ptr-val y + *a* < *ptr-val x*
apply (*erule swap*)
apply (*rule intvl-inter-le*[**where** *k*=*0* **and** *ka*=*unat* (*ptr-val x* − *ptr-val y*)])
apply *clarsimp*
apply (*metis* (*no-types*, *opaque-lifting*) *add.commute less-imp-le less-le-trans not-le unat-less-helper word-diff-ls'*(*4*))
apply *simp*
done

lemma *ptr-aligned-is-aligned*:⟦*ptr-aligned* (*x*::('*a*::*c-type*) *ptr*); *align-of TYPE*('*a*) = *2* ^ *n*⟧ ⟹ *is-aligned* (*ptr-val x*) *n*

by (*clarsimp simp*: *ptr-aligned-def is-aligned-def*)

lemma *intvl-no-overflow*:
assumes *no-overflow*: *unat a + b < 2 ^ len-of TYPE('a::len)*
shows (*x ∈ {(a :: 'a word) ..+ b}*) = (*a ≤ x ∧ x < (a + of-nat b)*))
proof −
obtain *sk :: 'a word ⇒ 'a word ⇒ nat ⇒ nat*
where *f1*: *⋀x y z. x ∉ {y..+z} ∨ x = y + of-nat (sk x y z) ∧ sk x y z < z*
using [[*metis-new-skolem*]] **by** (*metis intvlD*)

have *f2*: *⋀x. a + x < a + of-nat b ∨ ¬ x < of-nat b*
using *no-overflow*
by (*metis PackedTypes.of-nat-mono-maybe-le add-lessD1 le-add1*
add.commute olen-add-eqv unat-of-nat-eq word-arith-nat-add
word-plus-strict-mono-right)

have *f3*: *∀x y. y ∉ {x..+b} ∨ of-nat (sk y x b) < (of-nat b :: 'a word)*
using *no-overflow f1*
by (*metis add-lessD1 add.commute of-nat-mono-maybe*)

have *x < a + of-nat b ∨ ¬ of-nat (sk x a b) < (of-nat b :: 'a word) ∨ ?thesis*
using *f1 f2* **by** *metis*

hence *x < a + of-nat b ∨ ?thesis*
using *f3* **by** *metis*

thus *?thesis*
apply (*rule disjE*)
apply (*rule iffI*)
apply (*clarsimp simp*: *intvl-def*)
apply (*rename-tac k*)
apply (*clarsimp simp*: *unat-sub-if-size word-le-nat-alt word-less-nat-alt*)
apply (*cut-tac no-overflow*)
apply (*subgoal-tac k + (b + unat a) < 2 ^ len-of (TYPE('a)) + b*)
apply (*subgoal-tac k + unat a < 2 ^ len-of (TYPE('a))*)
apply (*metis add-lessD1 le-def less-not-refl2 add.commute unat-eq-of-nat*
word-arith-nat-add)
apply *clarsimp*
apply *clarsimp*
apply (*clarsimp simp*: *intvl-def*)
apply (*rule exI* [**where** *x=unat (x − a)*])
apply (*clarsimp simp*: *unat-sub-if-size word-le-nat-alt word-less-nat-alt*)
apply (*cut-tac no-overflow*)
apply (*metis diff-le-self le-add-diff-inverse le-diff-conv le-eq-less-or-eq le-unat-uoi*
add.commute nat-neq-iff unat-of-nat-eq word-arith-nat-add)
apply *simp*
done
qed

lemma *FCP-arg-cong*:$f = g \Longrightarrow FCP\ f = FCP\ g$
by *simp*

lemma *h-val-id*:
$h\text{-}val\ (hrs\text{-}mem\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ x\ y)\ s))\ x = (y::'a::mem\text{-}type)$
apply (*subst hrs-mem-update*)
apply (*rule h-val-heap-update*)
done

lemma *h-val-id-padding*:
$length\ bs = size\text{-}of\ TYPE('a) \Longrightarrow h\text{-}val\ (hrs\text{-}mem\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\text{-}padding\ x\ y\ bs)\ s))\ x = (y::'a::mem\text{-}type)$
apply (*subst hrs-mem-update*)
apply (*simp add*: *h-val-heap-update-padding*)
done

lemma *heap-update-id2*:
$hrs\text{-}mem\text{-}update\ (heap\text{-}update\ p\ ((h\text{-}val\ (hrs\text{-}mem\ s)\ p)::'a::packed\text{-}type))\ s = s$
apply (*clarsimp simp*:*hrs-mem-update-def case-prod-beta*)
apply (*subst heap-update-id*)
apply (*simp add*:*hrs-mem-def*)+
done

lemma *intvlI-unat*:$unat\ b < unat\ c \Longrightarrow a + b \in \{a\ ..+\ unat\ c\}$
by (*metis intvlI word-unat.Rep-inverse*)

lemma *neq-imp-bytes-disjoint*:
$[\![\ c\text{-}guard\ (x::'a::c\text{-}type\ ptr);\ c\text{-}guard\ y;\ unat\ j < align\text{-}of\ TYPE('a);$
$unat\ i < align\text{-}of\ TYPE('a);\ x \neq y;\ 2\ \hat{}\ n = align\text{-}of\ TYPE('a);\ n < 32\ ]\!] \Longrightarrow$
$ptr\text{-}val\ x + j \neq ptr\text{-}val\ y + i$
apply (*rule ccontr*)
apply (*subgoal-tac is-aligned* (*ptr-val x*) *n*)
apply (*subgoal-tac is-aligned* (*ptr-val y*) *n*)
apply (*subgoal-tac* (*ptr-val x* + *j* &&& ~~ *mask n*) = (*ptr-val y* + *i* &&& ~~ *mask n*))
apply (*subst* (*asm*) *neg-mask-add-aligned*, *simp*, *simp add*: *word-less-nat-alt*)
apply (*subst* (*asm*) *neg-mask-add-aligned*, *simp*, *simp add*: *word-less-nat-alt*)
apply *clarsimp*
apply *simp*
apply (*clarsimp simp*: *c-guard-def ptr-aligned-def is-aligned-def*)
apply (*clarsimp simp*: *c-guard-def ptr-aligned-def is-aligned-def*)
done

lemma *heap-update-list-base'*:*heap-update-list p* [] = *id*

by (*rule ext*, *simp*)

lemma *hrs-mem-update-id3*: *hrs-mem-update id = id*
unfolding *hrs-mem-update-def* **by** *simp*

lemma *h-t-valid-ptr-retyp-inside-eq*:
fixes *p* :: *'a* :: *mem-type ptr* **and** *p'* :: *'a* :: *mem-type ptr*
assumes *inside*: *ptr-val p' ∈ {ptr-val p ..+ size-of TYPE('a)}*
and *ht*: *ptr-retyp p td*, *g ⊨ₜ p'*
shows *p = p'*
using *ptr-retyp-same-cleared-region*[*OF ht*] *inside mem-type-self*[**where** *p=p'*]
by *blast*

lemma *typ-slice-t-self-nth*:
∃ n < length (*typ-slice-t td m*). *∃ b. typ-slice-t td m ! n = (td*, *b)*
using *typ-slice-t-self* [**where** *td = td* **and** *m = m*]
by (*fastforce simp add*: *in-set-conv-nth*)

lemma *ptr-retyp-other-cleared-region*:
fixes *p* :: *'a* :: *mem-type ptr* **and** *p'* :: *'b* :: *mem-type ptr*
assumes *ht*: *ptr-retyp p td*, *g ⊨ₜ p'*
and *tdisj*: *typ-uinfo-t TYPE('a) ⊥ₜ typ-uinfo-t TYPE('b :: mem-type)*
and *clear*: *∀ x ∈ {ptr-val p ..+ size-of TYPE('a)}. ∀ n b. snd* (*td x*) *n ≠ Some* (*typ-uinfo-t TYPE('b)*, *b*)
shows *{ptr-val p'..+ size-of TYPE('b)} ∩ {ptr-val p ..+ size-of TYPE('a)} = {}*
proof (*rule classical*)
assume *asm*: *{ptr-val p'..+ size-of TYPE('b)} ∩ {ptr-val p ..+ size-of TYPE('a)} ≠ {}*
then obtain *mv* **where** *mvp*: *mv ∈ {ptr-val p..+size-of TYPE('a)}*
and *mvp'*: *mv ∈ {ptr-val p'..+size-of TYPE('b)}*
by *blast*

then obtain *k'* **where** *mv*: *mv = ptr-val p' + of-nat k'* **and** *klt*: *k' < size-td* (*typ-info-t TYPE('b)*)
by (*clarsimp dest!*: *intvlD simp*: *size-of-def*)

let *?mv = ptr-val p' + of-nat k'*

obtain *n b* **where** *nl*: *n < length* (*typ-slice-t* (*typ-uinfo-t TYPE('b)*) *k'*)
and *tseq*: *typ-slice-t* (*typ-uinfo-t TYPE('b)*) *k' ! n = (typ-uinfo-t TYPE('b)*, *b)*
using *typ-slice-t-self-nth* [**where** *td = typ-uinfo-t TYPE('b)* **and** *m = k'*]
by *clarsimp*

with *ht* **have** *snd* (*ptr-retyp p td ?mv*) *n = Some* (*typ-uinfo-t TYPE('b)*, *b*)
unfolding *h-t-valid-def*
apply −
apply (*clarsimp simp*: *valid-footprint-def Let-def*)
apply (*drule spec*, *drule mp* [*OF - klt*])
apply (*clarsimp simp*: *map-le-def*)

apply (*drule bspec*)
apply *simp*
apply *simp*
done

moreover {
assume *snd* (*ptr-retyp p empty-htd ?mv*) *n* = *Some* (*typ-uinfo-t TYPE*('*b*), *b*)
hence (*typ-uinfo-t TYPE*('*b*)) ∈ *fst* ‘ *set* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*))
(*unat* (*ptr-val p'* + *of-nat k'* − *ptr-val p*)))
using *asm mv mvp*
apply −
apply (*rule image-eqI*[**where** *x* = (*typ-uinfo-t TYPE*('*b*), *b*)])
apply *simp*
apply (*fastforce simp add*: *ptr-retyp-footprint list-map-eq in-set-conv-nth split*: *if-split-asm*)
done

with *typ-slice-set* **have** (*typ-uinfo-t TYPE*('*b*)) ∈ *fst* ‘ *td-set* (*typ-uinfo-t TYPE*('*a*)) *0*
by (*rule subsetD*)

hence *False* **using** *tdisj* **by** (*clarsimp simp*: *tag-disj-def typ-tag-le-def*)
} ultimately show *?thesis* **using** *mvp mvp' mv* **unfolding** *h-t-valid-def valid-footprint-def*
apply −
apply (*subst* (*asm*) *ptr-retyp-d-eq-snd*)
apply (*auto simp add*: *map-add-Some-iff clear*)
done
qed

end

theory *Cong-Tactic*
imports
Main
HOL−Eisbach.Eisbach-Tools

begin

Simple congruence prover
Replaces a goal of the shape:
f x (*g y*) (λ*x*. *x* + *1*) = *f x'* (*i y'*) (λ*z*. *z* + *Suc 0*)
by
x = *x' g y* = *i y' 1* = *Suc 0*
The tactic essentially applies ⟦*?f* = *?g*; *?x* = *?y*⟧ ⟹ *?f ?x* = *?g ?y*, but using first-order matching.

ML ‹

fun fo-cong-tac ctxt = *CSUBGOAL* (*fn* (*cgoal*, *i*) =>

```
  let
    val goal = Thm.term-of cgoal;
  in
    (case Logic.strip-assums-concl goal of
      - $ (- $ t $ s) =>
        let
          val (f, xs) = strip-comb t
          val (g, ys) = strip-comb s
        in
          if f aconv g andalso length xs = length ys
          then REPEAT-DETERM-N (length xs) (cong-tac ctxt i)
               THEN resolve-tac ctxt [@{thm refl}] i
          else no-tac
        end
    | - => no-tac)
  end);


fun abs-cong-tac ctxt = CSUBGOAL (fn (cgoal, i) =>
  let
    val goal = Thm.term-of cgoal;
  in
    (case Logic.strip-assums-concl goal of
      - $ (- $ Abs - $ Abs -) => resolve-tac ctxt [@{thm ext}] i
    | - => no-tac)
  end);

›

method-setup app-cong =
  ‹Scan.succeed (fn ctxt => Method.SIMPLE-METHOD' (fo-cong-tac ctxt))›
  congruence method

method-setup abs-cong =
  ‹Scan.succeed (fn ctxt => Method.SIMPLE-METHOD' (abs-cong-tac ctxt))›
  congruence method

method cong-step = (app-cong | abs-cong | rule refl | rule eq-reflection)
method cong = (cong-step ; cong?)
```

Preserve context information from [*cong*] rules

```
ML ‹
fun const-cong ctxt = CSUBGOAL (fn (cgoal, i) =>
  let
    val goal = Thm.term-of cgoal;
  in
    (case Logic.strip-assums-concl goal of
      - $ (- $ t $ s) => (case (head-of t, head-of s) of
        (Const (c, -), Const (c', -)) =>
```

```
          if c = c'
          then
           case AList.lookup (op =) (Simplifier.dest-congs (simpset-of ctxt)) (true,
c) of
              SOME thm =>
                resolve-tac ctxt [Thm.transfer' ctxt thm RS meta-eq-to-obj-eq] i
            | NONE => no-tac
         else no-tac
       | - => no-tac
     )
   | - => no-tac)
  end);
›

method-setup const-cong =
  ‹Scan.succeed (fn ctxt => Method.SIMPLE-METHOD' (const-cong ctxt))›
  congruence method (constant with [cong])

method cong-context-step = (const-cong | cong-step | rule simp-impliesI)
method cong-context = (cong-context-step ; cong-context?)

end
```

Part III

AutoCorres

Chapter 16

Spec-Monad

theory *Spec-Monad*
imports
Basic-Runs-To-VCG
HOL−Library.Complete-Partial-Order2
HOL−Library.Monad-Syntax
AutoCorres-Utils
begin

16.1 *rel-map* and *rel-project*

definition *rel-map* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ **where**
rel-map f r x = $(x = f\ r)$

lemma *rel-map-direct*[*simp*]: *rel-map f a* (*f a*)
by (*simp add*: *rel-map-def*)

abbreviation *rel-project* ≡ *rel-map*

lemmas *rel-project-def* = *rel-map-def*

lemma *rel-project-id*: *rel-project id* = (=)
rel-project (λ*v*. *v*) = (=)
by (*auto simp add*: *rel-project-def fun-eq-iff*)

lemma *rel-project-unit*: *rel-project* (λ-. ()) *x y* = *True*
by (*simp add*: *rel-project-def*)

lemma *rel-projectI*: *y* = *prj x* ⟹ *rel-project prj x y*
by (*simp add*: *rel-project-def*)

lemma *rel-project-conv*: *rel-project prj x y* = (*y* = *prj x*)
by (*simp add*: *rel-project-def*)

16.2 Misc Theorems

declare *case-unit-Unity* [*simp*] — without this rule simplifier seems loops in unexpected ways

lemma *abs-const-unit*: $(\lambda(v::unit).\ f) = (\lambda().\ f)$
  by *auto*

lemma *SUP-mono''*: $(\bigwedge x.\ x{\in}A \Longrightarrow f\ x \leq g\ x) \Longrightarrow (\bigsqcup x{\in}A.\ f\ x) \leq (\bigsqcup x{\in}A.\ g\ x ::$ *-::complete-lattice*)
  by (*rule SUP-mono*) *auto*

lemma *wf-nat-bound*: *wf* $\{(a,\ b).\ b < a \wedge b \leq (n::nat)\}$
  apply (*rule wf-subset*)
  apply (*rule wf-measure*[**where** $f{=}\lambda a.\ Suc\ n - a$])
  apply *auto*
  done

lemma (**in** *complete-lattice*) *admissible-le*:
  ccpo.admissible Inf $(\leq)$ $(\lambda x.\ (y \leq x))$
  by (*simp add*: *ccpo.admissibleI local.Inf-greatest*)

lemma *mono-const*: *mono* $(\lambda x.\ c)$
  by (*simp add*: *mono-def*)

lemma *mono-lam*: $(\bigwedge a.\ mono\ (\lambda x.\ F\ x\ a)) \Longrightarrow mono\ F$
  by (*simp add*: *mono-def le-fun-def*)

lemma *mono-app*: *mono* $(\lambda x.\ x\ a)$
  by (*simp add*: *mono-def le-fun-def*)

lemma *all-cong-map*:
  assumes *f-f'*: $\bigwedge y.\ f\ (f'\ y) = y$ **and** *P-Q*: $\bigwedge x.\ P\ x \longleftrightarrow Q\ (f\ x)$
  shows $(\forall x.\ P\ x) \longleftrightarrow (\forall y.\ Q\ y)$
  using *assms* **by** *metis*

lemma *rel-set-refl*: $(\bigwedge x.\ x \in A \Longrightarrow R\ x\ x) \Longrightarrow rel\text{-}set\ R\ A\ A$
  by (*auto simp*: *rel-set-def*)

lemma *rel-set-converse-iff*: *rel-set R X Y* $\longleftrightarrow$ *rel-set* R^{-1-1} *Y X*
  by (*auto simp add*: *rel-set-def*)

lemma *rel-set-weaken*:
  $(\bigwedge x\ y.\ x \in A \Longrightarrow y \in B \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y) \Longrightarrow rel\text{-}set\ P\ A\ B \Longrightarrow rel\text{-}set\ Q\ A\ B$
  by (*force simp*: *rel-set-def*)

lemma *sim-set-refl*: $(\bigwedge x.\ x \in X \Longrightarrow R\ x\ x) \Longrightarrow sim\text{-}set\ R\ X\ X$
  by (*auto simp*: *sim-set-def*)

16.3 Galois Connections

lemma *mono-of-Sup-cont*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
assumes *cont*: $\bigwedge X.\ f\ (Sup\ X) = (SUP\ x{\in}X.\ f\ x)$
assumes *xy*: $x \leq y$
shows $f\ x \leq f\ y$
proof −
have $f\ x \leq sup\ (f\ x)\ (f\ y)$ **by** (*rule sup-ge1*)
also have $\ldots = (SUP\ x{\in}\{x,\ y\}.\ f\ x)$ **by** *simp*
also have $\ldots = f\ (Sup\ \{x,\ y\})$ **by** (*rule cont*[*symmetric*])
finally show *?thesis* **using** *xy* **by** (*simp add*: *sup-absorb2*)
qed

lemma *gc-of-Sup-cont*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
assumes *cont*: $\bigwedge X.\ f\ (Sup\ X) = (SUP\ x{\in}X.\ f\ x)$
shows $f\ x \leq y \longleftrightarrow x \leq Sup\ \{x.\ f\ x \leq y\}$
proof *safe*
assume $f\ x \leq y$ **then show** $x \leq Sup\ \{x.\ f\ x \leq y\}$
by (*intro Sup-upper*) *simp*
next
assume $x \leq Sup\ \{x.\ f\ x \leq y\}$
then have $f\ x \leq f\ (Sup\ \{x.\ f\ x \leq y\})$
by (*rule mono-of-Sup-cont*[*OF cont*])
also have $\ldots = (SUP\ x{\in}\{x.\ f\ x \leq y\}.\ f\ x)$ **by** (*rule cont*)
also have $\ldots \leq y$ **by** (*rule Sup-least*) *auto*
finally show $f\ x \leq y$ **.**
qed

lemma *mono-of-Inf-cont*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
assumes *cont*: $\bigwedge X.\ f\ (Inf\ X) = (INF\ x{\in}X.\ f\ x)$
assumes *xy*: $x \leq y$
shows $f\ x \leq f\ y$
proof −
have $f\ x = f\ (Inf\ \{x,\ y\})$ **using** *xy* **by** (*simp add*: *inf-absorb1*)
also have $\ldots = (INF\ x{\in}\{x,\ y\}.\ f\ x)$ **by** (*rule cont*)
also have $\ldots = inf\ (f\ x)\ (f\ y)$ **by** *simp*
also have $\ldots \leq f\ y$ **by** (*rule inf-le2*)
finally show *?thesis* **.**
qed

lemma *gc-of-Inf-cont*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice$
assumes *cont*: $\bigwedge X.\ f\ (Inf\ X) = (INF\ x{\in}X.\ f\ x)$
shows $Inf\ \{y.\ x \leq f\ y\} \leq y \longleftrightarrow x \leq f\ y$
proof *safe*
assume $x \leq f\ y$ **then show** $Inf\ \{y.\ x \leq f\ y\} \leq y$

```
    by (intro Inf-lower) simp
next
  assume ∗: Inf {y. x ≤ f y} ≤ y
  have x ≤ (INF y∈{y. x ≤ f y}. f y) by (rule Inf-greatest) auto
  also have ... = f (Inf {y. x ≤ f y}) by (rule cont[symmetric])
  also have ... ≤ f y
    using ∗ by (rule mono-of-Inf-cont[OF cont])
  finally show x ≤ f y .
qed

lemma gfp-fusion:
  assumes f-g: ⋀x y. g x ≤ y ⟷ x ≤ f y
  assumes a: mono a
  assumes b: mono b
  assumes ∗: ⋀x. f (a x) = b (f x)
  shows f (gfp a) = gfp b
  apply (intro antisym gfp-upperbound)
  subgoal
    apply (subst ∗[symmetric])
    apply (subst gfp-fixpoint[OF a])
    ..
  apply (rule f-g[THEN iffD1])
  apply (rule gfp-upperbound)
  apply (rule f-g[THEN iffD2])
  apply (subst ∗)
  apply (subst gfp-fixpoint[OF b, symmetric])
  apply (rule monoD[OF b])
  apply (rule f-g[THEN iffD1])
  apply (rule order-refl)
  done

lemma lfp-fusion:
  assumes f-g: ⋀x y. g x ≤ y ⟷ x ≤ f y
  assumes a: mono a
  assumes b: mono b
  assumes ∗: ⋀x. g (a x) = b (g x)
  shows g (lfp a) = lfp b
  apply (intro antisym lfp-lowerbound)
  subgoal
    apply (rule f-g[THEN iffD2])
    apply (rule lfp-lowerbound)
    apply (rule f-g[THEN iffD1])
    apply (subst ∗)
    apply (subst (2) lfp-fixpoint[OF b, symmetric])
    apply (rule monoD[OF b])
    apply (rule f-g[THEN iffD2])
    apply (rule order-refl)
    done
  subgoal
```

apply (*subst* ∗[*symmetric*])
apply (*subst lfp-fixpoint*[*OF a*])
..
done

16.4 *post-state* type

datatype *'r post-state* = *Failure* | *Success 'r set*

Failure is supposed to model things like undefined behaviour in C. We usually have to show the absence of *Failure* for all possible executions of the program. Moreover, it is used to model the 'result' of a non terminating computation.

lemma *split-post-state*:
x = (*case x of Success X* ⇒ *Success X* | *Failure* ⇒ *Failure*)
for *x*::*'a post-state*
by (*cases x*) *auto*

instantiation *post-state* :: (*type*) *order*
begin

inductive *less-eq-post-state* :: *'a post-state* ⇒ *'a post-state* ⇒ *bool* **where**
Failure-le[*simp*, *intro*]: *less-eq-post-state p Failure*
| *Success-le-Success*[*intro*]: *r* ⊆ *q* ⟹ *less-eq-post-state* (*Success r*) (*Success q*)

definition *less-post-state* :: *'a post-state* ⇒ *'a post-state* ⇒ *bool* **where**
less-post-state p q ⟷ *p* ≤ *q* ∧ ¬ *q* ≤ *p*

instance
proof
fix *p q r* :: *'a post-state*
show *p* ≤ *p* **by** (*cases p*) *auto*
show *p* ≤ *q* ⟹ *q* ≤ *r* ⟹ *p* ≤ *r* **by** (*blast elim*: *less-eq-post-state.cases*)
show *p* ≤ *q* ⟹ *q* ≤ *p* ⟹ *p* = *q* **by** (*blast elim*: *less-eq-post-state.cases*)
qed (*fact less-post-state-def*)
end

lemma *Success-le-Success-iff*[*simp*]: *Success r* ≤ *Success q* ⟷ *r* ⊆ *q*
by (*auto elim*: *less-eq-post-state.cases*)

lemma *Failure-le-iff*[*simp*]: *Failure* ≤ *q* ⟷ *q* = *Failure*
by (*auto elim*: *less-eq-post-state.cases*)

instantiation *post-state* :: (*type*) *complete-lattice*
begin

definition *top-post-state* :: *'a post-state* **where**
top-post-state = *Failure*

definition *bot-post-state* :: *'a post-state* **where**
bot-post-state = *Success* {}

definition *inf-post-state* :: *'a post-state* ⇒ *'a post-state* ⇒ *'a post-state* **where**
inf-post-state =
(λ*Failure* ⇒ *id* | *Success res1* ⇒
(λ*Failure* ⇒ *Success res1* | *Success res2* ⇒ *Success* (*res1* ∩ *res2*)))

definition *sup-post-state* :: *'a post-state* ⇒ *'a post-state* ⇒ *'a post-state* **where**
sup-post-state =
(λ*Failure* ⇒ (λ-. *Failure*) | *Success res1* ⇒
(λ*Failure* ⇒ *Failure* | *Success res2* ⇒ *Success* (*res1* ∪ *res2*)))

definition *Inf-post-state* :: *'a post-state set* ⇒ *'a post-state* **where**
Inf-post-state s = (*if Success* −‘ *s* = {} *then Failure else Success* (⋂ (*Success* −‘ *s*)))

definition *Sup-post-state* :: *'a post-state set* ⇒ *'a post-state* **where**
Sup-post-state s = (*if Failure* ∈ *s then Failure else Success* (⋃ (*Success* −‘ *s*)))

instance
proof
fix *x y z* :: *'a post-state* **and** *A* :: *'a post-state set*
show *inf x y* ≤ *y* **by** (*simp add*: *inf-post-state-def split*: *post-state.split*)
show *inf x y* ≤ *x* **by** (*simp add*: *inf-post-state-def split*: *post-state.split*)
show *x* ≤ *y* ⟹ *x* ≤ *z* ⟹ *x* ≤ *inf y z*
by (*auto simp add*: *inf-post-state-def elim*!: *less-eq-post-state.cases*)
show *x* ≤ *sup x y* **by** (*simp add*: *sup-post-state-def split*: *post-state.split*)
show *y* ≤ *sup x y* **by** (*simp add*: *sup-post-state-def split*: *post-state.split*)
show *x* ≤ *y* ⟹ *z* ≤ *y* ⟹ *sup x z* ≤ *y*
by (*auto simp add*: *sup-post-state-def elim*!: *less-eq-post-state.cases*)
show *x* ∈ *A* ⟹ *Inf A* ≤ *x* **by** (*cases x*) (*auto simp add*: *Inf-post-state-def*)
show (⋀*x*. *x* ∈ *A* ⟹ *z* ≤ *x*) ⟹ *z* ≤ *Inf A* **by** (*cases z*) (*force simp add*: *Inf-post-state-def*)+
show *x* ∈ *A* ⟹ *x* ≤ *Sup A* **by** (*cases x*) (*auto simp add*: *Sup-post-state-def*)
show (⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *z*) ⟹ *Sup A* ≤ *z* **by** (*cases z*) (*force simp add*: *Sup-post-state-def*)+
qed (*simp-all add*: *top-post-state-def Inf-post-state-def bot-post-state-def Sup-post-state-def*)
end

primrec *holds-post-state* :: (*'a* ⇒ *bool*) ⇒ *'a post-state* ⇒ *bool* **where**
holds-post-state P Failure ⟷ *False*
| *holds-post-state P* (*Success X*) ⟷ (∀ *x*∈*X*. *P x*)

primrec *holds-partial-post-state* :: (*'a* ⇒ *bool*) ⇒ *'a post-state* ⇒ *bool* **where**
holds-partial-post-state P Failure ⟷ *True*
| *holds-partial-post-state P* (*Success X*) ⟷ (∀ *x*∈*X*. *P x*)

inductive
  sim-post-state :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a post-state* ⇒ *'b post-state* ⇒ *bool*
  for *R*
where
  [*simp*]: ⋀*p*. *sim-post-state R p Failure*
| ⋀*A B*. *sim-set R A B* ⟹ *sim-post-state R* (*Success A*) (*Success B*)

inductive
  rel-post-state :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a post-state* ⇒ *'b post-state* ⇒ *bool*
  for *R*
where
  [*simp*]: *rel-post-state R Failure Failure*
| ⋀*A B*. *rel-set R A B* ⟹ *rel-post-state R* (*Success A*) (*Success B*)

primrec *lift-post-state* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'b post-state* ⇒ *'a post-state* **where**
  lift-post-state R Failure = *Failure*
| *lift-post-state R* (*Success Y*) = *Success* {*x*. ∃ *y*∈*Y*. *R x y*}

primrec *unlift-post-state* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a post-state* ⇒ *'b post-state* **where**
  unlift-post-state R Failure = *Failure*
| *unlift-post-state R* (*Success X*) = *Success* {*y*. ∀ *x*. *R x y* ⟶ *x* ∈ *X*}

primrec *map-post-state* :: (*'a* ⇒ *'b*) ⇒ *'a post-state* ⇒ *'b post-state* **where**
  map-post-state f Failure = *Failure*
| *map-post-state f* (*Success r*) = *Success* (*f* ‘ *r*)

primrec *vmap-post-state* :: (*'a* ⇒ *'b*) ⇒ *'b post-state* ⇒ *'a post-state* **where**
  vmap-post-state f Failure = *Failure*
| *vmap-post-state f* (*Success r*) = *Success* (*f* −‘ *r*)

definition *pure-post-state* :: *'a* ⇒ *'a post-state* **where**
  pure-post-state v = *Success* {*v*}

primrec *bind-post-state* :: *'a post-state* ⇒ (*'a* ⇒ *'b post-state*) ⇒ *'b post-state*
where
  bind-post-state Failure p = *Failure*
| *bind-post-state* (*Success res*) *p* = ⋃ (*p* ‘ *res*)

16.4.1 Order Properties

lemma *top-ne-bot*[*simp*]: (⊥:: - *post-state*) ≠ ⊤
  and *bot-ne-top*[*simp*]: (⊤:: - *post-state*) ≠ ⊥
  by (*auto simp*: *top-post-state-def bot-post-state-def*)

lemma *Success-ne-top*[*simp*]:
  Success X ≠ ⊤ ⊤ ≠ *Success X*
  by (*auto simp*: *top-post-state-def*)

lemma *Success-eq-bot-iff*[*simp*]:

Success X = ⊥ ⟷ *X* = {} ⊥ = *Success X* ⟷ *X* = {}
by (*auto simp*: *bot-post-state-def*)

lemma *Sup-Success*: (⨆ *x*∈*X*. *Success* (*f x*)) = *Success* (⋃ *x*∈*X*. *f x*)
by (*auto simp*: *Sup-post-state-def*)

lemma *Sup-Success-pair*: (⨆ (*x*, *y*)∈*X*. *Success* (*f x y*)) = *Success* (⋃ (*x*, *y*)∈*X*. *f x y*)
by (*simp add*: *split-beta′ Sup-Success*)

16.4.2 *holds-post-state*

lemma *holds-post-state-iff*:
holds-post-state P p ⟷ (∃ *X*. *p* = *Success X* ∧ (∀ *x*∈*X*. *P x*))
by (*cases p*) *auto*

lemma *holds-post-state-weaken*:
holds-post-state P p ⟹ (⋀*x*. *P x* ⟹ *Q x*) ⟹ *holds-post-state Q p*
by (*cases p*; *auto*)

lemma *holds-post-state-combine*:
holds-post-state P p ⟹ *holds-post-state Q p* ⟹
(⋀*x*. *P x* ⟹ *Q x* ⟹ *R x*) ⟹ *holds-post-state R p*
by (*cases p*; *auto*)

lemma *holds-post-state-Ball*:
Y ≠ {} ∨ *p* ≠ *Failure* ⟹
holds-post-state (λ*x*. ∀ *y*∈*Y*. *P x y*) *p* ⟷ (∀ *y*∈*Y*. *holds-post-state* (λ*x*. *P x y*) *p*)
by (*cases p*; *auto*)

lemma *holds-post-state-All*:
holds-post-state (λ*x*. ∀ *y*. *P x y*) *p* ⟷ (∀ *y*. *holds-post-state* (λ*x*. *P x y*) *p*)
by (*cases p*; *auto*)

lemma *holds-post-state-BexI*:
y ∈ *Y* ⟹ *holds-post-state* (λ*x*. *P x y*) *p* ⟹ *holds-post-state* (λ*x*. ∃ *y*∈*Y*. *P x y*) *p*
by (*cases p*; *auto*)

lemma *holds-post-state-conj*:
holds-post-state (λ*x*. *P x* ∧ *Q x*) *p* ⟷ *holds-post-state P p* ∧ *holds-post-state Q p*
by (*cases p*; *auto*)

lemma *sim-post-state-iff*:
sim-post-state R p q ⟷
(∀ *P*. *holds-post-state* (λ*y*. ∀ *x*. *R x y* ⟶ *P x*) *q* ⟶ *holds-post-state P p*)
apply (*cases p*; *cases q*; *simp add*: *sim-set-def sim-post-state.simps*)

apply *safe*
apply *force*
apply *force*
subgoal premises *prems* **for** *A B*
using *prems*(*3*)[*THEN spec*, *of* $\lambda a.\ \exists\, b{\in}B.\ R\ a\ b$] *prems*(*4*)
by *auto*
done

lemma *post-state-le-iff*:
$p \leq q \longleftrightarrow (\forall\, P.\ \textit{holds-post-state}\ P\ q \longrightarrow \textit{holds-post-state}\ P\ p)$
by (*cases p*; *cases q*; *force simp add*: *subset-eq Ball-def*)

lemma *post-state-eq-iff*:
$p = q \longleftrightarrow (\forall\, P.\ \textit{holds-post-state}\ P\ p \longleftrightarrow \textit{holds-post-state}\ P\ q)$
by (*simp add*: *order-eq-iff post-state-le-iff*)

lemma *holds-top-post-state*[*simp*]: $\neg\ \textit{holds-post-state}\ P\ \top$
by (*simp add*: *top-post-state-def*)

lemma *holds-bot-post-state*[*simp*]: *holds-post-state P* $\bot$
by (*simp add*: *bot-post-state-def*)

lemma *holds-Sup-post-state*[*simp*]: *holds-post-state P* (*Sup F*) $\longleftrightarrow$ ($\forall\, f{\in}F.$ *holds-post-state P f*)
by (*subst* (*2*) *split-post-state*)
(*auto simp*: *Sup-post-state-def split*: *post-state.splits*)

lemma *holds-post-state-gfp*:
holds-post-state P (*gfp f*) $\longleftrightarrow$ ($\forall\, p.\ p \leq f\ p \longrightarrow$ *holds-post-state P p*)
by (*simp add*: *gfp-def*)

lemma *holds-post-state-gfp-apply*:
holds-post-state P (*gfp f x*) $\longleftrightarrow$ ($\forall\, p.\ p \leq f\ p \longrightarrow$ *holds-post-state P* (*p x*))
by (*simp add*: *gfp-def*)

lemma *holds-lift-post-state*[*simp*]:
holds-post-state P (*lift-post-state R x*) $\longleftrightarrow$ *holds-post-state* ($\lambda y.\ \forall\, x.\ R\ x\ y \longrightarrow P\ x$) *x*
by (*cases x*) *auto*

lemma *holds-map-post-state*[*simp*]:
holds-post-state P (*map-post-state f x*) $\longleftrightarrow$ *holds-post-state* ($\lambda x.\ P\ (f\ x)$) *x*
by (*cases x*) *auto*

lemma *holds-vmap-post-state*[*simp*]:
holds-post-state P (*vmap-post-state f x*) $\longleftrightarrow$ *holds-post-state* ($\lambda x.\ \forall\, y.\ f\ y = x \longrightarrow P\ y$) *x*
by (*cases x*) *auto*

lemma *holds-pure-post-state*[*simp*]: *holds-post-state P* (*pure-post-state x*) ⟷ *P x*
by (*simp add*: *pure-post-state-def*)

lemma *holds-bind-post-state*[*simp*]:
holds-post-state P (*bind-post-state f g*) ⟷ *holds-post-state* (λ*x*. *holds-post-state P* (*g x*)) *f*
by (*cases f*) *auto*

lemma *holds-post-state-False*: *holds-post-state* (λ*x*. *False*) *f* ⟷ *f* = ⊥
by (*cases f*) (*auto simp*: *bot-post-state-def*)

16.4.3 *holds-post-state-partial*

lemma *holds-partial-post-state-of-holds*:
holds-post-state P p ⟹ *holds-partial-post-state P p*
by (*cases p*; *simp*)

lemma *holds-partial-post-state-iff*:
holds-partial-post-state P p ⟷ (∀ *X*. *p* = *Success X* ⟶ (∀ *x*∈*X*. *P x*))
by (*cases p*) *auto*

lemma *holds-partial-post-state-True*[*simp*]: *holds-partial-post-state* (λ*x*. *True*) *p*
by (*cases p*; *simp*)

lemma *holds-partial-post-state-weaken*:
holds-partial-post-state P p ⟹ (⋀*x*. *P x* ⟹ *Q x*) ⟹ *holds-partial-post-state Q p*
by (*cases p*; *auto*)

lemma *holds-partial-post-state-Ball*:
holds-partial-post-state (λ*x*. ∀ *y*∈*Y*. *P x y*) *p* ⟷ (∀ *y*∈*Y*. *holds-partial-post-state* (λ*x*. *P x y*) *p*)
by (*cases p*; *auto*)

lemma *holds-partial-post-state-All*:
holds-partial-post-state (λ*x*. ∀ *y*. *P x y*) *p* ⟷ (∀ *y*. *holds-partial-post-state* (λ*x*. *P x y*) *p*)
by (*cases p*; *auto*)

lemma *holds-partial-post-state-conj*:
holds-partial-post-state (λ*x*. *P x* ∧ *Q x*) *p* ⟷
holds-partial-post-state P p ∧ *holds-partial-post-state Q p*
by (*cases p*; *auto*)

lemma *holds-partial-top-post-state*[*simp*]: *holds-partial-post-state P* ⊤
by (*simp add*: *top-post-state-def*)

lemma *holds-partial-bot-post-state*[*simp*]: *holds-partial-post-state P* ⊥
by (*simp add*: *bot-post-state-def*)

lemma *holds-partial-pure-post-state*[*simp*]: *holds-partial-post-state P* (*pure-post-state x*) ⟷ *P x*
by (*simp add*: *pure-post-state-def*)

lemma *holds-partial-Sup-post-stateI*:
(⋀*x*. *x* ∈ *X* ⟹ *holds-partial-post-state P x*) ⟹ *holds-partial-post-state P* (*Sup X*)
by (*force simp*: *Sup-post-state-def*)

lemma *holds-partial-bind-post-state*:
holds-partial-post-state (λ*x*. *holds-partial-post-state P* (*g x*)) *f* ⟹
holds-partial-post-state P (*bind-post-state f g*)
by (*cases f*) (*auto intro*: *holds-partial-Sup-post-stateI*)

lemma *holds-partial-map-post-state*[*simp*]:
holds-partial-post-state P (*map-post-state f x*) ⟷ *holds-partial-post-state* (λ*x*. *P* (*f x*)) *x*
by (*cases x*) *auto*

lemma *holds-partial-vmap-post-state*[*simp*]:
holds-partial-post-state P (*vmap-post-state f x*) ⟷
holds-partial-post-state (λ*x*. ∀ *y*. *f y* = *x* ⟶ *P y*) *x*
by (*cases x*) *auto*

lemma *holds-partial-post-state-rel*:
rel-post-state (*R* :: *'a* ⇒ *'b* ⇒ *bool*) *p q* ⟹
holds-partial-post-state (λ*b*. ∀ *a*. *R a b* ⟶ *P a*) *q* ⟹
holds-partial-post-state P p
by (*force elim*!: *rel-post-state.cases simp*: *rel-set-def*)

16.4.4 *sim-post-state*

lemma *sim-post-state-eq-iff-le*: *sim-post-state* (=) *p q* ⟷ *p* ≤ *q*
by (*simp add*: *post-state-le-iff sim-post-state-iff*)

lemma *sim-post-state-Success-Success-iff*[*simp*]:
sim-post-state R (*Success r*) (*Success q*) ⟷ (∀ *a*∈*r*. ∃ *b*∈*q*. *R a b*)
by (*auto simp add*: *sim-set-def elim*: *sim-post-state.cases intro*: *sim-post-state.intros*)

lemma *sim-post-state-Success2*:
sim-post-state R f (*Success q*) ⟷ *holds-post-state* (λ*a*. ∃ *b*∈*q*. *R a b*) *f*
by (*cases f*; *simp add*: *sim-post-state.simps sim-set-def*)

lemma *sim-post-state-Failure1*[*simp*]: *sim-post-state R Failure q* ⟷ *q* = *Failure*
by (*auto elim*: *sim-post-state.cases intro*: *sim-post-state.intros*)

lemma *sim-post-state-top2*[*simp*, *intro*]: *sim-post-state R p* ⊤
by (*simp add*: *top-post-state-def*)

lemma *sim-post-state-top1*[*simp*]: *sim-post-state R* ⊤ *q* ⟷ *q* = ⊤
using *sim-post-state-Failure1* **by** (*metis top-post-state-def*)

lemma *sim-post-state-bot2*[*simp*, *intro*]: *sim-post-state R p* ⊥ ⟷ *p* = ⊥
by (*cases p*; *simp add*: *bot-post-state-def*)

lemma *sim-post-state-bot1*[*simp*, *intro*]: *sim-post-state R* ⊥ *q*
by (*cases q*; *simp add*: *bot-post-state-def*)

lemma *sim-post-state-le1*: *sim-post-state R f' g* ⟹ *f* ≤ *f'* ⟹ *sim-post-state R f g*
by (*simp add*: *post-state-le-iff sim-post-state-iff*)

lemma *sim-post-state-le2*: *sim-post-state R f g* ⟹ *g* ≤ *g'* ⟹ *sim-post-state R f g'*
by (*simp add*: *post-state-le-iff sim-post-state-iff*)

lemma *sim-post-state-Sup1*:
sim-post-state R (*Sup A*) *f* ⟷ (∀ *a*∈*A*. *sim-post-state R a f*)
by (*auto simp add*: *sim-post-state-iff*)

lemma *sim-post-state-Sup2*:
a ∈ *A* ⟹ *sim-post-state R f a* ⟹ *sim-post-state R f* (*Sup A*)
by (*auto simp add*: *sim-post-state-iff*)

lemma *sim-post-state-Sup*:
∀ *a*∈*A*. ∃ *b*∈*B*. *sim-post-state R a b* ⟹ *sim-post-state R* (*Sup A*) (*Sup B*)
by (*auto simp*: *sim-post-state-Sup1 intro*: *sim-post-state-Sup2*)

lemma *sim-post-state-weaken*:
sim-post-state R f g ⟹ (⋀*x y*. *R x y* ⟹ *Q x y*) ⟹ *sim-post-state Q f g*
by (*cases f*; *cases g*; *force*)

lemma *sim-post-state-trans*:
sim-post-state R f g ⟹ *sim-post-state Q g h* ⟹ (⋀*x y z*. *R x y* ⟹ *Q y z* ⟹ *S x z*) ⟹
sim-post-state S f h
by (*fastforce elim*!: *sim-post-state.cases intro*: *sim-post-state.intros simp*: *sim-set-def*)

lemma *sim-post-state-refl'*: *holds-partial-post-state* (λ*x*. *R x x*) *f* ⟹ *sim-post-state R f f*
by (*cases f*; *auto simp*: *rel-set-def*)

lemma *sim-post-state-refl*: (⋀*x*. *R x x*) ⟹ *sim-post-state R f f*
by (*simp add*: *sim-post-state-refl'*)

lemma *sim-post-state-pure-post-state2*:
sim-post-state F f (*pure-post-state x*) ⟷ *holds-post-state* (λ*y*. *F y x*) *f*

by (*cases f*; *simp add*: *pure-post-state-def*)

16.4.5 *rel-post-state*

lemma *rel-post-state-top*[*simp*, *intro*!]: *rel-post-state R* ⊤ ⊤
by (*auto simp*: *top-post-state-def*)

lemma *rel-post-state-top-iff*[*simp*]:
rel-post-state R ⊤ *p* ⟷ *p* = ⊤
rel-post-state R p ⊤ ⟷ *p* = ⊤
by (*auto simp*: *top-post-state-def rel-post-state.simps*)

lemma *rel-post-state-Success-iff*[*simp*]:
rel-post-state R (*Success A*) (*Success B*) ⟷ *rel-set R A B*
by (*auto elim*: *rel-post-state.cases intro*: *rel-post-state.intros*)

lemma *rel-post-state-bot*[*simp*, *intro*!]: *rel-post-state R* ⊥ ⊥
by (*auto simp*: *bot-post-state-def Lifting-Set.empty-transfer*)

lemma *rel-post-state-eq-sim-post-state*:
rel-post-state R p q ⟷ *sim-post-state R p q* ∧ *sim-post-state* R^{-1-1} *q p*
by (*auto simp*: *rel-set-def sim-set-def elim*!: *sim-post-state.cases rel-post-state.cases*)

lemma *rel-post-state-weaken*:
rel-post-state R f g ⟹ (⋀*x y*. *R x y* ⟹ *Q x y*) ⟹ *rel-post-state Q f g*
by (*auto intro*: *sim-post-state-weaken simp*: *rel-post-state-eq-sim-post-state*)

lemma *rel-post-state-eq*[*relator-eq*]: *rel-post-state* (=) = (=)
by (*simp add*: *rel-post-state-eq-sim-post-state fun-eq-iff sim-post-state-eq-iff-le order-eq-iff*)

lemma *rel-post-state-mono*[*relator-mono*]:
A ≤ *B* ⟹ *rel-post-state A* ≤ *rel-post-state B*
using *rel-set-mono*[*of A B*]
by (*auto simp add*: *le-fun-def intro*!: *rel-post-state.intros elim*!: *rel-post-state.cases*)

lemma *rel-post-state-refl'*: *holds-partial-post-state* (λ*x*. *R x x*) *f* ⟹ *rel-post-state R f f*
by (*cases f*; *auto simp*: *rel-set-def*)

lemma *rel-post-state-refl*: (⋀*x*. *R x x*) ⟹ *rel-post-state R f f*
by (*simp add*: *rel-post-state-refl'*)

16.4.6 *lift-post-state*

lemma *lift-post-state-top*: *lift-post-state R* ⊤ = ⊤
by (*auto simp*: *top-post-state-def*)

lemma *lift-post-state-unlift-post-state*: *lift-post-state R p* ≤ *q* ⟷ *p* ≤ *unlift-post-state R q*

by (*cases p*; *cases q*; *auto simp add*: *subset-eq*)

lemma *lift-post-state-eq-Sup*: *lift-post-state R p* = *Sup* {*q*. *sim-post-state R q p*}
unfolding *post-state-eq-iff holds-Sup-post-state*
using *holds-lift-post-state sim-post-state-iff*
by *blast*

lemma *le-lift-post-state-iff*: *q* ≤ *lift-post-state R p* ⟷ *sim-post-state R q p*
by (*simp add*: *post-state-le-iff sim-post-state-iff*)

lemma *lift-post-state-eq*[*simp*]: *lift-post-state* (=) *p* = *p*
by (*simp add*: *post-state-eq-iff*)

lemma *lift-post-state-comp*:
lift-post-state R (*lift-post-state Q p*) = *lift-post-state* (*R OO Q*) *p*
by (*cases p*) *auto*

lemma *sim-post-state-lift*:
sim-post-state Q q (*lift-post-state R p*) ⟷ *sim-post-state* (*Q OO R*) *q p*
unfolding *le-lift-post-state-iff*[*symmetric*] *lift-post-state-comp* **..**

lemma *lift-post-state-Sup*: *lift-post-state R* (*Sup F*) = (*SUP f*∈*F*. *lift-post-state R f*)
by (*simp add*: *post-state-eq-iff*)

lemma *lift-post-state-mono*: *p* ≤ *q* ⟹ *lift-post-state R p* ≤ *lift-post-state R q*
by (*simp add*: *post-state-le-iff*)

16.4.7 *map-post-state*

lemma *map-post-state-top*: *map-post-state f* ⊤ = ⊤
by (*auto simp*: *top-post-state-def*)

lemma *mono-map-post-state*: *s1* ≤ *s2* ⟹ *map-post-state f s1* ≤ *map-post-state f s2*
by (*simp add*: *post-state-le-iff*)

lemma *map-post-state-eq-lift-post-state*: *map-post-state f p* = *lift-post-state* (λ*a b*. *a* = *f b*) *p*
by (*cases p*) *auto*

lemma *map-post-state-Sup*: *map-post-state f* (*Sup X*) = (*SUP x*∈*X*. *map-post-state f x*)
by (*simp add*: *post-state-eq-iff*)

lemma *map-post-state-comp*: *map-post-state f* (*map-post-state g p*) = *map-post-state* (*f* ∘ *g*) *p*
by (*simp add*: *post-state-eq-iff*)

lemma *map-post-state-id*[*simp*]: *map-post-state id p* = *p*
by (*simp add*: *post-state-eq-iff*)

lemma *map-post-state-pure*[*simp*]: *map-post-state f* (*pure-post-state x*) = *pure-post-state* (*f x*)
by (*simp add*: *pure-post-state-def*)

lemma *sim-post-state-map-post-state1*:
sim-post-state R (*map-post-state f p*) *q* ⟷ *sim-post-state* (λ*x y*. *R* (*f x*) *y*) *p q*
by (*cases p*; *cases q*; *simp*)

lemma *sim-post-state-map-post-state2*:
sim-post-state R p (*map-post-state f q*) ⟷ *sim-post-state* (λ*x y*. *R x* (*f y*)) *p q*
unfolding *map-post-state-eq-lift-post-state sim-post-state-lift* **by** (*simp add*: *OO-def*[*abs-def*])

lemma *map-post-state-eq-top*[*simp*]: *map-post-state f p* = ⊤ ⟷ *p* = ⊤
by (*cases p*; *simp add*: *top-post-state-def*)

lemma *map-post-state-eq-bot*[*simp*]: *map-post-state f p* = ⊥ ⟷ *p* = ⊥
by (*cases p*; *simp add*: *bot-post-state-def*)

16.4.8 *vmap-post-state*

lemma *vmap-post-state-top*: *vmap-post-state f* ⊤ = ⊤
by (*auto simp*: *top-post-state-def*)

lemma *vmap-post-state-Sup*:
vmap-post-state f (*Sup X*) = (*SUP x*∈*X*. *vmap-post-state f x*)
by (*simp add*: *post-state-eq-iff*)

lemma *vmap-post-state-le-iff*: (*vmap-post-state f p* ≤ *q*) = (*p* ≤ ⨆ {*p*. *vmap-post-state f p* ≤ *q*})
using *vmap-post-state-Sup* **by** (*rule gc-of-Sup-cont*)

lemma *vmap-post-state-eq-lift-post-state*: *vmap-post-state f p* = *lift-post-state* (λ*a b*. *f a* = *b*) *p*
by (*cases p*) *auto*

lemma *vmap-post-state-comp*: *vmap-post-state f* (*vmap-post-state g p*) = *vmap-post-state* (*g* ∘ *f*) *p*
apply (*simp add*: *post-state-eq-iff*)
apply (*intro allI arg-cong*[**where** *f*=λ*P*. *holds-post-state P p*])
apply *auto*
done

lemma *vmap-post-state-id*: *vmap-post-state id p* = *p*
by (*simp add*: *post-state-eq-iff*)

lemma *sim-post-state-vmap-post-state2*:

sim-post-state R p (vmap-post-state f q) $\longleftrightarrow$
sim-post-state ($\lambda x\ y.\ \exists y'.\ f\ y' = y \wedge R\ x\ y'$) *p q*
by (*cases p*; *cases q*; *auto*) *blast+*

lemma *vmap-post-state-le-iff-le-map-post-state*:
map-post-state f p $\leq$ *q* $\longleftrightarrow$ *p* $\leq$ *vmap-post-state f q*
by (*simp flip*: *sim-post-state-eq-iff-le*
add: *sim-post-state-map-post-state1 sim-post-state-vmap-post-state2*)

16.4.9 *pure-post-state*

lemma *pure-post-state-Failure*[*simp*]: *pure-post-state v* $\neq$ *Failure*
by (*simp add*: *pure-post-state-def*)

lemma *pure-post-state-top*[*simp*]: *pure-post-state v* $\neq \top$
by (*simp add*: *pure-post-state-def top-post-state-def*)

lemma *pure-post-state-bot*[*simp*]: *pure-post-state v* $\neq \bot$
by (*simp add*: *pure-post-state-def bot-post-state-def*)

lemma *pure-post-state-inj*[*simp*]: *pure-post-state v* = *pure-post-state w* $\longleftrightarrow$ *v* = *w*
by (*simp add*: *pure-post-state-def*)

lemma *sim-pure-post-state-iff*[*simp*]:
sim-post-state R (*pure-post-state a*) (*pure-post-state b*) $\longleftrightarrow$ *R a b*
by (*simp add*: *pure-post-state-def*)

lemma *rel-pure-post-state-iff*[*simp*]:
rel-post-state R (*pure-post-state a*) (*pure-post-state b*) $\longleftrightarrow$ *R a b*
by (*simp add*: *rel-post-state-eq-sim-post-state*)

lemma *pure-post-state-le*[*simp*]: *pure-post-state v* $\leq$ *pure-post-state w* $\longleftrightarrow$ *v* = *w*
by (*simp add*: *pure-post-state-def*)

lemma *Success-eq-pure-post-state*[*simp*]: *Success X* = *pure-post-state x* $\longleftrightarrow$ *X* = $\{x\}$
by (*auto simp add*: *pure-post-state-def*)

lemma *pure-post-state-eq-Success*[*simp*]: *pure-post-state x* = *Success X* $\longleftrightarrow$ *X* = $\{x\}$
by (*auto simp add*: *pure-post-state-def*)

lemma *pure-post-state-le-Success*[*simp*]: *pure-post-state x* $\leq$ *Success X* $\longleftrightarrow$ *x* $\in$ *X*
by (*auto simp add*: *pure-post-state-def*)

lemma *Success-le-pure-post-state*[*simp*]: *Success X* $\leq$ *pure-post-state x* $\longleftrightarrow$ *X* $\subseteq$ $\{x\}$
by (*auto simp add*: *pure-post-state-def*)

lemma *case-pure-post-state*[*simp*]:
(*case pure-post-state x of Failure* ⇒ *f* | *Success X* ⇒ *S X*) = *S* {*x*}
by (*simp add*: *pure-post-state-def*)

lemma *sim-post-state-Success-pure-post-state*[*simp*]:
sim-post-state R (*Success X*) (*pure-post-state y*) ⟷ (∀ *x*∈*X*. *R x y*)
by (*simp add*: *pure-post-state-def*)

16.4.10 *bind-post-state*

lemma *bind-post-state-top*: *bind-post-state* ⊤ *g* = ⊤
by (*auto simp*: *top-post-state-def*)

lemma *bind-post-state-Sup1*[*simp*]:
bind-post-state (*Sup F*) *g* = (*SUP f*∈*F*. *bind-post-state f g*)
by (*simp add*: *post-state-eq-iff*)

lemma *bind-post-state-Sup2*[*simp*]:
G ≠ {} ∨ *f* ≠ *Failure* ⟹ *bind-post-state f* (*Sup G*) = (*SUP g*∈*G*. *bind-post-state f g*)
by (*simp add*: *post-state-eq-iff holds-post-state-Ball*)

lemma *bind-post-state-sup1*[*simp*]:
bind-post-state (*sup f1 f2*) *g* = *sup* (*bind-post-state f1 g*) (*bind-post-state f2 g*)
using *bind-post-state-Sup1*[*of* {*f1*, *f2*} *g*] **by** *simp*

lemma *bind-post-state-sup2*[*simp*]:
bind-post-state f (*sup g1 g2*) = *sup* (*bind-post-state f g1*) (*bind-post-state f g2*)
using *bind-post-state-Sup2*[*of* {*g1*, *g2*} *f*] **by** *simp*

lemma *bind-post-state-top1*[*simp*]: *bind-post-state* ⊤ *f* = ⊤
by (*simp add*: *post-state-eq-iff*)

lemma *bind-post-state-bot1*[*simp*]: *bind-post-state* ⊥ *f* = ⊥
by (*simp add*: *post-state-eq-iff*)

lemma *bind-post-state-eq-top*:
bind-post-state f g = ⊤ ⟷ ¬ *holds-post-state* (λ*x*. *g x* ≠ ⊤) *f*
by (*cases f*) (*force simp add*: *Sup-post-state-def simp flip*: *top-post-state-def*)+

lemma *bind-post-state-eq-bot*:
bind-post-state f g = ⊥ ⟷ *holds-post-state* (λ*x*. *g x* = ⊥) *f*
by (*cases f*) (*auto simp flip*: *top-post-state-def*)

lemma *lift-post-state-bind-post-state*:
lift-post-state R (*bind-post-state x g*) = *bind-post-state x* (λ*v*. *lift-post-state R* (*g v*))
by (*simp add*: *post-state-eq-iff*)

lemma *vmap-post-state-bind-post-state*:
vmap-post-state f (*bind-post-state p g*) = *bind-post-state p* (λ*v*. *vmap-post-state f* (*g v*))
by (*simp add*: *post-state-eq-iff*)

lemma *map-post-state-bind-post-state*:
map-post-state f (*bind-post-state x g*) = *bind-post-state x* (λ*v*. *map-post-state f* (*g v*))
by (*simp add*: *post-state-eq-iff*)

lemma *bind-post-state-pure-post-state1*[*simp*]:
bind-post-state (*pure-post-state v*) *f* = *f v*
by (*simp add*: *post-state-eq-iff*)

lemma *bind-post-state-pure-post-state2*:
bind-post-state p (λ*x*. *pure-post-state* (*f x*)) = *map-post-state f p*
by (*simp add*: *post-state-eq-iff*)

lemma *bind-post-state-map-post-state*:
bind-post-state (*map-post-state f p*) *g* = *bind-post-state p* (λ*x*. *g* (*f x*))
by (*simp add*: *post-state-eq-iff*)

lemma *bind-post-state-assoc*[*simp*]:
bind-post-state (*bind-post-state f g*) *h* =
bind-post-state f (λ*v*. *bind-post-state* (*g v*) *h*)
by (*simp add*: *post-state-eq-iff*)

lemma *sim-bind-post-state′*:
sim-post-state (λ*x y*. *sim-post-state R* (*g x*) (*k y*)) *f h* ⟹
sim-post-state R (*bind-post-state f g*) (*bind-post-state h k*)
by (*cases f*; *cases h*; *simp add*: *sim-post-state-Sup*)

lemma *sim-bind-post-state-left-iff*:
sim-post-state R (*bind-post-state f g*) *h* ⟷
h = *Failure* ∨ *holds-post-state* (λ*x*. *sim-post-state R* (*g x*) *h*) *f*
by (*cases f*; *cases h*) (*simp-all add*: *sim-post-state-Sup1*)

lemma *sim-bind-post-state-left*:
holds-post-state (λ*x*. *sim-post-state R* (*g x*) *h*) *f* ⟹
sim-post-state R (*bind-post-state f g*) *h*
by (*simp add*: *sim-bind-post-state-left-iff*)

lemma *sim-bind-post-state-right*:
g ≠ ⊥ ⟹ *holds-partial-post-state* (λ*x*. *sim-post-state R f* (*h x*)) *g* ⟹
sim-post-state R f (*bind-post-state g h*)
by (*cases g*) (*auto intro*: *sim-post-state-Sup2*)

lemma *sim-bind-post-state*:
sim-post-state Q f h ⟹ (⋀*x y*. *Q x y* ⟹ *sim-post-state R* (*g x*) (*k y*)) ⟹

sim-post-state R (bind-post-state f g) (bind-post-state h k)
by (*rule sim-bind-post-state'[OF sim-post-state-weaken, of Q]*)

lemma *rel-bind-post-state'*:
rel-post-state (λa b. rel-post-state R (g1 a) (g2 b)) f1 f2 ⟹
rel-post-state R (bind-post-state f1 g1) (bind-post-state f2 g2)
by (*auto simp*: *rel-post-state-eq-sim-post-state*
intro: *sim-bind-post-state' sim-post-state-weaken*)

lemma *rel-bind-post-state*:
rel-post-state Q f1 f2 ⟹ (⋀a b. Q a b ⟹ rel-post-state R (g1 a) (g2 b)) ⟹
rel-post-state R (bind-post-state f1 g1) (bind-post-state f2 g2)
by (*rule rel-bind-post-state'[OF rel-post-state-weaken, of Q]*)

lemma *mono-bind-post-state*: *f1 ≤ f2 ⟹ g1 ≤ g2 ⟹ bind-post-state f1 g1 ≤ bind-post-state f2 g2*
by (*simp flip*: *sim-post-state-eq-iff-le add*: *le-fun-def sim-bind-post-state*)

lemma *Sup-eq-Failure*[*simp*]: *Sup X = Failure ⟷ Failure ∈ X*
by (*simp add*: *Sup-post-state-def*)

lemma *Failure-inf-iff*: *(Failure = x ⊓ y) ⟷ (x = Failure ∧ y = Failure)*
by (*metis Failure-le-iff le-inf-iff*)

lemma *Success-vimage-singleton-cancel*: *Success −‘ {Success X} = {X}*
by *auto*

lemma *Success-vimage-image-cancel*: *Success −‘ (λx. Success (f x)) ‘ X = f ‘ X*
by *auto*

lemma *Success-image-comp*: *(λx. Success (g x)) ‘ X = Success ‘ (g ‘ X)*
by *auto*

lemma *case-top-post-state*[*simp*]:
(case ⊤ of Failure ⇒ f | Success X ⇒ S X) = f
by (*simp add*: *top-post-state-def*)

lemma *case-bot-post-state*[*simp*]:
(case ⊥ of Failure ⇒ f | Success X ⇒ S X) = S {}
by (*simp add*: *bot-post-state-def*)

16.5 *exception-or-result* type

instantiation *option* :: (*type*) *default*
begin

definition *default-option* = *None*

instance ..
end

lemma *Some-ne-default*[*simp*]:
Some x ≠ *default default* ≠ *Some x*
default = *None* ⟷ *True None* = *default* ⟷ *True*
by (*auto simp*: *default-option-def*)

typedef (**overloaded**) ('*a*::*default*, '*b*) *exception-or-result* =
Inl ‘ (*UNIV* − {*default*}) ∪ *Inr* ‘ *UNIV* :: ('*a* + '*b*) *set*
by *auto*

setup-lifting *type-definition-exception-or-result*

context assumes *SORT-CONSTRAINT*('*e*::*default*)
begin

lift-definition *Exception* :: '*e* ⇒ ('*e*, '*v*) *exception-or-result* **is**
λ*e*. *if e* = *default then Inr undefined else Inl e*
by *auto*

lift-definition *Result* :: '*v* ⇒ ('*e*, '*v*) *exception-or-result* **is**
λ*v*. *Inr v*
by *auto*

end

lift-definition
case-exception-or-result :: ('*e*::*default* ⇒ '*a*) ⇒ ('*v* ⇒ '*a*) ⇒ ('*e*, '*v*) *exception-or-result* ⇒ '*a*
is *case-sum* **.**

declare [[*case-translation case-exception-or-result Exception Result*]]

lemma *Result-eq-Result*[*simp*]: *Result a* = *Result b* ⟷ *a* = *b*
by *transfer simp*

lemma *Exception-eq-Exception*[*simp*]: *Exception a* = *Exception b* ⟷ *a* = *b*
by *transfer simp*

lemma *Result-eq-Exception*[*simp*]: *Result a* = *Exception e* ⟷ (*e* = *default* ∧ *a* = *undefined*)
by *transfer simp*

lemma *Exception-eq-Result*[*simp*]: *Exception e* = *Result a* ⟷ (*e* = *default* ∧ *a* = *undefined*)
by (*metis Result-eq-Exception*)

lemma *exception-or-result-cases*[*case-names Exception Result*, *cases type*: *excep-*

tion-or-result]:
(⋀*e*. *e* ≠ *default* ⟹ *x* = *Exception e* ⟹ *P*) ⟹ (⋀*v*. *x* = *Result v* ⟹ *P*) ⟹ *P*
by (*transfer fixing*: *P*) *auto*

lemma *case-exception-or-result-Result*[*simp*]:
(*case Result v of Exception e* ⇒ *f e* | *Result w* ⇒ *g w*) = *g v*
by *transfer simp*

lemma *case-exception-or-result-Exception*[*simp*]:
(*case Exception e of Exception e* ⇒ *f e* | *Result w* ⇒ *g w*) = (*if e* = *default then g undefined else f e*)
by *transfer simp*

Caution: for split rules don't use syntax *case r of Exception e* ⇒ *f e* | *Result w* ⇒ *g w*, as this introduces non eta-contracted *f* and *g*, which don't work with the splitter.

lemma *exception-or-result-split*:
P (*case-exception-or-result f g r*) ⟷
(∀ *e*. *e* ≠ *default* ⟶ *r* = *Exception e* ⟶ *P* (*f e*)) ∧
(∀ *v*. *r* = *Result v* ⟶ *P* (*g v*))
by (*cases r*) *simp-all*

lemma *exception-or-result-split-asm*:
P (*case-exception-or-result f g r*) ⟷
¬ ((∃ *e*. *r* = *Exception e* ∧ *e* ≠ *default* ∧ ¬ *P* (*f e*)) ∨
(∃ *v*. *r* = *Result v* ∧ ¬ *P* (*g v*)))
by (*cases r*) *simp-all*

lemmas *exception-or-result-splits* = *exception-or-result-split exception-or-result-split-asm*

lemma *split-exception-or-result*:
r = (*case r of Exception e* ⇒ *Exception e* | *Result v* ⇒ *Result v*)
by (*cases r*) *auto*

lemma *exception-or-result-nchotomy*:
¬ ((∀ *e*. *e* ≠ *default* ⟶ *x* ≠ *Exception e*) ∧ (∀ *v*. *x* ≠ *Result v*)) **by** (*cases x*) *auto*

lemma *val-split*:
fixes *r*::(*unit*, *'a*) *exception-or-result*
shows
P (*case-exception-or-result f g r*) ⟷
(∀ *v*. *r* = *Result v* ⟶ *P* (*g v*))
by (*cases r*) *simp-all*

lemma *val-split-asm*:
fixes *r*::(*unit*, *'a*) *exception-or-result*
shows *P* (*case-exception-or-result f g r*) ⟷

$\neg\ (\exists\, v.\ r = \mathit{Result}\ v \wedge \neg\ P\ (g\ v))$
by (*cases r*) *simp-all*

lemmas *val-splits*[*split*] = *val-split val-split-asm*

instantiation *exception-or-result* :: ({*equal*, *default*}, *equal*) *equal* **begin**

definition *equal-exception-or-result a b* =
(*case a of*
Exception e ⇒ (*case b of Exception e'* ⇒ *HOL.equal e e'* | *Result s* ⇒ *False*)
| *Result r* ⇒ (*case b of Exception e* ⇒ *False* | *Result s* ⇒ *HOL.equal r s*)
)
instance proof qed (*simp add*: *equal-exception-or-result-def equal-eq*
split: *exception-or-result-splits*)

end

definition *is-Exception*:: ('*e*::*default*, '*a*) *exception-or-result* ⇒ *bool* **where**
is-Exception x ≡ (*case x of Exception e* ⇒ *e* ≠ *default* | *Result* - ⇒ *False*)

definition *is-Result*:: ('*e*::*default*, '*a*) *exception-or-result* ⇒ *bool* **where**
is-Result x ≡ (*case x of Exception e* ⇒ *e* = *default* | *Result* - ⇒ *True*)

definition *the-Exception*:: ('*e*::*default*, '*a*) *exception-or-result* ⇒ '*e* **where**
the-Exception x ≡ (*case x of Exception e* ⇒ *e* | *Result* - ⇒ *default*)

definition *the-Result*:: ('*e*::*default*, '*a*) *exception-or-result* ⇒ '*a* **where**
the-Result x ≡ (*case x of Exception e* ⇒ *undefined* | *Result v* ⇒ *v*)

lemma *is-Exception-simps*[*simp*]:
is-Exception (*Exception e*) = (*e* ≠ *default*)
is-Exception (*Result v*) = *False*
by (*auto simp add*: *is-Exception-def*)

lemma *is-Result-simps*[*simp*]:
is-Result (*Exception e*) = (*e* = *default*)
is-Result (*Result v*) = *True*
by (*auto simp add*: *is-Result-def*)

lemma *the-Exception-simp*[*simp*]:
the-Exception (*Exception e*) = *e*
by (*auto simp add*: *the-Exception-def*)

lemma *the-Exception-Result*:
the-Exception (*Result v*) = *default*
by (*simp add*: *the-Exception-def*)

definition *undefined-unit*::*unit* ⇒ '*b* **where** *undefined-unit x* ≡ *undefined*

syntax *-Res :: pttrn ⇒ pttrn (‹(‹open-block notation=‹prefix Res››Res -)›)*
syntax-consts *-Res ⇌ case-exception-or-result*
translations *λRes x. b ⇌ CONST case-exception-or-result (CONST undefined-unit) (λx. b)*

term *λRes x. f x*
term *λRes (x, y, z). f x y z*
term *λRes (x, y, z) s. P x y s z*

lifting-update *exception-or-result.lifting*
lifting-forget *exception-or-result.lifting*

inductive *rel-exception-or-result:: ('e::default ⇒ 'f::default ⇒ bool) ⇒ ('a ⇒ 'b ⇒ bool) ⇒*
('e, 'a) exception-or-result ⇒ ('f, 'b) exception-or-result ⇒ bool
where
Exception:
E e f ⟹ e ≠ default ⟹ f ≠ default ⟹
rel-exception-or-result E R (Exception e) (Exception f) |
Result:
R a b ⟹ rel-exception-or-result E R (Result a) (Result b)

lemma *All-exception-or-result-cases:*
(∀x. P (x::(-, -) exception-or-result)) ⟷
(∀err. err ≠ default ⟶ P (Exception err)) ∧ (∀v. P (Result v))
by *(metis exception-or-result-cases)*

lemma *Ball-exception-or-result-cases:*
(∀x∈s. P (x::(-, -) exception-or-result)) ⟷
(∀err. err ≠ default ⟶ Exception err ∈ s ⟶ P (Exception err)) ∧
(∀v. Result v ∈ s ⟶ P (Result v))
by *(metis exception-or-result-cases)*

lemma *Bex-exception-or-result-cases:*
(∃x∈s. P (x::(-, -) exception-or-result)) ⟷
(∃err. err ≠ default ∧ Exception err ∈ s ∧ P (Exception err)) ∨
(∃v. Result v ∈ s ∧ P (Result v))
by *(metis exception-or-result-cases)*

lemma *Ex-exception-or-result-cases:*
(∃x. P (x::(-, -) exception-or-result)) ⟷
(∃err. err ≠ default ∧ P (Exception err)) ∨
(∃v. P (Result v))
by *(metis exception-or-result-cases)*

lemma *case-distrib-exception-or-result:*
f (case x of Exception e ⇒ E e | Result v ⇒ R v) = (case x of Exception e ⇒ f (E e) | Result v ⇒ f (R v))

by (*auto split*: *exception-or-result-split*)

lemma *rel-exception-or-result-Results*[*simp*]:
rel-exception-or-result E R (*Result a*) (*Result b*) ⟷ *R a b*
by (*simp add*: *rel-exception-or-result.simps*)

lemma *rel-exception-or-result-Exception*[*simp*]:
e ≠ *default* ⟹ *f* ≠ *default* ⟹
rel-exception-or-result E R (*Exception e*) (*Exception f*) ⟷ *E e f*
by (*simp add*: *rel-exception-or-result.simps*)

lemma *rel-exception-or-result-Result-Exception*[*simp*]:
e ≠ *default* ⟹ ¬ *rel-exception-or-result E R* (*Exception e*) (*Result b*)
f ≠ *default* ⟹ ¬ *rel-exception-or-result E R* (*Result a*) (*Exception f*)
by (*auto simp add*: *rel-exception-or-result.simps*)

lemma *is-Exception-iff*: *is-Exception x* ⟷ (∃ *e*. *e* ≠ *default* ∧ *x* = *Exception e*)
apply (*cases x*)
apply (*auto*)
done

lemma *rel-exception-or-result-converse*:
(*rel-exception-or-result E R*)$^{-1-1}$ = *rel-exception-or-result* E^{-1-1} R^{-1-1}
apply (*rule ext*)+
apply (*auto simp add*: *rel-exception-or-result.simps*)
done

lemma *rel-eception-or-result-Result*[*simp*]:
rel-exception-or-result E R (*Result x*) (*Result y*) = *R x y*
by (*auto simp add*: *rel-exception-or-result.simps*)

lemma *rel-exception-or-result-eq-conv*: *rel-exception-or-result* (=) (=) = (=)
apply (*rule ext*)+
by (*auto simp add*: *rel-exception-or-result.simps*)
(*meson exception-or-result-cases*)

lemma *rel-exception-or-result-sum-eq*: *rel-exception-or-result* (=) (=) = (=)
apply (*rule ext*)+
apply (*subst* (*1*) *split-exception-or-result*)
apply (*auto simp add*: *rel-exception-or-result.simps split*: *exception-or-result-splits*)
done

definition
map-exception-or-result::
('*e*::*default* ⇒ '*f*::*default*) ⇒ ('*a* ⇒ '*b*) ⇒ ('*e*, '*a*) *exception-or-result* ⇒
('*f*, '*b*) *exception-or-result*
where
map-exception-or-result E F x =
(*case x of Exception e* ⇒ *Exception* (*E e*) | *Result v* ⇒ *Result* (*F v*))

lemma *map-exception-or-result-Exception*[*simp*]:
e ≠ *default* ⟹ *map-exception-or-result E F* (*Exception e*) = *Exception* (*E e*)
by (*simp add*: *map-exception-or-result-def*)

lemma *map-exception-or-result-Result*[*simp*]:
map-exception-or-result E F (*Result v*) = *Result* (*F v*)
by (*simp add*: *map-exception-or-result-def*)

lemma *map-exception-or-result-id*: *map-exception-or-result id id x* = *x*
by (*cases x*; *simp*)

lemma *map-exception-or-result-comp*:
assumes *E2*: ⋀*x*. *x* ≠ *default* ⟹ *E2 x* ≠ *default*
shows *map-exception-or-result E1 F1* (*map-exception-or-result E2 F2 x*) =
map-exception-or-result (*E1* ∘ *E2*) (*F1* ∘ *F2*) *x*
by (*cases x*; *auto dest*: *E2*)

lemma *le-bind-post-state-exception-or-result-cases*[*case-names Exception Result*]:
assumes
holds-partial-post-state (λ(*x*, *t*). ∀ *e*. *e* ≠ *default* ⟶ *x* = *Exception e* ⟶ *X e t* ≤ *X*′ *e t*) *x*
assumes *holds-partial-post-state* (λ(*x*, *t*). ∀ *v*. *x* = *Result v* ⟶ *V v t* ≤ *V*′ *v t*) *x*
shows *bind-post-state x* (λ(*r*, *t*). *case r of Exception e* ⇒ *X e t* | *Result v* ⇒ *V v t*) ≤
bind-post-state x (λ(*r*, *t*). *case r of Exception e* ⇒ *X*′ *e t*| *Result v* ⇒ *V*′ *v t*)
using *assms*
by (*cases x*; *force intro*!: *SUP-mono*′′ *split*: *exception-or-result-splits prod.splits*)

16.6 *spec-monad* type

typedef (**overloaded**) ('*e*::*default*, '*a*, '*s*) *spec-monad* =
UNIV :: ('*s* ⇒ (('*e*::*default*, '*a*) *exception-or-result* × '*s*) *post-state*) *set*
morphisms *run Spec*
by (*fact UNIV-witness*)

lemma *run-case-prod-distrib*[*simp*]:
run (*case x of* (*r*, *s*) ⇒ *f r s*) *t* = (*case x of* (*r*, *s*) ⇒ *run* (*f r s*) *t*)
by (*rule prod.case-distrib*)

lemma *image-case-prod-distrib*[*simp*]:
(λ(*r*, *t*). *f r t*) ‘ (λ*v*. (*g v*, *h v*)) ‘ *R* = (λ*v*. (*f* (*g v*) (*h v*))) ‘ *R*
by *auto*

lemma *run-Spec*[*simp*]: *run* (*Spec p*) *s* = *p s*
by (*simp add*: *Spec-inverse*)

setup-lifting *type-definition-spec-monad*

lemma *spec-monad-ext*: $(\bigwedge s.\ run\ f\ s = run\ g\ s) \Longrightarrow f = g$
apply *transfer*
apply *auto*
done

lemma *spec-monad-ext-iff*: $f = g \longleftrightarrow (\forall s.\ run\ f\ s = run\ g\ s)$
using *spec-monad-ext* **by** *auto*

instantiation *spec-monad* :: (*default*, *type*, *type*) *complete-lattice*
begin

lift-definition
less-eq-spec-monad :: ('*e*::*default*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad* ⇒ *bool*
is (≤) .

lift-definition
less-spec-monad :: ('*e*::*default*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad* ⇒ *bool*
is (<) .

lift-definition *bot-spec-monad* :: ('*e*::*default*, '*r*, '*s*) *spec-monad* **is** *bot* .

lift-definition *top-spec-monad* :: ('*e*::*default*, '*r*, '*s*) *spec-monad* **is** *top* .

lift-definition *Inf-spec-monad* :: ('*e*::*default*, '*r*, '*s*) *spec-monad set* ⇒ ('*e*, '*r*, '*s*) *spec-monad*
is *Inf* .

lift-definition *Sup-spec-monad* :: ('*e*::*default*, '*r*, '*s*) *spec-monad set* ⇒ ('*e*, '*r*, '*s*) *spec-monad*
is *Sup* .

lift-definition
sup-spec-monad ::
('*e*::*default*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad*
is *sup* .

lift-definition
inf-spec-monad ::
('*e*::*default*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad*
is *inf* .

instance
apply (*standard*; *transfer*)
subgoal by (*rule less-le-not-le*)
subgoal by (*rule order-refl*)
subgoal by (*rule order-trans*)

subgoal by (*rule antisym*)
subgoal by (*rule inf-le1*)
subgoal by (*rule inf-le2*)
subgoal by (*rule inf-greatest*)
subgoal by (*rule sup-ge1*)
subgoal by (*rule sup-ge2*)
subgoal by (*rule sup-least*)
subgoal by (*rule Inf-lower*)
subgoal by (*rule Inf-greatest*)
subgoal by (*rule Sup-upper*)
subgoal by (*rule Sup-least*)
subgoal by (*rule Inf-empty*)
subgoal by (*rule Sup-empty*)
done
end

lift-definition *fail* :: ('*e*::*default*, '*a*, '*s*) *spec-monad*
is λ*s*. *Failure* **.**

lift-definition
bind-exception-or-result ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒
(('*e*, '*a*) *exception-or-result* ⇒ ('*f*::*default*, '*b*, '*s*) *spec-monad*) ⇒ ('*f*, '*b*, '*s*) *spec-monad*
is
λ*f h s*. *bind-post-state* (*f s*) (λ(*v*, *t*). *h v t*) **.**

lift-definition *bind-handle* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒
('*a* ⇒ ('*f*, '*b*, '*s*) *spec-monad*) ⇒ ('*e* ⇒ ('*f*, '*b*, '*s*) *spec-monad*) ⇒
('*f*::*default*, '*b*, '*s*) *spec-monad*
is λ*f g h s*. *bind-post-state* (*f s*) (λ(*r*, *t*). *case r of Exception e* ⇒ *h e t* | *Result v* ⇒ *g v t*) **.**

lift-definition *yield* :: ('*e*, '*a*) *exception-or-result* ⇒ ('*e*::*default*, '*a*, '*s*) *spec-monad*
is λ*r s*. *pure-post-state* (*r*, *s*) **.**

abbreviation *return r* ≡ *yield* (*Result r*)

abbreviation *skip* ≡ *return* ()

abbreviation *throw-exception-or-result e* ≡ *yield* (*Exception e*)

lift-definition *get-state* :: ('*e*::*default*, '*s*, '*s*) *spec-monad*
is λ*s*. *pure-post-state* (*Result s*, *s*) **.**

lift-definition *set-state* :: '*s* ⇒ ('*e*::*default*, *unit*, '*s*) *spec-monad*
is λ*t s*. *pure-post-state* (*Result* (), *t*) **.**

lift-definition *map-value* ::
  ((*'e*::*default*, *'a*) *exception-or-result* ⇒ (*'f*::*default*, *'b*) *exception-or-result*) ⇒
    (*'e*, *'a*, *'s*) *spec-monad* ⇒ (*'f*, *'b*, *'s*) *spec-monad*
 is *λf g s. map-post-state* (*apfst f*) (*g s*) **.**

lift-definition *vmap-value* ::
  ((*'e*::*default*, *'a*) *exception-or-result* ⇒ (*'f*::*default*, *'b*) *exception-or-result*) ⇒
    (*'f*, *'b*, *'s*) *spec-monad* ⇒ (*'e*, *'a*, *'s*) *spec-monad*
 is *λf g s. vmap-post-state* (*apfst f*) (*g s*) **.**

definition *bind* ::
  (*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ (*'a* ⇒ (*'e*, *'b*, *'s*) *spec-monad*) ⇒ (*'e*, *'b*, *'s*)
spec-monad
where
 bind f g = *bind-handle f g throw-exception-or-result*

adhoc-overloading
 Monad-Syntax.bind ⇌ *bind*

lift-definition *lift-state* ::
  (*'s* ⇒ *'t* ⇒ *bool*) ⇒ (*'e*::*default*, *'a*, *'t*) *spec-monad* ⇒ (*'e*, *'a*, *'s*) *spec-monad*
 is *λR p s. lift-post-state* (*rel-prod* (=) *R*) (*SUP t*∈{*t. R s t*}*. p t*) **.**

definition *exec-concrete* ::
 (*'s* ⇒ *'t*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ (*'e*, *'a*, *'t*) *spec-monad*
where
 exec-concrete st = *lift-state* (*λt s. t* = *st s*)

definition *exec-abstract* ::
 (*'s* ⇒ *'t*) ⇒ (*'e*::*default*, *'a*, *'t*) *spec-monad* ⇒ (*'e*, *'a*, *'s*) *spec-monad*
where
 exec-abstract st = *lift-state* (*λs t. t* = *st s*)

definition *select-value* :: (*'e*, *'a*) *exception-or-result set* ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad*
where
 select-value R = (*SUP r*∈*R. yield r*)

definition *select* :: *'a set* ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
 select S = (*SUP a*∈*S. return a*)

definition *unknown* :: (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
 unknown ≡ *select UNIV*

definition *gets* :: (*'s* ⇒ *'a*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
 gets f = *bind get-state* (*λs. return* (*f s*))

definition *assert-opt* :: *'a option* ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
 assert-opt v = (*case v of None* ⇒ *fail* | *Some v* ⇒ *return v*)

definition *gets-the* :: (*'s* ⇒ *'a option*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
gets-the f = *bind* (*gets f*) *assert-opt*

definition *modify* :: (*'s* ⇒ *'s*) ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
modify f = *bind get-state* (λ*s*. *set-state* (*f s*))

definition *assume* :: *bool* ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
assume P = (*if P then return* () *else bot*)

definition *assert* :: *bool* ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
assert P = (*if P then return* () *else top*)

definition *assuming* :: (*'s* ⇒ *bool*) ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
assuming P = *do* {*s* ← *get-state*; *assume* (*P s*)}

definition *guard*:: (*'s* ⇒ *bool*) ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
guard P = *do* {*s* ← *get-state*; *assert* (*P s*)}

definition *assume-result-and-state* :: (*'s* ⇒ (*'a* × *'s*) *set*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
assume-result-and-state f = *do* {*s* ← *get-state*; (*v*, *t*) ← *select* (*f s*); *set-state t*; *return v*}

lift-definition *assume-outcome* ::
(*'s* ⇒ ((*'e*, *'a*) *exception-or-result* × *'s*) *set*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad*
is λ*f s*. *Success* (*f s*) **.**

definition *assert-result-and-state* ::
(*'s* ⇒ (*'a* × *'s*) *set*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad*
where
assert-result-and-state f =
do {*s* ← *get-state*; *assert* (*f s* ≠ {}); (*v*, *t*) ← *select* (*f s*); *set-state t*; *return v*}

abbreviation *state-select* :: (*'s* × *'s*) *set* ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad*
where
state-select r ≡ *assert-result-and-state* (λ*s*. {((), *s'*). (*s*, *s'*) ∈ *r*})

definition *condition* ::
(*'s* ⇒ *bool*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ (*'e*, *'a*, *'s*) *spec-monad* ⇒
(*'e*, *'a*, *'s*) *spec-monad*
where
condition C T F =
bind get-state (λ*s*. *if C s then T else F*)

notation (**output**)
condition (‹(‹*notation*=‹*prefix condition*››*condition* (-)// (-)// (-))› [*1000*,*1000*,*1000*] *999*)

definition *when* ::*bool* ⇒ ('*e*::*default*, *unit*, '*s*) *spec-monad* ⇒ ('*e*, *unit*, '*s*) *spec-monad*
where *when c f* ≡ *condition* (λ-. *c*) *f skip*

abbreviation *unless* ::*bool* ⇒ ('*e*::*default*, *unit*, '*s*) *spec-monad* ⇒ ('*e*, *unit*, '*s*) *spec-monad*
where *unless c* ≡ *when* (¬*c*)

definition *on-exit'* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*e*, *unit*, '*s*) *spec-monad* ⇒ ('*e*, '*a*, '*s*) *spec-monad*
where
on-exit' f c ≡
bind-exception-or-result f (λ*r*. *do* { *c*; *yield r* })

definition *on-exit* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*s* × '*s*) *set* ⇒ ('*e*, '*a*, '*s*) *spec-monad*
where
on-exit f cleanup ≡ *on-exit' f* (*state-select cleanup*)

abbreviation *guard-on-exit* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*s* ⇒ *bool*) ⇒ ('*s* × '*s*) *set* ⇒ ('*e*, '*a*, '*s*) *spec-monad*
where
guard-on-exit f grd cleanup ≡ *on-exit' f* (*bind* (*guard grd*) (λ-. *state-select cleanup*))

abbreviation *assume-on-exit* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*s* ⇒ *bool*) ⇒ ('*s* × '*s*) *set* ⇒ ('*e*, '*a*, '*s*) *spec-monad*
where
assume-on-exit f grd cleanup ≡
on-exit' f (*bind* (*assuming grd*) (λ-. *state-select cleanup*))

lift-definition *run-bind* ::
('*e*::*default*, '*a*, '*t*) *spec-monad* ⇒ '*t* ⇒
(('*e*, '*a*) *exception-or-result* ⇒ '*t* ⇒ ('*f*::*default*, '*b*, '*s*) *spec-monad*) ⇒
('*f*::*default*, '*b*, '*s*) *spec-monad*
is λ*f t g s*. *bind-post-state* (*f t*) (λ(*r*, *t*). *g r t s*) .
— *run-bind* might be a more canonical building block compared to *lift-state*. See subsection on *run-bind* below.

type-synonym ('*e*, '*a*, '*s*) *predicate* = ('*e*, '*a*) *exception-or-result* ⇒ '*s* ⇒ *bool*

lift-definition *runs-to* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ '*s* ⇒ ('*e*, '*a*, '*s*) *predicate* ⇒ *bool*
(‹(‹*open-block notation*=‹*mixfix runs-to*››-/ ⋅ - ⦃ - ⦄)› [*61*, *1000*, *0*] *30*) — syntax -*do-block* has 62
is λ*f s Q*. *holds-post-state* (λ(*r*, *t*). *Q r t*) (*f s*) .

lift-definition *runs-to-partial* ::

(*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ *'s* ⇒ (*'e*, *'a*, *'s*) *predicate* ⇒ *bool*
(‹(‹*open-block notation*=‹*mixfix runs-to-partial*››-/ ⋅ - *?*⦃ - ⦄)› [*61*, *1000*, *0*] *30*)
is λ*f s Q*. *holds-partial-post-state* (λ(*r*, *t*). *Q r t*) (*f s*) .

lift-definition *refines* ::
(*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ (*'f*::*default*, *'b*, *'t*) *spec-monad* ⇒ *'s* ⇒ *'t* ⇒
((('*e*, *'a*) *exception-or-result* × *'s*) ⇒ (('*f*, *'b*) *exception-or-result* × *'t*) ⇒ *bool*) ⇒ *bool*
is λ*f g s t R*. *sim-post-state R* (*f s*) (*g t*) .

lift-definition *rel-spec* ::
(*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ (*'f*::*default*, *'b*, *'t*) *spec-monad* ⇒ *'s* ⇒ *'t* ⇒
((('*e*, *'a*) *exception-or-result* × *'s*) ⇒ (('*f*, *'b*) *exception-or-result* × *'t*) ⇒ *bool*) ⇒ *bool*
is
λ*f g s t R*. *rel-post-state R* (*f s*) (*g t*) .

definition *rel-spec-monad* ::
('*s* ⇒ *'t* ⇒ *bool*) ⇒ (('*e*, *'a*) *exception-or-result* ⇒ (*'f*, *'b*) *exception-or-result* ⇒ *bool*) ⇒
(*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ (*'f*::*default*, *'b*, *'t*) *spec-monad* ⇒ *bool*
where
rel-spec-monad R Q f g =
(∀ *s t*. *R s t* ⟶ *rel-post-state* (*rel-prod Q R*) (*run f s*) (*run g t*))

lift-definition *always-progress* :: (*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ *bool* **is**
λ*p*. ∀ *s*. *p s* ≠ *bot* .

named-theorems *run-spec-monad Simplification rules to run a Spec monad*
named-theorems *runs-to-iff Equivalence theorems for runs to predicate*
named-theorems *always-progress-intros intro rules for always-progress predicate*

lemma *runs-to-partialI*: (⋀*r x*. *P r x*) ⟹ *f* ⋅ *s ?*⦃ *P* ⦄
by (*cases run f s*) (*auto simp*: *runs-to-partial.rep-eq*)

lemma *runs-to-partial-True*: *f* ⋅ *s ?*⦃ λ*r s*. *True* ⦄
by (*simp add*: *runs-to-partialI*)

lemma *runs-to-partial-conj*:
f ⋅ *s ?*⦃ *P* ⦄ ⟹ *f* ⋅ *s ?*⦃ *Q* ⦄ ⟹ *f* ⋅ *s ?*⦃ λ*r s*. *P r s* ∧ *Q r s* ⦄
by (*cases run f s*) (*auto simp*: *runs-to-partial.rep-eq*)

lemma *refines-iff'*:
refines f g s t R ⟷ (∀ *P*. *g* ⋅ *t* ⦃ λ*r s*. ∀ *p t*. *R* (*p*, *t*) (*r*, *s*) ⟶ *P p t* ⦄ ⟶ *f* ⋅ *s* ⦃ *P* ⦄)
apply *transfer*
apply (*simp add*: *sim-post-state-iff le-fun-def split-beta' all-comm*[**where** *'a*=*'a*])
apply (*rule all-cong-map*[**where** *f'*=λ*P* (*x*, *y*). *P x y* **and** *f*=λ*P x y*. *P* (*x*, *y*)])

apply *auto*
done

lemma *refines-weaken*:
refines f g s t R $\Longrightarrow$ ($\bigwedge$*r s q t. R* (*r*, *s*) (*q*, *t*) $\Longrightarrow$ *Q* (*r*, *s*) (*q*, *t*)) $\Longrightarrow$ *refines f g s t Q*
by *transfer* (*auto intro*: *sim-post-state-weaken*)

lemma *refines-mono*:
($\bigwedge$*r s q t. R* (*r*, *s*) (*q*, *t*) $\Longrightarrow$ *Q* (*r*, *s*) (*q*, *t*)) $\Longrightarrow$ *refines f g s t R* $\Longrightarrow$ *refines f g s t Q*
by (*rule refines-weaken*)

lemma *refines-refl*: ($\bigwedge$*r t. R* (*r*, *t*) (*r*, *t*)) $\Longrightarrow$ *refines f f s s R*
by *transfer* (*auto intro*: *sim-post-state-refl*)

lemma *refines-trans*:
refines f g s t R $\Longrightarrow$ *refines g h t u Q* $\Longrightarrow$
($\bigwedge$*r s p t q u. R* (*r*, *s*) (*p*, *t*) $\Longrightarrow$ *Q* (*p*, *t*) (*q*, *u*) $\Longrightarrow$ *S* (*r*, *s*) (*q*, *u*)) $\Longrightarrow$
refines f h s u S
by *transfer* (*auto intro*: *sim-post-state-trans*)

lemma *refines-trans'*: *refines f g s t1 R* $\Longrightarrow$ *refines g h t1 t2 Q* $\Longrightarrow$ *refines f h s t2* (*R OO Q*)
by (*rule refines-trans*; *assumption?*) *auto*

lemma *refines-strengthen*:
refines f g s t R $\Longrightarrow$ *f* $\cdot$ *s* ?$\{\!|$ *F* $|\!\}$ $\Longrightarrow$ *g* $\cdot$ *t* ?$\{\!|$ *G* $|\!\}$ $\Longrightarrow$
($\bigwedge$*x s y t. R* (*x*, *s*) (*y*, *t*) $\Longrightarrow$ *F x s* $\Longrightarrow$ *G y t* $\Longrightarrow$ *Q* (*x*, *s*) (*y*, *t*)) $\Longrightarrow$
refines f g s t Q
by *transfer* (*fastforce elim*!: *sim-post-state.cases simp*: *sim-set-def split-beta'*)

lemma *refines-strengthen1*:
refines f g s t R $\Longrightarrow$ *f* $\cdot$ *s* ?$\{\!|$ *F* $|\!\}$ $\Longrightarrow$
($\bigwedge$*x s y t. R* (*x*, *s*) (*y*, *t*) $\Longrightarrow$ *F x s* $\Longrightarrow$ *Q* (*x*, *s*) (*y*, *t*)) $\Longrightarrow$
refines f g s t Q
by (*rule refines-strengthen*[*OF - - runs-to-partial-True*]; *assumption?*)

lemma *refines-strengthen2*:
refines f g s t R $\Longrightarrow$ *g* $\cdot$ *t* ?$\{\!|$ *G* $|\!\}$ $\Longrightarrow$
($\bigwedge$*x s y t. R* (*x*, *s*) (*y*, *t*) $\Longrightarrow$ *G y t* $\Longrightarrow$ *Q* (*x*, *s*) (*y*, *t*)) $\Longrightarrow$
refines f g s t Q
by (*rule refines-strengthen*[*OF - runs-to-partial-True*]; *assumption?*)

lemma *refines-cong-cases*:
assumes $\bigwedge$*e s e' s'. e* $\neq$ *default* $\Longrightarrow$ *e'* $\neq$ *default* $\Longrightarrow$
R (*Exception e*, *s*) (*Exception e'*, *s'*) $\longleftrightarrow$ *Q* (*Exception e*, *s*) (*Exception e'*, *s'*)
assumes $\bigwedge$*e s x' s'. e* $\neq$ *default* $\Longrightarrow$
R (*Exception e*, *s*) (*Result x'*, *s'*) $\longleftrightarrow$ *Q* (*Exception e*, *s*) (*Result x'*, *s'*)

assumes $\bigwedge x\ s\ e'\ s'.\ e' \neq default \Longrightarrow$
$R\ (Result\ x,\ s)\ (Exception\ e',\ s') \longleftrightarrow Q\ (Result\ x,\ s)\ (Exception\ e',\ s')$
assumes $\bigwedge x\ s\ x'\ s'.\ R\ (Result\ x,\ s)\ (Result\ x',\ s') \longleftrightarrow Q\ (Result\ x,\ s)\ (Result\ x',\ s')$
shows $refines\ f\ f'\ s\ s'\ R \longleftrightarrow refines\ f\ f'\ s\ s'\ Q$
apply (*clarsimp intro*!: *arg-cong*[**where** *f*=*refines f f' s s'*] *simp*: *fun-eq-iff del*: *iffI*)
subgoal for $v\ s\ v'\ s'$ **by** (*cases v*; *cases v'*; *simp add*: *assms*)
done

lemma *refines-parallel*:
$always\text{-}progress\ g \Longrightarrow f \cdot s \{\!|\ P\ |\!\} \Longrightarrow g \cdot t \{\!|\ Q\ |\!\} \Longrightarrow$
$refines\ f\ g\ s\ t\ (\lambda(r,\ s)\ (q,\ t).\ P\ r\ s \wedge Q\ q\ t)$
apply (*transfer fixing*: *P Q*)
subgoal for $g\ f\ s\ t$
by (*cases f s*; *cases g t*)
(*auto simp*: *split-beta' bot-post-state-def dest*!: *spec*[*of* - *t*])
done

lemma *runs-to-partial-weaken*[*runs-to-vcg-weaken*]:
$f \cdot s\ ?\{\!|Q|\!\} \Longrightarrow (\bigwedge r\ t.\ Q\ r\ t \Longrightarrow Q'\ r\ t) \Longrightarrow f \cdot s\ ?\{\!|Q'|\!\}$
by *transfer* (*auto intro*: *holds-partial-post-state-weaken*)

lemma *runs-to-partial-Res-True*[*simp*]: $p \cdot s\ ?\{\!|\ \lambda Res\ a\ s.\ True\ |\!\}$
by (*rule runs-to-partial-weaken*[*OF runs-to-partial-True*]) *auto*

lemma *runs-to-weaken*[*runs-to-vcg-weaken*]: $f \cdot s \{\!|Q|\!\} \Longrightarrow (\bigwedge r\ t.\ Q\ r\ t \Longrightarrow Q'\ r\ t) \Longrightarrow f \cdot s \{\!|Q'|\!\}$
by *transfer* (*auto intro*: *holds-post-state-weaken*)

lemma *runs-to-partial-imp-runs-to-partial*[*mono*]:
$(\bigwedge a\ s.\ P\ a\ s \longrightarrow Q\ a\ s) \Longrightarrow f \cdot s\ ?\{\!|\ P\ |\!\} \longrightarrow f \cdot s\ ?\{\!|\ Q\ |\!\}$
using *runs-to-partial-weaken*[*of f s P Q*] **by** *auto*

lemma *runs-to-imp-runs-to*[*mono*]:
$(\bigwedge a\ s.\ P\ a\ s \longrightarrow Q\ a\ s) \Longrightarrow f \cdot s \{\!|\ P\ |\!\} \longrightarrow f \cdot s \{\!|\ Q\ |\!\}$
using *runs-to-weaken*[*of f s P Q*] **by** *auto*

lemma *refines-imp-refines*[*mono*]:
$(\bigwedge a\ s.\ P\ a\ s \longrightarrow Q\ a\ s) \Longrightarrow refines\ f\ g\ s\ t\ P \longrightarrow refines\ f\ g\ s\ t\ Q$
using *refines-weaken*[*of f g s t P Q*] **by** *auto*

lemma *runs-to-strengthen*:
$f \cdot s \{\!|\ P\ |\!\} \Longrightarrow f \cdot s\ ?\{\!|\ Q\ |\!\} \Longrightarrow (\bigwedge a\ s.\ P\ a\ s \Longrightarrow Q\ a\ s \Longrightarrow R\ a\ s) \Longrightarrow f \cdot s \{\!|\ R\ |\!\}$
by (*cases run f s*) (*auto simp*: *runs-to.rep-eq runs-to-partial.rep-eq split-beta'*)

lemma *runs-to-partial-weaken-rel-spec*:
$g \cdot t\ ?\{\!|\ \lambda r\ t'.\ \forall p\ s'.\ R\ (p,\ s')\ (r,\ t') \longrightarrow P\ p\ s'\ |\!\} \Longrightarrow rel\text{-}spec\ f\ g\ s\ t\ R \Longrightarrow f \cdot s$

?⦃ P ⦄
apply *transfer*
apply (*rule holds-partial-post-state-rel*, *assumption*)
apply (*auto simp*: *split-beta'*)
done

lemma *runs-to-cong-cases*:
assumes $\bigwedge e\ s.\ e \neq default \Longrightarrow P\ (Exception\ e)\ s \longleftrightarrow Q\ (Exception\ e)\ s$
assumes $\bigwedge x\ s.\ P\ (Result\ x)\ s \longleftrightarrow Q\ (Result\ x)\ s$
shows $f \cdot s \Lbag P \Rbag \longleftrightarrow f \cdot s \Lbag Q \Rbag$
apply (*clarsimp intro*!: *arg-cong*[**where** *f*=*runs-to - -*] *simp*: *fun-eq-iff del*: *iffI*)
subgoal for *v s* **by** (*cases v*; *simp add*: *assms*)
done

lemma *le-spec-monad-le-refines-iff*: $f \leq g \longleftrightarrow (\forall s.\ refines\ f\ g\ s\ s\ (=))$
by *transfer* (*simp add*: *le-fun-def sim-post-state-eq-iff-le*)

lemma *spec-monad-le-iff*: $f \leq g \longleftrightarrow (\forall P\ (s::'a).\ g \cdot s \Lbag P \Rbag \longrightarrow f \cdot s \Lbag P \Rbag)$
by (*simp add*: *le-spec-monad-le-refines-iff refines-iff' all-comm*[**where** *'a='a*])

lemma *spec-monad-eq-iff*: $f = g \longleftrightarrow (\forall P\ s.\ f \cdot s \Lbag P \Rbag \longleftrightarrow g \cdot s \Lbag P \Rbag)$
by (*simp add*: *order-eq-iff spec-monad-le-iff*)

lemma *spec-monad-eqI*: $(\bigwedge P\ s.\ f \cdot s \Lbag P \Rbag \longleftrightarrow g \cdot s \Lbag P \Rbag) \Longrightarrow f = g$
by (*simp add*: *spec-monad-eq-iff*)

lemma *runs-to-Sup-iff*[*runs-to-iff*]: $(Sup\ X) \cdot s \Lbag P \Rbag \longleftrightarrow (\forall x{\in}X.\ x \cdot s \Lbag P \Rbag)$
by *transfer simp*

lemma *runs-to-lfp*:
assumes *f*: *mono f*
assumes *x-s*: *P x s*
assumes $*$: $\bigwedge p.\ (\bigwedge x\ s.\ P\ x\ s \Longrightarrow p\ x \cdot s \Lbag R \Rbag) \Longrightarrow (\bigwedge x\ s.\ P\ x\ s \Longrightarrow f\ p\ x \cdot s \Lbag R \Rbag)$
shows *lfp f x* $\cdot$ *s* ⦃ *R* ⦄
proof −
have $\forall x\ s.\ P\ x\ s \longrightarrow lfp\ f\ x \cdot s \Lbag R \Rbag$
apply (*rule lfp-ordinal-induct*[*OF f*])
subgoal using $*$ **by** *blast*
subgoal by (*simp add*: *runs-to-Sup-iff*)
done
with *x-s* **show** *?thesis* **by** *auto*
qed

lemma *runs-to-partial-alt*:
f $\cdot$ *s ?⦃ Q ⦄* $\longleftrightarrow$ *run f s* $= \top \vee f \cdot s \Lbag Q \Rbag$
by (*cases run f s*; *simp add*: *runs-to.rep-eq runs-to-partial.rep-eq top-post-state-def*)

lemma *runs-to-of-runs-to-partial-runs-to'*:

$f \cdot s \{\!|\ P\ |\!\} \Longrightarrow f \cdot s\ ?\{\!|\ Q\ |\!\} \Longrightarrow f \cdot s \{\!|\ Q\ |\!\}$
by (*auto simp*: *runs-to-partial-alt runs-to.rep-eq*)

lemma *runs-to-partial-of-runs-to*:
$f \cdot s \{\!|\ Q\ |\!\} \Longrightarrow f \cdot s\ ?\{\!|\ Q\ |\!\}$
by (*auto simp*: *runs-to-partial-alt*)

lemma *runs-to-partial-tivial*[*simp*]: $f \cdot s\ ?\{\!|\ \lambda\text{- -}.\ True\ |\!\}$
by (*cases run f s*; *simp add*: *runs-to-partial.rep-eq*)

lemma *le-spec-monad-le-run-iff*: $f \leq g \longleftrightarrow (\forall s.\ run\ f\ s \leq run\ g\ s)$
by (*simp add*: *le-fun-def less-eq-spec-monad.rep-eq*)

lemma *refines-le-run-trans*: *refines f g s s1 R* $\Longrightarrow$ *run g s1* $\leq$ *run h s2* $\Longrightarrow$ *refines f h s s2 R*
by *transfer* (*rule sim-post-state-le2*)

lemma *le-run-refines-trans*: *run f s* $\leq$ *run g s1* $\Longrightarrow$ *refines g h s1 s2 R* $\Longrightarrow$ *refines f h s s2 R*
by *transfer* (*rule sim-post-state-le1*)

lemma *refines-le-trans*: *refines f g s t R* $\Longrightarrow$ $g \leq h \Longrightarrow$ *refines f h s t R*
by (*auto simp*: *le-spec-monad-le-run-iff refines-le-run-trans*)

lemma *le-refines-trans*: $f \leq g \Longrightarrow$ *refines g h s t R* $\Longrightarrow$ *refines f h s t R*
by (*auto simp*: *le-spec-monad-le-run-iff intro*: *le-run-refines-trans*)

lemma *le-run-refines-iff*: *run f s* $\leq$ *run g t* $\longleftrightarrow$ *refines f g s t* (=)
by *transfer* (*simp add*: *sim-post-state-eq-iff-le*)

lemma *refines-top*[*simp*]: *refines f* $\top$ *s t R*
unfolding *top-spec-monad.rep-eq refines.rep-eq* **by** *simp*

lemma *refines-Sup1*: *refines* (*Sup F*) *g s s′ R* $\longleftrightarrow$ $(\forall f{\in}F.$ *refines f g s s′ R*$)$
by (*simp add*: *refines.rep-eq Sup-spec-monad.rep-eq image-image sim-post-state-Sup1*)

lemma *monotone-le-iff-refines*:
monotone R ($\leq$) *F* $\longleftrightarrow$ $(\forall x\ y.\ R\ x\ y \longrightarrow (\forall s.\ refines\ (F\ x)\ (F\ y)\ s\ s\ (=)))$
by (*auto simp*: *monotone-def le-fun-def le-run-refines-iff*[*symmetric*] *le-spec-monad-le-run-iff*)

lemma *refines-iff-runs-to*:
refines f g s t R $\longleftrightarrow$ $(\forall P.\ g \cdot t \{\!|\ \lambda r\ t'.\ \forall p\ s'.\ R\ (p,\ s')\ (r,\ t') \longrightarrow P\ p\ s'\ |\!\} \longrightarrow f \cdot s \{\!|\ P\ |\!\})$
by *transfer* (*auto simp*: *sim-post-state-iff split-beta′*)

lemma *runs-to-refines-weaken′*:
refines f g s t R $\Longrightarrow$ $g \cdot t \{\!|\ \lambda r\ t'.\ \forall p\ s'.\ R\ (p,\ s')\ (r,\ t') \longrightarrow P\ p\ s'\ |\!\} \Longrightarrow f \cdot s \{\!|\ P\ |\!\}$
by (*simp add*: *refines-iff-runs-to*)

lemma *runs-to-refines-weaken*: *refines f f′ s t* (=) ⟹ *f′* ⋅ *t* ⦃ *P* ⦄ ⟹ *f* ⋅ *s* ⦃ *P* ⦄
by (*auto simp*: *runs-to-refines-weaken′*)

lemma *refinesD-runs-to*:
assumes *f-g*: *refines f g s t R* **and** *g*: *g* ⋅ *t* ⦃ *P* ⦄
shows *f* ⋅ *s* ⦃ λ*r t*. ∃ *r′ t′*. *R* (*r*, *t*) (*r′*, *t′*) ∧ *P r′ t′* ⦄
by (*rule runs-to-refines-weaken′*[*OF f-g*]) (*auto intro*: *runs-to-weaken*[*OF g*])

lemma *rel-spec-iff-refines*:
rel-spec f g s t R ⟷ *refines f g s t R* ∧ *refines g f t s* (R^{-1-1})
unfolding *rel-spec-def* **by** *transfer* (*simp add*: *rel-post-state-eq-sim-post-state*)

lemma *rel-spec-symm*: *rel-spec f g s t R* ⟷ *rel-spec g f t s* R^{-1-1}
by (*simp add*: *rel-spec-iff-refines ac-simps*)

lemma *rel-specD-refines1*: *rel-spec f g s t R* ⟹ *refines f g s t R*
by (*simp add*: *rel-spec-iff-refines*)

lemma *rel-spec-mono*: *P* ≤ *Q* ⟹ *rel-spec f g s t P* ⟹ *rel-spec f g s t Q*
using *rel-post-state-mono* **by** (*auto simp*: *rel-spec-def*)

lemma *rel-spec-eq*: *rel-spec f g s t* (=) ⟷ *run f s* = *run g t*
by (*simp add*: *rel-spec-def rel-post-state-eq*)

lemma *rel-spec-eq-conv*: (∀ *s*. *rel-spec f g s s* (=)) ⟷ *f* = *g*
using *rel-spec-eq spec-monad-ext* **by** *metis*

lemma *rel-spec-eqD*: (⋀*s*. *rel-spec f g s s* (=)) ⟹ *f* = *g*
using *rel-spec-eq-conv* **by** *metis*

lemma *rel-spec-refl′*: *f* ⋅ *s* ?⦃ λ*r s*. *R* (*r*, *s*) (*r*, *s*)⦄ ⟹ *rel-spec f f s s R*
by *transfer* (*simp add*: *rel-post-state-refl′ split-beta′*)

lemma *rel-spec-refl*: (⋀*r s*. *R* (*r*, *s*) (*r*, *s*)) ⟹ *rel-spec f f s s R*
by (*rule rel-spec-mono*[*OF - rel-spec-eq*[*THEN iffD2*]]) *auto*

lemma *refines-same-runs-to-partialI*:
f ⋅ *s* ?⦃λ*r s′*. *R* (*r*, *s′*) (*r*, *s′*)⦄ ⟹ *refines f f s s R*
by *transfer*
(*auto intro*!: *sim-post-state-refl′ simp*: *split-beta′*)

lemma *refines-same-runs-toI*:
f ⋅ *s* ⦃λ*r s′*. *R* (*r*, *s′*) (*r*, *s′*)⦄ ⟹ *refines f f s s R*
by (*rule refines-same-runs-to-partialI*[*OF runs-to-partial-of-runs-to*])

lemma *always-progress-case-prod*[*always-progress-intros*]:
always-progress (*f* (*fst p*) (*snd p*)) ⟹ *always-progress* (*case-prod f p*)

by (*simp add*: *split-beta*)

lemma *refines-runs-to-partial-fuse*:
refines f f′ s s′ Q ⟹ *f* · *s* ?⦃*P*⦄ ⟹
refines f f′ s s′ (λ(*r*,*t*) (*r*′,*t*′). *Q* (*r*,*t*) (*r*′,*t*′) ∧ *P r t*)
by *transfer* (*auto elim*!: *sim-post-state.cases simp*: *sim-set-def split-beta*′)

lemma *runs-to-refines*:
f′ · *s*′ ⦃ *P* ⦄ ⟹ *refines f f*′ *s s*′ *Q* ⟹ (⋀*x s y t*. *P x s* ⟹ *Q* (*y*, *t*) (*x*, *s*) ⟹ *R y t*) ⟹
f · *s* ⦃ *R* ⦄
by *transfer* (*force elim*!: *sim-post-state.cases simp*: *sim-set-def split-beta*′)

lemma *runs-to-partial-subst-Res*: *f* · *s* ?⦃λ*r*. *P* (*the-Result r*)⦄ ⟷ *f* · *s* ?⦃λ*Res r*. *P r*⦄
by (*intro arg-cong*[**where** *f*=λ*P*. *f* · *s* ?⦃ *P* ⦄]) (*auto simp*: *fun-eq-iff the-Result-def*)

lemma *runs-to-subst-Res*: *f* · *s* ⦃λ*r*. *P* (*the-Result r*)⦄ ⟷ *f* · *s* ⦃λ*Res r*. *P r*⦄
by (*intro arg-cong*[**where** *f*=λ*P*. *f* · *s* ⦃ *P* ⦄]) (*auto simp*: *fun-eq-iff the-Result-def*)

lemma *runs-to-Res*[*simp*]: *f* · *s* ⦃λ*r t*. ∀*v*. *r* = *Result v* ⟶ *P v t*⦄ ⟷ *f* · *s* ⦃λ*Res v t*. *P v t*⦄
by (*intro arg-cong*[**where** *f*=λ*P*. *f* · *s* ⦃ *P* ⦄]) (*auto simp*: *fun-eq-iff the-Result-def*)

lemma *runs-to-partial-Res*[*simp*]:
f · *s* ?⦃λ*r t*. ∀*v*. *r* = *Result v* ⟶ *P v t*⦄ ⟷ *f* · *s* ?⦃λ*Res v t*. *P v t*⦄
by (*intro arg-cong*[**where** *f*=λ*P*. *f* · *s* ?⦃ *P* ⦄]) (*auto simp*: *fun-eq-iff the-Result-def*)

lemma *rel-spec-runs-to*:
assumes *f*: *f* · *s* ⦃ *P* ⦄ *always-progress f*
and *g*: *g* · *t* ⦃ *Q* ⦄ *always-progress g*
and *P*: (⋀*r s*′ *p t*′. *P r s*′ ⟹ *Q p t*′ ⟹ *R* (*r*, *s*′) (*p*, *t*′))
shows *rel-spec f g s t R*
using *assms* **unfolding** *rel-spec-def always-progress.rep-eq runs-to.rep-eq*
by (*cases run f s*; *cases run g t*; *simp add*: *rel-set-def split-beta*′)
(*metis all-not-in-conv bot-post-state-def prod.exhaust-sel*)

lemma *runs-to-res-independent-res*: *f* · *s* ⦃λ-. *P*⦄ ⟷ *f* · *s* ⦃λ*Res* -. *P*⦄
by (*rule runs-to-subst-Res*)

lemma *lift-state-spec-monad-eq*[*simp*]: *lift-state* (=) *p* = *p*
by *transfer* (*simp add*: *prod.rel-eq fun-eq-iff rel-post-state-eq*)

lemma *rel-post-state-Sup*:
rel-set (λ*x y*. *rel-post-state Q* (*f x*) (*g y*)) *X Y* ⟹ *rel-post-state Q* (⨆*x*∈*X*. *f x*) (⨆*x*∈*Y*. *g x*)
by (*force simp*: *rel-post-state-eq-sim-post-state rel-set-def intro*!: *sim-post-state-Sup*)

lemma *rel-set-Result-image-iff*:

rel-set (*rel-prod* (λ*Res v1 Res v2*. (*P v1 v2*)) *R*)
((λ*x*. *case x of* (*v*, *s*) ⇒ (*Result v*, *s*)) ‘ *Vals1*)
((λ*x*. *case x of* (*v*, *s*) ⇒ (*Result v*, *s*)) ‘ *Vals2*)
⟷
rel-set (*rel-prod P R*) *Vals1 Vals2*
by (*force simp add*: *rel-set-def*)

lemma *rel-post-state-converse-iff*:
rel-post-state R X Y ⟷ *rel-post-state* R^{-1-1} *Y X*
by (*metis conversep-conversep rel-post-state-eq-sim-post-state*)

lemma *runs-to-le-post-state-iff*: *runs-to f s Q* ⟷ *run f s* ≤ *Success* {(*r*, *s*). *Q r s*}
by (*cases run f s*) (*auto simp add*: *runs-to.rep-eq*)

lemma *runs-to-partial-runs-to-iff*: *runs-to-partial f s Q* ⟷ (*run f s* = *Failure* ∨ *runs-to f s Q*)
by (*cases run f s*; *simp add*: *runs-to-partial.rep-eq runs-to.rep-eq*)

lemma *run-runs-to-extensionality*:
run f s = *run g s* ⟷ (∀ *P*. *f* · *s* ⦇*P*⦈ ⟷ *g* · *s* ⦇*P*⦈)
apply *transfer*
apply (*simp add*: *post-state-eq-iff*)
apply (*rule all-cong-map*[**where** *f′*=λ*P* (*x*, *y*). *P x y* **and** *f*=λ*P x y*. *P* (*x*, *y*)])
apply *auto*
done

lemma *le-spec-monad-runI*: (⋀*s*. *run f s* ≤ *run g s*) ⟹ *f* ≤ *g*
by *transfer* (*simp add*: *le-fun-def*)

16.6.1 *rel-spec-monad*

lemma *rel-spec-monad-iff-rel-spec*:
rel-spec-monad R Q f g ⟷ (∀ *s t*. *R s t* ⟶ *rel-spec f g s t* (*rel-prod Q R*))
unfolding *rel-spec-monad-def rel-fun-def rel-spec.rep-eq* **..**

lemma *rel-spec-monadI*:
(⋀*s t*. *R s t* ⟹ *rel-spec f g s t* (*rel-prod Q R*)) ⟹ *rel-spec-monad R Q f g*
by (*auto simp*: *rel-spec-monad-iff-rel-spec*)

lemma *rel-spec-monadD*:
rel-spec-monad R Q f g ⟹ *R s t* ⟹ *rel-spec f g s t* (*rel-prod Q R*)
by (*auto simp*: *rel-spec-monad-iff-rel-spec*)

lemma *rel-spec-monad-eq-conv*: *rel-spec-monad* (=) (=) = (=)
unfolding *rel-spec-monad-def* **by** *transfer* (*simp add*: *fun-eq-iff prod.rel-eq rel-post-state-eq*)

lemma *rel-spec-monad-converse-iff*:
rel-spec-monad R Q f g ⟷ *rel-spec-monad* R^{-1-1} Q^{-1-1} *g f*

using *rel-post-state-converse-iff*
by (*metis conversep.cases conversep.intros prod.rel-conversep rel-spec-monad-def*)

lemma *rel-spec-monad-iff-refines*:
rel-spec-monad S R f g $\longleftrightarrow$
(∀ *s t*. *S s t* $\longrightarrow$ (*refines f g s t* (*rel-prod R S*) ∧ *refines g f t s* (*rel-prod* R^{-1-1} S^{-1-1})))
using *rel-spec-iff-refines rel-spec-monad-iff-rel-spec*
by (*metis prod.rel-conversep*)

lemma *rel-spec-monad-rel-exception-or-resultI*:
rel-spec-monad R (*rel-exception-or-result* (=) (=)) *f g* ⟹ *rel-spec-monad R* (=) *f g*
by (*simp add*: *rel-exception-or-result-sum-eq*)

lemma *runs-to-partial-runs-to-fuse*:
assumes *part*: *f* · *s* ?⦃*Q*⦄
assumes *tot*: *f* · *s* ⦃*P*⦄
shows *f* · *s* ⦃λ*r t*. *Q r t* ∧ *P r t*⦄
using *assms*
apply (*cases run f s*)
apply (*auto simp add*: *runs-to-def runs-to-partial-def*)
done

16.7 VCG basic setup

lemma *runs-to-cong-pred-only*:
P = *Q* ⟹ (*p* · *s* ⦃ *P* ⦄) $\longleftrightarrow$ (*p* · *s* ⦃ *Q* ⦄)
by *simp*

lemma *runs-to-cong-state-only*[*runs-to-vcg-cong-state-only*]:
s = *t* ⟹ (*p* · *s* ⦃ *Q* ⦄) $\longleftrightarrow$ (*p* · *t* ⦃ *Q* ⦄)
by *simp*

lemma *runs-to-partial-cong-state-only*[*runs-to-vcg-cong-state-only*]:
s = *t* ⟹ (*p* · *s* ?⦃ *Q* ⦄) $\longleftrightarrow$ (*p* · *t* ?⦃ *Q* ⦄)
by *simp*

lemma *runs-to-cong-program-only*[*runs-to-vcg-cong-program-only*]:
p = *q* ⟹ (*p* · *s* ⦃ *Q* ⦄) $\longleftrightarrow$ (*q* · *s* ⦃ *Q* ⦄)
by *simp*

lemma *runs-to-partial-cong-program-only*[*runs-to-vcg-cong-program-only*]:
p = *q* ⟹ (*p* · *s* ?⦃ *Q* ⦄) $\longleftrightarrow$ (*q* · *s* ?⦃ *Q* ⦄)
by *simp*

lemma *runs-to-case-conv*[*simp*]:
((*case* (*a*, *b*) *of* (*x*, *y*) ⇒ *f x y*) · *s* ⦃ *Q* ⦄) $\longleftrightarrow$ ((*f a b*) · *s* ⦃ *Q* ⦄)
by *simp*

lemma *runs-to-partial-case-conv*[*simp*]:
(($case$ $(a,\ b)$ of $(x,\ y) \Rightarrow f\ x\ y) \cdot s\ ?\{\!|\ Q\ |\!\}) \longleftrightarrow ((f\ a\ b) \cdot s\ ?\{\!|\ Q\ |\!\})$
by *simp*

lemma *always-progress-prod-case*[*always-progress-intros*]:
always-progress $(f\ (fst\ p)\ (snd\ p)) \Longrightarrow$ *always-progress* $(case\ p\ of\ (x,\ y) \Rightarrow f\ x\ y)$
by (*auto simp add*: *always-progress-def split*: *prod.splits*)

lemma *runs-to-conj*:
$(f \cdot s \{\!|\ \lambda r\ s.\ P\ r\ s \wedge Q\ r\ s |\!\}) \longleftrightarrow (f \cdot s \{\!|\ \lambda r\ s.\ P\ r\ s\ |\!\}) \wedge (f \cdot s \{\!|\ \lambda r\ s.\ Q\ r\ s |\!\})$
by (*cases run f s*) (*auto simp*: *runs-to.rep-eq*)

lemma *runs-to-all*:
$(f \cdot s \{\!|\ \lambda r\ s.\ \forall x.\ P\ x\ r\ s\ |\!\}) \longleftrightarrow (\forall x.\ f \cdot s \{\!|\ \lambda r\ s.\ P\ x\ r\ s\ |\!\})$
by (*cases run f s*) (*auto simp*: *runs-to.rep-eq*)

lemma *runs-to-imp-const*:
$(f \cdot s \{\!|\ \lambda r\ s.\ P\ r\ s \longrightarrow Q\ |\!\}) \longleftrightarrow (Q \wedge (f \cdot s \{\!|\lambda\text{- -. } True\ |\!\})) \vee (\neg\ Q \wedge (f \cdot s \{\!|$
$\lambda r\ s.\ \neg\ P\ r\ s |\!\}))$
by (*cases run f s*) (*auto simp*: *runs-to.rep-eq*)

16.7.1 *res-monad* and *exn-monad* Types

type-synonym $'a\ val = (unit,\ 'a)\ exception\text{-}or\text{-}result$
type-synonym $('e,\ 'a)\ xval = ('e\ option,\ 'a)\ exception\text{-}or\text{-}result$
type-synonym $('a,\ 's)\ res\text{-}monad = (unit,\ 'a\ ,\ 's)\ spec\text{-}monad$
type-synonym $('e,\ 'a,\ 's)\ exn\text{-}monad = ('e\ option,\ 'a,\ 's)\ spec\text{-}monad$

definition $Exn :: 'e \Rightarrow ('e,\ 'a)\ xval$ **where**
$Exn\ e = Exception\ (Some\ e)$

definition
$case\text{-}xval :: ('e \Rightarrow 'a) \Rightarrow ('v \Rightarrow 'a) \Rightarrow ('e,\ 'v)\ xval \Rightarrow 'a$ **where**
case-xval f g x =
(*case x of*
Exception v ⇒ (*case v of Some e* ⇒ *f e* | *None* ⇒ *undefined*)
| *Result v* ⇒ *g v*)

declare [[*case-translation case-xval Exn Result*]]

inductive *rel-xval*:: $('e \Rightarrow 'f \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow$
$('e,\ 'a)\ xval \Rightarrow ('f,\ 'b)\ xval \Rightarrow bool$
where
Exn: $E\ e\ f \Longrightarrow rel\text{-}xval\ E\ R\ (Exn\ e)\ (Exn\ f)$ |
Result: $R\ a\ b \Longrightarrow rel\text{-}xval\ E\ R\ (Result\ a)\ (Result\ b)$

definition $map\text{-}xval :: ('e \Rightarrow 'f) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('e,\ 'a)\ xval \Rightarrow ('f,\ 'b)\ xval$ **where**
$map\text{-}xval\ f\ g\ x \equiv case\ x\ of\ Exn\ e \Rightarrow Exn\ (f\ e) \mid Result\ v \Rightarrow Result\ (g\ v)$

lemma *rel-xval-eq*: *rel-xval* (=) (=) = (=)
apply (*rule ext*)+
subgoal for *x y*
apply (*cases x*)
apply (*auto simp add*: *rel-xval.simps default-option-def Exn-def*)
done
done

lemma *rel-xval-rel-exception-or-result-conv*:
rel-xval E R = *rel-exception-or-result* (*rel-map the OO E OO rel-map Some*) *R*
apply (*rule ext*)+
subgoal for *x y*
apply (*cases x*)
subgoal
by (*auto simp add*: *rel-exception-or-result.simps rel-xval.simps Exn-def default-option-def rel-map-def*)
(*metis option.sel rel-map-direct relcomppI*)+
subgoal
by (*auto simp add*: *rel-exception-or-result.simps rel-xval.simps Exn-def default-option-def rel-map-def*)
done
done

lemmas *rel-xval-Exn* = *rel-xval.intros*(*1*)
lemmas *rel-xval-Result* = *rel-xval.intros*(*2*)

lemma *case-xval-simps*[*simp*]:
case-xval f g (*Exn v*) = *f v*
case-xval f g (*Result e*) = *g e*
by (*auto simp add*: *case-xval-def Exn-def*)

lemma *case-exception-or-result-Exn*[*simp*]:
case-exception-or-result f g (*Exn x*) = *f* (*Some x*)
by (*simp add*: *Exn-def*)

lemma *xval-split*:
P (*case-xval f g r*) ⟷
(∀ *e*. *r* = *Exn e* ⟶ *P* (*f e*)) ∧
(∀ *v*. *r* = *Result v* ⟶ *P* (*g v*))
by (*auto simp add*: *case-xval-def Exn-def split*: *exception-or-result-splits option.splits*)

lemma *xval-split-asm*:
P (*case-xval f g r*) ⟷
¬ ((∃ *e*. *r* = *Exn e* ∧ ¬ *P* (*f e*)) ∨
(∃ *v*. *r* = *Result v* ∧ ¬ *P* (*g v*)))
by (*auto simp add*: *case-xval-def Exn-def split*: *exception-or-result-splits option.splits*)

lemmas *xval-splits* = *xval-split xval-split-asm*

lemma *Exn-eq-Exn*[*simp*]: *Exn x = Exn y* ⟷ *x = y*
by (*simp add*: *Exn-def*)

lemma *Exn-neq-Result*[*simp*]: *Exn x = Result e* ⟷ *False*
by (*simp add*: *Exn-def*)

lemma *Result-neq-Exn*[*simp*]: *Result e = Exn x* ⟷ *False*
by (*simp add*: *Exn-def*)

lemma *Exn-eq-Exception*[*simp*]:
Exn x = Exception a ⟷ *a = Some x*
Exception a = Exn x ⟷ *a = Some x*
by (*auto simp*: *Exn-def*)

lemma *map-exception-or-result-Exn*[*simp*]:
(⋀*x*. *f* (*Some x*) ≠ *None*) ⟹ *map-exception-or-result f g* (*Exn x*) = *Exn* (*the* (*f* (*Some x*)))
by (*simp add*: *Exn-def*)

lemma *case-xval-Exception-Some-simp*[*simp*]: (*case Exception* (*Some y*) *of Exn e* ⇒ *f e* | *Result v* ⇒ *g v*) = *f y*
by (*metis Exn-def case-xval-simps*(*1*))

lemma *rel-xval-simps*[*simp*]:
rel-xval E R (*Exn e*) (*Exn f*) = *E e f*
rel-xval E R (*Result v*) (*Result w*) = *R v w*
rel-xval E R (*Exn e*) (*Result w*) = *False*
rel-xval E R (*Result v*) (*Exn f*) = *False*
by (*auto simp add*: *rel-xval.simps*)

lemma *map-xval-simps*[*simp*]:
map-xval f g (*Exn e*) = *Exn* (*f e*)
map-xval f g (*Result v*) = *Result* (*g v*)
by (*auto simp add*: *map-xval-def*)

lemma *map-xval-Exn*: *map-xval f g x = Exn y* ⟷ (∃ *e*. *x = Exn e* ∧ *y = f e*)
by (*auto simp add*: *map-xval-def split*: *xval-splits*)

lemma *map-xval-Result*: *map-xval f g x = Result y* ⟷ (∃ *v*. *x = Result v* ∧ *y = g v*)
by (*auto simp add*: *map-xval-def split*: *xval-splits*)

lemma *Result-unit-eq*: (*x*:: *unit val*) = *Result* ()
by (*cases x*) *auto*

simproc-setup *Result-unit-eq* (*x*::(*unit val*)) = ‹
let
fun is-Result-unit ***Const*** ‹*Result* ‹@{*typ unit*}› ‹@{*typ unit*}› *for* ***Const*** ‹*Product-Type.Unity*››

```
= true
    | is-Result-unit - = false
  in
    K (K (fn ct =>
      if is-Result-unit (Thm.term-of ct) then NONE
      else SOME (mk-meta-eq @{thm Result-unit-eq})))
  end
›

lemma ex-val-Result1:
  ∃ v1. (x::'v1 val) = Result v1
  by (cases x) auto

lemma ex-val-Result2:
  ∃ v1 v2. (x::('v1 * 'v2) val) = Result (v1, v2)
  by (cases x) auto

lemma ex-val-Result3:
  ∃ v1 v2 v3. (x::('v1 * 'v2 * 'v3) val) = Result (v1, v2, v3)
  by (cases x) auto

lemma ex-val-Result4:
  ∃ v1 v2 v3 v4. (x::('v1 * 'v2 * 'v3 * 'v4) val) = Result (v1, v2, v3, v4)
  by (cases x) auto

lemma ex-val-Result5:
  ∃ v1 v2 v3 v4 v5. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5) val) = Result (v1, v2, v3, v4,
v5)
  by (cases x) auto

lemma ex-val-Result6:
  ∃ v1 v2 v3 v4 v5 v6. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5  * 'v6) val) = Result (v1,
v2, v3, v4, v5, v6)
  by (cases x) auto

lemma ex-val-Result7:
  ∃ v1 v2 v3 v4 v5 v6 v7. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5  * 'v6 * 'v7) val) = Result
(v1, v2, v3, v4, v5, v6, v7)
  by (cases x) auto

lemma ex-val-Result8:
  ∃ v1 v2 v3 v4 v5 v6 v7 v8. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5  * 'v6 * 'v7 * 'v8) val)
= Result (v1, v2, v3, v4, v5, v6, v7, v8)
  by (cases x) auto

lemma ex-val-Result9:
  ∃ v1 v2 v3 v4 v5 v6 v7 v8 v9. (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5  * 'v6 * 'v7 * 'v8 *
'v9) val) = Result (v1, v2, v3, v4, v5, v6, v7, v8, v9)
  by (cases x) auto
```

lemma *ex-val-Result10*:
∃ *v1 v2 v3 v4 v5 v6 v7 v8 v9 v10*. (*x*::(*'v1* ∗ *'v2* ∗ *'v3* ∗ *'v4* ∗ *'v5* ∗ *'v6* ∗ *'v7* ∗ *'v8* ∗ *'v9* ∗ *'v10*) *val*) = *Result* (*v1*, *v2*, *v3*, *v4*, *v5*, *v6*, *v7*, *v8*, *v9*, *v10*)
by (*cases x*) *auto*

lemma *ex-val-Result11*:
∃ *v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11*. (*x*::(*'v1* ∗ *'v2* ∗ *'v3* ∗ *'v4* ∗ *'v5* ∗ *'v6* ∗ *'v7* ∗ *'v8* ∗ *'v9* ∗ *'v10* ∗ *'v11*) *val*) = *Result* (*v1*, *v2*, *v3*, *v4*, *v5*, *v6*, *v7*, *v8*, *v9*, *v10*, *v11*)
by (*cases x*) *auto*

lemmas *ex-val-Result*[*simp*] =
ex-val-Result1
ex-val-Result2
ex-val-Result3
ex-val-Result4
ex-val-Result5
ex-val-Result6
ex-val-Result7
ex-val-Result8
ex-val-Result9
ex-val-Result10
ex-val-Result11

lemma *all-val-imp-iff*: (∀ *v*. (*r*::*'a val*) = *Result v* ⟶ *P*) ⟷ *P*
by (*cases r*) *auto*

definition *map-exn* :: (*'e* ⇒ *'f*) ⇒ (*'e*, *'a*) *xval* ⇒ (*'f*, *'a*) *xval* **where**
map-exn f x =
(*case x of*
Exn e ⇒ *Exn* (*f e*)
| *Result v* ⇒ *Result v*)

lemma *map-exn-simps*[*simp*]:
map-exn f (*Exn e*) = *Exn* (*f e*)
map-exn f (*Result v*) = *Result v*
map-exn f x = *Result v* ⟷ *x* = *Result v*
by (*auto simp add*: *map-exn-def split*: *xval-splits*)

lemma *map-exn-id*[*simp*]: *map-exn* (λ*x*. *x*) = (λ*x*. *x*)
by (*auto simp add*: *map-exn-def split*: *xval-splits*)

definition *unnest-exn* :: (*'e* + *'a*, *'a*) *xval* ⇒ (*'e*, *'a*) *xval* **where**
unnest-exn x =
(*case x of*
Exn e ⇒ (*case e of Inl l* ⇒ *Exn l* | *Inr r* ⇒ *Result r*)

| *Result v* ⇒ *Result v*)

lemma *unnest-exn-simps*[*simp*]:
unnest-exn (*Exn* (*Inl l*)) = *Exn l*
unnest-exn (*Exn* (*Inr r*)) = *Result r*
unnest-exn (*Result v*) = *Result v*
by (*auto simp add*: *unnest-exn-def split*: *xval-splits*)

lemma *unnest-exn-eq-simps*[*simp*]:
unnest-exn (*Result r*) = *Result r'* ⟷ *r* = *r'*
unnest-exn (*Result e*) ≠ *Exn e*
unnest-exn (*Exn e*) = *Result r* ⟷ *e* = *Inr r*
unnest-exn (*Exn e*) = *Exn e'* ⟷ *e* = *Inl e'*
by (*auto simp*: *unnest-exn-def split*: *xval-splits sum.splits*)

definition *the-Exn* :: ('*e*, '*a*) *xval* ⇒ '*e* **where**
the-Exn x ≡ (*case x of Exn e* ⇒ *e* | *Result* - ⇒ *undefined*)

abbreviation *the-Res* :: '*a val* ⇒ '*a* **where** *the-Res* ≡ *the-Result*

lemma *is-Exception-val*[*simp*]: *is-Exception* (*x*::'*a val*) = *False*
by (*auto simp add*: *is-Exception-def*)

lemma *is-Exception-Exn*[*simp*]: *is-Exception* (*Exn x*)
by (*simp add*: *Exn-def*)

lemma *is-Result-val*[*simp*]: *is-Result* (*v*::'*a val*)
by (*auto simp add*: *is-Result-def*)

lemma *the-Exception-Exn*[*simp*]: *the-Exception* (*Exn e*) = *Some e*
by (*auto simp add*: *the-Exception-def*)

lemma *the-Exn-Exn*[*simp*]:
the-Exn (*Exn e*) = *e*
the-Exn (*Exception* (*Some e*)) = *e*
by (*auto simp add*: *the-Exn-def case-xval-def*)

lemma *the-Result-simp*[*simp*]:
the-Result (*Result v*) = *v*
by (*auto simp add*: *the-Result-def*)

lemma *Result-the-Result-val*[*simp*]: *Result* (*the-Result* (*x*::'*a val*)) = *x*
by (*auto simp add*: *the-Result-def*)

lemma *rel-exception-or-result-val-apply*:
fixes *x*::'*a val*
fixes *y* ::'*b val*
shows *rel-exception-or-result E R x y* ⟷ *rel-exception-or-result E' R x y*
by (*auto simp add*: *rel-exception-or-result.simps*)

lemma *rel-exception-or-result-val*:
shows ((*rel-exception-or-result E R*)::'*a val* ⇒ '*b val* ⇒ *bool*) = *rel-exception-or-result E' R*
apply (*rule ext*)+
apply (*auto simp add*: *rel-exception-or-result.simps*)
done

lemma *rel-exception-or-result-Res-val*:
(λ*Res x Res y*. *R x y*) = *rel-exception-or-result* (λ- -. *False*) *R*
apply (*rule ext*)+
apply *auto*
done

definition *unite*:: ('*a*, '*a*) *xval* ⇒ '*a val* **where**
unite x = (*case x of Exn v* ⇒ *Result v* | *Result v* ⇒ *Result v*)

lemma *unite-simps*[*simp*]:
unite (*Exn v*) = *Result v*
unite (*Result v*) = *Result v*
by (*auto simp add*: *unite-def*)

lemma *val-exhaust*: (⋀*v*. (*x*::'*a val*) = *Result v* ⟹ *P*) ⟹ *P*
by (*cases x*) *auto*

lemma *val-iff*: *P* (*x*::'*a val*) ⟷ (∃ *v*. *x* = *Result v* ∧ *P* (*Result v*))
by (*cases x*) *auto*

lemma *val-nchotomy*: ∀ (*x*::'*a val*). ∃ *v*. *x* = *Result v*
by (*meson val-exhaust*)

lemma (⋀*x*::'*a val*. *PROP P x*) ⟹ *PROP P* (*Result* (*v*::'*a*))
by (*rule meta-spec*)

lemma *val-Result-the-Result-conv*: (*x*::'*a val*) = *Result* (*the-Result x*)
by (*cases x*) (*auto simp add*: *the-Result-def*)

lemma *split-val-all*[*simp*]: (⋀*x*::'*a val*. *PROP P x*) ≡ (⋀*v*::'*a*. *PROP P* (*Result v*))
proof
fix *v*::'*a*
assume ⋀*x*::'*a val*. *PROP P x*
then show *PROP P* (*Result v*) **.**
next
fix *x*::'*a val*
assume ⋀*v*::'*a*. *PROP P* (*Result v*)
hence *PROP P* (*Result* (*the-Result x*)) **.**
thus *PROP P x*
apply (*subst val-Result-the-Result-conv*)

apply (*assumption*)
done
qed

lemma *split-val-Ex*[*simp*]: (∃ (*x*::'*a val*). *P x*) ⟷ (∃ (*v*::'*a*). *P* (*Result v*))
by (*auto*)

lemma *split-val-All*[*simp*]: (∀ (*x*::'*a val*). *P x*) ⟷ (∀ (*v*::'*a*). *P* (*Result v*))
by (*auto*)

definition *to-xval* :: ('*e* + '*a*) ⇒ ('*e*, '*a*) *xval* **where**
to-xval x = (*case x of Inl e* ⇒ *Exn e* | *Inr v* ⇒ *Result v*)

lemma *to-xval-simps*[*simp*]:
to-xval (*Inl e*) = *Exn e*
to-xval (*Inr v*) = *Result v*
by (*auto simp add*: *to-xval-def*)

lemma *to-xval-Result-iff*[*simp*]: *to-xval x* = *Result v* ⟷ *x* = *Inr v*
apply (*cases x*)
apply (*auto simp add*: *to-xval-def default-option-def*)
done

lemma *to-xval-Exn-iff*[*simp*]:
to-xval x = *Exn v* ⟷
(*x* = *Inl v*)
apply (*cases x*)
apply (*auto simp add*: *to-xval-def default-option-def Exn-def*)
done

lemma *to-xval-Exception-iff*[*simp*]:
to-xval x = *Exception v* ⟷
((*v* = *None* ∧ *x* = *Inr undefined*) ∨ (∃ *e*. *v* = *Some e* ∧ *x* = *Inl e*))
apply (*cases x*)
apply (*auto simp add*: *to-xval-def default-option-def Exn-def*)
done

definition *from-xval* :: ('*e*, '*a*) *xval* ⇒ ('*e* + '*a*) **where**
from-xval x = (*case x of Exn e* ⇒ *Inl e* | *Result v* ⇒ *Inr v*)

lemma *from-xval-simps*[*simp*]:
from-xval (*Exn e*) = *Inl e*
from-xval (*Result v*) = *Inr v*
by (*auto simp add*: *from-xval-def*)

lemma *from-xval-Inr-iff*[*simp*]: *from-xval x* = *Inr v* ⟷ *x* = *Result v*
apply (*cases x*)
apply (*auto simp add*: *from-xval-def split*: *xval-splits*)

done

lemma *from-xval-Inl-iff*[*simp*]: *from-xval x* = *Inl e* ⟷ *x* = *Exn e*
apply (*cases x*)
apply (*auto simp add*: *from-xval-def split*: *xval-splits*)
done

lemma *to-xval-from-xval*[*simp*]: *to-xval* (*from-xval x*) = *x*
apply (*cases x*)
apply (*auto simp add*: *Exn-def default-option-def*)
done

lemma *from-xval-to-xval*[*simp*]: *from-xval* (*to-xval x*) = *x*
apply (*cases x*)
apply (*auto simp add*: *Exn-def default-option-def*)
done

lemma *rel-map-to-xval-Exn-iff*[*simp*]: *rel-map to-xval y* (*Exn e*) ⟷ *y* = *Inl e*
by (*auto simp add*: *rel-map-def to-xval-def split*: *sum.splits*)

lemma *rel-map-to-xval-Inr-iff*[*simp*]: *rel-map to-xval* (*Inr r*) *x* ⟷ *x* = *Result r*
by (*auto simp add*: *rel-map-def*)

lemma *rel-map-to-xval-Inl-iff*[*simp*]: *rel-map to-xval* (*Inl l*) *x* ⟷ *x* = *Exn l*
by (*auto simp add*: *rel-map-def*)

lemma *rel-map-to-xval-Result-iff*[*simp*]: *rel-map to-xval y* (*Result r*) ⟷ *y* = *Inr r*
by (*auto simp add*: *rel-map-def to-xval-def split*: *sum.splits*)

lemma *rel-map-from-xval-Exn-iff*[*simp*]: *rel-map from-xval* (*Exn l*) *x* ⟷ *x* = *Inl l*
by (*auto simp add*: *rel-map-def*)

lemma *rel-map-from-xval-Inl-iff*[*simp*]: *rel-map from-xval y* (*Inl e*) ⟷ *y* = *Exn e*
by (*auto simp add*: *rel-map-def from-xval-def split*: *xval-splits*)

lemma *rel-map-from-xval-Result-iff*[*simp*]: *rel-map from-xval* (*Result r*) *x* ⟷ *x* = *Inr r*
by (*auto simp add*: *rel-map-def*)

lemma *rel-map-from-xval-Inr-iff*[*simp*]: *rel-map from-xval y* (*Inr r*) ⟷ *y* = *Result r*
by (*auto simp add*: *rel-map-def from-xval-def split*: *xval-splits*)

16.8 *res-monad* and *exn-monad* functions

abbreviation *throw e* ≡ *yield* (*Exn e*)

definition *try* :: (*'e* + *'a*, *'a*, *'s*) *exn-monad* ⇒ (*'e*, *'a*, *'s*) *exn-monad* **where**
try = *map-value unnest-exn*

definition *finally* :: (*'a*, *'a*, *'s*) *exn-monad* ⇒ (*'a*, *'s*) *res-monad* **where**
finally = *map-value unite*

definition
catch :: (*'e*, *'a*, *'s*) *exn-monad* ⇒
(*'e* ⇒ (*'f*::*default*, *'a*, *'s*) *spec-monad*) ⇒
(*'f*::*default*, *'a*, *'s*) *spec-monad* (**infix** ‹<*catch*>› *10*)
where
f <*catch*> *handler* ≡ *bind-handle f return* (*handler o the*)

abbreviation *bind-finally* ::
(*'e*, *'a*, *'s*) *exn-monad* ⇒ ((*'e*, *'a*) *xval* ⇒ (*'b*, *'s*) *res-monad*) ⇒ (*'b*, *'s*) *res-monad*
where
bind-finally ≡ *bind-exception-or-result*

definition *ignoreE* ::(*'e*, *'a*, *'s*) *exn-monad* ⇒ (*'a*, *'s*) *res-monad* **where**
ignoreE f = *catch f* (λ-. *bot*)

definition *liftE* :: (*'a*, *'s*) *res-monad* ⇒ (*'e*, *'a*, *'s*) *exn-monad* **where**
liftE = *map-value* (*map-exception-or-result* (λ*x*. *undefined*) *id*)

definition *check*:: *'e* ⇒ (*'s* ⇒ *'a* ⇒ *bool*) ⇒ (*'e*, *'a*, *'s*) *exn-monad* **where**
check e p =
condition (λ*s*. ∃ *x*. *p s x*)
(*do* { *s* ← *get-state*; *select* {*x*. *p s x*} })
(*throw e*)

abbreviation *check'* :: *'e* ⇒ (*'s* ⇒ *bool*) ⇒ (*'e*, *unit*, *'s*) *exn-monad* **where**
check' e p ≡ *check e* (λ*s* -. *p s*)

16.9 Monad operations

16.9.1 ⊤

declare *top-spec-monad.rep-eq*[*run-spec-monad*, *simp*]

lemma *always-progress-top*[*always-progress-intros*]: *always-progress* ⊤
by *transfer simp*

lemma *runs-to-top*[*simp*]: ⊤ ⋅ *s* ⦃ *Q* ⦄ ⟷ *False*
by *transfer simp*

lemma *runs-to-partial-top*[*simp*]: ⊤ ⋅ *s* ?⦃ *Q* ⦄ ⟷ *True*
by *transfer simp*

16.9.2 $\bot$

declare *bot-spec-monad.rep-eq*[*run-spec-monad*, *simp*]

lemma *always-progress-bot-iff*[*iff*]: *always-progress* $\bot \longleftrightarrow$ *False*
by *transfer simp*

lemma *runs-to-bot*[*simp*]: $\bot \cdot s \{\!|\ Q\ |\!\} \longleftrightarrow$ *True*
by *transfer simp*

lemma *runs-to-partial-bot*[*simp*]: $\bot \cdot s\ ?\{\!|\ Q\ |\!\} \longleftrightarrow$ *True*
by *transfer simp*

16.9.3 *fail*

lemma *always-progress-fail*[*always-progress-intros*]: *always-progress fail*
by *transfer* (*simp add*: *bot-post-state-def*)

lemma *run-fail*[*run-spec-monad*, *simp*]: *run fail s* $= \top$
by *transfer* (*simp add*: *top-post-state-def*)

lemma *runs-to-fail*[*simp*]: *fail* $\cdot$ *s* $\{\!|\ R\ |\!\} \longleftrightarrow$ *False*
by *transfer simp*

lemma *runs-to-partial-fail*[*simp*]: *fail* $\cdot$ *s* $?\{\!|\ R\ |\!\} \longleftrightarrow$ *True*
by *transfer simp*

lemma *refines-fail*[*simp*]: *refines f fail s t R*
by (*simp add*: *refines.rep-eq*)

lemma *rel-spec-fail*[*simp*]: *rel-spec fail fail s t R*
by (*simp add*: *rel-spec-def*)

16.9.4 *yield*

lemma *run-yield*[*run-spec-monad*, *simp*]: *run* (*yield r*) *s* $=$ *pure-post-state* (*r*, *s*)
by *transfer simp*

lemma *always-progress-yield*[*always-progress-intros*]: *always-progress* (*yield r*)
by (*simp add*: *always-progress-def*)

lemma *yield-inj*[*simp*]: *yield x* $=$ *yield y* $\longleftrightarrow$ *x* $=$ *y*
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *runs-to-yield-iff*[*simp*]: ((*yield r*) $\cdot$ *s* $\{\!|\ Q\ |\!\}) \longleftrightarrow$ *Q r s*
by *transfer simp*

lemma *runs-to-yield*[*runs-to-vcg*]: *Q r s* $\Longrightarrow$ *yield r* $\cdot$ *s* $\{\!|\ Q\ |\!\}$
by *simp*

lemma *runs-to-partial-yield-iff*[*simp*]: $((yield\ r) \cdot s\ ?\{\!|\ Q\ |\!\}) \longleftrightarrow Q\ r\ s$
by *transfer simp*

lemma *runs-to-partial-yield*[*runs-to-vcg*]: $Q\ r\ s \Longrightarrow yield\ r \cdot s\ ?\{\!|\ Q\ |\!\}$
by *simp*

lemma *refines-yield-iff*[*simp*]: *refines* $(yield\ r)\ (yield\ r')\ s\ s'\ R \longleftrightarrow R\ (r,\ s)\ (r',\ s')$
by *transfer simp*

lemma *refines-yield*: $R\ (a,\ s)\ (b,\ t) \Longrightarrow$ *refines* $(yield\ a)\ (yield\ b)\ s\ t\ R$
by *simp*

lemma *refines-yield-right-iff*:
refines $f\ (yield\ e)\ s\ t\ R \longleftrightarrow (f \cdot s\ \{\!|\ \lambda r\ s'.\ R\ (r,\ s')\ (e,\ t)\ |\!\})$
by (*auto simp*: *refines.rep-eq runs-to.rep-eq sim-post-state-pure-post-state2 split-beta'*)

lemma *rel-spec-yield*: $R\ (a,\ s)\ (b,\ t) \Longrightarrow$ *rel-spec* $(yield\ a)\ (yield\ b)\ s\ t\ R$
by (*simp add*: *rel-spec-def*)

lemma *rel-spec-yield-iff*[*simp*]: *rel-spec* $(yield\ x)\ (yield\ y)\ s\ t\ Q \longleftrightarrow Q\ (x,\ s)\ (y,\ t)$
by (*simp add*: *rel-spec.rep-eq*)

lemma *rel-spec-monad-yield*: $Q\ x\ y \Longrightarrow$ *rel-spec-monad* $R\ Q\ (yield\ x)\ (yield\ y)$
by (*auto simp add*: *rel-spec-monad-iff-rel-spec*)

16.9.5 *throw-exception-or-result*

lemma *throw-exception-or-result-bind*[*simp*]:
$e \neq default \Longrightarrow$ *throw-exception-or-result* $e >>= f =$ *throw-exception-or-result* e
unfolding *bind-def* **by** *transfer auto*

16.9.6 *throw*

lemma *throw-bind*[*simp*]: *throw* $e >>= f =$ *throw* e
by (*simp add*: *Exn-def*)

16.9.7 *get-state*

lemma *always-progress-get-state*[*always-progress-intros*]: *always-progress get-state*
by *transfer simp*

lemma *run-get-state*[*run-spec-monad*, *simp*]: *run get-state* $s =$ *pure-post-state* (*Result* $s,\ s$)
by *transfer simp*

lemma *runs-to-get-state*[*runs-to-vcg*]: *get-state* $\cdot\ s\ \{\!|\lambda r\ t.\ r = Result\ s \wedge t = s|\!\}$
by *transfer simp*

lemma *runs-to-get-state-iff*[*runs-to-iff*]: *get-state* · *s* ⦃*Q*⦄ ⟷ (*Q* (*Result s*) *s*)
by *transfer simp*

lemma *runs-to-partial-get-state*[*runs-to-vcg*]: *get-state* · *s* ?⦃λ*r t*. *r* = *Result s* ∧ *t* = *s*⦄
by *transfer simp*

lemma *refines-get-state*: *R* (*Result s*, *s*) (*Result t*, *t*) ⟹ *refines get-state get-state s t R*
by *transfer simp*

lemma *rel-spec-get-state*: *R* (*Result s*, *s*) (*Result t*, *t*) ⟹ *rel-spec get-state get-state s t R*
by (*auto simp*: *rel-spec-iff-refines intro*: *refines-get-state*)

16.9.8 *set-state*

lemma *always-progress-set-state*[*always-progress-intros*]: *always-progress* (*set-state t*)
by *transfer simp*

lemma *set-state-inj*[*simp*]: *set-state x* = *set-state y* ⟷ *x* = *y*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *run-set-state*[*run-spec-monad*, *simp*]: *run* (*set-state t*) *s* = *pure-post-state* (*Result* (), *t*)
by *transfer simp*

lemma *runs-to-set-state*[*runs-to-vcg*]: *set-state t* · *s* ⦃λ*r t*′. *t*′ = *t*⦄
by *transfer simp*

lemma *runs-to-set-state-iff*[*runs-to-iff*]: *set-state t* · *s* ⦃*Q*⦄ ⟷ *Q* (*Result* ()) *t*
by *transfer simp*

lemma *runs-to-partial-set-state*[*runs-to-vcg*]: *set-state t* · *s* ?⦃λ*r t*′. *t*′ = *t*⦄
by *transfer simp*

lemma *refines-set-state*:
R (*Result* (), *s*′) (*Result* (), *t*′) ⟹ *refines* (*set-state s*′) (*set-state t*′) *s t R*
by *transfer simp*

lemma *rel-spec-set-state*:
R (*Result* (), *s*′) (*Result* (), *t*′) ⟹ *rel-spec* (*set-state s*′) (*set-state t*′) *s t R*
by (*auto simp add*: *rel-spec-iff-refines intro*: *refines-set-state*)

16.9.9 *select*

lemma *always-progress-select*[*always-progress-intros*]: *S* ≠ {} ⟹ *always-progress* (*select S*)
unfolding *select-def* **by** *transfer simp*

lemma *runs-to-select*[*runs-to-vcg*]: $(\bigwedge x.\ x \in S \Longrightarrow Q\ (Result\ x)\ s) \Longrightarrow select\ S \cdot s \ \{\!|\ Q\ |\!\}$
unfolding *select-def* **by** *transfer simp*

lemma *runs-to-select-iff*[*runs-to-iff*]: $select\ S \cdot s\ \{\!|Q|\!\} \longleftrightarrow (\forall x{\in}S.\ Q\ (Result\ x)\ s)$
unfolding *select-def* **by** *transfer auto*

lemma *runs-to-partial-select*[*runs-to-vcg*]: $(\bigwedge x.\ x \in S \Longrightarrow Q\ (Result\ x)\ s) \Longrightarrow select\ S \cdot s\ ?\{\!|\ Q\ |\!\}$
using *runs-to-select* **by** (*rule runs-to-partial-of-runs-to*)

lemma *run-select*[*run-spec-monad*, *simp*]: $run\ (select\ S)\ s = Success\ ((\lambda v.\ (Result\ v,\ s))\ `\ S)$
unfolding *select-def* **by** *transfer* (*auto simp add*: *image-image pure-post-state-def Sup-Success*)

lemma *refines-select*:
$(\bigwedge x.\ x \in P \Longrightarrow \exists xa{\in}Q.\ R\ (Result\ x,\ s)\ (Result\ xa,\ t)) \Longrightarrow refines\ (select\ P)\ (select\ Q)\ s\ t\ R$
unfolding *select-def* **by** *transfer* (*auto intro*!: *sim-post-state-Sup*)

lemma *rel-spec-select*:
$rel\text{-}set\ (\lambda a\ b.\ R\ (Result\ a,\ s)\ (Result\ b,\ t))\ P\ Q \Longrightarrow rel\text{-}spec\ (select\ P)\ (select\ Q)\ s\ t\ R$
by (*auto simp add*: *rel-spec-def rel-set-def*)

16.9.10 *unknown*

lemma *runs-to-unknown*[*runs-to-vcg*]: $(\bigwedge x.\ Q\ (Result\ x)\ s) \Longrightarrow unknown \cdot s \ \{\!|\ Q\ |\!\}$
by (*simp add*: *unknown-def runs-to-select-iff*)

lemma *runs-to-unknown-iff*[*runs-to-iff*]: $unknown \cdot s \ \{\!|\ Q\ |\!\} \longleftrightarrow (\forall x.\ Q\ (Result\ x)\ s)$
by (*simp add*: *unknown-def runs-to-select-iff*)

lemma *runs-to-partial-unknown*[*runs-to-vcg*]: $(\bigwedge x.\ Q\ (Result\ x)\ s) \Longrightarrow unknown \cdot s\ ?\{\!|\ Q\ |\!\}$
using *runs-to-unknown* **by** (*rule runs-to-partial-of-runs-to*)

lemma *run-unknown*[*run-spec-monad*, *simp*]: $run\ unknown\ s = Success\ ((\lambda v.\ (Result\ v,\ s))\ `\ UNIV)$
unfolding *unknown-def* **by** *simp*

lemma *always-progress-unknown*[*always-progress-intros*]: *always-progress unknown*
unfolding *unknown-def* **by** (*simp add*: *always-progress-intros*)

16.9.11 *lift-state*

lemma *run-lift-state*[*run-spec-monad*]:
run (*lift-state R f*) *s* = *lift-post-state* (*rel-prod* (=) *R*) (*SUP t*∈{*t*. *R s t*}. *run f t*)
by *transfer standard*

lemma *runs-to-lift-state-iff*[*runs-to-iff*]:
(*lift-state R f*) · *s* ⦃ *Q* ⦄ ⟷ (∀ *s'*. *R s s'* ⟶ *f* · *s'* ⦃λ*r t'*. ∀ *t*. *R t t'* ⟶ *Q r t*⦄)
by (*simp add*: *runs-to.rep-eq lift-state.rep-eq rel-prod-sel split-beta'*)

lemma *runs-to-lift-state*[*runs-to-vcg*]:
(⋀*s'*. *R s s'* ⟹ *f* · *s'* ⦃λ*r t'*. ∀ *t*. *R t t'* ⟶ *Q r t*⦄) ⟹ *lift-state R f* · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to-lift-state-iff*)

lemma *runs-to-partial-lift-state*[*runs-to-vcg*]:
(⋀*s'*. *R s s'* ⟹ *f* · *s'* *?*⦃λ*r t'*. ∀ *t*. *R t t'* ⟶ *Q r t*⦄) ⟹ *lift-state R f* · *s* *?*⦃ *Q* ⦄
by (*cases* ∀ *s'*. *R s s'* ⟶ *run f s'* ≠ *Failure*)
(*auto simp*: *runs-to-partial-alt run-lift-state lift-post-state-Sup image-image image-iff runs-to-lift-state-iff top-post-state-def*)

lemma *mono-lift-state*: *f* ≤ *f'* ⟹ *lift-state R f* ≤ *lift-state R f'*
by *transfer* (*auto simp*: *le-fun-def intro*!: *lift-post-state-mono SUP-mono*)

16.9.12 **const** *exec-concrete*

lemma *run-exec-concrete*[*run-spec-monad*]:
run (*exec-concrete st f*) *s* = *map-post-state* (*apsnd st*) (⨆ (*run f* ‘ *st* −‘ {*s*}))
by (*auto simp add*: *exec-concrete-def lift-state-def map-post-state-eq-lift-post-state fun-eq-iff*
intro!: *arg-cong2*[**where** *f*=*lift-post-state*] *SUP-cong*)

lemma *runs-to-exec-concrete-iff*[*runs-to-iff*]:
exec-concrete st f · *s* ⦃ *Q* ⦄ ⟷ (∀ *t*. *s* = *st t* ⟶ *f* · *t* ⦃λ*r t*. *Q r* (*st t*)⦄)
by (*auto simp*: *runs-to.rep-eq run-exec-concrete split-beta'*)

lemma *runs-to-exec-concrete*[*runs-to-vcg*]:
(⋀*t*. *s* = *st t* ⟹ *f* · *t* ⦃λ*r t*. *Q r* (*st t*)⦄) ⟹ *exec-concrete st f* · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to-exec-concrete-iff*)

lemma *runs-to-partial-exec-concrete*[*runs-to-vcg*]:
(⋀*t*. *s* = *st t* ⟹ *f* · *t* *?*⦃λ*r t*. *Q r* (*st t*)⦄) ⟹ *exec-concrete st f* · *s* *?*⦃ *Q* ⦄
by (*simp add*: *exec-concrete-def runs-to-partial-lift-state*)

lemma *mono-exec-concrete*: *f* ≤ *f'* ⟹ *exec-concrete st f* ≤ *exec-concrete st f'*
unfolding *exec-concrete-def* **by** (*rule mono-lift-state*)

lemma *monotone-exec-concrete-le*[*partial-function-mono*]:
monotone Q (≤) *f* ⟹ *monotone Q* (≤) (λ*f'*. *exec-concrete st* (*f f'*))

using *mono-exec-concrete*
by (*fastforce simp add*: *monotone-def le-fun-def*)

lemma *monotone-exec-concrete-ge*[*partial-function-mono*]:
monotone Q (≥) *f* ⟹ *monotone Q* (≥) (λ*f'*. *exec-concrete st* (*f f'*))
using *mono-exec-concrete*
by (*fastforce simp add*: *monotone-def le-fun-def*)

16.9.13 const *exec-abstract*

lemma *run-exec-abstract*[*run-spec-monad*]:
run (*exec-abstract st f*) *s* = *vmap-post-state* (*apsnd st*) (*run f* (*st s*))
by (*auto simp add*: *exec-abstract-def vmap-post-state-eq-lift-post-state lift-state-def fun-eq-iff*
intro!: *arg-cong2*[**where** *f*=*lift-post-state*])

lemma *runs-to-exec-abstract-iff*[*runs-to-iff*]:
exec-abstract st f · *s* ⦃ *Q* ⦄ ⟷ *f* · (*st s*) ⦃λ*r t*. ∀ *s'*. *t* = *st s'* ⟶ *Q r s'*⦄
by (*simp add*: *runs-to.rep-eq run-exec-abstract split-beta' prod-eq-iff eq-commute*)

lemma *runs-to-exec-abstract*[*runs-to-vcg*]:
f · (*st s*) ⦃λ*r t*. ∀ *s'*. *t* = *st s'* ⟶ *Q r s'*⦄ ⟹ *exec-abstract st f* · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to-exec-abstract-iff*)

lemma *runs-to-partial-exec-abstract*[*runs-to-vcg*]:
f · (*st s*) *?*⦃λ*r t*. ∀ *s'*. *t* = *st s'* ⟶ *Q r s'*⦄ ⟹ *exec-abstract st f* · *s ?*⦃ *Q* ⦄
by (*simp add*: *runs-to-partial.rep-eq run-exec-abstract split-beta' prod-eq-iff eq-commute*)

lemma *mono-exec-abstract*: *f* ≤ *f'* ⟹ *exec-abstract st f* ≤ *exec-abstract st f'*
unfolding *exec-abstract-def* **by** (*rule mono-lift-state*)

lemma *monotone-exec-abstract-le*[*partial-function-mono*]:
monotone Q (≤) *f* ⟹ *monotone Q* (≤) (λ*f'*. *exec-abstract st* (*f f'*))
by (*simp add*: *monotone-def mono-exec-abstract*)

lemma *monotone-exec-abstract-ge*[*partial-function-mono*]:
monotone Q (≥) *f* ⟹ *monotone Q* (≥) (λ*f'*. *exec-abstract st* (*f f'*))
by (*simp add*: *monotone-def mono-exec-abstract*)

16.9.14 *bind-exception-or-result*

lemma *runs-to-bind-exception-or-result-iff*[*runs-to-iff*]:
bind-exception-or-result f g · *s* ⦃ *Q* ⦄ ⟷ *f* · *s* ⦃ λ*r t*. *g r* · *t* ⦃ *Q* ⦄⦄
by *transfer* (*simp add*: *split-beta'*)

lemma *runs-to-bind-exception-or-result*[*runs-to-vcg*]:
f · *s* ⦃ λ*r t*. *g r* · *t* ⦃ *Q* ⦄⦄ ⟹ *bind-exception-or-result f g* · *s* ⦃ *Q* ⦄
by (*auto simp*: *runs-to-bind-exception-or-result-iff*)

lemma *runs-to-partial-bind-exception-or-result*[*runs-to-vcg*]:

$f \cdot s$?⦃ $\lambda r\ t.\ g\ r \cdot t$?⦃ Q ⦄⦄ $\Longrightarrow$ *bind-exception-or-result f g* $\cdot\ s$?⦃ Q ⦄
by *transfer* (*simp add*: *split-beta' holds-partial-bind-post-state*)

lemma *refines-bind-exception-or-result*:
refines f f' s s' (λ(*r*, *t*) (*r'*, *t'*). *refines* (*g r*) (*g' r'*) *t t' R*) $\Longrightarrow$
refines (*bind-exception-or-result f g*) (*bind-exception-or-result f' g'*) *s s' R*
by *transfer* (*auto intro*: *sim-bind-post-state*)

lemma *refines-bind-exception-or-result'*:
assumes *f*: *refines f f' s s' Q*
assumes *g*: $\bigwedge r\ t\ r'\ t'$. *Q* (*r*, *t*) (*r'*, *t'*) $\Longrightarrow$ *refines* (*g r*) (*g' r'*) *t t' R*
shows *refines* (*bind-exception-or-result f g*) (*bind-exception-or-result f' g'*) *s s' R*
by (*auto intro*: *refines-bind-exception-or-result refines-mono*[*OF* - *f*] *g*)

lemma *mono-bind-exception-or-result*:
$f \leq f' \Longrightarrow g \leq g' \Longrightarrow$ *bind-exception-or-result f g* $\leq$ *bind-exception-or-result f' g'*
unfolding *le-fun-def*
by *transfer* (*auto simp add*: *le-fun-def intro*!: *mono-bind-post-state*)

lemma *monotone-bind-exception-or-result-le*[*partial-function-mono*]:
monotone R ($\leq$) ($\lambda f'$. *f f'*) $\Longrightarrow$ ($\bigwedge v$. *monotone R* ($\leq$) ($\lambda f'$. *g f' v*)) $\Longrightarrow$
monotone R ($\leq$) ($\lambda f'$. *bind-exception-or-result* (*f f'*) (*g f'*))
by (*simp add*: *monotone-def*) (*metis le-fun-def mono-bind-exception-or-result*)

lemma *monotone-bind-exception-or-result-ge*[*partial-function-mono*]:
monotone R ($\geq$) ($\lambda f'$. *f f'*) $\Longrightarrow$ ($\bigwedge v$. *monotone R* ($\geq$) ($\lambda f'$. *g f' v*)) $\Longrightarrow$
monotone R ($\geq$) ($\lambda f'$. *bind-exception-or-result* (*f f'*) (*g f'*))
by (*simp add*: *monotone-def*) (*metis le-fun-def mono-bind-exception-or-result*)

lemma *run-bind-exception-or-result-cong*:
assumes $*$: *run f s* = *run f' s*
assumes $**$: *f* $\cdot\ s$?⦃ $\lambda x\ s'$. *run* (*g x*) *s'* = *run* (*g' x*) *s'* ⦄
shows *run* (*bind-exception-or-result f g*) *s* = *run* (*bind-exception-or-result f' g'*) *s*
using *assms*
by (*cases run f' s*)
(*auto simp*: *bind-exception-or-result-def runs-to-partial.rep-eq*
intro!: *SUP-cong split*: *exception-or-result-splits*)

16.9.15 *bind-handle*

lemma *bind-handle-eq*:
bind-handle f g h =
bind-exception-or-result f (λr. *case r of Exception e* $\Rightarrow$ *h e* | *Result v* $\Rightarrow$ *g v*)
unfolding *spec-monad-ext-iff bind-handle.rep-eq bind-exception-or-result.rep-eq*
by (*intro allI arg-cong*[**where** *f*=*bind-post-state* -] *ext*)
(*auto split*: *exception-or-result-split*)

lemma *runs-to-bind-handle-iff*[*runs-to-iff*]:
  bind-handle f g h · s {‌{ *Q* }‌} ⟷ *f · s* {‌{ λ*r t*.
    (∀ *v*. *r* = *Result v* ⟶ *g v · t* {‌{ *Q* }‌}) ∧
    (∀ *e*. *r* = *Exception e* ⟶ *e* ≠ *default* ⟶ *h e · t* {‌{ *Q* }‌})}‌}
  by *transfer*
    (*auto intro*!: *arg-cong*[**where** *f*=λ*p*. *holds-post-state p* -] *simp*: *fun-eq-iff*
        split: *prod.splits exception-or-result-splits*)

lemma *runs-to-bind-handle*[*runs-to-vcg*]:
  f · s {‌{ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *g v · t* {‌{ *Q* }‌}) ∧
              (∀ *e*. *r* = *Exception e* ⟶ *e* ≠ *default* ⟶ *h e · t* {‌{ *Q* }‌})}‌} ⟹
  bind-handle f g h · s {‌{ *Q* }‌}
  by (*auto simp*: *runs-to-bind-handle-iff*)

lemma *runs-to-bind-handle-exeption-monad*[*runs-to-vcg*]:
  fixes *f* :: (′*e*, ′*a*, ′*s*) *exn-monad*
  assumes *f*:
    f · s {‌{ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ (*g v · t* {‌{ *Q* }‌})) ∧
                (∀ *e*. *r* = *Exception* (*Some e*) ⟶ (*h* (*Some e*) · *t* {‌{ *Q* }‌}))}‌}
  shows *bind-handle f g h · s* {‌{ *Q* }‌}
  by (*rule runs-to-bind-handle*[*OF runs-to-weaken*[*OF f*]]) (*auto simp*: *default-option-def*)

lemma *runs-to-bind-handle-res-monad*[*runs-to-vcg*]:
  fixes *f* :: (′*a*, ′*s*) *res-monad*
  assumes *f*: *f · s* {‌{ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ (*g v · t* {‌{ *Q* }‌}))}‌}
  shows *bind-handle f g h · s* {‌{ *Q* }‌}
  by (*rule runs-to-bind-handle*[*OF runs-to-weaken*[*OF f*]]) (*auto simp*: *default-option-def*)

lemma *runs-to-partial-bind-handle*[*runs-to-vcg*]:
  f · s ?{‌{ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *g v · t ?*{‌{ *Q* }‌}) ∧
              (∀ *e*. *r* = *Exception e* ⟶ *e* ≠ *default* ⟶ *h e · t ?*{‌{ *Q* }‌})}‌} ⟹
  bind-handle f g h · s ?{‌{ *Q* }‌}
  by *transfer*
    (*auto intro*!: *holds-partial-bind-post-state intro*: *holds-partial-post-state-weaken*
        split: *exception-or-result-splits prod.splits*)

lemma *runs-to-partial-bind-handle-exeption-monad*[*runs-to-vcg*]:
  fixes *f* :: (′*e*, ′*a*, ′*s*) *exn-monad*
  assumes *f*: *f · s ?*{‌{ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ (*g v · t ?*{‌{ *Q* }‌})) ∧
    (∀ *e*. *r* = *Exception* (*Some e*) ⟶ (*h* (*Some e*) · *t ?*{‌{ *Q* }‌}))}‌}
  shows *bind-handle f g h · s ?*{‌{ *Q* }‌}
  by (*rule runs-to-partial-bind-handle*[*OF runs-to-partial-weaken*[*OF f*]])
    (*auto simp*: *default-option-def*)

lemma *runs-to-partial-bind-handle-res-monad*[*runs-to-vcg*]:
  fixes *f* :: (′*a*, ′*s*) *res-monad*
  assumes *f*: *f · s ?*{‌{ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ (*g v · t ?*{‌{ *Q* }‌}))}‌}
  shows *bind-handle f g h · s ?*{‌{ *Q* }‌}
  by (*rule runs-to-partial-bind-handle*[*OF runs-to-partial-weaken*[*OF f*]])

(*auto simp*: *default-option-def*)

lemma *mono-bind-handle*:
f ≤ *f'* ⟹ *g* ≤ *g'* ⟹ *h* ≤ *h'* ⟹ *bind-handle f g h* ≤ *bind-handle f' g' h'*
unfolding *le-fun-def*
by *transfer*
(*auto simp add*: *le-fun-def intro*!: *mono-bind-post-state split*: *exception-or-result-splits*)

lemma *monotone-bind-handle-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*) ⟹ (⋀*v*. *monotone R* (≤) (λ*f'*. *g f' v*)) ⟹
(⋀*e*. *monotone R* (≤) (λ*f'*. *h f' e*)) ⟹
monotone R (≤) (λ*f'*. *bind-handle* (*f f'*) (*g f'*) (*h f'*))
by (*simp add*: *monotone-def*) (*metis le-fun-def mono-bind-handle*)

lemma *monotone-bind-handle-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ (⋀*v*. *monotone R* (≥) (λ*f'*. *g f' v*)) ⟹
(⋀*e*. *monotone R* (≥) (λ*f'*. *h f' e*)) ⟹
monotone R (≥) (λ*f'*. *bind-handle* (*f f'*) (*g f'*) (*h f'*))
by (*simp add*: *monotone-def*) (*metis le-fun-def mono-bind-handle*)

lemma *run-bind-handle*[*run-spec-monad*]:
run (*bind-handle f g h*) *s* = *bind-post-state* (*run f s*)
(λ(*r*, *t*). *case r of Exception e* ⇒ *run* (*h e*) *t* | *Result v* ⇒ *run* (*g v*) *t*)
by *transfer simp*

lemma *always-progress-bind-handle*[*always-progress-intros*]:
always-progress f ⟹ (⋀*v*. *always-progress* (*g v*)) ⟹ (⋀*e*. *always-progress* (*h e*))
⟹ *always-progress* (*bind-handle f g h*)
by (*auto simp*: *always-progress.rep-eq run-bind-handle bind-post-state-eq-bot prod-eq-iff*
exception-or-result-nchotomy holds-post-state-False
split: *exception-or-result-splits prod.splits*)

lemma *refines-bind-handle'*:
refines f f' s s' (λ(*r*, *t*) (*r'*, *t'*).
(∀ *e e'*. *e* ≠ *default* ⟶ *e'* ≠ *default* ⟶ *r* = *Exception e* ⟶ *r'* = *Exception*
e' ⟶
refines (*h e*) (*h' e'*) *t t' R*) ∧
(∀ *e x'*. *e* ≠ *default* ⟶ *r* = *Exception e* ⟶ *r'* = *Result x'* ⟶
refines (*h e*) (*g' x'*) *t t' R*) ∧
(∀ *x e'*. *e'* ≠ *default* ⟶ *r* = *Result x* ⟶ *r'* = *Exception e'* ⟶
refines (*g x*) (*h' e'*) *t t' R*) ∧
(∀ *x x'*. *r* = *Result x* ⟶ *r'* = *Result x'* ⟶
refines (*g x*) (*g' x'*) *t t' R*)) ⟹
refines (*bind-handle f g h*) (*bind-handle f' g' h'*) *s s' R*
apply *transfer*
apply (*auto intro*!: *sim-bind-post-state'*)[*1*]
apply (*rule sim-post-state-weaken*, *assumption*)
apply (*auto split*: *exception-or-result-splits prod.splits*)
done

lemma *refines-bind-handle-bind-handle*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Exception\ e',\ t') \Longrightarrow$
$e \neq default \Longrightarrow e' \neq default \Longrightarrow$
refines $(h\ e)\ (h'\ e')\ t\ t'\ R$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Result\ v',\ t') \Longrightarrow e \neq default \Longrightarrow$
refines $(h\ e)\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exception\ e',\ t') \Longrightarrow e' \neq default \Longrightarrow$
refines $(g\ v)\ (h'\ e')\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
refines $(g\ v)\ (g'\ v')\ t\ t'\ R$
shows *refines* $(bind\text{-}handle\ f\ g\ h)\ (bind\text{-}handle\ f'\ g'\ h')\ s\ s'\ R$
apply (*rule refines-bind-handle'*)
apply (*rule refines-weaken*[*OF f*])
apply (*auto split*: *exception-or-result-splits prod.splits intro*: *ll lr rl rr*)
done

lemma *refines-bind-handle-bind-handle-exn*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Exn\ e',\ t') \Longrightarrow$
refines $(h\ (Some\ e))\ (h'\ (Some\ e'))\ t\ t'\ R$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Result\ v',\ t') \Longrightarrow$
refines $(h\ (Some\ e))\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e',\ t') \Longrightarrow$
refines $(g\ v)\ (h'\ (Some\ e'))\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
refines $(g\ v)\ (g'\ v')\ t\ t'\ R$
shows *refines* $(bind\text{-}handle\ f\ g\ h)\ (bind\text{-}handle\ f'\ g'\ h')\ s\ s'\ R$
apply (*rule refines-bind-handle'*)
apply (*rule refines-weaken*[*OF f*])
using *ll lr rl rr*
apply (*auto split*: *exception-or-result-splits prod.splits simp*: *Exn-def default-option-def*)
done

lemma *bind-handle-return-spec-monad*[*simp*]: $bind\text{-}handle\ (return\ v)\ g\ h = g\ v$
by *transfer simp*

lemma *bind-handle-throw-spec-monad*[*simp*]:
$v \neq default \Longrightarrow bind\text{-}handle\ (throw\text{-}exception\text{-}or\text{-}result\ v)\ g\ h = h\ v$
by *transfer simp*

lemma *bind-handle-bind-handle-spec-monad*:
$bind\text{-}handle\ (bind\text{-}handle\ f\ g1\ h1)\ g2\ h2 =$
$bind\text{-}handle\ f$
$(\lambda v.\ bind\text{-}handle\ (g1\ v)\ g2\ h2)$
$(\lambda e.\ bind\text{-}handle\ (h1\ e)\ g2\ h2)$
apply *transfer*
apply (*auto simp*: *fun-eq-iff intro*!: *arg-cong*[**where** *f*=*bind-post-state* -])[*1*]

subgoal for *g1 h1 g2 h2 r v* **by** (*cases r*) *simp-all*
done

lemma *mono-bind-handle-spec-monad*:
mono f $\Longrightarrow$ ($\bigwedge$*v. mono* (λ*x. g x v*)) $\Longrightarrow$ ($\bigwedge$*e. mono* (λ*x. h x e*)) $\Longrightarrow$
mono (λ*x. bind-handle* (*f x*) (*g x*) (*h x*))
unfolding *mono-def*
apply *transfer*
apply (*auto simp*: *le-fun-def intro*!: *mono-bind-post-state*)[*1*]
subgoal for *f g h x y r q* **by** (*cases r*) *simp-all*
done

lemma *rel-spec-bind-handle*:
rel-spec f f' s s' (λ(*r*, *t*) (*r'*, *t'*).
($\forall$ *e e'*. *e* $\neq$ *default* $\longrightarrow$ *e'* $\neq$ *default* $\longrightarrow$ *r* = *Exception e* $\longrightarrow$ *r'* = *Exception e'* $\longrightarrow$
rel-spec (*h e*) (*h' e'*) *t t' R*) $\wedge$
($\forall$ *e x'*. *e* $\neq$ *default* $\longrightarrow$ *r* = *Exception e* $\longrightarrow$ *r'* = *Result x'* $\longrightarrow$
rel-spec (*h e*) (*g' x'*) *t t' R*) $\wedge$
($\forall$ *x e'*. *e'* $\neq$ *default* $\longrightarrow$ *r* = *Result x* $\longrightarrow$ *r'* = *Exception e'* $\longrightarrow$
rel-spec (*g x*) (*h' e'*) *t t' R*) $\wedge$
($\forall$ *x x'*. *r* = *Result x* $\longrightarrow$ *r'* = *Result x'* $\longrightarrow$
rel-spec (*g x*) (*g' x'*) *t t' R*)) $\Longrightarrow$
rel-spec (*bind-handle f g h*) (*bind-handle f' g' h'*) *s s' R*
by (*auto simp*: *rel-spec-iff-refines intro*!: *refines-bind-handle' intro*: *refines-weaken*)

lemma *bind-finally-bind-handle-conv*: *bind-finally f g* = *bind-handle f* (λ*v. g* (*Result v*)) (λ*e. g* (*Exception e*))
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff Exn-def* [*symmetric*] *default-option-def elim*!: *runs-to-weaken*)[*1*]
using *exception-or-result-nchotomy* **by** *force*

16.9.16 (⩾=)

lemma *run-bind*[*run-spec-monad*]: *run* (*bind f g*) *s* =
bind-post-state (*run f s*) (λ(*r*, *t*). *case r of*
Exception e $\Rightarrow$ *pure-post-state* (*Exception e*, *t*)
| *Result v* $\Rightarrow$ *run* (*g v*) *t*)
by (*auto simp add*: *bind-def run-bind-handle*)

lemma *run-bind-res*: *run* (*f* >>= *g*) *s* = *bind-post-state* (*run f s*) (λ(*Res r*, *s*). *run* (*g r*) *s*)
by (*auto simp add*: *run-bind intro*!: *arg-cong2*[**where** *f*=*bind-post-state*])

lemma *run-bind-eq-top-iff*:
run (*bind f g*) *s* = $\top$ $\longleftrightarrow$ $\neg$ (*f* ⋆ *s* ⦃ λ*x s*. $\forall$ *a*. *x* = *Result a* $\longrightarrow$ *run* (*g a*) *s* $\neq$ $\top$ ⦄)
by (*simp add*: *run-bind bind-post-state-eq-top runs-to.rep-eq prod-eq-iff split-beta'*

split: *exception-or-result-splits prod.splits*)

lemma *always-progress-bind*[*always-progress-intros*]:
always-progress f $\Longrightarrow$ ($\bigwedge$*v*. *always-progress* (*g v*)) $\Longrightarrow$ *always-progress* (*bind f g*)
by (*simp add*: *always-progress-intros bind-def*)

lemma *run-bind-cong*:
assumes $*$: *run f s* = *run f' s*
assumes $**$: *f* · *s* ?⦃ $\lambda x\ s'$. $\forall\, v.\ x = (Result\ v) \longrightarrow run\ (g\ v)\ s' = run\ (g'\ v)\ s'$ ⦄
shows *run* (*bind f g*) *s* = *run* (*bind f' g'*) *s*
using *assms*
by (*cases run f' s*)
(*auto simp*: *run-bind runs-to-partial.rep-eq*
intro!: *SUP-cong split*: *exception-or-result-splits*)

lemma *runs-to-bind-iff*[*runs-to-iff*]:
bind f g · *s* ⦃ *Q* ⦄ $\longleftrightarrow$ *f* · *s* ⦃ $\lambda r\ t.$
($\forall\, v.\ r = Result\ v \longrightarrow g\ v$ · *t* ⦃ *Q* ⦄) $\wedge$
($\forall\, e.\ r = Exception\ e \longrightarrow e \neq default \longrightarrow Q\ (Exception\ e)\ t$)⦄
by (*simp add*: *bind-def runs-to-bind-handle-iff*)

lemma *runs-to-bind*[*runs-to-vcg*]:
f · *s* ⦃ $\lambda r\ t.$ ($\forall\, v.\ r = Result\ v \longrightarrow g\ v$ · *t* ⦃ *Q* ⦄) $\wedge$
($\forall\, e.\ r = Exception\ e \longrightarrow e \neq default \longrightarrow Q\ (Exception\ e)\ t$)⦄ $\Longrightarrow$
bind f g · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to-bind-iff*)

lemma *runs-to-bind-exception*[*runs-to-vcg*]:
fixes *f* :: ('*e*, '*a*, '*s*) *exn-monad*
assumes [*runs-to-vcg*]: *f* · *s* ⦃ $\lambda r\ t.$ ($\forall\, v.\ r = Result\ v \longrightarrow g\ v$ · *t* ⦃ *Q* ⦄) $\wedge$
($\forall\, e.\ r = Exn\ e \longrightarrow Q\ (Exn\ e)\ t$)⦄
shows *bind f g* · *s* ⦃ *Q* ⦄
supply *runs-to-bind*[*runs-to-vcg*]
by *runs-to-vcg* (*auto simp*: *Exn-def default-option-def*)

lemma *runs-to-bind-res*[*runs-to-vcg*]:
fixes *f* :: ('*a*, '*s*) *res-monad*
assumes [*runs-to-vcg*]: *f* · *s* ⦃ $\lambda Res\ v\ t.\ g\ v$ · *t* ⦃ *Q* ⦄⦄
shows *bind f g* · *s* ⦃ *Q* ⦄
supply *runs-to-bind*[*runs-to-vcg*]
by *runs-to-vcg*

lemma *runs-to-partial-bind*[*runs-to-vcg*]:
f · *s* ?⦃ $\lambda r\ t.$ ($\forall\, v.\ r = Result\ v \longrightarrow g\ v$ · *t* ?⦃ *Q* ⦄) $\wedge$
($\forall\, e.\ r = Exception\ e \longrightarrow e \neq default \longrightarrow Q\ (Exception\ e)\ t$)⦄ $\Longrightarrow$
bind f g · *s* ?⦃ *Q* ⦄
apply (*simp add*: *bind-def*)
apply (*rule runs-to-partial-bind-handle*)
apply *simp*

done

lemma *runs-to-partial-bind-exeption-monad*[*runs-to-vcg*]:
fixes f :: ('*e*, '*a*, '*s*) *exn-monad*
assumes [*runs-to-vcg*]: $f \cdot s$ *?*⦃ $\lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow g\ v \cdot t$ *?*⦃ Q ⦄) ∧
($\forall e.\ r = Exn\ e \longrightarrow Q\ (Exn\ e)\ t$)⦄
shows *bind* $f\ g \cdot s$ *?*⦃ Q ⦄
by *runs-to-vcg* (*auto simp add*: *default-option-def Exn-def*)

lemma *runs-to-partial-bind-res-monad*[*runs-to-vcg*]:
fixes f :: ('*a*, '*s*) *res-monad*
assumes [*runs-to-vcg*]: $f \cdot s$ *?*⦃ $\lambda Res\ v\ t.\ g\ v \cdot t$ *?*⦃ Q ⦄⦄
shows *bind* $f\ g \cdot s$ *?*⦃ Q ⦄
by *runs-to-vcg*

lemma *bind-return*[*simp*]: (*bind m return*) = m
apply (*clarsimp simp*: *spec-monad-eq-iff runs-to-bind-iff fun-eq-iff*
intro!: *runs-to-cong-pred-only*)
subgoal for $P\ r\ t$
by (*cases r*) *auto*
done

lemma *bind-skip*[*simp*]: *bind m* ($\lambda x.$ *skip*) = m
using *bind-return*[*of m*] **by** *simp*

lemma *return-bind*[*simp*]: *bind* (*return x*) $f = f\ x$
unfolding *bind-def* **by** *transfer simp*

lemma *bind-assoc*: *bind* (*bind f g*) h = *bind f* ($\lambda x.$ *bind* (*g x*) h)
apply (*rule spec-monad-ext*)
apply (*auto simp*: *split-beta' run-bind intro*!: *arg-cong*[**where** *f*=*bind-post-state*
-]
split: *exception-or-result-splits*)
done

lemma *mono-bind*: $f \le f' \Longrightarrow g \le g' \Longrightarrow$ *bind f g* ≤ *bind f' g'*
unfolding *bind-def*
by (*auto intro*: *mono-bind-handle*)

lemma *mono-bind-spec-monad*:
mono f ⟹ (⋀*v*. *mono* ($\lambda x.$ *g x v*)) ⟹ *mono* ($\lambda x.$ *bind* (*f x*) (*g x*))
unfolding *bind-def*
by (*intro mono-bind-handle-spec-monad mono-const*) *auto*

lemma *monotone-bind-le*[*partial-function-mono*]:
monotone R (≤) ($\lambda f'.$ *f f'*) ⟹ (⋀*v*. *monotone R* (≤) ($\lambda f'.$ *g f' v*))
⟹ *monotone R* (≤) ($\lambda f'.$ *bind* (*f f'*) (*g f'*))
unfolding *bind-def*
by (*auto intro*: *monotone-bind-handle-le*)

lemma *monotone-bind-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ (⋀*v*. *monotone R* (≥) (λ*f'*. *g f' v*))
⟹ *monotone R* (≥) (λ*f'*. *bind* (*f f'*) (*g f'*))
unfolding *bind-def*
by (*auto intro*: *monotone-bind-handle-ge*)

lemma *refines-bind'*:
assumes *f*: *refines f f' s s'* (λ(*x*, *t*) (*x'*, *t'*).
(∀ *e*. *e* ≠ *default* ⟶ *x* = *Exception e* ⟶
((∀ *e'*. *e'* ≠ *default* ⟶ *x'* = *Exception e'* ⟶ *R* (*Exception e*, *t*) (*Exception e'*, *t'*)) ∧
(∀ *v'*. *x'* = *Result v'* ⟶ *refines* (*throw-exception-or-result e*) (*g' v'*) *t t' R*)))
∧
(∀ *v*. *x* = *Result v* ⟶
((∀ *e'*. *e'* ≠ *default* ⟶ *x'* = *Exception e'* ⟶
refines (*g v*) (*throw-exception-or-result e'*) *t t' R*) ∧
(∀ *v'*. *x'* = *Result v'* ⟶ *refines* (*g v*) (*g' v'*) *t t' R*))))
shows *refines* (*bind f g*) (*bind f' g'*) *s s' R*
unfolding *bind-def*
by (*rule refines-bind-handle'*[*OF refines-weaken*, *OF f*]) *auto*

lemma *refines-bind*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*.
Q (*Exception e*, *t*) (*Exception e'*, *t'*) ⟹ *e* ≠ *default* ⟹ *e'* ≠ *default* ⟹
R (*Exception e*, *t*) (*Exception e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exception e*, *t*) (*Result v'*, *t'*) ⟹ *e* ≠ *default* ⟹
refines (*yield* (*Exception e*)) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exception e'*, *t'*) ⟹ *e'* ≠ *default* ⟹
refines (*g v*) (*yield* (*Exception e'*)) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹
refines (*g v*) (*g' v'*) *t t' R*
shows *refines* (*bind f g*) (*bind f' g'*) *s s' R*
by (*rule refines-bind'*[*OF refines-weaken*[*OF f*]])
(*auto simp*: *ll lr rl rr*)

lemma *refines-bind-bind-exn*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*. *Q* (*Exn e*, *t*) (*Exn e'*, *t'*) ⟹ *R* (*Exn e*, *t*) (*Exn e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exn e*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*throw e*) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exn e'*, *t'*) ⟹ *refines* (*g v*) (*throw e'*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*g v*) (*g' v'*) *t t' R*
shows *refines* (*f* ⤜ *g*) (*f'* ⤜ *g'*) *s s' R*
using *ll lr rl rr*
by (*intro refines-bind*[*OF f*])

(*auto simp*: *Exn-def default-option-def*)

lemma *refines-bind-res*:
assumes *f*: *refines f f' s s'* (λ(*Res r*, *t*) (*Res r'*, *t'*). *refines* (*g r*) (*g' r'*) *t t' R*)
shows *refines* ((*bind f g*)::('*a*,'*s*) *res-monad*) ((*bind f' g'*)::('*b*, '*t*) *res-monad*) *s s' R*
by (*rule refines-bind*[*OF f*]) *auto*

lemma *refines-bind-res'*:
assumes *f*: *refines f f' s s' Q*
assumes *g*: ⋀*r t r' t'*. *Q* (*Result r*, *t*) (*Result r'*, *t'*) ⟹ *refines* (*g r*) (*g' r'*) *t t' R*
shows *refines* ((*bind f g*)::('*a*,'*s*) *res-monad*) ((*bind f' g'*)::('*b*, '*t*) *res-monad*) *s s' R*
by (*auto intro*!: *refines-bind-res refines-weaken*[*OF f*] *g*)

lemma *refines-bind-res-bind-exn*:
assumes *f*: *refines f f' s s' Q*
assumes *rl*: ⋀*v e t t'*. *Q* (*Result v*, *t*) (*Exn e*, *t'*) ⟹
refines (*g v*) (*throw e*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹
refines (*g v*) (*g' v'*) *t t' R*
shows *refines* (*bind f g* :: (-, -) *res-monad*) (*bind f' g'*) *s s' R*
using *rl rr*
by (*intro refines-bind*[*OF f*]) (*auto simp*: *default-option-def simp flip*: *Exn-def*)

lemma *refines-bind-bind-exn-wp*:
assumes *f*: *refines f f' s s'* (λ(*r*, *t*) (*r'*,*t'*).
(*case r of*
Exn e ⇒ (*case r' of Exn e'* ⇒ *R* (*Exn e*, *t*) (*Exn e'*, *t'*) | *Result v'* ⇒ *refines* (*throw e*) (*g' v'*) *t t' R*)
| *Result v* ⇒ (*case r' of Exn e'* ⇒ *refines* (*g v*) (*throw e'*) *t t' R* | *Result v'* ⇒ *refines* (*g v*) (*g' v'*) *t t' R*)))
shows *refines* (*f* ⤜ *g*) (*f'* ⤜ *g'*) *s s' R*
apply (*rule refines-bind'*)
apply (*rule refines-weaken* [*OF f*])
apply (*auto simp add*: *Exn-def default-option-def split*: *xval-splits*)
done

lemma *rel-spec-bind-res*:
rel-spec f f' s s' (λ(*Res r*, *t*) (*Res r'*, *t'*). *rel-spec* (*g r*) (*g' r'*) *t t' R*) ⟹
rel-spec ((*bind f g*)::('*a*,'*s*) *res-monad*) ((*bind f' g'*)::('*b*, '*t*) *res-monad*) *s s' R*
unfolding *rel-spec-iff-refines*
by (*safe intro*!: *refines-bind-res*; (*rule refines-weaken*, *assumption*, *simp*))

lemma *rel-spec-bind-res'*:
assumes *f*: *rel-spec f f' s s' Q*
assumes *g*: ⋀*r t r' t'*. *Q* (*Result r*, *t*) (*Result r'*, *t'*) ⟹ *rel-spec* (*g r*) (*g' r'*) *t t' R*

shows *rel-spec* ((*bind f g*)::('*a*,'*s*) *res-monad*) ((*bind f' g'*)::('*b*, '*t*) *res-monad*) *s s' R*
by (*auto intro*!: *rel-spec-bind-res rel-spec-mono*[*OF* - *f*] *g*)

lemma *rel-spec-bind-bind*:
assumes *f*: *rel-spec f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*.
Q (*Exception e*, *t*) (*Exception e'*, *t'*) ⟹ *e* ≠ *default* ⟹ *e'* ≠ *default* ⟹
R (*Exception e*, *t*) (*Exception e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exception e*, *t*) (*Result v'*, *t'*) ⟹ *e* ≠ *default* ⟹
rel-spec (*yield* (*Exception e*)) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exception e'*, *t'*) ⟹ *e'* ≠ *default* ⟹
rel-spec (*g v*) (*yield* (*Exception e'*)) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹
rel-spec (*g v*) (*g' v'*) *t t' R*
shows *rel-spec* (*bind f g*) (*bind f' g'*) *s s' R*
using *assms* **unfolding** *rel-spec-iff-refines*
apply (*intro conjI*)
subgoal by (*rule refines-bind*[**where** *Q*=*Q*]) *auto*
subgoal by (*rule refines-bind*[**where** *Q*=Q^{-1-1}]) *auto*
done

lemma *rel-spec-bind-exn*:
assumes *f*: *rel-spec f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*. *Q* (*Exn e*, *t*) (*Exn e'*, *t'*) ⟹ *R* (*Exn e*, *t*) (*Exn e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exn e*, *t*) (*Result v'*, *t'*) ⟹ *rel-spec* (*throw e*) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exn e'*, *t'*) ⟹ *rel-spec* (*g v*) (*throw e'*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹ *rel-spec* (*g v*) (*g' v'*) *t t' R*
shows *rel-spec* (*bind f g*) (*bind f' g'*) *s s' R*
using *ll lr rl rr*
by (*intro rel-spec-bind-bind*[*OF f*])
(*auto simp*: *Exn-def default-option-def*)

lemma *refines-bind-right*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*.
Q (*Exception e*, *t*) (*Exception e'*, *t'*) ⟹ *e* ≠ *default* ⟹ *e'* ≠ *default* ⟹
R (*Exception e*, *t*) (*Exception e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exception e*, *t*) (*Result v'*, *t'*) ⟹ *e* ≠ *default* ⟹
refines (*yield* (*Exception e*)) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exception e'*, *t'*) ⟹ *e'* ≠ *default* ⟹
R (*Result v*, *t*) (*Exception e'*, *t'*)
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹
refines (*return v*) (*g' v'*) *t t' R*
shows *refines f* (*bind f' g'*) *s s' R*
proof −

have *refines* (*bind f return*) (*bind f′ g′*) *s s′ R*
using *ll lr rl rr* **by** (*intro refines-bind*[*OF f*]) *auto*
then show *?thesis* **by** *simp*
qed

lemma *refines-bind-left*:
assumes *f*: *refines f f′ s s′ Q*
assumes *ll*: ⋀*e e′ t t′*.
Q (*Exception e*, *t*) (*Exception e′*, *t′*) ⟹ *e* ≠ *default* ⟹ *e′* ≠ *default* ⟹
R (*Exception e*, *t*) (*Exception e′*, *t′*)
assumes *lr*: ⋀*e v′ t t′*. *Q* (*Exception e*, *t*) (*Result v′*, *t′*) ⟹ *e* ≠ *default* ⟹
R (*Exception e*, *t*) (*Result v′*, *t′*)
assumes *rl*: ⋀*v e′ t t′*. *Q* (*Result v*, *t*) (*Exception e′*, *t′*) ⟹ *e′* ≠ *default* ⟹
refines (*g v*) (*yield* (*Exception e′*)) *t t′ R*
assumes *rr*: ⋀*v v′ t t′*. *Q* (*Result v*, *t*) (*Result v′*, *t′*) ⟹
refines (*g v*) (*return v′*) *t t′ R*
shows *refines* (*bind f g*) *f′ s s′ R*
proof −
have *refines* (*bind f g*) (*bind f′ return*) *s s′ R*
using *ll lr rl rr* **by** (*intro refines-bind*[*OF f*]) *auto*
then show *?thesis* **by** *simp*
qed

lemma *rel-spec-bind-right*:
assumes *f*: *rel-spec f f′ s s′ Q*
assumes *ll*: ⋀*e e′ t t′*.
Q (*Exception e*, *t*) (*Exception e′*, *t′*) ⟹ *e* ≠ *default* ⟹ *e′* ≠ *default* ⟹
R (*Exception e*, *t*) (*Exception e′*, *t′*)
assumes *lr*: ⋀*e v′ t t′*. *Q* (*Exception e*, *t*) (*Result v′*, *t′*) ⟹ *e* ≠ *default* ⟹
rel-spec (*yield* (*Exception e*)) (*g′ v′*) *t t′ R*
assumes *rl*: ⋀*v e′ t t′*. *Q* (*Result v*, *t*) (*Exception e′*, *t′*) ⟹ *e′* ≠ *default* ⟹
R (*Result v*, *t*) (*Exception e′*, *t′*)
assumes *rr*: ⋀*v v′ t t′*. *Q* (*Result v*, *t*) (*Result v′*, *t′*) ⟹
rel-spec (*return v*) (*g′ v′*) *t t′ R*
shows *rel-spec f* (*bind f′ g′*) *s s′ R*
using *assms* **unfolding** *rel-spec-iff-refines*
apply (*intro conjI*)
subgoal by (*rule refines-bind-right*[**where** *Q*=*Q*]) *auto*
subgoal by (*rule refines-bind-left*[**where** *Q* =*Q*$^{-1-1}$]) *auto*
done

lemma *refines-bind-handle-left′*:
f · *s* {‖ λ*v s′*. (∀ *r*. *v* = *Result r* ⟶ *refines* (*g r*) *k s′ t R*) ∧
(∀ *e*. *e* ≠ *default* ⟶ *v* = *Exception e* ⟶ *refines* (*h e*) *k s′ t R*) ‖} ⟹
refines (*bind-handle f g h*) *k s t R*
by (*auto simp*: *runs-to.rep-eq refines.rep-eq run-bind-handle split-beta′ ac-simps*
intro!: *sim-bind-post-state-left split*: *prod.splits exception-or-result-splits*)

lemma *refines-bind-left-res*:

$f \cdot s \{\!|\ \lambda Res\ r\ s'.\ refines\ (g\ r)\ h\ s'\ t\ R\ |\!\} \Longrightarrow refines\ (f >>= g)\ h\ s\ t\ R$
unfolding *bind-def* **by** (*rule refines-bind-handle-left'*) *simp*

lemma *refines-bind-left-exn*:
$f \cdot s \{\!|\ \lambda r\ s'.\ (\forall a.\ r = Result\ a \longrightarrow refines\ (g\ a)\ h\ s'\ t\ R) \wedge$
$(\forall e.\ r = Exn\ e \longrightarrow refines\ (throw\ e)\ h\ s'\ t\ R)\ |\!\} \Longrightarrow$
$refines\ (f >>= g)\ h\ s\ t\ R$
unfolding *bind-def*
by (*rule refines-bind-handle-left'*)
(*auto simp add*: *Exn-def default-option-def imp-ex*)

lemma *runs-to-partial-bind1*:
$(\bigwedge r\ s.\ (g\ r) \cdot s\ ?\{\!|P|\!\}) \Longrightarrow ((f >>= g)::('a, 's)\ res\text{-}monad) \cdot s\ ?\{\!|P|\!\}$
apply (*rule runs-to-partial-bind*)
apply (*rule runs-to-partial-weaken*[*OF runs-to-partial-True*])
apply *auto*
done

lemma *unknown-bind-const*[*simp*]: $unknown >>= (\lambda x.\ f) = f$
by (*rule spec-monad-ext*) (*simp add*: *run-bind*)

lemma *bind-cong-left*:
fixes $f::('e::default, 'a, 's)\ spec\text{-}monad$
shows $(\bigwedge r.\ g\ r = g'\ r) \Longrightarrow (f >>= g) = (f >>= g')$
by (*rule spec-monad-ext*) (*simp add*: *run-bind*)

lemma *bind-cong-right*:
fixes $f::('e::default, 'a, 's)\ spec\text{-}monad$
shows $f = f' \Longrightarrow (f >>= g) = (f' >>= g)$
by (*rule spec-monad-ext*) (*simp add*: *run-bind*)

lemma *rel-spec-bind-res''*:
fixes $f::('a, 's)\ res\text{-}monad$
shows $f \cdot s\ ?\{\!|\ \lambda Res\ r\ t.\ rel\text{-}spec\ (g\ r)\ (g'\ r)\ t\ t\ R|\!\} \Longrightarrow rel\text{-}spec\ (f >>= g)\ (f >>= g')\ s\ s\ R$
by (*intro rel-spec-bind-res rel-spec-refl'*) (*auto simp*: *split-beta'*)

lemma *rel-spec-monad-bind-rel-exception-or-result*:
assumes *mn*: *rel-spec-monad R* (*rel-exception-or-result E P*) *m n*
and *fg*: *rel-fun P* (*rel-spec-monad R* (*rel-exception-or-result E Q*)) *f g*
shows *rel-spec-monad R* (*rel-exception-or-result E Q*) (*m >>= f*) (*n >>= g*)
by (*intro rel-spec-monadI rel-spec-bind-bind*[*OF rel-spec-monadD*[*OF mn*]])
(*simp-all add*: *fg*[*THEN rel-funD*, *THEN rel-spec-monadD*])

lemma *rel-spec-monad-bind-rel-exception-or-result'*:
$(\bigwedge x.\ rel\text{-}spec\text{-}monad\ (=)\ (rel\text{-}exception\text{-}or\text{-}result\ (=)\ Q)\ (f\ x)\ (g\ x)) \Longrightarrow$
rel-spec-monad (=) (*rel-exception-or-result* (=) *Q*) (*m >>= f*) (*m >>= g*)
apply (*rule rel-spec-monad-bind-rel-exception-or-result*[**where** *P*=(=)])
apply (*auto simp add*: *rel-exception-or-result-eq-conv rel-spec-monad-eq-conv*)

done

lemma *rel-spec-monad-bind*:
rel-spec-monad R (λ*Res v1 Res v2*. *P v1 v2*) *m n* ⟹ *rel-fun P* (*rel-spec-monad R Q*) *f g* ⟹
rel-spec-monad R Q (*m* >>= *f*) (*n* >>= *g*)
apply (*clarsimp simp add*: *rel-spec-monad-iff-rel-spec*[*abs-def*] *rel-fun-def*
intro!: *rel-spec-bind-res*)
subgoal premises *prems* **for** *s t*
by (*rule rel-spec-mono*[*OF - prems*(*1*)[*rule-format*, *OF prems*(*3*)]])
(*clarsimp simp*: *le-fun-def prems*(*2*)[*rule-format*])
done

lemma *rel-spec-monad-bind-left*:
assumes *mn*: *rel-spec-monad R* (λ*Res v1 Res v2*. *P v1 v2*) *m n*
and *fg*: *rel-fun P* (*rel-spec-monad R Q*) *f return*
shows *rel-spec-monad R Q* (*m* >>= *f*) *n*
proof −
from *mn fg*
have *rel-spec-monad R Q* (*m* >>= *f*) (*n* >>= *return*)
by (*rule rel-spec-monad-bind*)
then show *?thesis*
by (*simp*)
qed

lemma *rel-spec-monad-bind′*:
fixes *m*:: (′*a*, ′*s*) *res-monad*
shows (⋀*x*. *rel-spec-monad* (=) *Q* (*f x*) (*g x*)) ⟹ *rel-spec-monad* (=) *Q* (*m* >>= *f*) (*m* >>= *g*)
by (*rule rel-spec-monad-bind*[**where** *R*=(=) **and** *P*=(=)])
(*auto simp*: *rel-fun-def rel-spec-monad-iff-rel-spec intro*!: *rel-spec-refl*)

lemma *run-bind-cong-simple*:
run f s = *run f′ s* ⟹ (⋀*v s*. *run* (*g v*) *s* = *run* (*g′ v*) *s*) ⟹
run (*bind f g*) *s* = *run* (*bind f′ g′*) *s*
by (*simp add*: *run-bind*)

lemma *run-bind-cong-simple-same-head*: (⋀*v s*. *run* (*g v*) *s* = *run* (*g′ v*) *s*) ⟹
run (*bind f g*) *s* = *run* (*bind f g′*) *s*
by (*rule run-bind-cong-simple*) *auto*

lemma *refines-bind-same*:
refines (*f* >>= *g*) (*f* >>= *g′*) *s s R* **if** *f* · *s* {|λ*Res y t*. *refines* (*g y*) (*g′ y*) *t t R* |}
apply (*rule refines-bind-res*)
apply (*rule refines-same-runs-toI*)
apply (*rule runs-to-weaken*[*OF that*])
apply *auto*
done

lemma *refines-bind-handle-right-runs-to-partialI*:
g · *t* ?⦃ λ*r t'*. (∀ *e*. *e* ≠ *default* ⟶ *r* = *Exception e* ⟶ *refines f* (*k e*) *s t' R*) ∧
(∀ *a*. *r* = *Result a* ⟶ *refines f* (*h a*) *s t' R*) ⦄ ⟹ *always-progress g* ⟹
refines f (*bind-handle g h k*) *s t R*
by *transfer*
(*auto simp*: *prod-eq-iff split-beta'*
intro!: *sim-bind-post-state-right*
split: *exception-or-result-splits prod.splits*)

lemma *refines-bind-right-runs-to-partialI*:
g · *t* ?⦃ λ*r t'*. (∀ *e*. *e* ≠ *default* ⟶ *r* = *Exception e* ⟶
refines f (*throw-exception-or-result e*) *s t' R*) ∧
(∀ *a*. *r* = *Result a* ⟶ *refines f* (*h a*) *s t' R*) ⦄ ⟹ *always-progress g* ⟹
refines f (*bind g h*) *s t R*
unfolding *bind-def* **by** (*rule refines-bind-handle-right-runs-to-partialI*)

lemma *refines-bind-right-runs-toI*:
g · *t* ⦃ λ*Res r t'*. *refines f* (*h r*) *s t' R* ⦄ ⟹ *always-progress g* ⟹
refines f (*g* >>= *h*) *s t R*
unfolding *bind-def*
by (*rule refines-bind-handle-right-runs-to-partialI*)
(*auto intro*: *runs-to-partial-of-runs-to*)

lemma *refines-bind-left-refine*:
refines (*f* >>= *g*) *h s t R*
if *refines f f' s s* (=) *refines* (*f'* >>= *g*) *h s t R*
by (*rule refines-trans*[*OF refines-bind*[*OF that*(*1*)] *that*(*2*), **where** *R*=(=)])
(*auto simp add*: *refines-refl*)

lemma *refines-bind-right-single*:
assumes *x*: *pure-post-state* (*Result x*, *u*) ≤ *run g t* **and** *h*: *refines f* (*h x*) *s u R*
shows *refines f* (*g* ⤜ *h*) *s t R*
apply (*rule refines-le-run-trans*[*OF h*])
apply (*simp add*: *run-bind*)
apply (*rule order-trans*[*OF - mono-bind-post-state*, *OF - x order-refl*])
apply *simp*
done

lemma *return-let-bind*: (*return* (*let v* = *f' in* (*g' v*))) = *do* {*v* <− *return f'*; *return* (*g' v*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *rel-spec-monad-rel-xval-bind*:
assumes *f-f'*: *rel-spec-monad S* (*rel-xval L P*) *f f'*
assumes *Res-Res*: ⋀*v v'*. *P v v'* ⟹ *rel-spec-monad S* (*rel-xval L R*) (*g v*) (*g' v'*)
shows *rel-spec-monad S* (*rel-xval L R*) (*f* ⤜ *g*) (*f'* ⤜ *g'*)

apply (*intro rel-spec-monadI rel-spec-bind-exn*)
apply (*rule f-f'*[*THEN rel-spec-monadD*], *assumption*)
using *Res-Res*[*THEN rel-spec-monadD*]
apply *auto*
done

lemma *rel-spec-monad-rel-xval-same-bind*:
assumes *f-f'*: *rel-spec-monad S* (*rel-xval L R*) *f f'*
assumes *Res-Res*: $\bigwedge v\ v'.\ R\ v\ v' \Longrightarrow$ *rel-spec-monad S* (*rel-xval L R*) (*g v*) (*g' v'*)
shows *rel-spec-monad S* (*rel-xval L R*) (*f* $\gg\!=$ *g*) (*f'* $\gg\!=$ *g'*)
using *assms* **by** (*rule rel-spec-monad-rel-xval-bind*)

lemma *rel-spec-monad-rel-xval-result-eq-bind*:
assumes *f-f'*: *rel-spec-monad S* (*rel-xval L* (=)) *f f'*
assumes *Res-Res*: $\bigwedge v.$ *rel-spec-monad S* (*rel-xval L* (=)) (*g v*) (*g' v*)
shows *rel-spec-monad S* (*rel-xval L* (=)) (*f* $\gg\!=$ *g*) (*f'* $\gg\!=$ *g'*)
apply (*rule rel-spec-monad-rel-xval-bind* [*OF f-f'*])
subgoal using *Res-Res* **by** *auto*
done

lemma *rel-spec-monad-fail*: *rel-spec-monad Q R fail fail*
by (*auto simp add*: *rel-spec-monad-def rel-set-def*)

lemma *bind-fail*[*simp*]: *fail* $\gg\!=$ *X* = *fail*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)
done

16.9.17 *assert*

lemma *assert-simps*[*simp*]:
assert True = *return* ()
assert False = *top*
by (*auto simp add*: *assert-def*)

lemma *always-progress-assert*[*always-progress-intros*]: *always-progress* (*assert P*)
by (*simp add*: *always-progress-def assert-def*)

lemma *run-assert*[*run-spec-monad*]:
run (*assert P*) *s* = (*if P then pure-post-state* (*Result* (), *s*) *else* ⊤)
by (*simp add*: *assert-def*)

lemma *runs-to-assert-iff*[*simp*]: *assert P* · *s* ⦃ *Q* ⦄ ⟷ *P* ∧ *Q* (*Result* ()) *s*
by (*simp add*: *runs-to.rep-eq run-assert*)

lemma *runs-to-assert*[*runs-to-vcg*]: *P* ⟹ *Q* (*Result* ()) *s* ⟹ *assert P* · *s* ⦃ *Q* ⦄
by *simp*

lemma *runs-to-partial-assert-iff*[*simp*]: *assert* P · s *?⦃* Q *⦄* $\longleftrightarrow$ ($P \longrightarrow Q$ (*Result* ()) s)
by (*simp add*: *runs-to-partial.rep-eq run-assert*)

lemma *refines-top-iff*[*simp*]: *refines* $\top$ g s t $R \longleftrightarrow$ *run* g $t = \top$
by *transfer auto*

lemma *refines-assert*:
refines (*assert* P) (*assert* Q) s t $R \longleftrightarrow$ ($Q \longrightarrow P \wedge R$ (*Result* (), s) (*Result* (), t))
by (*simp add*: *assert-def*)

16.9.18 *assume*

lemma *assume-simps*[*simp*]:
assume True = *return* ()
assume False = *bot*
by (*auto simp add*: *assume-def*)

lemma *always-progress-assume*[*always-progress-intros*]: $P \Longrightarrow$ *always-progress* (*assume* P)
by (*simp add*: *always-progress-def assume-def*)

lemma *run-assume*[*run-spec-monad*]:
run (*assume* P) s = (*if* P *then pure-post-state* (*Result* (), s) *else* $\bot$)
by (*simp add*: *assume-def*)

lemma *run-assume-simps*[*run-spec-monad*, *simp*]:
$P \Longrightarrow$ *run* (*assume* P) s = *pure-post-state* (*Result* (), s)
$\neg$ $P \Longrightarrow$ *run* (*assume* P) $s = \bot$
by (*simp-all add*: *run-assume*)

lemma *runs-to-assume-iff*[*simp*]: *assume* P · s ⦃ Q ⦄ $\longleftrightarrow$ ($P \longrightarrow Q$ (*Result* ()) s)
by (*simp add*: *runs-to.rep-eq run-assume*)

lemma *runs-to-partial-assume-iff*[*simp*]: *assume* P · s *?⦃* Q *⦄* $\longleftrightarrow$ ($P \longrightarrow Q$ (*Result* ()) s)
by (*simp add*: *runs-to-partial.rep-eq run-assume*)

lemma *runs-to-assume*[*runs-to-vcg*]: ($P \Longrightarrow Q$ (*Result* ()) s) $\Longrightarrow$ *assume* P · s ⦃ Q ⦄
by *simp*

16.9.19 *assume-outcome*

lemma *run-assume-outcome*[*run-spec-monad*, *simp*]: *run* (*assume-outcome* f) s = *Success* (f s)
apply *transfer*
apply *simp*

done

lemma *always-progress-assume-outcome*[*always-progress-intros*]:
$(\bigwedge s.\ f\ s \neq \{\}) \Longrightarrow$ *always-progress* (*assume-outcome f*)
by (*auto simp add*: *always-progress-def bot-post-state-def*)

lemma *runs-to-assume-outcome*[*runs-to-vcg*]:
(*assume-outcome f*) · *s* $\{\!|\lambda r\ t.\ (r,\ t) \in f\ s|\!\}$
by (*simp add*: *runs-to.rep-eq*)

lemma *runs-to-assume-outcome-iff*[*runs-to-iff*]:
(*assume-outcome f*) · *s* $\{\!|Q|\!\} \longleftrightarrow (\forall (r,\ t) \in f\ s.\ Q\ r\ t)$
by (*auto simp add*: *runs-to.rep-eq*)

lemma *runs-to-partial-assume-outcome*[*runs-to-vcg*]:
(*assume-outcome f*) · *s* ?$\{\!|\lambda r\ t.\ (r,\ t) \in f\ s|\!\}$
by (*simp add*: *runs-to-partial.rep-eq*)

lemma *assume-outcome-elementary*:
assume-outcome f = *do* {*s* ← *get-state*; (*r*, *t*) ← *select* (*f s*); *set-state t*; *yield r*}
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind Sup-Success pure-post-state-def*)
done

16.9.20 *assume-result-and-state*

lemma *run-assume-result-and-state*[*run-spec-monad*, *simp*]:
run (*assume-result-and-state f*) *s* = *Success* ((λ(*v*, *t*). (*Result v*, *t*)) ‘ *f s*)
by (*auto simp add*: *assume-result-and-state-def run-bind Sup-Success-pair pure-post-state-def*)

lemma *always-progress-assume-result-and-state*[*always-progress-intros*]:
$(\bigwedge s.\ f\ s \neq \{\}) \Longrightarrow$ *always-progress* (*assume-result-and-state f*)
by (*simp add*: *always-progress-def*)

lemma *runs-to-assume-result-and-state*[*runs-to-vcg*]:
assume-result-and-state f · *s* $\{\!|\lambda r\ t.\ \exists v.\ r = Result\ v \wedge (v,\ t) \in f\ s|\!\}$
by (*auto simp add*: *runs-to.rep-eq*)

lemma *runs-to-assume-result-and-state-iff*[*runs-to-iff*]:
assume-result-and-state f · *s* $\{\!|Q|\!\} \longleftrightarrow (\forall (v,\ t) \in f\ s.\ Q\ (Result\ v)\ t)$
by (*auto simp add*: *runs-to.rep-eq*)

lemma *runs-to-partial-assume-result-and-state*[*runs-to-vcg*]:
assume-result-and-state f · *s* ?$\{\!|\lambda r\ t.\ \exists v.\ r = Result\ v \wedge (v,\ t) \in f\ s|\!\}$
by (*auto simp add*: *runs-to-partial.rep-eq*)

lemma *refines-assume-result-and-state-right*:
$f \cdot s\ \{\!|\lambda r\ s'.\ \exists r'\ t'.\ (r',\ t') \in g\ t \wedge R\ (r,\ s')\ (Result\ r',\ t')\ |\!\} \Longrightarrow$
refines f (*assume-result-and-state g*) *s t R*

by (*simp add*: *refines.rep-eq runs-to.rep-eq sim-post-state-Success2 split-beta′ Bex-def*)

lemma *refines-assume-result-and-state*:
sim-set R (*P s*) (*Q t*) ⟹
refines (*assume-result-and-state P*) (*assume-result-and-state Q*) *s t*
(λ(*Res v*, *s*) (*Res w*, *t*). *R* (*v*, *s*) (*w*, *t*))
by (*force simp*: *refines.rep-eq sim-set-def*)

lemma *refines-assume-result-and-state-iff*:
refines (*assume-result-and-state A*) (*assume-result-and-state B*) *s t Q* ⟷
sim-set (λ(*v*, *s′*) (*w*, *t′*). *Q* (*Result v*, *s′*) (*Result w*, *t′*)) (*A s*) (*B t*)
apply (*simp add*: *refines.rep-eq*, *intro iffI*)
subgoal
by (*fastforce simp add*: *sim-set-def*)
subgoal
by (*force simp add*: *sim-set-def*)
done

16.9.21 *gets*

lemma *run-gets*[*run-spec-monad*, *simp*]:*run* (*gets f*) *s* = *pure-post-state* (*Result* (*f s*), *s*)
by (*simp add*: *gets-def run-bind*)

lemma *always-progress-gets*[*always-progress-intros*]: *always-progress* (*gets f*)
by (*simp add*: *always-progress-def*)

lemma *runs-to-gets*[*runs-to-vcg*]: *gets f* · *s* ⦃λ*r t*. *r* = *Result* (*f s*) ∧ *t* = *s*⦄
by (*simp add*: *runs-to.rep-eq*)

lemma *runs-to-gets-iff*[*runs-to-iff*]: *gets f* · *s* ⦃*Q*⦄ ⟷ *Q* (*Result* (*f s*)) *s*
by (*simp add*: *runs-to.rep-eq*)

lemma *runs-to-partial-gets*[*runs-to-vcg*]: *gets f* · *s* ?⦃λ*r t*. *r* = *Result* (*f s*) ∧ *t* = *s*⦄
by (*simp add*: *runs-to-partial.rep-eq*)

lemma *refines-gets*: *R* (*Result* (*f s*), *s*) (*Result* (*g t*), *t*) ⟹ *refines* (*gets f*) (*gets g*) *s t R*
by (*auto simp add*: *refines.rep-eq*)

lemma *rel-spec-gets*: *R* (*Result* (*f s*), *s*) (*Result* (*g t*), *t*) ⟹ *rel-spec* (*gets f*) (*gets g*) *s t R*
by (*auto simp add*: *rel-spec-iff-refines intro*: *refines-gets*)

lemma *runs-to-always-progress-to-gets*:
(⋀*s*. *f* · *s* ⦃λ*r t*. *t* = *s* ∧ *r* = *Result* (*v s*)⦄) ⟹ *always-progress f* ⟹ *f* = *gets v*
apply (*clarsimp simp*: *spec-monad-ext-iff runs-to.rep-eq always-progress.rep-eq*)
subgoal premises *prems* **for** *s*

using *prems*[*rule-format*, *of s*]
by (*cases run f s*; *auto simp add*: *pure-post-state-def Ball-def*)
done

lemma *gets-let-bind*: (*gets* (λ*s*. *let v* = *f' s in* (*g' v s*))) = *do* {*v* <− *gets f'*; *gets* (*g' v*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

16.9.22 *assert-result-and-state*

lemma *run-assert-result-and-state*[*run-spec-monad*]:
run (*assert-result-and-state f*) *s* =
(*if f s* = {} *then* ⊤ *else Success* ((λ(*v*, *t*). (*Result v*, *t*)) ‘ *f s*))
by (*auto simp add*: *assert-result-and-state-def run-bind pure-post-state-def Sup-Success-pair*)

lemma *always-progress-assert-result-and-state*[*always-progress-intros*]:
always-progress (*assert-result-and-state f*)
by (*auto simp add*: *always-progress-def run-assert-result-and-state*)

lemma *runs-to-assert-result-and-state-iff*[*runs-to-iff*]:
assert-result-and-state f · *s* ⦃*Q*⦄ ⟷ (*f s* ≠ {} ∧ (∀ (*v*,*t*) ∈ *f s*. *Q* (*Result v*) *t*))
by (*auto simp add*: *runs-to.rep-eq run-assert-result-and-state*)

lemma *runs-to-assert-result-and-state*[*runs-to-vcg*]:
f s ≠ {} ⟹ *assert-result-and-state f* · *s* ⦃λ*r t*. ∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*⦄
by (*simp add*: *runs-to-assert-result-and-state-iff*)

lemma *runs-to-partial-assert-result-and-state*[*runs-to-vcg*]:
assert-result-and-state f · *s* ?⦃λ*r t*. ∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*⦄
by (*auto simp add*: *runs-to-partial.rep-eq run-assert-result-and-state*)

lemma *runs-to-state-select*[*runs-to-vcg*]:
∃ *t*. (*s*, *t*) ∈ *R* ⟹ *state-select R* · *s* ⦃ λ*r t*. *r* = *Result* () ∧ (*s*, *t*) ∈ *R* ⦄
by (*auto intro*!: *runs-to-assert-result-and-state*[*THEN runs-to-weaken*])

lemma *runs-to-partial-state-select*[*runs-to-vcg*]:
state-select R · *s* ?⦃ λ*r t*. *r* = *Result* () ∧ (*s*, *t*) ∈ *R* ⦄
by (*simp add*: *runs-to-partial.rep-eq run-assert-result-and-state*)

lemma *refines-assert-result-and-state*:
assumes *sim*: (⋀*r' s'*. (*r'*, *s'*) ∈ *f s* ⟹ (∃ *v' t'*. (*v'*, *t'*) ∈ *g t* ∧ *R* (*Result r'*, *s'*) (*Result v'*, *t'*)))
assumes *emp*: *f s* = {} ⟹ *g t* = {}
shows *refines* (*assert-result-and-state f*) (*assert-result-and-state g*) *s t R*
using *sim emp* **by** (*fastforce simp add*: *refines-iff-runs-to runs-to-iff*)

lemma *refines-state-select*:

assumes *sim*: $(\bigwedge s'.\ (s,\ s') \in f \Longrightarrow (\exists\, t'.\ (t,\ t') \in g \wedge R\ (\mathit{Result}\ (),\ s')\ (\mathit{Result}\ (),\ t')))$
assumes *emp*: $\nexists\, s'.\ (s,\ s') \in f \Longrightarrow \nexists\, t'.\ (t,\ t') \in g$
shows *refines* (*state-select f*) (*state-select g*) *s t R*
using *sim emp* **by** (*intro refines-assert-result-and-state*) *auto*

16.9.23 *assuming*

lemma *run-assuming*[*run-spec-monad*]:
run (*assuming g*) *s* = (*if g s then pure-post-state* (*Result* (), *s*) *else* ⊥)
by (*simp add*: *assuming-def run-bind*)

lemma *always-progress-assuming*[*always-progress-intros*]: *always-progress* (*assuming g*) ⟷ (∀ *s*. *g s*)
by (*auto simp add*: *always-progress-def run-assuming*)

lemma *runs-to-assuming*[*runs-to-vcg*]: *assuming g* • *s* ⦃λ*r t*. *g s* ∧ *r* = *Result* () ∧ *t* = *s*⦄
by (*simp add*: *runs-to.rep-eq run-assuming*)

lemma *runs-to-assuming-iff*[*runs-to-iff*]: *assuming g* • *s* ⦃*Q*⦄ ⟷ (*g s* ⟶ *Q* (*Result* ()) *s*)
by (*auto simp add*: *runs-to.rep-eq run-assuming*)

lemma *runs-to-partial-assuming*[*runs-to-vcg*]:
assuming g • *s* ?⦃λ*r t*. *g s* ∧ *r* = *Result* () ∧ *t* = *s* ∧ *g s*⦄
by (*simp add*: *runs-to-partial.rep-eq run-assuming*)

lemma *assuming-state-assume*:
assuming P = *assume-result-and-state* (λ*s*. (*if P s then* {((), *s*)} *else* {}))
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *assuming-True*[*simp*]: *assuming* (λ*s*. *True*) = *skip*
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *refines-assuming*:
(*P s* ⟹ *Q t*) ⟹ (*P s* ⟹ *Q t* ⟹ *R* (*Result* (), *s*) (*Result* (), *t*)) ⟹
refines (*assuming P*) (*assuming Q*) *s t R*
by (*auto simp add*: *refines.rep-eq run-assuming*)

lemma *rel-spec-assuming*:
(*Q t* ⟷ *P s*) ⟹ (*P s* ⟹ *Q t* ⟹ *R* (*Result* (), *s*) (*Result* (), *t*)) ⟹
rel-spec (*assuming P*) (*assuming Q*) *s t R*
by (*auto simp add*: *rel-spec-def run-assuming rel-set-def*)

lemma *refines-bind-assuming-right*:
P t ⟹ (*P t* ⟹ *refines f* (*g* ()) *s t R*) ⟹ *refines f* (*assuming P* ⤜ *g*) *s t R*
by (*simp add*: *refines.rep-eq run-assuming run-bind*)

lemma *refines-bind-assuming-left*:
(*P s* $\Longrightarrow$ *refines* (*f* ()) *g s t R*) $\Longrightarrow$ *refines* (*assuming P* >>= *f*) *g s t R*
by (*simp add*: *refines.rep-eq run-assuming run-bind*)

16.9.24 *guard*

lemma *run-guard*[*run-spec-monad*]:
run (*guard g*) *s* = (*if g s then pure-post-state* (*Result* (), *s*) *else* $\top$)
by (*simp add*: *guard-def run-bind*)

lemma *always-progress-guard*[*always-progress-intros*]: *always-progress* (*guard g*)
by (*auto simp add*: *always-progress-def run-guard*)

lemma *runs-to-guard*[*runs-to-vcg*]: *g s* $\Longrightarrow$ *guard g* · *s* ⦃$\lambda r\ t.\ r$ = *Result* () $\wedge$ *t* = *s*⦄
by (*simp add*: *runs-to.rep-eq run-guard*)

lemma *runs-to-guard-iff*[*runs-to-iff*]: *guard g* · *s* ⦃*Q*⦄ $\longleftrightarrow$ (*g s* $\wedge$ *Q* (*Result* ()) *s*)
by (*auto simp add*: *runs-to.rep-eq run-guard*)

lemma *runs-to-partial-guard*[*runs-to-vcg*]: *guard g* · *s* ?⦃$\lambda r\ t.\ r$ = *Result* () $\wedge$ *t* = *s* $\wedge$ *g s*⦄
by (*simp add*: *runs-to-partial.rep-eq run-guard*)

lemma *refines-guard*:
(*Q t* $\Longrightarrow$ *P s*) $\Longrightarrow$ (*P s* $\Longrightarrow$ *Q t* $\Longrightarrow$ *R* (*Result* (), *s*) (*Result* (), *t*)) $\Longrightarrow$
refines (*guard P*) (*guard Q*) *s t R*
by (*auto simp add*: *refines.rep-eq run-guard*)

lemma *rel-spec-guard*:
(*Q t* $\longleftrightarrow$ *P s*) $\Longrightarrow$ (*P s* $\Longrightarrow$ *Q t* $\Longrightarrow$ *R* (*Result* (), *s*) (*Result* (), *t*)) $\Longrightarrow$
rel-spec (*guard P*) (*guard Q*) *s t R*
by (*auto simp add*: *rel-spec-def run-guard rel-set-def*)

lemma *refines-bind-guard-right*:
refines f (*guard P* $\gg=$ *g*) *s t R* **if** *P t* $\Longrightarrow$ *refines f* (*g* ()) *s t R*
using *that*
by (*auto simp*: *refines.rep-eq run-guard run-bind*)

lemma *guard-False-fail*: *guard* (λ-. *False*) = *fail*
by (*simp add*: *spec-monad-ext run-guard*)

lemma *rel-spec-monad-bind-guard*:
shows ($\bigwedge x$. *rel-spec-monad* (=) *Q* (*f x*) (*g x*)) $\Longrightarrow$
rel-spec-monad (=) *Q* (*guard P* >>= *f*) (*guard P* >>= *g*)
by (*auto simp add*: *rel-spec-monad-def run-bind run-guard*)

lemma *runs-to-guard-bind-iff*: ((*guard P* >>= *f*) · *s* ⦃ *Q* ⦄) $\longleftrightarrow$ *P s* $\wedge$ ((*f* ()) · *s* ⦃ *Q* ⦄)

by (*simp add*: *runs-to-iff*)

lemma *refines-bind-guard-right-iff*:
refines f (*guard P* ⤜ *g*) *s t R* ⟷ (*P t* ⟶ *refines f* (*g* ()) *s t R*)
by (*auto simp*: *refines.rep-eq run-guard run-bind*)

lemma *refines-bind-guard-right-end*:
assumes *f-g*: *refines f g s t R*
shows *refines f* (*do* {*res* <− *g*; *guard G*; *return res*}) *s t*
(λ(*r*, *s*) (*q*, *t*). *R* (*r*, *s*) (*q*, *t*) ∧
(*case q of Exception e* ⇒ *True* | *Result* - ⇒ *G t*))
apply (*subst bind-return*[*symmetric*])
apply (*rule refines-bind*[*OF f-g*])
apply (*auto simp*: *refines-bind-guard-right-iff*)
done

lemma *refines-bind-guard-right-end′*:
assumes *f-g*: *refines f g s t R*
shows *refines f* (*do* {*res* <− *g*; *guard* (*G res*); *return res*}) *s t*
(λ(*r*, *s*) (*q*, *t*). *R* (*r*, *s*) (*q*, *t*) ∧
(*case q of Exception e* ⇒ *True* | *Result v* ⇒ *G v t*))
apply (*subst bind-return*[*symmetric*])
apply (*rule refines-bind*[*OF f-g*])
apply (*auto simp*: *refines-bind-guard-right-iff*)
done

16.9.25 *assert-opt*

lemma *run-assert-opt*[*run-spec-monad*, *simp*]:
run (*assert-opt x*) *s* = (*case x of Some v* ⇒ *pure-post-state* (*Result v*, *s*) | *None* ⇒ ⊤)
by (*simp add*: *assert-opt-def run-bind split*: *option.splits*)

lemma *always-progress-assert-opt*[*always-progress-intros*]: *always-progress* (*assert-opt x*)
by (*simp add*: *always-progress-intros assert-opt-def split*: *option.splits*)

lemma *runs-to-assert-opt*[*runs-to-vcg*]: (∃ *v*. *x* = *Some v* ∧ *Q* (*Result v*) *s*) ⟹ *assert-opt x* ⋅ *s* ⦃*Q*⦄
by (*auto simp add*: *runs-to.rep-eq*)

lemma *runs-to-assert-opt-iff*[*runs-to-iff*]:
assert-opt x ⋅ *s* ⦃*Q*⦄ ⟷ (∃ *v*. *x* = *Some v* ∧ *Q* (*Result v*) *s*)
by (*auto simp add*: *runs-to.rep-eq split*: *option.split*)

lemma *runs-to-partial-assert-opt*[*runs-to-vcg*]:
(⋀*v*. *x* = *Some v* ⟹ *Q* (*Result v*) *s*) ⟹ *assert-opt x* ⋅ *s* ?⦃*Q*⦄
by (*auto simp add*: *runs-to-partial.rep-eq split*: *option.split*)

16.9.26 *gets-the*

lemma *run-gets-the'*:
run (*gets-the f*) *s* = (*case f s of Some v* ⇒ *pure-post-state* (*Result v*, *s*) | *None* ⇒ ⊤)
by (*simp add*: *gets-the-def run-bind top-post-state-def pure-post-state-def split*: *option.split*)

lemma *run-gets-the*[*run-spec-monad*, *simp*]:
run (*gets-the f*) *s* = (*case* (*f s*) *of Some v* ⇒ *pure-post-state* (*Result v*, *s*) | *None* ⇒ ⊤)
by (*simp add*: *gets-the-def run-bind*)

lemma *always-progress-gets-the*[*always-progress-intros*]: *always-progress* (*gets-the f*)
by (*simp add*: *always-progress-intros gets-the-def split*: *option.splits*)

lemma *runs-to-gets-the*[*runs-to-vcg*]: (∃ *v*. *f s* = *Some v* ∧ *Q* (*Result v*) *s*) ⟹ *gets-the f* · *s* ⦃ *Q* ⦄
by (*auto simp add*: *runs-to.rep-eq*)

lemma *runs-to-gets-the-iff*[*runs-to-iff*]:
gets-the f · *s* ⦃ *Q* ⦄ ⟷ (∃ *v*. *f s* = *Some v* ∧ *Q* (*Result v*) *s*)
by (*auto simp add*: *runs-to.rep-eq split*: *option.split*)

lemma *runs-to-partial-gets-the*[*runs-to-vcg*]:
(⋀*v*. *f s* = *Some v* ⟹ *Q* (*Result v*) *s*) ⟹ *gets-the f* · *s* ?⦃ *Q* ⦄
by (*auto simp add*: *runs-to-partial.rep-eq split*: *option.split*)

16.9.27 *modify*

lemma *run-modify*[*run-spec-monad*, *simp*]: *run* (*modify f*) *s* = *pure-post-state* (*Result* (), *f s*)
by (*simp add*: *modify-def run-bind*)

lemma *always-progress-modify*[*always-progress-intros*]: *always-progress* (*modify f*)
by (*simp add*: *modify-def always-progress-intros*)

lemma *runs-to-modify*[*runs-to-vcg*]: *modify f* · *s* ⦃ λ*r t*. *r* = *Result* () ∧ *t* = *f s* ⦄
by (*simp add*: *runs-to.rep-eq*)

lemma *runs-to-modify-res*[*runs-to-vcg*]: ((*modify f*)::(*unit*, *'s*) *res-monad*) · *s* ⦃ λ*r t*. *t* = *f s* ⦄
by (*simp add*: *runs-to.rep-eq*)

lemma *runs-to-modify-iff*[*runs-to-iff*]: *modify f* · *s* ⦃ *Q* ⦄ ⟷ *Q* (*Result* ()) (*f s*)
by (*simp add*: *runs-to.rep-eq*)

lemma *runs-to-partial-modify*[*runs-to-vcg*]: *modify f* · *s* ?⦃ λ*r t*. *r* = *Result* () ∧ *t* = *f s* ⦄

by (*simp add*: *runs-to-partial.rep-eq*)

lemma *runs-to-partial-modify-res*[*runs-to-vcg*]:
(($modify\ f$)::($unit$, 's) *res-monad*) • s ?{⦇ $\lambda r\ t.\ t = f\ s$ ⦈}
by (*simp add*: *runs-to-partial.rep-eq*)

lemma *refines-modify*:
$R\ (Result\ ()\text{, }f\ s)\ (Result\ ()\text{, }g\ t) \Longrightarrow refines\ (modify\ f)\ (modify\ g)\ s\ t\ R$
by (*auto simp add*: *refines.rep-eq*)

lemma *rel-spec-modify*:
$R\ (Result\ ()\text{, }f\ s)\ (Result\ ()\text{, }g\ t) \Longrightarrow rel\text{-}spec\ (modify\ f)\ (modify\ g)\ s\ t\ R$
by (*auto simp add*: *rel-spec-iff-refines intro*: *refines-modify*)

16.9.28 condition

lemma *run-condition*[*run-spec-monad*]: $run\ (condition\ c\ f\ g)\ s = (if\ c\ s\ then\ run\ f\ s\ else\ run\ g\ s)$
by (*simp add*: *condition-def run-bind*)

lemma *run-condition-True*[*run-spec-monad*, *simp*]: $c\ s \Longrightarrow run\ (condition\ c\ f\ g)\ s = run\ f\ s$
by (*simp add*: *run-condition*)

lemma *run-condition-False*[*run-spec-monad*, *simp*]: $\neg c\ s \Longrightarrow run\ (condition\ c\ f\ g)\ s = run\ g\ s$
by (*simp add*: *run-condition*)

lemma *always-progress-condition*[*always-progress-intros*]:
$always\text{-}progress\ f \Longrightarrow always\text{-}progress\ g \Longrightarrow always\text{-}progress\ (condition\ c\ f\ g)$
by (*auto simp add*: *always-progress-def run-condition*)

lemma *condition-swap*: $(condition\ C\ A\ B) = (condition\ (\lambda s.\ \neg\ C\ s)\ B\ A)$
by (*rule spec-monad-ext*) (*clarsimp simp add*: *run-condition*)

lemma *condition-fail-rhs*: $(condition\ C\ X\ fail) = (guard\ C >>= (\lambda\text{-}.\ X))$
by (*rule spec-monad-ext*) (*simp add*: *run-bind run-guard run-condition*)

lemma *condition-fail-lhs*: $(condition\ C\ fail\ X) = (guard\ (\lambda s.\ \neg\ C\ s) >>= (\lambda\text{-}.\ X))$
by (*metis condition-fail-rhs condition-swap*)

lemma *condition-bind-fail*[*simp*]:
$(condition\ C\ A\ B >>= (\lambda\text{-}.\ fail)) = condition\ C\ (A >>= (\lambda\text{-}.\ fail))\ (B >>= (\lambda\text{-}.\ fail))$
apply (*rule spec-monad-ext*)
apply (*clarsimp simp add*: *run-condition run-bind*)
done

lemma *condition-True*[*simp*]: $condition\ (\lambda\text{-}.\ True)\ f\ g = f$

apply (*rule spec-monad-ext*)
apply (*clarsimp simp add*: *run-condition run-bind*)
done

lemma *condition-False*[*simp*]: *condition* (λ-. *False*) *f g* = *g*
apply (*rule spec-monad-ext*)
apply (*clarsimp simp add*: *run-condition run-bind*)
done

lemma *le-condition-runI*:
(⋀*s*. *c s* ⟹ *run h s* ≤ *run f s*) ⟹ (⋀*s*. ¬ *c s* ⟹ *run h s* ≤ *run g s*)
⟹ *h* ≤ *condition c f g*
by (*simp add*: *le-spec-monad-runI run-condition*)

lemma *mono-condition-spec-monad*:
mono T ⟹ *mono F* ⟹ *mono* (λ*x*. *condition C* (*F x*) (*T x*))
by (*auto simp*: *condition-def intro*!: *mono-bind-spec-monad mono-const*)

lemma *mono-condition*: *f* ≤ *f'* ⟹ *g* ≤ *g'* ⟹ *condition c f g* ≤ *condition c f' g'*
by (*simp add*: *le-fun-def less-eq-spec-monad.rep-eq run-condition*)

lemma *monotone-condition-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*) ⟹ (*monotone R* (≤) (λ*f'*. *g f'*))
⟹ *monotone R* (≤) (λ*f'*. *condition c* (*f f'*) (*g f'*))
by (*auto simp add*: *monotone-def intro*!: *mono-condition*)

lemma *monotone-condition-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ (*monotone R* (≥) (λ*f'*. *g f'*))
⟹ *monotone R* (≥) (λ*f'*. *condition c* (*f f'*) (*g f'*))
by (*auto simp add*: *monotone-def intro*!: *mono-condition*)

lemma *runs-to-condition*[*runs-to-vcg*]:
(*c s* ⟹ *f* · *s* ⦃ *Q* ⦄) ⟹ (¬ *c s* ⟹ *g* · *s* ⦃ *Q* ⦄) ⟹ *condition c f g* · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to.rep-eq run-condition*)

lemma *runs-to-condition-iff*[*runs-to-iff*]:
condition c f g · *s* ⦃ *Q* ⦄ ⟷ (*if c s then f* · *s* ⦃ *Q* ⦄ *else g* · *s* ⦃ *Q* ⦄)
by (*simp add*: *runs-to.rep-eq run-condition*)

lemma *runs-to-partial-condition*[*runs-to-vcg*]:
(*c s* ⟹ *f* · *s* ?⦃ *Q* ⦄) ⟹ (¬ *c s* ⟹ *g* · *s* ?⦃ *Q* ⦄) ⟹ *condition c f g* · *s* ?⦃ *Q*
⦄
by (*simp add*: *runs-to-partial.rep-eq run-condition*)

lemma *refines-condition-iff*:
assumes *c' s'* ⟷ *c s*
shows *refines* (*condition c f g*) (*condition c' f' g'*) *s s' R* ⟷
(*if c' s' then refines f f' s s' R else refines g g' s s' R*)
using *assms*

by (*auto simp add*: *refines.rep-eq run-condition*)

lemma *refines-condition*:
P s $\longleftrightarrow$ *P' s'* $\Longrightarrow$
(*P s* $\Longrightarrow$ *P' s'* $\Longrightarrow$ *refines f f' s s' R*) $\Longrightarrow$
(¬ *P s* $\Longrightarrow$ ¬ *P' s'* $\Longrightarrow$ *refines g g' s s' R*) $\Longrightarrow$
refines (*condition P f g*) (*condition P' f' g'*) *s s' R*
using *refines-condition-iff*
by *metis*

lemma *refines-condition-TrueI*:
assumes *c' s'* = *c s* **and** *c' s' refines f f' s s' R*
shows *refines* (*condition c f g*) (*condition c' f' g'*) *s s' R*
by (*simp add*: *refines-condition-iff*[**where** *c'*=*c'* **and** *c*=*c* **and** *s'*=*s'* **and** *s*=*s*, *OF assms*(*1*)] *assms*(*2*, *3*))

lemma *refines-condition-FalseI*:
assumes *c' s'* = *c s* **and** ¬*c' s' refines g g' s s' R*
shows *refines* (*condition c f g*) (*condition c' f' g'*) *s s' R*
by (*simp add*: *refines-condition-iff*[**where** *c'*=*c'* **and** *c*=*c* **and** *s'*=*s'* **and** *s*=*s*, *OF assms*(*1*)] *assms*(*2*, *3*))

lemma *refines-condition-bind-left*:
refines (*condition C T F* $\gg\!=$ *X*) *Y s t R* $\longleftrightarrow$
(*C s* $\longrightarrow$ *refines* (*T* $\gg\!=$ *X*) *Y s t R*) ∧ (¬ *C s* $\longrightarrow$ *refines* (*F* $\gg\!=$ *X*) *Y s t R*)
by (*simp add*: *refines.rep-eq run-bind run-condition*)

lemma *refines-condition-bind-right*:
refines X (*condition C T F* $\gg\!=$ *Y*) *s t R* $\longleftrightarrow$
(*C t* $\longrightarrow$ *refines X* (*T* $\gg\!=$ *Y*) *s t R*) ∧ (¬ *C t* $\longrightarrow$ *refines X* (*F* $\gg\!=$ *Y*) *s t R*)
by (*simp add*: *refines.rep-eq run-bind run-condition*)

lemma *rel-spec-condition-iff*:
assumes *c' s'* $\longleftrightarrow$ *c s*
shows *rel-spec* (*condition c f g*) (*condition c' f' g'*) *s s' R* $\longleftrightarrow$
(*if c' s' then rel-spec f f' s s' R else rel-spec g g' s s' R*)
using *assms*
by (*auto simp add*: *rel-spec-def run-condition*)

lemma *rel-spec-condition*:
P s $\longleftrightarrow$ *P' s'* $\Longrightarrow$
(*P s* $\Longrightarrow$ *P' s'* $\Longrightarrow$ *rel-spec f f' s s' R*) $\Longrightarrow$
(¬ *P s* $\Longrightarrow$ ¬ *P' s'* $\Longrightarrow$ *rel-spec g g' s s' R*) $\Longrightarrow$
rel-spec (*condition P f g*) (*condition P' f' g'*) *s s' R*
using *rel-spec-condition-iff*
by *metis*

lemma *rel-spec-condition-TrueI*:
assumes *c' s'* = *c s* **and** *c' s' rel-spec f f' s s' R*

shows *rel-spec* (*condition c f g*) (*condition c′ f′ g′*) *s s′ R*
by (*simp add*: *rel-spec-condition-iff*[**where** *c′*=*c′* **and** *c*=*c* **and** *s′*=*s′* **and** *s*=*s*, *OF assms*(*1*)] *assms*(*2*, *3*))

lemma *rel-spec-condition-FalseI*:
assumes *c′ s′* = *c s* **and** ¬*c′ s′ rel-spec g g′ s s′ R*
shows *rel-spec* (*condition c f g*) (*condition c′ f′ g′*) *s s′ R*
by (*simp add*: *rel-spec-condition-iff*[**where** *c′*=*c′* **and** *c*=*c* **and** *s′*=*s′* **and** *s*=*s*, *OF assms*(*1*)] *assms*(*2*, *3*))

lemma *refines-condition-left*:
(*P s* ⟹ *refines f h s t R*) ⟹ (¬ *P s* ⟹ *refines g h s t R*) ⟹
refines (*condition P f g*) *h s t R*
by (*simp add*: *refines.rep-eq run-condition*)

lemma *rel-spec-condition-left*:
(*P s* ⟹ *rel-spec f h s t R*) ⟹ (¬ *P s* ⟹ *rel-spec g h s t R*) ⟹
rel-spec (*condition P f g*) *h s t R*
by (*auto simp add*: *rel-spec-def run-condition*)

lemma *refines-condition-true*:
P t ⟹ *refines f g s t R* ⟹ *refines f* (*condition P g h*) *s t R*
by (*simp add*: *refines.rep-eq run-condition*)

lemma *rel-spec-condition-true*:
P t ⟹ *rel-spec f g s t R* ⟹
rel-spec f (*condition P g h*) *s t R*
by (*auto simp add*: *rel-spec-def run-condition*)

lemma *refines-condition-false*:
¬ *P t* ⟹ *refines f h s t R* ⟹
refines f (*condition P g h*) *s t R*
by (*simp add*: *refines.rep-eq run-condition*)

lemma *rel-spec-condition-false*:
¬ *P t* ⟹ *rel-spec f h s t R* ⟹
rel-spec f (*condition P g h*) *s t R*
by (*auto simp add*: *rel-spec-def run-condition*)

lemma *condition-bind*:
(*condition P f g* >>= *h*) = *condition P* (*f* >>= *h*) (*g* >>= *h*)
by (*simp add*: *spec-monad-ext-iff run-condition run-bind*)

lemma *rel-spec-monad-condition*:
assumes *rel-fun R* (=) *P P′*
and *rel-spec-monad R Q f f′*
and *rel-spec-monad R Q g g′*
shows *rel-spec-monad R Q* (*condition P f g*) (*condition P′ f′ g′*)
using *assms*

by (*auto simp add*: *rel-spec-monad-def run-condition rel-fun-def*)

lemma *rel-spec-monad-condition-const*:
$P \longleftrightarrow P' \Longrightarrow (P \Longrightarrow \textit{rel-spec-monad}\ R\ Q\ f\ f') \Longrightarrow$
$(\neg\ P \Longrightarrow \textit{rel-spec-monad}\ R\ Q\ g\ g') \Longrightarrow$
rel-spec-monad R Q (*condition* (λ-. *P*) *f g*) (*condition* (λ-. *P'*) *f' g'*)
by (*cases P*) (*auto simp add*: *condition-def*)

16.9.29 *when*

lemma *run-when*[*run-spec-monad*]:
run (*when c f*) *s* = (*if c then run f s else pure-post-state* (*Result* (), *s*))
unfolding *when-def* **by** *simp*

lemma *always-progress-when*[*always-progress-intros*]:
always-progress f ⟹ *always-progress* (*when c f*)
unfolding *when-def* **by** (*simp add*: *always-progress-intros*)

lemma *runs-to-when*[*runs-to-vcg*]:
(*c* ⟹ *f* · *s* ⦃ *Q* ⦄) ⟹ (¬ *c* ⟹ *Q* (*Result* ()) *s*) ⟹ *when c f* · *s* ⦃ *Q* ⦄
by (*auto simp add*: *runs-to.rep-eq run-when*)

lemma *runs-to-when-iff*[*runs-to-iff*]:
(*when c f*) · *s* ⦃ *Q* ⦄ ⟷ (*if c then f* · *s* ⦃ *Q* ⦄ *else Q* (*Result* ()) *s*)
by (*auto simp add*: *runs-to.rep-eq run-when*)

lemma *runs-to-partial-when*[*runs-to-vcg*]:
(*c* ⟹ *f* · *s* ?⦃ *Q* ⦄) ⟹ (¬ *c* ⟹ *Q* (*Result* ()) *s*) ⟹ *when c f* · *s* ?⦃ *Q* ⦄
by (*auto simp add*: *runs-to-partial.rep-eq run-when*)

lemma *mono-when*: *f* ≤ *f'* ⟹ *when c f* ≤ *when c f'*
unfolding *when-def* **by** (*simp add*: *mono-condition*)

lemma *monotone-when-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*)
⟹ *monotone R* (≤) (λ*f'*. *when c* (*f f'*))
unfolding *when-def* **by** (*simp add*: *monotone-condition-le*)

lemma *monotone-when-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*)
⟹ *monotone R* (≥) (λ*f'*. *when c* (*f f'*))
unfolding *when-def* **by** (*simp add*: *monotone-condition-ge*)

lemma *when-True*[*simp*]: *when True f* = *f*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-when*)
done

lemma *when-False*[*simp*]: *when False f* = *return* ()

apply (*rule spec-monad-ext*)
apply (*simp add*: *run-when*)
done

16.9.30 While

context fixes $C :: 'a \Rightarrow 's \Rightarrow bool$ **and** $B :: 'a \Rightarrow ('e::default, 'a, 's)$ *spec-monad*
begin

definition *whileLoop* :: $'a \Rightarrow ('e, 'a, 's)$ *spec-monad* **where**
whileLoop =
gfp ($\lambda W\ a.$ *condition* (*C a*) (*bind* (*B a*) *W*) (*return a*))
— Collapses to *Failure* in case of any non terminating computation.

definition *whileLoop-finite* :: $'a \Rightarrow ('e, 'a, 's)$ *spec-monad* **where**
whileLoop-finite =
lfp ($\lambda W\ a.$ *condition* (*C a*) (*bind* (*B a*) *W*) (*return a*))
— Does not collapse to *Failure* in presence of a non terminating computation. *Failure* can still occur when the body fails in some iteration. It captures the outcomes of all terminating and thus finite computations.

inductive *whileLoop-terminates* :: $'a \Rightarrow 's \Rightarrow bool$ **where**
step: $\bigwedge a\ s.$ (*C a s* $\Longrightarrow$ *B a* $\bullet$ *s* $?\{\!|\ \lambda v\ s.\ \forall a.\ v = Result\ a \longrightarrow$ *whileLoop-terminates a s* $|\!\}$) $\Longrightarrow$
whileLoop-terminates a s
— This is weaker than *run* (*whileLoop a*) *s* $\neq \top$: as it uses partial correctness

lemma *mono-whileLoop-functional*:
mono ($\lambda W\ a.$ *condition* (*C a*) (*bind* (*B a*) *W*) (*return a*))
by (*intro mono-app mono-lam mono-const mono-bind-spec-monad mono-condition-spec-monad*)

lemma *whileLoop-unroll*:
whileLoop a =
condition (*C a*) (*bind* (*B a*) *whileLoop*) (*return a*)
unfolding *whileLoop-def*
by (*subst gfp-unfold*[*OF mono-whileLoop-functional*]) *simp*

lemma *whileLoop-finite-unfold*:
whileLoop-finite a =
condition (*C a*) (*bind* (*B a*) *whileLoop-finite*) (*return a*)
unfolding *whileLoop-finite-def*
by (*subst lfp-unfold*[*OF mono-whileLoop-functional*]) *simp*

lemma *whileLoop-ne-Failure*:
(*C a s* $\Longrightarrow$ *B a* $\bullet$ *s* $\{\!|\ \lambda x\ s.\ \forall a.\ x = Result\ a \longrightarrow$ *run* (*whileLoop a*) *s* $\neq$ *Failure* $|\!\}$) $\Longrightarrow$
run (*whileLoop a*) *s* $\neq$ *Failure*
by (*subst whileLoop-unroll*)
(*simp add*: *run-condition run-bind-eq-top-iff flip*: *top-post-state-def*)

lemma *whileLoop-ne-top-induct*[*consumes 1*, *case-names step*]:
assumes *a-s*: *run* (*whileLoop a*) *s* ≠ ⊤
and *step*: ⋀*a s*. (*C a s* ⟹ *B a* · *s* ⦃ λ*x s*. ∀ *a*. *x* = *Result a* ⟶ *P a s* ⦄) ⟹ *P a s*
shows *P a s*
proof −
have (λ*a*. *Spec* (λ*s*. *if P a s then* ⊥ *else* ⊤)) ≤ *whileLoop*
unfolding *whileLoop-def*
by (*intro gfp-upperbound*)
(*auto intro*: *step runs-to-weaken*
simp: *le-fun-def less-eq-spec-monad.rep-eq run-condition top-unique run-bind-eq-top-iff*)
with *a-s* **show** *?thesis*
by (*auto simp add*: *le-fun-def less-eq-spec-monad.rep-eq top-unique top-post-state-def*
split: *if-splits*)
qed

lemma *runs-to-whileLoop*:
assumes *R*: *wf R*
assumes ∗: *I* (*Result a*) *s*
assumes *P-Result*: ⋀*a s*. ¬ *C a s* ⟹ *I* (*Result a*) *s* ⟹ *P* (*Result a*) *s*
assumes *P-Exception*: ⋀*a s*. *a* ≠ *default* ⟹ *I* (*Exception a*) *s* ⟹ *P* (*Exception a*) *s*
assumes *B*: ⋀*a s*. *C a s* ⟹ *I* (*Result a*) *s* ⟹
B a · *s* ⦃λ*r t*. *I r t* ∧ (∀ *b*. *r* = *Result b* ⟶ ((*b*, *t*), (*a*, *s*)) ∈ *R*)⦄
shows *whileLoop a* · *s* ⦃*P*⦄
proof (*use R* ∗ **in** ‹*induction x*≡(*a*, *s*) *arbitrary*: *a s*›)
case (*less a s*)
show *?case*
by (*auto simp*: *P-Result P-Exception whileLoop-unroll*[*of a*]
runs-to-condition-iff runs-to-bind-iff less
intro!: *runs-to-weaken*[*OF B*])
qed

lemma *runs-to-whileLoop-finite*:
assumes ∗: *I* (*Result a*) *s*
assumes *P-Result*: ⋀*a s*. ¬ *C a s* ⟹ *I* (*Result a*) *s* ⟹ *P* (*Result a*) *s*
assumes *P-Exception*: ⋀*a s*. *a* ≠ *default* ⟹ *I* (*Exception a*) *s* ⟹ *P* (*Exception a*) *s*
assumes *B*: ⋀*a s*. *C a s* ⟹ *I* (*Result a*) *s* ⟹ *B a* · *s* ⦃ *I* ⦄
shows *whileLoop-finite a* · *s* ⦃*P*⦄
proof −
have *whileLoop-finite a* · *s*
⦃ λ*x s*. *I x s* ∧ (∀ *a*. *x* = *Result a* ⟶ *C a s* ⟶ *P* (*Result a*) *s*) ⦄
unfolding *whileLoop-finite-def*
apply (*rule runs-to-lfp*[*OF mono-whileLoop-functional*, *of* λ*a*. *I* (*Result a*), *OF* ∗])
subgoal premises *prems* **for** *W x s*
using *prems*(*2*)

```
    by (intro runs-to-condition runs-to-bind runs-to-yield-iff
          runs-to-weaken[OF B] runs-to-weaken[OF prems(1)] conjI allI impI)
       simp-all
   done
 then show ?thesis
   apply (rule runs-to-weaken)
   subgoal for r s
    by (cases r; cases ∃ a. r = Result a ∧ C a s; simp add: P-Result P-Exception)
   done
qed

lemma runs-to-partial-whileLoop-finite:
 assumes *: I (Result a) s
 assumes B: ⋀a s. C a s ⟹ I (Result a) s ⟹ (B a) • s ?{| I |}
 assumes P-Result: ⋀a s. ¬ C a s ⟹ I (Result a) s ⟹ P (Result a) s
 assumes P-Exception: ⋀a s. a ≠ default ⟹ I (Exception a) s ⟹ P (Exception
a) s
 shows whileLoop-finite a • s ?{| P |}
proof −
 have whileLoop-finite a • s {|P|} if **: run (whileLoop-finite a) s ≠ ⊤
   apply (rule runs-to-whileLoop-finite[where
        I=λx s. I x s ∧ (∀ a. x = Result a ⟶ run (whileLoop-finite a) s ≠ ⊤)])
     apply (auto simp: * ** P-Result P-Exception)[3]
   subgoal for a s using B[of a s]
     by (subst (asm) whileLoop-finite-unfold)
      (auto simp: runs-to-conj run-bind-eq-top-iff dest: runs-to-of-runs-to-partial-runs-to')
   done
 then show ?thesis
   by (auto simp: runs-to-partial-alt top-post-state-def)
qed

lemma whileLoop-finite-eq-whileLoop-of-whileLoop-terminates:
 assumes whileLoop-terminates a s
 shows run (whileLoop-finite a) s = run (whileLoop a) s
proof (use assms in ‹induction›)
 case (step a s)
 show ?case
   apply (subst whileLoop-unroll)
   apply (subst whileLoop-finite-unfold)
  apply (auto simp: run-condition intro!: run-bind-cong[OF refl runs-to-partial-weaken,
OF step])
   done
qed

lemma whileLoop-terminates-of-succeeds:
 run (whileLoop a) s ≠ ⊤ ⟹ whileLoop-terminates a s
 by (induction rule: whileLoop-ne-top-induct)
    (auto intro: whileLoop-terminates.step runs-to-partial-of-runs-to)
```

lemma *whileLoop-finite-eq-whileLoop*:
  run (*whileLoop a*) *s* ≠ ⊤ ⟹ *run* (*whileLoop-finite a*) *s* = *run* (*whileLoop a*) *s*
  by (*rule whileLoop-finite-eq-whileLoop-of-whileLoop-terminates*[*OF*
    whileLoop-terminates-of-succeeds])

lemma *runs-to-whileLoop-of-runs-to-whileLoop-finite-if-terminates*:
  whileLoop-terminates i s ⟹ *whileLoop-finite i* · *s* ⦃*Q*⦄ ⟹ *whileLoop i* · *s* ⦃*Q*⦄
  unfolding *runs-to.rep-eq whileLoop-finite-eq-whileLoop-of-whileLoop-terminates*
.

lemma *whileLoop-finite-le-whileLoop*: *whileLoop-finite a* ≤ *whileLoop a*
  using *le-fun-def lfp-le-gfp mono-whileLoop-functional*
  unfolding *whileLoop-finite-def whileLoop-def*
  by *fastforce*

lemma *runs-to-partial-whileLoop-finite-whileLoop*:
  whileLoop-finite i · *s ?*⦃*Q*⦄ ⟹ *whileLoop i* · *s ?*⦃*Q*⦄
  by (*cases run* (*whileLoop i*) *s* = ⊤)
    (*auto simp*:
    runs-to.rep-eq runs-to-partial-alt whileLoop-finite-eq-whileLoop top-post-state-def)

lemma *runs-to-partial-whileLoop*:
  assumes *I* (*Result a*) *s*
  assumes ⋀*a s*. ¬ *C a s* ⟹ *I* (*Result a*) *s* ⟹ *P* (*Result a*) *s*
  assumes ⋀*a s*. *a* ≠ *default* ⟹ *I* (*Exception a*) *s* ⟹ *P* (*Exception a*) *s*
  assumes ⋀*a s*. *C a s* ⟹ *I* (*Result a*) *s* ⟹ (*B a*) · *s ?*⦃ *I* ⦄
  shows *whileLoop a* · *s ?*⦃ *P* ⦄
  using *assms*(*1*,*4*,*2*,*3*)
 by (*rule runs-to-partial-whileLoop-finite-whileLoop*[*OF runs-to-partial-whileLoop-finite*])

lemma *always-progress-whileLoop*[*always-progress-intros*]:
  assumes *B*: (⋀*v*. *always-progress* (*B v*))
  shows *always-progress* (*whileLoop a*)
  unfolding *always-progress.rep-eq*
proof
  fix *s* **have** *run* (*whileLoop a*) *s* ≠ ⊥ **if** ∗: *run* (*whileLoop a*) *s* ≠ ⊤
  proof (*use* ∗ **in** ‹*induction rule*: *whileLoop-ne-top-induct*›)
    case (*step a s*) **then show** *?case*
      apply (*subst whileLoop-unroll*)
        apply (*simp add*: *run-condition run-bind bind-post-state-eq-bot prod-eq-iff*
runs-to.rep-eq
                  split: *prod.splits exception-or-result-splits*)
      apply *clarsimp*
      subgoal premises *prems*
        using *holds-post-state-combine*[*OF prems*(*1*,*3*), **where** *R*=λ*x*. *False*] *B*
      by (*auto simp*: *holds-post-state-False exception-or-result-nchotomy always-progress.rep-eq*)
      done
  qed
  then show *run* (*whileLoop a*) *s* ≠ ⊥

by (*cases run* (*whileLoop a*) *s* = *Failure*) (*auto simp*: *bot-post-state-def*)
qed

lemma *runs-to-partial-whileLoop-cond-false*:
(*whileLoop I*) · *s* ?⦃ λ*r t*. ∀ *a*. *r* = *Result a* ⟶ ¬ *C a t* ⦄
by (*rule runs-to-partial-whileLoop*[*of* λ- -. *True*]) *simp-all*

end

context
fixes *R*
fixes *C* :: *'a* ⇒ *'s* ⇒ *bool* **and** *B* :: *'a* ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad*
and *C'* :: *'b* ⇒ *'t* ⇒ *bool* **and** *B'* :: *'b* ⇒ (*'f*::*default*, *'b*, *'t*) *spec-monad*
assumes *C*: ⋀*x s x' s'*. *R* (*Result x*, *s*) (*Result x'*, *s'*) ⟹ *C x s* ⟷ *C' x' s'*
assumes *B*: ⋀*x s x' s'*. *R* (*Result x*, *s*) (*Result x'*, *s'*) ⟹ *C x s* ⟹ *C' x' s'* ⟹
refines (*B x*) (*B' x'*) *s s' R*
assumes *R*: ⋀*v s v' s'*. *R* (*v*, *s*) (*v'*, *s'*) ⟹ (∃ *x*. *v* = *Result x*) ⟷ (∃ *x'*. *v'* = *Result x'*)
begin

lemma *refines-whileLoop-finite-strong*:
assumes *x-x'*: *R* (*Result x*, *s*) (*Result x'*, *s'*)
shows *refines* (*whileLoop-finite C B x*) (*whileLoop-finite C' B' x'*) *s s'*
(λ(*r*, *s*) (*r'*, *s'*). (∀ *v*. *r* = *Result v* ⟶ (∃ *v'*. *r'* = *Result v'* ∧ ¬ *C v s* ∧ ¬ *C' v' s'*)) ∧
R (*r*, *s*) (*r'*, *s'*))
(**is** *refines* - - - - *?R*)
proof −
let *?P* = λ*p*. ∀ *x s x' s'*. *R* (*Result x*, *s*) (*Result x'*, *s'*) ⟶
refines (*p x*) (*whileLoop-finite C' B' x'*) *s s' ?R*

have *?P* (*whileLoop-finite C B*)
unfolding *whileLoop-finite-def*[*of C B*]
proof (*rule lfp-ordinal-induct*[*OF mono-whileLoop-functional*], *intro allI impI*)
fix *S x s x' s'* **assume** *S*: *?P S* **and** *x-x'*: *R* (*Result x*, *s*) (*Result x'*, *s'*)
from *x-x'* **show**
refines (*condition* (*C x*) (*B x* ⤜ *S*) (*return x*)) (*whileLoop-finite C' B' x'*) *s s' ?R*
by (*subst whileLoop-finite-unfold*)
(*auto dest*: *R simp*: *C*[*OF x-x'*] *refines-condition-iff*
intro!: *refines-bind'*[*OF refines-mono*[*OF - B*[*OF x-x'*]]] *S*[*rule-format*])
qed (*simp add*: *refines-Sup1*)
with *x-x'* **show** *?thesis* **by** *simp*
qed

lemma *refines-whileLoop-finite*:
assumes *x-x'*: *R* (*Result x*, *s*) (*Result x'*, *s'*)
shows *refines* (*whileLoop-finite C B x*) (*whileLoop-finite C' B' x'*) *s s' R*
by (*rule refines-mono*[*OF - refines-whileLoop-finite-strong*, *OF - x-x'*]) *simp*

lemma *whileLoop-succeeds-terminates-of-refines*:
assumes *run* (*whileLoop* C' B' x') $s' \neq \top$
shows R (*Result* x, s) (*Result* x', s') $\Longrightarrow$ *run* (*whileLoop* C B x) $s \neq$ *Failure*
proof (*use assms* **in** ‹*induction arbitrary*: x s *rule*: *whileLoop-ne-top-induct*›)
case (*step* a' s' a s)

show *?case*
proof (*rule whileLoop-ne-Failure*)
assume C a s
with C[*of* a s a' s'] *step.prems* **have** C' a' s' **by** *simp*

show B a $\cdot$ s {| λx s. $\forall$ a. x = *Result* a $\longrightarrow$ *run* (*whileLoop* C B a) $s \neq$ *Failure* |}
using *step.IH*[*OF* ‹C' a' s'›] B[*OF step.prems* ‹C a s› ‹C' a' s'›]
by (*rule runs-to-refines*) (*metis* R)
qed
qed

lemma *refines-whileLoop-strong*:
assumes *x-x'*: R (*Result* x, s) (*Result* x', s')
shows *refines* (*whileLoop* C B x) (*whileLoop* C' B' x') s s'
(λ(r, s) (r', s'). ($\forall$ v. r = *Result* v $\longrightarrow$ ($\exists$ v'. r' = *Result* v' $\wedge$ $\neg$ C v s $\wedge$ $\neg$ C' v' s')) $\wedge$
R (r, s) (r', s'))
apply (*cases run* (*whileLoop* C' B' x') s' = *Failure*)
subgoal by (*simp add*: *refines.rep-eq*)
subgoal
using
whileLoop-succeeds-terminates-of-refines[*OF* - *x-x'*]
refines-whileLoop-finite-strong[*OF x-x'*]
by (*simp add*: *refines.rep-eq whileLoop-finite-eq-whileLoop top-post-state-def*)
done

lemma *refines-whileLoop*:
assumes *x-x'*: R (*Result* x, s) (*Result* x', s')
shows *refines* (*whileLoop* C B x) (*whileLoop* C' B' x') s s' R
by (*rule refines-mono*[*OF* - *refines-whileLoop-strong*, *OF* - *x-x'*]) *simp*

end

lemma *runs-to-whileLoop-res'*:
assumes R: *wf* R
assumes $*$: I a s
assumes *P-Result*: $\bigwedge a$ s. $\neg$ C a s $\Longrightarrow$ I a s $\Longrightarrow$ P (*Result* a) s
assumes B: $\bigwedge a$ s. C a s $\Longrightarrow$ I a s $\Longrightarrow$
B a $\cdot$ s {|λr t. ($\forall$ b. r = *Result* b $\longrightarrow$ I b t $\wedge$ ((b, t), (a, s)) $\in$ R)|}
shows (*whileLoop* C B a::('a, 's) *res-monad*) $\cdot$ s {|P|}
apply (*rule runs-to-whileLoop*[*OF* R, **where** I = λ*Exception* - $\Rightarrow$ (λ-. *False*) |

Result v ⇒ I v])
subgoal using ∗ **by** *auto*
subgoal using *P-Result* **by** *auto*
subgoal by *auto*
subgoal for *a b* **by** (*rule B*[*of a b*, *THEN runs-to-weaken*]) *auto*
done

lemma *runs-to-whileLoop-res*:
assumes *B*: ⋀*a s*. *C a s* ⟹ *I a s* ⟹
B a · *s* {|λ*Res r t*. *I r t* ∧ ((*r*, *t*), (*a*, *s*)) ∈ *R*|}
assumes *P-Result*: ⋀*a s*. *I a s* ⟹ ¬ *C a s* ⟹ *P a s*
assumes *R*: *wf R*
assumes ∗: *I a s*
shows (*whileLoop C B a*::('*a*, '*s*) *res-monad*) · *s* {|λ*Res r*. *P r*|}
apply (*rule runs-to-whileLoop-res'* [**where** *R* = *R* **and** *I* = *I* **and** *B* = *B*, *OF R* ∗])
using *B P-Result* **by** *auto*

lemma *runs-to-whileLoop-variant-res*:
assumes *I*: ⋀*r s c*. *I r s c* ⟹
C r s ⟹ (*B r*) · *s* {| λ*Res q t*. ∃ *c'*. *I q t c'* ∧ (*c'*, *c*) ∈ *R* |}
assumes *Q*: ⋀*r s c*. *I r s c* ⟹ ¬ *C r s* ⟹ *Q r s*
assumes *R*: *wf R*
shows *I r s c* ⟹ (*whileLoop C B r*::('*a*, '*s*) *res-monad*) · *s* {|λ*Res r*. *Q r* |}
proof (*induction arbitrary*: *r s rule*: *wf-induct*[*OF R*])
case (*1 c*) **show** *?case*
apply (*subst whileLoop-unroll*)
supply *I*[**where** *c*=*c*, *runs-to-vcg*]
apply (*runs-to-vcg*)
subgoal by (*simp add*: *1*)
subgoal using *1* **by** *simp*
subgoal using *Q 1* **by** *blast*
done
qed

lemma *runs-to-whileLoop-inc-res*:
assumes ∗: ⋀*i*. *i* < *M* ⟹ (*B* (*F i*)) · (*S i*) {| λ*Res r' t'*. *r'* = (*F* (*Suc i*)) ∧ *t'* = (*S* (*Suc i*)) |}
and [*simp*]: ⋀*i*. *i* ≤ *M* ⟹ *C* (*F i*) (*S i*) ⟷ *i* < *M*
and [*simp*]: *i* = *F 0 si* = *S 0 t* = *F M st* = *S M*
shows (*whileLoop C B i*::('*a*, '*s*) *res-monad*) · *si* {| λ*Res r' t'*. *r'* = *t* ∧ *t'* = *st* |}
apply (*rule runs-to-whileLoop-variant-res*[**where** *I*=λ*r t i*. *r* = *F i* ∧ *t* = *S i* ∧ *i* ≤ *M* **and** *c*=*0*,
OF - - wf-nat-bound[*of M*]])
apply *simp-all*
apply (*rule runs-to-weaken*[*OF* ∗])
apply *auto*
done

lemma *runs-to-whileLoop-dec-res*:
assumes $*$: $\bigwedge i{::}nat.\ i > 0 \Longrightarrow i \leq M \Longrightarrow$
$(B\ (F\ i)) \cdot (S\ i)\ \{\!|\ \lambda Res\ r'\ t'.\ r' = (F\ (i - 1)) \wedge t' = (S\ (i - 1))\ |\!\}$
and [*simp*]: $\bigwedge i.\ C\ (F\ i)\ (S\ i) \longleftrightarrow i > 0$
and [*simp*]: $i = F\ M\ si = S\ M\ t = F\ 0\ st = S\ 0$
shows $(whileLoop\ C\ B\ i{::}('a, 's)\ res\text{-}monad) \cdot si\ \{\!|\ \lambda Res\ r'\ t'.\ r' = t \wedge t' = st\ |\!\}$
apply (*rule runs-to-whileLoop-variant-res*[**where** $I{=}\lambda r\ t\ i.\ r = F\ i \wedge t = S\ i \wedge i \leq M$ **and** $c{=}M$,
OF - - wf-measure[*of* $\lambda x.\ x$]])
apply (*fastforce intro*!: *runs-to-weaken*[*OF* $*$])+
done

lemma *runs-to-whileLoop-exn*:
assumes R: *wf R*
assumes $*$: $I\ (Result\ a)\ s$
assumes *P-Result*: $\bigwedge a\ s.\ \neg\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow P\ (Result\ a)\ s$
assumes *P-Exn*: $\bigwedge a\ s.\ I\ (Exn\ a)\ s \Longrightarrow P\ (Exn\ a)\ s$
assumes B: $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow$
$B\ a\ \cdot s\ \{\!|\lambda r\ t.\ I\ r\ t \wedge (\forall b.\ r = Result\ b \longrightarrow ((b,\ t),\ (a,\ s)) \in R)|\!\}$
shows $whileLoop\ C\ B\ a \cdot s\ \{\!|P|\!\}$
apply (*rule runs-to-whileLoop*[**where** $I = I$, *OF R* $*$])
subgoal using *P-Result* **by** *auto*
subgoal using *P-Exn* **by** (*auto simp add*: *default-option-def Exn-def*)
subgoal using B **by** *blast*
done

lemma *runs-to-whileLoop-exn'*:
assumes B: $\bigwedge a\ s.\ I\ (Result\ a)\ s \Longrightarrow C\ a\ s \Longrightarrow$
$B\ a\ \cdot s\ \{\!|\lambda r\ t.\ I\ r\ t \wedge (\forall b.\ r = Result\ b \longrightarrow ((b,\ t),\ (a,\ s)) \in R)|\!\}$
assumes *P-Result*: $\bigwedge a\ s.\ \ I\ (Result\ a)\ s\ \Longrightarrow \neg\ C\ a\ s \Longrightarrow P\ (Result\ a)\ s$
assumes *P-Exn*: $\bigwedge a\ s.\ I\ (Exn\ a)\ s \Longrightarrow P\ (Exn\ a)\ s$
assumes R: *wf R*
assumes $*$: $I\ (Result\ a)\ s$
shows $whileLoop\ C\ B\ a \cdot s\ \{\!|P|\!\}$
by (*rule runs-to-whileLoop-exn*[**where** $R{=}R$ **and** $I{=}I$ **and** $B{=}B$ **and** $C{=}C$ **and** $P{=}P$,
OF R $*$ *P-Result P-Exn B*])

lemma *runs-to-partial-whileLoop-res*:
assumes $*$: $I\ a\ s$
assumes *P-Result*: $\bigwedge a\ s.\ \neg\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow P\ (Result\ a)\ s$
assumes B: $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow$
$(B\ a) \cdot s\ ?\{\!|\lambda r\ t.\ (\forall b.\ r = Result\ b \longrightarrow I\ b\ t)|\!\}$
shows $(whileLoop\ C\ B\ a{::}('a, 's)\ res\text{-}monad) \cdot s\ ?\{\!|P|\!\}$
apply (*rule runs-to-partial-whileLoop*[**where** $I = \lambda Exception\ \text{-} \Rightarrow (\lambda\text{-}.\ False) \mid Result\ v \Rightarrow I\ v$])

subgoal using $*$ **by** *auto*
subgoal using *P-Result* **by** *auto*
subgoal by *auto*
subgoal by (*rule B*[*THEN runs-to-partial-weaken*]) *auto*
done

lemma *runs-to-partial-whileLoop-exn*:
assumes $*$: *I* (*Result a*) *s*
assumes *P-Result*: $\bigwedge a\ s.\ \neg\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow P\ (Result\ a)\ s$
assumes *P-Exn*: $\bigwedge a\ s.\ I\ (Exn\ a)\ s \Longrightarrow P\ (Exn\ a)\ s$
assumes *B*: $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow$
(*B a*) · *s* ?⦃$\lambda r\ t.\ I\ r\ t$⦄
shows *whileLoop C B a* · *s* ?⦃*P*⦄
apply (*rule runs-to-partial-whileLoop*[**where** *I* = *I*, *OF* $*$])
subgoal using *P-Result* **by** *auto*
subgoal using *P-Exn* **by** (*auto simp add*: *default-option-def Exn-def*)
subgoal using *B* **by** *blast*
done

notation (**output**)
whileLoop (‹(‹*notation*=‹*prefix whileLoop*››*whileLoop* (-)// (-))› [*1000*, *1000*] *1000*)

lemma *whileLoop-mono*: $b \leq b' \Longrightarrow$ *whileLoop c b i* $\leq$ *whileLoop c b' i*
unfolding *whileLoop-def*
by (*simp add*: *gfp-mono le-funD le-funI mono-bind mono-condition*)

lemma *whileLoop-finite-mono*: $b \leq b' \Longrightarrow$ *whileLoop-finite c b i* $\leq$ *whileLoop-finite c b' i*
unfolding *whileLoop-finite-def*
by (*simp add*: *le-funD le-funI lfp-mono mono-bind mono-condition*)

lemma *monotone-whileLoop-le*[*partial-function-mono*]:
($\bigwedge x.$ *monotone R* ($\leq$) ($\lambda f.$ *b f x*)) $\Longrightarrow$ *monotone R* ($\leq$) ($\lambda f.$ *whileLoop c* (*b f*) *i*)
using *whileLoop-mono* **unfolding** *monotone-def*
by (*metis le-fun-def*)

lemma *monotone-whileLoop-ge*[*partial-function-mono*]:
($\bigwedge x.$ *monotone R* ($\geq$) ($\lambda f.$ *b f x*)) $\Longrightarrow$ *monotone R* ($\geq$) ($\lambda f.$ *whileLoop c* (*b f*) *i*)
using *whileLoop-mono* **unfolding** *monotone-def*
by (*metis le-fun-def*)

lemma *monotone-whileLoop-finite-le*[*partial-function-mono*]:
($\bigwedge x.$ *monotone R* ($\leq$) ($\lambda f.$ *b f x*)) $\Longrightarrow$ *monotone R* ($\leq$) ($\lambda f.$ *whileLoop-finite c* (*b f*) *i*)
using *whileLoop-finite-mono* **unfolding** *monotone-def*
by (*metis le-fun-def*)

lemma *monotone-whileLoop-finite-ge*[*partial-function-mono*]:

$(\bigwedge x.\ monotone\ R\ (\geq)\ (\lambda f.\ b\ f\ x)) \Longrightarrow monotone\ R\ (\geq)\ (\lambda f.\ whileLoop\text{-}finite\ c\ (b\ f)\ i)$
using *whileLoop-finite-mono* **unfolding** *monotone-def*
by (*metis le-fun-def*)

lemma *run-whileLoop-le-invariant-cong*:
assumes *I*: $I\ (Result\ i)\ s$
assumes *invariant*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow I\ (Result\ r)\ s \Longrightarrow B\ r \cdot s\ ?\{\!|I|\!\}$
assumes *C*: $\bigwedge r\ s.\ I\ (Result\ r)\ s \Longrightarrow C\ r\ s = C'\ r\ s$
assumes *B*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow I\ (Result\ r)\ s \Longrightarrow run\ (B\ r)\ s = run\ (B'\ r)\ s$
shows $run\ (whileLoop\ C\ B\ i)\ s \leq run\ (whileLoop\ C'\ B'\ i)\ s$
unfolding *le-run-refines-iff*
apply (*rule refines-mono*[*of* $\lambda a\ b.\ a = b \wedge I\ (fst\ a)\ (snd\ a)$])
apply *simp*
apply (*rule refines-whileLoop*)
subgoal using *C* **by** *auto*
subgoal for *x s y t* **using** *invariant*[*of x s*] *B*[*of x t*]
by (*cases run* $(B'\ y)\ t$)
(*auto simp add*: *refines.rep-eq runs-to-partial-alt runs-to.rep-eq split-beta'*)
subgoal by *simp*
subgoal using *I* **by** *simp*
done

lemma *run-whileLoop-invariant-cong*:
assumes *I*: $I\ (Result\ i)\ s$
assumes *invariant*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow I\ (Result\ r)\ s \Longrightarrow B\ r \cdot s\ ?\{\!|I|\!\}$
assumes *C*: $\bigwedge r\ s.\ I\ (Result\ r)\ s \Longrightarrow C\ r\ s = C'\ r\ s$
assumes *B*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow I\ (Result\ r)\ s \Longrightarrow run\ (B\ r)\ s = run\ (B'\ r)\ s$
shows $run\ (whileLoop\ C\ B\ i)\ s = run\ (whileLoop\ C'\ B'\ i)\ s$
proof (*rule antisym*)
from *I invariant C B*
show $run\ (whileLoop\ C\ B\ i)\ s \leq run\ (whileLoop\ C'\ B'\ i)\ s$
by (*rule run-whileLoop-le-invariant-cong*)
next
from *B invariant* **have** *invariant-sym*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow I\ (Result\ r)\ s \Longrightarrow B'\ r \cdot s\ ?\{\!|I|\!\}$
by (*simp add*: *runs-to-partial.rep-eq*)
show $run\ (whileLoop\ C'\ B'\ i)\ s \leq run\ (whileLoop\ C\ B\ i)\ s$
apply (*rule run-whileLoop-le-invariant-cong*
[**where** *I*=*I*, *OF I invariant-sym C*[*symmetric*] *B*[*symmetric*]])
apply (*auto simp add*: *C*)
done
qed

lemma *whileLoop-cong*:
assumes *C*: $\bigwedge r\ s.\ C\ r\ s = C'\ r\ s$
assumes *B*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow run\ (B\ r)\ s = run\ (B'\ r)\ s$
shows $whileLoop\ C\ B = whileLoop\ C'\ B'$
proof (*rule ext*)

```
  fix i
  show whileLoop C B i = whileLoop C' B' i
  proof (rule spec-monad-ext)
    fix s

    show run (whileLoop C B i) s = run (whileLoop C' B' i) s
      by (rule run-whileLoop-invariant-cong[where I = λ- -. True])
        (simp-all add: C B)
  qed
qed

lemma refines-whileLoop':
  assumes C: ⋀a s b t. R (Result a, s) (Result b, t) ⟹ C a s ⟷ C' b t
    and B:
      ⋀a s b t. R (Result a, s) (Result b, t) ⟹ C a s ⟹ C' b t ⟹ refines (B a)
(B' b) s t R
    and I: R (Result I, s) (Result I', s')
    and R:
      ⋀r s r' s'. R (r, s) (r', s') ⟹ rel-exception-or-result (λ- - . True) (λ- - .
True) r r'
  shows refines (whileLoop C B I) (whileLoop C' B' I') s s'
    (λ(r, s) (r', s'). (∀ v. r = Result v ⟶ (∃ v'. r' = Result v' ∧ ¬ C v s ∧ ¬ C'
v' s')) ∧
     R (r, s) (r', s'))
  apply (rule refines-whileLoop-strong)
  subgoal using C by simp
  subgoal using B by simp
 subgoal for v s v' s' using R[of v s v' s'] by (auto elim!: rel-exception-or-result.cases)
  subgoal using I by simp
  done

lemma rel-spec-whileLoop':
  assumes C: ⋀a s b t. R (Result a, s) (Result b, t) ⟹ C a s ⟷ C' b t
   and B: ⋀a s b t. R (Result a, s) (Result b, t) ⟹ C a s ⟹ C' b t ⟹ rel-spec
(B a) (B' b) s t R
    and I: R (Result I, s) (Result I', s')
    and R: ⋀r s r' s'. R (r, s) (r', s') ⟹ rel-exception-or-result (λ- - . True) (λ-
- . True) r r'
  shows rel-spec (whileLoop C B I) (whileLoop C' B' I') s s'
    (λ(r, s) (r', s'). (∀ v. r = Result v ⟶ (∃ v'. r' = Result v' ∧ ¬ C v s ∧ ¬ C'
v' s')) ∧
     R (r, s) (r', s'))
  using C B I R
  unfolding rel-spec-iff-refines
  apply (intro conjI)
  subgoal
    by (intro refines-whileLoop') auto
  subgoal
    apply (intro refines-mono[OF - refines-whileLoop'[where R=R⁻¹⁻¹]])
```

```
    subgoal by simp (smt (verit) Exception-eq-Result rel-exception-or-result.cases)
    subgoal by auto
    subgoal by auto
    subgoal by auto
    subgoal by simp (smt (verit, best) rel-exception-or-result.simps)
    done
  done

lemma rel-spec-whileLoop:
  assumes C: ⋀a s b t. R (Result a, s) (Result b, t) ⟹ C a s ⟷ C' b t
    and B: ⋀a s b t. R (Result a, s) (Result b, t) ⟹ C a s ⟹ C' b t ⟹ rel-spec
(B a) (B' b) s t R
    and I: R (Result I, s) (Result I', s')
    and R: ⋀r s r' s'. R (r, s) (r', s') ⟹ rel-exception-or-result (λ- - . True) (λ-
- . True) r r'
  shows rel-spec (whileLoop C B I) (whileLoop C' B' I') s s' R
  by (rule rel-spec-mono[OF - rel-spec-whileLoop'[OF assms]]) auto

lemma do-whileLoop-combine:
  do { x1 ← body x0; whileLoop P body x1 } =
    do {
      (b, y) ← whileLoop (λ(b, x) s. b ⟶ P x s) (λ(b, x). do { y ← body x; return
(True, y) })
        (False, x0);
      return y
    }
  apply (subst (2) whileLoop-unroll)
  apply (simp add: bind-assoc)
  apply (rule arg-cong[where f=bind -, OF ext])
  apply (subst bind-return[symmetric])
  apply (rule rel-spec-eqD)
  apply (rule rel-spec-bind-bind[where
    Q=rel-prod (rel-exception-or-result (=) (λx (b, y). b = True ∧ x = y)) (=)])
  subgoal for x s
    apply (rule rel-spec-whileLoop)
    subgoal by auto
    subgoal
      apply (simp split: prod.splits) [1]
      apply (subst bind-return[symmetric])
      apply clarify
      apply (rule rel-spec-bind-bind[where Q=(=)])
        apply (simp-all add: rel-spec-refl)
      done
    by (auto simp: rel-exception-or-result.simps split: prod.splits)
     apply (simp-all split: prod.splits)
  done

lemma rel-spec-whileLoop-res:
  assumes C: ⋀a s b t. R (Result a, s) (Result b, t) ⟹ C a s ⟷ C' b t
```

and B: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \Longrightarrow C'\ b\ t \Longrightarrow rel\text{-}spec\ (B\ a)\ (B'\ b)\ s\ t\ R$
and I: $R\ (Result\ I,\ s)\ (Result\ I',\ s')$
shows $rel\text{-}spec\ ((whileLoop\ C\ B\ I)::('a,\ 's)\ res\text{-}monad)\ ((whileLoop\ C'\ B'\ I')::('b,\ 't)\ res\text{-}monad)\ s\ s'\ R$
by $(rule\ rel\text{-}spec\text{-}whileLoop[OF\ C\ B\ I])\ auto$

lemma *rel-spec-monad-whileLoop*:
assumes *init*: $R\ I\ I'$
assumes *cond*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow (rel\text{-}fun\ S\ (=))\ (C\ x)\ (C'\ y)$
assumes *body*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow rel\text{-}spec\text{-}monad\ S\ (rel\text{-}exception\text{-}or\text{-}result\ E\ R)\ (B\ x)\ (B'\ y)$
shows $rel\text{-}spec\text{-}monad\ S\ (rel\text{-}exception\text{-}or\text{-}result\ E\ R)\ (whileLoop\ C\ B\ I)\ (whileLoop\ C'\ B'\ I')$
apply $(clarsimp\ simp\ add: rel\text{-}spec\text{-}monad\text{-}iff\text{-}rel\text{-}spec)$
apply $(rule\ rel\text{-}spec\text{-}whileLoop)$
subgoal for $s\ t\ x\ s'\ y\ t'$ **using** *cond* **by** $(simp\ add: rel\text{-}fun\text{-}def)$
subgoal for $s\ t\ x\ s'\ y\ t'$ **using** $body[of\ x\ y]$ **by** $(simp\ add: rel\text{-}spec\text{-}monad\text{-}iff\text{-}rel\text{-}spec)$
subgoal using *init* **by** *simp*
subgoal by $(auto\ elim!: rel\text{-}exception\text{-}or\text{-}result.cases)$
done

lemma *rel-spec-monad-whileLoop-res'*:
assumes *init*: $R\ I\ I'$
assumes *cond*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow (rel\text{-}fun\ S\ (=))\ (C\ x)\ (C'\ y)$
assumes *body*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow rel\text{-}spec\text{-}monad\ S\ (\lambda Res\ x\ Res\ y.\ R\ x\ y)\ (B\ x)\ (B'\ y)$
shows $rel\text{-}spec\text{-}monad\ S\ (\lambda Res\ x\ Res\ y.\ R\ x\ y)$
$((whileLoop\ C\ B\ I)::('a,\ 's)\ res\text{-}monad)$
$((whileLoop\ C'\ B'\ I')::('b,\ 't)\ res\text{-}monad)$
using *assms*
by $(auto\ intro: rel\text{-}spec\text{-}monad\text{-}whileLoop\ simp\ add: rel\text{-}exception\text{-}or\text{-}result\text{-}Res\text{-}val)$

lemma *rel-spec-monad-whileLoop-res*:
assumes *init*: $R\ I\ I'$
assumes *cond*: $rel\text{-}fun\ R\ (rel\text{-}fun\ S\ (=))\ C\ C'$
assumes *body*: $rel\text{-}fun\ R\ (rel\text{-}spec\text{-}monad\ S\ (\lambda Res\ x\ Res\ y.\ R\ x\ y))\ B\ B'$
shows $rel\text{-}spec\text{-}monad\ S\ (\lambda Res\ x\ Res\ y.\ R\ x\ y)$
$((whileLoop\ C\ B\ I)::('a,\ 's)\ res\text{-}monad)$
$((whileLoop\ C'\ B'\ I')::('b,\ 't)\ res\text{-}monad)$
using *assms*
by $(auto\ intro: rel\text{-}spec\text{-}monad\text{-}whileLoop\text{-}res'\ simp\ add: rel\text{-}fun\text{-}def)$

lemma *runs-to-whileLoop-finite-exn*:
assumes B: $\bigwedge r\ s.\ I\ (Result\ r)\ s \Longrightarrow C\ r\ s \Longrightarrow (B\ r) \bullet s \{\!|\ I\ |\!\}$
assumes Qr: $\bigwedge r\ s.\ I\ (Result\ r)\ s \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ (Result\ r)\ s$
assumes Ql: $\bigwedge e\ s.\ I\ (Exn\ e)\ s \Longrightarrow Q\ (Exn\ e)\ s$
assumes I: $I\ (Result\ r)\ s$
shows $(whileLoop\text{-}finite\ C\ B\ r) \bullet s \{\!|\ Q\ |\!\}$

using *assms* **by** (*intro runs-to-whileLoop-finite*[*of I*]) (*auto simp*: *Exn-def default-option-def*)

lemma *runs-to-whileLoop-finite-res*:
assumes *B*: $\bigwedge r\ s.\ I\ r\ s \Longrightarrow C\ r\ s \Longrightarrow (B\ r) \cdot s\ \{\!|\lambda Res\ r.\ I\ r\ |\!\}$
assumes *Q*: $\bigwedge r\ s.\ I\ r\ s \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ r\ s$
assumes *I*: *I r s*
shows $((whileLoop\text{-}finite\ C\ B\ r)::('a, 's)\ res\text{-}monad) \cdot s\ \{\!|\lambda Res\ r.\ Q\ r\ |\!\}$
using *assms* **by** (*intro runs-to-whileLoop-finite*[*of* $\lambda Res\ r.\ I\ r$]) *simp-all*

lemma *runs-to-whileLoop-eq-whileLoop-finite*:
$run\ (whileLoop\ C\ B\ r)\ s \neq \top \Longrightarrow$
$(whileLoop\ C\ B\ r) \cdot s\ \{\!|\ Q\ |\!\} \longleftrightarrow (whileLoop\text{-}finite\ C\ B\ r) \cdot s\ \{\!|\ Q\ |\!\}$
by (*simp add*: *whileLoop-finite-eq-whileLoop runs-to.rep-eq*)

lemma *whileLoop-finite-cond-fail*:
$\neg\ C\ r\ s \Longrightarrow (run\ (whileLoop\text{-}finite\ C\ B\ r)\ s) = (run\ (return\ r)\ s)$
apply (*subst whileLoop-finite-unfold*)
apply *simp*
done

lemma *runs-to-whileLoop-finite-cond-fail*:
$\neg\ C\ r\ s \Longrightarrow (whileLoop\text{-}finite\ C\ B\ r) \cdot s\ \{\!|\ Q\ |\!\} \longleftrightarrow (return\ r) \cdot s\ \{\!|\ Q\ |\!\}$
apply (*subst whileLoop-finite-unfold*)
apply (*simp add*: *runs-to.rep-eq*)
done

lemma *runs-to-whileLoop-cond-fail*:
$\neg\ C\ r\ s \Longrightarrow (whileLoop\ C\ B\ r) \cdot s\ \{\!|\ Q\ |\!\} \longleftrightarrow (return\ r) \cdot s\ \{\!|\ Q\ |\!\}$
apply (*subst whileLoop-unroll*)
apply (*simp add*: *runs-to.rep-eq*)
done

lemma *whileLoop-finite-unfold'*:
$(whileLoop\text{-}finite\ C\ B\ r) =$
$((condition\ (C\ r)\ (B\ r)\ (return\ r)) >>= (whileLoop\text{-}finite\ C\ B))$
apply (*rule spec-monad-ext*)
apply (*subst* (*1*) *whileLoop-finite-unfold*)
subgoal for *s*
apply (*cases C r s*)
subgoal by (*simp add*: *run-bind*)
subgoal by (*simp add*: *whileLoop-finite-cond-fail run-bind*)
done
done

lemma *runs-to-whileLoop-unroll*:
assumes $\neg\ C\ r\ s \Longrightarrow P\ r\ s$
assumes [*runs-to-vcg*]: $C\ r\ s \Longrightarrow (B\ r) \cdot s\ \{\!|\ \lambda Res\ r\ t.\ ((whileLoop\ C\ B\ r) \cdot t\ \{\!|\lambda Res\ r.\ P\ r\ |\!\})\ |\!\}$

shows (*whileLoop C B r*) · *s* {|λ*Res r*. *P r* |}
using *assms*(*1*)
by (*subst whileLoop-unroll*) *runs-to-vcg*

lemma *runs-to-partial-whileLoop-unroll*:
assumes ¬ *C r s* ⟹ *P r s*
assumes *C r s* ⟹ (*B r*) · *s ?*{| λ*Res r t*. ((*whileLoop C B r*) · *t ?*{|λ*Res r*. *P r* |}) |}
shows (*whileLoop C B r*) · *s ?*{|λ*Res r*. *P r* |}
using *assms*
by (*subst whileLoop-unroll*) *runs-to-vcg*

lemma *runs-to-whileLoop-unroll-exn*:
assumes ¬ *C r s* ⟹ *P* (*Result r*) *s*
assumes [*runs-to-vcg*]: *C r s* ⟹ (*B r*) · *s* {| λ*r t*.
(∀ *a*. *r* = *Result a* ⟶ ((*whileLoop C B a*) · *t* {| *P* |})) ∧
(∀ *e*. *r* = *Exn e* ⟶ *P* (*Exn e*) *t*) |}
shows (*whileLoop C B r*) · *s* {|λ*r s*′. *P r s*′ |}
using *assms*(*1*)
by (*subst whileLoop-unroll*) (*runs-to-vcg*, *auto simp*: *Exn-def default-option-def*)

lemma *runs-to-partial-whileLoop-unroll-exn*:
assumes ¬ *C r s* ⟹ *P* (*Result r*) *s*
assumes *C r s* ⟹ (*B r*) · *s ?*{| λ*r t*.
(∀ *a*. *r* = *Result a* ⟶ ((*whileLoop C B a*) · *t ?*{| *P* |})) ∧
(∀ *e*. *r* = *Exn e* ⟶ *P* (*Exn e*) *t*) |}
shows (*whileLoop C B r*) · *s ?*{|λ*r s*′. *P r s*′ |}
using *assms*
by (*subst whileLoop-unroll*) *runs-to-vcg*

lemma *refines-whileLoop-exn*:
assumes *C*: ⋀*a s b t*. *R* (*Result a*, *s*) (*Result b*, *t*) ⟹ *C a s* ⟷ *C*′ *b t*
and *B*: ⋀*a s b t*. *R* (*Result a*, *s*) (*Result b*, *t*) ⟹ *C a s* ⟹ *C*′ *b t* ⟹ *refines* (*B a*) (*B*′ *b*) *s t R*
and *I*: *R* (*Result I*, *s*) (*Result I*′, *s*′)
and *R1*: ⋀*a s b t*. *R* (*Exn a*, *s*) (*Result b*, *t*) ⟹ *False*
and *R2*: ⋀*a s b t*. *R* (*Result a*, *s*) (*Exn b*, *t*) ⟹ *False*
shows *refines* (*whileLoop C B I*) (*whileLoop C*′ *B*′ *I*′) *s s*′ *R*
proof −
have *ref*: *refines* (*whileLoop C B I*) (*whileLoop C*′ *B*′ *I*′) *s s*′
(λ(*r*, *s*) (*r*′, *s*′). (∀ *v*. *r* = *Result v* ⟶ (∃ *v*′. *r*′ = *Result v*′ ∧ ¬ *C v s* ∧ ¬ *C*′ *v*′ *s*′)) ∧
R (*r*, *s*) (*r*′, *s*′))

apply (*rule refines-whileLoop*′ [*OF C*])
subgoal by *assumption*
subgoal by (*rule B*)

subgoal by (*rule I*)
subgoal using *R1 R2*
by (*auto simp add*: *rel-exception-or-result.simps Exn-def default-option-def*)
(*metis default-option-def exception-or-result-cases not-None-eq*)+
done
show *?thesis*
apply (*rule refines-mono* [*OF - ref*])
apply (*auto*)
done
qed

lemma *refines-whileLoop''*:
assumes *C*: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \longleftrightarrow C'\ b\ t$
and *B*: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \Longrightarrow C'\ b\ t \Longrightarrow refines\ (B\ a)\ (B'\ b)\ s\ t\ R$
and *I*: $R\ (Result\ I,\ s)\ (Result\ I',\ s')$
and *R*: $\bigwedge r\ s\ r'\ s'.\ R\ (r,\ s)\ (r',\ s') \Longrightarrow rel\text{-}exception\text{-}or\text{-}result\ (\lambda\text{- - }.\ True)\ (\lambda\text{- - }.\ True)\ r\ r'$
shows *refines* (*whileLoop C B I*) (*whileLoop C' B' I'*) *s s' R*
proof −
have *refines* (*whileLoop C B I*) (*whileLoop C' B' I'*) *s s'*
$(\lambda(r,\ s)\ (r',\ s').\ (\forall v.\ r = Result\ v \longrightarrow (\exists v'.\ r' = Result\ v' \wedge \neg\ C\ v\ s \wedge \neg\ C'\ v'\ s')) \wedge$
$R\ (r,\ s)\ (r',\ s'))$
by (*rule refines-whileLoop'* [*OF C B I R*])
then show *?thesis*
apply (*rule refines-weaken*)
apply *auto*
done
qed

lemma *rel-spec-monad-whileLoop-exn*:
assumes *init*: *R I I'*
assumes *cond*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow (rel\text{-}fun\ S\ (=))\ (C\ x)\ (C'\ y)$
assumes *body*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow rel\text{-}spec\text{-}monad\ S\ (rel\text{-}xval\ E\ R)\ (B\ x)\ (B'\ y)$
shows *rel-spec-monad S* (*rel-xval E R*) (*whileLoop C B I*) (*whileLoop C' B' I'*)
unfolding *rel-xval-rel-exception-or-result-conv*
by (*rule rel-spec-monad-whileLoop* [*OF init cond body* [*simplified rel-xval-rel-exception-or-result-conv*]])

lemma *refines-whileLoop-guard-right*:
assumes $\bigwedge x\ s\ x'\ s'.\ R\ (Result\ x,\ s)\ (Result\ x',\ s') \Longrightarrow G'\ x'\ s' \Longrightarrow C\ x\ s = C'\ x'\ s'$
assumes $\bigwedge x\ s\ x'\ s'.\ R\ (Result\ x,\ s)\ (Result\ x',\ s') \Longrightarrow C\ x\ s \Longrightarrow C'\ x'\ s' \Longrightarrow G'\ x'\ s' \Longrightarrow refines\ (B\ x)\ (B'\ x')\ s\ s'\ R$
assumes $\bigwedge v\ s\ v'\ s'.\ R\ (v,\ s)\ (v',\ s') \Longrightarrow (\exists x.\ v = Result\ x) = (\exists x'.\ v' = Result\ x')$
assumes $R\ (Result\ x,\ s)\ (Result\ x',\ s')$
assumes $G\ s' = G'\ x'\ s'$
shows *refines* (*whileLoop C B x*) (*guard G* $\gg$= (λ-. *whileLoop C'* ($\lambda r.$ *do* {*res*

$<-$ B' r; *guard* (G' *res*); *return res*}) x')) s s' R
apply (*rule refines-bind-guard-right*)
apply (*rule refines-mono* [**where** R=$\lambda(r,\ s)$ $(q,\ t)$. R $(r,\ s)$ $(q,\ t)$ $\wedge$
(*case q of Exception e* $\Rightarrow$ *True* | *Result v* $\Rightarrow$ G' v t)])
subgoal by *simp*
subgoal
apply (*rule refines-whileLoop*)
subgoal using *assms*(*1*) **by** *auto*
subgoal apply (*rule refines-bind-guard-right-end'*)
using *assms*(*2*)
by *auto*
subgoal using *assms*(*3*) **by** (*auto split*: *exception-or-result-splits*)
subgoal using *assms*(*4*,*5*) **by** *auto*
done
done

16.9.31 *map-value*

lemma *map-value-lift-state*: *map-value f* (*lift-state R g*) = *lift-state R* (*map-value f g*)
apply *transfer*
apply (*simp add*: *map-post-state-eq-lift-post-state lift-post-state-comp lift-post-state-Sup image-image OO-def prod-eq-iff rel-prod.simps*)
apply (*simp add*: *ac-simps*)
done

lemma *run-map-value*[*run-spec-monad*]: *run* (*map-value f g*) *s* =
map-post-state ($\lambda(v,\ s)$. (*f v*, *s*)) (*run g s*)
by *transfer* (*auto simp add*: *apfst-def map-prod-def map-post-state-def split*: *post-state.splits prod.splits*)

lemma *always-progress-map-value*[*always-progress-intros*]:
always-progress g $\Longrightarrow$ *always-progress* (*map-value f g*)
by (*simp add*: *always-progress-def run-map-value*)

lemma *runs-to-map-value-iff*[*runs-to-iff*]: *map-value f g* $\cdot$ *s* $\{\!|$ *Q* $|\!\}$ $\longleftrightarrow$ *g* $\cdot$ *s* $\{\!|\lambda r$ *t*. *Q* (*f r*) *t*$|\!\}$
by (*auto simp add*: *runs-to.rep-eq run-map-value split-beta'*)

lemma *runs-to-partial-map-value-iff*[*runs-to-iff*]:
map-value f g $\cdot$ *s* ?$\{\!|$ *Q* $|\!\}$ $\longleftrightarrow$ *g* $\cdot$ *s* ?$\{\!|\lambda r$ *t*. *Q* (*f r*) *t*$|\!\}$
by (*auto simp add*: *runs-to-partial.rep-eq run-map-value split-beta'*)

lemma *runs-to-map-value*[*runs-to-vcg*]: *g* $\cdot$ *s* $\{\!|\lambda r$ *t*. *Q* (*f r*) *t*$|\!\}$ $\Longrightarrow$ *map-value f g* $\cdot$ *s* $\{\!|$ *Q* $|\!\}$
by (*simp add*: *runs-to-map-value-iff*)

lemma *runs-to-partial-map-value*[*runs-to-vcg*]:
g $\cdot$ *s* ?$\{\!|\lambda r$ *t*. *Q* (*f r*) *t*$|\!\}$ $\Longrightarrow$ *map-value f g* $\cdot$ *s* ?$\{\!|$ *Q* $|\!\}$

by (*auto simp add*: *runs-to-partial.rep-eq run-map-value split-beta'*)

lemma *mono-map-value*: $g \leq g' \Longrightarrow$ *map-value f g* $\leq$ *map-value f g'*
by *transfer* (*simp add*: *le-fun-def mono-map-post-state*)

lemma *monotone-map-value-le*[*partial-function-mono*]:
monotone R ($\leq$) ($\lambda f'$. *g f'*) $\Longrightarrow$ *monotone R* ($\leq$) ($\lambda f'$. *map-value f* (*g f'*))
by (*auto simp*: *monotone-def mono-map-value*)

lemma *monotone-map-value-ge*[*partial-function-mono*]:
monotone R ($\geq$) ($\lambda f'$. *g f'*) $\Longrightarrow$ *monotone R* ($\geq$) ($\lambda f'$. *map-value f* (*g f'*))
by (*auto simp*: *monotone-def mono-map-value*)

lemma *map-value-fail*[*simp*]: *map-value f fail* = *fail*
by *transfer simp*

lemma *map-value-map-exn-gets*[*simp*]: *map-value* (*map-exn emb*) (*gets x*) = *gets x*
by (*simp add*: *spec-monad-ext-iff run-map-value*)

lemma *refines-map-value-right-iff*:
refines f (*map-value m g*) *s t R* $\longleftrightarrow$ *refines f g s t* ($\lambda(x, s)$ (y, t). *R* (x, *s*) (*m y*, *t*))
by *transfer* (*simp add*: *sim-post-state-map-post-state2 split-beta' apfst-def map-prod-def*)

lemma *refines-map-value-left-iff*:
refines (*map-value m f*) *g s t R* $\longleftrightarrow$ *refines f g s t* ($\lambda(x, s)$ (y, t). *R* (*m x*, *s*) (y, *t*))
by *transfer* (*simp add*: *sim-post-state-map-post-state1 split-beta' apfst-def map-prod-def*)

lemma *rel-spec-map-value-right-iff*:
rel-spec f (*map-value m g*) *s t R* $\longleftrightarrow$ *rel-spec f g s t* ($\lambda(x, s)$ (y, t). *R* (x, *s*) (*m y*, *t*))
by (*simp add*: *rel-spec-iff-refines refines-map-value-right-iff refines-map-value-left-iff conversep.simps*[*abs-def*] *split-beta'*)

lemma *rel-spec-map-value-left-iff*:
rel-spec (*map-value m f*) *g s t R* $\longleftrightarrow$ *rel-spec f g s t* ($\lambda(x, s)$ (y, t). *R* (*m x*, *s*) (y, *t*))
by (*simp add*: *rel-spec-iff-refines refines-map-value-right-iff refines-map-value-left-iff conversep.simps*[*abs-def*] *split-beta'*)

lemma *refines-map-value-right-same*:
refines f (*map-value m f*) *s s* ($\lambda(x, s)$ (y, t). $y = m\ x \wedge s = t$)
by (*simp add*: *refines-map-value-right-iff refines-refl*)

lemma *refines-map-value*:
assumes *refines f f' s t Q*
assumes $\bigwedge r\ s'\ w\ t'$. *Q* (r, s') (w, t') $\Longrightarrow$ *R* (*g r*, s') (*g' w*, t')

shows *refines* (*map-value g f*) (*map-value g′ f′*) *s t R*
apply (*simp add*: *refines-map-value-left-iff refines-map-value-right-iff*)
apply (*rule refines-weaken* [*OF assms*(*1*)])
using *assms*(*2*) **by** *auto*

lemma *map-value-id*[*simp*]: *map-value* (λ*x*. *x*) = (λ*x*. *x*)
apply (*simp add*: *fun-eq-iff*, *intro allI iffI*)
apply (*rule spec-monad-eqI*)
apply (*rule runs-to-iff*)
done

16.9.32 *liftE*

lemma *run-liftE*[*run-spec-monad*]:
run (*liftE f*) *s* =
map-post-state (λ(*v*, *s*). (*map-exception-or-result* (λ*x*. *undefined*) *id v*, *s*)) (*run f s*)
by (*simp add*: *run-map-value liftE-def*)

lemma *always-progress-liftE*[*always-progress-intros*]:
always-progress f ⟹ *always-progress* (*liftE f*)
by (*simp add*: *liftE-def always-progress-intros*)

lemma *runs-to-liftE-iff*:
liftE f · *s* ⦃ *Q* ⦄ ⟷ *f* · *s* ⦃λ*r t*. *Q* (*map-exception-or-result* (λ*x*. *undefined*) *id r*) *t*⦄
by (*simp add*: *liftE-def runs-to-map-value-iff*)

lemma *runs-to-liftE-iff-Res*[*runs-to-iff*]:
liftE f · *s* ⦃ *Q* ⦄ ⟷ *f* · *s* ⦃ λ*Res r*. *Q* (*Result r*)⦄
by (*auto intro*!: *runs-to-cong-pred-only simp*: *runs-to-liftE-iff*)

lemma *runs-to-liftE′*:
f · *s* ⦃λ*r t*. *Q* (*map-exception-or-result* (λ*x*. *undefined*) *id r*) *t*⦄ ⟹ *liftE f* · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to-liftE-iff*)

lemma *runs-to-liftE*[*runs-to-vcg*]: *f* · *s* ⦃ λ*Res r*. *Q* (*Result r*)⦄ ⟹ *liftE f* · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to-liftE-iff-Res*)

lemma *refines-liftE-left-iff*:
refines (*liftE f*) *g s t R* ⟷
refines f g s t (λ(*x*, *s′*) (*y*, *t′*). ∀ *v*. *x* = *Result v* ⟶ *R* (*Result v*, *s′*) (*y*, *t′*))
by (*auto simp add*: *liftE-def refines-map-value-left-iff intro*!: *refines-cong-cases del*: *iffI*)

lemma *refines-liftE-right-iff*:
refines f (*liftE g*) *s t R* ⟷

refines f g s t (λ(*x*, *s'*) (*y*, *t'*). ∀ *v*. *y* = *Result v* ⟶ *R* (*x*, *s'*) (*Result v*, *t'*))
by (*auto simp add*: *liftE-def refines-map-value-right-iff intro*!: *refines-cong-cases del*: *iffI*)

lemma *rel-spec-liftE*:
rel-spec (*liftE f*) *g s t R* ⟷
rel-spec f g s t (λ(*x*, *s'*) (*y*, *t'*). ∀ *v*. *x* = *Result v* ⟶ *R* (*Result v*, *s'*) (*y*, *t'*))
by (*simp add*: *rel-spec-iff-refines refines-liftE-right-iff refines-liftE-left-iff conversep.simps*[*abs-def*] *split-beta'*)

lemma *rel-spec-monad-bind-liftE*:
rel-spec-monad R (λ*Res v1 Res v2*. *P v1 v2*) *m n* ⟹ *rel-fun P* (*rel-spec-monad R Q*) *f g* ⟹
rel-spec-monad R Q ((*liftE m*) >>= *f*) ((*liftE n*) >>= *g*)
apply (*auto intro*!: *rel-bind-post-state' rel-post-state-refl simp*: *rel-spec-monad-def run-bind run-liftE bind-post-state-map-post-state rel-fun-def*)[1]
subgoal premises *prems* **for** *s t*
by (*rule rel-post-state-weaken*[*OF prems*(1)[*rule-format*, *OF prems*(3)]])
(*auto simp add*: *rel-prod.simps prems*(2))
done

lemma *rel-spec-monad-bind-liftE'*:
(⋀*x*. *rel-spec-monad* (=) *Q* (*f x*) (*g x*)) ⟹ *rel-spec-monad* (=) *Q* (*liftE m* >>= *f*) (*liftE m* >>= *g*)
by (*auto simp add*: *rel-spec-monad-def run-bind run-liftE bind-post-state-map-post-state intro*!: *rel-bind-post-state' rel-post-state-refl*)

lemma *runs-to-partial-liftE'*:
f · *s* ?⦃λ*r t*. *Q* (*map-exception-or-result* (λ*x*. *undefined*) *id r*) *t*⦄ ⟹ *liftE f* · *s* ?⦃ *Q* ⦄
by (*simp add*: *runs-to-partial.rep-eq run-liftE split-beta'*)

lemma *runs-to-partial-liftE*[*runs-to-vcg*]:
assumes [*runs-to-vcg*]: *f* · *s* ?⦃ λ*Res r*. *Q* (*Result r*)⦄ **shows** *liftE f* · *s* ?⦃ *Q* ⦄
supply *runs-to-partial-liftE'*[*runs-to-vcg*] **by** *runs-to-vcg*

lemma *mono-lifE*: *f* ≤ *f'* ⟹ *liftE f* ≤ *liftE f'*
unfolding *liftE-def* **by** (*rule mono-map-value*)

lemma *monotone-liftE-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*) ⟹ *monotone R* (≤) (λ*f'*. *liftE* (*f f'*))
unfolding *liftE-def* **by** (*rule monotone-map-value-le*)

lemma *monotone-liftE-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ *monotone R* (≥) (λ*f'*. *liftE* (*f f'*))
unfolding *liftE-def* **by** (*rule monotone-map-value-ge*)

lemma *bind-handle-liftE*: *bind-handle* (*liftE f*) *g h* = *bind f g*
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *liftE-top*[*simp*]: *liftE* ⊤ = ⊤
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-bot*[*simp*]: *liftE bot* = *bot*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-fail*[*simp*]: *liftE fail* = *fail*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-return*[*simp*]: *liftE* (*return x*) = *return x*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-yield-Exception*[*simp*]:
liftE (*yield* (*Exception x*)) = *skip*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-throw-exception-or-result*[*simp*]:
liftE (*throw-exception-or-result x*) = *return undefined*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE map-exception-or-result-def*)

lemma *liftE-get-state*[*simp*]: *liftE* (*get-state*) = *get-state*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-set-state*[*simp*]: *liftE* (*set-state s*) = *set-state s*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-select*[*simp*]: *liftE* (*select S*) = *select S*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-unknown*[*simp*]: *liftE* (*unknown*) = *unknown*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-lift-state*: *liftE* (*lift-state R f*) = *lift-state R* (*liftE f*)
unfolding *liftE-def* **by** (*rule map-value-lift-state*)

lemma *liftE-exec-concrete*: *liftE* (*exec-concrete st f*) = *exec-concrete st* (*liftE f*)
unfolding *exec-concrete-def* **by** (*rule liftE-lift-state*)

lemma *liftE-exec-abstract*: *liftE* (*exec-abstract st f*) = *exec-abstract st* (*liftE f*)
unfolding *exec-abstract-def* **by** (*rule liftE-lift-state*)

lemma *liftE-assert*[*simp*]: *liftE* (*assert P*) = *assert P*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE run-assert*)

lemma *liftE-assume*[*simp*]: *liftE* (*assume P*) = *assume P*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE run-assume*)

lemma *liftE-gets*[*simp*]: *liftE* (*gets f*) = *gets f*

by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-guard*[*simp*]: *liftE* (*guard P*) = *guard P*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE run-guard*)

lemma *liftE-assert-opt*[*simp*]: *liftE* (*assert-opt v*) = *assert-opt v*
by (*rule spec-monad-ext*) (*auto simp add*: *run-liftE split*: *option.splits*)

lemma *liftE-gets-the*[*simp*]: *liftE* (*gets-the f*) = *gets-the f*
by (*rule spec-monad-ext*) (*auto simp add*: *run-liftE split*: *option.splits*)

lemma *liftE-modify*[*simp*]: *liftE* (*modify f*) = *modify f*
by (*rule spec-monad-ext*) (*simp add*: *run-liftE*)

lemma *liftE-bind*: *liftE x* >>= (λ*a*. *liftE* (*y a*)) = *liftE* (*x* >>= *y*)
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *bindE-liftE-skip*: *liftE* (*f* ≫= (λ*y*. *skip*)) ≫= *g* = *liftE f* ≫= (λ-. *g* ())
by (*auto simp add*: *spec-monad-eq-iff runs-to-iff intro*!: *arg-cong*[**where** *f*=*runs-to f* -])

lemma *liftE-state-select*[*simp*]: *liftE* (*state-select f*) = *state-select f*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *liftE-assume-result-and-state*[*simp*]:
liftE (*assume-result-and-state f*) = *assume-result-and-state f*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *map-value-map-exn-liftE* [*simp*]:
map-value (*map-exn emb*) (*liftE f*) = *liftE f*
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *liftE-condition*: *liftE* (*condition c f g*) = *condition c* (*liftE f*) (*liftE g*)
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *liftE-whileLoop*: *liftE* (*whileLoop C B I*) = *whileLoop C* (λ*r*. *liftE* (*B r*)) *I*
apply (*rule rel-spec-eqD*)
apply (*subst rel-spec-liftE*)
apply (*rule rel-spec-whileLoop*)
subgoal by *simp*
apply (*subst rel-spec-symm*)
apply (*subst rel-spec-liftE*)
apply (*simp add*: *rel-spec-refl*)
apply *auto*
done

16.9.33 *try*

lemma *run-try*[*run-spec-monad*]:
run (*try f*) *s* = *map-post-state* (λ(*v*, *s*). (*unnest-exn v*, *s*)) (*run f s*)
by (*simp add*: *try-def run-map-value*)

lemma *always-progress-try*[*always-progress-intros*]: *always-progress f* ⟹ *always-progress* (*try f*)
by (*simp add*: *try-def always-progress-intros*)

lemma *runs-to-try*[*runs-to-vcg*]: *f* · *s* ⦃λ*r t*. *Q* (*unnest-exn r*) *t*⦄ ⟹ *try f* · *s* ⦃ *Q* ⦄
by (*simp add*: *try-def runs-to-map-value*)

lemma *runs-to-try-iff*[*runs-to-iff*]: *try f* · *s* ⦃ *Q* ⦄ ⟷ *f* · *s* ⦃λ*r t*. *Q* (*unnest-exn r*) *t*⦄
by (*auto simp add*: *try-def runs-to-iff*)

lemma *runs-to-partial-try*[*runs-to-vcg*]: *f* · *s* ?⦃λ*r t*. *Q* (*unnest-exn r*) *t*⦄ ⟹ *try f* · *s* ?⦃ *Q* ⦄
by (*simp add*: *try-def runs-to-partial-map-value*)

lemma *mono-try*: *f* ≤ *f'* ⟹ *try f* ≤ *try f'*
unfolding *try-def*
by (*rule mono-map-value*)

lemma *monotone-try-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*) ⟹ *monotone R* (≤) (λ*f'*. *try* (*f f'*))
unfolding *try-def* **by** (*rule monotone-map-value-le*)

lemma *monotone-try-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ *monotone R* (≥) (λ*f'*. *try* (*f f'*))
unfolding *try-def* **by** (*rule monotone-map-value-ge*)

lemma *refines-try-right-same*:
refines f (*try f*) *s s* (λ(*x*, *s*) (*y*, *t*). *y* = *unnest-exn x* ∧ *s* = *t*)
by (*auto simp add*: *try-def refines-map-value-right-same*)

16.9.34 *finally*

lemma *run-finally*[*run-spec-monad*]:
run (*finally f*) *s* = *map-post-state* (λ(*v*, *s*). (*unite v*, *s*)) (*run f s*)
by (*simp add*: *finally-def run-map-value*)

lemma *always-progress-finally*[*always-progress-intros*]:
always-progress f ⟹ *always-progress* (*finally f*)
by (*simp add*: *finally-def always-progress-intros*)

lemma *runs-to-finally-iff*[*runs-to-iff*]:
finally f · *s* ⦃ *Q* ⦄ ⟷

$f \cdot s \ \{\!|\lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q\ (Result\ v)\ t)|\!\}$
by (*auto simp*: *finally-def runs-to-map-value-iff unite-def*
intro!: *arg-cong*[**where** *f*=*runs-to - -*] *split*: *xval-splits*)

lemma *runs-to-finally'*: $f \cdot s \ \{\!|\lambda r\ t.\ Q\ (unite\ r)\ t|\!\} \Longrightarrow finally\ f \cdot s \ \{\!|\ Q\ |\!\}$
by (*simp add*: *finally-def runs-to-map-value*)

lemma *runs-to-finally*[*runs-to-vcg*]:
$f \cdot s \ \{\!|\lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q\ (Result\ v)\ t)|\!\} \Longrightarrow$
$finally\ f \cdot s \ \{\!|\ Q\ |\!\}$
by (*simp add*: *runs-to-finally-iff*)

lemma *runs-to-partial-finally'*: $f \cdot s \ ?\{\!|\lambda r\ t.\ Q\ (unite\ r)\ t|\!\} \Longrightarrow finally\ f \cdot s \ ?\{\!|\ Q\ |\!\}$
by (*simp add*: *finally-def runs-to-partial-map-value*)

lemma *runs-to-partial-finally-iff*:
$finally\ f \cdot s \ ?\{\!|\ Q\ |\!\} \longleftrightarrow$
$f \cdot s \ ?\{\!|\lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q\ (Result\ v)\ t)|\!\}$
by (*auto simp*: *finally-def runs-to-partial-map-value-iff unite-def*
intro!: *arg-cong*[**where** *f*=*runs-to-partial - -*] *split*: *xval-splits*)

lemma *runs-to-partial-finally*[*runs-to-vcg*]:
$f \cdot s \ ?\{\!|\lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q\ (Result\ v)\ t)|\!\} \Longrightarrow$
$finally\ f \cdot s \ ?\{\!|\ Q\ |\!\}$
by (*simp add*: *runs-to-partial-finally-iff*)

lemma *mono-finally*: $f \leq f' \Longrightarrow finally\ f \leq finally\ f'$
unfolding *finally-def*
by (*rule mono-map-value*)

lemma *monotone-finally-le*[*partial-function-mono*]:
$monotone\ R\ (\leq)\ (\lambda f'.\ f\ f') \Longrightarrow monotone\ R\ (\leq)\ (\lambda f'.\ finally\ (f\ f'))$
unfolding *finally-def* **by** (*rule monotone-map-value-le*)

lemma *monotone-finally-ge*[*partial-function-mono*]:
$monotone\ R\ (\geq)\ (\lambda f'.\ f\ f') \Longrightarrow monotone\ R\ (\geq)\ (\lambda f'.\ finally\ (f\ f'))$
unfolding *finally-def* **by** (*rule monotone-map-value-ge*)

16.9.35 (<*catch*>)

lemma *run-catch*[*run-spec-monad*]:
$run\ (f <catch> h)\ s =$
$bind\text{-}post\text{-}state\ (run\ f\ s)$
$(\lambda(r, t).\ case\ r\ of\ Exn\ e \Rightarrow run\ (h\ e)\ t \mid Result\ v \Rightarrow run\ (return\ v)\ t)$

apply (*simp add*: *catch-def run-bind-handle*)
apply (*rule arg-cong*[**where** *f*=*bind-post-state* -, *OF ext*])
apply (*auto simp add*: *Exn-def default-option-def split*: *exception-or-result-splits xval-splits*)
done

lemma *always-progress-catch*[*always-progress-intros*]:
always-progress f $\Longrightarrow$ ($\bigwedge$*e*. *always-progress* (*h e*)) $\Longrightarrow$ *always-progress* (*f* <*catch*> *h*)
unfolding *catch-def*
by (*auto intro*: *always-progress-intros*)

lemma *runs-to-catch-iff*[*runs-to-iff*]:
(*f* <*catch*> *h*) · *s* {‖ *Q* ‖} $\longleftrightarrow$
f · *s* {‖ λ*r t*. ($\forall$ *v*. *r* = *Result v* $\longrightarrow$ *Q* (*Result v*) *t*) $\wedge$ ($\forall$ *e*. *r* = *Exn e* $\longrightarrow$ *h e* · *t* {‖ *Q* ‖)‖}
by (*simp add*: *runs-to.rep-eq run-catch split-beta*′ *ac-simps split*: *xval-splits prod.splits*)

lemma *runs-to-catch*[*runs-to-vcg*]:
f · *s* {‖ λ*r t*. ($\forall$ *v*. *r* = *Result v* $\longrightarrow$ *Q* (*Result v*) *t*) $\wedge$
($\forall$ *e*. *r* = *Exn e* $\longrightarrow$ *h e* · *t* {‖ *Q* ‖)‖} $\Longrightarrow$
(*f* <*catch*> *h*) · *s* {‖ *Q* ‖}
by (*simp add*: *runs-to-catch-iff*)

lemma *runs-to-partial-catch*[*runs-to-vcg*]:
f · *s* ?{‖ λ*r t*. ($\forall$ *v*. *r* = *Result v* $\longrightarrow$ *Q* (*Result v*) *t*) $\wedge$
($\forall$ *e*. *r* = *Exn e* $\longrightarrow$ *h e* · *t* ?{‖ *Q* ‖)‖} $\Longrightarrow$
(*f* <*catch*> *h*) · *s* ?{‖ *Q* ‖}
unfolding *runs-to-partial.rep-eq run-catch*
by (*rule holds-partial-bind-post-state*)
(*simp add*: *split-beta*′ *ac-simps split*: *xval-splits prod.splits*)

lemma *mono-catch*: *f* $\leq$ *f*′ $\Longrightarrow$ *h* $\leq$ *h*′ $\Longrightarrow$ *catch f h* $\leq$ *catch f*′ *h*′
unfolding *catch-def*
apply (*rule mono-bind-handle*)
by (*simp-all add*: *le-fun-def*)

lemma *monotone-catch-le*[*partial-function-mono*]:
monotone R ($\leq$) (λ*f*′. *f f*′) $\Longrightarrow$ ($\bigwedge$*e*. *monotone R* ($\leq$) (λ*f*′. *h f*′ *e*))
$\Longrightarrow$ *monotone R* ($\leq$) (λ*f*′. *catch* (*f f*′) (*h f*′))
unfolding *catch-def*
apply (*rule monotone-bind-handle-le*)
by *auto*

lemma *monotone-catch-ge*[*partial-function-mono*]:
monotone R ($\geq$) (λ*f*′. *f f*′) $\Longrightarrow$ ($\bigwedge$*e*. *monotone R* ($\geq$) (λ*f*′. *h f*′ *e*))
$\Longrightarrow$ *monotone R* ($\geq$) (λ*f*′. *catch* (*f f*′) (*h f*′))
unfolding *catch-def*
apply (*rule monotone-bind-handle-ge*)

by *auto*

lemma *catch-liftE*: *catch* (*liftE g*) *h* = *g*
by (*simp add*: *catch-def bind-handle-liftE*)

lemma *refines-catch*:
assumes *f*: *refines f f′ s s′ Q*
assumes *ll*: ⋀*e e′ t t′*. *Q* (*Exn e*, *t*) (*Exn e′*, *t′*) ⟹
refines (*h e*) (*h′ e′*) *t t′ R*
assumes *lr*: ⋀*e v′ t t′*. *Q* (*Exn e*, *t*) (*Result v′*, *t′*) ⟹
refines (*h e*) (*return v′*) *t t′ R*
assumes *rl*: ⋀*v e′ t t′*. *Q* (*Result v*, *t*) (*Exn e′*, *t′*) ⟹
refines (*return v*) (*h′ e′*) *t t′ R*
assumes *rr*: ⋀*v v′ t t′*. *Q* (*Result v*, *t*) (*Result v′*, *t′*) ⟹
R (*Result v*, *t*) (*Result v′*, *t′*)
shows *refines* (*catch f h*) (*catch f′ h′*) *s s′ R*
unfolding *catch-def*
apply (*rule refines-bind-handle-bind-handle-exn*[*OF f*])
subgoal using *ll* **by** *auto*
subgoal using *lr* **by** *auto*
subgoal using *rl* **by** *auto*
subgoal using *rr* **by** (*auto simp add*: *refines-yield*)
done

lemma *rel-spec-catch*:
assumes *f*: *rel-spec f f′ s s′ Q*
assumes *ll*: ⋀*e e′ t t′*. *Q* (*Exn e*, *t*) (*Exn e′*, *t′*) ⟹
rel-spec (*h e*) (*h′ e′*) *t t′ R*
assumes *lr*: ⋀*e v′ t t′*. *Q* (*Exn e*, *t*) (*Result v′*, *t′*) ⟹
rel-spec (*h e*) (*return v′*) *t t′ R*
assumes *rl*: ⋀*v e′ t t′*. *Q* (*Result v*, *t*) (*Exn e′*, *t′*) ⟹
rel-spec (*return v*) (*h′ e′*) *t t′ R*
assumes *rr*: ⋀*v v′ t t′*. *Q* (*Result v*, *t*) (*Result v′*, *t′*) ⟹
R (*Result v*, *t*) (*Result v′*, *t′*)
shows *rel-spec* (*catch f h*) (*catch f′ h′*) *s s′ R*
using *assms* **by** (*auto simp*: *rel-spec-iff-refines intro*!: *refines-catch*)

16.9.36 *check*

lemma *run-check*[*run-spec-monad*]: *run* (*check e p*) *s* =
(*if* (∃ *x*. *p s x*) *then Success* ((λ*x*. (*x*, *s*)) ‘ (*Result* ‘ {*x*. *p s x*})) *else*
pure-post-state (*Exn e*, *s*))
by (*auto simp add*: *check-def run-bind*)

lemma *always-progress-check*[*always-progress-intros*, *simp*]: *always-progress* (*check e p*)
by (*auto simp add*: *always-progress-def run-check*)

lemma *runs-to-check-iff*[*runs-to-iff*]:

(*check e p*) · *s* ⦃*Q*⦄ ⟷ (*if* (∃ *x*. *p s x*) *then* (∀ *x*. *p s x* ⟶ *Q* (*Result x*) *s*) *else* *Q* (*Exn e*) *s*)
by (*auto simp add*: *runs-to.rep-eq run-check*)

lemma *runs-to-check*[*runs-to-vcg*]:
(∃ *x*. *p s x* ⟹ (∀ *x*. *p s x* ⟶ *Q* (*Result x*) *s*)) ⟹ (∀ *x*. ¬ *p s x*) ⟹ *Q* (*Exn e*) *s* ⟹
check e p · *s* ⦃*Q*⦄
by (*simp add*: *runs-to-check-iff*)

lemma *runs-to-partial-check*[*runs-to-vcg*]:
(∃ *x*. *p s x* ⟹ (∀ *x*. *p s x* ⟶ *Q* (*Result x*) *s*)) ⟹ (∀ *x*. ¬ *p s x*) ⟹ *Q* (*Exn e*) *s* ⟹
check e p · *s* ?⦃*Q*⦄
by (*simp add*: *runs-to-partial.rep-eq run-check*)

lemma *refines-check-right-ok*:
Q t a ⟹ *refines f* (*g a*) *s t R* ⟹ *refines f* (*check q Q* >>= *g*) *s t R*
by (*rule refines-bind-right-single*[*of a t*])
(*auto simp add*: *run-check*)

lemma *refines-check-right-fail*:
(∀ *a*. ¬ *Q t a*) ⟹ *f* · *s* ⦃ λ*r s'*. *R* (*r*, *s'*) (*Exn q*, *t*) ⦄ ⟹ *refines f* (*check q Q* >>= *g*) *s t R*
apply (*rule refines-bind-right-runs-to-partialI*[*OF runs-to-partial-of-runs-to*])
apply (*auto simp*: *runs-to-check-iff Exn-def refines-yield-right-iff*)
done

lemma *refines-throwError-check*:
(∀ *a*. ¬ *P t a*) ⟹ *R* (*Exn r*, *s*) (*Exn q*, *t*) ⟹
refines (*throw r*) (*check q P* >>= *f*) *s t R*
by (*intro refines-check-right-fail*) *auto*

lemma *refines-condition-neg-check*:
P s ⟷ (∀ *a*. ¬ *Q t a*) ⟹
(*P s* ⟹ ∀ *a*. ¬ *Q t a* ⟹ *R* (*Result r*, *s*) (*Exn q*, *t*)) ⟹
(⋀*a*. ¬ *P s* ⟹ *Q t a* ⟹ *refines f* (*g a*) *s t R*) ⟹
refines (*condition P* (*return r*) *f*) (*check q Q* >>= *g*) *s t R*
by (*intro refines-condition-left*)
(*auto intro*: *refines-check-right-ok refines-check-right-fail*)

16.9.37 *ignoreE*

named-theorems *ignoreE-simps* ‹*Rewrite rules to push* ***const***‹*ignoreE*› *inside.*›

lemma *ignoreE-eq*: *ignoreE f* = *vmap-value* (*map-exception-or-result* (λ*x*. *undefined*) *id*) *f*
unfolding *ignoreE-def catch-def*
by *transfer*

(*simp add*: *fun-eq-iff post-state-eq-iff eq-commute all-comm*[**where** *'a='a*]
split: *prod.splits exception-or-result-splits*)

lemma *run-ignoreE*: *run* (*ignoreE f*) *s* =
bind-post-state (*run f s*)
(λ(*r*, *t*). *case r of Exn e* ⇒ ⊥ | *Result v* ⇒ *pure-post-state* (*Result v*, *t*))
unfolding *ignoreE-def* **by** (*simp add*: *run-catch*)

lemma *runs-to-ignoreE-iff*[*runs-to-iff*]:
(*ignoreE f*) · *s* ⦃ *Q* ⦄ ⟷ *f* · *s* ⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *Q* (*Result v*) *t*)⦄
unfolding *ignoreE-eq runs-to.rep-eq*
by *transfer*
(*simp add*: *split-beta' eq-commute prod-eq-iff split*: *prod.splits exception-or-result-splits*)

lemma *runs-to-ignoreE*[*runs-to-vcg*]:
f · *s* ⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *Q* (*Result v*) *t*)⦄ ⟹ (*ignoreE f*) · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to-ignoreE-iff*)

lemma *liftE-le-iff-le-ignoreE*: *liftE f* ≤ *g* ⟷ *f* ≤ *ignoreE g*
unfolding *liftE-def ignoreE-eq*[*abs-def*]
by *transfer* (*simp add*: *le-fun-def vmap-post-state-le-iff-le-map-post-state id-def*)

lemma *runs-to-partial-ignoreE-iff*:
ignoreE f · *s* ?⦃ *P* ⦄ ⟷ *f* · *s* ?⦃ λ*r t*. ∀ *v*. *r* = *Result v* ⟶ *P* (*Result v*) *t* ⦄
unfolding *ignoreE-eq*
by *transfer* (*simp add*: *split-beta' prod-eq-iff eq-commute*)

lemma *runs-to-partial-ignoreE*[*runs-to-vcg*]:
f · *s* ?⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *Q* (*Result v*) *t*)⦄ ⟹ (*ignoreE f*) · *s* ?⦃ *Q* ⦄
by (*simp add*: *runs-to-partial-ignoreE-iff*)

lemma *mono-ignoreE*: *f* ≤ *f'* ⟹ *ignoreE f* ≤ *ignoreE f'*
unfolding *ignoreE-def*
apply (*rule mono-catch*)
by *simp-all*

lemma *monotone-ignoreE-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*)
⟹ *monotone R* (≤) (λ*f'*. *ignoreE* (*f f'*))
unfolding *ignoreE-def*
apply (*rule monotone-catch-le*)
by *simp-all*

lemma *monotone-ignoreE-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*)
⟹ *monotone R* (≥) (λ*f'*. *ignoreE* (*f f'*))
unfolding *ignoreE-def*
apply (*rule monotone-catch-ge*)
by *simp-all*

lemma *ignoreE-liftE* [*simp*]: *ignoreE* (*liftE f*) = *f*
by (*simp add*: *catch-liftE ignoreE-def*)

lemma *ignoreE-top*[*simp*]: *ignoreE* ⊤ = ⊤
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-bot*[*simp*]: *ignoreE bot* = *bot*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-fail*[*simp*]: *ignoreE fail* = *fail*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-return*[*simp*]: *ignoreE* (*return x*) = *return x*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-throw*[*simp*]: *ignoreE* (*throw x*) = *bot*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-yield-Exception*[*simp*]: *e* ≠ *default* ⟹ *ignoreE* (*yield* (*Exception e*)) = *bot*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-guard*[*simp*]: *ignoreE* (*guard P*) = *guard P*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-get-state*[*simp*]: *ignoreE* (*get-state*) = *get-state*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-set-state*[*simp*]: *ignoreE* (*set-state s*) = *set-state s*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-select*[*simp*]: *ignoreE* (*select S*) = *select S*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-unknown*[*simp*]: *ignoreE* (*unknown*) = *unknown*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-exec-concrete*[*simp*]: *ignoreE* (*exec-concrete st f*) = *exec-concrete st* (*ignoreE f*)
by (*rule spec-monad-eqI*) (*auto simp add*: *runs-to-iff*)

lemma *ignoreE-assert*[*simp*]: *ignoreE* (*assert P*) = *assert P*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-assume*[*simp*]: *ignoreE* (*assume P*) = *assume P*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-gets*[*simp*]: *ignoreE* (*gets f*) = *gets f*

by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-assert-opt*[*simp*]: *ignoreE* (*assert-opt v*) = *assert-opt v*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-gets-the*[*simp*]: *ignoreE* (*gets-the f*) = *gets-the f*
by (*rule spec-monad-eqI*) (*auto simp add*: *runs-to-iff*)

lemma *ignoreE-modify*[*simp*]: *ignoreE* (*modify f*) = *modify f*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-condition*[*ignoreE-simps*]:
ignoreE (*condition c f g*) = *condition c* (*ignoreE f*) (*ignoreE g*)
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-when*[*simp*]: *ignoreE* (*when P f*) = *when P* (*ignoreE f*)
by (*simp add*: *when-def ignoreE-condition*)

lemma *ignoreE-bind*[*ignoreE-simps*]:
ignoreE (*bind f g*) = *bind* (*ignoreE f*) (λ*v*. (*ignoreE* (*g v*)))
by (*simp add*: *spec-monad-eq-iff runs-to-ignoreE-iff runs-to-bind-iff*)

lemma *ignoreE-map-exn*[*simp*]: *ignoreE* (*map-value* (*map-exn f*) *g*) = *ignoreE g*
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *ignoreE-whileLoop*[*ignoreE-simps*]:
ignoreE (*whileLoop C B I*) = *whileLoop C* (λ*x*. *ignoreE* (*B x*)) *I*
proof −
have (λ*I*. *ignoreE* (*whileLoop C B I*)) = *whileLoop C* (λ*x*. *ignoreE* (*B x*))
unfolding *whileLoop-def*
apply (*rule gfp-fusion*[*OF - mono-whileLoop-functional mono-whileLoop-functional*,
where *g*=λ*x a*. *liftE* (*x a*)])
subgoal by (*simp add*: *le-fun-def liftE-le-iff-le-ignoreE*)
subgoal by (*simp add*: *ignoreE-bind ignoreE-condition*)
done
then show *?thesis* **by** (*simp add*: *fun-eq-iff*)
qed

16.9.38 *on-exit′*

lemmas *bind-finally-def* = *bind-exception-or-result-def*

lemma *bind-exception-or-result-liftE-assoc*:
bind-exception-or-result (*bind* (*liftE f*) *g*) *h* = *bind f* (λ*v*. *bind-exception-or-result* (*g v*) *h*)
by (*rule spec-monad-eqI*) (*auto simp add*: *runs-to-iff*)

lemma *bind-exception-or-result-bind-guard-assoc*:
bind-exception-or-result (*bind* (*guard P*) *g*) *h* =

bind (*guard P*) (λ*v*. *bind-exception-or-result* (*g v*) *h*)
by (*rule spec-monad-eqI*) (*auto simp add*: *runs-to-iff*)

lemma *on-exit-bind-exception-or-result-conv*:
on-exit f cleanup = *bind-exception-or-result f* (λ*x*. *do* {*state-select cleanup*; *yield x*})
by (*simp add*: *on-exit-def on-exit′-def*)

lemma *guard-on-exit-bind-exception-or-result-conv*:
guard-on-exit f P cleanup =
bind-exception-or-result f (λ*x*. *do* {*guard P*; *state-select cleanup*; *yield x*})
by (*simp add*: *on-exit′-def bind-assoc*)

lemma *assume-result-and-state-check-only-state*:
assume-result-and-state (λ*s*. {((), *s′*). *s′* = *s* ∧ *P s*}) = *assuming P*
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *assume-on-exit-bind-exception-or-result-conv*:
assume-on-exit f P cleanup = *bind-exception-or-result f*
(λ*x*. *do* {*assume-result-and-state* (λ*s*. {((), *s′*). *s′* = *s* ∧ *P s*}); *state-select cleanup*; *yield x*})
by (*simp add*: *on-exit′-def bind-assoc assume-result-and-state-check-only-state*)

lemma *monotone-on-exit′-le*[*partial-function-mono*]:
monotone R (≤) (λ*f′*. *f f′*) ⟹ *monotone R* (≤) (λ*f′*. *g f′*) ⟹
monotone R (≤) (λ*f′*. *on-exit′* (*f f′*) (*g f′*))
unfolding *on-exit′-def* **by** (*intro partial-function-mono*)

lemma *monotone-on-exit′-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f′*. *f f′*) ⟹ *monotone R* (≥) (λ*f′*. *g f′*) ⟹
monotone R (≥) (λ*f′*. *on-exit′* (*f f′*) (*g f′*))
unfolding *on-exit′-def* **by** (*intro partial-function-mono*)

lemma *runs-to-on-exit′-iff*[*runs-to-iff*]:
on-exit′ f c · *s* ⦃ *Q* ⦄ ⟷
f · *s* ⦃ λ*r t*. *c* · *t* ⦃ λ*q t*. *Q* (*case q of Exception e* ⇒ *Exception e* | *Result* - ⇒ *r*) *t* ⦄ ⦄
by (*auto simp add*: *on-exit′-def runs-to-iff fun-eq-iff split*: *exception-or-result-split*
intro!: *arg-cong*[**where** *f*=*runs-to* - -])

lemma *runs-to-on-exit′*[*runs-to-vcg*]:
f · *s* ⦃ λ*r t*. *c* · *t* ⦃ λ*q t*. *Q* (*case q of Exception e* ⇒ *Exception e* | *Result* - ⇒ *r*) *t* ⦄ ⦄ ⟹ *on-exit′ f c* · *s* ⦃ *Q* ⦄
by (*simp add*: *runs-to-on-exit′-iff*)

lemma *runs-to-partial-on-exit′*[*runs-to-vcg*]:
f · *s* ?⦃ λ*r t*. *c* · *t* ?⦃ λ*q t*. *Q* (*case q of Exception e* ⇒ *Exception e* | *Result* - ⇒ *r*) *t* ⦄ ⦄ ⟹ *on-exit′ f c* · *s* ?⦃ *Q* ⦄
unfolding *on-exit′-def*

apply (*rule runs-to-partial-bind-exception-or-result*)
apply (*rule runs-to-partial-weaken*, *assumption*)
apply (*rule runs-to-partial-bind*)
apply (*rule runs-to-partial-weaken*, *assumption*)
apply *auto*
done

lemma *refines-on-exit'*:
assumes *f*: *refines f f' s s'*
(λ(*r*, *t*) (*r'*, *t'*). *refines c c' t t'* (λ(*q*, *t*) (*q'*, *t'*). *R*
(*case q of Exception e ⇒ Exception e | Result - ⇒ r*, *t*)
(*case q' of Exception e' ⇒ Exception e' | Result - ⇒ r'*, *t'*)))
shows *refines* (*on-exit' f c*) (*on-exit' f' c'*) *s s' R*
unfolding *on-exit'-def*
apply (*rule refines-bind-exception-or-result*[*OF f*[*THEN refines-weaken*]])
apply (*clarsimp intro*!: *refines-bind'*)
apply (*rule refines-weaken*, *assumption*)
apply *auto*
done

lemma *on-exit'-skip*: *on-exit' f skip* = *f*
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

16.9.39 *run-bind*

lemma *run-run-bind*: *run* (*run-bind f t g*) *s* = *bind-post-state* (*run f t*) (λ(*r*, *t*). *run* (*g r t*) *s*)
by (*transfer*) *simp*

lemma *runs-to-run-bind-iff*[*runs-to-iff*]:
(*run-bind f t g*) · *s* ⦃*Q*⦄ ⟷ *f* · *t* ⦃λ*r t*. (*g r t*) · *s* ⦃*Q*⦄ ⦄
by (*simp add*: *runs-to.rep-eq run-run-bind split-beta'*)

lemma *runs-to-run-bind*[*runs-to-vcg*]:
f · *t* ⦃λ*r t*. (*g r t*) · *s* ⦃*Q*⦄ ⦄ ⟹ (*run-bind f t g*) · *s* ⦃*Q*⦄
by (*simp add*: *runs-to-run-bind-iff*)

lemma *runs-to-partial-run-bind*[*runs-to-vcg*]:
f · *t* ?⦃λ*r t*. (*g r t*) · *s* ?⦃*Q*⦄ ⦄ ⟹ (*run-bind f t g*) · *s* ?⦃*Q*⦄
by (*auto simp*: *runs-to-partial.rep-eq run-run-bind split-beta'*
intro!: *holds-partial-bind-post-state*)

lemma *mono-run-bind*: *f* ≤ *f'* ⟹ *g* ≤ *g'* ⟹ *run-bind f t g* ≤ *run-bind f' t g'*
apply (*rule le-spec-monad-runI*)
apply (*simp add*: *run-run-bind*)
apply (*rule mono-bind-post-state*)
apply (*auto simp add*: *le-fun-def less-eq-spec-monad.rep-eq split*: *exception-or-result-splits*)
done

lemma *monotone-run-bind-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*) ⟹ (⋀*r t*. *monotone R* (≤) (λ*f'*. *g f' r t*))
⟹ *monotone R* (≤) (λ*f'*. *run-bind* (*f f'*) *t* (*g f'*))
apply (*simp add*: *monotone-def*)
using *mono-run-bind*
by (*metis le-fun-def*)

lemma *monotone-run-bind-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ (⋀*r t*. *monotone R* (≥) (λ*f'*. *g f' r t*))
⟹ *monotone R* (≥) (λ*f'*. *run-bind* (*f f'*) *t* (*g f'*))
apply (*simp add*: *monotone-def*)
using *mono-run-bind*
by (*metis le-fun-def*)

lemma *liftE-run-bind*: *liftE* (*run-bind f t g*) = *run-bind f t* (λ*r t*. *liftE* (*g r t*))
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *exec-concrete-run-bind*: *exec-concrete st f* =
do {
s ← *get-state*;
t ← *select* {*t*. *st t* = *s*};
run-bind f t (λ*r' t'*. *do* {*set-state* (*st t'*); *yield r'* })
}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *exec-abstract-run-bind*: *exec-abstract st f* =
do {
s ← *get-state*;
run-bind f (*st s*) (λ*r' t'*. *do* {*s'* ← *select* {*s'*. *t'* = *st s'*}; *set-state s'*; *yield r'* })
}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *refines-run-bind*:
refines f f' x y (λ(*r*,*x'*) (*r'*, *y'*). *refines* (*g r x'*) (*g' r' y'*) *s t R*) ⟹
refines (*run-bind f x g*) (*run-bind f' y g'*) *s t R*
apply *transfer*
apply (*rule sim-bind-post-state*, *assumption*)
apply *auto*
done

16.9.40 Iteration of monadic actions

fun *iter-spec-monad* **where**

iter-spec-monad f 0 = *skip* |
iter-spec-monad f (*Suc n*) = *do* {*iter-spec-monad f n*; *f n* }

lemma *iter-spec-monad-unfold*:
0 < *n* ⟹ *iter-spec-monad f n* = *do* {*iter-spec-monad f* (*n*−*1*); *f* (*n*−*1*) }
by (*metis Suc-pred′ iter-spec-monad.simps*(*2*))

lemma *iter-spec-monad-cong*:
(⋀*j*. *j* < *i* ⟹ *f j* = *g j*) ⟹ *iter-spec-monad f i* = *iter-spec-monad g i*
by (*induction i*) *auto*

16.9.41 *forLoop*

definition *forLoop*:: *int* ⇒ *int* ⇒ (*int* ⇒ ('*a*, '*s*) *res-monad*) ⇒ (*int*, '*s*) *res-monad*
where
forLoop z (*m*::*int*) *B* = *whileLoop* (λ*i s*. *i* < *m*) (λ*i*. *do* {*B i*; *return* (*i* + *1*) }) *z*

lemma *runs-to-forLoop*:
assumes *z* ≤ *m*
assumes *I*: ⋀*r s*. *I r s* ⟹ *z* ≤ *r* ⟹ *r* < *m* ⟹ (*B r*) · *s* ⦃ λ- *t*. *I* (*r* + *1*) *t* ⦄
assumes *Q*: ⋀*s*. *I m s* ⟹ *Q m s*
shows *I z s* ⟹ (*forLoop z m B*) · *s* ⦃λ*Res r*. *Q r*⦄
unfolding *forLoop-def*
using ‹*z* ≤ *m*› *Q I*
by (*intro runs-to-whileLoop-res*[**where** *I*=λ*q t*. *I q t* ∧ *z* ≤ *q* ∧ *q* ≤ *m*
and *P*=*Q*
and *R*=*measure* (λ(*i*, -). *nat* (*m* − *i*))])
(*auto intro*!: *runs-to-bind*)

lemma *whileLoop-cong-inv*:
run ((*whileLoop C f z*)::('*a*, '*s*) *res-monad*) *s* = *run* (*whileLoop D g z*) *s*
if *I*: ⋀*i s*. *I i s* ⟹ *C i s* ⟹ *f i* · *s* ?⦃ λ*v′ s′*. ∀ *i′*. *v′* = *Result i′* ⟶ *I i′ s′*⦄
and *I-eq*: ⋀*i s*. *I i s* ⟹ *C i s* ⟹ *run* (*f i*) *s* = *run* (*g i*) *s*
and *C-eq*: ⋀*i s*. *I i s* ⟹ *C i s* ⟷ *D i s*
and *I z s*
for *s*::'*s* **and** *z*::'*a* **and** *R*::('*a* × '*s*) *rel*
proof −
have *rel-spec* (*whileLoop C f z*) (*whileLoop D g z*) *s s*
(λ(*Res i*, *s*) (*Res i′*, *s′*). *i* = *i′* ∧ *s* = *s′* ∧ *I i s*)
apply (*intro rel-spec-whileLoop-res*)
subgoal for *i s j t* **using** *C-eq*[*of i s*] **by** *auto*
subgoal premises *prems* **for** *i s j t*
using *prems I*[*of i s*]
apply (*clarsimp simp*: *rel-spec.rep-eq runs-to-partial.rep-eq I-eq*
intro!: *rel-post-state-refl′*)
apply (*auto intro*: *holds-partial-post-state-weaken*)
done
subgoal using ‹*I z s*› **by** *simp*
done

from *rel-spec-mono*[*OF* - *this*, *of* (=)]
show *?thesis*
by (*auto simp*: *le-fun-def rel-spec-eq*)
qed

lemma *forLoop-skip*: *forLoop z m f* = *return z* **if** $z \geq m$
using *that* **unfolding** *forLoop-def*
by (*subst whileLoop-unroll*) *simp*

lemma *forLoop-eq-whileLoop*: *forLoop z m f* = *whileLoop C g z*
if $\bigwedge i.\ z \leq i \Longrightarrow i < m \Longrightarrow g\ i = do\ \{\ y \leftarrow f\ i;\ return\ (i + 1)\ \}$
$\bigwedge i\ s.\ z \leq i \Longrightarrow i \leq m \Longrightarrow C\ i\ s \longleftrightarrow i < m$
$\bigwedge s.\ z > m \Longrightarrow \neg\ C\ z\ s$
proof *cases*
assume $z \leq m$
then show *?thesis*
unfolding *forLoop-def*
apply (*intro spec-monad-ext ext whileLoop-cong-inv*[**where** $I{=}\lambda i\ s.\ z \leq i \wedge i \leq m$])
apply (*auto simp*: *that intro*!: *runs-to-partial-bind*)
done
next
assume $\neg\ z \leq m$
then show *?thesis*
apply (*subst whileLoop-unroll*)
apply (*simp add*: *forLoop-skip*)
apply (*rule spec-monad-ext*)
apply (*simp add*: *that*)
done
qed

lemma *runs-to-partial-forLoopE*: *forLoop z m f* $\cdot$ *s ?*$\{\!|$ $\lambda x\ s'.\ \forall\, v.\ x = Result\ v \longrightarrow v = m$ $|\!\}$
if $z \leq m$
proof −
have $*$: *forLoop z m f* = *whileLoop* (λi -. $i \neq m$) ($\lambda i.$ *do* {*f i*; *return* $(i + 1)$ }) *z*
using ‹$z \leq m$›
by (*auto intro*!: *forLoop-eq-whileLoop*)
show *?thesis* **unfolding** $*$
by (*rule runs-to-partial-whileLoop-cond-false*[*THEN runs-to-partial-weaken*]) *simp*
qed

16.9.42 *whileLoop-unroll-reachable*

context fixes *C* :: $'a \Rightarrow 's \Rightarrow bool$ **and** *B* :: $'a \Rightarrow ('e{::}default, 'a, 's)$ *spec-monad*
begin

inductive *whileLoop-unroll-reachable* :: $'a \Rightarrow 's \Rightarrow 'a \Rightarrow 's \Rightarrow bool$ **for** *a s* **where**

initial[*intro*, *simp*]: *whileLoop-unroll-reachable a s a s*
| *step*: $\bigwedge b\ t\ X\ c\ u.$
whileLoop-unroll-reachable a s b t $\Longrightarrow$ *C b t* $\Longrightarrow$
run (*B b*) *t* = *Success X* $\Longrightarrow$ (*Result c*, *u*) $\in$ *X* $\Longrightarrow$
whileLoop-unroll-reachable a s c u

lemma *whileLoop-unroll-reachable-trans*:
assumes *a-b*: *whileLoop-unroll-reachable a s b t* **and** *b-c*: *whileLoop-unroll-reachable b t c u*
shows *whileLoop-unroll-reachable a s c u*
proof (*use b-c* **in** ‹*induction arbitrary*: ›)
case (*step c u X d v*)
show *?case*
by (*rule whileLoop-unroll-reachable.step*[*OF step*(*5*,*2*,*3*,*4*)])
qed (*simp add*: *a-b*)

end

lemma *run-whileLoop-unroll-reachable-cong*:
assumes *eq*:
$\bigwedge b\ t.$ *whileLoop-unroll-reachable C B a s b t* $\Longrightarrow$ *C b t* $\longleftrightarrow$ *C′ b t*
$\bigwedge b\ t.$ *whileLoop-unroll-reachable C B a s b t* $\Longrightarrow$ *C b t* $\Longrightarrow$ *run* (*B b*) *t* = *run* (*B′ b*) *t*
shows *run* (*whileLoop C B a*) *s* = *run* (*whileLoop C′ B′ a*) *s*
proof −
have *rel-spec* (*whileLoop C B a*) (*whileLoop C′ B′ a*) *s s*
(λ(*x′*, *s′*) (*y*, *t*). *y* = *x′* $\wedge$ *t* = *s′*$\wedge$
($\forall$ *a′*. *x′* = *Result a′* $\longrightarrow$ *whileLoop-unroll-reachable C B a s a′ s′*))
apply (*rule rel-spec-whileLoop*)
subgoal by (*simp add*: *eq*)
subgoal for *a′ s′*
using *whileLoop-unroll-reachable.step*[*of C B a s a′ s′*]
by (*cases run* (*B′ a′*) *s′*) (*auto simp add*: *rel-spec-def eq intro*!: *rel-set-refl*)
subgoal by *simp*
subgoal for *a′ s′ b′ t′* **by** (*cases a′*) (*auto simp add*: *rel-exception-or-result.simps*)
done
then have *rel-spec* (*whileLoop C B a*) (*whileLoop C′ B′ a*) *s s* (=)
by (*rule rel-spec-mono*[*rotated*]) *auto*
then show *?thesis*
by (*simp add*: *rel-spec-eq*)
qed

16.9.43 *on-exit*

lemma *refines-rel-prod-on-exit*:
assumes *f*: *refines* f_c f_a s_c s_a (*rel-prod R S′*)
assumes *cleanup*: $\bigwedge s_c\ s_a\ t_c.$ *S′ s_c s_a* $\Longrightarrow$ (s_c, t_c) $\in$ $cleanup_c$ $\Longrightarrow$ $\exists\, t_a.$ (s_a, t_a) $\in$ $cleanup_a$ $\wedge$ *S* t_c t_a
assumes *emp*: $\bigwedge s_c\ s_a.$ *S′ s_c s_a* $\Longrightarrow$ $\nexists\, t_c.$ (s_c, t_c) $\in$ $cleanup_c$ $\Longrightarrow$ $\nexists\, t_a.$ (s_a, t_a) $\in$

$cleanup_a$
shows *refines* (*on-exit* f_c $cleanup_c$) (*on-exit* f_a $cleanup_a$) s_c s_a (*rel-prod* R S)
unfolding *on-exit-bind-exception-or-result-conv*
apply (*rule refines-bind-exception-or-result*)
apply (*rule refines-mono* [*OF* - f])
apply (*clarsimp*)
apply (*rule refines-bind'*)
apply (*rule refines-state-select*)
using *cleanup emp*
apply *auto*
done

lemma *refines-runs-to-partial-rel-prod-on-exit*:
assumes f: *refines* f_c f_a s_c s_a (*rel-prod* R S')
assumes *runs-to*: $f_c \bullet s_c$?$\{\!|\lambda r\ t.\ P\ t|\!\}$
assumes *cleanup*: $\bigwedge s_c\ s_a\ t_c.\ S'\ s_c\ s_a \Longrightarrow (s_c,\ t_c) \in cleanup_c \Longrightarrow P\ s_c \Longrightarrow \exists\, t_a.\ (s_a,\ t_a) \in cleanup_a \wedge S\ t_c\ t_a$
assumes *emp*: $\bigwedge s_c\ s_a.\ S'\ s_c\ s_a \Longrightarrow P\ s_c \Longrightarrow \nexists\, t_c.\ (s_c,\ t_c) \in cleanup_c \Longrightarrow \nexists\, t_a.\ (s_a,\ t_a) \in cleanup_a$
shows *refines* (*on-exit* f_c $cleanup_c$) (*on-exit* f_a $cleanup_a$) s_c s_a (*rel-prod* R S)
proof −
from *refines-runs-to-partial-fuse* [*OF f runs-to*]
have *refines* f_c f_a s_c s_a ($\lambda(r,\ t)$ (r', t'). *rel-prod* R S' (r, t) (r', t') $\wedge$ P t) **.**
moreover have ($\lambda(r,\ t)$ (r', t'). *rel-prod* R S' (r, t) (r', t') $\wedge$ P t) = *rel-prod* R ($\lambda t\ t'.\ S'\ t\ t' \wedge P\ t$)
by (*auto simp add*: *rel-prod-conv*)
ultimately have *refines* f_c f_a s_c s_a (*rel-prod* R ($\lambda t\ t'.\ S'\ t\ t' \wedge P\ t$)) **by** *simp*
then show *?thesis*
apply (*rule refines-rel-prod-on-exit*)
subgoal using *cleanup* **by** *blast*
subgoal using *emp* **by** *blast*
done
qed

lemma *rel-spec-monad-mono*:
assumes Q: *rel-spec-monad* R Q f g **and** QQ': $\bigwedge x\ y.\ Q\ x\ y \Longrightarrow Q'\ x\ y$
shows *rel-spec-monad* R Q' f g
proof −
have *rel-spec-monad* R $Q \leq$ *rel-spec-monad* R Q'
unfolding *rel-spec-monad-def* **using** QQ'
by (*auto simp add*: *rel-post-state.simps rel-set-def*)
(*metis rel-prod-sel*)+
with Q **show** *?thesis* **by** *auto*
qed

lemma *gets-return*: *gets* (λ-. x) = *return* x
by (*rule spec-monad-eqI*) (*auto simp add*: *runs-to-iff*)

lemma *bind-handle-bind-exception-or-result-conv*:
bind-handle f g h =
bind-exception-or-result f
(λ*Exception e* ⇒ *h e* | *Result v* ⇒ *g v*)
apply (*rule spec-monad-eqI*)
by (*auto simp add*: *runs-to-iff*)
(*auto elim*!: *runs-to-weaken split*: *exception-or-result-splits*)

lemma *bind-handle-bind-exception-or-result-conv-exn*: *bind-handle f g* (λ*Some e* ⇒ *h e*) =
bind-exception-or-result f
(λ*Exn e* ⇒ *h e* | *Result v* ⇒ *g v*)
by (*simp add*: *bind-handle-bind-exception-or-result-conv case-xval-def*)

lemma *try-nested-bind-exception-or-result-conv*:
shows *try* (*f* >>= *g*) =
(*bind-exception-or-result f*
(λ*Exn e* ⇒ (*case e of Inl l* ⇒ *throw l* | *Inr r* ⇒ *return r*)
| *Result v* ⇒ *try* (*g v*)))
apply (*rule spec-monad-eqI*)
by (*auto simp add*: *runs-to-iff*)
(*auto elim*!: *runs-to-weaken simp add*: *runs-to-iff unnest-exn-def*
split: *xval-splits sum.splits*)

lemma *try-nested-bind-handle-conv*:
shows *try* (*f* >>= *g*) =
(*bind-handle f* (λ*v*. *try* (*g v*))
(λ*Some e* ⇒ (*case e of Inl l* ⇒ *throw l* | *Inr r* ⇒ *return r*)))
by (*simp add*: *bind-handle-bind-exception-or-result-conv-exn try-nested-bind-exception-or-result-conv*)

definition *no-fail*:: ('*s* ⇒ *bool*) ⇒ ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ *bool* **where**
no-fail P f ≡ ∀ *s*. *P s* ⟶ *run f s* ≠ ⊤

definition *no-throw*:: ('*s* ⇒ *bool*) ⇒ ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ *bool* **where**
no-throw P f ≡ ∀ *s*. *P s* ⟶ *f* ⋅ *s* ?⦃ λ*r t*. ∃ *v*. *r* = *Result v*⦄

definition *no-return*:: ('*s* ⇒ *bool*) ⇒ ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ *bool* **where**
no-return P f ≡ ∀ *s*. *P s* ⟶ *f* ⋅ *s* ?⦃ λ*r t*. ∃ *e*. *r* = *Exception e* ∧ *e* ≠ *default*⦄

lemma *no-return-exn-def*: *no-return P f* ⟷ (∀ *s*. *P s* ⟶ *f* ⋅ *s* ?⦃ λ*r t*. ∃ *e*. *r* = *Exn e*⦄)
by (*auto simp add*: *no-return-def Exn-def default-option-def elim*!: *runs-to-partial-weaken*
)

lemma *no-throw-gets*[*simp*]: *no-throw P* (*gets f*)
by (*auto simp add*: *no-throw-def runs-to-partial-def*)

lemma *no-throw-modify*[*simp*]: *no-throw P* (*modify f*)
by (*auto simp add*: *no-throw-def runs-to-partial-def*)

lemma *no-throw-select*[*simp*]: *no-throw P* (*select f*)
by (*auto simp add*: *no-throw-def runs-to-partial-def*)

lemma *always-progress-select-UNIV*[*simp*]: *always-progress* (*select UNIV*)
by (*auto simp add*: *always-progress-def bot-post-state-def*)

lemma *rel-spec-monad-rel-xval-catch*:
assumes *fh*: *rel-spec-monad R* (*rel-xval E Q*) *f h*
assumes *gi*: *rel-fun E* (*rel-spec-monad R* (*rel-xval E2 Q*)) *g i*
shows *rel-spec-monad R* (*rel-xval E2 Q*) (*f* <*catch*> *g*) (*h* <*catch*> *i*)
using *assms* **unfolding** *rel-spec-monad-iff-rel-spec*
by (*auto intro*!: *rel-spec-catch simp*: *rel-fun-def*)

16.10 Setup for Tagging

lemma *runs-to-tag*:
(¶ *tag* ⟹ *f* · *s* {‖ *P* ‖}) ⟹ (*tag* | *f*) · *s* {‖ *P* ‖}
unfolding *TAG-def ASM-TAG-def* **by** *auto*

lemma *runs-to-tag-guard*:
fixes *g* :: *'a* ⇒ *bool*
and *s* :: *'a*
assumes *tag* | *g s*
assumes *P* (*Result* ()) *s*
shows (*tag* | *guard g*) · *s* {‖ *P* ‖}
using *assms* **unfolding** *TAG-def* **by** − (*rule runs-to-weaken*, *rule runs-to-guard*; *simp*)

bundle *runs-to-vcg-tagging-setup*
begin

unbundle *basic-vcg-tagging-setup*

lemmas [*runs-to-vcg*] = *runs-to-tag runs-to-tag-guard*

end

lemma *refines-set-state-right*:
refines f (*set-state t'* ⤜ *g*) *s t R* ⟷ *refines f* (*g* ()) *s t' R*
by (*auto simp add*: *refines-iff' runs-to-iff*)

lemma *refines-bind-modify-right-iff*:
refines f (*modify m* ⤜ *g*) *s t R* ⟷ *refines f* (*g* ()) *s* (*m t*) *R*
by (*auto simp*: *refines-iff' runs-to-iff*)

lemma *refines-guard-right-iff*:
refines f (*guard P*) *s t R* ⟷ (*P t* ⟶ *refines f skip s t R*)
using *refines-bind-guard-right-iff*[*of f P return s t R*] **by** *simp*

lemma *refines-select-right-witness*:
x ∈ *X* ⟹ *refines f* (*g x*) *s t Q* ⟹ *refines f* ((*select X*) >>= *g*) *s t Q*
by (*simp add*: *refines-iff' runs-to-iff*)

lemmas *refines-select-right* = *refines-select-right-witness*

lemma *refines-unknown-right*:
refines f (*g x*) *s t R* ⟹ *refines f* (*unknown* >>= *g*) *s t R*
unfolding *unknown-def* **by** (*simp add*: *refines-select-right*)

lemma *refines-bind-liftE-modify-iff*:
refines f (*do* {
liftE g;
modify h
}) *x y R* =
refines f (*do* {
g;
modify h
}) *x y* (λ(*x*, *s'*) (*y*, *t'*). ∀ *v*. *y* = *Result v* ⟶ *R* (*x*, *s'*) (*Result v*, *t'*))
by (*subst refines-liftE-right-iff*[*symmetric*]) (*simp flip*: *liftE-bind*)

lemma *refines-get-state-right-iff*:
refines f (*do* {*x* ← *get-state*; *h x* }) *s t Q* ⟷
refines f (*h t*) *s t Q*
by (*auto simp add*: *refines-iff' runs-to-iff*)

lemma *refines-gets-the-right-iff*:
refines f (*do* {*x* ← *gets-the m*; *h x* } :: ('*a*::*default*, '*b*, '*c*) *spec-monad*) *s t Q* ⟷
(∀ *r*. *m t* = *Some r* ⟶ *refines f* (*h r*) *s t Q*)
by (*auto simp add*: *refines-iff' runs-to-iff*)

lemma *refines-gets-the-right*:
(⋀*r*. *m t* = *Some r* ⟹ *refines f* (*h r*) *s t Q*) ⟹
refines f (*do* {*x* ← *gets-the m*; *h x* } :: ('*a*::*default*, '*b*, '*c*) *spec-monad*) *s t Q*
by (*simp add*: *refines-gets-the-right-iff*)

lemma *refines-assert-right-iff*:
refines f (*do* {*x* ← *assert P*; *h x* } :: ('*a*::*default*, '*b*, '*c*) *spec-monad*) *s t Q* ⟷
(*P* ⟶ *refines f* (*h r*) *s t Q*)
by (*auto simp add*: *refines-iff' runs-to-iff*)

lemma *refines-modify-right*:
refines C (*modify f*) *s t R* ⟷ (*C* ⋆ *s* ⦇ λ*r s'*. *R* (*r*, *s'*) (*Result* (), *f t*) ⦈)
by (*cases run C s*) (*simp-all add*: *refines.rep-eq runs-to.rep-eq*)

lemma *refines-liftE-left*:
refines (*liftE f*) *g s t R* ⟷ *refines f g s t* (λ(*Res a*, *s*) *b*. *R* (*Result a*, *s*) *b*)
by (*auto simp add*: *refines-liftE-left-iff intro*!: *arg-cong*[**where** *f*=*refines f g s t*])

lemma *refines-liftE*:
refines f g s t R $\Longrightarrow$
refines (*liftE f*) (*liftE g*) *s t*
(λ(*x*, *s'*) (*y*, *t'*). $\exists$ *r q*. *x* = *Result r* $\wedge$ *y* = *Result q* $\wedge$ *R* (*Result r*, *s'*) (*Result q*, *t'*))
unfolding *refines-liftE-left refines-liftE-right-iff*
apply (*rule refines-weaken*, *assumption*)
apply *auto*
done

lemma *refines-catch-right*:
assumes *refines f g s t* (λ(*x*, *s'*).
λ(*Result y*, *t'*) $\Rightarrow$ *R* (*x*, *s'*) (*Result y*, *t'*)
|(*Exn e*, *t'*) $\Rightarrow$ *refines* (*yield x*) (*h e*) *s' t' R*)
shows *refines f* (*g* <*catch*> *h*) *s t R*
using *assms*
apply (*auto simp add*: *refines-iff' runs-to-iff*)
subgoal premises *prems* **for** *P*
apply (*rule prems*[*rule-format*])
apply (*rule runs-to-weaken*[*OF prems*(*2*)])
apply (*force split*: *xval-splits*)
done
done

lemma *return-catch*:
(*return x* <*catch*> *H*) = *return x*
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *refines-gets-right*:
f $\cdot$ *s* $\{\!|$ λ*r s'*. *R* (*r*, *s'*) (*Result* (*m t*), *t*) $|\!\}$ $\Longrightarrow$ *refines f* (*gets m*) *s t R*
by (*auto simp add*: *refines-iff' runs-to-iff intro*: *runs-to-weaken*)

lemma *refines-when*:
P $\longleftrightarrow$ *P'* $\Longrightarrow$
(*P* $\Longrightarrow$ *P'* $\Longrightarrow$ *refines f f' s t R*) $\Longrightarrow$
($\neg$ *P* $\Longrightarrow$ $\neg$ *P'* $\Longrightarrow$ *R* (*Result* (), *s*) (*Result* (), *t*)) $\Longrightarrow$
refines (*when P f*) (*when P' f'*) *s t R*
by (*cases P*; *simp*)

lemma *refines-case-prod*:
($\bigwedge$*a b c d*. *x* = (*a*, *b*) $\Longrightarrow$ *y* = (*c*, *d*) $\Longrightarrow$ *refines* (*f a b*) (*g c d*) *s t R*) $\Longrightarrow$
refines (*case x of* (*a*, *b*) $\Rightarrow$ *f a b*) (*case y of* (*c*, *d*) $\Rightarrow$ *g c d*) *s t R*
by (*cases x*; *cases y*; *auto*)

lemma *refines-unless*:
($\neg$ *P* $\Longrightarrow$ *refines f g s t R*) $\Longrightarrow$ (*P* $\longrightarrow$ *R* (*Result* (), *s*) (*Result* (), *t*)) $\Longrightarrow$
refines (*unless P f*) (*unless P g*) *s t R*
unfolding *when-def*

apply (*rule refines-condition*)
apply (*auto intro*: *refines-yield*)
done

lemma *refines-ext-left*:
$f \cdot s\text{-}A \{\!|\ P\ |\!\} \Longrightarrow$ *refines g f s-C s-A R* $\Longrightarrow$
refines g f s-C s-A ($\lambda a\ b.\ R\ a\ b \wedge P\ (fst\ b)\ (snd\ b)$)
by *transfer* (*force elim*!: *sim-post-state.cases simp*: *sim-set-def*)

lemma *refines-return-throw*:
refines (*return x*) (*throw e*) *s t R* $\longleftrightarrow$ *R* (*Result x*, *s*) (*Exn e*, *t*)
by (*simp add*: *refines-def Exn-def*)

lemma *refines-returnOk-throw*:
refines (*return x*) (*throw e*) *s t R* $\longleftrightarrow$ *R* (*Result x*, *s*) (*Exn e*, *t*)
by (*simp*)

lemma *refines-throw-returnOk*:
refines (*throw e*) (*return x*) *s t R* $\longleftrightarrow$ *R* (*Exn e*, *s*) (*Result x*, *t*)
by (*simp add*: *refines-def Exn-def*)

lemma *refines-condition-right-iff*:
refines f (*condition P g h*) *s t R* $\longleftrightarrow$ *refines f* (*if P t then g else h*) *s t R*
by (*simp add*: *refines-def*)

lemma *refines-gets-theE-right-iff*:
m t = *Some x* $\Longrightarrow$
refines f (*gets-the m* >>= *g*) *s t R* $\longleftrightarrow$ *refines f* (*g x*) *s t R*
by (*simp add*: *refines-iff' runs-to-iff*)

lemma *refines-bind-bindE*:
assumes *f*: *refines f f' s t Q*
assumes *g'*: $\bigwedge r\ s\ x\ t.\ Q$ (*Result r*, *s*) (*Result x*, *t*) $\Longrightarrow$ *refines* (*g r*) (*g' x*) *s t R*
assumes *g*: $\bigwedge r\ s\ e\ t.\ Q$ (*Result r*, *s*) (*Exn e*, *t*) $\Longrightarrow$ (*g r*) $\cdot$ *s* $\{\!|\ \lambda r\ s'.\ R\ (r,\ s')$
(*Exn e*, *t*) $|\!\}$
shows *refines* ((*bind f g*)::('*a*, '*s*) *res-monad*) (*f'* >>= *g'*) *s t R*
apply (*rule refines-bind*[*OF f*])
subgoal by *simp*
subgoal by *simp*
subgoal
using *g* **by** (*auto simp add*: *refines-yield-right-iff default-option-def Exn-def*)
subgoal using *g'* **by** *simp*
done

lemma *refines-whileLoop-cond-exn*:
assumes *C*: $\bigwedge x\ s\ x'\ s'.\ R$ (*Result x*, *s*) (*Result x'*, *s'*) $\Longrightarrow$ *C x s* = *C' x' s'*
assumes *B*: $\bigwedge x\ s\ x'\ s'.\ R$ (*Result x*, *s*) (*Result x'*, *s'*) $\Longrightarrow$ *C x s* $\Longrightarrow$ *C' x' s'* $\Longrightarrow$
refines (*B x*) (*B' x'*) *s s' R*
assumes *R-ER*: $\bigwedge e\ s\ x'\ s'.\ e \neq default \Longrightarrow R$ (*Exception e*, *s*) (*Result x'*, *s'*)

$\Longrightarrow \neg\ C'\ x'\ s'$
assumes *R-RE*: $\bigwedge x\ s\ e'\ s'.\ e' \neq default \Longrightarrow R\ (Result\ x,\ s)\ (Exception\ e',\ s')$
$\Longrightarrow \neg\ C\ x\ s$
assumes *x*: $R\ (Result\ x,\ s)\ (Result\ x',\ s')$
shows *refines* $(whileLoop\ C\ B\ x)\ (whileLoop\ C'\ B'\ x')\ s\ s'\ R$
proof *cases*
assume $*$: $run\ (whileLoop\ C'\ B'\ x')\ s' \neq \top$
show *refines* $(whileLoop\ C\ B\ x)\ (whileLoop\ C'\ B'\ x')\ s\ s'\ R$ **using** *x*
proof (*induction arbitrary*: *x s rule*: *whileLoop-ne-top-induct*[*OF* $*$])
case (*1 a' s' a s*)
show *?case*
apply (*subst* (*1 2*) *whileLoop-unroll*)
apply (*intro refines-condition C 1 refines-yield*)
apply (*rule refines-bind'*)
apply (*rule refines-strengthen2*[*OF B runs-to-partial-of-runs-to*[*OF 1*(*1*)]])
apply (*rule 1*)
apply *simp*
apply *simp*
apply *simp*
apply *auto*
subgoal
apply (*subst whileLoop-unroll*)
apply (*simp add*: *refines-condition-right-iff R-ER*)
done
subgoal
apply (*subst whileLoop-unroll*)
apply (*intro refines-condition-left*)
apply (*simp-all add*: *R-RE*)
done
done
qed
qed (*simp add*: *refines.rep-eq*)

lemma *refines-on-exit'-right*:
assumes *f*: *refines f f' s s'*
$(\lambda(r,\ t)\ (r',\ t').$ *refines skip c' t t'* $(\lambda(q,\ t)\ (q',\ t').\ R$
(*case q of Exception e* $\Rightarrow$ *Exception e* | *Result -* $\Rightarrow$ *r*, *t*)
(*case q' of Exception e'* $\Rightarrow$ *Exception e'* | *Result -* $\Rightarrow$ *r'*, *t'*)))
shows *refines f* (*on-exit' f' c'*) *s s' R*
using *refines-on-exit'*[*of f f' s s' skip c' R*] *f*
by (*simp add*: *on-exit'-skip*)
lemma *runs-to-whileLoop'*:
assumes *B*[*runs-to-vcg*]: $\bigwedge r\ s\ x.\ I\ x\ (Result\ r)\ s \Longrightarrow C\ r\ s \Longrightarrow$
$(B\ r) \bullet s\ \{\!|\ \lambda q\ t.\ \exists y.\ I\ y\ q\ t \wedge (\forall a.\ q = Result\ a \longrightarrow (y,\ x) \in R)\ |\!\}$
assumes *Qr*: $\bigwedge x\ r\ s.\ I\ x\ (Result\ r)\ s \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ (Result\ r)\ s$
assumes *Ql*: $\bigwedge x\ e\ s.\ I\ x\ (Exn\ e)\ s \Longrightarrow Q\ (Exn\ e)\ s$
assumes *R*: *wf R*
assumes *I*: $I\ x\ (Result\ r)\ s$
shows $(whileLoop\ C\ B\ r) \bullet s\ \{\!|\ Q\ |\!\}$

using *I*
proof (*induction x arbitrary*: *r s rule*: *wf-induct-rule*[*OF R*])
case (*1 x*)
note *1*(*1*)[*runs-to-vcg*]

show *?case*
apply (*subst whileLoop-unroll*)
apply *runs-to-vcg*
apply (*rule 1*)
apply *assumption*
apply *assumption*
subgoal by (*rule Ql*)
subgoal using *1*(*2*) **by** (*rule Qr*)
done
qed

lemma *runs-to-whileLoop-variant-exn*:
assumes *I*: $\bigwedge r\ s\ x.\ I\ (Result\ r)\ s\ x \Longrightarrow$
$C\ r\ s \Longrightarrow (B\ r) \cdot s \{\!|\ \lambda q\ t.\ \exists y.\ I\ q\ t\ y \wedge (\forall a.\ q = Result\ a \longrightarrow (y, x) \in R)\ |\!\}$
assumes *Q-Result*: $\bigwedge r\ s\ c.\ I\ (Result\ r)\ s\ c \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ (Result\ r)\ s$
assumes *Q-Exn*: $\bigwedge e\ s\ c.\ I\ (Exn\ e)\ s\ c \Longrightarrow Q\ (Exn\ e)\ s$
assumes *R*: *wf R*
shows *I* (*Result r*) *s x* $\Longrightarrow$ (*whileLoop C B r*::('*e*, '*a*, '*s*) *exn-monad*) $\cdot$ *s* $\{\!|$ *Q* $|\!\}$
using *runs-to-whileLoop'*[*OF assms*] **by** *auto*

lemma *when-when-combine*:
when P (*when Q f*) = *when* ($P \wedge Q$) *f*
by (*rule spec-monad-eqI*) (*auto simp*: *runs-to-iff*)

lemma *whileLoop-False*[*simp*]: *whileLoop* (λ- -. *False*) *B a* = *return a*
apply (*subst whileLoop-unroll*)
apply *simp*
done

lemma *refines-throw-refl*:
refines (*throw i*) (*throw i*) *s-A s-A* (λ(*r-C*, *t-C*) (*r-A*, *t-A*). *r-C* = *r-A* $\wedge$ *t-C* = *s-A* $\wedge$ *t-A* = *s-A*)
by (*auto simp add*: *refines-def*)

lemma *refinesI-runs-to*:
(*f* $\cdot$ *s* $\{\!|$ λ*Res r s'*. *refines* (*return r*) *g s' t R* $|\!\}$) $\Longrightarrow$ *refines f g s t R*
by (*force simp add*: *refines-iff-runs-to intro*: *runs-to-weaken*)

lemma *refinesI-runs-to'*:
(*f* $\cdot$ *s* $\{\!|$ λ*r s'*. *refines* (*yield r*) *g s' t R* $|\!\}$) $\Longrightarrow$ *refines f g s t R*
by (*force simp add*: *refines-iff-runs-to intro*: *runs-to-weaken*)

lemma *refines-selectE-right*:
x $\in$ *X* $\Longrightarrow$ *refines f* (*g x*) *s t R* $\Longrightarrow$ *refines f* (*select X* >>= *g*) *s t R*

by (*fastforce simp*: *refines-iff' runs-to-iff*)

lemma *refines-unknownE-right*:
refines f (*g x*) *s t R* ⟹ *refines f* (*unknown* >>= *g*) *s t R*
unfolding *unknown-def* **by** (*simp add*: *refines-selectE-right*)

lemma *refines-liftE-set-state-right*:
refines f (*g* ()) *s t' R* ⟹ *refines f* (*liftE* (*set-state t'*) >>= *g*) *s t R*
by (*fastforce simp*: *refines-iff' runs-to-iff*)

lemma *refines-modifyE*:
refines (*modify m*) (*modify n*) *s t* (λ*x y*. *x* = (*Result* (), *m s*) ∧ *y* = (*Result* (), *n t*))
apply (*rule refines-modify*)
by *simp*

lemma *refines-modifyE-bindE*:
refines (*f* ()) (*g* ()) (*m s*) (*n t*) *R* ⟹ *refines* (*modify m* >>= *f*) (*modify n* >>= *g*) *s t R*
by (*rule refines-bind*[*OF refines-modifyE*]) *auto*

lemma *refines-bindE-right*:
assumes *f-g*: *refines f g s t* (λ(*Res r*, *s'*) (*r'*, *t'*).
(∀ *x*. *r'* = *Result x* ⟶ *refines* (*return r*) (*h x*) *s' t' R*) ∧
(∀ *e*. *r'* = *Exn e* ⟶ *R* (*Result r*, *s'*) (*Exn e*, *t'*)))
shows *refines f* (*g* >>= *h*) *s t R*
apply (*subst bind-return*[*symmetric*])
apply (*rule refines-bind*[*OF f-g*])
apply (*auto simp add*: *default-option-def Exn-def*)
done

lemma *refines-gets-left-iff*:
refines (*gets r*) *g s t R* ⟷ *refines* (*return* (*r s*)) *g s t R*
by (*simp add*: *refines-def*)

lemma *refines-right*:
assumes *f*: *refines* (*skip*::(*unit*,'*t*) *res-monad*) *f s t R* **and**
g: ⋀*x t*. *R* (*Result* (), *s*) (*Result x*, *t*) ⟹ *refines g* (*g' x*) *s t Q*
shows *refines* (*g*::('*b*, '*t*) *res-monad*) ((*f* ≫= *g'*)::('*a*, '*s*) *res-monad*) *s t Q*
proof −
have ∗: *refines skip f s t* (λ(*x*, *s'*) *y*. *s'* = *s* ∧ *R* (*x*, *s*) *y*)
using *refines-runs-to-partial-fuse*[*OF f runs-to-partial-yield*, **where** *P*=λ- *t*. *t* = *s*]
by (*rule refines-weaken*) *auto*
then have *refines* (*bind* (*skip*::(*unit*, '*t*) *res-monad*) (λ-. *g*)) (*f* >>= *g'*) *s t Q*
apply (*rule refines-bind-res'*)
apply (*simp add*: *g*)
done
then show *?thesis*

by *simp*
qed

lemma *refines-skip-select*:
y ∈ *Y* ⟹
refines skip (*select Y*) *s t* (λ*x* (*y′*, *t′*). *x* = (*Result* (), *s*) ∧ *t′* = *t* ∧ *y′* = *Result y*)
unfolding *select-def*
by *transfer* (*simp add*: *pure-post-state-def image-image Sup-Success*)

lemma *refines-modify-left-iff*:
refines (*modify m*) *g s t R* ⟷ *refines* (*return* ()) *g* (*m s*) *t R*
by (*simp add*: *refines-def*)

lemma *refines-modify-bind-return-bind*:
Q (*m H*) *c* ⟹ (⋀*H′ b*. *Q H′ b* ⟹ *refines* (*g* ()) (*h b*) *H′ A R*) ⟹
refines (*modify m* >>= *g*) (*return c* >>= *h*) *H A R*
apply (*rule refines-bind* [**where** *Q*=λ(*x*, *H′*) (*b′*, *A′*).
(∃ *v*. *x*= *Result v*) ∧ (∃ *v*. *b′*=*Result v* ∧ *Q H′ v* ∧ *A′* = *A*)])
apply (*auto*)
apply (*simp add*: *refines-def*)
done

lemma *refines-return-of-runs-to*:
f · *s* ⦃ *P* ⦄ ⟹ *refines f* (*return x*) *s t* (λ(*a*, *b*) (*c*, *d*). *P a b* ∧ *c* = *Result x* ∧ *d* = *t*)
by (*cases run f s*; *auto simp add*: *refines.rep-eq runs-to.rep-eq*)

lemma *refines-select-iff*:
refines (*select P*) (*select Q*) *s t R* ⟷ (∀ *x*∈*P*. ∃ *xa*∈*Q*. *R* (*Result x*, *s*) (*Result xa*, *t*))
by (*auto simp add*: *refines-def*)

lemma *select-empty-bind*[*simp*]:
select {} >>= *f* = *select* {}
apply (*simp add*: *spec-monad-eq-iff runs-to-iff*)
done

lemma *on-exit-eq*:
on-exit f cleanup =
do {
r <− *catch f* (λ*e*.
do {
- <− *liftE* (*state-select cleanup*);
throw e
});
liftE (*state-select cleanup*);
return r
}

apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff on-exit-def Exn-def elim*!: *runs-to-weaken*)
apply (*metis default-option-def exception-or-result-nchotomy option.exhaust-sel*)+
done

lemma *select-spec-commute*:
assumes *X*: $(X = \{\} \Longrightarrow \forall\, s.\ \exists\, t.\ (s,\ t) \in H)$
shows (*do* { $x \leftarrow$ *select X*; - $\leftarrow$ *state-select H*; *return x* }) =
(*do* { - $\leftarrow$ *state-select H*; *select X* })
using *X*
by (*auto simp add*: *spec-monad-eq-iff runs-to-iff Exn-def default-option-def*)

lemma *refines-skip-right*:
assumes *f*: $f \bullet s \Downarrow \lambda r\ s'.\ R\ (r,\ s')\ (\mathit{Result}\ (),\ t)\Downarrow$ **shows** *refines f skip s t R*
by (*rule refines-mono*[*OF - refines-return-of-runs-to*[*OF f*]]) *simp*

lemma *refines-select-right-iff*:
refines f (*select X*) *s t R* $\longleftrightarrow$ $(f \bullet s \Downarrow \lambda r\ s'.\ \exists\, x{\in}X.\ R\ (r,\ s')\ (\mathit{Result}\ x,\ t)\Downarrow)$
by (*cases run f s*) (*auto simp add*: *refines.rep-eq runs-to.rep-eq*)

lemma *refines-return-right-iff*:
refines f (*return a*) *s t R* $\longleftrightarrow$
$(f \bullet s \Downarrow \lambda r\ s'.\ R\ (r,\ s')\ (\mathit{Result}\ a,\ t)\ \Downarrow)$
by (*cases run f s*) (*simp-all add*: *refines.rep-eq runs-to.rep-eq pure-post-state-def*)

lemma *refines-skip-right-iff*:
refines f skip s t R $\longleftrightarrow$ $(f \bullet s \Downarrow \lambda r\ s'.\ R\ (r,\ s')\ (\mathit{Result}\ (),\ t)\Downarrow)$
unfolding *refines-return-right-iff* **..**

lemma *select-state-assume*:
(*do* {
$x \leftarrow$ *select X*;
assume-result-and-state (*F x*)
}) =
assume-result-and-state ($\lambda s.\ \bigcup x{\in}X.\ F\ x\ s$)
apply (*auto simp add*: *spec-monad-eq-iff runs-to-iff*)
done

lemma *getsE-fold-into-modifyE*:
((*do* { $x \leftarrow$ *gets f*; *modify* (*g x*) }) :: ('*e*, *unit*,'*s*) *exn-monad*) =
modify ($\lambda s.$ *g* (*f s*) *s*)
((*do* { $x \leftarrow$ *gets f*; - $\leftarrow$ *modify* (*g x*); *h* }) :: ('*e*, '*a*, '*s*) *exn-monad*) =
do { *modify* ($\lambda s.$ *g* (*f s*) *s*); *h* }
by (*auto simp add*: *spec-monad-eq-iff runs-to-iff*)+

lemma *refines-whileLoop-right*:
assumes *I*: *R f I s t*
assumes *final*: $\bigwedge f\ x\ s\ t.\ R\ f\ x\ s\ t \Longrightarrow \neg\ C\ x\ t \Longrightarrow f \bullet s \Downarrow \lambda y\ s'.\ P\ (y,\ s')\ (\mathit{Result}$

```
x, t) ⦘
  assumes R-step: ⋀f x s t. R f x s t ⟹ C x t ⟹
    ∃(g::('a, 's) res-monad) f'. run f s = run (bind g f') s ∧
      refines g (B x) s t (λ(Res a, s) (Res b, t). R (f' a) b s t)
  shows refines f (whileLoop C B I) s t P
proof cases
  assume run (whileLoop C B I) t ≠ ⊤ from this I show ?thesis
  proof (induction arbitrary: f s rule: whileLoop-ne-top-induct)
    case (step I t f s)
    have *: refines f (B I >>= whileLoop C B) s t P if C I t
    proof −
      obtain g::('a, 's) res-monad and f' where eq: run f s = run (bind g f') s
        and g: refines g (B I) s t (λ(Res a, s) (Res b, t). R (f' a) b s t)
        using R-step[OF step.prems ‹C I t›] by blast
      have refines (g >>= f') (B I >>= whileLoop C B) s t P
       by (rule refines-bind[OF refines-ext-left[OF step.IH[OF ‹C I t›] g]]) simp-all
      with eq show ?thesis
        by (simp add: refines.rep-eq)
    qed
    show ?case
      by (subst whileLoop-unroll)
        (simp add: refines-condition-right-iff refines-return-right-iff final step.prems
*)
  qed
qed (simp add: refines.rep-eq)

lemma gets-condition:
  gets r >>= (λx. condition (P x) f g) = condition (λs. P (r s) s) f g
  by (auto simp add: spec-monad-eq-iff runs-to-iff)

lemma condition-select-empty:
  condition P (select {}) f =
  do {
    b ← gets P;
    assume (¬ b);
    f
  }
  by (auto simp add: spec-monad-eq-iff runs-to-iff)

lemma not-is-Result-Exn[simp]: ¬ is-Result (Exn e)
  by (simp add: Exn-def default-option-def)

lemma map-value-map-lift-result-return:
  map-value (map-exception-or-result E (λx. x)) (return r) = return r
  by (auto simp add: spec-monad-eq-iff runs-to-iff)

lemma refines-throw-throw:
  refines (throw e) (throw e') s t R = R (Exn e, s) (Exn e', t)
  by (simp add: refines-yield-right-iff)
```

lemma *refines-bind-bind-exn'*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Exn\ e',\ t') \Longrightarrow$
$e \neq default \Longrightarrow$
$R\ (Exception\ e,\ t)\ (Exn\ e',\ t')$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Result\ v',\ t') \Longrightarrow$
$e \neq default \Longrightarrow$
$refines\ (yield\ (Exception\ e))\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e,\ t') \Longrightarrow$
$refines\ (g\ v)\ (throw\ e)\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
$refines\ (g\ v)\ (g'\ v')\ t\ t'\ R$
shows *refines* (*bind f g*) (*bind f' g'*) *s s' R*
using *assms*
by (*auto intro*!: *refines-bind*[*OF f*] *simp*: *default-option-def Exn-def*)
lemma *guard-bind-cong*:
$g = g' \Longrightarrow (\bigwedge s.\ g'\ s \Longrightarrow run\ c\ s = run\ c'\ s) \Longrightarrow$
do {- <− *guard g*;
c
} =
do {- <− *guard g'*;
c'
}
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *modify-guard-swap*:
$(\bigwedge s.\ P\ (f\ s) = P\ s) \Longrightarrow$
do {
- <− *modify f*;
guard P
} =
do {
- <− *guard P*;
modify f
}
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *modify-guard-bind-swap*:
$(\bigwedge s.\ P\ (f\ s) = P\ s) \Longrightarrow$
do {
- <− *modify f*;
- <− *guard P*;
X
} =

```
  do {
    - <- guard P;
    - <- modify f;
    X
  }
  apply (rule spec-monad-ext)
  apply (simp add: run-bind run-guard)
  done

lemma when-throw-cong: P = P' ⟹ (⋀s. P' ⟹ run Y s = run Y' s) ⟹
  do { - <- when (¬ P) (throw x); Y } =
  do {- <- when (¬ P') (throw x); Y' }
  apply (rule spec-monad-ext)
  apply (auto simp add: run-bind run-when)
  done

lemma condition-throw:
  condition (λs. P) (throw e) B = do { when P (throw e); B }
  apply (auto simp add: spec-monad-eq-iff runs-to-iff Exn-def default-option-def)
  done

lemma map-value-map-lift-result-bind[simp]:
  map-value (map-exception-or-result (λx::unit. undefined) f) (bind m g) =
   bind m (λx. map-value (map-exception-or-result (λx. undefined) f) (g x))
  by (simp add: spec-monad-eq-iff runs-to-iff)

lemma map-value-map-result-bind[simp]:
  map-value (map-exception-or-result id f) (bind m g) =
   bind m (λx. map-value (map-exception-or-result id f) (g x))
  apply (auto simp add: spec-monad-eq-iff runs-to-iff)
  done

lemma map-value-yield[simp]: map-value f (yield x) = yield (f x)
  apply (rule spec-monad-ext)
  apply (simp add: run-map-value)
  done

lemma map-result-return[simp]: map-value (map-exception-or-result id f) (return
x) = return (f x)
  apply (rule spec-monad-ext)
  apply (simp add: run-map-value)
  done

lemma map-result-throw[simp]: map-value (map-exception-or-result id f) (throw
x) = throw x
  apply (rule spec-monad-ext)
  apply (simp add: run-map-value)
  done
```

lemma *map-value-compose*[*simp*]: *map-value f* (*map-value g m*) = (*map-value* (*f* ∘ *g*) *m*)
apply (*auto simp add*: *runs-to-iff spec-monad-eq-iff*)
done

lemma *map-result-compose*[*simp*]:
map-exception-or-result id f o map-exception-or-result id g = map-exception-or-result id (*f o g*)
by (*simp add*: *map-exception-or-result-comp fun-eq-iff*)

lemma *map-value-condition*:
map-value f (*condition c g h*) = *condition c* (*map-value f g*) (*map-value f h*)
apply (*auto simp add*: *runs-to-iff spec-monad-eq-iff*)
done

lemma *whenE-throwError-cong*:
P = *P'* ⟹ (⋀*s*. *P'* ⟹ *run Y s* = *run Y' s*) ⟹
do { - <− *when* (¬ *P*) (*throw x*); *Y* } =
do { - <− *when* (¬ *P'*) (*throw x*); *Y'* }
by (*rule when-throw-cong*)

lemma *modifyE-guardE-bindE-swap*:
(⋀*s*. *P* (*f s*) = *P s*) ⟹
do {
- <− *modify f*;
- <− *guard P*;
X
} =
do {
- <− *guard P*;
- <− *modify f*;
X
}
by (*rule modify-guard-bind-swap*)

lemma *condition-throwError*:
condition (λ*s*. *P*) (*throw e*) *B* = *do* { *when P* (*throw e*); *B* }
by (*rule condition-throw*)

lemma *refines-bind-condition-check*:
assumes *f*: *f* • *s* ⦃ λ*Res r s'*. *s'* = *s* ∧
(*P r s* ⟷ (∀ *a*. ¬ *Q t a*)) ∧
(*P r s* ⟶ (∀ *a*. ¬ *Q t a*) ⟶ *R* (*Result e*, *s*) (*Exn q*, *t*)) ⦄
assumes *g-h*: ⋀*x y*. ¬ *P x s* ⟹ *Q t y* ⟹ *refines* (*g x*) (*h y*) *s t R*
shows *refines*
(*do* { *x* ← *f*; *condition* (*P x*) (*return e*) (*g x*) })
(*do* { *x* ← *check q Q*; *h x* }) *s t R*
apply (*rule refines-bind-left-res*)
apply (*rule f*[*THEN runs-to-weaken*])

apply *clarsimp*
apply (*rule refines-condition-neg-check*)
apply (*simp-all add*: *g-h*)
done

lemma *refinesI-runs-to-vcg*:
$f \cdot s \{\!|\ P\ |\!\} \Longrightarrow (\bigwedge r\ t.\ P\ r\ t \Longrightarrow Q\ (r,\ t)\ (r,\ t)) \Longrightarrow$ *refines f f s s Q*
by (*cases run f s*; *force simp*: *runs-to.rep-eq refines.rep-eq*)

lemma *refines-gets-the-right-fixed*:
assumes *m*:
(*gets-the m* :: (*'a*::*default*, *'b*, *'c*) *spec-monad*) $\cdot$ *t* $\{\!|$ $\lambda r\ t.\ t = t' \wedge r =$ *Result r'* $|\!\}$
shows *refines f* (*h r'*) *s t' Q* $\Longrightarrow$
refines f (*do* {*x* $\leftarrow$ *gets-the m*; *h x* } :: (*'a*::*default*, *'b*, *'c*) *spec-monad*) *s t Q*
apply (*rule refines-bind-right-runs-to-partialI*[*OF - always-progress-gets-the*])
apply (*rule runs-to-partial-of-runs-to*)
apply (*rule runs-to-weaken*[*OF m*])
apply *simp*
done

lemma *refines-gets-the*:
$(\bigwedge b.\ n\ t =$ *Some b* $\Longrightarrow \exists a.\ m\ s =$ *Some a* $\wedge$ *Q* (*Result a*, *s*) (*Result b*, *t*)) $\Longrightarrow$
refines (*gets-the m*) (*gets-the n*) *s t Q*
by (*auto simp add*: *refines-iff-runs-to runs-to-gets-the-iff*)

lemma *refines-bind-gets-the*:
$(\bigwedge x.$ *refines* (*f x*) (*g x*) *s s Q*) $\Longrightarrow$
refines (*do* { *x* $\leftarrow$ *gets-the m*; *f x* }) (*do* { *x* $\leftarrow$ *gets-the m*; *g x* }) *s s Q*
by (*intro refines-bind*[*OF refines-gets-the*[*of - - - -*
$\lambda(a,\ x)\ (b,\ y).\ \exists v.\ a =$ *Result v* $\wedge$ *b* = *Result v* $\wedge$ *x* = *s* $\wedge$ *y* = *s*]])
simp-all

lemma *refines-bind-left'*:
refines f f' s t (λ(*Res r*, *s'*) (*Res q*, *t'*). *refines* (*g r*) (*return q*) *s' t' Q*) $\Longrightarrow$
refines (*do* { *x* $\leftarrow$ *f*; *g x* }) *f' s t Q*
using *refines-bind-res*[*of f f' s t g return Q*] **by** *simp*

lemma *refines-check-eq*:
refines (*check i p*) (*check i p*) *s-A s-A*
(λ(*r-C*, *t-C*) (*r-A*, *t-A*). *r-C* = *r-A* $\wedge$ *t-C* = *s-A* $\wedge$ *t-A* = *s-A*)
unfolding *check-def*
apply (*intro refines-condition refl*
refines-bind[**where** *Q*=(λ(*r-C*, *t-C*) (*r-A*, *t-A*). *r-C* = *r-A* $\wedge$ *t-C* = *s-A* $\wedge$ *t-A* = *s-A*)])
apply (*auto intro*: *refines-get-state refines-select refines-throw-refl*)
done

lemma *runs-to-check'*[*runs-to-vcg*]:

$(\bigwedge x.\ p\ s\ x \Longrightarrow Q\ (Result\ x)\ s) \Longrightarrow ((\bigwedge x.\ \neg\ p\ s\ x) \Longrightarrow Q\ (Exn\ e)\ s) \Longrightarrow (check\ e\ p) \cdot s\ \{\!|Q|\!\}$
by (*simp add*: *runs-to-check-iff*)

lemma *refines-when-check*:
$P = (\forall a.\ \neg\ Q\ t\ a) \Longrightarrow$
$(P \Longrightarrow \forall a.\ \neg\ Q\ t\ a \Longrightarrow R\ (Exn\ r,\ s)\ (Exn\ q,\ t)) \Longrightarrow$
$(\bigwedge a.\ \neg\ P \Longrightarrow Q\ t\ a \Longrightarrow refines\ (f\ ())\ (g\ a)\ s\ t\ R) \Longrightarrow$
$refines\ (bind\ (when\ P\ (throw\ r))\ f)\ (bind\ (check\ q\ Q)\ g)\ s\ t\ R$
apply (*rule refines-bind-left-exn*)
apply *runs-to-vcg*
apply (*auto intro*: *refines-check-right-ok refines-throwError-check*)
done

lemma *refines-bind-gets-right*:
$refines\ f\ (bind\ (gets\ g)\ h)\ s\ t\ R = refines\ f\ (h\ (g\ t))\ s\ t\ R$
by (*simp add*: *refines-iff' runs-to-iff*)

lemma *modify-id-return*:
$modify\ id = return\ ()$
by (*simp add*: *spec-monad-eq-iff runs-to-iff*)

lemma *refines-bind-exn-bind-exn*:
assumes *f*: $refines\ f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Exn\ e',\ t') \Longrightarrow R\ (Exn\ e,\ t)\ (Exn\ e',\ t')$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Result\ v',\ t') \Longrightarrow refines\ (throw\ e)\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e,\ t') \Longrightarrow refines\ (g\ v)\ (throw\ e)\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow refines\ (g\ v)\ (g'\ v')\ t\ t'\ R$
shows $refines\ (bind\ f\ g)\ (bind\ f'\ g')\ s\ s'\ R$
using *assms*
by (*auto intro*!: *refines-bind*[*OF f*] *simp*: *default-option-def Exn-def*)

lemma *refines-finally-left*:
assumes $*$:
$refines\ f\ g\ s\ t\ (\lambda(r\text{-}C,\ t\text{-}C)\ A.\ \forall x.\ r\text{-}C = Exn\ x \vee r\text{-}C = Result\ x \longrightarrow R\ (Result\ x,\ t\text{-}C)\ A)$
shows $refines\ (finally\ f)\ g\ s\ t\ R$
unfolding *finally-def refines-map-value-left-iff*
by (*rule refines-weaken*[*OF* $*$]) (*simp add*: *unite-def split*: *xval-split*)

lemma *refines-map-value-left*:
$refines\ f\ (map\text{-}value\ m\ g)\ s\ t\ R = refines\ f\ g\ s\ t\ (\lambda A\ (r,\ u).\ R\ A\ (m\ r,\ u))$
unfolding *refines-map-value-right-iff*
by (*intro arg-cong*[**where** *f*=*refines - - - -*] *ext*) *auto*

lemma *refines-map-value-right*:
refines (*map-value m f*) *g s t R* = *refines f g s t* (λ(*r*, *u*). *R* (*m r*, *u*))
unfolding *refines-map-value-left-iff*
by (*intro arg-cong*[**where** *f*=*refines - - - -*] *ext*) *auto*

lemma *refines-when'*:
(*P* ⟹ *refines f g s t R*) ⟹ (¬*P* ⟶ *R* (*Result* (), *s*) (*Result* (), *t*)) ⟹
refines (*when P f*) (*when P g*) *s t R*
by (*simp add*: *refines-iff' runs-to-iff*)

lemma *top-eq-fail*: *top* = *fail*
by *transfer*
(*auto simp*: *top-post-state-def*)

lemma *refines-bind-get-state*:
refines (*f s*) (*g t*) *s t R* ⟹ *refines* (*get-state* ⤜ *f*) (*get-state* ⤜ *g*) *s t R*
by (*simp add*: *refines-iff' runs-to-iff*)

lemma *refines-bind-assert*:
refines (*assert P* ⤜ *f*) (*assert Q* ⤜ *g*) *s t R*
if ∗: *Q* ⟹ *P Q* ⟹ *P* ⟹ *refines* (*f* ()) (*g* ()) *s t R*
using ∗ **by** (*simp add*: *refines-iff' runs-to-iff*)

lemma *refines-bind-set-state*: *refines* (*set-state s'* ⤜ *f*) (*set-state t'* ⤜ *g*) *s t R*
if ∗: *refines* (*f* ()) (*g* ()) *s' t' R*
using ∗ **by** (*simp add*: *refines-iff' runs-to-iff*)

lemma *refines-bind-select*: *refines* (*select P* ⤜ *f*) (*select Q* ⤜ *g*) *s t R*
if ∗: *sim-set S P Q* ⋀*v v'*. *S v v'* ⟹ *refines* (*f v*) (*g v'*) *s t R*
using ∗ **by** (*auto simp add*: *refines-iff' runs-to-iff sim-set-def*)

lemma *refines-assert-result-and-state'*:
refines (*assert-result-and-state f*) (*assert-result-and-state g*) *s t*
(*rel-prod* (*rel-exception-or-result E R*) *S*)
if ∗: *S s t sim-set* (*rel-prod R S*) (*f s*) (*g t*) *f s* = {} ⟹ *g t* = {}
using ∗
by (*intro refines-assert-result-and-state*; *force simp*: *sim-set-def rel-prod.simps*)

lemma *runs-to-partial-assert-result-and-state-iff*:
assert-result-and-state f · *s* ?{⦇ *P* ⦈} ⟷ (∀ (*a*, *s*)∈*f s*. *P* (*Result a*) *s*)
apply *safe*
subgoal
by (*auto simp*: *runs-to-partial-def run-assert-result-and-state split*: *if-splits*)
subgoal by *runs-to-vcg auto*
done

lemma *bind-state-assume-eq*: *run* (*do* { (-::*unit*) <− *assume-result-and-state P*; *f*
}) *s* = *run f s*
if ⋀*s s'*. ((), *s'*) ∈ *P s* ⟹ *s'* = *s* ((), *s*) ∈ *P s*

using *that* **by** (*auto simp add*: *runs-to-iff run-runs-to-extensionality*)

lemma *refines-bindE-right'*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Exn\ e',\ t') \Longrightarrow R\ (Exn\ e,\ t)\ (Exn\ e',\ t')$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Result\ v',\ t') \Longrightarrow$ *refines* (*throw e*) (g' v') $t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e',\ t') \Longrightarrow$ *refines* ((*return v*)) (*throw* e') $t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$ *refines* ((*return v*)) (g' v') $t\ t'\ R$
shows *refines f* (f' >>= g') *s s' R*
proof −
have *eq*: $f = (f >>= (\lambda v.\ return\ v))$
by *simp*
show *?thesis*
apply (*subst eq*)
using *assms*
by (*rule refines-bind-bind-exn*)
qed

lemma *refines-exec-modify-step-right*:
assumes *refines* (*return x*) *g s* (*upd t*) *Q*
shows *refines* (*return x*) (*do* { - <− (*modify* (*upd*)); *g* }) *s t Q*
using *assms*
by (*auto simp add*: *refines-iff' runs-to-iff*)

lemma *runs-to-whileLoop-variant*:
assumes *I*: $\bigwedge r\ s\ c.\ I\ r\ s\ c \Longrightarrow$
$C\ r\ s \Longrightarrow (B\ r) \cdot s \{\!|\ \lambda Res\ q\ t.\ \exists c'.\ I\ q\ t\ c' \wedge (c',\ c) \in R\ |\!\}$
assumes *Q*: $\bigwedge r\ s\ c.\ I\ r\ s\ c \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ r\ s$
assumes *R*: *wf R*
shows $I\ r\ s\ c \Longrightarrow$ (*whileLoop C B r*) $\cdot\ s \{\!|\ \lambda Res\ r\ s.\ Q\ r\ s\ |\!\}$
proof (*induction arbitrary*: *r s rule*: *wf-induct*[*OF R*])
case (*1 c*)
note *1*(*1*)[*rule-format*, *runs-to-vcg*]
note *I*[*OF 1*(*2*), *runs-to-vcg*]
show *?case*
apply (*subst whileLoop-unroll*)
apply *runs-to-vcg*
apply *assumption*+
apply (*rule Q*[*OF 1*(*2*)])
apply *assumption*+
done
qed

lemma *catch-throw*[*simp*]: *catch f throw* = *f*
apply (*rule spec-monad-eqI*)
apply (*simp add*: *runs-to-iff*)

apply (*rule arg-cong*[**where** *f*=*runs-to* - -])
apply (*intro ext*)
subgoal for *P s x t*
by (*cases x*) (*auto simp add*: *default-option-def simp flip*: *Exn-def*)
done

lemma *select-singleton*: *select* {*x*} = *return x*
by (*simp add*: *select-def*)

lemma *gets-gets-combine*:
(*do* { *x* ← *gets f*; *gets* (*g x*) }) = *gets* (λ*h*. *g* (*f h*) *h*)
by (*intro spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *gets-gets*:
(*do* { *x* ← *gets f*; *y* ← *gets* (*g x*); *C y* }) =
(*do* { *y* ← *gets* (λ*h*. *g* (*f h*) *h*); *C y* })
by (*simp add*: *gets-gets-combine flip*: *bind-assoc*)

lemma *refines-whileLoop-exn-strong*:
assumes *C*: ⋀*v s v′ s′*.
R (*v*, *s*) (*v′*, *s′*) ⟹
(∃ *x*. *v* = *Result x* ∧ *C x s*) ⟷ (∃ *x′*. *v′* = *Result x′* ∧ *C′ x′ s′*)
assumes *B*: ⋀*x s x′ s′*. *R* (*Result x*, *s*) (*Result x′*, *s′*) ⟹ *C x s* ⟹ *C′ x′ s′* ⟹
refines (*B x*) (*B′ x′*) *s s′ R*
assumes *Q*: ⋀*x s x′ s′*. *R* (*x*, *s*) (*x′*, *s′*) ⟹
(⋀*r*. *x* = *Result r* ⟹ ¬ *C r s*) ⟹ (⋀*r′*. *x′* = *Result r′* ⟹ ¬ *C′ r′ s′*) ⟹
Q (*x*, *s*) (*x′*, *s′*)
assumes *x*: *R* (*Result x*, *s*) (*Result x′*, *s′*)
shows *refines* (*whileLoop C B x*) (*whileLoop C′ B′ x′*) *s s′ Q*
proof *cases*
assume ∗: *run* (*whileLoop C′ B′ x′*) *s′* ≠ ⊤
show *refines* (*whileLoop C B x*) (*whileLoop C′ B′ x′*) *s s′ Q* **using** *x*
proof (*induction arbitrary*: *x s rule*: *whileLoop-ne-top-induct*[*OF* ∗])
case (*1 a′ s′ a s*)
show *?case*
apply (*subst* (*1 2*) *whileLoop-unroll*)
apply (*intro refines-condition 1 refines-yield*)
subgoal
using *C*[*OF 1*(*2*)] **by** *simp*
apply (*rule refines-bind′*)
apply (*rule refines-strengthen2*[*OF B runs-to-partial-of-runs-to*[*OF 1*(*1*)]])
apply (*rule 1*)
subgoal by *simp*
subgoal by *simp*
subgoal by *simp*
subgoal for *x t x′ t′*
using *Q*[*of x t x′ t′*] *C*[*of x t x′ t′*]
apply *auto*
subgoal by (*subst whileLoop-unroll*) (*simp add*: *refines-condition-right-iff*)

```
      subgoal by (subst whileLoop-unroll) (auto intro!: refines-condition-left)
      done
    subgoal using C[OF 1(2)] Q[OF 1(2)] by auto
    done
  qed
qed (simp add: refines.rep-eq)

lemma not-default-Some:
  x ≠ default ⟷ (∃ a. x = Some a)
  by (auto simp: default-option-def)

lemma gets-fold-into-modify:
  do { x ← gets f; - ← modify (g x); h } = do { modify (λs. g (f s) s); h }
  do { x ← gets f; modify (g x) } = modify (λs. g (f s) s)
   by (rule spec-monad-eqI; auto simp add: runs-to-iff)+

lemma assume-eq-select: assume p = select {x. p}
  by (rule spec-monad-eqI) (simp add: runs-to-select-iff)

lemma condition-bot:
  condition P ⊥ f =
  do {
    b ← gets P;
    assume (¬ b);
    f
  }
  by (rule spec-monad-eqI; auto simp add: runs-to-iff)

lemma unless-not: unless (¬ P) = when P
  by simp

lemma select-bind-select:
  select X >>= (λx. select (Y x)) = select {y. ∃ x∈X. y∈Y x}
  by (intro spec-monad-eqI) (auto simp add: runs-to-iff)

lemma case-exception-or-result-iff:
  case-exception-or-result P Q r ⟷
    (∀ x. r = Result x ⟶ Q x) ∧ (∀ e. r = Exception e ⟶ e ≠ default ⟶ P e)
  by (cases r) auto

lemma case-exception-or-result-lam-iff:
  case-exception-or-result P Q r =
    (λs. (∀ x. r = Result x ⟶ Q x s) ∧ (∀ e. r = Exception e ⟶ e ≠ default ⟶
P e s))
  by (cases r) auto

lemma runs-to-cong-split:
  (∀ a s. P (Result a) s ⟷ Q (Result a) s) ⟹
    (∀ e s. e ≠ default ⟶ P (Exception e) s ⟷ Q (Exception e) s) ⟹
```

$f \cdot s \{\!|\ P\ |\!\} \longleftrightarrow f \cdot s \{\!|\ Q\ |\!\}$
apply (*intro arg-cong*[**where** $f{=}\lambda P.\ f \cdot s \{\!|\ P\ |\!\}$] *ext*)
subgoal for *r t* **by** (*cases r*; *simp*)
done

lemma *left-total-rel-map*: *left-total* (*rel-map f*)
by (*simp add*: *left-total-def rel-map-def*)

lemma *rel-exception-or-result-Exception-right*:
$e \neq default \Longrightarrow$
rel-exception-or-result E A x (*Exception e*) $\longleftrightarrow$
$(\exists\, e'.\ e' \neq default \wedge x = Exception\ e' \wedge E\ e'\ e)$
by (*auto simp*: *rel-exception-or-result.simps*)

lemma *rel-exception-or-result-Exception-left*:
$e \neq default \Longrightarrow$
rel-exception-or-result E A (*Exception e*) *y* $\longleftrightarrow$
$(\exists\, e'.\ e' \neq default \wedge y = Exception\ e' \wedge E\ e\ e')$
by (*auto simp*: *rel-exception-or-result.simps*)

lemma *rel-exception-or-result-Result-right*:
rel-exception-or-result E A x (*Result a*) $\longleftrightarrow$ $(\exists\, a'.\ x = Result\ a' \wedge A\ a'\ a)$
by (*auto simp*: *rel-exception-or-result.simps*)

lemma *rel-exception-or-result-Result-left*:
rel-exception-or-result E A (*Result a*) *x* $\longleftrightarrow$ $(\exists\, a'.\ x = Result\ a' \wedge A\ a\ a')$
by (*auto simp*: *rel-exception-or-result.simps*)

lemma *right-unique-rel-exception-of-regular*:
right-unique E $\Longrightarrow$ *right-unique F* $\Longrightarrow$ *right-unique* (*rel-exception-or-result E F*)
by (*auto simp*: *right-unique-def rel-exception-or-result.simps*)

lemma *left-total-rel-exception-of-regular*:
$(\bigwedge x.\ x \neq default \Longrightarrow \exists\, y.\ y \neq default \wedge E\ x\ y) \Longrightarrow$ *left-total F* $\Longrightarrow$
left-total (*rel-exception-or-result E F*)
apply (*auto simp*: *left-total-def*)
subgoal for *x*
apply (*cases x*)
apply (*simp-all add*:
rel-exception-or-result-Result-left rel-exception-or-result-Exception-left)
done
done

lemma *left-unique-rel-map*: *inj f* $\Longrightarrow$ *left-unique* (*rel-map f*)
by (*simp add*: *left-unique-def rel-map-def inj-def*)

lemma *rel-exception-or-result-sym*:
rel-exception-or-result E^{-1-1} F^{-1-1} *s t* $\Longrightarrow$ *rel-exception-or-result E F t s*
by (*auto elim*!: *rel-exception-or-result.cases intro*:*rel-exception-or-result.intros*)

lemma *rel-post-state-inv*: *rel-post-state* R^{-1-1} *t s* $\longleftrightarrow$ *rel-post-state R s t*
by (*auto simp add*: *rel-post-state.simps*)

lemma *rel-spec-inv*: *rel-spec B A t s* R^{-1-1} $\longleftrightarrow$ *rel-spec A B s t R*
by (*simp add*: *rel-spec-def rel-post-state-inv*)
lemma *refines-gets'-right*:
refines f (*do* { *x* ← *gets g*; *h x* }) *s t R* $\longleftrightarrow$ *refines f* (*h* (*g t*)) *s t R*
by (*simp add*: *refines.rep-eq run-bind*)

lemma *sim-post-state-Success-right'*:
sim-post-state R f (*Success s*) $\longleftrightarrow$ *f* ≤ *Success* {*x*. ∃ *y*∈*s*. *R x y*}
by (*cases f*) *auto*

lemma *Sup-eq-top-post-state*[*simp*]:
Sup (*X* :: *'a post-state set*) = ⊤ $\longleftrightarrow$ ⊤ ∈ *X*
by (*simp add*: *top-post-state-def Sup-post-state-def*)

lemma *sim-post-state-Success-right*:
sim-post-state R x (*Success t*) $\longleftrightarrow$ (∃ *s*. *x* = *Success s* ∧ (∀ *a*∈*s*. ∃ *b*∈*t*. *R a b*))
by (*cases x*) *auto*

lemma *sim-post-state-Success-left*:
sim-post-state R (*Success s*) *y* $\longleftrightarrow$ (∀ *t*. *y* = *Success t* ⟶ (∀ *a*∈*s*. ∃ *b*∈*t*. *R a b*))
by (*cases y*) *auto*

lemma *sim-post-state-bind-Success-right*:
sim-post-state R f (*bind-post-state g* (λ*x*. *Success* (*p x*))) $\longleftrightarrow$
sim-post-state (λ*a b*. ∃ *x*∈*p b*. *R a x*) *f g*
by (*cases g*; *simp add*: *sim-post-state-Success-right Sup-Success*)

lemma *sim-post-state-bind-Success-left*:
sim-post-state (λ*x y*. ∀ *a*∈*p x*. *R a y*) *f g* ⟹
sim-post-state R (*bind-post-state f* (λ*x*. *Success* (*p x*))) *g*
by (*cases f*; *cases g*; *auto simp add*: *Sup-Success*)

lemma *rel-post-state-Failure-left*[*simp*]:
rel-post-state R Failure y $\longleftrightarrow$ *y* = *Failure*
by (*auto simp*: *rel-post-state.simps*)

lemma *rel-post-state-Failure-right*[*simp*]:
rel-post-state R x Failure $\longleftrightarrow$ *x* = *Failure*
by (*auto simp*: *rel-post-state.simps*)

lemma *rel-post-state-Success-right*:
rel-post-state R x (*Success t*) $\longleftrightarrow$ (∃ *s*. *x* = *Success s* ∧ *rel-set R s t*)
by (*auto simp*: *rel-post-state.simps*)

lemma *rel-post-state-Success-left*:

rel-post-state R (Success s) y $\longleftrightarrow$ $(\exists\, t.\ y = Success\ t \wedge$ *rel-set R s t*$)$
by (*auto simp*: *rel-post-state.simps*)

lemma *rel-post-state-bind-Success-left*:
assumes $*$: $\bigwedge a\ b.$ *rel-set R* $(\bigcup\ (p\ `\ a))\ b =$ *rel-set Q a b*
shows *rel-post-state R* (*bind-post-state f* $(\lambda x.$ *Success* $(p\ x)))\ g \longleftrightarrow$ *rel-post-state Q f g*
by (*cases f*; *cases g*; *simp add*: $*$ *Sup-Success*)

lemma *sim-post-state-mono*:
sim-post-state R f g $\Longrightarrow$ $(\bigwedge x\ y.$ *R x y* $\Longrightarrow$ *Q x y*$)$ $\Longrightarrow$ *sim-post-state Q f g*
by (*force elim*!: *sim-post-state.cases intro*!: *sim-post-state.intros simp*: *sim-set-def*)

lemma *rel-set-iff-eq*:
left-total R $\Longrightarrow$ *right-unique R* $\Longrightarrow$ *rel-set R a b* $\longleftrightarrow$ $b = (\bigcup x{\in}a.\ \{y.\ R\ x\ y\})$
by (*fastforce simp*: *rel-set-def right-unique-def left-total-def*)

lemma *left-total-rel-exception-or-result*:
left-total A $\Longrightarrow$ $(\forall\, x.\ \exists\, y.\ y \neq default \wedge E\ x\ y) \Longrightarrow$ *left-total* (*rel-exception-or-result E A*)
apply (*clarsimp simp*: *left-total-def*)
subgoal for *x*
apply (*cases x*)
apply (*force intro*: *rel-exception-or-result.intros*)+
done
done

lemma *bind-return-eq-map-value*:
(*do* { *x* $\leftarrow$ *f*; *return* (*F x*) }) = *map-value* (*map-exception-or-result id F*) *f*
by (*auto simp add*: *spec-monad-eq-iff runs-to-iff map-exception-or-result-def fun-eq-iff*
intro!: *runs-to-cong-pred-only split*: *exception-or-result-split*)

lemma *refines-bind-return-iff*:
fixes *f* :: (-, -) *res-monad* **and** *g* :: (-, -) *res-monad*
shows *refines* (*do* { *x* $\leftarrow$ *f*; *return* (*F x*) }) (*do* { *x* $\leftarrow$ *g*; *return* (*G x*) }) *s t R* $\longleftrightarrow$
refines f g s t $(\lambda(Res\ r,\ s')\ (Res\ q,\ t').$ *R* (*Result* (*F r*), *s*$'$) (*Result* (*G q*), *t*$'$))
unfolding *bind-return-eq-map-value refines-map-value-right-iff refines-map-value-left-iff*
by (*intro arg-cong*[**where** *f*=*refines f g s t*]) (*auto simp*: *fun-eq-iff*)

lemma *refines-bind-return-left-iff*:
fixes *g* :: (-, -) *res-monad*
shows *refines f* (*do* { *x* $\leftarrow$ *g*; *return* (*G x*) }) *s t R* $\longleftrightarrow$
refines f g s t $(\lambda x\ (Res\ q,\ t').$ *R x* (*Result* (*G q*), *t*$'$))
unfolding *bind-return-eq-map-value refines-map-value-right-iff refines-map-value-left-iff*
by (*intro arg-cong*[**where** *f*=*refines f g s t*]) (*auto simp*: *fun-eq-iff*)

lemma *refines-bind-return-right-iff*:
fixes *f* :: (-, -) *res-monad*

shows *refines* (*do* { $x \leftarrow f$; *return* (*G x*) }) *g s t R* $\longleftrightarrow$
refines f g s t (λ(*Res r*, *s'*) *y*. *R* (*Result* (*G r*), *s'*) *y*)
unfolding *bind-return-eq-map-value refines-map-value-right-iff refines-map-value-left-iff*
by (*intro arg-cong*[**where** *f*=*refines f g s t*]) (*auto simp*: *fun-eq-iff*)

lemma *refines-runs-to-strenghten*:
refines f g s t R $\Longrightarrow$ *g* · *t* {‖ *P* ‖} $\Longrightarrow$ ($\bigwedge$*a x t'*. *R a* (*x*, *t'*) $\Longrightarrow$ *P x t'* $\Longrightarrow$ *Q a* (*x*, *t'*)) $\Longrightarrow$
refines f g s t Q
apply (*rule refines-strengthen2*[*OF* - *runs-to-partial-of-runs-to*])
apply *assumption*
apply *assumption*
apply *auto*
done

lemma *refines-liftE-right*:
refines f (*liftE g*) *s t R* $\longleftrightarrow$ *refines f g s t* (λ*a* (*Res b*, *s*). *R a* (*Result b*, *s*))
unfolding *liftE-def refines-map-value-right-iff*
by (*intro arg-cong*[**where** *f*=*refines f g s t*]) (*auto simp*: *fun-eq-iff*)

lemma *refines-return-iff*:
refines (*return x*) (*return y*) *s t R* $\longleftrightarrow$ *R* (*Result x*, *s*) (*Result y*, *t*)
by (*simp add*: *refines.rep-eq*)

lemma *on-exit'-res-monad*:
fixes *f* :: ('*a*, '*s*) *res-monad*
shows *on-exit' f c* = *do* { *r* ← *f*; *c*; *return r* }
by (*auto simp add*: *spec-monad-eq-iff runs-to-iff intro*!: *runs-to-cong-pred-only*)

lemma *ignoreE-case-prod*[*ignoreE-simps*]:
ignoreE (*case-prod f p*) = (*case p of* (*a*, *b*) ⇒ *ignoreE* (*f a b*))
by (*cases p*) *simp*

lemma *ignoreE-bind-handle*[*ignoreE-simps*]:
ignoreE (*bind-handle f g h*) = *bind-handle f* (λ*x*. *ignoreE* (*g x*)) (λ*x*. *ignoreE* (*h x*))
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *ignoreE-assume-result-and-state*[*ignoreE-simps*]:
ignoreE (*assume-result-and-state X*) = *assume-result-and-state X*
by (*simp add*: *assume-result-and-state-def ignoreE-simps*)

lemma *ignoreE-assert-result-and-state*[*ignoreE-simps*]:
ignoreE (*assert-result-and-state X*) = *assert-result-and-state X*
by (*simp add*: *assert-result-and-state-def ignoreE-simps*)

lemma *ignoreE-throw-exception-or-result*[*ignoreE-simps*]:
x ≠ *default* $\Longrightarrow$ *ignoreE* (*throw-exception-or-result x*) = ⊥
by (*simp add*: *ignoreE-def catch-def*)

lemma *bind-handle-cong*:
assumes *eq*: $f = f'$ $g = g'$
shows $(\bigwedge x.\ x \neq default \Longrightarrow h\ x = h'\ x) \Longrightarrow$
bind-handle f g h = *bind-handle f′ g′ h′*
unfolding *eq*
apply *transfer*
apply (*intro ext arg-cong*[**where** *f*=*bind-post-state* -])
apply (*simp add*: *split-beta′ fun-eq-iff split*: *exception-or-result-split*)
done

lemma *bind-handle-ignoreE*[*ignoreE-simps*]: *bind-handle* (*ignoreE f*) *g h* = *bind* (*ignoreE f*) *g*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *bind-handle-eq-bind*:
bind-handle (*f* :: (-, -) *res-monad*) *g h* = *bind f g*
unfolding *bind-def*
by (*rule bind-handle-cong*) *simp-all*

lemma *bind-handle-bot-eq-ignoreE*[*ignoreE-simps*]:
bind-handle f g (λ*x*. ⊥) = *bind* (*ignoreE f*) *g*
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *state-select-bot*: ∀ *s*. ∃ *t*. (*s*, *t*) ∈ *h* ⟹ *do* { *x* ← *state-select h*; ⊥ } = ⊥
by (*rule spec-monad-eqI*) (*simp add*: *runs-to-iff*)

lemma *runs-to-cong*:
f · *s* ?⦃ λ*r t*. *P r t* ⟷ *Q r t*⦄ ⟹ *f* · *s* ⦃ *P* ⦄ ⟷ *f* · *s* ⦃ *Q* ⦄
by (*cases run f s*) (*auto simp*: *runs-to.rep-eq runs-to-partial.rep-eq*)

lemma *run-ignoreE-on-exit′*:
assumes *f-h*: *f* · *s* ?⦃ λ*r t*. ∀ *e*. *e* ≠ *default* ⟶ *r* = *Exception e* ⟶ *h* · *t* ⦃λ- -. *True* ⦄⦄
shows *run* (*ignoreE* (*on-exit′ f h*)) *s* = *run* (*on-exit′* (*ignoreE f*) (*ignoreE h*)) *s*
apply (*simp add*: *run-runs-to-extensionality runs-to-iff*)
apply (*intro runs-to-cong allI*)
apply (*rule runs-to-partial-weaken*[*OF f-h*])
subgoal for *P r t*
by (*cases r*; *simp add*: *case-distrib-exception-or-result*[*of* λ*x*. *x* = -]
split: *exception-or-result-split*)
done

lemma *ignoreE-on-exit′*[*ignoreE-simps*]:
(⋀*s*. *f* · *s* ?⦃ λ*r t*. ∀ *e*. *e* ≠ *default* ⟶ *r* = *Exception e* ⟶ *h* · *t* ⦃λ- -. *True* ⦄⦄) ⟹
ignoreE (*on-exit′ f h*) = *on-exit′* (*ignoreE f*) (*ignoreE h*)
by (*simp add*: *spec-monad-ext-iff run-ignoreE-on-exit′*)

lemma *ignoreE-on-exit*[*ignoreE-simps*]:
  $\forall s.\ \exists t.\ (s,\ t) \in h \Longrightarrow$ *ignoreE* (*on-exit f h*) = *on-exit* (*ignoreE f*) *h*
  by (*simp add*: *runs-to-iff ignoreE-simps on-exit-def*)

lemma *sim-post-state-vmap-post-state1*:
  sim-post-state ($\lambda x\ y.\ \forall x'.\ f\ x' = x \longrightarrow R\ x'\ y$) *p q* $\Longrightarrow$
    sim-post-state R (*vmap-post-state f p*) *q*
  by (*cases p*; *cases q*; *auto simp*: *Ball-def Bex-def*)

lemma *refines-vmap-value-left*:
  refines f g s t ($\lambda(r,\ s)\ y.\ \forall x.\ r = m\ x \longrightarrow R\ (x,\ s)\ y$) $\Longrightarrow$
    refines (*vmap-value m f*) *g s t R*
  apply *transfer*
  apply (*rule sim-post-state-vmap-post-state1*[*OF sim-post-state-weaken*], *assumption*)
  apply *auto*
  done

lemma *refines-ignoreE-left*:
  refines f g s t ($\lambda(r,\ s)\ y.\ \forall x.\ r = Result\ x \longrightarrow R\ (Result\ x,\ s)\ y$) $\Longrightarrow$
    refines (*ignoreE f*) *g s t R*
  unfolding *ignoreE-eq*
  by (*rule refines-vmap-value-left*[*OF refines-weaken*], *assumption*) *auto*

lemma *finally-when-throw*:
  finally (*when P* (*throw x*) >>= *f*) =
    condition (λ-. *P*) (*return x*) (*finally* (*f* ()))
  apply (*rule spec-monad-eqI*)
  apply (*simp add*: *runs-to-iff*)
  done

lemma *finally-gets-bind*:
  finally (*do* { *r* ← *gets X*; *Y r* }) = *do* { *r* ← *gets X*; *finally* (*Y r*) }
  apply (*rule spec-monad-eqI*)
  apply (*simp add*: *runs-to-iff*)
  done

lemma *finally-modify-bind*:
  finally (*do* { *r* ← *modify X*; *Y r* }) = *do* { *r* ← *modify X*; *finally* (*Y r*) }
  apply (*rule spec-monad-eqI*)
  apply (*simp add*: *runs-to-iff*)
  done

lemma *finally-return*: *finally* (*return x*) = *return x*
  apply (*rule spec-monad-eqI*)
  apply (*simp add*: *runs-to-iff*)
  done

lemma *f-catch-throw*: (*f* <*catch*> *throw*) = *f*

by (*rule spec-monad-eqI*) (*clarsimp simp add*: *runs-to-iff*)

end

theory *Reaches*
imports *Spec-Monad*
begin

The core notions on spec monads are

- Properties of a monad: *runs-to*, *runs-to-partial*
- Refinement / Simulation of monads: *refines*, *rel-spec* and *rel-spec-monad*.

  It is considered good style to use these more abstract concepts as much as possible.

  Next we introduce some notions that are more in a 'pointwise' spirit in the sense that they talk about a particular outcome of a monadic computation, e.g. that a particular state is a reachable outcome of a monadic computation.

  There is nothing *wrong* with these notions but we consider them as less elegant and more 'brute force'. So we encourage to think twice before using them and prefer the more abstract notions.

primrec *outcomes* :: *'a post-state* ⇒ *'a set* **where**
outcomes Failure = {}
| *outcomes* (*Success X*) = *X*

lemma *le-post-state-iff*:
p ≤ *q* ⟷ (*q* ≠ *Failure* ⟶ (*p* ≠ *Failure* ∧ *outcomes p* ⊆ *outcomes q*))
by (*cases q*; *cases p*) *simp-all*

lemma *outcomes-map*[*simp*]: *outcomes* (*map-post-state f x*) = *f* ‘ *outcomes x*
by (*cases x*) *simp-all*

lemma *map-post-state-eq-Failure*[*simp*]:
map-post-state f x = *Failure* ⟷ *x* = *Failure*
by (*cases x*; *simp*)

lemma *outcomes-Sup*[*simp*]: *Failure* ∉ *F* ⟹ *outcomes* (*Sup F*) = (⋃*f*∈*F*. *outcomes f*)
by (*auto simp*: *Sup-post-state-def*)
(*metis outcomes.simps*(*2*) *post-state.exhaust vimage-eq*)

lemma *mem-outcomes-iff*: *x* ∈ *outcomes p* ⟷ (∃ *X*. *p* = *Success X* ∧ *x* ∈ *X*)
by (*cases p*) *auto*

lemma *outcomes-Sup-eq-if*:
outcomes (*Sup X*) = (*if Failure* ∈ *X then* {} *else* ⋃*x*∈*X*. *outcomes x*)
by (*auto simp*: *Sup-post-state-def mem-outcomes-iff Bex-def*)

lemmas *runs-to-holds-def* = *runs-to.rep-eq*
lemmas *runs-to-partial-holds-partial-def* = *runs-to-partial.rep-eq*

16.11 *succeeds* and *reaches*

definition *succeeds* :: ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ '*s* ⇒ *bool* **where**
succeeds f s ⟷ *run f s* ≠ ⊤

definition *reaches* :: ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ '*s* ⇒ ('*e*, '*a*) *exception-or-result* ⇒ '*s* ⇒ *bool* **where**
reaches f s r' s' ⟷ (*r'*, *s'*) ∈ *outcomes* (*run f s*)

There seems to a similarity between what happens in HOL with the dualism of sets as type vs. the characterisation as predicate as expressed in *Collect*. Similar '*a post-state* has a set in the *Success* case. In particular with *holds-post-state* we switch to the predicate view. Which is then continued in *runs-to* which is the predicate view and the set view with *reaches*, which is more the element wise set view similar to *x* ∈ *X* and finally culminates in (*?f* = *?g*) = (∀ *P s*. (*?f* ⋆ *s* ⦇ *P* ⦈) = (*?g* ⋆ *s* ⦇ *P* ⦈)). The predicate view seems to be the one we finally aim for. So maybe we could get completely rid of the set like things and start of with a predicate already in '*a post-state*?

lemma *runs-to-partial-def-old*:
runs-to-partial f s Q ⟷ (*succeeds f s* ⟶ (∀ *r t*. *reaches f s r t* ⟶ *Q r t*))
unfolding *succeeds-def reaches-def runs-to-partial.rep-eq*
by (*cases run f s*) (*auto simp*: *top-post-state-def*)

lemma *runs-to-def-old*: *runs-to f s Q* ⟷ *succeeds f s* ∧ (∀ *r t*. *reaches f s r t* ⟶ *Q r t*)
unfolding *succeeds-def reaches-def runs-to.rep-eq*
by (*cases run f s*) (*auto simp*: *top-post-state-def*)

lemma *runs-to-succeeds-runsto-partial-conv*:
runs-to f s Q ⟷ (*succeeds f s* ∧ *runs-to-partial f s Q*)
by (*auto simp add*: *runs-to-def-old runs-to-partial-def-old*)

lemma *run-Success-succeeds*: *run f s* = *Success x* ⟹ *succeeds f s*
by (*auto simp add*: *succeeds-def top-post-state-def*)

lemma *runs-to-partial-le-outcomes-conv*:
runs-to-partial f s Q ⟷ (*outcomes* (*run f s*)) ≤ {(*r*,*t*). *Q r t*}
apply (*simp add*: *runs-to-partial-def-old succeeds-def outcomes-def reaches-def top-post-state-def*)
apply *transfer*

apply *standard*
subgoal
by (*auto split*: *post-state.splits*)
subgoal
by *blast*
done

lemma *not-succeeds-empty-outcomes*: ¬ *succeeds f s* ⟹ *outcomes* (*run f s*) = {}
by (*simp add*: *succeeds-def top-post-state-def*)

lemma *reaches-succeeds*: *reaches f s r t* ⟹ *succeeds f s*
apply (*simp add*: *reaches-def succeeds-def top-post-state-def*)
apply *transfer*
apply *auto*
done

lemma *always-progress-succeeds-reaches-conv*:
always-progress f ⟷ (∀ *s*. *succeeds f s* ⟶ (∃ *r t*. *reaches f s r t*))
apply (*simp add*: *always-progress-def succeeds-def reaches-def*)
apply *transfer*
apply (*clarsimp simp add*: *top-post-state-def bot-post-state-def*, *safe*)
apply (*metis ex-in-conv outcomes.simps*(*2*) *post-state.exhaust surj-pair*)
by (*metis empty-iff post-state.distinct*(*1*) *outcomes.simps*(*2*))

lemma *le-succeedsD*: *g* ≤ *f* ⟹ *succeeds f s* ⟹ *succeeds g s*
apply (*simp add*: *succeeds-def*)
apply *transfer*
by (*auto simp add*: *le-fun-def less-eq-post-state.simps top-post-state-def*)

lemma *outcomes-succeeds-run-conv*: *outcomes* (*run f s*) = *X* ⟹ *succeeds f s* ⟹
run f s = *Success X*
by (*cases run f s*) (*auto simp add*: *succeeds-def top-post-state-def*)

lemma *succeeds-outcomes-run-eqI*: *succeeds f s* ⟷ *succeeds g s* ⟹
(*succeeds f s* ⟹ *succeeds g s* ⟹ (*outcomes* (*run f s*) = *outcomes* (*run g s*)))
⟹
run f s = *run g s*
by (*cases run f s*; *cases run g s*) (*auto simp add*: *succeeds-def top-post-state-def*)

lemma *succeeds-outcomes-spec-monad-eqI*:
(⋀*s*. *succeeds f s* ⟷ *succeeds g s*) ⟹
(⋀*s*. *succeeds f s* ⟹ *succeeds g s* ⟹ (*outcomes* (*run f s*) = *outcomes* (*run g s*)))
⟹
f = *g*
apply (*rule spec-monad-ext*)
apply (*auto intro*: *succeeds-outcomes-run-eqI*)
done

lemma *succeeds-reaches-spec-monad-eqI*:

$(\bigwedge s.\ succeeds\ f\ s \longleftrightarrow succeeds\ g\ s) \Longrightarrow$
$(\bigwedge s\ r\ t.\ succeeds\ f\ s \Longrightarrow succeeds\ g\ s \Longrightarrow (reaches\ f\ s\ r\ t \longleftrightarrow reaches\ g\ s\ r\ t)) \Longrightarrow$
$f = g$
apply (*rule succeeds-outcomes-spec-monad-eqI*)
apply *simp*
apply (*auto simp add*: *reaches-def*)
done

lemma *succeeds-runs-to-iff*: $succeeds\ f\ s \longleftrightarrow runs\text{-}to\ f\ s\ (\lambda\text{- -}.\ True)$
by (*simp add*: *runs-to-def-old*)

named-theorems *outcomes-spec-monad Simplification rules for outcomes of Spec monad*

16.11.1 Relational rewriting for Monads

lemma *refines-def-old*:
$refines\ f\ g\ s\ t\ R \longleftrightarrow$
$(succeeds\ g\ t \longrightarrow succeeds\ f\ s \wedge$
$(\forall r\ s'.\ reaches\ f\ s\ r\ s' \longrightarrow (\exists x\ t'.\ reaches\ g\ t\ x\ t' \wedge R\ (r,\ s')\ (x,\ t'))))$
by (*cases run g t*; *cases run f s*)
(*force simp*: *refines.rep-eq succeeds-def reaches-def top-post-state-def*)+

lemma *rel-specI*:
assumes $succeeds\ f\ s \longleftrightarrow succeeds\ g\ t$
assumes $\bigwedge r\ s'.\ reaches\ f\ s\ r\ s' \Longrightarrow \exists q\ t'.\ reaches\ g\ t\ q\ t' \wedge R\ (r,\ s')\ (q,\ t')$
assumes $\bigwedge q\ t'.\ reaches\ g\ t\ q\ t' \Longrightarrow \exists r\ s'.\ reaches\ f\ s\ r\ s' \wedge R\ (r,\ s')\ (q,\ t')$
shows *rel-spec f g s t R*
using *assms*
by (*cases run f s*; *cases run g t*;
fastforce simp: *rel-spec-def rel-prod.simps rel-set-def*
reaches-def succeeds-def top-post-state-def)

lemma *rel-specD-succeeds*:
$rel\text{-}spec\ f\ g\ s\ t\ R \Longrightarrow succeeds\ f\ s \longleftrightarrow succeeds\ g\ t$
by (*cases run f s*; *cases run g t*;
simp add: *rel-post-state.simps rel-spec-def rel-prod.simps succeeds-def top-post-state-def*)

lemma *rel-specD-reaches1*:
$rel\text{-}spec\ f\ g\ s\ t\ R \Longrightarrow reaches\ f\ s\ r\ s' \Longrightarrow \exists q\ t'.\ reaches\ g\ t\ q\ t' \wedge R\ (r,\ s')\ (q,\ t')$
by (*cases run f s*; *cases run g t*;
fastforce simp: *rel-post-state.simps rel-spec-def rel-prod.simps rel-set-def*
reaches-def top-post-state-def)

lemma *rel-specD-reaches2*:
$rel\text{-}spec\ f\ g\ s\ t\ R \Longrightarrow reaches\ g\ t\ q\ t' \Longrightarrow \exists r\ s'.\ reaches\ f\ s\ r\ s' \wedge R\ (r,\ s')\ (q,\ t')$
by (*cases run f s*; *cases run g t*;
fastforce simp: *rel-post-state.simps rel-spec-def rel-prod.simps rel-set-def*
reaches-def top-post-state-def)

lemma *refines-iff*:
refines f g s t R $\longleftrightarrow$
(*succeeds g t* $\longrightarrow$ *succeeds f s*) $\wedge$
(*succeeds g t* $\longrightarrow$ *succeeds f s* $\longrightarrow$
($\forall$ *r s'*. *reaches f s r s'* $\longrightarrow$ ($\exists$ *q t'*. *reaches g t q t'* $\wedge$ *R* (*r*, *s'*) (*q*, *t'*)))))
by (*simp add*: *refines-def-old*) *blast*

lemma *refinesI*:
assumes *succeeds g t* $\Longrightarrow$ *succeeds f s*
assumes $\bigwedge$*r s'*. *succeeds g t* $\Longrightarrow$ *succeeds f s* $\Longrightarrow$ *reaches f s r s'* $\Longrightarrow$
($\exists$ *q t'*. *reaches g t q t'* $\wedge$ *R* (*r*, *s'*) (*q*, *t'*))
shows *refines f g s t R*
using *assms*
by (*auto simp add*: *refines-def-old*)

lemma *refinesD-succeeds*:
refines f g s t R $\Longrightarrow$ *succeeds g t* $\Longrightarrow$ *succeeds f s*
by (*auto simp*: *refines-def-old*)

lemma *refinesD-reaches*:
refines f g s t R $\Longrightarrow$ *reaches f s r s'* $\Longrightarrow$ *succeeds g t*
$\Longrightarrow$ $\exists$ *q t'*. *reaches g t q t'* $\wedge$ *R* (*r*, *s'*) (*q*, *t'*)
by (*fastforce simp*: *refines-def-old*)

lemma *refines-strengthen'*:
assumes *refines f g s t R*
assumes $\bigwedge$*r u q v*. *reaches f s r u* $\Longrightarrow$ *reaches g t q v* $\Longrightarrow$ *succeeds f s* $\Longrightarrow$ *succeeds g t*
$\Longrightarrow$ *R* (*r*, *u*) (*q*, *v*) $\Longrightarrow$ *Q* (*r*, *u*) (*q*, *v*)
shows *refines f g s t Q*
using *assms* **by** (*fastforce simp*: *refines-def-old*)

lemma *always-progressD*: *always-progress f* $\Longrightarrow$ *succeeds f s* $\Longrightarrow$ $\exists$ *r t*. *reaches f s r t*
by (*auto simp*:*always-progress-succeeds-reaches-conv*)

lemma *Ex-reaches*: *f* $\cdot$ *s* $\{\!|$ *P* $|\!\}$ $\Longrightarrow$ *always-progress f* $\Longrightarrow$ $\exists$ *r t*. *reaches f s r t*
by (*auto simp*: *runs-to-def-old always-progress-succeeds-reaches-conv*)

lemma *witness-outcomes-succeeds*: *x* $\in$ *outcomes* (*run f s*) $\Longrightarrow$ *succeeds f s*
using *not-succeeds-empty-outcomes* **by** *fastforce*

lemma *runs-toD2*: *f* $\cdot$ *s* $\{\!|$ *P* $|\!\}$ $\Longrightarrow$ *reaches f s r t* $\Longrightarrow$ *P r t*
by (*simp add*: *runs-to-def-old*)

lemma *runs-toD2-res*: **fixes** *f*:: ('*a*, '*s*) *res-monad*
shows *f* $\cdot$ *s* $\{\!|\lambda$*Res r*. *P r* $|\!\}$ $\Longrightarrow$ *reaches f s* (*Result r*) *t* $\Longrightarrow$ *P r t*
by (*simp add*: *runs-to-def-old*)

lemma *rel-post-state-runI*:
assumes *succeeds f s* $\longleftrightarrow$ *succeeds g t*
assumes *succeeds f s* $\Longrightarrow$ *succeeds g t* $\Longrightarrow$ *rel-set R* (*outcomes* (*run f s*)) (*outcomes* (*run g t*))
shows *rel-post-state R* (*run f s*) (*run g t*)
using *assms*
by (*cases run f s*; *cases run g t*)
(*auto simp add*: *rel-post-state.simps succeeds-def top-post-state-def*)

lemma *rel-post-state-runI'*:
assumes *succeeds f s* $\longleftrightarrow$ *succeeds g t*
assumes *succeeds f s* $\Longrightarrow$ *succeeds g t* $\Longrightarrow$ *rel-post-state R* (*run f s*) (*run g t*)
shows *rel-post-state R* (*run f s*) (*run g t*)
using *assms*
by (*auto simp add*: *rel-post-state.simps succeeds-def top-post-state-def*)

lemma *rel-post-state-runD*:
assumes *rel-post-state R* (*run f s*) (*run g t*)
shows (*succeeds f s* $\longleftrightarrow$ *succeeds g t*) $\wedge$
(*succeeds f s* $\longrightarrow$ *succeeds g t* $\longrightarrow$ *rel-set R* (*outcomes* (*run f s*)) (*outcomes* (*run g t*)))
using *assms*
by (*auto simp add*: *rel-post-state.simps succeeds-def top-post-state-def*)

lemma *rel-post-state-run-iff*:
rel-post-state R (*run f s*) (*run g t*) $\longleftrightarrow$
((*succeeds f s* $\longleftrightarrow$ *succeeds g t*) $\wedge$
(*succeeds f s* $\longrightarrow$ *succeeds g t* $\longrightarrow$ *rel-set R* (*outcomes* (*run f s*)) (*outcomes* (*run g t*))))
using *rel-post-state-runI rel-post-state-runD* **by** *metis*

lemma *rel-spec-monad-succeeds-iff*: *rel-spec-monad R Q f g* $\Longrightarrow$ *R s t* $\Longrightarrow$ *succeeds f s* $\longleftrightarrow$ *succeeds g t*
by (*meson rel-specD-succeeds rel-spec-monad-iff-rel-spec*)

lemma *outcomes-top-ps*[*simp*]: *outcomes* $\top$ = {} *outcomes* (*pure-post-state x*) = {*x*}
outcomes $\bot$ = {}
by (*auto simp*: *top-post-state-def pure-post-state-def bot-post-state-def*)

16.11.2 $\top$

lemma *outcomes-top*[*outcomes-spec-monad*]: *outcomes* (*run* $\top$ *s*) = {}
by *transfer* (*simp add*: *top-spec-monad-def top-post-state-def*)

lemma *succeeds-top*[*simp*]: *succeeds* $\top$ *s* $\longleftrightarrow$ *False*
unfolding *succeeds-def*
by *transfer* (*simp add*: *top-spec-monad-def top-post-state-def*)

lemma *reaches-top*[*simp*]: *reaches* ⊤ *s r t* ⟷ *False*
unfolding *reaches-def*
by (*simp add*: *outcomes-top top-post-state-def*)

16.11.3 ⊥

lemma *outcomes-bot*[*outcomes-spec-monad*]: *outcomes* (*run* ⊥ *s*) = {}
by *transfer* (*simp add*: *bot-spec-monad-def bot-post-state-def*)

lemma *succeeds-bot*[*simp*]: *succeeds* ⊥ *s* ⟷ *True*
unfolding *succeeds-def*
by *transfer* (*simp add*: *bot-spec-monad-def bot-post-state-def top-post-state-def*)

lemma *reaches-bot*[*simp*]: *reaches* ⊥ *s r t* ⟷ *False*
unfolding *reaches-def*
by (*simp add*: *outcomes-bot bot-post-state-def*)

16.11.4 *fail*

lemma *outcomes-fail*[*outcomes-spec-monad*]: *outcomes* (*run fail s*) = {}
by *transfer simp*

lemma *succeeds-fail-iff*[*iff*]: ¬ (*succeeds fail f*)
by (*simp add*: *succeeds-def*)

lemma *reaches-fail-iff*[*iff*]: ¬ *reaches fail s r t*
by (*simp add*: *reaches-def outcomes-spec-monad top-post-state-def*)

16.11.5 *yield*

lemma *succeeds-yield*[*simp*]: *succeeds* (*yield r*) *s*
unfolding *succeeds-def*
apply *transfer*
apply (*simp add*: *pure-post-state-def top-post-state-def*)
done

lemma *outcomes-yield* [*outcomes-spec-monad*]: *outcomes* (*run* (*yield r*) *s*) = {(*r*, *s*)}
by (*simp add*: *pure-post-state-def*)

lemma *reaches-yield*[*simp*]: *reaches* (*yield v*) *s r t* ⟷ (*r* = *v* ∧ (*t* = *s*))
by (*simp add*: *reaches-def outcomes-spec-monad pure-post-state-def*)

16.11.6 *return*

lemma *outcomes-return* [*outcomes-spec-monad*]:
outcomes (*run* (*return v*) *s*) = {(*Result v*, *s*)}
by *transfer* (*simp add*: *pure-post-state-def*)

lemma *succeeds-return* [*iff*]: *succeeds* (*return v*) *s*
by (*simp add*: *succeeds-def top-post-state-def*)

lemma *reaches-return*[*simp*]: *reaches* (*return v*) *s r t* ⟷ (*r* = *Result v* ∧ (*t* = *s*))
by (*simp add*: *reaches-def outcomes-spec-monad pure-post-state-def*)

lemma *refines-yield-left*:
refines (*yield x*) *f s t R*
if *succeeds f t* ⟹ *reaches f t x′ s′* ∧ *R* (*x*, *s*) (*x′*, *s′*)
using *that*
unfolding *refines-def-old*
by (*auto simp*: *succeeds-def top-post-state-def intro*!: *bexI*[**where** *x*=(*x′*, *s′*)])

16.11.7 *skip*

lemma *outcomes-skip* [*outcomes-spec-monad*]:
outcomes (*run* (*skip*) *s*) = {(*Result* (), *s*)}
by (*simp add*: *outcomes-spec-monad pure-post-state-def*)

lemma *succeeds-skip*: *succeeds skip s*
by *simp*

lemma *reaches-skip*: *reaches* (*return v*) *s r t* ⟷ (*r* = *Result* () ∧ (*t* = *s*))
by (*simp add*: *reaches-def outcomes-spec-monad pure-post-state-def*)

lemma *runs-to-skip*[*runs-to-vcg*]: *skip* · *s* ⦃ λ- *t*. *t* = *s* ⦄
by *simp*

lemma *runs-to-partial-skip*[*runs-to-vcg*]: *skip* · *s* ?⦃ λ- *t*. *t* = *s* ⦄
by *simp*

lemma *runs-to-skip-iff*[*runs-to-iff*]: *skip* · *s* ⦃*Q*⦄ ⟷ *Q* (*Result* ()) *s*
by (*simp add*: *runs-to-def-old*)

16.11.8 *throw-exception-or-result*

lemma *outcomes-throw-exception-or-result* [*outcomes-spec-monad*]:
outcomes (*run* (*throw-exception-or-result x*) *s*) = {(*Exception x*, *s*)}
by *transfer* (*simp add*: *pure-post-state-def*)

lemma *succeeds-throw-exception-or-result* [*iff*]: *succeeds* (*throw-exception-or-result v*) *s*
by (*simp add*: *succeeds-def top-post-state-def*)

lemma *reaches-throw-exception-or-result*[*simp*]:
reaches (*throw-exception-or-result x*) *s r t* ⟷ (*r* = *Exception x* ∧ (*t* = *s*))
by (*simp add*: *reaches-def outcomes-spec-monad pure-post-state-def*)

16.11.9 *throw*

lemma *outcomes-throw* [*outcomes-spec-monad*]:
outcomes (*run* (*throw e*) *s*) = {(*Exn e*, *s*)}
by (*simp add*: *pure-post-state-def*)

16.11.10 *get-state*

lemma *outcomes-get-state* [*outcomes-spec-monad*]:
outcomes (*run get-state s*) = {(*Result s*, *s*)}
by *transfer* (*simp add*: *pure-post-state-def*)

lemma *succeeds-get-state* [*iff*]: *succeeds get-state s*
by (*simp add*: *succeeds-def*)

lemma *reaches-get-state*[*simp*]:
reaches get-state s r t ⟷ (*r* = *Result s* ∧ (*t* = *s*))
by (*simp add*: *reaches-def outcomes-spec-monad pure-post-state-def*)

16.11.11 *set-state*

lemma *outcomes-set-state* [*outcomes-spec-monad*]:
outcomes (*run* (*set-state t*) *s*) = {(*Result* (), *t*)}
by *transfer* (*simp add*: *pure-post-state-def*)

lemma *succeeds-set-state* [*iff*]: *succeeds* (*set-state t*) *s*
by (*simp add*: *succeeds-def*)

lemma *reaches-set-state*[*simp*]: *reaches* (*set-state s′*) *s r t* ⟷ (*r* = *Result* () ∧ (*t* = *s′*))
by (*simp add*: *reaches-def outcomes-spec-monad pure-post-state-def*)

16.11.12 *select*

lemma *succeeds-holds*: *succeeds f s* ⟷ *holds-post-state* (λ*x*. *True*) (*run f s*)
by (*cases run f s*) (*simp-all add*: *succeeds-def top-post-state-def*)

lemma *succeeds-select*[*simp*]: *succeeds* (*select S*) *s*
apply (*simp add*: *succeeds-holds select-def*)
apply *transfer*
apply (*auto simp add*: *pure-post-state-def*)
done

lemma *select-outcomes*[*outcomes-spec-monad*]: *outcomes* (*run* (*select S*) *s*) = (λ*v*. (*Result v*, *s*)) ‘ *S*
apply (*simp add*: *select-def*)
apply *transfer*
apply (*force simp add*: *pure-post-state-def outcomes-def Inf-set-def Sup-post-state-def*
split: *post-state.splits prod.splits*)
done

lemma *reaches-select* [*simp*]: *reaches* (*select S*) *s r t* ⟷ (*r* ∈ *Result* ‘ *S* ∧ *t* = *s*)
apply (*auto simp add*: *reaches-def outcomes-spec-monad*)
done

16.11.13 *unknown*

lemma *succeeds-unknown*[*simp*]: *succeeds unknown s*
unfolding *unknown-def* **by** *simp*

lemma *unknown-outcomes*[*outcomes-spec-monad*]: *outcomes* (*run unknown s*) = (λ*v*. (*Result v*, *s*)) ‘ *UNIV*
by *simp*

lemma *reaches-unknown* [*simp*]: *reaches unknown s r t* ⟷ (*r* ∈ *Result* ‘ *UNIV* ∧ *t* = *s*)
unfolding *unknown-def* **by** *simp*

16.11.14 *lift-state*

lemma *succeeds-lift-state-iff*: *succeeds* (*lift-state R f*) *s* ⟷ (∀ *s'*. *R s s'* ⟶ *succeeds f s'*)
by (*simp add*: *succeeds-holds lift-state-def Spec-inverse*)

16.11.15 **const***exec-concrete*

lemma *succeeds-exec-concrete-iff*: *succeeds* (*exec-concrete st f*) *s* ⟷ (∀ *t*. *s* = *st t* ⟶ *succeeds f t*)
by (*simp add*: *exec-concrete-def succeeds-lift-state-iff*)

lemma *reaches-exec-concrete*:
assumes *succeeds*: *succeeds* (*exec-concrete st f*) *s*
shows *reaches* (*exec-concrete st f*) *s r s'* ⟷ (∃ *t t'*. *s* = *st t* ∧ *reaches f t r t'* ∧ *s'* = *st t'*)
apply (*simp add*: *reaches-def run-exec-concrete*)
apply *standard*
subgoal
by (*auto simp add*: *map-post-state-def Sup-post-state-def image-def vimage-def*
split: *prod.splits post-state.splits if-split-asm*)
(*metis outcomes.simps*(*2*))
subgoal
using *succeeds*
apply (*simp add*: *succeeds-def run-exec-concrete*)
by (*auto simp add*: *map-post-state-def Sup-post-state-def image-def vimage-def top-post-state-def*
split: *prod.splits post-state.splits if-split-asm*)
(*metis apsnd-conv outcomes.simps*(*2*) *post-state.exhaust*)
done

16.11.16 const *exec-abstract*

lemma *succeeds-exec-abstract-iff*[*simp*]: *succeeds* (*exec-abstract st f*) *s* ⟷ *succeeds f* (*st s*)
by (*simp add*: *exec-abstract-def succeeds-lift-state-iff*)

lemma *reaches-exec-abstract*:
shows *reaches* (*exec-abstract st f*) *s r s'* ⟷ (∃ *t'*. *reaches f* (*st s*) *r t'* ∧ *t'* = *st s'*)
by (*cases run f* (*st s*)) (*simp-all add*: *reaches-def run-exec-abstract*)

16.11.17 *bind-handle*

lemma *succeeds-bind-handle*:
succeeds (*bind-handle f g h*) *s* ⟷
(*succeeds f s* ∧
(∀ *x s'*. *reaches f s x s'* ⟶
(*case x of Exception e* ⇒ *succeeds* (*h e*) *s'* | *Result v* ⇒ *succeeds* (*g v*) *s'*)))
apply (*cases run f s*)
apply (*simp-all add*:
succeeds-def run-bind-handle bot-post-state-def top-post-state-def
image-iff eq-commute[*of Failure*] *reaches-def*
split: *prod.splits exception-or-result-splits*)
apply *auto*
done

lemma *succeeds-bind-handle-res*:
succeeds (*bind-handle* (*f*::('*s*, '*a*) *res-monad*) *g h*) *s* ⟷
(*succeeds f s* ∧
(∀ *v s'*. *reaches f s* (*Result v*) *s'* ⟶
succeeds (*g v*) *s'*))
by (*simp add*: *succeeds-bind-handle*)

lemma *outcomes-bind-handle-succeeds*: *succeeds* (*bind-handle f g h*) *s* ⟹
outcomes (*run* (*bind-handle f g h*) *s*) =
⋃ ((λ(*r*, *s'*). *case r of Exception e* => *outcomes* (*run* (*h e*) *s'*)
| *Result v* ⇒ *outcomes* (*run* (*g v*) *s'*))
‘ (*outcomes* (*run f s*)))
apply (*cases run f s*)
apply (*simp-all add*: *succeeds-bind-handle succeeds-def run-bind-handle bind-post-state-eq-top*)
apply (*simp add*: *top-post-state-def image-iff split-beta'*
split: *exception-or-result-splits*)
apply (*intro SUP-cong refl*)
subgoal for *X x* **by** (*cases fst x*) *auto*
done

lemma *outcomes-bind-handle-succeeds-res*: *succeeds* (*bind-handle* (*f*::('*a*, '*s*) *res-monad*) *g h*) *s* ⟹
outcomes (*run* (*bind-handle f g h*) *s*) =
⋃ ((λ(*r*, *s'*). *case r of Exception e* => {} | *Result v* ⇒ *outcomes* (*run* (*g v*)

s'))
‘ (*outcomes* (*run f s*)))
apply (*simp add*: *outcomes-bind-handle-succeeds*)
apply (*auto split*: *exception-or-result-splits*)
done

lemma *reaches-bind-handle*: *reaches* (*bind-handle f g h*) *s r t* ⟷
(*succeeds* (*bind-handle f g h*) *s* ∧
(∃ *r' s'*. *reaches f s r' s'* ∧
(*case r' of*
Exception e ⇒ *reaches* (*h e*) *s' r t*
| *Result v* ⇒ *reaches* (*g v*) *s' r t*)))
apply *standard*
subgoal
apply (*frule reaches-succeeds*)
apply (*simp add*: *reaches-def outcomes-bind-handle-succeeds*)
apply (*clarsimp split*: *exception-or-result-splits*)
apply (*metis* (*no-types*, *opaque-lifting*) *Exception-eq-Result the-Exception-simp*)
apply (*metis* (*no-types*, *opaque-lifting*) *Result-eq-Result the-Exception-simp the-Exception-Result*)
done
subgoal
apply (*simp add*: *reaches-def outcomes-bind-handle-succeeds*)
by (*auto split*: *exception-or-result-splits*)
(*metis* (*mono-tags*) *case-exception-or-result-Exception case-exception-or-result-Result case-prod-conv exception-or-result-cases*)
done

lemma *reaches-bind-handle-res*: *reaches* ((*bind-handle* (*f*::('*a*, '*s*) *res-monad*) *g h*)) *s r t* ⟷
(*succeeds* (*bind-handle f g h*) *s* ∧
(∃ *v s'*. *reaches f s* (*Result v*) *s'* ∧ *reaches* (*g v*) *s' r t*))
by (*simp add*: *reaches-bind-handle*)

16.11.18 (⩥)

lemma *succeeds-bind*:
succeeds (*bind f g*) *s* ⟷
(*succeeds f s* ∧ (∀ *v s'*. *reaches f s* (*Result v*) *s'* ⟶ *succeeds* (*g v*) *s'*))
apply (*simp add*: *bind-def succeeds-bind-handle*)
apply (*auto split*: *exception-or-result-splits*)
done

lemma *outcomes-bind-succeeds*: *succeeds* (*bind f g*) *s* ⟹ *outcomes* (*run* (*bind f g*) *s*) =
⋃ ((λ(*r*, *s'*). *case r of Exception e* => {(*Exception e*, *s'*)} | *Result v* ⇒ *outcomes* (*run* (*g v*) *s'*))
‘ (*outcomes* (*run f s*)))
by (*simp add*: *bind-def outcomes-bind-handle-succeeds outcomes-spec-monad pure-post-state-def*)

lemma *outcomes-bind-succeeds-res*: *succeeds* (*bind* (*f*::('*a*, '*s*) *res-monad*) *g*) *s* ⟹
outcomes (*run* (*bind f g*) *s*) =
⋃ ((λ(*r*, *s'*). *case r of Exception e* => {} | *Result v* ⇒ *outcomes* (*run* (*g v*) *s'*))
‘ (*outcomes* (*run f s*)))
by (*simp add*: *bind-def outcomes-bind-handle-succeeds-res outcomes-spec-monad*)

lemma *reaches-bind*: *reaches* (*bind f g*) *s r t* ⟷
(*succeeds* (*bind f g*) *s* ∧
(∃ *r' s'*. *reaches f s r' s'* ∧
(*case r' of*
Exception e ⇒ *r* = *Exception e* ∧ *t* = *s'*
| *Result v* ⇒ *reaches* (*g v*) *s' r t*)))
by (*simp add*: *bind-def reaches-bind-handle*)

lemma *reaches-bind-res*: *reaches* ((*bind f g*)::('*a*, '*s*) *res-monad*) *s r t* ⟷
(*succeeds* (*bind f g*) *s* ∧
(∃ *v s'*. *reaches f s* (*Result v*) *s'* ∧ *reaches* (*g v*) *s' r t*))
by (*simp add*: *bind-def reaches-bind-handle-res*)

lemma *runs-toD-outcomes*: *f* • *s* {| *P* |} ⟹ (*x*, *s*) ∈ *outcomes* (*run f s*) ⟹ *P x s*
by (*cases run f s*) (*auto simp*: *runs-to.rep-eq*)

lemma *run-bind-reaches-cong*:
assumes *f*: *run f s* = *run f' s*
assumes *g*: ⋀*v s'*. *succeeds f s* ⟹ *succeeds f' s* ⟹
reaches f' s (*Result v*) *s'* ⟹ *run* (*g v*) *s'* = *run* (*g' v*) *s'*
shows *run* (*bind f g*) *s* = *run* (*bind f' g'*) *s*
using *f g* **apply** (*cases run f' s*) **apply** (*auto simp add*: *run-bind succeeds-def*
reaches-def intro!: *SUP-cong*
split : *exception-or-result-splits*)
done

lemma *refines-bind-right'*:
succeeds g t ⟹ *reaches g t* (*Result a*) *t'* ⟹
refines f (*h a*) *s t' R* ⟹
refines f (*g* >>= *h*) *s t R*
by (*force simp*: *refines-iff succeeds-bind reaches-bind*)

lemma *outcomes-empty-bind*:
assumes *emp*: *outcomes* (*run f s*) = {}
shows *outcomes* (*run* (*f* >>= *g*) *s*) = {}
proof (*cases run f s*)
case *Failure*
then show *?thesis* **by** (*auto simp add*: *run-bind*)
next
case (*Success x2*)
then show *?thesis* **using** *emp*
by (*auto simp add*: *run-bind bot-post-state-def*)

qed

lemma *refines-bind-bind-exn*:
assumes *f*: *refines f f′ s s′ Q*
assumes *ll*: ⋀*e e′ t t′*.
Q (*Exn e*, *t*) (*Exn e′*, *t′*) ⟹
R (*Exn e*, *t*) (*Exn e′*, *t′*)
assumes *lr*: ⋀*e v′ t t′*.
Q (*Exn e*, *t*) (*Result v′*, *t′*) ⟹
refines (*throw e*) (*g′ v′*) *t t′ R*
assumes *rl*: ⋀*v e′ t t′*.
Q (*Result v*, *t*) (*Exn e′*, *t′*) ⟹
refines (*g v*) (*throw e′*) *t t′ R*
assumes *rr*: ⋀*v v′ t t′*.
Q (*Result v*, *t*) (*Result v′*, *t′*) ⟹
refines (*g v*) (*g′ v′*) *t t′ R*
shows *refines* (*f* ⤜ *g*) (*f′* ⤜ *g′*) *s s′ R*
apply (*rule refines-bind′*)
apply (*rule refines-mono* [*OF - f*])
using *assms*
apply (*auto simp add*: *default-option-def Exn-def*)
done

16.11.19 *assert*

lemma *outcomes-assert*[*outcomes-spec-monad*]:
outcomes (*run* (*assert P*) *s*) = (*if P then* {(*Result* (), *s*)} *else* {})
by (*simp add*: *assert-def outcomes-bind-succeeds outcomes-spec-monad*)

lemma *succeeds-assert*[*simp*]: *succeeds* (*assert P*) *s* ⟷ *P*
by (*simp add*: *assert-def*)

lemma *reaches-assert*[*simp*]: *reaches* (*assert P*) *s r t* ⟷ (*P* ∧ *r* = *Result* () ∧ *t* = *s*)
by (*simp add*: *assert-def*)

16.11.20 *assume*

lemma *outcomes-assume*[*outcomes-spec-monad*]:
outcomes (*run* (*assume P*) *s*) = (*if P then* {(*Result* (), *s*)} *else* {})
by (*simp add*: *assume-def outcomes-spec-monad*)

lemma *succeeds-assume*[*simp*]: *succeeds* (*assume P*) *s*
by (*simp add*: *assume-def*)

lemma *reaches-assume*[*simp*]: *reaches* (*assume P*) *s r t* ⟷ (*P* ∧ *r* = *Result* () ∧ *t* = *s*)
by (*simp add*: *assume-def*)

16.11.21 *assume-outcome*

lemma *outcomes-assume-outcome*[*outcomes-spec-monad*]:
outcomes (*run* (*assume-outcome f*) *s*) = *f s*
by *simp*

lemma *succeeds-assume-outcome*[*simp*]:
succeeds (*assume-outcome f*) *s*
by (*simp add*: *succeeds-def top-post-state-def*)

lemma *reaches-assume-outcome*[*simp*]:
reaches (*assume-outcome f*) *s r t* ⟷ (*r*, *t*) ∈ *f s*
by (*simp add*: *reaches-def*)

16.11.22 *assume-result-and-state*

lemma *outcomes-assume-result-and-state*[*outcomes-spec-monad*]:
outcomes (*run* (*assume-result-and-state f*) *s*) = ((λ(*v*, *t*). (*Result v*, *t*)) ‘ *f s*)
by *simp*

lemma *succeeds-assume-result-and-state*[*simp*]: *succeeds* (*assume-result-and-state f*) *s*
by (*simp add*: *assume-result-and-state-def succeeds-bind*)

lemma *Union-outcomes-split*:
(⋃*x*∈*f s*.
(*outcomes* (*run* (*case x of* (*v*, *y*) ⇒ *g v y*) *s*))) = (⋃(*v*, *y*)∈ *f s*. *outcomes* (*run* (*g v y*) *s*))
using *Sup.SUP-cong* **by** *auto*

lemma *reaches-assume-result-and-state* [*simp*]: *reaches* (*assume-result-and-state f*) *s r t* ⟷ (∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*)
by (*auto simp add*: *reaches-def outcomes-spec-monad*)

16.11.23 *gets*

lemma *succeeds-gets*[*simp*]: *succeeds* (*gets f*) *s*
by (*simp add*: *gets-def succeeds-bind*)

lemma *outcomes-gets*[*simp*]: *outcomes* (*run* (*gets f*) *s*) = {(*Result* (*f s*), *s*)}
by *simp*

lemma *reaches-gets*[*simp*]: *reaches* (*gets f*) *s r t* ⟷ (*r* = *Result* (*f s*) ∧ *t* = *s*)
by (*simp add*: *reaches-def*)

16.11.24 *assert-result-and-state*

lemma *succeeds-assert-result-and-state*[*simp*]: *succeeds* (*assert-result-and-state f*) *s* ⟷ *f s* ≠ {}
by (*simp add*: *assert-result-and-state-def succeeds-bind*)

lemma *outcomes-assert-result-and-state*[*outcomes-spec-monad*]:
outcomes (*run* (*assert-result-and-state f*) *s*) = (*if f s* = {} *then* {} *else* ((λ(*v*, *t*). (*Result v*, *t*)) ‘ *f s*))
proof (*cases f s* = {})
case *True*
hence ¬ *succeeds* (*assert-result-and-state f*) *s* **by** *simp*
hence *outcomes* (*run* (*assert-result-and-state f*) *s*) = {}
by (*simp add*: *not-succeeds-empty-outcomes*)
thus *?thesis* **by** (*simp add*: *True*)
next
case *False*
then show *?thesis*
by (*auto simp add*: *assert-result-and-state-def run-bind Sup-Success-pair pure-post-state-def*)
qed

lemma *reaches-assert-result-and-state* [*simp*]: *reaches* (*assert-result-and-state f*) *s r t* ⟷ (∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*)
by (*auto simp add*: *reaches-def outcomes-spec-monad*)

16.11.25 *assuming*

lemma *succeeds-assuming*[*simp*]: *succeeds* (*assuming g*) *s*
by (*simp add*: *assuming-def succeeds-bind*)

lemma *outcomes-assuming*[*outcomes-spec-monad*]: *outcomes* (*run* (*assuming g*) *s*) = (*if g s then* {(*Result* (), *s*)} *else* {})
by (*simp add*: *run-assuming*)

lemma *reaches-assuming*[*simp*]: *reaches* (*assuming g*) *s r t* ⟷ (*g s* ∧ *r* = *Result* () ∧ *t* = *s*)
by (*simp add*: *reaches-def outcomes-assuming*)

16.11.26 *guard*

lemma *succeeds-guard*[*simp*]: *succeeds* (*guard g*) *s* ⟷ *g s*
by (*simp add*: *guard-def succeeds-bind*)

lemma *outcomes-guard*[*outcomes-spec-monad*]: *outcomes* (*run* (*guard g*) *s*) = (*if g s then* {(*Result* (), *s*)} *else* {})
by (*simp add*: *run-guard*)

lemma *reaches-guard*[*simp*]: *reaches* (*guard g*) *s r t* ⟷ (*g s* ∧ *r* = *Result* () ∧ *t* = *s*)
by (*simp add*: *reaches-def outcomes-guard*)

16.11.27 *assert-opt*

lemma *succeeds-assert-opt*[*simp*]: *succeeds* (*assert-opt x*) *s* ⟷ (∃ *v*. *x* = *Some v*)

by (*simp add*: *assert-opt-def split*: *option.splits*)

lemma *outcomes-assert-opt*[*simp*]:
outcomes (*run* (*assert-opt x*) *s*) = (*case x of Some v* ⇒ {(*Result v*, *s*)} | *None* ⇒ {})
by (*simp split*: *option.splits*)

lemma *reaches-assert-opt*[*simp*]: *reaches* (*assert-opt x*) *s r t* ⟷ (∃ *v*. *x* = *Some v* ∧ *r* = *Result v* ∧ *t* = *s*)
by (*simp add*: *reaches-def split*: *option.splits*)

16.11.28 *gets-the*

lemma *succeeds-gets-the*[*simp*]: *succeeds* (*gets-the f*) *s* ⟷ (∃ *v*. *f s* = *Some v*)
by (*simp add*: *gets-the-def succeeds-bind*)

lemma *outcomes-gets-the*[*simp*]:
outcomes (*run* (*gets-the f*) *s*) = (*case f s of Some v* ⇒ {(*Result v*, *s*)} | *None* ⇒ {})
by (*simp split*: *option.splits*)

lemma *reaches-gets-the*[*simp*]: *reaches* (*gets-the f*) *s r t* ⟷ (∃ *v*. *f s* = *Some v* ∧ *r* = *Result v* ∧ *t* = *s*)
by (*simp add*: *reaches-def split*: *option.splits*)

16.11.29 *modify*

lemma *outcomes-run-modify*[*outcomes-spec-monad*]: *outcomes* (*run* (*modify f*) *s*) = {(*Result* (), *f s*)}
by *simp*

lemma *succeeds-modify*[*simp*]: *succeeds* (*modify f*) *s*
by (*simp add*: *modify-def succeeds-bind*)

lemma *reaches-modify*[*simp*]: *reaches* (*modify f*) *s r t* ⟷ (*r* = *Result* () ∧ *t* = *f s*)
by (*auto simp add*: *modify-def reaches-bind succeeds-bind*)

16.11.30 condition

lemma *outcomes-condition*:
outcomes (*run* (*condition c f g*) *s*) = (*if c s then outcomes* (*run f s*) *else outcomes* (*run g s*))
proof (*cases succeeds* (*condition c f g*) *s*)
case *True*
then show *?thesis* **by** (*auto simp add*: *condition-def outcomes-bind-succeeds outcomes-spec-monad*
split: *exception-or-result-splits*)
next
case *False*

then show *?thesis*
by (*simp add*: *condition-def run-bind*)
qed

lemma *succeeds-condition-iff*:
succeeds (*condition c f g*) *s* ⟷ *succeeds* (*if c s then f else g*) *s*
by (*simp add*: *condition-def succeeds-bind*)

lemma *reaches-condition-iff*:
reaches (*condition c f g*) *s r′ s′* ⟷ *reaches* (*if c s then f else g*) *s r′ s′*
by (*simp add*: *reaches-def outcomes-condition*)

lemma *reaches-condition-True*[*simp*]:
c s ⟹ *reaches* (*condition c f g*) *s r′ s′* ⟷ *reaches f s r′ s′*
by (*simp add*: *reaches-condition-iff*)

lemma *reaches-condition-False*[*simp*]:
¬ *c s* ⟹ *reaches* (*condition c f g*) *s r′ s′* ⟷ *reaches g s r′ s′*
by (*simp add*: *reaches-condition-iff*)

lemma *succeeds-condition-True*[*simp*]:
c s ⟹ *succeeds* (*condition c f g*) *s* ⟷ *succeeds f s*
by (*simp add*: *succeeds-condition-iff*)

lemma *succeeds-condition-False*[*simp*]:
¬(*c s*) ⟹ *succeeds* (*condition c f g*) *s* ⟷ *succeeds g s*
by (*simp add*: *succeeds-condition-iff*)

16.11.31 *when*

lemma *succeeds-when*: *succeeds* (*when c f*) *s* ⟷ ¬*c* ∨ *succeeds f s*
unfolding *when-def* **by** (*simp add*: *succeeds-condition-iff*)

lemma *outcomes-when*[*outcomes-spec-monad*]:
outcomes (*run* (*when c f*) *s*) = (*if c then outcomes* (*run f s*) *else* {(*Result* (), *s*)})
by (*simp add*: *run-when*)

lemma *reaches-when-iff*:
reaches (*when c f*) *s r′ s′* ⟷ (*if c then reaches f s r′ s′ else* (*r′* = *Result* () ∧ *s′* = *s*))
by (*simp add*: *reaches-def outcomes-when*)

16.11.32 While

lemma *reaches-whileLoop-cond-false*:
reaches (*whileLoop C B r*) *s* (*Result r′*) *s′* ⟹ ¬ *C r′ s′*
using *runs-to-partial-whileLoop-cond-false*[*of C B r s*]
by (*auto simp*: *reaches-succeeds runs-to-partial-def-old*)

lemma *bind-post-state-cong*:

$(\bigwedge r\ .\ r \in outcomes\ x \Longrightarrow g\ r = g'\ r) \Longrightarrow bind\text{-}post\text{-}state\ x\ g = bind\text{-}post\text{-}state\ x\ g'$
by (*cases x*) *simp-all*

16.11.33 *map-value*

lemma *succeeds-map-value*[*simp*]: *succeeds* (*map-value f g*) *s* $\longleftrightarrow$ *succeeds g s*
by (*simp add*: *succeeds-def run-map-value top-post-state-def bot-post-state-def split*: *post-state.splits*)

lemma *outcomes-map-value*[*outcomes-spec-monad*]:
outcomes (*run* (*map-value f g*) *s*) = ((λ(*v*, *s*). (*f v*, *s*)) ‘ (*outcomes* (*run g s*)))
by (*auto simp add*: *always-progress-def run-map-value top-post-state-def bot-post-state-def split*: *post-state.splits*)

lemma *reaches-map-value*: *reaches* (*map-value f g*) *s r t* $\longleftrightarrow$ (∃ *r'*. *reaches g s r'* *t* ∧ *r* = *f r'*)
by (*auto simp add*: *reaches-def outcomes-map-value*)

16.11.34 *liftE*

lemma *succeeds-liftE*[*simp*]: *succeeds* (*liftE f*) *s* $\longleftrightarrow$ *succeeds f s*
by (*simp add*: *liftE-def*)

lemma *outcomes-liftE*[*outcomes-spec-monad*]:
outcomes (*run* (*liftE f*) *s*) =
((λ(*v*, *s*). (*map-exception-or-result* (λ*x*. *undefined*) *id v*, *s*)) ‘ (*outcomes* (*run f s*)))
by (*simp add*: *liftE-def outcomes-spec-monad*)

lemma *reaches-liftE*: *reaches* (*liftE f*) *s r t* $\longleftrightarrow$ (∃ *r'*. *reaches f s* (*Result r'*) *t* ∧ *r* = *Result r'*)
by (*auto simp add*: *liftE-def reaches-map-value*)

lemma *bind-cong1*:
fixes *f*::('*a*, '*s*) *res-monad*
shows ⟦ *f* = *f'*; ⋀*v s s'*. *reaches f s* (*Result v*) *s'* ⟹ *g v* = *g' v* ⟧ ⟹ *f* >>= *g* = *f'* >>= *g'*
apply (*rule spec-monad-ext*)
apply (*rule run-bind-reaches-cong*)
apply *auto*
done

lemma *bind-liftE-cong1*:
fixes *f*::('*a*, '*s*) *res-monad*
shows ⟦ *f* = *f'*; ⋀*v s s'*. *reaches f s* (*Result v*) *s'* ⟹ *g v* = *g' v* ⟧ ⟹
(*liftE f*) >>= *g* = (*liftE f'*) >>= *g'*
apply (*rule spec-monad-ext*)
apply (*rule run-bind-reaches-cong*)
apply (*auto simp add*: *reaches-liftE*)

done

16.11.35 *try*

lemma *succeeds-try*[*simp*]: *succeeds* (*try f*) *s* ⟷ *succeeds f s*
by (*simp add*: *try-def*)

lemma *outcomes-try*[*outcomes-spec-monad*]:
outcomes (*run* (*try f*) *s*) = ((λ(*v*, *s*). (*unnest-exn v*, *s*)) ‘ (*outcomes* (*run f s*)))
by (*simp add*: *try-def outcomes-spec-monad*)

lemma *reaches-try*: *reaches* (*try f*) *s r t* ⟷ (∃ *r'*. *reaches f s r' t* ∧ *r* = *unnest-exn r'*)
by (*simp add*: *try-def reaches-map-value*)

16.11.36 *finally*

lemma *succeeds-finally*[*simp*]: *succeeds* (*finally f*) *s* ⟷ *succeeds f s*
by (*simp add*: *finally-def*)

lemma *outcomes-finally*[*outcomes-spec-monad*]:
outcomes (*run* (*finally f*) *s*) = ((λ(*v*, *s*). (*unite v*, *s*)) ‘ (*outcomes* (*run f s*)))
by (*simp add*: *finally-def outcomes-spec-monad*)

lemma *reaches-finally*: *reaches* (*finally f*) *s r t* ⟷ (∃ *r'*. *reaches f s r' t* ∧ *r* = *unite r'*)
by (*simp add*: *finally-def reaches-map-value*)

16.11.37 (<*catch*>)

lemma *succeeds-catch*: *succeeds* (*f* <*catch*> *h*) *s* ⟷
(*succeeds f s* ∧
(∀ *x s'*. *reaches f s x s'* ⟶ (*case x of Exn e* ⇒ *succeeds* (*h e*) *s'* | *Result v* ⇒ *True*)))
unfolding *catch-def*
by (*fastforce simp add*: *succeeds-bind-handle default-option-def Exn-def split*: *exception-or-result-splits xval-splits*)

lemma *outcomes-catch-succeeds*: *succeeds* (*f* <*catch*> *h*) *s* ⟹
outcomes (*run* (*f* <*catch*> *h*) *s*) =
⋃ ((λ(*r*, *s'*). *case r of Exn e* => *outcomes* (*run* (*h e*) *s'*) | *Result v* ⇒ {(*Result v*, *s'*)})
‘ (*outcomes* (*run f s*)))
unfolding *catch-def*
by (*auto simp add*: *outcomes-bind-handle-succeeds default-option-def Exn-def split*: *prod.splits exception-or-result-splits xval-splits*)
(*metis Exn-def Exn-neq-Result the-Exception-simp fst-conv option.sel snd-conv Pair-inject Result-eq-Result*)+

lemma *reaches-catch*: *reaches* (*f* <*catch*> *h*) *s r t* ⟷
(*succeeds* (*f* <*catch*> *h*) *s* ∧
(∃ *r' s'*. *reaches f s r' s'* ∧
(*case r' of*
Exn e ⇒ *reaches* (*h e*) *s' r t*
| *Result v* ⇒ *r* = *Result v* ∧ *t* = *s'*)))
unfolding *catch-def*
by (*auto simp add*: *reaches-bind-handle default-option-def Exn-def split*: *prod.splits exception-or-result-splits xval-splits*)
(*metis option.sel*)+

16.11.38 *check*

lemma *succeeds-check*[*simp*]: *succeeds* (*check e p*) *s*
by (*simp add*: *succeeds-def run-check top-post-state-def*)

lemma *outcomes-check*[*outcomes-spec-monad*]: *outcomes* (*run* (*check e p*) *s*) =
(*if* (∃ *x*. *p s x*) *then* ((λ*x*. (*x*, *s*)) ‘ (*Result* ‘ {*x*. *p s x*})) *else* {(*Exn e*, *s*)})
by (*simp add*: *run-check*)

lemma *reaches-check*: *reaches* (*check e p*) *s r t* ⟷
(*t* = *s*) ∧
(*if* (∃ *x*. *p s x*) *then* (∃ *x*. *r* = *Result x* ∧ *p s x*) *else r* = *Exn e*)
by (*auto simp add*: *reaches-def outcomes-check*)

lemma *refines-bind-ok*:
succeeds g t ⟹ *reaches g t* (*Result a*) *t'* ⟹
refines f (*h a*) *s t' R* ⟹
refines f (*g* >>= *h*) *s t R*
by (*fastforce simp*: *refines-iff succeeds-bind reaches-bind*)

16.11.39 *ignoreE*

lemma *succeeds-ignoreE*[*simp*]:
succeeds (*ignoreE f*) *s* ⟷ (*succeeds f s*)
unfolding *ignoreE-def*
by (*auto simp add*: *succeeds-catch split*: *xval-splits*)

lemma *outcomes-ignoreE-succeeds*: *succeeds f s* ⟹ *outcomes* (*run* (*ignoreE f*) *s*) =
⋃ ((λ(*r*, *s'*). *case r of Exn e* => {} | *Result v* ⇒ {(*Result v*, *s'*)})
‘ (*outcomes* (*run f s*)))
apply (*subst* (*asm*) *succeeds-ignoreE* [*symmetric*])
by (*simp add*: *ignoreE-def outcomes-catch-succeeds*)

lemma *reaches-ignoreE*:
reaches (*ignoreE f*) *s r t* ⟷ (*succeeds f s*) ∧ (∃ *v*. *r* = *Result v* ∧ *reaches f s* (*Result v*) *t*)
apply (*simp add*: *reaches-catch succeeds-catch ignoreE-def*)
apply (*fastforce split*: *xval-splits*)

done

16.11.40 *bind-exception-or-result*

lemma *succeeds-bind-exception-or-result*:
succeeds (*bind-exception-or-result f g*) *s* ⟷
succeeds f s ∧ (∀ *v s′*. *reaches f s v s′* ⟶ *succeeds* (*g v*) *s′*)
by (*cases run f s*;
force simp: *succeeds-def bind-exception-or-result.rep-eq*
bind-post-state-eq-top top-post-state-def reaches-def)

lemma *reaches-bind-exception-or-result*:
reaches (*bind-exception-or-result f g*) *s r t* ⟷
(*succeeds* (*bind-exception-or-result f g*) *s* ∧
(∃ *r′ s′*. *reaches f s r′ s′* ∧ *reaches* (*g r′*) *s′ r t*))
apply (*cases run f s*)
apply (*simp-all add*: *reaches-def succeeds-def bind-exception-or-result.rep-eq*)
subgoal for *X*
apply (*cases* ∃ *x*∈*X*. *case x of* (*v*, *x*) ⇒ *run* (*g v*) *x* = ⊤)
apply (*simp-all add*: *Sup-post-state-def image-iff split-beta′ Bex-def Ball-def eq-commute*
flip: *top-post-state-def*)
by (*metis outcomes.simps*(*2*) *post-state.exhaust top-post-state-def*)
done

lemma *refines-bind-exception-or-result-strong*:
assumes *f*: *refines f f′ s s′ Q*
assumes *g*: ⋀*r t r′ t′*. *Q* (*r*, *t*) (*r′*, *t′*) ⟹ *reaches f s r t* ⟹ *reaches f′ s′ r′ t′* ⟹ *refines* (*g r*) (*g′ r′*) *t t′ R*
shows *refines* (*bind-exception-or-result f g*) (*bind-exception-or-result f′ g′*) *s s′ R*
using *refines-bind-exception-or-result refines-mono f g*
by (*smt* (*verit*, *ccfv-threshold*) *case-prod-conv refines-strengthen′*)

16.11.41 *bind-finally*

lemma *succeeds-bind-finally*:
succeeds (*bind-finally f g*) *s* = (*succeeds f s* ∧ (∀ *v s′*. *reaches f s v s′* ⟶ *succeeds* (*g v*) *s′*))
by (*rule succeeds-bind-exception-or-result*)

lemma *reaches-bind-finally*: *reaches* (*bind-finally f g*) *s r t* ⟷ (*succeeds* (*bind-finally f g*) *s* ∧ (∃ *r′ s′*. *reaches f s r′ s′* ∧ *reaches* (*g r′*) *s′ r t*))
by (*rule reaches-bind-exception-or-result*)

16.11.42 *run-bind*

lemma *succeeds-run-bind*:
succeeds (*run-bind f t g*) *s* ⟷ *succeeds f t* ∧ (∀ *r t′*. *reaches f t r t′* ⟶ *succeeds* (*g r t′*) *s*)
by (*cases run f t*)

(*force simp*: *succeeds-def run-run-bind top-post-state-def reaches-def*
split: *post-state.splits*)+

lemma *reaches-run-bind*: *reaches* (*run-bind f t g*) *s r s'* ⟷
(*succeeds* (*run-bind f t g*) *s*) ∧
(∃ *r' t'*. *reaches f t r' t'* ∧ *reaches* (*g r' t'*) *s r s'*)
apply (*cases run f t*)
apply (*simp-all add*: *reaches-def run-run-bind succeeds-run-bind*)
apply (*simp add*: *succeeds-def split-beta'*)
subgoal for *A*
apply (*cases* ∀ *r t'*. (*r*, *t'*) ∈ *A* ⟶ *run* (*g r t'*) *s* ≠ ⊤)
subgoal by (*subst outcomes-Sup*; *force simp flip*: *top-post-state-def*)
subgoal
apply (*subst Sup-eq-Failure*[*THEN iffD2*])
apply (*force simp flip*: *top-post-state-def*)
apply *auto*
done
done
done

lemma *runs-to-partial-reaches*: *f* ⋅ *s* ?{⦇ *reaches f s* ⦈}
apply (*cases run f s*)
apply (*auto simp add*: *runs-to-partial-def reaches-def*)
done

lemma *refines-strengthen-reaches*:
assumes *f-g*: *refines f g s t R*
assumes *reach*: (⋀*x s' y t'*. *R* (*x*, *s'*) (*y*, *t'*) ⟹ *reaches f s x s'* ⟹ *reaches g t y t'* ⟹ *Q* (*x*, *s'*) (*y*, *t'*))
shows *refines f g s t Q*
apply (*rule refines-strengthen* [*OF f-g runs-to-partial-reaches runs-to-partial-reaches*])
using *reach*
by *blast*

lemma *refines-bind-bind-strong'*:
assumes *f*: *refines f f' s s' Q*
assumes *Ex-Ex*: (⋀*e e' t t'*.
Q (*Exception e*, *t*) (*Exception e'*, *t'*) ⟹
e ≠ *default* ⟹
e' ≠ *default* ⟹
reaches f s (*Exception e*) *t* ⟹
reaches f' s' (*Exception e'*) *t'* ⟹
R (*Exception e*, *t*) (*Exception e'*, *t'*))
assumes *Res-Ex*: (⋀*e v' t t'*.
Q (*Exception e*, *t*) (*Result v'*, *t'*) ⟹
e ≠ *default* ⟹

reaches f s (Exception e) t ⟹
reaches f′ s′ (Result v′) t′ ⟹
refines (yield (Exception e)) (g′ v′) t t′ R)
assumes *Ex-Ex*: (⋀*v e′ t t′*.
Q (Result v, t) (Exception e′, t′) ⟹
e′ ≠ default ⟹
reaches f s (Result v) t ⟹
reaches f′ s′ (Exception e′) t′ ⟹
refines (g v) (yield (Exception e′)) t t′ R)
assumes *Res-Res*: (⋀*v v′ t t′*.
Q (Result v, t) (Result v′, t′) ⟹
reaches f s (Result v) t ⟹
reaches f′ s′ (Result v′) t′ ⟹
refines (g v) (g′ v′) t t′ R)
shows *refines (f ≫= g) (f′ ≫= g′) s s′ R*
apply (*rule refines-bind′*)
apply (*rule refines-strengthen-reaches* [*OF f*])
apply (*auto intro*: *assms*)
done

lemma *rel-spec-monad-bind-strong*:
assumes *f-f′*: *rel-spec-monad S P f f′*
assumes *Ex-Ex*: ⋀*e e′ s s′ t t′*. *S s s′* ⟹ *S t t′* ⟹ *P (Exception e) (Exception e′)* ⟹ *e ≠ default* ⟹ *e′ ≠ default* ⟹
reaches f s (Exception e) t ⟹ *reaches f′ s′ (Exception e′) t′* ⟹
Q (Exception e) (Exception e′)
assumes *Ex-Res*: ⋀*e v′ s s′ t t′*. *S s s′* ⟹ *S t t′* ⟹ *P (Exception e) (Result v′)* ⟹ *e ≠ default* ⟹
reaches f s (Exception e) t ⟹ *reaches f′ s′ (Result v′) t′* ⟹
rel-spec-monad S Q (yield (Exception e)) (g′ v′)
assumes *Res-Ex*: ⋀*v e′ s s′ t t′*. *S s s′* ⟹ *S t t′* ⟹ *P (Result v) (Exception e′)* ⟹ *e′ ≠ default* ⟹
reaches f s (Result v) t ⟹ *reaches f′ s′ (Exception e′) t′* ⟹
rel-spec-monad S Q (g v) (yield (Exception e′))
assumes *Res-Res*: ⋀*v v′ s s′ t t′*. *S s s′* ⟹ *S t t′* ⟹ *P (Result v) (Result v′)* ⟹
reaches f s (Result v) t ⟹ *reaches f′ s′ (Result v′) t′* ⟹
rel-spec-monad S Q (g v) (g′ v′)
shows *rel-spec-monad S Q (f ≫= g) (f′ ≫= g′)*
apply (*simp add*: *rel-spec-monad-iff-refines*)
apply (*clarify*, *intro conjI*)
subgoal for *s t*
apply (*rule refines-bind-bind-strong′* [**where** *Q*=(*rel-prod P S*)])
subgoal using *f-f′* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
subgoal
using *Ex-Ex* **by** *auto*
subgoal
using *Ex-Res* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)

subgoal
using *Res-Ex* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
subgoal
using *Res-Res* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
done
subgoal for *s t*
apply (*rule refines-bind-bind-strong'* [**where** *Q*=(*rel-prod* P^{-1-1} S^{-1-1})])
subgoal using *f-f'* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
subgoal
using *Ex-Ex* **by** *auto*
subgoal
using *Res-Ex* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
subgoal
using *Ex-Res* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
subgoal
using *Res-Res* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
done
done

lemma *rel-spec-monad-bind-strong-exn*:
assumes *f-f'*: *rel-spec-monad S P f f'*
assumes *Ex-Ex*: $\bigwedge e\ e'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Exn\ e)\ (Exn\ e') \Longrightarrow$
reaches f s (*Exn e*) *t* $\Longrightarrow$ *reaches f' s'* (*Exn e'*) *t'* $\Longrightarrow$
Q (*Exn e*) (*Exn e'*)
assumes *Ex-Res*: $\bigwedge e\ v'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Exn\ e)\ (Result\ v') \Longrightarrow$
reaches f s (*Exn e*) *t* $\Longrightarrow$ *reaches f' s'* (*Result v'*) *t'* $\Longrightarrow$
rel-spec-monad S Q (*throw e*) (*g' v'*)
assumes *Res-Ex*: $\bigwedge v\ e'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Result\ v)\ (Exn\ e') \Longrightarrow$
reaches f s (*Result v*) *t* $\Longrightarrow$ *reaches f' s'* (*Exn e'*) *t'* $\Longrightarrow$
rel-spec-monad S Q (*g v*) (*throw e'*)
assumes *Res-Res*: $\bigwedge v\ v'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Result\ v)\ (Result\ v')$
$\Longrightarrow$
reaches f s (*Result v*) *t* $\Longrightarrow$ *reaches f' s'* (*Result v'*) *t'* $\Longrightarrow$
rel-spec-monad S Q (*g v*) (*g' v'*)
shows *rel-spec-monad S Q* (*f* $\gg\!=$ *g*) (*f'* $\gg\!=$ *g'*)
apply (*rule rel-spec-monad-bind-strong* [*OF f-f'*])
subgoal using *Ex-Ex* **by** (*auto simp add*: *Exn-def default-option-def*)
subgoal using *Ex-Res* **by** (*auto simp add*: *Exn-def default-option-def*)
subgoal using *Res-Ex* **by** (*auto simp add*: *Exn-def default-option-def*)
subgoal using *Res-Res* **by** (*auto*)
done

lemma *rel-spec-monad-rel-xval-bind-strong*:
assumes *f-f'*: *rel-spec-monad S* (*rel-xval L P*) *f f'*
assumes *Res-Res*: $\bigwedge v\ v'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ v\ v' \Longrightarrow$
reaches f s (*Result v*) *t* $\Longrightarrow$ *reaches f' s'* (*Result v'*) *t'* $\Longrightarrow$
rel-spec-monad S (*rel-xval L R*) (*g v*) (*g' v'*)
shows *rel-spec-monad S* (*rel-xval L R*) (*f* $\gg\!=$ *g*) (*f'* $\gg\!=$ *g'*)

apply (*rule rel-spec-monad-bind-strong-exn* [*OF f-f'*])
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *Res-Res* **by** *auto*
done

lemma *rel-spec-monad-bind-exception-or-result-strong*:
assumes *f-f'*: *rel-spec-monad S P f f'*
assumes *g-g'*: $\bigwedge r\ r'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ r\ r' \Longrightarrow$
reaches f s r t $\Longrightarrow$ *reaches f' s' r' t'* $\Longrightarrow$
rel-spec-monad S Q (*g r*) (*g' r'*)
shows *rel-spec-monad S Q* (*bind-exception-or-result f g*) (*bind-exception-or-result f' g'*)
apply (*simp add*: *rel-spec-monad-iff-refines*)
apply (*clarify*, *intro conjI*)
subgoal for *s t*
apply (*rule refines-bind-exception-or-result-strong* [**where** *Q*=*rel-prod P S*])
subgoal
using *f-f'* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
subgoal
using *g-g'* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
done
subgoal for *s t*
apply (*rule refines-bind-exception-or-result-strong* [**where** *Q*=*rel-prod* P^{-1-1} S^{-1-1}])
subgoal
using *f-f'* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
subgoal
using *g-g'* **by** (*auto simp add*: *rel-spec-monad-iff-refines*)
done
done

end

theory *Simp-Trace*
imports *AutoCorres-Utils*
begin

ATTENTION: to activate these methods use the following line:
setup ‹*Raw-Simplifier.set-trace-ops Simp-Trace.trace-ops*›

Provide a tactic wrapper to activate simplifier tracing and produce a statistic how many conditional rules were tried for how long. Also provides a shorthand for simp trace activation by adding *T* to the method name: *simpT simp-allT autoT*

ML ‹

```
structure Simp-Trace =
struct

type cumulative-statistics =
  {
    cumulative-time : Time.time,
    calls : int
  }

type call-statistics =
  { successes : cumulative-statistics, failures : cumulative-statistics }

type trace-statistics =
  {
    conditional-rules : call-statistics Thm-Name.Table.table,
    simprocs : call-statistics Symtab.table,
    rules : int Thm-Name.Table.table,
    steps : int,
    max-steps : int,
    depth : int,
    max-depth : int,
    do-trace : bool,
    in-simproc : bool
  }

structure Data = Proof-Data
(
  type T = trace-statistics Synchronized.var option
  fun init - = NONE
)

val trace-data : Proof.context -> trace-statistics Synchronized.var option = Data.get

fun initial n do-trace : trace-statistics =
  {
    conditional-rules = Thm-Name.Table.empty,
    simprocs = Symtab.empty,
    rules = Thm-Name.Table.empty,
    steps = 0,
    max-steps = n,
    depth = 0,
    max-depth = 0,
    do-trace = do-trace,
    in-simproc = false
  }

fun activate n do-trace = Data.put (SOME (Synchronized.var trace-data (initial n
do-trace)))
```

```
fun call-statistics-ord (x, y:('a * call-statistics)) =
  (x, y) |> apply2 (snd #> #failures #> #cumulative-time) |> Time.compare


fun print-cumulative-statistics txt { cumulative-time = t, calls = n} =
  if n = 0 then no ^ txt else txt ^ ":  ^
  Time.toString t ^ "  /  " ^ string-of-int n ^ " =  "^
    Real.fmt (StringCvt.FIX (SOME 3)) (Time.toReal t / Real.fromInt n)

fun print ({conditional-rules, rules, simprocs, steps, max-depth, ...} : trace-statistics)
=
  let
    val - = writeln (=== SIMP STATISTICS ( ^
      string-of-int steps ^ " steps, " ^ string-of-int max-depth ^ " max depth) ===)
    val - = writeln "Conditional Rules (sorted in failure time):"
    val - = conditional-rules |> Thm-Name.Table.dest
      |> sort (call-statistics-ord #> rev-order)
      |> app (fn (name, cs) =>
       writeln ("  " ^ Thm-Name.print name ^ ":  " ^ print-cumulative-statistics failures
(#failures cs) ^
          ( " " ^ print-cumulative-statistics successes (#successes cs) ^ " ")))
    val - = writeln "Rules:"
    val - = rules |> Thm-Name.Table.dest
      |> sort (make-ord (fn ((-, c1), (-, c2)) => c2 < c1))
     |> app (fn (name, c) => writeln ("  " ^ Thm-Name.print name ^ ":  " ^ string-of-int
c))
    val - = writeln "Simprocs (sorted in failure time):"
    val - = simprocs |> Symtab.dest
      |> sort (call-statistics-ord #> rev-order)
      |> app (fn (name, cs) =>
       writeln ("  " ^ name ^ ":  " ^ print-cumulative-statistics failures (#failures cs) ^
          ( " " ^ print-cumulative-statistics successes (#successes cs) ^ " ")))
    val - = writeln (======)
  in
    ()
  end

fun map-success f ({ successes, failures } : call-statistics) : call-statistics =
  { successes = f successes, failures = failures }

fun map-failure f ({ successes, failures } : call-statistics) : call-statistics =
  { successes = successes, failures = f failures }

fun incr-cumulative t ({ cumulative-time, calls } : cumulative-statistics) =
  { cumulative-time = Time.+ (t, cumulative-time), calls = calls + 1 }

fun incr-call success t =
  (if success then map-success else map-failure) (incr-cumulative t)
```

```
val zero-cumulative = { cumulative-time = Time.zeroTime, calls = 0 }

val zero-calls = { successes = zero-cumulative, failures = zero-cumulative }

fun map-depth f
  ({ conditional-rules, simprocs, rules, steps, depth, max-steps, max-depth, do-trace,
in-simproc} :
    trace-statistics) :
  trace-statistics =
let
  val d = f depth
in
  { conditional-rules = conditional-rules,
    simprocs = simprocs,
    rules = rules,
    steps = steps,
    depth = d,
    max-depth = if max-depth < d then d else max-depth,
    max-steps = max-steps,
    do-trace = do-trace,
    in-simproc = in-simproc }
end

fun map-in-simproc f
  ({ conditional-rules, simprocs, rules, steps, depth, max-steps, max-depth, do-trace,
in-simproc} :
    trace-statistics) :
  trace-statistics =
{
  conditional-rules = conditional-rules,
  simprocs = simprocs,
  rules = rules,
  steps = steps,
  depth = depth,
  max-steps = max-steps,
  max-depth = max-depth,
  do-trace = do-trace,
  in-simproc = f in-simproc
}

fun trace-apply var {unconditional, rrule, ...} ctxt cont =
  let
    val d = Synchronized.value var
   val - = #max-steps d <= #steps d andalso error (simp trace: max steps reaches)
    fun store success time = Synchronized.change var (fn
        ({ conditional-rules, simprocs, rules, steps, depth, max-steps, max-depth,
do-trace,
        in-simproc } : trace-statistics) =>
```

```
      {
        conditional-rules = (if unconditional then conditional-rules else
          conditional-rules |>
          Thm-Name.Table.map-default (#name rrule, zero-calls) (incr-call success
(#cpu time))),
        simprocs = simprocs,
        rules = rules |> Thm-Name.Table.map-default (#name rrule, 0) (fn c =>
c + 1),
        steps = steps + 1,
        max-steps = max-steps,
        depth = depth,
        max-depth = max-depth,
        do-trace = do-trace,
        in-simproc = in-simproc
      }
    )
  in
    Timing.timing (Exn.capture cont) ctxt
    |> (fn (t, x) => (store (Exn.is-res x) t; Exn.release x))
  end

fun trace-simproc var name ctxt cont =
  let
    val d = Synchronized.value var
   val - = #max-steps d <= #steps d andalso error (simp trace: max steps reaches)
    fun store success time = Synchronized.change var (fn
        ({ conditional-rules, simprocs, rules, steps, depth, max-steps, max-depth,
do-trace, in-simproc } : trace-statistics) =>
      {
        conditional-rules = conditional-rules,
        simprocs = simprocs |>
          Symtab.map-default (name, zero-calls) (incr-call success (#cpu time)),
        rules = rules,
        steps = steps + 1,
        max-steps = max-steps,
        depth = depth,
        max-depth = max-depth,
        do-trace = do-trace,
        in-simproc = false
      }
    )
    fun is-success (Exn.Res (SOME -)) = true
      | is-success - = false
    fun set-in-simproc f x =
      (Synchronized.change var (map-in-simproc (K f)); x)
  in
    Timing.timing (Exn.capture
      (set-in-simproc true #> cont)) ctxt
    |> (fn (t, x) => (store (is-success x) t; Exn.release x))
```

```
  end

fun increase-depth var ctxt =
  (Synchronized.change var (map-depth (fn d => d + 1)); ctxt)

fun check-trace-data ctxt =
  case trace-data ctxt of SOME var =>
    let
      val d = Synchronized.value var
    in
      if #in-simproc d then NONE else SOME var
    end
  | NONE => NONE

val trace-ops : Raw-Simplifier.trace-ops = {
  trace-invoke = fn - => fn ctxt =>
    (case check-trace-data ctxt of SOME var => increase-depth var ctxt
      | NONE => ctxt),
  trace-rrule = fn rule => fn ctxt => fn cont =>
    case check-trace-data ctxt of
      SOME var => trace-apply var rule ctxt cont
    | NONE => cont ctxt,
  trace-simproc = fn {name, ...} => fn ctxt => fn cont =>
    case check-trace-data ctxt of
      SOME var => trace-simproc var name ctxt cont
    | NONE => cont ctxt
}

fun wrapper n do-trace tac ctxt st =
  let
    val ctxt' = ctxt
      |> activate n do-trace
      |> Config.put Raw-Simplifier.simp-trace do-trace
      |> Config.put Raw-Simplifier.simp-trace-depth-limit 4
    val res = Exn.capture (fn () => (tac ctxt' st |> Seq.pull)) ()
    val - = print (Synchronized.value (the (trace-data ctxt')))
  in
    case Exn.release res of
      SOME (x, -) => Seq.cons x (Seq.make (fn () =>
        error (stat simp tracer does not allow back ^ Position.here here)))
    | NONE => Seq.empty
  end

fun method-wrapper (n : int option) (do-trace : bool) (m : Proof.context -> Method.method)
ctxt :
  Method.method =
  fn thms => Method.RUNTIME
    (wrapper (the-default 1000 n) do-trace
      (fn ctxt => fn (-, thm) => m ctxt thms (ctxt, thm)) ctxt)
```

```
val no-asmN = no-asm;
val no-asm-useN = no-asm-use;
val no-asm-simpN = no-asm-simp;
val asm-lrN = asm-lr;

val simp-options =
 (Args.parens (Scan.option Parse.nat -- Args.$$$ no-asmN) >> apsnd (K simp-tac)
||
  Args.parens (Scan.option Parse.nat -- Args.$$$ no-asm-simpN) >> apsnd (K
asm-simp-tac) ||
  Args.parens (Scan.option Parse.nat -- Args.$$$ no-asm-useN) >> apsnd (K
full-simp-tac) ||
 Args.parens (Scan.option Parse.nat -- Args.$$$ asm-lrN) >> apsnd (K asm-lr-simp-tac)
||
  Args.parens Parse.nat >> (fn n => (SOME n, asm-full-simp-tac)) ||
  Scan.succeed (NONE, asm-full-simp-tac));

val no-traceN = no-trace;

val wrapper : (Proof.context -> Method.method) context-parser =
  (Scan.lift (
    Args.parens Parse.nat >> (fn n => (SOME n, true)) ||
    Args.parens (Parse.nat -- Args.$$$ no-traceN) >> (fn (n, -) => (SOME n,
false)) ||
    Args.parens (Args.$$$ no-traceN) >> (fn - => (NONE, false)) ||
    Scan.succeed (NONE, true)) --
  Method.text-closure)
  >> (fn ((n, do-trace), text) =>
    method-wrapper n do-trace (fn ctxt => Method.evaluate-runtime text ctxt))

fun simpT-method meth =
  Scan.lift simp-options --|
    Method.sections Simplifier.simp-modifiers' >>
    (fn (n, tac) => method-wrapper n true (fn ctxt => METHOD (fn facts =>
meth ctxt tac facts)))

val simp-wrapper =
  simpT-method (fn ctxt => fn tac => fn facts =>
    HEADGOAL (Method.insert-tac ctxt facts THEN'
      (CHANGED-PROP oo tac) ctxt))

val simp-all-wrapper =
  simpT-method (fn ctxt => fn tac => fn facts =>
    ALLGOALS (Method.insert-tac ctxt facts) THEN
      (CHANGED-PROP o PARALLEL-ALLGOALS o tac) ctxt)

val auto-wrapper =
  (Scan.lift (Scan.option (Args.parens Parse.nat)) --
```

```
    Scan.lift (Scan.option (Parse.nat -- Parse.nat))) --|
    Method.sections clasimp-modifiers >>
  (fn (i, NONE) =>
      method-wrapper i true (SIMPLE-METHOD o CHANGED-PROP o auto-tac)
    | (i, SOME (m, n)) =>
      method-wrapper i true (fn ctxt => SIMPLE-METHOD (CHANGED-PROP
(mk-auto-tac ctxt m n))));

end

›

setup ‹Method.setup binding‹stat› Simp-Trace.wrapper
  (Simplifier statisitics: wraps a method call and computes statistics from simplifier
calls. ^
    Needs active Simp-Trace.trace-ops)›

setup ‹Method.setup binding‹simpT› Simp-Trace.simp-wrapper
  (simplification with tracing ^
    Needs active Simp-Trace.trace-ops)›

setup ‹Method.setup binding‹simp-allT› Simp-Trace.simp-all-wrapper
  (simplification on all goals with tracing ^
    Needs active Simp-Trace.trace-ops)›

setup ‹Method.setup binding‹autoT› Simp-Trace.auto-wrapper
  (auto with tracing ^
    Needs active Simp-Trace.trace-ops)›

end
```

Chapter 17

Basic Stuff

theory *AutoCorres-Base*
  imports
  TypHeapLib
  LemmaBucket-C
  CTranslation
  Synthesize
  Cong-Tactic

  Reaches
  Mutual-CCPO-Recursion
  Simp-Trace
begin

definition *THIN* :: *prop* ⇒ *prop* **where** *PROP THIN* (*PROP P*) ≡ *PROP P*

lemma *THIN-I*: *PROP P* ⟹ *PROP THIN* (*PROP P*)
  by (*simp add*: *THIN-def*)

ML-file ‹*utils.ML*›

named-theorems *corres-admissible* **and** *corres-top*
named-theorems *funp-intros* **and** *fun-of-rel-intros*

definition *fun-of-rel*:: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ *bool* **where**
  fun-of-rel r f = (∀ *x y*. *r x y* ⟶ *y* = *f x*)

definition *funp*:: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ *bool* **where**
 funp r = (∃ *f*. *fun-of-rel r f*)

lemma *funp-witness*: *fun-of-rel r f* ⟹ *funp r*
  by (*auto simp add*: *funp-def*)

lemma *funp-to-single-valuedp*: *funp r* ⟹ *single-valuedp r*

by (*auto simp add*: *single-valuedp-def funp-def fun-of-rel-def*)

lemma *fun-of-rel-xval*[*fun-of-rel-intros*]:
fun-of-rel L f-l $\Longrightarrow$ *fun-of-rel R f-r* $\Longrightarrow$ *fun-of-rel* (*rel-xval L R*) (*map-xval f-l f-r*)
by (*auto simp add*: *fun-of-rel-def rel-xval.simps*)

lemma *funp-rel-xval*[*funp-intros*, *corres-admissible*]:
assumes *L*: *funp L*
assumes *R*: *funp R*
shows *funp* (*rel-xval L R*)
proof −
from *L* **obtain** *f-l* **where** *f-l*: $\bigwedge x\ y.\ L\ x\ y \longrightarrow y = f\text{-}l\ x$ **by** (*auto simp add*: *funp-def fun-of-rel-def*)
from *R* **obtain** *f-r* **where** *f-r*: $\bigwedge x\ y.\ R\ x\ y \longrightarrow y = f\text{-}r\ x$ **by** (*auto simp add*: *funp-def fun-of-rel-def*)
define *f* **where** *f* = *map-xval f-l f-r*
{
fix *x y*
have *rel-xval L R x y* $\longrightarrow$ *y* = *f x*
using *f-l f-r*
by (*auto simp add*: *rel-xval.simps f-def*)
}
then show *?thesis*
by (*auto simp add*: *funp-def fun-of-rel-def*)
qed

lemma *fun-of-rel-eq*[*fun-of-rel-intros*]: *fun-of-rel* (=) ($\lambda x.\ x$)
by (*auto simp add*: *fun-of-rel-def*)

lemma *fun-of-rel-bot*[*fun-of-rel-intros*]: *fun-of-rel* (λ- -. *False*) *f*
by (*auto simp add*: *fun-of-rel-def*)

lemma *funp-eq*[*funp-intros*, *corres-admissible*]: *funp* (=)
by (*auto simp add*: *funp-def fun-of-rel-def*)

lemma *admissible-mem*: *ccpo.admissible Inf* ($\geq$) ($\lambda A.\ x \in A$)
by (*auto simp*: *ccpo.admissible-def*)

lemma *funp-rel-prod*[*funp-intros*, *corres-admissible*]:
assumes *L*: *funp L*
assumes *R*: *funp R*
shows *funp* (*rel-prod L R*)
proof −
from *L* **obtain** *f-l* **where** *f-l*: $\bigwedge x\ y.\ L\ x\ y \longrightarrow y = f\text{-}l\ x$ **by** (*auto simp add*: *funp-def fun-of-rel-def*)
from *R* **obtain** *f-r* **where** *f-r*: $\bigwedge x\ y.\ R\ x\ y \longrightarrow y = f\text{-}r\ x$ **by** (*auto simp add*: *funp-def fun-of-rel-def*)
define *f* **where** *f* = *map-prod f-l f-r*
{

```
    fix x y
    have rel-prod L R x y ⟶ y = f x
      using f-l f-r
      by (auto simp add: rel-prod.simps f-def map-prod-def split: prod.splits)
  }
  then show ?thesis
    by (auto simp add: funp-def fun-of-rel-def)
qed

lemma funp-rel-sum[funp-intros, corres-admissible]:
  assumes L: funp L
  assumes R: funp R
  shows funp (rel-sum L R)
proof −
  from L obtain f-l where f-l: ⋀x y. L x y ⟶ y = f-l x by (auto simp add:
funp-def fun-of-rel-def)
  from R obtain f-r where f-r: ⋀x y. R x y ⟶ y = f-r x by (auto simp add:
funp-def fun-of-rel-def)
  define f where f = sum-map f-l f-r
  {
    fix x y
    have rel-sum L R x y ⟶ y = f x
      using f-l f-r
      by (auto simp add: rel-sum.simps f-def)
  }
  then show ?thesis
    by (auto simp add: funp-def fun-of-rel-def)
qed

lemma fun-of-rel-bottom: fun-of-rel ((λ- -. False)) f
  by (auto simp add: fun-of-rel-def)

lemma funp-bottom [funp-intros, corres-admissible]: funp (λ- -. False)
  using fun-of-rel-bottom
  by (metis funp-def)

lemma fun-of-rel-prod[fun-of-rel-intros]:
  fun-of-rel L f-l ⟹ fun-of-rel R f-r ⟹ fun-of-rel (rel-prod L R) (map-prod f-l
f-r)
  by (auto simp add: fun-of-rel-def)

lemma fun-of-rel-sum[fun-of-rel-intros]:
  fun-of-rel L f-l ⟹ fun-of-rel R f-r ⟹ fun-of-rel (rel-sum L R) (sum-map f-l
f-r)
  by (auto simp add: fun-of-rel-def rel-sum.simps)

lemma fun-of-relcompp[fun-of-rel-intros]: fun-of-rel F f ⟹ fun-of-rel G g ⟹
fun-of-rel (F OO G) (g o f)
  by (auto simp add: fun-of-rel-def)
```

lemma *funp-relcompp*[*funp-intros*, *corres-admissible*]: *funp F* $\Longrightarrow$ *funp G* $\Longrightarrow$ *funp* (*F OO G*)
using *fun-of-relcompp*
by (*force simp add*: *funp-def*)

lemma *fun-of-rel-rel-map*[*fun-of-rel-intros*]: *fun-of-rel* (*rel-map f*) *f*
by (*auto simp add*: *fun-of-rel-def rel-map-def*)

lemma *funp-rel-map*[*funp-intros*, *corres-admissible*]: *funp* (*rel-map f*)
using *fun-of-rel-rel-map*
by (*auto simp add*:*funp-def*)

lemma *ccpo-prod-gfp-gfp*:
class.ccpo
(*prod-lub Inf Inf* :: (('*a*::*complete-lattice* $*$ '*b* :: *complete-lattice*) *set* $\Rightarrow$ -))
(*rel-prod* ($\geq$) ($\geq$)) (*mk-less* (*rel-prod* ($\geq$) ($\geq$)))
by (*rule ccpo-rel-prodI ccpo-Inf*)+

lemma *runs-to-partial-top*[*corres-top*]: $\top \cdot s$ *?*$\{\!|$ *Q* $|\!\}$
by (*simp add*: *runs-to-partial-def-old*)

lemma *refines-top*[*corres-top*]: *refines C* $\top$ *s t R*
by (*auto simp add*: *refines-def-old*)

lemma *pred-andE*[*elim*!]: $[\![$ (*A and B*) *x*; $[\![$ *A x*; *B x* $]\!] \Longrightarrow R$ $]\!] \Longrightarrow R$
by(*simp add*:*pred-conj-def*)

lemma *pred-andI*[*intro*!]: $[\![$ *A x*; *B x* $]\!] \Longrightarrow$ (*A and B*) *x*
by(*simp add*:*pred-conj-def*)

definition *measure'* :: ('*a* $\Rightarrow$ '*b*::*wellorder*) => ('*a* $\times$ '*a*) *set*
where *measure'* = (λf. {(*a*, *b*). *f a* < *f b*})

lemma *in-measure'*[*simp*, *code-unfold*]:
((*x*,*y*) : *measure' f*) = (*f x* < *f y*)
by (*simp add*:*measure'-def*)

lemma *wf-measure'* [*iff*]: *wf* (*measure' f*)
apply (*clarsimp simp*: *measure'-def*)
apply (*insert wf-inv-image* [*OF wellorder-class.wf*, **where** *f*=*f*])
apply (*clarsimp simp*: *inv-image-def*)
done

lemma *wf-wellorder-measure*: *wf* {(*a*, *b*). (*M a* :: '*a* :: *wellorder*) < *M b*}
apply (*subgoal-tac wf* (*inv-image* ({(*a*, *b*). *a* < *b*}) *M*))
apply (*clarsimp simp*: *inv-image-def*)

apply (*rule wf-inv-image*)
apply (*rule wellorder-class.wf*)
done

lemma *wf-custom-measure*:
⟦ ⋀*a b*. (*a*, *b*) ∈ *R* ⟹ *f a* < (*f* :: *'a* ⇒ *nat*) *b* ⟧ ⟹ *wf R*
by (*metis in-measure wf-def wf-measure*)

lemma *rel-fun-conversep'*: *rel-fun R Q f g* ⟷ *rel-fun* R^{-1-1} Q^{-1-1} *g f*
unfolding *rel-fun-def* **by** *auto*

end

theory *SimplBucket*
imports *AutoCorres-Base*
begin

lemma *Normal-resultE*:
⟦ Γ ⊢ ⟨*c*, *s*⟩ ⇒ *Normal t'*; ⋀*t*. ⟦ Γ ⊢ ⟨*c*, *Normal t*⟩ ⇒ *Normal t'*; *s* = *Normal t*⟧
⟹ *P* ⟧ ⟹ *P*
by (*metis noAbrupt-startD noFault-startD noStuck-startD xstate.exhaust*)

lemma *exec-While-final-cond'*:
⟦ Γ ⊢ ⟨*b*, *s*⟩ ⇒ *s'*; *b* = *While C B*; *s* = *Normal v*; *s'* = *Normal x* ⟧ ⟹ *x* ∉ *C*
apply (*induct arbitrary*: *v x rule*: *exec.induct*)
apply *simp-all*
apply (*atomize*, *clarsimp*)
apply (*erule exec-elim-cases*, *auto*)
done

lemma *exec-While-final-cond*:
⟦ Γ ⊢ ⟨*While C B*, *s*⟩ ⇒ *Normal s'*⟧ ⟹ *s'* ∉ *C*
apply (*erule Normal-resultE*)
apply (*erule exec-While-final-cond'*, *auto*)
done

lemma *exec-While-final-inv'*:
⟦ Γ ⊢ ⟨*b*, *s*⟩ ⇒ *s'*; *b* = *While C B*; *s* = *Normal v*; *s'* = *Normal x*; *I v*; ⋀*s s'*. ⟦ *I*
s; Γ ⊢ ⟨*B*, *Normal s*⟩ ⇒ *Normal s'* ⟧ ⟹ *I s'* ⟧ ⟹ *I x*
apply *atomize*
apply (*induct arbitrary*: *v x rule*: *exec.induct*)

apply (*clarify* | (*atomize*, *erule Normal-resultE*, *metis*))+
done

lemma *exec-While-final-inv*:
⟦ Γ ⊢ ⟨*While C B*, *Normal s*⟩ ⇒ *Normal s′*; *I s*; ⋀*s s′*. ⟦ *I s*; Γ ⊢ ⟨*B*, *Normal s*⟩ ⇒ *Normal s′* ⟧ ⟹ *I s′* ⟧ ⟹ *I s′*
apply (*erule exec-While-final-inv′*, *auto*)
done

primrec
exceptions-thrown :: (′*a*, ′*p*, ′*e*) *com* ⇒ *bool*
where
exceptions-thrown Skip = *False*
| *exceptions-thrown* (*Seq a b*) = (*exceptions-thrown a* ∨ *exceptions-thrown b*)
| *exceptions-thrown* (*Basic a*) = *False*
| *exceptions-thrown* (*Language.Spec a*) = *False*
| *exceptions-thrown* (*Cond a b c*) = (*exceptions-thrown b* ∨ *exceptions-thrown c*)
| *exceptions-thrown* (*Catch a b*) = (*exceptions-thrown a* ∧ *exceptions-thrown b*)
| *exceptions-thrown* (*While a b*) = *exceptions-thrown b*
| *exceptions-thrown Throw* = *True*
| *exceptions-thrown* (*Call p*) = *True*
| *exceptions-thrown* (*Guard f g a*) = *exceptions-thrown a*
| *exceptions-thrown* (*DynCom a*) = (∃ *s*. *exceptions-thrown* (*a s*))

primrec
exceptions-unresolved :: (′*a*, ′*p*, ′*e*) *com list* ⇒ *bool*
where
exceptions-unresolved [] = *True*
| *exceptions-unresolved* (*x*#*xs*) = (*exceptions-thrown x* ∧ *exceptions-unresolved xs*)

lemma *exceptions-thrown-not-abrupt*:
⟦ Γ ⊢ ⟨*p*, *s*⟩ ⇒ *s′*; ¬ *exceptions-thrown p*; ¬ *isAbr s* ⟧ ⟹ ¬ *isAbr s′*
apply (*induct rule*: *exec.induct*, *auto*)
done

end

theory *CCorresE*
imports
SimplBucket
begin

definition
ccorresE :: ('*t* ⇒ '*s*) ⇒ *bool* ⇒ '*ee set* ⇒ ('*p* ⇒ ('*t*, '*p*, '*ee*) *com option*)
⇒ ('*s* ⇒ *bool*) ⇒ ('*t set*)
⇒ (*unit*, *unit*, '*s*) *exn-monad* ⇒ ('*t*, '*p*, '*ee*) *com* ⇒ *bool*
where
ccorresE st check-term AF Γ *G G'* ≡
λ*m c*. ∀ *s*. *G* (*st s*) ∧ (*s* ∈ *G'*) ∧ *succeeds m* (*st s*) ⟶
((∀ *t*. Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *t* ⟶
(*case t of*
Normal s' ⇒ *reaches m* (*st s*) (*Result* ()) (*st s'*)
| *Abrupt s'* ⇒ *reaches m* (*st s*) (*Exn* ()) (*st s'*)
| *Fault e* ⇒ *e* ∈ *AF*
| - ⇒ *False*))
∧ (*check-term* ⟶ Γ ⊢ *c* ↓ *Normal s*))

lemma *ccorresE-cong*:
⟦ ⋀*s*. *P s* = *P' s*;
⋀*s*. (*s* ∈ *Q*) = (*s* ∈ *Q'*);
⋀*s*. *P' s* ⟹ *run f s* = *run f' s*;
⋀*s x*. *s* ∈ *Q'* ⟹ Γ⊢ ⟨*g*, *Normal s*⟩ ⇒ *x* = Γ⊢ ⟨*g'*, *Normal s*⟩ ⇒ *x*
⟧ ⟹
ccorresE st ct AF Γ *P Q f g* = *ccorresE st ct AF* Γ *P' Q' f' g*
apply *atomize*
apply (*clarsimp simp*: *ccorresE-def split*: *xstate.splits*)
by (*auto simp add*: *succeeds-def reaches-def*)

lemma *ccorresE-guard-imp*:
⟦ *ccorresE st ct AF* Γ *Q Q' A B*; ⋀*s*. *P s* ⟹ *Q s*; ⋀*t*. *t* ∈ *P'* ⟹ *t* ∈ *Q'* ⟧ ⟹
ccorresE st ct AF Γ *P P' A B*
apply *atomize*
apply (*clarsimp simp*: *ccorresE-def split*: *xstate.splits*)
done

lemma *ccorresE-guard-imp-stronger*:
⟦ *ccorresE st ct AF* Γ *Q Q' A B*;
⋀*s*. ⟦ *P* (*st s*); *s* ∈ *P'* ⟧ ⟹ *Q* (*st s*);
⋀*s*. ⟦ *P* (*st s*); *s* ∈ *P'* ⟧ ⟹ *s* ∈ *Q'* ⟧ ⟹
ccorresE st ct AF Γ *P P' A B*
apply *atomize*
apply (*clarsimp simp*: *ccorresE-def split-def split*: *xstate.splits*)
done

lemma *ccorresE-assume-pre*:
⟦ ⋀*s*. ⟦ *G* (*st s*); *s* ∈ *G'* ⟧ ⟹
ccorresE st ct AF Γ (*G and* (λ*s'*. *s'* = *st s*)) (*G'* ∩ {*t'*. *t'* = *s*}) *A B* ⟧ ⟹
ccorresE st ct AF Γ *G G' A B*
apply *atomize*
apply (*simp add*: *ccorresE-def pred-conj-def*)

done

lemma *ccorresE-Seq*:
⟦ *ccorresE st ct AF Γ ⊤ UNIV L L′*;
ccorresE st ct AF Γ ⊤ UNIV R R′ ⟧ ⟹
ccorresE st ct AF Γ ⊤ UNIV (*do* {- ← *L*; *R* }) (*L′* ;; *R′*)
apply (*clarsimp simp*: *ccorresE-def*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule exec-Normal-elim-cases*)
subgoal
apply (*clarsimp split*: *xstate.splits*
simp add: *succeeds-bind reaches-bind*, *intro conjI*)
by (*metis* (*mono-tags*, *lifting*) *Normal-resultE case-exception-or-result-Result*)
(*metis* (*mono-tags*, *lifting*) *case-exception-or-result-Exn*
case-exception-or-result-Result exec-Normal-elim-cases(*1*)
exec-Normal-elim-cases(*3*) *xstate.exhaust*)+
subgoal
apply (*clarsimp split*: *xstate.splits*
simp add: *succeeds-bind reaches-bind*)
by (*metis Abrupt Fault terminates.Seq xstate.exhaust*)
done

lemma *ccorresE-Cond*:
⟦ *ccorresE st ct AF Γ ⊤ C A L′*;
ccorresE st ct AF Γ ⊤ (*UNIV* − *C*) *A R′* ⟧ ⟹
ccorresE st ct AF Γ ⊤ UNIV A (*Cond C L′ R′*)
apply (*clarsimp simp*: *ccorresE-def pred-neg-def*)
subgoal for *s*
apply (*rule conjI*)
apply *clarsimp*
apply (*erule exec-Normal-elim-cases*)
apply (*erule-tac x*=*s* **in** *allE*, *erule impE*, *fastforce*, *fastforce*)
apply (*erule-tac x*=*s* **in** *allE*, *erule impE*, *fastforce*, *fastforce*)
apply *clarsimp*
apply (*cases s* ∈ *C*)
apply (*rule terminates.CondTrue*, *assumption*)
apply (*erule allE*, *erule impE*, *fastforce*)
apply *clarsimp*
apply (*rule terminates.CondFalse*, *assumption*)
apply (*erule allE*, *erule impE*, *fastforce*)
apply *clarsimp*
done
done

lemma *ccorresE-Cond-match*:
⟦ *ccorresE st ct AF Γ C C′ L L′*;
ccorresE st ct AF Γ (*not C*) (*UNIV* − *C′*) *R R′*;
⋀*s*. *C* (*st s*) = (*s* ∈ *C′*) ⟧ ⟹

ccorresE st ct AF Γ ⊤ UNIV (condition C L R) (Cond C′ L′ R′)
apply *atomize*
apply (*simp add*: *ccorresE-def pred-neg-def*)
apply *clarify*
apply (*intro conjI allI impI*)

subgoal for *s t*
apply (*auto elim*!: *exec-Normal-elim-cases split*: *xstate.splits*)
done
subgoal for *s*
apply (*cases s ∈ C′*)
subgoal
by (*simp add*: *terminates.CondTrue*)
subgoal
by (*simp add*: *terminates.CondFalse*)
done
done

lemma *ccorresE-Guard*:
⟦ *ccorresE st ct AF Γ ⊤ G X Y* ⟧ ⟹ *ccorresE st ct AF Γ ⊤ G X* (*Guard F G Y*)
apply (*clarsimp simp*: *ccorresE-def*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule exec-Normal-elim-cases*, *auto*)[*1*]
apply *clarsimp*
apply (*rule terminates.Guard*, *assumption*)
apply *force*
done

lemma *ccorresE-Catch*:
⟦*ccorresE st ct AF Γ ⊤ UNIV A A′*; *ccorresE st ct AF Γ ⊤ UNIV B B′*⟧ ⟹
ccorresE st ct AF Γ ⊤ UNIV (*A* <*catch*> (λ-. *B*)) (*TRY A′ CATCH B′ END*)
apply (*clarsimp simp*: *ccorresE-def*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule-tac x*=*s* **in** *allE*)
apply (*erule exec-Normal-elim-cases*)
subgoal for *s t s′*
apply (*cases t*)
subgoal
using *reaches-catch*
by (*metis case-xval-simps*(*1*) *succeeds-catch xstate.simps*(*16*) *xstate.simps*(*17*))
subgoal
using *reaches-catch*
by (*metis case-xval-simps*(*1*) *succeeds-catch xstate.simps*(*17*))
subgoal
using *reaches-catch*
by (*metis case-xval-simps*(*1*) *succeeds-catch xstate.simps*(*17*) *xstate.simps*(*18*))

subgoal
using *reaches-catch*
by (*metis case-xval-simps*(*1*) *succeeds-catch xstate.simps*(*17*) *xstate.simps*(*19*))
done
subgoal for *s t*

apply (*cases t*)
apply (*fastforce simp add*: *reaches-catch succeeds-catch*)+
done
subgoal for *s*
by (*metis case-xval-simps*(*1*) *succeeds-catch terminates.Catch xstate.simps*(*17*))
done

lemma *ccorresE-Call*:
⟦ Γ *X'* = *Some Z'*; *ccorresE st ct AF* Γ ⊤ *UNIV Z Z'* ⟧ ⟹
ccorresE st ct AF Γ ⊤ *UNIV Z* (*Call X'*)
apply (*clarsimp simp*: *ccorresE-def*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule exec-Normal-elim-cases*)
apply (*clarsimp*)
apply *clarsimp*
apply *clarify*
apply (*erule terminates.Call*)
apply (*erule allE*, *erule* (*1*) *impE*)
apply *clarsimp*
done

lemma *ccorresE-exec-Normal*:
⟦ *ccorresE st ct AF* Γ *G G' B B'*; Γ⊢ ⟨*B'*, *Normal s*⟩ ⇒ *Normal t*; *s* ∈ *G'*; *G* (*st s*); *succeeds B* (*st s*) ⟧
⟹ *reaches B* (*st s*) (*Result* ()) (*st t*)
apply (*clarsimp simp*: *ccorresE-def*)
apply *force*
done

lemma *ccorresE-exec-Abrupt*:
⟦ *ccorresE st ct AF* Γ *G G' B B'*; Γ⊢ ⟨*B'*, *Normal s*⟩ ⇒ *Abrupt t*; *s* ∈ *G'*; *G* (*st s*); *succeeds B* (*st s*) ⟧
⟹ *reaches B* (*st s*) (*Exn* ()) (*st t*)
apply (*clarsimp simp*: *ccorresE-def*)
apply *force*
done

lemma *ccorresE-exec-Fault*:
⟦ *ccorresE st ct AF* Γ *G G' B B'*; Γ⊢ ⟨*B'*, *Normal s*⟩ ⇒ *Fault f*; *f* ∉ *AF*; *s* ∈ *G'*; *G* (*st s*); *succeeds B* (*st s*) ⟧ ⟹ *P*
apply (*clarsimp simp*: *ccorresE-def*)

apply *force*
done

lemma *ccorresE-exec-Stuck*:
⟦ *ccorresE st ct AF Γ G G′ B B′*; Γ⊢ ⟨*B′*, *Normal s*⟩ ⇒ *Stuck*; *s* ∈ *G′*; *G* (*st s*); *succeeds B* (*st s*) ⟧ ⟹ *P*
apply (*clarsimp simp*: *ccorresE-def*)
apply *force*
done

lemma *ccorresE-exec-cases* [*consumes 5*]:
⟦ *ccorresE st ct AF Γ G G′ B B′*; Γ⊢ ⟨*B′*, *Normal s*⟩ ⇒ *s′*; *s* ∈ *G′*; *G* (*st s*); *succeeds B* (*st s*);
⋀*t′*. ⟦ *s′* = *Normal t′*; *reaches B* (*st s*) (*Result* ()) (*st t′*)⟧ ⟹ *R*;
⋀*t′*. ⟦ *s′* = *Abrupt t′*; *reaches B* (*st s*) (*Exn* ()) (*st t′*) ⟧ ⟹ *R*;
⋀*f*. ⟦ *s′* = *Fault f*; *f* ∈ *AF* ⟧ ⟹ *R*
⟧ ⟹ *R*
apply *atomize*
apply (*cases s′*)
apply (*drule ccorresE-exec-Normal*, *auto*)[*1*]
apply (*drule ccorresE-exec-Abrupt*, *auto*)[*1*]
subgoal for *f*
apply (*cases f* ∈ *AF*)
subgoal
by *auto*
subgoal
by (*drule ccorresE-exec-Fault*, *auto*)[*1*]
done
apply (*drule ccorresE-exec-Stuck*, *auto*)[*1*]
done

lemma *ccorresE-terminates*:
⟦ *ccorresE st ct AF Γ* ⊤ *UNIV B B′*; *succeeds B* (*st s*); *ct* ⟧ ⟹ Γ ⊢ *B′* ↓ *Normal s*
by (*clarsimp simp*: *ccorresE-def*)

lemma *exec-While-final-inv′*:
assumes *exec*: Γ⊢ ⟨*b*, *x*⟩ ⇒ *s′*
shows
⟦ *b* = *While C B*; *x* = *Normal s*;
⋀*s*. ⟦ *s* ∉ *C* ⟧ ⟹ *I s* (*Normal s*);
⋀*t t′*. ⟦ *t* ∈ *C*; Γ⊢ ⟨*B*, *Normal t*⟩ ⇒ *Normal t′*; *I t′ s′* ⟧ ⟹ *I t s′*;
⋀*t t′*. ⟦ *t* ∈ *C*; Γ⊢ ⟨*B*, *Normal t*⟩ ⇒ *Abrupt t′* ⟧ ⟹ *I t* (*Abrupt t′*);
⋀*t*. ⟦ *t* ∈ *C*; Γ⊢ ⟨*B*, *Normal t*⟩ ⇒ *Stuck* ⟧ ⟹ *I t Stuck*;
⋀*t f*. ⟦ *t* ∈ *C*; Γ⊢ ⟨*B*, *Normal t*⟩ ⇒ *Fault f* ⟧ ⟹ *I t* (*Fault f*) ⟧
⟹ *I s s′*
using *exec*
apply (*induct arbitrary*: *s rule*: *exec.induct*, *simp-all*)
apply *clarsimp*

apply *atomize*
apply *clarsimp*
apply (*erule allE*, *erule* (*1*) *impE*)
apply (*erule exec-elim-cases*, *auto*)
done

lemma *exec-While-final-inv*:
⟦ Γ⊢ ⟨*While C B*, *Normal s*⟩ ⇒ *s′*;
⋀*s*. ⟦ *s* ∉ *C* ⟧ ⟹ *I s* (*Normal s*);
⋀*t t′*. ⟦ *t* ∈ *C*; Γ⊢ ⟨*B*, *Normal t*⟩ ⇒ *Normal t′*; *I t′ s′* ⟧ ⟹ *I t s′*;
⋀*t t′*. ⟦ *t* ∈ *C*; Γ⊢ ⟨*B*, *Normal t*⟩ ⇒ *Abrupt t′* ⟧ ⟹ *I t* (*Abrupt t′*);
⋀*t*. ⟦ *t* ∈ *C*; Γ⊢ ⟨*B*, *Normal t*⟩ ⇒ *Stuck* ⟧ ⟹ *I t Stuck*;
⋀*t f*. ⟦ *t* ∈ *C*; Γ⊢ ⟨*B*, *Normal t*⟩ ⇒ *Fault f* ⟧ ⟹ *I t* (*Fault f*) ⟧
⟹ *I s s′*
apply (*erule exec-While-final-inv′*, (*rule refl*)+, *simp-all*)
done

lemma *ccorresE-termination′*:
assumes *no-fail*: *succeeds* (*whileLoop CC BB r*) *s*
and *s-match*: *s* = *st s′* ∧ *CC* = (λ-. *C*) ∧ *BB* = (λ-. *B*)
and *corres*: *ccorresE st ct AF* Γ ⊤ *UNIV B B′*
and *cond-match*: ⋀*s*. *C* (*st s*) = (*s* ∈ *C′*)
and *ct*: *ct*
shows Γ⊢ *While C′ B′* ↓ *Normal s′*
proof −
from *no-fail* **have** *run* (*whileLoop CC BB r*) *s* ≠ ⊤
by (*simp add*: *succeeds-def*)
from *this* **show** *?thesis*
using *s-match*
proof (*induct arbitrary*: *s′ rule*: *whileLoop-ne-top-induct*)
case (*step a s*)
then show *?case* **using** *corres cond-match ct*
apply (*cases s′* ∉ *C′*)
apply (*simp add*: *terminates.WhileFalse*)
apply (*rule terminates.WhileTrue*)
apply *simp*
subgoal
by (*simp add*: *runs-to-def-old ccorresE-terminates*)
subgoal
apply (*clarsimp simp*: *runs-to-def-old*)
by (*metis Abrupt Fault ccorresE-exec-Normal ccorresE-exec-Stuck*
iso-tuple-UNIV-I top1I xstate.exhaust)
done
qed
qed

lemma *ccorresE-termination*:
assumes *no-fail*: *succeeds* (*whileLoop* (λ-. *C*) (λ-. *B*) *r*) *s*
and *s-match*: *s* = *st s′*

and *corres*: *ccorresE st ct AF* Γ ⊤ *UNIV B B′*
and *cond-match*: ⋀*s*. *C* (*st s*) = (*s* ∈ *C′*)
and *ct*: *ct*
shows Γ⊢ *While C′ B′* ↓ *Normal s′*
apply (*auto intro*: *ccorresE-termination′* [*OF no-fail - corres - ct*] *simp*: *s-match cond-match*)
done

lemma *ccorresE-While*:
assumes *body-refines*: *ccorresE st ct AF* Γ ⊤ *UNIV B B′*
and *cond-match*: ⋀*s*. *C* (*st s*) = (*s* ∈ *C′*)
shows *ccorresE st ct AF* Γ *G G′* (*whileLoop* (λ-. *C*) (λ-. *B*) ()) (*While C′ B′*)
proof −
{
fix *s t*
assume *guard-abs*: *G* (*st s*)
assume *guard-conc*: *s* ∈ *G′*

assume *no-fail*: *succeeds* (*whileLoop* (λ-. *C*) (λ-. *B*) ()) (*st s*)
assume *conc-steps*: Γ⊢ ⟨*While C′ B′*, *Normal s*⟩ ⇒ *t*
have *case t of*
Normal s′ ⇒ *reaches* (*whileLoop* (λ-. *C*) (λ-. *B*) ()) (*st s*) (*Result* ()) (*st s′*)
| *Abrupt s′* ⇒ *reaches* (*whileLoop* (λ-. *C*) (λ-. *B*) ()) (*st s*) (*Exn* ()) (*st s′*)
| *Fault e* ⇒ *e* ∈ *AF*
| - ⇒ *False*
apply (*insert no-fail*, *erule rev-mp*)
apply (*rule exec-While-final-inv* [*OF conc-steps*])
subgoal
using *cond-match*
apply *clarsimp*
apply (*subst whileLoop-unroll*)
apply (*simp add*: *reaches-condition-iff*)
done
subgoal for *t1 t′*
apply (*subst* (*1 2 3*) *whileLoop-unroll*)
apply (*clarsimp simp add*: *cond-match*)
using *ccorresE-exec-Normal* [*OF body-refines*, *of t1 t′*]
using *cond-match*
apply (*force split*: *xstate.splits simp add*: *succeeds-bind reaches-bind*)
done
subgoal for *t1 t′*
apply (*subst* (*1 2 3*) *whileLoop-unroll*)
apply (*clarsimp simp add*: *cond-match*)
using *ccorresE-exec-Abrupt* [*OF body-refines*, *of t1 t′*]
using *cond-match*
apply (*clarsimp simp add*: *succeeds-bind reaches-bind*)
using *Exn-def* **by** *force*
subgoal for *t*
apply (*subst* (*1 2 3*) *whileLoop-unroll*)

```
      apply (clarsimp simp add: cond-match)
      using ccorresE-exec-Stuck [OF body-refines, of t]
      apply (metis UNIV-I succeeds-bind top1I)
      done
    subgoal for t
      apply (subst (1 2 3) whileLoop-unroll)
      apply (clarsimp simp add: cond-match)
      using ccorresE-exec-Fault [OF body-refines, of t]
      apply (metis UNIV-I succeeds-bind top1I)
      done
    done
  }
  moreover
  {
    fix s
    assume guard-abs: G (st s)
    assume guard-conc: s ∈ G'
    assume no-fail: succeeds (whileLoop (λ-. C) (λ-. B) ()) (st s)
    have ct ⟶ Γ⊢While C' B' ↓ Normal s
      apply clarify
      apply (rule ccorresE-termination [OF no-fail])
        apply (rule refl)
       apply (rule body-refines)
      apply (rule cond-match)
     apply simp
     done
  }
  ultimately show ?thesis
    by (auto simp: ccorresE-def)
qed

lemma ccorresE-get:
  (⋀s. ccorresE st ct AF Γ (P and (λs'. s' = s)) Q (L s) R) ⟹ ccorresE st ct AF
Γ P Q ((get-state) >>= L) R
  apply atomize
  apply (auto simp add: ccorresE-def succeeds-bind reaches-bind pred-conj-def split:
xstate.splits )
  done

lemma ccorresE-fail:
  ccorresE st ct AF Γ P Q fail R
  apply (clarsimp simp: ccorresE-def)
  done

lemma ccorresE-DynCom:
  ⟦ ⋀t. ⟦ t ∈ P' ⟧ ⟹ ccorresE st ct AF Γ P (P' ∩ {t'. t' = t}) A (B t) ⟧ ⟹
ccorresE st ct AF Γ P P' A (DynCom B)
  apply atomize
  apply (clarsimp simp: ccorresE-def)
```

apply (*rule conjI*)
apply *clarsimp*
apply (*erule exec-Normal-elim-cases*)
apply (*erule allE*, *erule*(*1*) *impE*)
apply *clarsimp*
apply *clarify*
apply (*rule terminates.DynCom*)
apply *clarsimp*
done

lemma *ccorresE-Catch-nothrow*:
⟦*ccorresE st ct AF* Γ ⊤ *UNIV A A'*; ¬ *exceptions-thrown A'*⟧ ⟹
ccorresE st ct AF Γ ⊤ *UNIV A* (*TRY A' CATCH B' END*)
apply (*clarsimp simp*: *ccorresE-def*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule exec-Normal-elim-cases*)
apply (*frule exceptions-thrown-not-abrupt*, *simp*, *simp*)
apply *simp*
apply *simp*
apply *clarify*
apply (*rule terminates.Catch*)
apply *clarsimp*
apply *clarsimp*
apply (*drule* (*1*) *exceptions-thrown-not-abrupt*)
apply *simp*
apply *simp*
done

context *stack-heap-state*
begin

definition *with-fresh-stack-ptr* :: *nat* ⇒ (*'s* ⇒ *'a list set*) ⇒ (*'a*::*mem-type ptr* ⇒ (*'e*::*default*, *'v*, *'s*) *spec-monad*) ⇒ (*'e*::*default*, *'v*, *'s*) *spec-monad*
where
with-fresh-stack-ptr n I c ≡
do {
p ← *assume-result-and-state* (λ*s*. {(*p*, *t*). ∃ *d vs bs*. (*p*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*::*mem-type*) (*htd s*) ∧
vs ∈ *I s* ∧ *length vs* = *n* ∧ *length bs* = *n* ∗ *size-of TYPE*(*'a*) ∧
t = *hmem-upd* (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*vs*!*i*) (*take* (*size-of TYPE*(*'a*)) (*drop* (*i* ∗ *size-of TYPE*(*'a*)) *bs*))) [*0*..<*n*]) (*htd-upd* (λ-. *d*) *s*)});
on-exit (*c p*)
({(*s*,*t*). ∃ *bs*. *length bs* = *n* ∗ *size-of TYPE*(*'a*) ∧ *t* = *hmem-upd* (*heap-update-list* (*ptr-val p*) *bs*) (*htd-upd* (*stack-releases n p*) *s*)})
}

lemma *monotone-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone R* ($\leq$) ($\lambda f.$ *c f p*)
shows *monotone R* ($\leq$) ($\lambda f.$ *with-fresh-stack-ptr n I* (*c f*))
unfolding *with-fresh-stack-ptr-def on-exit-def*
by (*intro partial-function-mono*)

lemma *monotone-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone R* ($\geq$) ($\lambda f.$ *c f p*)
shows *monotone R* ($\geq$) ($\lambda f.$ *with-fresh-stack-ptr n I* (*c f*))
unfolding *with-fresh-stack-ptr-def on-exit-def*
by (*intro partial-function-mono*)

ML ‹
```
structure with-fresh-stack-ptr =
struct

type data = {
   match: term -> {n:term, init:term, c:term, ct-: term, instantiate: {n:term,
init:term, c:term} -> term},
   cterm-match: cterm -> {n:cterm, init:cterm, c:cterm, ct-: cterm, instantiate:
{n:cterm, init:cterm, c:cterm} -> cterm},
  term: typ -> term}

fun map-match f ({match, cterm-match, term}:data) =
  {match = f match, cterm-match = cterm-match, term = term}:data
fun map-cterm-match f ({match, cterm-match, term}:data) =
  {match = match, cterm-match = f cterm-match, term = term}:data
fun map-term f ({match, cterm-match, term}:data) =
  {match = match, cterm-match = cterm-match, term = f term}:data

fun merge-match (f, g) = Utils.fast-merge (fn (f, g) => Utils.first-match [f, g])
(f, g)


structure Data = Generic-Data (
  type T = data;
   val empty = {match = fn - => raise Match, cterm-match =  fn - => raise
Match, term = fn - => raise Match}
  val merge = Utils.fast-merge (fn ({match = f1, cterm-match = g1, term = t1},
{match = f2, cterm-match = g2, term = t2}) =>
          {match = merge-match (f1, f2), cterm-match = merge-match (g1, g2),
term = merge-match (t1, t2)});
)

fun match ctxt = #match (Data.get (Context.Proof ctxt))
```

```
fun cterm-match ctxt = #cterm-match (Data.get (Context.Proof ctxt))
fun term ctxt = #term (Data.get (Context.Proof ctxt))

fun add-match match = Data.map (map-match (Utils.add-match match))
fun add-cterm-match cterm-match = Data.map (map-cterm-match (Utils.add-match
cterm-match))
fun add-term match = Data.map (map-term (Utils.add-match match))


end
›

declaration ‹
fn phi => fn context =>
 let
    val T = Morphism.typ phi @{typ 's}
    val t = Morphism.term phi @{term with-fresh-stack-ptr}
    val thy = Context.theory-of context
    fun term T' =
      let
      in
        if can (Sign.typ-match thy (T, T')) Vartab.empty then t else raise Match
      end
     fun match t = @{morph-match (fo) ‹with-fresh-stack-ptr ?n ?init ?c›} phi
(Context.theory-of context) t
        handle Pattern.MATCH => raise Match
   fun cterm-match ct = @{cterm-morph-match (fo) ‹with-fresh-stack-ptr ?n ?init
?c›} phi ct
        handle Pattern.MATCH => raise Match
 in
   context
   |> with-fresh-stack-ptr.add-match match
   |> with-fresh-stack-ptr.add-cterm-match cterm-match
   |> with-fresh-stack-ptr.add-term term
 end
›

end

end
```

Chapter 18

L1 phase

theory *L1Defs*
imports *CCorresE*
begin

type-synonym *'s L1-monad* = (*unit*, *unit*, *'s*) *exn-monad*
definition *L1-seq* (*A* :: *'s L1-monad*) (*B* :: *'s L1-monad*) ≡ (*A* >>= (λ-. *B*)) :: *'s L1-monad*
definition *L1-skip* ≡ *return* () :: *'s L1-monad*
definition *L1-modify m* ≡ (*modify m*) :: *'s L1-monad*
definition *L1-condition c* (*A* :: *'s L1-monad*) (*B* :: *'s L1-monad*) ≡ *condition c A B*
definition *L1-catch* (*A* :: *'s L1-monad*) (*B* :: *'s L1-monad*) ≡ (*A* <*catch*> (λ-. *B*))
definition *L1-while c* (*A* :: *'s L1-monad*) ≡ (*whileLoop* (λ-. *c*) (λ-. *A*) ())
definition *L1-throw* ≡ *throw* () :: *'s L1-monad*
definition *L1-spec r* ≡ *state-select r* :: *'s L1-monad*
definition *L1-assume f* ≡ *assume-result-and-state f* :: *'s L1-monad*
definition *L1-guard c* ≡ *guard c* :: *'s L1-monad*
definition *L1-init v* ≡ (*do* { *x* ← *select UNIV*; *modify* (*v* (λ-. *x*)) }) :: *'s L1-monad*
definition *L1-call scope-setup* (*dest-fn* :: *'s L1-monad*) *scope-teardown result-exn return-xf* ≡
  do {
    s ← *get-state*;
    ((*do* { *modify scope-setup*;
        dest-fn })
      <*catch*> (λ-. *L1-seq* (*modify* (λ*t*. *result-exn* (*scope-teardown s t*) *t*)) *L1-throw*));
    t ← *get-state*;
    modify (*scope-teardown s*);
    modify (*return-xf t*)
  }

definition *L1-fail* ≡ *fail* :: *'s L1-monad*
definition *L1-set-to-pred S* ≡ λ*s*. *s* ∈ *S*
definition *L1-rel-to-fun R* = (λ*s*. *Pair* () ‘ *Image R* {*s*})

lemma *L1-rel-to-fun-alt*: *L1-rel-to-fun R* = (λ*s*. *Pair* () ‘ {*s'*. (*s*, *s'*) ∈ *R*})
by (*auto simp*: *L1-rel-to-fun-def*)

lemmas *L1-defs* = *L1-seq-def L1-skip-def L1-modify-def L1-condition-def*
L1-catch-def L1-while-def L1-throw-def L1-spec-def L1-assume-def L1-guard-def
L1-fail-def L1-init-def L1-set-to-pred-def L1-rel-to-fun-def

lemmas *L1-defs'* =
L1-seq-def L1-skip-def L1-modify-def L1-condition-def L1-catch-def
L1-while-def L1-throw-def L1-spec-def L1-assume-def
L1-guard-def
L1-fail-def L1-init-def L1-set-to-pred-def L1-rel-to-fun-def

declare *L1-set-to-pred-def* [*simp*]
declare *L1-rel-to-fun-def* [*simp*]

definition *L1-guarded*:: (*'s* ⇒ *bool*) ⇒ *'s L1-monad* ⇒ *'s L1-monad*
where
L1-guarded g f = *L1-seq* (*L1-guard g*) *f*

locale *L1-functions* =
fixes P:: *unit ptr* ⇒ *'s L1-monad*
begin
definition *L1-dyn-call g scope-setup* (*dest* :: *'s* ⇒ *unit ptr*) *scope-teardown result-exn return-xf* ≡
L1-guarded g (*gets dest* >>= (λ*p*. *L1-call scope-setup* (P *p*) *scope-teardown result-exn return-xf*))
end

definition
L1corres :: *bool* ⇒ (*'p* ⇒ (*'s*, *'p*, *strictc-errortype*) *com option*)
⇒ *'s L1-monad* ⇒ (*'s*, *'p*, *strictc-errortype*) *com* ⇒ *bool*
where
L1corres check-term Γ ≡
λ*A C*. ∀ *s*. *succeeds A s* ⟶
((∀ *t*. Γ ⊢ ⟨*C*, *Normal s*⟩ ⇒ *t* ⟶
(*case t of*
Normal s' ⇒ *reaches A s* (*Result* ()) *s'*
| *Abrupt s'* ⇒ *reaches A s* (*Exn* ()) *s'*
| *Fault e* ⇒ *e* ∈ {*AssumeError*, *StackOverflow*}
| - ⇒ *False*))

$\wedge$ (*check-term* $\longrightarrow \Gamma \vdash C \downarrow$ *Normal s*))

definition
L1corres' :: *bool* $\Rightarrow$ (*'p* $\Rightarrow$ (*'s*, *'p*, *strictc-errortype*) *com option*) $\Rightarrow$ (*'s* $\Rightarrow$ *bool*)
$\Rightarrow$ *'s L1-monad* $\Rightarrow$ (*'s*, *'p*, *strictc-errortype*) *com* $\Rightarrow$ *bool*
where
L1corres' check-term Γ *P* $\equiv$
$\lambda A\ C.\ \forall s.\ (P\ s) \wedge$ *succeeds A s* $\longrightarrow$
((*∀ t.* $\Gamma \vdash \langle C,$ *Normal s*$\rangle \Rightarrow t \longrightarrow$
(*case t of*
Normal s' $\Rightarrow$ *reaches A s* (*Result* ()) *s'*
| *Abrupt s'* $\Rightarrow$ *reaches A s* (*Exn* ()) *s'*
| *Fault e* $\Rightarrow$ *e* $\in$ {*AssumeError*, *StackOverflow*}
| - $\Rightarrow$ *False*))
$\wedge$ (*check-term* $\longrightarrow \Gamma \vdash C \downarrow$ *Normal s*))

lemma *L1corres-alt-def*: *L1corres ct* Γ = *ccorresE* ($\lambda x.\ x$) *ct* {*AssumeError*, *StackOverflow*} Γ $\top$ *UNIV*
apply (*rule ext*)+
apply (*clarsimp simp*: *L1corres-def ccorresE-def*)
done

lemma *L1corres'-alt-def*: *L1corres' ct* Γ *P* = *ccorresE* ($\lambda x.\ x$) *ct* {*AssumeError*, *StackOverflow*} Γ *P UNIV*
apply (*rule ext*)+
apply (*clarsimp simp*: *L1corres'-def ccorresE-def*)
done

lemma *admissible-nondet-ord-L1corres* [*corres-admissible*]:
ccpo.admissible Inf ($\geq$) ($\lambda A.$ *L1corres ct* Γ *A C*)
unfolding *L1corres-def imp-conjR disj-imp*[*symmetric*]
apply (*intro admissible-all*)
apply (*intro admissible-conj*)
apply (*rule ccpo.admissibleI*)
apply (*clarsimp split*: *xstate.splits*)
apply (*intro conjI admissible-mem*)
subgoal
apply (*simp add*: *ccpo.admissible-def chain-def*
succeeds-def reaches-def split: *prod.splits*)
apply *transfer*
apply (*clarsimp simp add*: *Inf-post-state-def top-post-state-def image-def vimage-def*
split: *if-split-asm*)
by (*metis outcomes.simps*(*2*) *post-state.simps*(*3*))
subgoal
apply (*simp add*: *ccpo.admissible-def chain-def*

succeeds-def reaches-def split: *prod.splits*)
apply *transfer*
apply (*clarsimp simp add*: *Inf-post-state-def top-post-state-def image-def vimage-def*
split: *if-split-asm*)
by (*metis outcomes.simps*(*2*) *post-state.simps*(*3*))
subgoal
apply (*simp add*: *ccpo.admissible-def chain-def*
succeeds-def reaches-def split: *prod.splits*)
apply *transfer*
apply (*clarsimp simp add*: *Inf-post-state-def top-post-state-def image-def vimage-def*
split: *if-split-asm*)
by (*metis outcomes.simps*(*2*) *post-state.simps*(*3*))
subgoal
apply (*simp add*: *ccpo.admissible-def chain-def*
succeeds-def reaches-def split: *prod.splits*)
apply *transfer*
apply (*auto simp add*: *Inf-post-state-def top-post-state-def image-def vimage-def*
split: *if-split-asm*)
done
apply (*rule ccpo.admissibleI*)
apply (*simp split*: *xstate.splits*)
apply (*clarsimp simp add*: *ccpo.admissible-def chain-def*
succeeds-def reaches-def split: *prod.splits*)
apply *transfer*
apply (*auto simp add*: *Inf-post-state-def top-post-state-def image-def vimage-def*
split: *if-split-asm*)
done

lemma *L1corres-top* [*corres-top*]: *L1corres ct P* ⊤ *C*
by (*auto simp add*: *L1corres-def*)

lemma *L1corres-guard-DynCom*:
⟦⋀*s*. *s* ∈ *g* ⟹ *L1corres ct* Γ (*B s*) (*B*′ *s*)⟧ ⟹
L1corres ct Γ (*L1-seq* (*L1-guard* (λ*s*. *s* ∈ *g*)) (*gets B* ⤜ (λ*b*. *b*))) (*Guard f g* (*DynCom B*′))
apply (*clarsimp simp*: *L1corres-def L1-defs*)
apply (*force elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind*)
done

lemma *L1corres*′*-guard-DynCom-conseq*:
assumes *conseq*: ⋀*s*. *P s* ⟹ *g s* ⟹ *s* ∈ *g*′
assumes *corres*: ⋀*s*. *P s* ⟹ *g s* ⟹ *L1corres*′ *ct* Γ (λ*s*. *P s* ∧ *g s*) (*B s*) (*B*′ *s*)
shows *L1corres*′ *ct* Γ *P* (*L1-seq* (*L1-guard g*) (*gets B* ⤜ (λ*b*. *b*))) (*Guard f g*′

(*DynCom B'*))
using *conseq corres*
apply (*clarsimp simp*: *L1corres'-def L1-defs*)
apply (*force elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind*)

done

lemma *L1corres'-guard-DynCom*:
⟦⋀*s*. *P s* ⟹ *s* ∈ *g* ⟹ *L1corres' ct* Γ (λ*s*. *P s* ∧ *s* ∈ *g*) (*B s*) (*B' s*)⟧ ⟹
L1corres' ct Γ *P* (*L1-seq* (*L1-guard* (λ*s*. *s* ∈ *g*)) (*gets B* ≫= (λ*b*. *b*))) (*Guard f g* (*DynCom B'*))
apply (*rule L1corres'-guard-DynCom-conseq* [**where** *B*=*B*])
apply *simp*
apply *simp*
done

lemma *L1corres-DynCom*:
assumes *corres-f*: ⋀*s*. *L1corres ct* Γ (*g s*) (*f s*)
shows *L1corres ct* Γ (*gets g* ≫= (λ*b*. *b*)) (*DynCom f*)
using *corres-f*
apply (*clarsimp simp*: *L1corres-def L1-defs*)
apply (*force elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind*)
done

lemma *L1corres'-DynCom*:
assumes *corres-f*: ⋀*s*. *P s* ⟹ *L1corres' ct* Γ *P* (*g s*) (*f s*)
shows *L1corres' ct* Γ *P* (*gets g* ≫= (λ*b*. *b*)) (*DynCom f*)
using *corres-f*
apply (*clarsimp simp*: *L1corres'-def L1-defs*)
apply (*force elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind*)
done

lemma *L1corres'-DynCom-fix-state*:
assumes *corres-f*: ⋀*s*. *P s* ⟹ *L1corres' ct* Γ (λ*s'*. *P s'* ∧ *s'* = *s*) (*g s*) (*f s*)
shows *L1corres' ct* Γ *P* (*gets g* ≫= (λ*b*. *b*)) (*DynCom f*)
using *corres-f*
apply (*clarsimp simp*: *L1corres'-def L1-defs*)
apply (*force elim*!: *exec-Normal-elim-cases*

intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind*)
done

lemma *L1corres′-guard′*:
⟦ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *s* ∈ *g*) *B B′* ⟧ ⟹
L1corres′ ct Γ *P* (*L1-seq* (*L1-guard* (λ*s*. *s* ∈ *g*)) *B*) (*Guard f g B′*)
apply (*clarsimp simp*: *L1corres′-def L1-defs*)
apply (*erule-tac x*=*s* **in** *allE*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule exec-Normal-elim-cases*)
subgoal
by (*auto simp*: *succeeds-bind reaches-bind split*: *xstate.splits*)
subgoal
by (*auto simp*: *succeeds-bind reaches-bind split*: *xstate.splits*)
subgoal
by (*auto simp*: *succeeds-bind intro*: *terminates.intros*)
done

lemma *L1corres′-guarded*:
⟦ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *s* ∈ *g*) *B B′* ⟧ ⟹
L1corres′ ct Γ *P* (*L1-guarded* (λ*s*. *s* ∈ *g*) *B*) (*Guard f g B′*)
unfolding *L1-guarded-def* **by** (*rule L1corres′-guard′*)

lemma *L1corres′-Guard-maybe-guard*:
L1corres′ ct Γ *P B* (*Guard f g B′*) ⟹ *L1corres′ ct* Γ *P B* (*maybe-guard f g B′*)
apply (*simp add*: *L1corres′-def maybe-guard-def*)
by (*meson exec.Guard iso-tuple-UNIV-I terminates-Normal-elim-cases*(*2*))

lemma *L1corres′-guarded-DynCom-conseq*:
assumes *conseq*: ⋀*s*. *P s* ⟹ *g s* ⟹ *s* ∈ *g′*
assumes *corres-B*: ⋀*s*. *P s* ⟹ *g s* ⟹ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *g s*) (*B s*) (*B′ s*)
shows *L1corres′ ct* Γ *P* (*L1-guarded g* (*gets B* ⤜ (λ*b*. *b*))) (*maybe-guard f g′* (*DynCom B′*))
apply (*rule L1corres′-Guard-maybe-guard*)
unfolding *L1-guarded-def L1-set-to-pred-def*
using *corres-B*
by (*simp add*: *L1corres′-guard-DynCom-conseq* [*OF conseq*])

lemma *L1corres′-guarded-DynCom*:
assumes *corres-B*: ⋀*s*. *P s* ⟹ *s* ∈ *g* ⟹ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *s* ∈ *g*) (*B s*) (*B′ s*)
shows *L1corres′ ct* Γ *P* (*L1-guarded* (*L1-set-to-pred g*) (*gets B* ⤜ (λ*b*. *b*))) (*maybe-guard f g* (*DynCom B′*))
apply (*rule L1corres′-guarded-DynCom-conseq* [**where** *B*=*B*])

apply *simp*
using *corres-B*
apply *simp*
done

lemma *L1corres′-conseq*:
assumes *corres-Q*: *L1corres′ ct* Γ *Q B B′*
assumes *conseq*: ⋀*s*. *P s* ⟹ *Q s*
shows *L1corres′ ct* Γ *P B B′*
using *conseq corres-Q*
by (*auto simp add*: *L1corres′-def*)

lemma *L1corres-to-L1corres′*: *L1corres ct* Γ = *L1corres′ ct* Γ ⊤
by (*simp add*: *L1corres-def L1corres′-def*)

lemma *L1corres-guarded-DynCom-conseq*:
assumes *conseq*: ⋀*s*. *g s* ⟹ *s* ∈ *g′*
assumes *corres-B*: ⋀*s*. *g s* ⟹ *L1corres ct* Γ (*B s*) (*B′ s*)
shows *L1corres ct* Γ (*L1-guarded g* (*gets B* ≫= (λ*b*. *b*))) (*maybe-guard f g′* (*DynCom B′*))
using *corres-B* **unfolding** *L1corres-to-L1corres′*
thm *L1corres′-guarded-DynCom*
apply −
apply (*rule L1corres′-guarded-DynCom-conseq* [**where** *B*=*B*])
using *conseq* **apply** *simp*
apply (*rule L1corres′-conseq* [**where** *Q* = ⊤])
apply (*simp*)
apply *simp*
done

lemma *L1corres-guarded-DynCom*:
assumes *corres-B*: ⋀*s*. *s* ∈ *g* ⟹ *L1corres ct* Γ (*B s*) (*B′ s*)
shows *L1corres ct* Γ (*L1-guarded* (*L1-set-to-pred g*) (*gets B* ≫= (λ*b*. *b*))) (*maybe-guard f g* (*DynCom B′*))
using *corres-B* **unfolding** *L1corres-to-L1corres′*
apply −
apply (*rule L1corres′-guarded-DynCom* [**where** *B*=*B*])
apply (*rule L1corres′-conseq* [**where** *Q* = ⊤])
apply *simp*
apply *simp*
done

definition
L1-call-simpl check-term Gamma proc
= *do* {*s* ← *get-state*;
assert (*check-term* ⟶ *Gamma* ⊢ *Call proc* ↓ *Normal s*);
xs ← *select* {*t*. *Gamma* ⊢ ⟨*Call proc*, *Normal s*⟩ ⇒ *t*};

```
      case xs :: (-, strictc-errortype) xstate of
          Normal s ⇒ set-state s
        | Abrupt s ⇒ do {set-state s; throw () }
        | Fault ft ⇒ fail
        | Stuck ⇒ fail
    }


lemma L1corres-call-simpl:
  L1corres ct Γ (L1-call-simpl ct Γ proc) (Call proc)
  apply (clarsimp simp: L1corres-def L1-call-simpl-def)
  apply safe
  subgoal for s t
    apply (cases t)
      apply (fastforce elim!:  exec-Normal-elim-cases
       intro: terminates.intros exec.intros

       simp add: succeeds-bind reaches-bind Exn-def )+
    done
  subgoal for s
    apply (auto intro!: terminates.intros simp add: succeeds-bind reaches-bind)
    done
  done

lemma L1corres-skip:
  L1corres ct Γ L1-skip SKIP
  unfolding L1corres-def L1-defs
  by (force elim!:  exec-Normal-elim-cases
      intro: terminates.intros
      split: xstate.splits
      simp add: succeeds-bind reaches-bind)

lemma L1corres-throw:
  L1corres ct Γ L1-throw Throw
  unfolding L1corres-def L1-defs
  by (auto elim!:  exec-Normal-elim-cases
      intro: terminates.intros
      split: xstate.splits
      simp add: succeeds-bind reaches-bind Exn-def)

lemma L1corres-seq:
  ⟦ L1corres ct Γ L L'; L1corres ct Γ R R' ⟧ ⟹
    L1corres ct Γ (L1-seq L R) (L' ;; R')
  apply (clarsimp simp: L1corres-alt-def)
  apply (clarsimp simp: L1-seq-def)
  apply (rule ccorresE-Seq)
  apply auto
  done
```

lemma *L1corres-modify*:
L1corres ct Γ (*L1-modify m*) (*Basic m*)
apply (*clarsimp simp*: *L1corres-def L1-defs*)
apply (*auto elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def*)
done

lemma *L1corres-condition*:
⟦ *L1corres ct* Γ *L L'*; *L1corres ct* Γ *R R'* ⟧ ⟹
L1corres ct Γ (*L1-condition* (*L1-set-to-pred c*) *L R*) (*Cond c L' R'*)
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *L1-defs*)
apply (*rule ccorresE-Cond-match*)
apply (*erule ccorresE-guard-imp*, *simp*+)[*1*]
apply (*erule ccorresE-guard-imp*, *simp*+)[*1*]
apply *simp*
done

lemma *L1corres-catch*:
⟦ *L1corres ct* Γ *L L'*; *L1corres ct* Γ *R R'* ⟧ ⟹
L1corres ct Γ (*L1-catch L R*) (*Catch L' R'*)
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *L1-catch-def*)
apply (*erule ccorresE-Catch*)
apply *force*
done

lemma *L1corres-while*:
⟦ *L1corres ct* Γ *B B'* ⟧ ⟹
L1corres ct Γ (*L1-while* (*L1-set-to-pred c*) *B*) (*While c B'*)
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *L1-defs*)
apply (*rule ccorresE-While*)
apply *clarsimp*
apply *simp*
done

lemma *L1corres-guard*:
⟦ *L1corres ct* Γ *B B'* ⟧ ⟹
L1corres ct Γ (*L1-seq* (*L1-guard* (*L1-set-to-pred c*)) *B*) (*Guard f c B'*)
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *ccorresE-def L1-defs*)
apply (*erule-tac x*=*s* **in** *allE*)
apply (*auto elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def*)

done

lemma *L1corres-spec*:
L1corres ct Γ (*L1-spec x*) (*com.Spec x*)
apply (*clarsimp simp*: *L1corres-def L1-defs*)
apply (*auto elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def*)
done

lemma *L1-init-alt-def*:
L1-init upd ≡ *L1-spec* {(*s*, *t*). ∃ *v*. *t* = *upd* (λ-. *v*) *s*}
apply (*rule eq-reflection*)
apply (*clarsimp simp*: *L1-defs*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L1corres-init*:
L1corres ct Γ (*L1-init upd*) (*lvar-nondet-init upd*)
by (*auto simp*: *L1-init-alt-def lvar-nondet-init-def intro*: *L1corres-spec*)

lemma *L1corres-guarded-spec*:
L1corres ct Γ (*L1-spec R*) (*guarded-spec-body F R*)
apply (*clarsimp simp*: *L1corres-alt-def ccorresE-def L1-spec-def guarded-spec-body-def*)
apply (*force simp*: *liftE-def bind-def*
elim: *exec-Normal-elim-cases intro*: *terminates.Guard terminates.Spec*)
done

lemma *L1corres-assume*:
L1corres ct Γ (*L1-assume* (*L1-rel-to-fun R*)) (*guarded-spec-body AssumeError R*)
apply (*clarsimp simp*: *L1corres-alt-def ccorresE-def L1-assume-def L1-rel-to-fun-alt guarded-spec-body-def*)
apply (*auto elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def*)
done

lemma *pred-conj-apply*[*simp*]: (*P and Q*) *s* ⟷ *P s* ∧ *Q s*
by (*auto simp add*: *pred-conj-def*)

lemma *L1corres-call*:
⟦ *L1corres ct* Γ *dest-fn* (*Call dest*) ⟧ ⟹
L1corres ct Γ
(*L1-call scope-setup dest-fn scope-teardown-norm scope-teardown-exn f*)
(*call-exn scope-setup dest scope-teardown-norm scope-teardown-exn* (λ- *t*.

Basic (*f t*)))
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*unfold call-exn-def block-exn-def L1-call-def*)
apply (*rule ccorresE-DynCom*)
apply *clarsimp*
apply (*rule ccorresE-get*)
apply (*rule ccorresE-assume-pre*, *clarsimp*)
apply (*rule ccorresE-guard-imp-stronger*)
apply (*rule ccorresE-Seq*)
apply (*rule ccorresE-Catch*)
apply (*rule ccorresE-Seq*)
apply (*rule L1corres-modify*[*unfolded L1-modify-def L1corres-alt-def*])
apply *assumption*
apply (*rule L1corres-seq*[*unfolded L1corres-alt-def*])
apply (*rule L1corres-modify*[*unfolded L1-modify-def L1corres-alt-def*])
apply (*rule L1corres-throw* [*unfolded L1corres-alt-def*])
apply (*rule ccorresE-DynCom*)
apply (*rule ccorresE-get*)
apply (*rule ccorresE-assume-pre*, *clarsimp*)
apply (*rule ccorresE-guard-imp*)
apply (*rule ccorresE-Seq*)
apply (*rule L1corres-modify*[*unfolded L1-modify-def L1corres-alt-def*])
apply (*rule L1corres-modify*[*unfolded L1-modify-def L1corres-alt-def*])
apply *simp*
apply *simp*
apply *simp*
apply *simp*
done

lemma (**in** *L1-functions*) *L1corres-dyn-call-conseq*:
assumes *conseq*: $\bigwedge s.\ g\ s \Longrightarrow s \in g'$
assumes *corres-dest*: $\bigwedge s.\ g\ s \Longrightarrow$ *L1corres ct* Γ ($\mathcal{P}$ (*dest s*)) (*Call* (*dest s*))
shows
L1corres ct Γ
(*L1-dyn-call g scope-setup dest scope-teardown-norm scope-teardown-exn result*)
(*dynCall-exn f g′ scope-setup dest scope-teardown-norm scope-teardown-exn* (λ- *t*. *Basic* (*result t*)))
proof −
have *eq*: (*gets* (λs. *L1-call scope-setup* ($\mathcal{P}$ (*dest s*)) *scope-teardown-norm scope-teardown-exn result*) $\gg\!=$ (λb. *b*)) =
(*gets dest* $\gg\!=$ (λp. *L1-call scope-setup* ($\mathcal{P}$ *p*) *scope-teardown-norm scope-teardown-exn result*))
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)
done
show *?thesis*
apply (*simp add*: *L1-dyn-call-def dynCall-exn-def*)
apply (*rule L1corres-guarded-DynCom-conseq* [**where** $B = \lambda s$. *L1-call scope-setup*

($\mathcal{P}$ (*dest s*)) *scope-teardown-norm scope-teardown-exn result*, *simplified eq*])
apply (*simp add*: *conseq*)
apply (*rule L1corres-call*)
apply (*rule corres-dest*)
by *simp*
qed

lemma (**in** *L1-functions*) *L1corres-dyn-call-same-guard*:
assumes *eq*: *L1-set-to-pred g* ≡ *g′*
assumes *corres-dest*: ⋀*s*. *g′ s* ⟹ *L1corres ct* Γ ($\mathcal{P}$ (*dest s*)) (*Call* (*dest s*))
shows
L1corres ct Γ
(*L1-dyn-call g′ scope-setup dest scope-teardown-norm scope-teardown-exn result*)
(*dynCall-exn f g scope-setup dest scope-teardown-norm scope-teardown-exn* (λ- *t*. *Basic* (*result t*)))
apply (*rule L1corres-dyn-call-conseq*)
apply (*simp add*: *eq* [*symmetric*])
by (*rule corres-dest*)

lemma (**in** *L1-functions*) *L1corres-dyn-call-add-and-select-guard*:
assumes *eq*: *L1-set-to-pred g* ≡ *g′*
assumes *corres-dest*: ⋀*s*. *G s* ⟹ *L1corres ct* Γ ($\mathcal{P}$ (*dest s*)) (*Call* (*dest s*))
shows
L1corres ct Γ
(*L1-dyn-call* (*G and g′*) *scope-setup dest scope-teardown-norm scope-teardown-exn result*)
(*dynCall-exn f g scope-setup dest scope-teardown-norm scope-teardown-exn* (λ- *t*. *Basic* (*result t*)))
apply (*rule L1corres-dyn-call-conseq*)
apply (*simp add*: *eq* [*symmetric*])
apply (*rule corres-dest*)
apply (*simp*)
done

lemma *L1-seq-guard-merge*: *L1-seq* (*L1-guard P*) (*L1-seq* (*L1-guard Q*) *c*) = *L1-seq* (*L1-guard* (*P and Q*)) *c*
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *and-unfold*: (*and*) = (λ*P Q s*. *P s* ∧ *Q s*)
by (*auto simp add*: *fun-eq-iff*)

lemma *L1-seq-guard-eq*: (⋀*s*. *P s* = *Q s*) ⟹ *L1-seq* (*L1-guard P*) *c* = *L1-seq* (*L1-guard Q*) *c*

unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *foldr-and-commute*: $\bigwedge s.\ foldr\ (and)\ gs\ (P\ and\ g)\ s = (g\ s \wedge foldr\ (and)\ gs\ P\ s)$
by (*induct gs arbitrary*: *P g*) (*auto simp add*:)

lemma *L1corres-fail*:
L1corres ct Γ *L1-fail X*
apply (*clarsimp simp*: *L1corres-alt-def*)
apply (*clarsimp simp*: *L1-fail-def*)
apply (*rule ccorresE-fail*)
done

lemma *L1corres-prepend-unknown-var'*:
$[\![$ *L1corres ct* Γ *A C*; $\bigwedge s::'s::type.\ X\ (\lambda\text{-}::'a::type.\ (X'\ s))\ s = s\ ]\!] \Longrightarrow$ *L1corres ct* Γ (*L1-seq* (*L1-init X*) *A*) *C*
unfolding *L1corres-def L1-defs*
by (*auto elim*!: *exec-Normal-elim-cases*
intro: *terminates.intros*
split: *xstate.splits*
simp add: *succeeds-bind reaches-bind Exn-def*)
metis+

lemma *L1-catch-seq-join*: *no-throw* (λ-. *True*) *A* ⟹ *L1-seq A* (*L1-catch B C*) = (*L1-catch* (*L1-seq A B*) *C*)
unfolding *no-throw-def L1-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*clarsimp simp add*: *runs-to-def-old runs-to-partial-def-old Exn-def default-option-def*)
apply (*intro conjI impI iffI*)
apply (*metis Exn-def Exn-neq-Result*)+
done

lemma *no-throw-L1-init* [*simp*]: *no-throw P* (*L1-init f*)
unfolding *L1-init-def no-throw-def*
apply (*clarsimp*)
apply (*runs-to-vcg*)
done

lemma *L1corres-prepend-unknown-var*:
$[\![$ *L1corres ct* Γ (*L1-catch A B*) *C*; $\bigwedge s.\ X\ (\lambda\text{-}::'d::type.\ (X'\ s))\ s = s\ ]\!]$

```
      ⟹ L1corres ct Γ (L1-catch (L1-seq (L1-init X) A) B) C
  unfolding L1corres-def L1-defs
  by (auto elim!: exec-Normal-elim-cases
      intro: terminates.intros
      split: xstate.splits
      simp add: succeeds-bind reaches-bind Exn-def succeeds-catch reaches-catch)
     metis+

lemma L1corres-Call:
  ⟦ Γ X' = Some Z'; L1corres ct Γ Z Z' ⟧ ⟹
    L1corres ct Γ Z (Call X')
  apply (clarsimp simp: L1corres-alt-def)
  apply (erule (1) ccorresE-Call)
  done

lemma L1-call-corres [fundef-cong]:
  ⟦ scope-setup = scope-setup';
     dest-fn = dest-fn';
     scope-teardown = scope-teardown';
     return-xf = return-xf' ⟧ ⟹
    L1-call scope-setup dest-fn scope-teardown return-xf =
    L1-call scope-setup' dest-fn' scope-teardown' return-xf'
  apply (clarsimp simp: L1-call-def)
  done


lemma L1-corres-cleanup:
  L1corres ct Γ (do {y <- state-select {(s,t). t = cleanup s};
                     return ()
                  })
     (Basic cleanup)
  unfolding L1corres-def L1-defs
  by (auto elim!: exec-Normal-elim-cases
      intro: terminates.intros
      split: xstate.splits
      simp add: succeeds-bind reaches-bind Exn-def)

lemma L1-corres-spec-cleanup:
  L1corres ct Γ (do { y <- state-select cleanup;
                     return ()
                  })
     (com.Spec cleanup)
  unfolding L1corres-def L1-defs
  by (auto elim!: exec-Normal-elim-cases
      intro: terminates.intros
      split: xstate.splits
      simp add: succeeds-bind reaches-bind Exn-def)

lemma L1-corres-cleanup-throw:
```

```
  L1corres ct Γ (do { - <− state-select {(s,t). t = cleanup s};
                      throw ()
                   })
    (Basic cleanup;; THROW)
  unfolding L1corres-def L1-defs
  by (auto elim!: exec-Normal-elim-cases
      intro: terminates.intros
      split: xstate.splits
      simp add: succeeds-bind reaches-bind Exn-def default-option-def)


lemma L1-corres-spec-cleanup-throw:
  L1corres ct Γ (do{ - <− state-select cleanup;
                      throw ()
                   })
    (com.Spec cleanup;; THROW)
  unfolding L1corres-def L1-defs
  by (auto elim!: exec-Normal-elim-cases
      intro: terminates.intros
      split: xstate.splits
      simp add: succeeds-bind reaches-bind Exn-def default-option-def)

lemma on-exit-unit-def: (on-exit f cleanup::(unit, unit, 's) exn-monad) =
  bind-handle f
    (λv. bind (state-select cleanup) (λ-. return ()))
    (λe. bind (state-select cleanup)(λ-. throw ()))
  unfolding on-exit-def on-exit'-def bind-handle-eq
  by (intro arg-cong2[where f=bind-exception-or-result])
     (auto simp: fun-eq-iff default-option-def Exn-def
          split: exception-or-result-split)

lemma on-exit-catch-conv: on-exit f cleanup =
 do {
    r ← (f <catch> (λe. state-select cleanup >>= (λ-. throw e)));
    state-select cleanup;
    return r
  }
  unfolding on-exit-def on-exit'-def
  apply (rule spec-monad-eqI)
  apply (clarsimp simp add: runs-to-iff fun-eq-iff Exn-def default-option-def
      intro!: arg-cong[where f=runs-to - -])
  by (metis default-option-def exception-or-result-cases not-None-eq)

lemma L2corres-on-exit':
  assumes m-c: L1corres ct Γ m c
   shows L1corres ct Γ (on-exit m {(s,t). t = cleanup s}) (On-Exit c (Basic
cleanup))
  unfolding on-exit-catch-conv
  apply (simp add: On-Exit-def)
```

apply (*rule L1corres-seq* [*simplified L1-seq-def*])
apply (*simp add*: *L1-catch-def* [*symmetric*])
apply (*rule L1corres-catch* [*OF m-c*])
apply (*rule L1-corres-cleanup-throw* [*simplified*])
apply (*rule L1-corres-cleanup* [*simplified*])
done

lemma *L2corres-on-exit*:
assumes *m-c*: *L1corres ct* Γ *m c*
shows *L1corres ct* Γ (*on-exit m cleanup*) (*On-Exit c* (*com.Spec cleanup*))
apply (*simp add*: *on-exit-catch-conv On-Exit-def*)
apply (*rule L1corres-seq* [*simplified L1-seq-def*])
apply (*simp add*: *L1-catch-def* [*symmetric*])
apply (*rule L1corres-catch* [*OF m-c*])
apply (*rule L1-corres-spec-cleanup-throw*[*simplified*])
apply (*rule L1-corres-spec-cleanup*[*simplified*])
done

definition
refines-simpl :: *bool* ⇒ (*'p* ⇒ (*'s*, *'p*, *strictc-errortype*) *com option*) ⇒
(*'s*, *'p*, *strictc-errortype*) *com* ⇒
((*'e*::*default*, *'a*, *'t*) *spec-monad*) ⇒
's ⇒ *'t* ⇒ ((*'s*, *strictc-errortype*) *xstate* ⇒ ((*'e*, *'a*) *exception-or-result* ∗ *'t*) ⇒
bool) ⇒ *bool* **where**
refines-simpl ct Γ *c m s t R* ≡
succeeds m t ⟶
((∀ *s'*. Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *s'* ⟶
(*s'* ∈ {*Fault AssumeError*, *Fault StackOverflow*} ∨
(∃ *r t'*. *reaches m t r t'* ∧ *R s'* (*r*, *t'*)))) ∧
(*ct* ⟶ Γ ⊢ *c* ↓ *Normal s*))

lemma *refines-simplI*:
assumes *termi*: *succeeds m t* ⟹ *ct* ⟹ Γ ⊢ *c* ↓ *Normal s*
assumes *sim*: ⋀*s'*. *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *s'* ⟹ *s'* ∉ {*Fault AssumeError*, *Fault StackOverflow*}
⟹ ∃ *r t'*. *reaches m t r t'* ∧ *R s'* (*r*, *t'*)
shows *refines-simpl ct* Γ *c m s t R*
using *termi sim* **unfolding** *refines-simpl-def*
by *blast*

definition
rel-L1 :: (*'s*, *strictc-errortype*) *xstate* ⇒ (*'e*, *'a*) *xval* × *'s* ⇒ *bool* **where**
rel-L1 ≡ λ*s* (*r*, *t*). (*case s of*
Normal s' ⇒ (∃ *x*. *r* = *Result x*) ∧ *t* = *s'*
| *Abrupt s'* ⇒ (∃ *x*. *r* = *Exn x*) ∧ *t* = *s'*
| *Fault e* ⇒ *False*
| *Stuck* ⇒ *False*)

lemma *rel-L1-unit*:
rel-L1 = (λ*s* (*r*, *t*). (*case s of*
Normal s′ ⇒ *r* = *Result* () ∧ *t* = *s′*
| *Abrupt s′* ⇒ *r* = *Exn* () ∧ *t* = *s′*
| *Fault e* ⇒ *False*
| *Stuck* ⇒ *False*))
by (*auto simp add*: *rel-L1-def split*: *xstate.splits*)

lemma *rel-L1-conv* [*simp*]:
rel-L1 (*Normal s*) (*r*, *t*) = ((∃ *x*. *r* = *Result x*) ∧ *t* = *s*)
rel-L1 (*Abrupt s*) (*r*, *t*) = ((∃ *x*. *r* = *Exn x*) ∧ *t* = *s*)
rel-L1 (*Fault e*) *x* = *False*
rel-L1 Stuck x = *False*
by (*auto simp add*: *rel-L1-def*)

lemma *refines-simpl-rel-L1I*:
assumes *termi*: *succeeds m t* ⟹ *ct* ⟹ Γ ⊢ *c* ↓ *Normal s*
assumes *sim-Normal*: ⋀*s′*. *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *Normal s′*
⟹ ∃ *r*. *reaches m t* (*Result r*) *s′*
assumes *sim-Abrupt*: ⋀*s′*. *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *Abrupt s′*
⟹ ∃ *r*. *reaches m t* (*Exn r*) *s′*
assumes *sim-Fault*: ⋀*e*. *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *Fault e*
⟹ *e* ∈ {*AssumeError*, *StackOverflow*}
assumes *sim-Stuck*: *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *Stuck*
⟹ *False*
shows *refines-simpl ct* Γ *c m s t rel-L1*
apply (*rule refines-simplI* [*OF termi*], *assumption*, *assumption*)
apply (*simp add*: *rel-L1-def*)
subgoal for *s′*
using *sim-Normal sim-Abrupt sim-Fault sim-Stuck*
apply (*cases s′*)
apply (*fastforce split*: *prod.splits sum.splits*)+
done
done

lemma *L1corres-refines-simpl*:
L1corres ct Γ *m c* ⟹ *refines-simpl ct* Γ *c m s s rel-L1*
apply (*clarsimp simp add*: *L1corres-def refines-simpl-def rel-L1-def split*: *xstate.splits prod.splits sum.splits*)
by (*smt* (*verit*) *Inl-Inr-False xstate.distinct*(*1*) *xstate.inject*(*1*) *xstate.inject*(*2*))

lemma *refines-simpl-L1corres*:
assumes ⋀*s*. *refines-simpl ct* Γ *c m s s rel-L1*
shows *L1corres ct* Γ *m c*
using *assms*
apply (*force simp add*: *L1corres-def refines-simpl-def rel-L1-def split*: *xstate.splits prod.splits sum.splits*)
done

theorem *L1corres-refines-simpl-conv*:
L1corres ct Γ *m c* ⟷ (∀ *s*. *refines-simpl ct* Γ *c m s s rel-L1*)
using *L1corres-refines-simpl refines-simpl-L1corres*
by *metis*

lemma *refines-simpl-DynCom*:
refines-simpl ct Γ (*c s*) *m s t R* ⟹ *refines-simpl ct* Γ (*DynCom c*) *m s t R*
by (*auto simp add*: *refines-simpl-def terminates.DynCom elim*: *exec-Normal-elim-cases*(*12*))

lemma *refines-simpl-StackOverflow*:
assumes *c*: *s* ∈ *g* ⟹ *refines-simpl ct* Γ *c m s t R*
shows *refines-simpl ct* Γ (*Guard StackOverflow g c*) *m s t R*
using *c*
by (*auto simp add*: *refines-simpl-def elim*: *exec-Normal-elim-cases intro*: *terminates.intros*)

lemma *refines-simpl-rel-L1-bind*:
fixes *m1*:: ('*e*, '*a*, '*s*) *exn-monad*
fixes *m2*:: '*a* ⇒ ('*e*, '*b*, '*s*) *exn-monad*
assumes *c1*: *refines-simpl ct* Γ *c1 m1 s s rel-L1*
assumes *c2*: ⋀*r s'*. *succeeds m1 s* ⟹ Γ ⊢ ⟨*c1*, *Normal s*⟩ ⇒ *Normal s'* ⟹ *reaches m1 s* (*Result r*) *s'* ⟹
refines-simpl ct Γ *c2* (*m2 r*) *s' s' rel-L1*
shows *refines-simpl ct* Γ (*c1*;;*c2*) (*m1* >>= *m2*) *s s rel-L1*
proof (*rule refines-simpl-rel-L1I*)
assume *nofail*: *succeeds* (*m1* >>= *m2*) *s*
assume *ct*: *ct*
from *nofail* **have** *nofail-m1*: *succeeds m1 s*
by (*simp add*: *succeeds-bind*)
with *ct c1* **have** *term-c1*: Γ⊢*c1* ↓ *Normal s*
by (*simp add*: *refines-simpl-def*)
then
show Γ⊢*c1*;;*c2* ↓ *Normal s*
proof (*rule terminates.intros*, *intro allI impI*)
fix *s'*
assume *exec-c1*: Γ⊢ ⟨*c1*,*Normal s*⟩ ⇒ *s'*
show Γ⊢*c2* ↓ *s'*
proof (*cases s'*)
case (*Normal s1*)
with *exec-c1 c1 nofail-m1 term-c1 ct* **obtain** *r* **where** *sim1*: *reaches m1 s* (*Result r*) *s1*
by (*force simp add*: *rel-L1-def refines-simpl-def*)
from *c2* [*OF nofail-m1 exec-c1* [*simplified Normal*] *this*]
have *refines-simpl ct* Γ *c2* (*m2 r*) *s1 s1 rel-L1* .
with *ct Normal nofail sim1*
show *?thesis*
by (*simp add*: *refines-simpl-def reaches-bind succeeds-bind*)

```
    qed simp-all
  qed
next
  fix s'
  assume nofail:  succeeds (m1 >>= m2) s
  from nofail have nofail-m1: succeeds m1 s
    by (simp add: succeeds-bind)
  assume exec: Γ⊢ ⟨c1;;c2,Normal s⟩ ⇒ Normal s'
  then show ∃ r. reaches (m1 >>= m2) s (Result r) s'
  proof (cases)
    case (1 s1)
    hence exec-c1: Γ⊢ ⟨c1,Normal s⟩ ⇒ s1 and
          exec-c2: Γ⊢ ⟨c2, s1⟩ ⇒ Normal s' .
    from exec-c2 obtain s1' where s1': s1 = Normal s1'
      by (meson Normal-resultE)
    from exec-c1 c1 nofail-m1  obtain r1 where sim1: reaches m1 s (Result r1)
s1'
     by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(1) s1' refines-simpl-def
xstate.simps(7))


    from c2 [OF nofail-m1 exec-c1[simplified s1'] sim1]
    have refines-simpl ct Γ c2 (m2 r1) s1' s1' rel-L1 .
    with nofail exec-c2 obtain r2 where reaches (m2 r1) s1' (Result r2) s'
      by (smt (verit) empty-iff insertE rel-L1-conv(1) s1' sim1 refines-simpl-def
succeeds-bind
          reaches-bind xstate.simps(7))
    with sim1 nofail show ?thesis
      by (fastforce simp add: reaches-bind succeeds-bind)
  qed
next
  fix s'
  assume nofail: succeeds (m1 >>= m2) s
  from nofail have nofail-m1: succeeds m1 s
    by (simp add: succeeds-bind)
  assume exec: Γ⊢ ⟨c1;;c2,Normal s⟩ ⇒ Abrupt s'
  then show ∃ r. reaches (m1 >>= m2) s (Exn r) s'
  proof (cases)
    case (1 s1)
    hence exec-c1: Γ⊢ ⟨c1,Normal s⟩ ⇒ s1 and
          exec-c2: Γ⊢ ⟨c2,s1⟩ ⇒ Abrupt s' .

    show ?thesis
    proof (cases s1)
      case (Normal s1')
     with exec-c1 c1 nofail-m1  obtain r1 where sim1: reaches m1 s (Result r1)
s1'
      by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(1)  refines-simpl-def
xstate.simps(7))
```

```
      from c2 [OF nofail-m1 exec-c1[simplified Normal] sim1]
      have refines-simpl ct Γ c2 (m2 r1) s1' s1' rel-L1 .
      with nofail exec-c2 sim1 obtain r2 where reaches (m2 r1) s1' (Exn r2) s'
        by (smt (verit) Normal empty-iff insertE rel-L1-conv(2) refines-simpl-def
succeeds-bind xstate.simps(11))
      with sim1 nofail show ?thesis
        by (fastforce simp add: reaches-bind succeeds-bind)
    next
      case (Abrupt s1')
      with exec-c1 c1 nofail-m1  obtain r1 where sim1: reaches m1 s (Exn r1)
s1'
      by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(2) refines-simpl-def
xstate.distinct(7))
      from Abrupt exec-c2 have s' = s1'
        by (meson Abrupt-end xstate.inject(2))
      with nofail
      show ?thesis
        apply (clarsimp simp add: reaches-bind succeeds-bind Exn-def)
        using sim1 by fastforce
    next
      case (Fault x)
      with exec-c2 show ?thesis
        by (meson Fault-end xstate.distinct(7))
    next
      case Stuck
      with exec-c2 show ?thesis
        using noStuck-startD' by blast
    qed
  qed
next
  fix e
  assume nofail:  succeeds (m1 >>= m2) s
  from nofail have nofail-m1: succeeds m1 s
    by (simp add: succeeds-bind)
  assume exec: Γ⊢ ⟨c1;;c2,Normal s⟩ ⇒ Fault e
  then show e ∈ {AssumeError, StackOverflow}
  proof (cases)
    case (1 s1)
    hence exec-c1: Γ⊢ ⟨c1,Normal s⟩ ⇒ s1 and
          exec-c2: Γ⊢ ⟨c2,s1⟩ ⇒ Fault e .
   show ?thesis
    proof (cases s1)
      case (Normal s1')
      with exec-c1 c1 nofail-m1  obtain r1 where sim1: reaches m1 s (Result r1)
s1'
      by (metis (no-types, lifting) empty-iff insertE rel-L1-conv(1)  refines-simpl-def
xstate.simps(7))
      from c2 [OF nofail-m1 exec-c1[simplified Normal] sim1]
      have refines-simpl ct Γ c2 (m2 r1) s1' s1' rel-L1 .
```

with *nofail exec-c2 sim1* **show** $e \in \{AssumeError, StackOverflow\}$
by (*metis* (*no-types*, *lifting*) *Normal insert-iff rel-L1-conv*(3) *refines-simpl-def singleton-iff*
succeeds-bind xstate.inject(3))
next
case (*Abrupt s1'*)
with *exec-c2* **show** *?thesis*
by (*metis Abrupt-end xstate.distinct*(7))
next
case (*Fault x*)
with *exec-c2 c1 exec-c1* **show** *?thesis*
by (*metis exec-Normal-elim-cases*(1) *insert-iff nofail-m1 rel-L1-conv*(3) *refines-simpl-def singleton-iff xstate.inject*(3))
next
case *Stuck*
with *exec-c2* **show** *?thesis*
using *noStuck-startD'* **by** *blast*
qed
qed
next
assume *nofail*: *succeeds* (*m1* >>= *m2*) *s*
from *nofail* **have** *nofail-m1*: *succeeds m1 s*
by (*simp add*: *succeeds-bind*)
assume *exec*: $\Gamma\vdash \langle c1;;c2,Normal\ s\rangle \Rightarrow Stuck$
then show *False*
proof (*cases*)
case (*1 s1*)
hence *exec-c1*: $\Gamma\vdash \langle c1,Normal\ s\rangle \Rightarrow s1$ **and**
exec-c2: $\Gamma\vdash \langle c2,s1\rangle \Rightarrow Stuck$ **.**
show *?thesis*
proof (*cases s1*)
case (*Normal s1'*)
with *exec-c1 c1 nofail-m1* **obtain** *r1* **where** *sim1*: *reaches m1 s* (*Result r1*) *s1'*
by (*metis* (*no-types*, *lifting*) *empty-iff insertE rel-L1-conv*(1) *refines-simpl-def xstate.simps*(7))
from *c2* [*OF nofail-m1 exec-c1*[*simplified Normal*] *sim1*]
have *refines-simpl ct* Γ *c2* (*m2 r1*) *s1' s1' rel-L1* **.**
with *nofail exec-c2 sim1* **show** *False*
by (*metis Normal insertE rel-L1-conv*(4) *refines-simpl-def singletonD succeeds-bind xstate.simps*(15))
next
case (*Abrupt s1'*)
with *exec-c2* **show** *?thesis*
by (*metis Abrupt-end xstate.distinct*(10))
next
case (*Fault x*)
with *exec-c2* **show** *?thesis*

```
      by (metis Fault-end isFault-simps(4) not-isFault-iff)
    next
      case Stuck
      with exec-c2 c1 exec-c1 show ?thesis
        by (metis insert-iff nofail-m1 rel-L1-conv(4) refines-simpl-def singletonD
xstate.simps(15))
    qed
  qed
qed


lemma refines-simpl-rel-L1-catch:
  assumes L: refines-simpl ct Γ L' L s s rel-L1
  assumes R: ⋀s. refines-simpl ct Γ R' R s s rel-L1
  shows refines-simpl ct Γ (Catch L' R') (L1-catch L R) s s rel-L1
proof (rule refines-simpl-rel-L1I)
  assume nofault: succeeds (L1-catch L R) s
  assume ct: ct
  show Γ⊢TRY L' CATCH R' END ↓ Normal s
  proof (rule terminates.intros, safe)
    show Γ⊢L' ↓ Normal s
      using nofault ct L
      by (simp add: refines-simpl-def L1-catch-def rel-L1-def succeeds-catch)
  next
    fix s'
    assume  Γ⊢ ⟨L',Normal s⟩ ⇒ Abrupt s'
    then show Γ⊢R' ↓ Normal s'
      using nofault ct L R
     by (fastforce simp add: refines-simpl-def L1-catch-def rel-L1-def succeeds-catch)
  qed
next
  fix s'
  assume nofault:  succeeds (L1-catch L R) s
  assume exec: Γ⊢ ⟨TRY L' CATCH R' END,Normal s⟩ ⇒ Normal s'
  then show ∃r. reaches (L1-catch L R) s (Result r) s'
  proof (cases)
    case (1 s1)
    hence exec-L': Γ⊢ ⟨L',Normal s⟩ ⇒ Abrupt s1 and
          exec-R': Γ⊢ ⟨R',Normal s1⟩ ⇒ Normal s'.
    from nofault L R exec-L' obtain
      nofault-L: succeeds L s and
      nofault-R: succeeds R s1
      by (smt (verit, best) L1-defs(5) case-xval-simps(1) insertE refines-simpl-def
          rel-L1-conv(2) singletonD succeeds-catch xstate.simps(11))
    from nofault-L exec-L' L obtain r1 where r1: reaches L s (Exn r1) s1
      by (metis (no-types, lifting) insertE rel-L1-conv(2) refines-simpl-def
          singletonD xstate.distinct(7))
    from r1 exec-R' R obtain r2 where  reaches R s1 (Result r2) s'
      by (metis (no-types, lifting) insertE nofault-R rel-L1-conv(1)
```

refines-simpl-def singletonD xstate.simps(*7*))
with *r1 nofault* **show** *?thesis*
by (*fastforce simp add*: *reaches-catch succeeds-catch L1-defs*)
next
case *2*
have *exec-L'*: Γ⊢ ⟨*L'*,*Normal s*⟩ ⇒ *Normal s'* **by** *fact*
from *nofault L R exec-L'* **obtain**
nofault-L: *succeeds L s*
by (*simp add*: *L1-defs*(*5*) *succeeds-catch*)
with *L exec-L'* **obtain** *r1* **where** *reaches L s* (*Result r1*) *s'*
by (*metis* (*no-types*, *lifting*) *insertE rel-L1-conv*(*1*) *refines-simpl-def singletonD xstate.distinct*(*3*))
then show *?thesis* **using** *nofault*
by (*fastforce simp add*: *L1-catch-def succeeds-catch reaches-catch*)
qed
next
fix *s'*
assume *nofault*: *succeeds* (*L1-catch L R*) *s*
assume *exec*: Γ⊢ ⟨*TRY L' CATCH R' END*,*Normal s*⟩ ⇒ *Abrupt s'*
then show ∃ *r*. *reaches* (*L1-catch L R*) *s* (*Exn r*) *s'*
proof (*cases*)
case (*1 s1*)
hence *exec-L'*: Γ⊢ ⟨*L'*,*Normal s*⟩ ⇒ *Abrupt s1* **and**
exec-R': Γ⊢ ⟨*R'*,*Normal s1*⟩ ⇒ *Abrupt s'* **.**
from *nofault L R exec-L'* **obtain**
nofault-L: *succeeds L s* **and**
nofault-R: *succeeds R s1*
by (*fastforce simp add*: *L1-catch-def refines-simpl-def reaches-catch succeeds-catch*)

from *nofault-L exec-L' L* **obtain** *r1* **where** *r1*: *reaches L s* (*Exn r1*) *s1*
by (*metis* (*no-types*, *lifting*) *insertE rel-L1-conv*(*2*) *refines-simpl-def singletonD xstate.distinct*(*7*))
from *nofault-R r1 exec-R' R* **obtain** *r2* **where** *reaches R s1* (*Exn r2*) *s'*
by (*metis* (*no-types*, *lifting*) *empty-iff insertE rel-L1-conv*(*2*) *refines-simpl-def xstate.distinct*(*7*))
with *r1 nofault* **show** *?thesis*
by (*fastforce simp add*: *L1-catch-def succeeds-catch reaches-catch*)
next
case *2*
then show *?thesis* **by** *simp*
qed
next
fix *e*
assume *nofault*: *succeeds* (*L1-catch L R*) *s*
assume *exec*: Γ⊢ ⟨*TRY L' CATCH R' END*,*Normal s*⟩ ⇒ *Fault e*
then show *e* ∈ {*AssumeError*, *StackOverflow*}
proof (*cases*)
case (*1 s1*)

```
    hence exec-L': Γ⊢ ⟨L',Normal s⟩ ⇒ Abrupt s1 and
          exec-R': Γ⊢ ⟨R',Normal s1⟩ ⇒ Fault e .
    from nofault L R exec-L' obtain
      nofault-L: succeeds L s and
      nofault-R: succeeds R s1
        by (fastforce simp add: L1-catch-def refines-simpl-def reaches-catch succeeds-catch)
    from nofault-L exec-L' L obtain r1 where r1: reaches L s (Exn r1) s1
      by (metis (no-types, lifting) insertE rel-L1-conv(2) refines-simpl-def
          singletonD xstate.distinct(7))
    from nofault-R r1 exec-R' R show ?thesis
     by (metis insert-iff rel-L1-conv(3) refines-simpl-def singletonD xstate.inject(3))
  next
    case 2
    have exec-L': Γ⊢ ⟨L',Normal s⟩ ⇒ Fault e by fact
    from nofault L R exec-L' obtain
      nofault-L: succeeds L s
      by (simp add: L1-defs(5) succeeds-catch)
    with L exec-L' show ?thesis
       by (metis insertCI insertE rel-L1-conv(3) refines-simpl-def singleton-iff xstate.inject(3))
  qed
next
  assume nofault: succeeds (L1-catch L R) s
  assume exec: Γ⊢ ⟨TRY L' CATCH R' END,Normal s⟩ ⇒ Stuck
  then show False
  proof (cases)
    case (1 s1)
    hence exec-L': Γ⊢ ⟨L',Normal s⟩ ⇒ Abrupt s1 and
          exec-R': Γ⊢ ⟨R',Normal s1⟩ ⇒ Stuck .
    from nofault L R exec-L' obtain
      nofault-L: succeeds L s and
      nofault-R: succeeds R s1
        by (fastforce simp add: L1-catch-def refines-simpl-def reaches-catch succeeds-catch)

    from nofault-R exec-R' R show ?thesis
     by (metis empty-iff insertE rel-L1-conv(4) refines-simpl-def xstate.distinct(11))
  next
    case 2
    have exec-L': Γ⊢ ⟨L',Normal s⟩ ⇒ Stuck by fact
    from nofault L R exec-L' obtain
      nofault-L: succeeds L s
      by (simp add: L1-defs(5) succeeds-catch)
    with L exec-L' show ?thesis
     by (metis insertE rel-L1-conv(4) refines-simpl-def singletonD xstate.distinct(11))
  qed
qed
```

lemmas *refines-simpl-cleanup = L1corres-refines-simpl [OF L1-corres-cleanup]*
lemmas *refines-simpl-cleanup-throw = L1corres-refines-simpl [OF L1-corres-cleanup-throw]*
lemmas *refines-simpl-spec-cleanup = L1corres-refines-simpl [OF L1-corres-spec-cleanup]*
lemmas *refines-simpl-spec-cleanup-throw = L1corres-refines-simpl [OF L1-corres-spec-cleanup-throw]*

lemma *refines-simpl-rel-L1-on-exit'*:
fixes *m:: 's L1-monad*
assumes *m-c*: *refines-simpl ct Γ c m s s rel-L1*
shows *refines-simpl ct Γ (On-Exit c (Basic cleanup)) (on-exit m {(s,t). t = cleanup s}) s s rel-L1*
unfolding *on-exit-catch-conv*
apply (*simp add*: *On-Exit-def*)
apply (*rule refines-simpl-rel-L1-bind*)
apply (*simp add*: *L1-catch-def* [*symmetric*])
apply (*rule refines-simpl-rel-L1-catch* [*OF m-c*])
apply (*rule refines-simpl-cleanup-throw* [*simplified*])
apply (*rule refines-simpl-cleanup* [*simplified*])
done

lemma *refines-simpl-rel-L1-on-exit*:
fixes *m:: 's L1-monad*
assumes *m-c*: *refines-simpl ct Γ c m s s rel-L1*
shows *refines-simpl ct Γ (On-Exit c (com.Spec cleanup)) (on-exit m cleanup) s s rel-L1*
apply (*simp add*: *on-exit-catch-conv On-Exit-def*)
apply (*rule refines-simpl-rel-L1-bind*)
apply (*simp add*: *L1-catch-def* [*symmetric*])
apply (*rule refines-simpl-rel-L1-catch* [*OF m-c*])
apply (*rule refines-simpl-spec-cleanup-throw* [*simplified*])
apply (*rule refines-simpl-spec-cleanup* [*simplified*])
done

named-theorems *L1corres-with-fresh-stack-ptr*

context *stack-heap-state*
begin
lemma *refines-simpl-rel-L1-with-fresh-stack-ptr*:
fixes *m:: 'a::mem-type ptr ⇒ 's L1-monad*
assumes *c-m*: *⋀p s. refines-simpl ct Γ (c p) (m p) s s rel-L1*
shows *refines-simpl ct Γ (With-Fresh-Stack-Ptr n I c) (with-fresh-stack-ptr n I m) s s rel-L1*
apply (*simp add*: *with-fresh-stack-ptr-def With-Fresh-Stack-Ptr-def*)
apply (*rule refines-simpl-StackOverflow*)
apply (*rule refines-simpl-DynCom*)
apply (*rule refines-simpl-rel-L1-bind*)

```
  subgoal
    apply (rule refines-simpl-rel-L1I)
    subgoal
      by (simp add: terminates.Spec)
    subgoal for s'
      apply (erule exec-Normal-elim-cases)
      subgoal for t
        by (auto simp add: succeeds-assume-result-and-state)
      by auto
    subgoal for s'
      by (meson exec-Normal-elim-cases(7) xstate.distinct(9) xstate.simps(5))
    subgoal for e
      by (meson exec-Normal-elim-cases(7) xstate.distinct(11) xstate.simps(7))
    subgoal
      apply (erule exec-Normal-elim-cases)
      using Ex-list-of-length by auto blast
    done
  apply (rule refines-simpl-DynCom)
  apply (clarsimp)
  apply (simp add: stack-allocs-allocated-ptrs)
  apply (rule refines-simpl-rel-L1-on-exit[OF c-m])
  done

lemma L1corres-with-fresh-stack-ptr[L1corres-with-fresh-stack-ptr]:
  fixes m:: 'a::mem-type ptr ⇒ 's L1-monad
  assumes c-m: ⋀p. L1corres ct Γ (m p) (c p)
  shows L1corres ct Γ (with-fresh-stack-ptr n I m) (With-Fresh-Stack-Ptr n I c)
  apply (rule refines-simpl-L1corres)
  apply (rule refines-simpl-rel-L1-with-fresh-stack-ptr)
  apply (rule L1corres-refines-simpl)
  apply (rule c-m)
  done
end




definition UNDEFINED-FUNCTION ≡ False

definition
  undefined-function-body :: ('a, int, strictc-errortype) com
where
  undefined-function-body ≡ Guard UndefinedFunction {x. UNDEFINED-FUNCTION}
SKIP



definition
```

init-return-undefined-function-body::(('*a* ⇒ '*a*) ⇒ (('*g*, '*l*, '*e*, '*z*) *state-scheme* ⇒ ('*g*, '*l*, '*e*, '*z*) *state-scheme*))
⇒ (('*g*, '*l*, '*e*, '*z*) *state-scheme*, *int*, *strictc-errortype*) *com*
where
init-return-undefined-function-body ret ≡ *Seq* (*lvar-nondet-init ret*) (*Guard UndefinedFunction* {*x*. *UNDEFINED-FUNCTION*} *SKIP*)

lemma *L1corres-undefined-call*:
L1corres ct Γ ((*L1-seq* (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*})) *L1-skip*)) (*Call X'*)
by (*clarsimp simp*: *L1corres-def L1-defs UNDEFINED-FUNCTION-def*)

lemma *L1-UNDEFINED-FUNCTION-fail*: (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*})) = *L1-fail*
apply (*clarsimp simp add*: *L1-defs UNDEFINED-FUNCTION-def*)
by (*simp add*: *run-guard spec-monad-ext*)

lemma *L1-seq-fail*: *L1-seq L1-fail X* = *L1-fail*
apply (*clarsimp simp add*: *L1-defs*)
done

lemma *L1-seq-init-fail*: (*L1-seq* (*L1-init ret*) *L1-fail*) = *L1-fail*
apply (*clarsimp simp add*: *L1-defs*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L1-corres-L1-fail*: *L1corres ct* Γ *L1-fail X*
by (*simp add*: *L1corres-def L1-defs*)

lemma *L1corres-init-return-undefined-call*:
L1corres ct Γ (*L1-seq* (*L1-init ret*) ((*L1-seq* (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*})) *L1-skip*))) (*Call X'*)
by (*simp only*: *L1-UNDEFINED-FUNCTION-fail L1-seq-fail L1-seq-init-fail L1-corres-L1-fail*)

named-theorems *L1unfold*

named-theorems *L1except*

lemma *signed-bounds-one-to-nat*: *n* <*s* *1* ⟹ *0* ≤*s* *n* ⟹ *unat n* = *0*
by (*metis signed.leD unat-1-0 unat-gt-0 unsigned-eq-0-iff word-msb-0 word-sle-msb-le*)

lemma *signed-bounds-to-nat-boundsF*: *n* <*s* *numeral B* ⟹ *0* ≤*s* *n* ⟹ *unat n* < *numeral B*
by (*metis linorder-not-less of-nat-numeral signed.leD unat-less-helper word-msb-0 word-sle-msb-le*)

lemma *word-bounds-to-nat-boundsF*: $(n::'a::len\ word) < numeral\ B \Longrightarrow 0 \leq s\ n \Longrightarrow unat\ n < numeral\ B$
by (*simp add*: *unat-less-helper*)

lemma *word-bounds-one-to-nat*: $(n::'a::len\ word) < 1 \Longrightarrow 0 \leq s\ n \Longrightarrow unat\ n = 0$
by (*simp add*: *unat-less-helper*)

lemma *monotone-L1-seq-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* $(\leq)$ *X*
assumes *mono-Y*: *monotone R* $(\leq)$ *Y*
shows *monotone R* $(\leq)$
$(\lambda f.\ (L1\text{-}seq\ (X\ f)\ (Y\ f)))$
unfolding *L1-defs*
apply (*intro partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-seq-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* $(\geq)$ *X*
assumes *mono-Y*: *monotone R* $(\geq)$ *Y*
shows *monotone R* $(\geq)$
$(\lambda f.\ (L1\text{-}seq\ (X\ f)\ (Y\ f)))$
unfolding *L1-defs*
apply (*intro partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-catch-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* $(\leq)$ *X*
assumes *mono-Y*: *monotone R* $(\leq)$ *Y*
shows *monotone R* $(\leq)$
$(\lambda f.\ (L1\text{-}catch\ (X\ f)\ (Y\ f)))$
unfolding *L1-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-catch-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* $(\geq)$ *X*
assumes *mono-Y*: *monotone R* $(\geq)$ *Y*
shows *monotone R* $(\geq)$
$(\lambda f.\ (L1\text{-}catch\ (X\ f)\ (Y\ f)))$
unfolding *L1-defs*
apply (*rule partial-function-mono*)

apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-condition-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: *monotone R* (≤) *Y*
shows *monotone R* (≤)
(λ*f*. (*L1-condition C* (*X f*) (*Y f*)))
unfolding *L1-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-condition-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: *monotone R* (≥) *Y*
shows *monotone R* (≥)
(λ*f*. (*L1-condition C* (*X f*) (*Y f*)))
unfolding *L1-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-L1-guarded-le* [*partial-function-mono*]:
assumes *mono-X* [*partial-function-mono*]: *monotone R* (≤) *X*
shows *monotone R* (≤)
(λ*f*. (*L1-guarded C* (*X f*)))
unfolding *L1-guarded-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-L1-guarded-ge* [*partial-function-mono*]:
assumes *mono-X* [*partial-function-mono*]: *monotone R* (≥) *X*
shows *monotone R* (≥)
(λ*f*. (*L1-guarded C* (*X f*)))
unfolding *L1-guarded-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-L1-while-le* [*partial-function-mono*]:
assumes *mono-B*: *monotone R* (≤) (λ*f*. *B f*)
shows *monotone R* (≤) (λ*f*. *L1-while C* (*B f*))
unfolding *L1-defs*

apply (*rule partial-function-mono*)
apply (*rule mono-B*)
done

lemma *monotone-L1-while-ge* [*partial-function-mono*]:
assumes *mono-B*: *monotone R* (≥) (λ*f*. *B f*)
shows *monotone R* (≥) (λ*f*. *L1-while C* (*B f*))
unfolding *L1-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-B*)
done

lemma *monotone-L1-call-le* [*partial-function-mono*]:
assumes *X*[*partial-function-mono*]: *monotone R* (≤) *X*
shows *monotone R* (≤)
(λ*f*. *L1-call scope-setup* (*X f*) *scope-teardown result-exn return-xf*)
unfolding *L1-call-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-L1-call-ge* [*partial-function-mono*]:
assumes *X*[*partial-function-mono*]: *monotone R* (≥) *X*
shows *monotone R* (≥)
(λ*f*. *L1-call scope-setup* (*X f*) *scope-teardown result-exn return-xf*)
unfolding *L1-call-def*
apply (*rule partial-function-mono*)+
done

end

18.1 Peep-hole L1 optimisations

theory *L1Peephole*
imports *L1Defs*
begin

named-theorems *L1opt*

lemma *L1-seq-assoc* [*L1opt*]: (*L1-seq* (*L1-seq X Y*) *Z*) = (*L1-seq X* (*L1-seq Y Z*))
apply (*clarsimp simp*: *L1-seq-def bind-assoc*)
done

lemma *L1-seq-skip* [*L1opt*]:
L1-seq A L1-skip = *A*
L1-seq L1-skip A = *A*
apply (*clarsimp simp*: *L1-seq-def L1-skip-def*)+
done

lemma *L1-condition-true* [*L1opt*]: *L1-condition* (λ-. *True*) *A B* = *A*
apply (*clarsimp simp*: *L1-condition-def condition-def*)
done

lemma *L1-condition-false* [*L1opt*]: *L1-condition* (λ-. *False*) *A B* = *B*
apply (*clarsimp simp*: *L1-condition-def condition-def*)
done

lemma *L1-condition-same* [*L1opt*]: *L1-condition C A A* = *A*
apply (*clarsimp simp*: *L1-defs condition-def spec-monad-ext run-bind*)
done

lemma *L1-fail-seq* [*L1opt*]: *L1-seq L1-fail X* = *L1-fail*
apply (*clarsimp simp*: *L1-seq-def L1-fail-def spec-monad-ext run-bind*)
done

lemma *L1-throw-seq* [*L1opt*]: *L1-seq L1-throw X* = *L1-throw*
apply (*clarsimp simp*: *L1-seq-def L1-throw-def*)
done

lemma *L1-fail-propagates* [*L1opt*]:
L1-seq L1-skip L1-fail = *L1-fail*
L1-seq (*L1-modify M*) *L1-fail* = *L1-fail*
L1-seq (*L1-spec S*) *L1-fail* = *L1-fail*
L1-seq (*L1-guard G*) *L1-fail* = *L1-fail*
L1-seq (*L1-init I*) *L1-fail* = *L1-fail*
L1-seq L1-fail L1-fail = *L1-fail*
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI*; *auto simp add*: *runs-to-iff*)+
done

lemma *L1-condition-distrib*:
L1-seq (*L1-condition C L R*) *X* = *L1-condition C* (*L1-seq L X*) (*L1-seq R X*)
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI*; *auto simp add*: *runs-to-iff*)
done

lemmas *L1-fail-propagate-condition* [*L1opt*] = *L1-condition-distrib* [**where** *X*=*L1-fail*]

lemma *L1-fail-propagate-catch* [*L1opt*]:
(*L1-seq* (*L1-catch L R*) *L1-fail*) = (*L1-catch* (*L1-seq L L1-fail*) (*L1-seq R L1-fail*))
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI*; *auto simp add*: *runs-to-iff*)
apply (*fastforce simp add*: *runs-to-def-old Exn-def*)+
done

lemma *L1-guard-false* [*L1opt*]:
L1-guard (λ-. *False*) = *L1-fail*
unfolding *L1-defs*
by (*simp add*: *guard-False-fail*)

lemma *L1-guard-true* [*L1opt*]:
L1-guard (λ-. *True*) = *L1-skip*
unfolding *L1-defs*
by (*simp add*: *spec-monad-ext run-guard*)

lemma *L1-condition-fail-lhs* [*L1opt*]:
L1-condition C L1-fail A = *L1-seq* (*L1-guard* (λ*s*. ¬ *C s*)) *A*
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI*; *auto simp add*: *runs-to-iff*)
done

lemma *L1-condition-fail-rhs* [*L1opt*]:
L1-condition C A L1-fail = *L1-seq* (*L1-guard C*) *A*
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI*; *auto simp add*: *runs-to-iff*)
done

lemma *L1-catch-fail* [*L1opt*]: *L1-catch L1-fail A* = *L1-fail*
unfolding *L1-defs*
apply (*rule spec-monad-eqI*; *auto simp add*: *runs-to-iff*)
done

lemma *L1-while-fail* [*L1opt*]: *L1-while C L1-fail* = *L1-guard* (λ*s*. ¬ *C s*)
unfolding *L1-defs*
apply (*subst whileLoop-unroll*)
apply (*rule spec-monad-ext*)
apply (*auto simp add*: *run-condition run-bind run-guard*)
done

lemma *whileLoop-succeeds-terminates-infinite*:
assumes *run* (*whileLoop* (λ-. *C*) (λ-. *skip*) ()) *s* ≠ ⊤
shows *C s* ⟹ *False*
using *assms* **by** (*induct rule*: *whileLoop-ne-top-induct*) *auto*

lemma *run-whileLoop-infinite*: *run* (*whileLoop* (λ-. *C*) (λ-. *skip*) ()) *s* = *run* (*guard* (λ*s*. ¬ *C s*)) *s*
proof (*cases C s*)
case *True*
show *?thesis*
apply (*rule antisym*)
subgoal

apply (*subst whileLoop-unroll*)
apply (*simp add*: *run-guard*)
done
subgoal
proof −
have *run* (*whileLoop* (λ-. *C*) (λ-. *skip*) ()) *s* = ⊤
using *True whileLoop-succeeds-terminates-infinite*[*of C s*] **by** *auto*
hence ¬ *succeeds* (*whileLoop* (λ-. *C*) (λ-. *skip*) ()) *s*
by (*simp add*: *succeeds-def*)
then show *?thesis*
by (*simp add*: *run-guard succeeds-def True top-post-state-def*)
qed
done
next
case *False*
then show *?thesis* **apply** (*subst whileLoop-unroll*) **by** (*simp add*: *run-guard*)
qed

lemma *whileLoop-infinite*: *whileLoop* (λ-. *C*) (λ-. *skip*) () = *guard* (λ*s*. ¬ *C s*)
apply (*rule spec-monad-ext*)
apply (*rule run-whileLoop-infinite*)
done

lemma *L1-while-infinite* [*L1opt*]: *L1-while C L1-skip* = *L1-guard* (λ*s*. ¬ *C s*)
unfolding *L1-defs*
apply (*rule whileLoop-infinite*)
done

lemma *L1-while-false* [*L1opt*]:
L1-while (λ-. *False*) *B* = *L1-skip*
by (*clarsimp simp*: *L1-while-def L1-skip-def*)

declare *ucast-id* [*L1opt*]
declare *scast-id* [*L1opt*]
declare *L1-set-to-pred-def* [*L1opt*]
declare *L1-rel-to-fun-def* [*L1opt*]

lemma *in-set-to-pred* [*L1opt*]: (λ*s*. *s* ∈ {*x*. *P x*}) = *P*
apply *simp*
done

lemma *in-set-if-then* [*L1opt*]: (*s* ∈ (*if P then A else B*)) = (*if P then* (*s* ∈ *A*) *else* (*s* ∈ *B*))
apply *simp*
done

lemma *Pair-unit-Image*[*L1opt*]: *Pair* () ‘ *S* ‘‘ {*x*} = {(*u*, *x'*). (*x*, *x'*) ∈ *S*}
by *auto*

lemmas *if-simps* =
if-x-Not if-Not-x if-cancel if-True if-False if-bool-simps

declare *empty-iff* [*L1opt*]
declare *UNIV-I* [*L1opt*]
declare *singleton-iff* [*L1opt*]
declare *if-simps* [*L1opt*]
declare *simp-thms* [*L1opt*]

lemma *L1-call-stop-cong*: (*L1-call f n g r*) = (*L1-call f n g r*)
by *simp*

lemma *L1-merge-assignments* : (*L1-seq* (*L1-modify f*)
(*L1-seq* (*L1-modify g*) *X*)) ≡ *L1-seq* (*L1-modify* (λ*s*. *g* (*f* *s*))) *X*
apply (*subst atomize-eq*)
apply (*unfold L1-defs*)
apply (*rule spec-monad-eqI*; *auto simp add*: *runs-to-iff*)
done

end

theory *SimplConv*
imports *L1Peephole*
begin

named-theorems *L1unfold*
declare *creturn-def* [*L1unfold*]
declare *creturn-void-def* [*L1unfold*]
declare *cbreak-def* [*L1unfold*]
declare *cgoto-def* [*L1unfold*]
declare *whileAnno-def* [*L1unfold*]
declare *ccatchbrk-def* [*L1unfold*]
declare *ccatchgoto-def* [*L1unfold*]
declare *ccatchreturn-def* [*L1unfold*]
declare *cexit-def* [*L1unfold*]

lemma *switch-alt-defs* [*L1unfold*]:
switch x [] ≡ *SKIP*

switch v ((a, b) # vs) ≡ Cond {s. v s ∈ a} b (switch v vs)
by *auto*

lemma *sless-positive* [*simp*]:
⟦ *a < n*; *n ≤ (2 ^ (len-of TYPE('a) − 1)) − 1* ⟧ ⟹ *(a :: ('a::{len}) word) <s n*
apply (*subst signed.less-le*)
apply *safe*
apply (*subst word-sle-msb-le*)
apply *safe*
apply (*force simp*: *not-msb-from-less*)
apply *simp*
apply *simp*
done

lemma *sle-positive* [*simp*]:
⟦ *a ≤ n*; *n ≤ (2 ^ (len-of TYPE('a) − 1)) − 1* ⟧ ⟹ *(a :: ('a::{len}) word) <=s n*
apply (*clarsimp simp*: *signed.le-less*)
done

lemmas [*L1except*] =
L1-set-to-pred-def in-set-to-pred in-set-if-then
L1-rel-to-fun-def Pair-unit-Image
L1-seq-assoc

end

theory *CorresXF*
imports
CCorresE
begin

definition *corresXF-simple st xf P M M′* ≡
∀ *s*. (*P s* ∧ *succeeds M (st s)*) ⟶ (∀ *r′ t′*. *reaches M′ s r′ t′* ⟶
reaches M (st s) (xf r′ t′) (st t′)) ∧ *succeeds M′ s*

definition *corresXF st ret-xf ex-xf P A C* ≡
∀ *s*. *P s* ∧ *succeeds A (st s)* ⟶
(∀ *r t*. *reaches C s r t* ⟶
(*case r of*

```
      Exn r    ⇒ reaches A (st s) (Exn (ex-xf r t)) (st t)
    | Result r ⇒ reaches A (st s) (Result (ret-xf r t)) (st t)))
  ∧ succeeds C s
```

definition *rel-XF st ret-xf ex-xf Q* ≡ λ(*r*, *t*) (*r′*, *t′*).
t′ = *st t* ∧
rel-xval (λ*e e′*. *e′* = *ex-xf e t*) (λ*v v′*. *v′* = *ret-xf v t*) *r r′* ∧
Q r t

lemma *corresXF-refines-iff*:
corresXF st ret-xf ex-xf P A C ⟷
(∀ *s*. *P s* ⟶ *refines C A s* (*st s*) (*rel-XF st ret-xf ex-xf* (λ- -. *True*)))
apply *standard*
subgoal
apply (*clarsimp simp add*: *corresXF-def refines-def-old rel-XF-def rel-xval.simps split*: *xval-splits*)
by (*metis Exn-def default-option-def exception-or-result-cases not-Some-eq*)
subgoal
apply (*fastforce simp add*: *corresXF-def refines-def-old rel-XF-def rel-xval.simps split*: *xval-splits*)
done
done

definition *corresXF-post st ret-xf ex-xf P Q A C* ≡
∀ *s*. *P s* ∧ *succeeds A* (*st s*) ⟶
(∀ *r t*. *reaches C s r t* ⟶ *Q s r t* ∧
(*case r of*
Exn r ⇒ *reaches A* (*st s*) (*Exn* (*ex-xf r t*)) (*st t*)
| *Result r* ⇒ *reaches A* (*st s*) (*Result* (*ret-xf r t*)) (*st t*)))
∧ *succeeds C s*

lemma *corresXF-post-refines-iff*:
corresXF-post st ret-xf ex-xf P Q A C ⟷
(∀ *s*. *P s* ⟶ *refines C A s* (*st s*) (*rel-XF st ret-xf ex-xf* (*Q s*)))
apply *standard*
subgoal
apply (*clarsimp simp add*: *corresXF-post-def refines-def-old rel-XF-def rel-xval.simps split*: *xval-splits*)
by (*metis Exn-def default-option-def exception-or-result-cases not-Some-eq*)
subgoal
apply (*fastforce simp add*: *corresXF-post-def refines-def-old rel-XF-def rel-xval.simps split*: *xval-splits*)
done
done

lemma *corresXF-post-to-corresXF*:
corresXF-post st ret-xf ex-xf P Q A C ⟹ *corresXF st ret-xf ex-xf P A C*

by (*auto simp add*: *corresXF-def corresXF-post-def*)

lemma *corresXF-corres-XF-post-conv*:
corresXF st ret-xf ex-xf P A C = *corresXF-post st ret-xf ex-xf P* (λ- - -. *True*) *A C*
by (*auto simp add*: *corresXF-def corresXF-post-def*)

lemma *corresXF-simple-corresXF*:
(*corresXF-simple st*
(λ*x s*. *case x of*
Exn r ⇒ *Exn* (*ex-state r s*)
| *Result r* ⇒ (*Result* (*ret-state r s*))) *P M M'*)
= (*corresXF st ret-state ex-state P M M'*)
unfolding *corresXF-simple-def corresXF-def*
by (*auto split*: *xval-splits*)

lemma *corresXF-simpleI*: ⟦
⋀*s' t' r'*. ⟦*P s'*; *succeeds M* (*st s'*); *reaches M' s' r' t'*⟧
⟹ *reaches M* (*st s'*) (*xf r' t'*) (*st t'*);
⋀*s'*. ⟦*P s'*; *succeeds M* (*st s'*) ⟧ ⟹ *succeeds M' s'*
⟧ ⟹ *corresXF-simple st xf P M M'*
apply *atomize*
apply (*clarsimp simp*: *corresXF-simple-def*)
done

lemma *corresXF-I*: ⟦
⋀*s' t' r'*. ⟦*P s'*; *succeeds M* (*st s'*); *reaches M' s'* (*Result r'*) *t'*⟧
⟹ *reaches M* (*st s'*) (*Result* (*ret-state r' t'*)) (*st t'*);
⋀*s' t' r'*. ⟦*P s'*; *succeeds M* (*st s'*); *reaches M' s'* (*Exn r'*) *t'*⟧
⟹ *reaches M* (*st s'*) (*Exn* (*ex-state r' t'*)) (*st t'*);
⋀*s'*. ⟦*P s'*; *succeeds M* (*st s'*) ⟧ ⟹ *succeeds M' s'*
⟧ ⟹ *corresXF st ret-state ex-state P M M'*
apply *atomize*
apply (*clarsimp simp*: *corresXF-def*)
subgoal for *s r t*
apply (*erule-tac x*=*s* **in** *allE*, *erule* (*1*) *impE*)
apply (*erule-tac x*=*s* **in** *allE*, *erule* (*1*) *impE*)
apply (*erule-tac x*=*s* **in** *allE*, *erule* (*1*) *impE*)
apply (*clarsimp split*: *xval-splits*)
apply *auto*
done
done

lemma *ccpo-prod-gfp-gfp*:
class.ccpo
(*prod-lub Inf Inf* :: ((*'a*::*complete-lattice* ∗ *'b* :: *complete-lattice*) *set* ⇒ -))

```
    (rel-prod (≥) (≥)) (mk-less (rel-prod (≥) (≥)))
  by (rule ccpo-rel-prodI ccpo-Inf)+

lemma admissible-mem: ccpo.admissible Inf (≥) (λA. x ∈ A)
  by (auto simp: ccpo.admissible-def)

lemma admissible-nondet-ord-corresXF:
  ccpo.admissible Inf (≥) (λA. corresXF st R E P A C)
  unfolding corresXF-def imp-conjL imp-conjR
  apply (intro admissible-all  admissible-conj admissible-imp)
  subgoal for s
  apply (rule ccpo.admissibleI)
    apply (clarsimp simp add: ccpo.admissible-def chain-def
        succeeds-def reaches-def split: prod.splits xval-splits)
    apply (intro conjI allI impI)
    subgoal
      apply transfer
      by (auto simp add: Inf-post-state-def top-post-state-def)
        (metis outcomes.simps(2) post-state.simps(3))
    subgoal
      apply transfer
      by (auto simp add: Inf-post-state-def top-post-state-def)
        (metis outcomes.simps(2) post-state.simps(3))
    done
  subgoal
    apply (rule ccpo.admissibleI)
    apply (clarsimp simp add: ccpo.admissible-def chain-def
        succeeds-def reaches-def split: prod.splits xval-splits)
    apply transfer
   apply (auto simp add: Inf-post-state-def top-post-state-def image-def vimage-def

        split: if-split-asm)
    done
  done



lemma corresXF-top: corresXF st ret-xf ex-xf P ⊤ C
  by (auto simp add: corresXF-def)

lemma admissible-nondet-ord-corresXF-post:
  ccpo.admissible Inf (≥) (λA. corresXF-post st R E P Q A C)
  unfolding corresXF-post-def imp-conjL imp-conjR
  apply (intro admissible-all  admissible-conj admissible-imp)
  subgoal
  apply (rule ccpo.admissibleI)
    apply (clarsimp simp add: ccpo.admissible-def chain-def
        succeeds-def reaches-def split: prod.splits xval-splits)
    apply (intro conjI allI impI)
```

subgoal
apply *transfer*
apply (*clarsimp simp add*: *Inf-post-state-def top-post-state-def*)
apply (*metis* (*no-types*, *lifting*) *INF-top-conv*(*1*) *Inf-post-state-def top-post-state-def*)
done
subgoal
apply *transfer*
by (*auto simp add*: *Inf-post-state-def top-post-state-def*)
(*metis outcomes.simps*(*2*) *post-state.simps*(*3*))
subgoal
apply *transfer*
apply (*clarsimp simp add*: *Inf-post-state-def top-post-state-def*)
apply (*metis* (*no-types*, *lifting*) *INF-top-conv*(*1*) *Inf-post-state-def top-post-state-def*)
done
subgoal
apply *transfer*
apply (*clarsimp simp add*: *Inf-post-state-def top-post-state-def image-def vimage-def*
split: *if-split-asm*)
by (*metis outcomes.simps*(*2*) *post-state.simps*(*3*))
done
subgoal
apply (*rule ccpo.admissibleI*)
apply (*clarsimp simp add*: *ccpo.admissible-def chain-def*
succeeds-def reaches-def split: *prod.splits xval-splits*)
apply *transfer*
apply (*auto simp add*: *Inf-post-state-def top-post-state-def image-def vimage-def*

split: *if-split-asm*)
done
done

lemma *corresXF-post-top*: *corresXF-post st ret-xf ex-xf P Q* ⊤ *C*
by (*auto simp add*: *corresXF-post-def*)

lemma *corresXF-assume-pre*:
⟦ ⋀*s s'*. ⟦ *P s'*; *s* = *st s'* ⟧ ⟹ *corresXF st xf-normal xf-exception P L R* ⟧ ⟹ *corresXF st xf-normal xf-exception P L R*
apply *atomize*
apply (*clarsimp simp*: *corresXF-def*)
apply *force*
done

lemma *corresXF-assume-fix-pre*:
⟦ ⋀*s s'*. ⟦ *P s'*; *s* = *st s'* ⟧ ⟹ *corresXF st xf-normal xf-exception* (λ*s*. *s* = *s'* ∧ *P s*) *L R* ⟧ ⟹ *corresXF st xf-normal xf-exception P L R*
apply *atomize*

apply (*clarsimp simp*: *corresXF-def*)
done

lemma *corresXF-guard-imp*:
⟦ *corresXF st xf-normal xf-exception Q f g*; ⋀*s*. *P s* ⟹ *Q s* ⟧
⟹ *corresXF st xf-normal xf-exception P f g*
apply (*clarsimp simp*: *corresXF-def*)
done

lemma *corresXF-return*:
⟦ ⋀*s*. ⟦ *P s* ⟧ ⟹ *xf-normal b s* = *a* ⟧ ⟹
corresXF st xf-normal xf-exception P (*return a*) (*return b*)
apply (*clarsimp simp*: *corresXF-def*)
done

lemma *corresXF-gets*:
⟦ ⋀*s*. *P s* ⟹ *ret* (*g s*) *s* = *f* (*st s*) ⟧ ⟹
corresXF st ret ex P (*gets f*) (*gets g*)
apply (*clarsimp simp*: *corresXF-def*)
done

lemma *corresXF-insert-guard*:
⟦ *corresXF st ret ex Q A C*; ⋀*s*. ⟦ *P s* ⟧ ⟹ *G* (*st s*) ⟶ *Q s* ⟧ ⟹
corresXF st ret ex P (*guard G* >>= (λ-. *A*)) *C*
apply (*auto simp*: *corresXF-def succeeds-guard succeeds-bind reaches-bind reaches-guard*)
done

lemma *corresXF-exec-abs-guard*:
corresXF st ret-xf ex-xf (λ*s*. *P s* ∧ *G* (*st s*)) (*A* ()) *C* ⟹ *corresXF st ret-xf ex-xf*
P (*guard G* >>= *A*) *C*
apply (*auto simp*: *corresXF-def succeeds-guard succeeds-bind reaches-bind reaches-guard*)
done

lemma *corresXF-simple-exec*:
⟦ *corresXF-simple st xf P A B*; *reaches B s r*′ *s*′; *succeeds A* (*st s*); *P s* ⟧
⟹ *reaches A* (*st s*) (*xf r*′ *s*′) (*st s*′)
apply (*fastforce simp*: *corresXF-simple-def*)
done

lemma *corresXF-simple-fail*:
⟦ *corresXF-simple st xf P A B*; ¬ *succeeds B s*; *P s* ⟧
⟹ ¬ *succeeds A* (*st s*)
apply (*fastforce simp*: *corresXF-simple-def*)
done

lemma *corresXF-simple-no-fail*:
⟦ *corresXF-simple st xf P A B*; *succeeds A* (*st s*); *P s* ⟧
⟹ *succeeds B s*
apply (*fastforce simp*: *corresXF-simple-def*)

done

lemma *corresXF-exec-normal*:
⟦ *corresXF st ret ex P A B*; *reaches B s* (*Result r′*) *s′*; *succeeds A* (*st s*); *P s* ⟧
⟹ *reaches A* (*st s*) (*Result* (*ret r′ s′*)) (*st s′*)
by (*auto simp add*: *corresXF-def split*: *xval-splits*)

lemma *corresXF-exec-except*:
⟦ *corresXF st ret ex P A B*; *reaches B s* (*Exn r′*) *s′*; *succeeds A* (*st s*); *P s* ⟧
⟹ *reaches A* (*st s*) (*Exn* (*ex r′ s′*)) (*st s′*)
by (*auto simp add*: *corresXF-def split*: *xval-splits*)

lemma *corresXF-exec-fail*:
⟦ *corresXF st ret ex P A B*; ¬ *succeeds B s*; *P s* ⟧
⟹ ¬ *succeeds A* (*st s*)
by (*auto simp add*: *corresXF-def split*: *xval-splits*)

lemma *corresXF-intermediate*:
⟦ *corresXF st ret-xf ex-xf P A′ C*;
corresXF id (λ*r s*. *r*) (λ*r s*. *r*) (λ*s*. ∃ *x*. *s* = *st x* ∧ *P x*) *A A′* ⟧ ⟹
corresXF st ret-xf ex-xf P A C
apply (*clarsimp simp*: *corresXF-def split*: *xval-splits*)
apply *fast*
done

lemma *corresXF-join*:
⟦ *corresXF st V E P L L′*; ⋀*x y*. *corresXF st V′ E* (*P′ x y*) (*R x*) (*R′ y*);
⋀*s*. *Q s* ⟹ *L′* ⋆ *s* ?{ λ*r t*. *case r of Exn* - ⇒ ⊤ | *Result v* ⇒ *P′* (*V v t*) *v t* }?;
⋀*s*. *Q s* ⟹ *P s* ⟧ ⟹
corresXF st V′ E Q (*L* >>= *R*) (*L′* >>= *R′*)
apply (*clarsimp simp add*: *corresXF-refines-iff split*: *xval-splits*)
apply (*rule refines-bind-bind-exn*[**where** *Q*=*rel-XF st V E* (λ- -. *True*) ⊓
(λ(*r*, *t*) *x*. *case r of Exn* - ⇒ ⊤ | *Result v* ⇒ *P′* (*V v t*) *v t*)])
subgoal
apply (*rule refines-strengthen1*[**where** *R*=*rel-XF st V E* (λ- -. *True*)])
by *auto*
by (*auto simp add*: *rel-XF-def*)

lemma *corresXF-join-xf-state-independent-same-state*:
⟦ *corresXF* (λ*s*. *s*) (λ*r s*. *V r*) (λ*r s*. *E r*) *P L L′*;
⋀*y*. *corresXF* (λ*s*. *s*) (λ*r s*. *V′ r*) (λ*r s*. *E r*) (*P′* (*V y*)) (*R* (*V y*)) (*R′ y*);
⋀*s*. *Q s* ⟹ *L* ⋆ *s* ?{ λ*r t*. *case r of Exn* - ⇒ ⊤ | *Result v* ⇒ *P′ v t* }?; ⋀*s*. *Q s*
⟹ *P s* ⟧ ⟹
corresXF (λ*s*. *s*) (λ*r s*. *V′ r*) (λ*r s*. *E r*) *Q* (*L* >>= *R*) (*L′* >>= *R′*)
apply (*clarsimp simp add*: *corresXF-refines-iff*)
apply (*rule refines-bind-bind-exn*[**where**
Q=*rel-XF* (λ*s*. *s*) (λ*r s*. *V r*) (λ*r s*. *E r*) (λ- -. *True*) ⊓
(λ*x* (*r*, *t*). *case r of Exn* - ⇒ ⊤ | *Result v* ⇒ *P′ v t*)])
subgoal

apply (*rule refines-strengthen2*[**where** *R*=*rel-XF* (λ*s*. *s*) (λ*r s*. *V r*) (λ*r s*. *E r*) (λ- -. *True*)])
by *auto*
by (*auto simp add*: *rel-XF-def*)

lemma *corresXF-except*:
⟦ *corresXF st V E P L L′*; ⋀*x y*. *corresXF st V E′* (*P′ x y*) (*R x*) (*R′ y*);
⋀*s*. *Q s* ⟹ *L′* • *s* ?{ λ*r s*. *case r of Exn r* ⇒ *P′* (*E r s*) *r s* | *Result* - ⇒ ⊤ }?;
⋀*s*. *Q s* ⟹ *P s* ⟧ ⟹
corresXF st V E′ Q (*L* <*catch*> *R*) (*L′* <*catch*> *R′*)
apply (*clarsimp simp add*: *corresXF-refines-iff*)
apply (*rule refines-catch* [**where** *Q*=(*rel-XF st V E* (λ- -. *True*)) ⊓
(λ(*r*, *s*) *x*. *case r of Exn r* ⇒ *P′* (*E r s*) *r s* | *Result* - ⇒ ⊤)])
subgoal
apply (*rule refines-strengthen1*[**where** *R*=*rel-XF st V E* (λ- -. *True*)])
by *auto*
apply (*auto simp add*: *rel-XF-def*)
done

lemma *corresXF-cond*:
⟦ *corresXF st V E P L L′*; *corresXF st V E P R R′*; ⋀*s*. *P s* ⟹ *A* (*st s*) = *A′ s* ⟧ ⟹
corresXF st V E P (*condition A L R*) (*condition A′ L′ R′*)
apply (*clarsimp simp add*: *corresXF-refines-iff*)
using *refines-condition* **by** *metis*

lemma *refines-assume-succeeds*: (*succeeds g t* ⟹ *refines f g s t R*) ⟹ *refines f g s t R*
by (*auto simp add*: *refines-def-old*)

lemma *corresXF-while*:
assumes *body-corres*: ⋀*x y*. *corresXF st ret ex* (λ*s*. *P x s* ∧ *y* = *ret x s*) (*A y*) (*B x*)
and *cond-match*: ⋀*s r*. *P r s* ⟹ *C r s* = *C′* (*ret r s*) (*st s*)
and *pred-inv*: ⋀*r s*. *P r s* ⟹ *C r s* ⟹ *succeeds* (*whileLoop C′ A* (*ret r s*)) (*st s*) ⟹
B r • *s* ?{ λ*r s*. *case r of Exn* - ⇒ *True* | *Result r* ⇒ *P r s* }?
and *init-match*: ⋀*s*. *P′ x s* ⟹ *y* = *ret x s*
and *pred-imply*: ⋀*s*. *P′ x s* ⟹ *P x s*
shows *corresXF st ret ex* (*P′ x*) (*whileLoop C′ A y*) (*whileLoop C B x*)
apply (*clarsimp simp add*: *corresXF-refines-iff*)
apply (*rule refines-assume-succeeds*)
subgoal for *s*
apply (*rule refines-mono* [*OF* - *refines-whileLoop′*
[**where** *R* = *rel-XF st ret ex* (λ*r s*. *case r of Exn* - ⇒ *True* | *Result r* ⇒ *P r s*) ⊓
(λ(*r*, *s*) -. *case r of Exn* - ⇒ *True* | *Result r* ⇒
succeeds (*whileLoop C′ A* (*ret r s*)) (*st s*))
and *C* = *C* **and** *C′*=*C′* **and** *B*=*B* **and** *B′*=*A* **and** *I*=*x* **and** *I′*=*y* **and**

s=*s* **and** *s'*=*st s*]])
subgoal by (*auto simp add*: *rel-XF-def*)
subgoal by (*auto simp add*: *rel-XF-def cond-match pred-imply*)
subgoal using *body-corres pred-inv*
apply (*subst* (*asm*) *rel-XF-def*)
apply (*clarsimp simp add*: *corresXF-refines-iff*)
apply (*rule refines-strengthen*[**where** *R*=*rel-XF st ret ex* (λ- -. *True*) **and**
G=λ*r s*. *case r of Exn* - ⇒ *True* | *Result r* ⇒ *succeeds* (*whileLoop C' A r*)
s])
subgoal by *auto*
apply *assumption*
subgoal
apply (*subst* (*asm*) (*3*) *whileLoop-unroll*)
apply (*auto simp add*: *succeeds-bind runs-to-partial-def-old split*: *xval-splits*)
done
apply (*auto simp*: *rel-XF-def rel-xval.simps*)
done
subgoal
by (*auto simp add*: *rel-XF-def init-match pred-imply*)
subgoal
by (*auto simp add*: *rel-XF-def rel-xval.simps rel-exception-or-result.simps*
Exn-def default-option-def split: *xval-splits*)
done
done

lemma *corresXF-name-pre*:
⟦ ⋀*s'*. *corresXF st ret ex* (λ*s*. *P s* ∧ *s* = *s'*) *A C* ⟧ ⟹
corresXF st ret ex P A C
by (*clarsimp simp*: *corresXF-def*)

lemma *corresXF-guarded-while-body*:
corresXF st ret ex P A B ⟹
corresXF st ret ex P
(*do*{ *r* ← *A*; - ← *guard* (*G r*); *return r* }) *B*
apply (*clarsimp simp add*: *corresXF-refines-iff*)
apply (*clarsimp simp add*: *refines-def-old succeeds-bind reaches-bind*)
by (*smt* (*verit*) *Exn-def case-exception-or-result-Exn case-exception-or-result-Result*

default-option-def the-Exception-simp the-Exception-Result exception-or-result-cases

is-Exception-simps(*1*) *is-Exception-simps*(*2*) *not-None-eq*)

lemma *whileLoop-succeeds-terminates-guard-body*:
assumes *B-succeeds*: ⋀*i s*. *succeeds* (*B i*) *s* ⟹ *succeeds* (*B' i*) *s*
assumes *B-reaches*: ⋀*i s r t*. *reaches* (*B' i*) *s r t* ⟹ *succeeds* (*B i*) *s* ⟹ *reaches*
(*B i*) *s r t*
assumes *termi*: *run* (*whileLoop C B I*) *s* ≠ ⊤
shows *run* (*whileLoop C B' I*) *s* ≠ ⊤
using *termi*

proof (*induct rule*: *whileLoop-ne-top-induct*)
case *step* **show** *?case* **unfolding** *top-post-state-def*
apply (*rule whileLoop-ne-Failure*)
using *step B-succeeds B-reaches*
apply (*clarsimp simp add*: *runs-to-def-old*)
by (*metis top-post-state-def*)
qed

lemma *whileLoop-succeeds-guard-body*:
assumes *B-succeeds*: $\bigwedge i\ s.\ succeeds\ (B\ i)\ s \Longrightarrow succeeds\ (B'\ i)\ s$
assumes *B-reaches*: $\bigwedge i\ s\ r\ t.\ reaches\ (B'\ i)\ s\ r\ t \Longrightarrow succeeds\ (B\ i)\ s \Longrightarrow reaches\ (B\ i)\ s\ r\ t$
assumes *termi*: *succeeds* (*whileLoop C B I*) *s*
shows *succeeds* (*whileLoop C B' I*) *s*
using *whileLoop-succeeds-terminates-guard-body* [*OF B-succeeds B-reaches*] *termi*
by (*auto simp*: *succeeds-def top-post-state-def*)

lemma *corresXF-guarded-while*:
assumes *body-corres*: $\bigwedge x\ y.\ corresXF\ st\ ret\ ex\ (\lambda s.\ P\ x\ s \wedge y = ret\ x\ s)\ (A\ y)\ (B\ x)$
and *cond-match*: $\bigwedge s\ r.\ [\![\ P\ r\ s;\ G\ (ret\ r\ s)\ (st\ s)\ ]\!] \Longrightarrow C\ r\ s = C'\ (ret\ r\ s)\ (st\ s)$
and *pred-inv*: $\bigwedge r\ s.\ P\ r\ s \Longrightarrow C\ r\ s \Longrightarrow succeeds\ (whileLoop\ C'\ A\ (ret\ r\ s))\ (st\ s) \Longrightarrow G\ (ret\ r\ s)\ (st\ s) \Longrightarrow$
$B\ r \bullet s\ ?\{\!|\ \lambda r\ s.\ case\ r\ of\ Exn\ \text{-} \Rightarrow True \mid Result\ r \Rightarrow G\ (ret\ r\ s)\ (st\ s) \longrightarrow P\ r\ s\ |\!\}$
and *pred-imply*: $\bigwedge s.\ [\![\ G\ y\ (st\ s);\ P'\ x\ s\ ]\!] \Longrightarrow P\ x\ s$
and *init-match*: $\bigwedge s.\ [\![\ G\ y\ (st\ s);\ P'\ x\ s\ ]\!] \Longrightarrow y = ret\ x\ s$
shows *corresXF st ret ex* (*P' x*)
(*do* {
- ← *guard* (*G y*);
whileLoop C' (λ*i*. (*do* {
r ← *A i*;
- ← *guard* (*G r*);
return r
})) *y*
})
(*whileLoop C B x*)
proof −

{fix *i s*
assume ∗: *succeeds* (*whileLoop C'* (λ*i*.
(*do* { *r* ← *A i*;
- ← *guard* (*G r*);
return r
})) *i*) *s*
have *succeeds* (*whileLoop C' A i*) *s*
apply (*rule whileLoop-succeeds-guard-body* [*OF* - - ∗])
subgoal by (*auto simp add*: *succeeds-bind*)

```
    subgoal apply (clarsimp simp add: reaches-bind succeeds-bind)
      by (smt (verit, best) dual-order.refl exception-or-result-split-asm le-boolD)
    done
  } note new-body-fails-more = this


  note new-body-corres = body-corres [THEN corresXF-guarded-while-body]

  show ?thesis
    apply (rule corresXF-exec-abs-guard)
    apply (rule corresXF-name-pre)
    apply (rule corresXF-assume-pre)
    apply clarsimp
    subgoal for s'
      apply (rule corresXF-guard-imp)
       apply (rule corresXF-while [where
            P=λx s. P x s ∧ G (ret x s) (st s) and P'=λx s. P' x s ∧ s = s'])
         apply (rule corresXF-guard-imp)
          apply (rule new-body-corres)
         apply (clarsimp)
        apply (clarsimp)
        apply (rule cond-match, simp, simp)
      subgoal for r s
        using pred-inv [of r s, OF - - new-body-fails-more ]
        apply clarsimp
        apply (subst (asm)  whileLoop-unroll)
        apply (clarsimp simp add: cond-match)
        apply (clarsimp simp add: succeeds-bind)
          apply (clarsimp simp add: runs-to-partial-def-old split: xval-splits, intro
conjI)
         apply (metis (mono-tags, lifting) body-corres corresXF-exec-normal)
        using body-corres corresXF-exec-normal by fastforce
      subgoal
        using init-match by auto
      subgoal
        using init-match pred-imply by auto
      subgoal by auto
      done
    done
qed




definition ac-corres st check-termination AF Γ rx ex G ≡
  λA B. ∀ s. (G s ∧ succeeds A (st s)) ⟶
```

$$
\begin{array}{l}
(\forall\, t.\ \Gamma \vdash \langle B,\ \mathit{Normal}\ s\rangle \Rightarrow t \longrightarrow \\
\quad (\mathit{case}\ t\ \mathit{of} \\
\quad\ \ \mathit{Normal}\ s' \Rightarrow \mathit{reaches}\ A\ (\mathit{st}\ s)\ (\mathit{Result}\ (\mathit{rx}\ s'))\ (\mathit{st}\ s') \\
\quad\ |\ \mathit{Abrupt}\ s' \Rightarrow \mathit{reaches}\ A\ (\mathit{st}\ s)\ (\mathit{Exn}\ (\mathit{ex}\ s'))\ (\mathit{st}\ s') \\
\quad\ |\ \mathit{Fault}\ e \Rightarrow e \in AF \\
\quad\ |\ \text{-} \Rightarrow \mathit{False})) \\
\ \wedge\ (\mathit{check\text{-}termination} \longrightarrow \Gamma \vdash B \downarrow \mathit{Normal}\ s)
\end{array}
$$

lemma *ccorresE-corresXF-merge*:
$[\![\ \mathit{ccorresE}\ \mathit{st1}\ \mathit{ct}\ \mathit{AF}\ \Gamma\ \top\ \mathit{G1}\ M\ B;$
$\mathit{corresXF}\ \mathit{st2}\ \mathit{rx}\ \mathit{ex}\ \mathit{G2}\ A\ M;$
$\bigwedge s.\ \mathit{st}\ s = \mathit{st2}\ (\mathit{st1}\ s);$
$\bigwedge r\ s.\ \mathit{rx}'\ s = \mathit{rx}\ r\ (\mathit{st1}\ s);$
$\bigwedge r\ s.\ \mathit{ex}'\ s = \mathit{ex}\ r\ (\mathit{st1}\ s);$
$\bigwedge s.\ G\ s \longrightarrow (s \in \mathit{G1} \wedge \mathit{G2}\ (\mathit{st1}\ s))\ ]\!] \Longrightarrow$
$\mathit{ac\text{-}corres}\ \mathit{st}\ \mathit{ct}\ \mathit{AF}\ \Gamma\ \mathit{rx}'\ \mathit{ex}'\ G\ A\ B$
apply (*unfold ac-corres-def*)
apply *clarsimp*
apply (*clarsimp simp*: *ccorresE-def*)
apply (*clarsimp simp*: *corresXF-def*)
apply (*erule allE*, *erule impE*, *force*)
apply (*erule allE*, *erule impE*, *force*)
apply *clarsimp*
apply (*erule allE*, *erule impE*, *fastforce*)
subgoal for *s t* **by** (*cases t*; *fastforce*)
done

lemma *corresXF-corresXF-merge*:
$[\![\ \mathit{corresXF}\ \mathit{st}\ \mathit{rx}\ \mathit{ex}\ P\ A\ B;\ \mathit{corresXF}\ \mathit{st'}\ \mathit{rx'}\ \mathit{ex'}\ P'\ B\ C\ ]\!] \Longrightarrow$
$\mathit{corresXF}\ (\mathit{st}\ o\ \mathit{st'})\ (\lambda \mathit{rv}\ s.\ \mathit{rx}\ (\mathit{rx'}\ \mathit{rv}\ s)\ (\mathit{st'}\ s))$
$(\lambda \mathit{rv}\ s.\ \mathit{ex}\ (\mathit{ex'}\ \mathit{rv}\ s)\ (\mathit{st'}\ s))\ (\lambda s.\ P'\ s \wedge P\ (\mathit{st'}\ s))\ A\ C$
apply (*clarsimp simp*: *corresXF-def split*: *xval-splits*)
apply *fastforce*
done

lemma *ac-corres-guard-imp*:
$[\![\ \mathit{ac\text{-}corres}\ \mathit{st}\ \mathit{ct}\ \mathit{AF}\ G\ \mathit{rx}\ \mathit{ex}\ P\ A\ C;\ \bigwedge s.\ P'\ s \Longrightarrow P\ s\ ]\!] \Longrightarrow \mathit{ac\text{-}corres}\ \mathit{st}\ \mathit{ct}\ \mathit{AF}$
$G\ \mathit{rx}\ \mathit{ex}\ P'\ A\ C$
apply *atomize*
apply (*clarsimp simp*: *ac-corres-def*)
done

lemma *corresXF-modify-local*:
$[\![\ \bigwedge s.\ \mathit{st}\ s = \mathit{st}\ (M\ s);\ \bigwedge s.\ P\ s \Longrightarrow \mathit{ret}\ ()\ (M\ s) = x\ ]\!]$

$\Longrightarrow$ *corresXF st ret ex P* (*return x*) (*modify M*)
by (*auto simp add*: *corresXF-def*)

lemma *corresXF-modify-global*:
⟦ $\bigwedge$*s. P s* $\Longrightarrow$ *M* (*st s*) = *st* (*M' s*) ⟧ $\Longrightarrow$
corresXF st ret ex P (*modify M*) (*modify M'*)
by (*auto simp add*: *corresXF-def*)

lemma *corresXF-select-modify*:
⟦ $\bigwedge$*s. P s* $\Longrightarrow$ *st s* = *st* (*M s*); $\bigwedge$*s. P s* $\Longrightarrow$ *ret* () (*M s*) ∈ *x* ⟧ $\Longrightarrow$
corresXF st ret ex P (*select x*) (*modify M*)
by (*auto simp add*: *corresXF-def*)

lemma *corresXF-select-select*:
⟦ $\bigwedge$*s a. st s* = *st* (*M* (*a*::('*a* ⇒ ('*a*::{*type*}))) *s*);
$\bigwedge$*s x.* ⟦ *P s*; *x* ∈ *b*⟧ $\Longrightarrow$ *ret x s* ∈ *a* ⟧ $\Longrightarrow$
corresXF st ret ex P (*select a*) (*select b*)
by (*auto simp add*: *corresXF-def*)

lemma *corresXF-modify-gets*:
⟦ $\bigwedge$*s. P s* $\Longrightarrow$ *st s* = *st* (*M s*); $\bigwedge$*s. P s* $\Longrightarrow$ *ret* () (*M s*) = *f* (*st* (*M s*)) ⟧ $\Longrightarrow$
corresXF st ret ex P (*gets f*) (*modify M*)
by (*auto simp add*: *corresXF-def*)

lemma *corresXF-guard*:
⟦ $\bigwedge$*s. P s* $\Longrightarrow$ *G' s* = *G* (*st s*) ⟧ $\Longrightarrow$ *corresXF st ret ex P* (*guard G*) (*guard G'*)
by (*auto simp add*: *corresXF-def*)

lemma *corresXF-fail*:
corresXF st return-xf exception-xf P fail X
by (*auto simp add*: *corresXF-def*)

lemma *corresXF-spec*:
⟦ $\bigwedge$*s s'*. ((*s*, *s'*) ∈ *A'*) = ((*st s*, *st s'*) ∈ *A*); *surj st* ⟧
$\Longrightarrow$ *corresXF st ret ex P* (*state-select A*) (*state-select A'*)
apply (*clarsimp simp add*: *corresXF-def*)
apply (*frule-tac y*=*undefined* **in** *surjD*)
apply (*clarsimp simp*: *image-def set-eq-UNIV*)
apply *metis*
done

lemma *corresXF-throw*:
⟦ $\bigwedge$*s. P s* $\Longrightarrow$ *E B s* = *A* ⟧ $\Longrightarrow$ *corresXF st V E P* (*throw A*) (*throw B*)
by (*auto simp add*: *corresXF-def Exn-def*)

lemma *corresXF-append-gets-abs*:
assumes *corres*: *corresXF st ret ex P L R*

and *consistent*: $\bigwedge s.\ P\ s \Longrightarrow R \cdot s\ ?\{\!|\lambda r\ s.\ case\ r\ of\ Exn\ -\ \Rightarrow \top \mid Result\ v \Rightarrow M\ (ret\ v\ s)\ (st\ s) = ret'\ v\ s\ |\!\}$
shows *corresXF st ret' ex P* $(L >>= (\lambda r.\ gets\ (M\ r)))\ R$
using *corres consistent*
apply (*clarsimp simp add*: *corresXF-refines-iff runs-to-partial-def-old refines-def-old*

succeeds-bind reaches-bind rel-XF-def rel-xval.simps split: *xval-splits*)
using *Exn-def* **by** *force*

lemma *corresXF-skipE*:
corresXF st ret ex P skip skip
by (*auto simp add*: *corresXF-def Exn-def*)

lemma *corresXF-id*:
corresXF id $(\lambda r\ s.\ r)$ $(\lambda r\ s.\ r)$ *P M M*
by (*fastforce simp*: *corresXF-def split*: *xval-splits*)

lemma *corresXF-cong*:
$[\![\bigwedge s.\ st\ s = st'\ s;$
$\bigwedge s\ r.\ ret\text{-}xf\ r\ s = ret\text{-}xf'\ r\ s;$
$\bigwedge s\ r.\ ex\text{-}xf\ r\ s = ex\text{-}xf'\ r\ s;$
$\bigwedge s.\ P\ s = P'\ s;$
$\bigwedge s\ s'.\ P'\ s' \Longrightarrow run\ A\ s = run\ A'\ s;$
$\bigwedge s.\ P'\ s \Longrightarrow run\ C\ s = run\ C'\ s\ ]\!] \Longrightarrow$
corresXF st ret-xf ex-xf P A C = *corresXF st' ret-xf' ex-xf' P' A' C'*
apply *atomize*
apply (*auto simp*: *corresXF-def reaches-def succeeds-def split*: *xval-splits*)
done

lemma *corresXF-exec-abs-select*:
$[\![x \in Q;\ x \in Q \Longrightarrow corresXF\ id\ rx\ ex\ P\ (A\ x)\ A'\ ]\!] \Longrightarrow corresXF\ id\ rx\ ex\ P\ (select\ Q >>= A)\ A'$
by (*fastforce simp add*: *corresXF-def succeeds-bind reaches-bind split*: *xval-splits*)

end

18.2 Hoare-Triples for L1 (internal use)

theory *L1Valid*
imports *L1Defs*
begin

definition
validE :: $('s \Rightarrow bool) \Rightarrow ('e, 'a, 's)\ exn\text{-}monad \Rightarrow$
$('a \Rightarrow 's \Rightarrow bool) \Rightarrow$
$('e \Rightarrow 's \Rightarrow bool) \Rightarrow bool$

where *validE P f Q E* $\equiv \forall s.\ P\ s \longrightarrow f \cdot s\ ?\{\!|\ \lambda v\ s.\ case\ v\ of\ Result\ r \Rightarrow Q\ r\ s \mid$

Exn e ⇒ E e s ⦘

open-bundle *validE-syntax*
begin
notation *validE* (‹(‹*open-block notation*=‹*mixfix L1 Hoare triple*››{⦇-⦈/ - /(⦇-⦈,/ ⦇-⦈))›)
end

lemma *hoareE-TrueI*: ⦇*P*⦈ *f* ⦇λ- -. *True*⦈, ⦇λ*r* -. *True*⦈
  by (*simp add*: *validE-def runs-to-partial-def-old split*: *xval-splits*)

lemma *combine-validE*: ⟦ ⦇ *P* ⦈ *x* ⦇ *Q* ⦈,⦇ *E* ⦈;
        ⦇ *P′* ⦈ *x* ⦇ *Q′* ⦈,⦇ *E′* ⦈ ⟧ ⟹
        ⦇ *P and P′* ⦈ *x* ⦇ λ*r*. (*Q r*) *and* (*Q′ r*) ⦈,⦇λ*r*. (*E r*) *and* (*E′ r*) ⦈
  by (*auto simp add*: *validE-def runs-to-partial-def-old split*: *xval-splits*)

lemma *L1-skip-wp*: ⦇ *P* () ⦈ *L1-skip* ⦇ *P* ⦈, ⦇ *Q* ⦈
  apply (*clarsimp simp*: *L1-skip-def validE-def*)
  done

lemma *L1-modify-wp*: ⦇ λ*s*. *P* () (*f s*) ⦈ *L1-modify f* ⦇ *P* ⦈, ⦇ *Q* ⦈
  apply (*clarsimp simp*: *L1-modify-def validE-def*)
  apply (*runs-to-vcg*)
  done

lemma *L1-spec-wp*: ⦇ λ*s*. ∀ *t*. (*s*, *t*) ∈ *f* ⟶ *P* () *t* ⦈ *L1-spec f* ⦇ *P* ⦈, ⦇ *Q* ⦈
  apply (*clarsimp simp add*: *L1-spec-def validE-def*)
  apply (*runs-to-vcg*)
  apply *auto*
  done

lemma *L1-assume-wp*: ⦇ λ*s*. ∀ *t*. ((), *t*) ∈ *f s* ⟶ *P* () *t* ⦈ *L1-assume f* ⦇ *P* ⦈, ⦇ *Q* ⦈
  apply (*clarsimp simp add*: *L1-assume-def validE-def*)
  apply (*runs-to-vcg*)
  apply *auto*
  done

lemma *L1-init-wp*: ⦇ λ*s*. ∀ *x*. *P* () (*f* (λ-. *x*) *s*) ⦈ *L1-init f* ⦇ *P* ⦈, ⦇ *Q* ⦈
  apply (*clarsimp simp add*: *L1-init-def validE-def*)
  apply (*runs-to-vcg*)
  apply *auto*
  done

lemma *L1-skip-lp*: ⟦ ⋀*s*. *P s* ⟹ *Q* () *s* ⟧ ⟹ ⦇*P*⦈ *L1-skip* ⦇*Q*⦈, ⦇*E*⦈

apply (*clarsimp simp*: *L1-skip-def*)
apply (*clarsimp simp*: *validE-def*)
done

lemma *L1-skip-lp-same-pre-post*: $\{\!|P|\!\}$ *L1-skip* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
by (*rule L1-skip-lp*)

lemma *L1-guard-lp*: $[\![\bigwedge s.\ P\ s \Longrightarrow Q\ ()\ s\ ]\!] \Longrightarrow \{\!|P|\!\}$ *L1-guard e* $\{\!|Q|\!\}$, $\{\!|E|\!\}$
apply (*clarsimp simp*: *L1-guard-def guard-def*)
apply (*clarsimp simp*: *validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-guard-lp-same-pre-post*: $\{\!|P|\!\}$ *L1-guard e* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
by (*rule L1-guard-lp*)

lemma *L1-guarded-lp-same-pre-post*: $\{\!|P|\!\}$ *c* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
$\Longrightarrow \{\!|P|\!\}$ *L1-guarded g c* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
unfolding *L1-defs L1-guarded-def validE-def*
by (*auto simp add*: *runs-to-partial-def-old succeeds-bind reaches-bind*)

lemma *L1-guarded-lp-gets*: $(\bigwedge p.\ \{\!|P|\!\}\ (c\ p)\ \{\!|\lambda\text{-}.\ P|\!\}, \{\!|\lambda\text{-}.\ P|\!\})$
$\Longrightarrow \{\!|P|\!\}$ *L1-guarded g* (*gets dest* $\gg\!=$ $(\lambda p.\ c\ p)$) $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
unfolding *L1-defs L1-guarded-def validE-def*
by (*auto simp add*: *runs-to-partial-def-old succeeds-bind reaches-bind*)

lemma *L1-fail-lp*: $\{\!|P|\!\}$ *L1-fail* $\{\!|Q|\!\}$, $\{\!|E|\!\}$
apply (*clarsimp simp*: *L1-fail-def validE-def*)
done

lemma *L1-fail-lp-same-pre-post*: $\{\!|P|\!\}$ *L1-fail* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
by (*rule L1-fail-lp*)

lemma *L1-throw-lp*: $[\![\bigwedge s.\ P\ s \Longrightarrow E\ ()\ s\ ]\!] \Longrightarrow \{\!|P|\!\}$ *L1-throw* $\{\!|Q|\!\}$, $\{\!|E|\!\}$
apply (*clarsimp simp*: *L1-throw-def validE-def*)
done

lemma *L1-throw-lp-same-pre-post*: $\{\!|P|\!\}$ *L1-throw* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
by (*rule L1-throw-lp*)

lemma *L1-spec-lp*: $[\![\bigwedge s\ r.\ [\![(s,\ r) \in e;\ P\ s\ ]\!] \Longrightarrow Q\ ()\ r\ ]\!] \Longrightarrow \{\!|P|\!\}$ *L1-spec e*
$\{\!|Q|\!\}$, $\{\!|E|\!\}$

apply (*clarsimp simp*: *L1-spec-def validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-spec-lp-same-pre-post*: ⟦ ⋀*s r*. ⟦ (*s*, *r*) ∈ *e*; *P s* ⟧ ⟹ *P r* ⟧
⟹ {|*P*|} *L1-spec e* {|λ-. *P*|}, {|λ-. *P*|}
by (*rule L1-spec-lp*)

lemma *L1-modify-lp*: ⟦ ⋀*s*. *P s* ⟹ *Q* () (*f s*) ⟧ ⟹ {|*P*|} *L1-modify f* {|*Q*|}, {|*E*|}
apply (*clarsimp simp*: *L1-modify-def validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-modify-lp-same-pre-post*: ⟦ ⋀*s*. *P s* ⟹ *P* (*f s*) ⟧ ⟹ {|*P*|} *L1-modify f*
{|λ-. *P*|}, {|λ-. *P*|}
by (*rule L1-modify-lp*)

lemma *L1-call-lp*:
⟦ ⋀*s r*. *P s* ⟹ *Q* () (*return-xf r* (*scope-teardown s r*));
⋀*s r*. *P s* ⟹ *E* () (*result-exn* (*scope-teardown s r*) *r*)⟧ ⟹
{|*P*|} *L1-call scope-setup dest-fn scope-teardown result-exn return-xf* {|*Q*|}, {|*E*|}
apply (*clarsimp simp*: *L1-defs L1-call-def validE-def*)
apply (*runs-to-vcg*)
apply (*auto simp add*: *runs-to-partial-def-old reaches-bind*)
done

lemma *L1-call-lp-same-pre-post*:
⟦ ⋀*s r*. *P s* ⟹ *P* (*return-xf r* (*scope-teardown s r*));
⋀*s r*. *P s* ⟹ *P* (*result-exn* (*scope-teardown s r*) *r*)⟧ ⟹
{|*P*|} *L1-call scope-setup dest-fn scope-teardown result-exn return-xf* {|λ-. *P*|}, {|λ-.
P|}
by (*rule L1-call-lp*)

lemma *L1-seq-lp*: ⟦
{|*P1*|} *A* {|*Q1*|},{|*E1*|};
{|*P2*|} *B* {|*Q2*|},{|*E2*|};
⋀*s*. *P s* ⟹ *P1 s*;
⋀*s*. *Q1* () *s* ⟹ *P2 s*;
⋀*s*. *Q2* () *s* ⟹ *Q* () *s*;
⋀*s*. *E1* () *s* ⟹ *E* () *s*;
⋀*s*. *E2* () *s* ⟹ *E* () *s*
⟧ ⟹ {|*P*|} *L1-seq A B* {|*Q*|}, {|*E*|}
apply (*clarsimp simp*: *L1-seq-def validE-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old reaches-bind split*: *xval-splits*)
done

lemma *L1-seq-lp-same-pre-post*: ⟦
  ⦃*P*⦄ *A* ⦃λ-. *P*⦄,⦃λ-. *P*⦄;
  ⦃*P*⦄ *B* ⦃λ-. *P*⦄,⦃λ-. *P*⦄
  ⟧ ⟹ ⦃*P*⦄ *L1-seq A B* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
by (*rule L1-seq-lp*)

lemma *L1-condition-lp*:
⟦ ⦃*P1*⦄ *A* ⦃*Q1*⦄,⦃*E1*⦄;
  ⦃*P2*⦄ *B* ⦃*Q2*⦄,⦃*E2*⦄;
  ⋀*s*. *P s* ⟹ *P1 s*;
  ⋀*s*. *P s* ⟹ *P2 s*;
  ⋀*s*. *Q1* () *s* ⟹ *Q* () *s*;
  ⋀*s*. *Q2* () *s* ⟹ *Q* () *s*;
  ⋀*s*. *E1* () *s* ⟹ *E* () *s*;
  ⋀*s*. *E2* () *s* ⟹ *E* () *s* ⟧ ⟹
⦃*P*⦄ *L1-condition c A B* ⦃*Q*⦄, ⦃*E*⦄
apply (*clarsimp simp*: *L1-condition-def validE-def*)
apply (*runs-to-vcg*)
 apply (*fastforce simp add*: *runs-to-partial-def-old split*: *xval-splits*)+
done

lemma *L1-condition-lp-same-pre-post*:
⟦⦃*P*⦄ *A* ⦃λ-. *P*⦄,⦃λ-. *P*⦄;
 ⦃*P*⦄ *B* ⦃λ-. *P*⦄,⦃λ-. *P*⦄
⟧ ⟹
⦃*P*⦄ *L1-condition c A B* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
by (*rule L1-condition-lp*)

lemma *L1-catch-lp*:
⟦ ⦃*P1*⦄ *A* ⦃*Q1*⦄,⦃*E1*⦄;
  ⦃*P2*⦄ *B* ⦃*Q2*⦄,⦃*E2*⦄;
  ⋀*s*. *P s* ⟹ *P1 s*;
  ⋀*s*. *E1* () *s* ⟹ *P2 s*;
  ⋀*s*. *Q1* () *s* ⟹ *Q* () *s*;
  ⋀*s*. *Q2* () *s* ⟹ *Q* () *s*;
  ⋀*s*. *E2* () *s* ⟹ *E* () *s* ⟧ ⟹
⦃*P*⦄ *L1-catch A B* ⦃*Q*⦄, ⦃*E*⦄
apply (*clarsimp simp*: *L1-catch-def validE-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old split*: *xval-splits*)
done

lemma *L1-catch-lp-same-pre-post*:
⟦⦃*P*⦄ *A* ⦃λ-. *P*⦄,⦃λ-. *P*⦄;
 ⦃*P*⦄ *B* ⦃λ-. *P*⦄,⦃λ-. *P*⦄
⟧ ⟹
⦃*P*⦄ *L1-catch A B* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄

by (*rule L1-catch-lp*)

lemma *L1-init-lp*: ⟦ ⋀*s*. *P s* ⟹ ∀ *x*. *Q* () (*f* (λ-. *x*) *s*) ⟧ ⟹ {‖*P*‖} *L1-init f* {‖*Q*‖}, {‖*E*‖}
apply (*clarsimp simp add*: *L1-init-def validE-def*)
apply (*runs-to-vcg*)
apply *auto*
done

lemma *L1-init-lp-same-pre-post*: ⟦ ⋀*s*. *P s* ⟹ ∀ *x*. *P* (*f* (λ-. *x*) *s*) ⟧ ⟹ {‖*P*‖} *L1-init f* {‖λ-. *P*‖}, {‖λ-. *P*‖}
by (*rule L1-init-lp*)

lemma *validE-weaken*:
⟦ {‖*P'*‖} *A* {‖*Q'*‖},{‖*E'*‖}; ⋀*s*. *P s* ⟹ *P' s*; ⋀*r s*. *Q' r s* ⟹ *Q r s*; ⋀*r s*. *E' r s* ⟹ *E r s* ⟧ ⟹ {‖*P*‖} *A* {‖*Q*‖},{‖*E*‖}
apply (*fastforce simp add*: *validE-def runs-to-partial-def-old split*: *xval-splits*)
done

lemma *L1-while-lp*:
assumes *body-lp*: {‖ *P'* ‖} *B* {‖ *Q'* ‖}, {‖ *E'* ‖}
and *p-impl*: ⋀*s*. *P s* ⟹ *P' s*
and *q-impl*: ⋀*s*. *Q'* () *s* ⟹ *Q* () *s*
and *e-impl*: ⋀*s*. *E'* () *s* ⟹ *E* () *s*
and *inv*: ⋀*s*. *Q'* () *s* ⟹ *P' s*
and *inv'*: ⋀*s*. *P' s* ⟹ *Q'* () *s*
shows {‖ *P* ‖} *L1-while c B* {‖ *Q* ‖},{‖ *E* ‖}
apply (*rule validE-weaken* [**where** *P'*=*P'* **and** *Q'*=*Q'* **and** *E'*=*E'*])
apply (*clarsimp simp*: *L1-while-def validE-def*)

apply (*rule runs-to-partial-whileLoop-exn* [**where** *I*=λ*r s*. (*case r of Exn e* ⇒ *E'* () *s* | - ⇒ *P' s*)])
apply *simp*
apply (*simp add*: *inv'*)
apply (*simp*)
apply *simp*
subgoal using *body-lp*
by (*simp add*: *inv validE-def runs-to-partial-def-old split*: *xval-splits*)
apply (*simp add*: *p-impl*)
apply (*simp add*: *q-impl*)
apply (*simp add*: *e-impl*)
done

lemma *L1-while-lp-same-pre-post*:
assumes *body-lp*: {‖ *P* ‖} *B* {‖ λ-. *P*‖}, {‖ λ-. *P* ‖}
shows {‖ *P* ‖} *L1-while c B* {‖λ-. *P* ‖},{‖ λ-. *P* ‖}
by (*rule L1-while-lp* [*OF body-lp*])

lemma *on-exit-lp-same-pre-post*:
assumes *cleanup*: $\bigwedge s\ t.\ (s,t) \in cleanup \Longrightarrow\ P\ t = P\ s$
assumes *c*: $\{\!|P|\!\}\ c\ \{\!|\lambda\text{-}.\ P|\!\}, \{\!|\lambda\text{-}.\ P|\!\}$
shows $\{\!|P|\!\}$ *on-exit c cleanup* $\{\!|\lambda\text{-}.\ P|\!\}, \{\!|\lambda\text{-}.\ P|\!\}$
apply (*clarsimp simp add*: *validE-def on-exit'-def on-exit-def*)
apply (*runs-to-vcg*)
using *cleanup c*
apply (*fastforce simp add*: *validE-def runs-to-partial-def-old succeeds-bind reaches-bind default-option-def Exn-def split*: *xval-splits*)
done

lemma *seqE*:
assumes *f-valid*: $\{\!|A|\!\}\ f\ \{\!|B|\!\},\{\!|E|\!\}$
assumes *g-valid*: $\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\},\{\!|E|\!\}$
shows $\{\!|A|\!\}$ *do* { $x \leftarrow f$; $g\ x$ } $\{\!|C|\!\},\{\!|E|\!\}$
using *assms*
apply (*clarsimp simp add*: *validE-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old split*: *xval-splits*)
done

named-theorems *with-fresh-stack-ptr-lp-same-pre-post*
context *stack-heap-state*
begin
lemma *with-fresh-stack-ptr-lp-same-pre-post*[*with-fresh-stack-ptr-lp-same-pre-post*]:

assumes *c*: $\bigwedge p.\ \{\!|P|\!\}\ (c\ p)\ \{\!|\lambda\text{-}.\ P|\!\}, \{\!|\lambda\text{-}.\ P|\!\}$
assumes *htd-indep*: $\bigwedge s\ g.\ P\ (htd\text{-}upd\ g\ s) = P\ s$
assumes *hmem-indep*: $\bigwedge s\ f.\ P\ (hmem\text{-}upd\ f\ s) = P\ s$
shows $\{\!|P|\!\}$ *with-fresh-stack-ptr n g c* $\{\!|\lambda\text{-}.\ P|\!\}, \{\!|\lambda\text{-}.\ P|\!\}$
apply (*clarsimp simp add*: *with-fresh-stack-ptr-def*)
apply (*rule seqE* [**where** $B{=}\lambda\text{-}.\ P$])
subgoal
apply (*clarsimp simp add*: *validE-def*)
apply (*runs-to-vcg*)
by (*clarsimp simp add*: *htd-indep hmem-indep*)
subgoal
apply (*rule on-exit-lp-same-pre-post* [*OF - c*])
apply (*auto simp add*: *htd-indep hmem-indep*)
done
done
end

lemma *validE-weaken-dependent*:
assumes *dep-spec*: $\bigwedge s.\ P\ s \Longrightarrow \{\!|P'\ s|\!\}\ A\ \{\!|Q'\ s|\!\}, \{\!|E'\ s|\!\}$
assumes *weaken-pre*: $\bigwedge s.\ P\ s \Longrightarrow P'\ s\ s$
assumes *weaken-norm*: $(\bigwedge s\ r\ t.\ P'\ s\ s \Longrightarrow Q'\ s\ r\ t \Longrightarrow Q\ r\ t)$
assumes *weaken-exn*: $(\bigwedge s\ r\ t.\ P'\ s\ s \Longrightarrow E'\ s\ r\ t \Longrightarrow E\ r\ t)$

shows $\{\!|P|\!\}\ A\ \{\!|Q|\!\}, \{\!|E|\!\}$
using *assms*
apply (*fastforce simp add*: *validE-def runs-to-partial-def-old split*: *xval-splits*)
done

lemma *validE-weaken-dependent-same*:
assumes *dep-spec*: $\bigwedge s.\ P\ s \Longrightarrow \{\!|P'\ s|\!\}\ A\ \{\!|\lambda\text{-}.\ P'\ s|\!\}, \{\!|\lambda\text{-}.\ P'\ s|\!\}$
assumes *weaken-post*: $(\bigwedge s\ t.\ P'\ s\ t \Longrightarrow P\ t)$
assumes *weaken-pre*: $\bigwedge s.\ P\ s \Longrightarrow P'\ s\ s$
shows $\{\!|P|\!\}\ A\ \{\!|\lambda\text{-}.\ P|\!\}, \{\!|\lambda\text{-}.\ P|\!\}$
using *dep-spec weaken-pre weaken-post weaken-post*
by (*rule validE-weaken-dependent*)

end

Chapter 19

L2 phase: local variable abstraction with lambdas

theory *L2Defs*
imports *CorresXF L1Defs L1Peephole L1Valid*
begin

type-synonym $('s, 'a, 'e)$ *L2-monad* $= ('e, 'a, 's)$ *exn-monad*

definition *L2-unknown* $(ns :: nat\ list) \equiv$ *select UNIV* $:: ('s, 'a, 'e)$ *L2-monad*
definition *L2-seq* $(A :: ('s, 'a, 'e)$ *L2-monad*$) (B :: 'a \Rightarrow ('s, 'b, 'e)$ *L2-monad*$) \equiv (A >>= B) :: ('s, 'b, 'e)$ *L2-monad*
definition *L2-modify* $m \equiv$*modify* $m :: ('s,$ *unit*$, 'e)$ *L2-monad*
definition *L2-gets* f $(names :: nat\ list) \equiv$ *gets* $f :: ('s, 'a, 'e)$ *L2-monad*
definition *L2-condition* c $(A :: ('s, 'a, 'e)$ *L2-monad*$) (B :: ('s, 'a, 'e)$ *L2-monad*$)$ $\equiv$ *condition* c A B
definition *L2-catch* $(A :: ('s, 'a, 'e)$ *L2-monad*$) (B :: 'e \Rightarrow ('s, 'a, 'ee)$ *L2-monad*$)$ $\equiv (A <catch> B)$
definition *L2-try* $(A :: ('s, 'a, 'e + 'a)$ *L2-monad*$) \equiv$ *try* A
definition *L2-while* c $(A :: 'a \Rightarrow ('s, 'a, 'e)$ *L2-monad*$)$ i $(ns :: nat\ list) \equiv$ *whileLoop* c A $i :: ('s, 'a, 'e)$ *L2-monad*
definition *L2-throw* x $(ns :: nat\ list) \equiv$ *throw* $x :: ('s, 'a, 'e)$ *L2-monad*
definition *L2-spec* $r \equiv$ (*state-select* $r >>= (\lambda$-. *select UNIV*$)) :: ('s, 'a, 'e)$ *L2-monad*
definition *L2-assume* $r \equiv$ (*assume-result-and-state* r) $:: ('s, 'a, 'e)$ *L2-monad*
definition *L2-guard* $c \equiv$ (*guard* c) $:: ('s,$ *unit*$, 'e)$ *L2-monad*
definition *L2-guarded* P $c \equiv$ *L2-seq* (*L2-guard* P) $(\lambda$-. c) — used to guard a function-pointer call
definition *L2-fail* $\equiv$ *fail* $:: ('s, 'a, 'e)$ *L2-monad*
definition *L2-call* x *emb* $(ns :: nat\ list) \equiv$ *map-value* (*map-exn emb*) $x :: ('s, 'a, 'e)$ *L2-monad*

abbreviation *L2-skip* $\equiv$ *L2-gets* $(\lambda$-. $())$ []
definition *L2-VARS* f $(names::nat\ list) \equiv f$ — Auxilliary construct to preserve names, like in *L2-unknown*, *L2-gets*, ...

definition *gets-theE* :: (*'s* ⇒ *'a option*) ⇒ (*'e*, *'a*, *'s*) *exn-monad* **where**
gets-theE ≡ λ*x*. *gets-the x* — Lifting into exception monad

definition *L2-folded-gets f names* ≡ *L2-gets f names* :: (*'s*, *'a*, *'e*) *L2-monad*
definition *L2-voidcall x emb ns* ≡ *L2-seq* (*L2-call x emb ns*) (λ*ret*. *L2-skip*) :: (*'s*, *unit*, *'e*) *L2-monad*
definition *L2-modifycall x m emb ns* ≡ *L2-seq* (*L2-call x emb ns*) (λ*ret*. *L2-modify* (*m ret*)) :: (*'s*, *unit*, *'e*) *L2-monad*
definition *L2-returncall x f emb ns* ≡ *L2-seq* (*L2-call x emb ns*) (λ*ret*. *L2-folded-gets* (*f ret*) []) :: (*'s*, *'a*, *'e*) *L2-monad*

definition *L2-seq-guard P A* ≡ *L2-seq* (*L2-guard P*) *A*
definition *L2-seq-gets c n A* ≡ *L2-seq* (*L2-gets* (λ-. *c*) *n*) *A*
definition *L2-seq-unknown ns A* ≡ *L2-seq* (*L2-unknown ns*) *A*
definition *L2-seq-condition c L R X* ≡ *L2-seq* (*L2-condition c L R*) *X*

definition *SANITIZE-NAMES prj ns ns'* = *True*

lemma *sanitize-namesI*: *SANITIZE-NAMES prj ns ns'*
by (*simp add*: *SANITIZE-NAMES-def*)

definition *DYN-CALL* :: *prop* ⇒ *prop* **where** *PROP DYN-CALL* (*PROP P*) ≡ *PROP P*

lemma *DYN-CALL-I*: *PROP P* ⟹ *PROP DYN-CALL* (*PROP P*)
by (*simp add*: *DYN-CALL-def*)

lemma *DYN-CALL-D*: *PROP DYN-CALL* (*PROP P*) ⟹ *PROP P*
by (*simp add*: *DYN-CALL-def*)

lemma *runs-to-partial-top*[*corres-top*]: ⊤ ⋅ *s* ?{⦇ *Q* ⦈}
by (*simp add*: *runs-to-partial-def-old*)

lemma *admissible-runs-to-partial*[*corres-admissible*]: *ccpo.admissible Inf* (≥) (λ*f* . *f* ⋅ *s* ?{⦇*Q*⦈})
unfolding *runs-to-partial-def-old*
apply (*rule ccpo.admissibleI*)
apply (*clarsimp simp add*: *ccpo.admissible-def chain-def*
succeeds-def reaches-def split: *prod.splits xval-splits*)
subgoal
apply *transfer*
apply (*clarsimp simp add*: *Inf-post-state-def top-post-state-def image-def vimage-def*
split: *if-split-asm*)
by (*metis outcomes.simps*(*2*) *post-state.distinct*(*1*))

done

lemma *reaches-gets-theE* [*simp*]:
reaches (*gets-theE M*) *s rv s'* ⟷ (∃ *v'*. *rv* = *Result v'* ∧ *s'* = *s* ∧ *M s* = *Some v'*)
by (*auto simp add*: *gets-theE-def*)

lemma *seucceeds-gets-theE* [*simp*]:
succeeds (*gets-theE M*) *s* = (∃ *v*. *M s* = *Some v*)
apply (*auto simp add*: *gets-theE-def*)
done

lemma *L2-folded-gets-bind*:
L2-seq (*L2-folded-gets* (λ-. *x*) *ns*) *f* = *f x*
unfolding *L2-seq-def L2-folded-gets-def L2-gets-def*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-call-throw-names*: *L2-call x emb ns* = (*x* <*catch*> (λ*r*. *L2-throw* (*emb r*) *ns*))
unfolding *L2-call-def L2-throw-def*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*auto simp add*: *runs-to-def-old map-exn-def split*: *xval-splits*)
done

lemmas *L2-remove-scaffolding-1* =
L2-folded-gets-bind [*THEN eq-reflection*]
L2-returncall-def
L2-modifycall-def
L2-voidcall-def

lemmas *L2-remove-scaffolding-2* =
L2-remove-scaffolding-1
L2-folded-gets-def

abbreviation (*input*) *L2-guarded-while G C B i n* ≡ *L2-seq* (*L2-guard* (*G i*))
(λ-. *L2-while C* (λ*i*. *L2-seq* (*B i*) (λ*r*. *L2-seq* (*L2-guard* (*G r*)) (λ-. *L2-gets* (λ-. *r*) *n*))) *i n*)

lemmas *L2-defs* = *L2-unknown-def L2-seq-def*
L2-modify-def L2-gets-def L2-condition-def L2-catch-def L2-while-def
L2-throw-def L2-spec-def L2-assume-def L2-guard-def L2-fail-def L2-folded-gets-def
L2-voidcall-def L2-modifycall-def L2-returncall-def

L2-try-def

lemma *L2-defs'*:
L2-unknown n ≡ unknown
L2-seq A' B' ≡ A' >>= B'
L2-modify m ≡ modify m
L2-gets f n ≡ gets f
L2-condition c L R ≡ condition c L R
L2-catch A B ≡ (A <catch> B)
L2-try A ≡ try A
L2-while c' B'' i n ≡ whileLoop c' B'' i
L2-throw x n ≡ throw x
L2-spec r ≡ (state-select r >>= (λ-. select UNIV))
L2-assume r' ≡ (assume-result-and-state r')
L2-guard c ≡ guard c
L2-fail ≡ fail
by (*auto simp*: *L2-defs unknown-def*)

definition
L2corres :: ('*s* ⇒ '*t*) ⇒ ('*s* ⇒ '*r*) ⇒ ('*s* ⇒ '*e*) ⇒ ('*s* ⇒ *bool*)
⇒ ('*e*, '*r*, '*t*) *exn-monad* ⇒ (*unit*, *unit*, '*s*) *exn-monad* ⇒ *bool*
where
L2corres st ret-xf ex-xf P A C
≡ *corresXF st* (λ-. *ret-xf*) (λ-. *ex-xf*) *P A C*

lemma *admissible-nondet-ord-L2corres* [*corres-admissible*]:
ccpo.admissible Inf (≥) (λ*A*. *L2corres st ret-xf ex-xf P A C*)
unfolding *L2corres-def*
apply (*rule admissible-nondet-ord-corresXF*)
done

lemma *L2corres-top* [*corres-top*]: *L2corres st ret-xf ex-xf P* ⊤ *C*
by (*auto simp add*: *L2corres-def corresXF-def*)

definition
L2-call-L1 arg-xf gs ret-xf l1body
= *do* {
s ← *get-state*;
t ← *select* {*t*. *gs t* = *s* ∧ *arg-xf t*};
run-bind l1body t (λ*r' t'*.
(*case r' of*
Exception - ⇒ *fail*
| *Result* - ⇒ *do* {*set-state* (*gs t'*); *return* (*ret-xf t'*)}))
}

lemma *L2corres-L2-call-L1*:
L2corres gs ret-xf ex-xf arg-xf
(L2-call-L1 arg-xf gs ret-xf f) f
apply (*clarsimp simp*: *L2corres-def corresXF-refines-iff L2-call-L1-def L1-seq-def L1-guard-def*
split: *xval-splits*)
apply (*simp add*: *refines-def-old*)
apply (*intro conjI impI*)
subgoal
by (*auto simp add*: *succeeds-bind succeeds-run-bind succeeds-exec-concrete-iff succeeds-catch*)
subgoal
by (*auto simp add*: *succeeds-bind succeeds-run-bind reaches-run-bind*
succeeds-exec-concrete-iff succeeds-catch reaches-bind default-option-def Exn-def
rel-XF-def rel-xval.simps Exn-def default-option-def
split: *exception-or-result-splits*)

(*smt* (*verit*, *ccfv-threshold*) *Exn-def Exn-neq-Result exception-or-result-cases*
imageI mem-Collect-eq old.unit.exhaust option.exhaust-sel)
done

lemma *L2corres-L2-call-simpl*:
l1-f ≡ *simpl-f* ⟹
L2corres gs ret-xf ex-xf arg-xf
(L2-call-L1 arg-xf gs ret-xf simpl-f) l1-f
by (*simp add*: *L2corres-L2-call-L1*)

lemma *L2corres-modify-global*:
⟦ ⋀*s*. *P s* ⟹ *M* (*st s*) = *st* (*M′ s*) ⟧ ⟹
L2corres st ret ex P (*L2-modify M*) (*L1-modify M′*)
unfolding *L2-defs L1-defs*
by (*auto simp add*: *L2corres-def corresXF-modify-global*)

lemma *L2corres-modify-unknown*:
⟦ ⋀*s*. *P s* ⟹ *st s* = *st* (*M s*) ⟧ ⟹
L2corres st ret ex P (*L2-unknown ns*) (*L1-modify M*)
apply (*clarsimp simp*: *L2corres-def L2-defs L1-defs*)
apply (*auto intro*: *corresXF-select-modify*)
done

lemma *L2corres-spec-unknown*:
⟦ ⋀*s a*. *st s* = *st* (*M* (*a*::(*′a* ⇒ (*′a*::{*type*}))) *s*) ⟧ ⟹
L2corres st ret ex P (*L2-unknown ns*) (*L1-init M*)
apply (*auto simp add*: *L2corres-def corresXF-def L1-defs L2-defs*

reaches-bind reaches-succeeds succeeds-bind)
done

lemma *L2corres-modify-gets*:
⟦ ⋀*s*. *P s* ⟹ *st s* = *st* (*M s*); ⋀*s*. *P s* ⟹ *ret* (*M s*) = *f* (*st s*) ⟧ ⟹
L2corres st ret ex P (*L2-gets* (λ*s*. *f s*) *n*) (*L1-modify M*)
by (*simp add*: *L2corres-def L2-defs L1-defs corresXF-modify-gets*)

lemma *L2corres-gets-skip*:
L2corres st ret ex P L2-skip L1-skip
by (*simp add*: *L2corres-def L2-defs L1-defs corresXF-def*)

lemma *L2corres-guard*:
⟦ ⋀*s*. *P s* ⟹ *G*′ *s* = *G* (*st s*) ⟧ ⟹
L2corres st return-xf exception-xf P (*L2-guard G*) (*L1-guard G*′)
by (*simp add*: *L2corres-def L2-defs L1-defs corresXF-guard*)

lemma *L2corres-fail*:
L2corres st return-xf exception-xf P L2-fail X
by (*clarsimp simp add*: *L2corres-def L1-defs L2-defs corresXF-def*)

lemma *L2corres-spec*:
⟦ ⋀*s s*′. ((*s*, *s*′) ∈ *A*′) = ((*st s*, *st s*′) ∈ *A*); *surj st* ⟧
⟹ *L2corres st return-xf exception-xf P* (*L2-spec A*) (*L1-spec A*′)
apply (*clarsimp simp*: *L2corres-def L2-defs L1-spec-def corresXF-def succeeds-bind reaches-bind*)
by (*metis surjD*)

lemma *L2corres-assume*:
⟦ ⋀*s s*′. ((((), *s*′) ∈ *A*′ *s*) = ((((), *st s*′) ∈ *A* (*st s*)) ⟧
⟹ *L2corres st return-xf exception-xf P* (*L2-assume A*) (*L1-assume A*′)
by (*clarsimp simp*: *L2corres-def L2-defs L1-assume-def corresXF-def*)

lemma *L2corres-seq*:
⟦ *L2corres st return-xf exception-xf P A A*′;
⋀*x*. *L2corres st return-xf*′ *exception-xf* (*P*′ *x*) (*B x*) *B*′;
{|*R*|} *A*′ {|λ- *s*. *P*′ (*return-xf s*) *s*|}, {|λ- -. *True*|};
⋀*s*. *R s* ⟹ *P s* ⟧ ⟹
L2corres st return-xf′ *exception-xf R* (*L2-seq A B*) (*L1-seq A*′ *B*′)
apply (*unfold L2corres-def L2-seq-def L1-seq-def validE-def*)
apply (*rule corresXF-join*)
apply *assumption*
apply *assumption*
apply *simp*
apply *assumption*
done

lemma *L2corres-guard-imp*:
⟦ *L2corres st ret-state ex-state Q M M'*; *pred-imp P Q* ⟧ ⟹
L2corres st ret-state ex-state P M M'
by (*simp add*: *L2corres-def pred-imp-def corresXF-guard-imp*)

lemma *L2corres-guarded''*:
assumes *bdy*: ⋀*s' s*. *g' s'* ⟹ *s* = *st s'* ⟹ *P s'* ⟹ *L2corres st ret ex P* (*c* (*dest s*)) (*c'* (*dest' s'*))
assumes *g-g'*: ⋀*s'*. *P s'* ⟹ *g' s'* = *g* (*st s'*)
assumes *dest*: ⋀*s' s*. *g' s'* ⟹ *s* = *st s'* ⟹ *P s'* ⟹ *dest' s'* = *dest* (*st s'*)
shows *L2corres st ret ex P* (*L2-guarded g* (*L2-seq* (*L2-gets dest* []) *c*)) (*L1-guarded g'* (*gets dest'* ≫= *c'*))
using *assms*
by (*auto simp add*: *L2corres-def L2-defs L1-defs L2-guarded-def L1-guarded-def corresXF-def*
succeeds-bind reaches-bind split: *xval-splits*)

lemma *L2corres-catch*:
⟦ *L2corres st V E P L L'*;
⋀*x*. *L2corres st V E'* (*P' x*) (*R x*) *R'*;
{‖*Q*‖} *L'* {‖λ- -. *True*‖}, {‖λ- *s*. *P'* (*E s*) *s*‖}; ⋀*s*. *Q s* ⟹ *P s* ⟧ ⟹
L2corres st V E' Q (*L2-catch L R*) (*L1-catch L' R'*)
apply (*clarsimp simp*: *L2corres-def L2-catch-def L1-catch-def validE-def*)
apply (*rule corresXF-except*)
apply (*assumption*)
apply (*assumption*)
apply *auto*
done

lemma *L2corres-throw*:
⟦ ⋀*s*. *P s* ⟹ *E s* = *A* ⟧ ⟹ *L2corres st V E P* (*L2-throw A n*) (*L1-throw*)
by (*clarsimp simp*: *L2corres-def L2-throw-def L1-throw-def corresXF-throw*)

lemma *L2corres-conseq*:
assumes *corres*: *L2corres st return-xf exception-xf P C C'*
assumes *conseq*: ⋀*s*. *Q s* ⟹ *P s*
shows *L2corres st return-xf exception-xf Q C C'*
apply (*rule L2corres-guard-imp* [*OF corres*])
using *conseq* **by** (*simp add*: *pred-imp-def*)

lemma *L2corres-cond*:
⟦ *L2corres st return-xf exception-xf P A A'*;
L2corres st return-xf exception-xf P' B B';
⋀*s*. *R s* ⟹ *P s*;
⋀*s*. *R s* ⟹ *P' s*;
⋀*s*. *R s* ⟹ *c' s* = *c* (*st s*) ⟧ ⟹
L2corres st return-xf exception-xf R (*L2-condition c A B*) (*L1-condition c' A'*

B')
apply (*unfold L2corres-def L2-condition-def L1-condition-def*)
apply (*rule corresXF-cond*)
apply (*auto intro*: *corresXF-guard-imp*)
done

lemma *L2corres-inject-return*:
⟦ *L2corres st V E P L R*; {|P'|} *R* {|λ- *s*. *W* (*V s*) = *V' s*|}, {| λ- -. *True* |}; ⋀*s*. *P' s* ⟹ *P s*⟧ ⟹
L2corres st V' E P' (*L2-seq L* (λ*x*. *L2-gets* (λ-. *W x*) *n*)) *R*
apply (*clarsimp simp*: *L2corres-def validE-def*)
apply (*drule corresXF-guard-imp* [**where** *P*=*P'*], *simp*)
apply (*unfold L2-seq-def L2-gets-def*)
apply (*rule corresXF-guard-imp*)
apply (*erule corresXF-append-gets-abs*)
apply *simp*
apply *simp*
done

lemma *L2corres-inject-return'*:
⟦ *L2corres st V E P L R*;*Gets* = (λ*x*. *L2-gets* (λ-. *W x*) *n*);{|P'|} *R* {|λ- *s*. *W* (*V s*) = *V' s*|}, {| λ- -. *True*|}; ⋀*s*. *P' s* ⟹ *P s*
⟧ ⟹
L2corres st V' E P' (*L2-seq L Gets*) *R*
by (*auto intro*: *L2corres-inject-return*)

lemma *L2corres-skip*:
L2corres st return-xf exception-xf P L2-skip L1-skip
by (*rule L2corres-gets-skip*)

lemma *L2corres-while*:
assumes *body-corres*: ⋀*x*. *L2corres st ret ex* (*P' x*) (*A x*) *B*
and *inv-holds*: {|λ*s*. *P* (*ret s*) *s* |} *B* {|λ- *s*. *P* (*ret s*) *s* |}, {|λ- -. *True*|}
and *cond-match*: ⋀*s*. *P* (*ret s*) *s* ⟹ *c' s* = *c* (*ret s*) (*st s*)
and *pred-imply*: ⋀*s x*. *P x s* ⟹ *P' x s*
and *pred-extract*: ⋀*s*. *P x s* ⟹ *ret s* = *x*
and *pred-imply2*: ⋀*s*. *Q x s* ⟹ *P x s*
shows *L2corres st ret ex* (*Q x*) (*L2-while c A x n*) (*L1-while c' B*)
apply (*clarsimp simp*: *L2corres-def L2-while-def L1-while-def*)
apply (*rule corresXF-guard-imp*)
apply (*rule corresXF-while* [
where *P*=λ*r s*. *P* (*ret s*) *s* **and** *C'*=*c* **and** *C*=λ-. *c'* **and** *A*=*A* **and** *B*=λ-. *B*
and *ret*=λ*r s*. *ret s* **and** *ex*=λ*r s*. *ex s* **and** *st*=*st* **and** *y*=*x* **and** *x*=() **and** *P'*=λ*r s*. *Q x s*])
apply (*rule corresXF-guard-imp*)
apply (*rule body-corres* [*unfolded L2corres-def*])
apply (*clarsimp simp*: *pred-imply*)
apply (*clarsimp simp*: *cond-match*)

apply (*rule inv-holds* [*unfolded validE-def*, *rule-format*])
apply *assumption*
apply (*metis pred-extract pred-imply2*)
apply (*metis pred-extract pred-imply2*)
apply *simp*
done

lemma *L2corres-returncall*:
⟦ *L2corres st ret′ ex′ P′ Z dest-fn*;
⋀*s*. *P s* ⟹ *P′* (*scope-setup s*);
⋀*t s*. *st* (*return-xf t* (*scope-teardown s t*)) = *st t*;
⋀*t s*. *st* (*result-exn* (*scope-teardown s t*) *t*) = *st t*;
⋀*s*. *st* (*scope-setup s*) = *st s*;
⋀*t s*. *st* (*scope-teardown s t*) = *st t*;
⋀*t s*. *P s* ⟹ *ret* (*return-xf t* (*scope-teardown s t*)) = *F* (*ret′ t*) (*st t*) ;
⋀*t s*. *P s* ⟹ (*ex* (*result-exn* (*scope-teardown s t*) *t*)) = *emb* (*ex′ t*) ⟧ ⟹
L2corres st ret ex P (*L2-returncall Z F emb ns*)
(*L1-call scope-setup dest-fn scope-teardown result-exn return-xf*)
unfolding *L1-call-def L1-seq-def L1-throw-def L2-returncall-def L2-call-def L2corres-def L2-defs*
apply (*rule corresXF-I*)
subgoal
apply (*clarsimp simp add*: *corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: *xval-splits exception-or-result-splits*)
by (*smt* (*z3*) *Exn-def Result-neq-Exn map-exn-simps*(*2*) *the-Result-simp*)
subgoal
apply (*clarsimp simp add*: *corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: *xval-splits exception-or-result-splits*)
by (*metis* (*mono-tags*, *opaque-lifting*) *Exception-eq-Exception Exn-def Exn-neq-Result map-exn-simps*(*1*))
subgoal
apply (*auto simp add*: *corresXF-def succeeds-bind succeeds-catch reaches-catch*
reaches-map-value reaches-bind default-option-def Exn-def split: *xval-splits exception-or-result-splits*)
done
done

lemma *L2corres-dest-fn-simp*:
assumes *dest-fn*: ⋀*s*. *P s* ⟹ *dest-fn* = *dest-fn′*
assumes *corres*: *L2corres st ret ex P X*
(*L1-call scope-setup* (*measure-call dest-fn′*) *scope-teardown result-exn return-xf*)
shows *L2corres st ret ex P X*
(*L1-call scope-setup* (*measure-call dest-fn*) *scope-teardown result-exn return-xf*)

using *corres dest-fn* **unfolding** *L2corres-def corresXF-def*
by *blast*

lemma *L2corres-l2-propagate-fixed-cong*:
$(\bigwedge s.\ P\ s = P'\ s) \Longrightarrow (\bigwedge s.\ P'\ s \Longrightarrow A = A') \Longrightarrow$ *L2corres st ret ex P A C* = *L2corres st ret ex P' A' C*
unfolding *L2corres-def corresXF-def simp-implies-def*
by (*auto split*: *sum.splits prod.splits*)

lemma *L2corres-modifycall*:
⟦ *L2corres st ret' ex' P' Z dest-fn*;
$\bigwedge s.\ P\ s \Longrightarrow P'$ (*scope-setup s*);
$\bigwedge s\ r.\ P\ r \Longrightarrow F$ (*ret' s*) (*st s*) = *st* (*return-xf s* (*scope-teardown r s*));
$\bigwedge s.$ *st* (*scope-setup s*) = *st s*;
$\bigwedge t\ s.$ *st* (*scope-teardown s t*) = *st t*;
$\bigwedge t\ s.$ *st* (*result-exn* (*scope-teardown s t*) *t*) = *st t*;
$\bigwedge t\ s.\ P\ s \Longrightarrow$ *ex* (*result-exn* (*scope-teardown s t*) *t*) = *emb* (*ex' t*)⟧ ⟹
L2corres st ret ex P (*L2-modifycall Z F emb ns*)
(*L1-call scope-setup dest-fn scope-teardown result-exn return-xf*)
apply (*clarsimp simp*: *L1-call-def L1-seq-def L1-throw-def L2-call-def L2-defs L2corres-def*)
apply (*rule corresXF-I*)
subgoal
apply (*clarsimp simp add*: *corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: *xval-splits exception-or-result-splits*)
by (*smt* (*z3*) *Exn-def Result-neq-Exn map-exn-simps*(*2*) *the-Result-simp*)
subgoal
apply (*clarsimp simp add*: *corresXF-def succeeds-bind succeeds-catch reaches-catch*

reaches-map-value reaches-bind default-option-def Exn-def split: *xval-splits exception-or-result-splits*)
by (*metis* (*mono-tags*, *opaque-lifting*) *Exception-eq-Exception Exn-def Exn-neq-Result map-exn-simps*(*1*))
subgoal
apply (*auto simp add*: *corresXF-def succeeds-bind succeeds-catch reaches-catch*
reaches-map-value reaches-bind default-option-def Exn-def split: *xval-splits exception-or-result-splits*)
done
done

lemma *L2corres-voidcall*:
⟦ *L2corres st ret' ex' P' Z dest-fn*;
$\bigwedge s.\ P\ s \Longrightarrow P'$ (*scope-setup s*);
$\bigwedge s\ r.$ *st* (*return-xf s* (*scope-teardown r s*)) = *st s*;
$\bigwedge s.$ *st* (*scope-setup s*) = *st s*;

```
    ⋀t s. st (scope-teardown s t) = st t;
    ⋀t s. st (result-exn (scope-teardown s t) t) = st t;
    ⋀t s. P s ⟹ ex (result-exn (scope-teardown s t) t) = emb (ex' t)⟧ ⟹
    L2corres st ret ex P (L2-voidcall Z emb ns)
        (L1-call scope-setup dest-fn scope-teardown result-exn return-xf)
  apply (unfold L2-voidcall-def)
  apply (rule subst[where t = L2-skip and s = L2-modify (λs. s)])
  subgoal
    apply (simp add: L2-defs)
    apply (rule spec-monad-ext)
    apply simp
    done
  apply (fold L2-modifycall-def L2corres-def)
  apply (fastforce elim!: L2corres-modifycall)
  done


lemma L2corres-call:
  ⟦ L2corres st ret' ex' P' Z dest-fn;
    ⋀s. P s ⟹ P' (scope-setup s);
    ⋀s r. st (return-xf s (scope-teardown r s)) = st s;
    ⋀s r. ret (return-xf s (scope-teardown r s)) = ret' s;
    ⋀s. st (scope-setup s) = st s;
    ⋀t s. st (scope-teardown s t) = st t;
    ⋀t s. st (result-exn (scope-teardown s t) t) = st t;
    ⋀t s. P s ⟹ ex (result-exn (scope-teardown s t) t) = emb (ex' t) ⟧ ⟹
    L2corres st ret ex P (L2-call Z emb ns)
                  (L1-call scope-setup dest-fn scope-teardown result-exn return-xf)
 apply (clarsimp simp: L2corres-def L2-call-def L1-call-def L1-seq-def L1-throw-def
L2-defs)
  apply (rule corresXF-I)
  subgoal
   apply (clarsimp simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch

        reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
exception-or-result-splits)
    by (smt (z3) Exn-def Result-neq-Exn map-exn-simps(2) the-Result-simp)
  subgoal
   apply (clarsimp simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch

        reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
exception-or-result-splits)
   by (metis (mono-tags, opaque-lifting) Exception-eq-Exception Exn-def Exn-neq-Result
map-exn-simps(1))
  subgoal
    apply (auto simp add: corresXF-def succeeds-bind succeeds-catch reaches-catch
        reaches-map-value reaches-bind default-option-def Exn-def split: xval-splits
exception-or-result-splits)
    done
```

done

lemma (**in** *L1-functions*) *L2corres-dyn-call*:
L2corres st ret ex P X
(*L1-guarded g* (*gets dest* >>= (λ*p*. *L1-call scope-setup* (*P p*) *scope-teardown result-exn return-xf*))) ⟹
L2corres st ret ex P X (*L1-dyn-call g scope-setup dest scope-teardown result-exn return-xf*)
by (*simp add*: *L1-dyn-call-def*)

lemma *L2-gets-bind*: ⟦ ⋀*s s'*. *V s* = *V s'* ⟧ ⟹ *L2-seq* (*L2-gets V n*) *f* = *f* (*V undefined*)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*simp add*: *runs-to-iff*)
by (*auto simp add*: *runs-to-def-old*) *metis*+

lemma *L2corres-folded-gets*:
⟦ ⋀*a*. *L2corres st ret ex* (*P and* (λ*s*. *a* = *c* (*st s*))) (*X a*) *Y* ⟧ ⟹
L2corres st ret ex P (*L2-seq* (*L2-folded-gets c ns*) *X*) *Y*
by (*fastforce simp add*: *L2-defs L2corres-def corresXF-def succeeds-bind reaches-bind split*: *xval-splits*)

lemma *L2-call-cong* [*fundef-cong*, *cong*]:
f = *f'* ⟹ *L2-call f* = *L2-call f'*
by *simp*

lemma *L2-call-liftE* [*simp*]:
L2-call (*liftE x*) *emb ns* ≡ *liftE x*
by (*clarsimp simp add*: *L2-call-def*)

lemma *L2-call-fail* [*simp*]: *L2-call fail emb ns* = *fail*
apply (*simp add*: *L2-call-def*)
done

lemma *L2-call-L2-gets* [*simp*]: *L2-call* (*L2-gets x n*) *emb ns* = *L2-gets x n*
apply (*simp add*: *L2-defs L2-call-def*)
done

lemma *L2-split-fixup-1*:
(*L2-seq A* (λ*x*. *case y of* (*a*, *b*) ⇒ *B a b x*)) =
(*case y of* (*a*, *b*) ⇒ *L2-seq A* (λ*x*. *B a b x*))
by (*auto simp*: *split-def*)

lemma *L2-split-fixup-2*:
(*L2-seq* (*case y of* (*a*, *b*) ⇒ *B a b*) *A*) =
(*case y of* (*a*, *b*) ⇒ *L2-seq* (*B a b*) *A*)
by (*auto simp*: *split-def*)

lemma *L2-split-L2-seq-fixup-both*:
(*L2-seq* (*case y of* (*a*,*b*) ⇒ *A a b*) (*case y of* (*a*, *b*) ⇒ *B a b*)) =
(*case y of* (*a*, *b*) ⇒ *L2-seq* (*A a b*) (*B a b*))
by (*auto simp*: *split-def*)

lemma *L2-split-L2-seq-gets-fixup-1*:
(*L2-seq-gets* (*case y of* (*a*,*b*) ⇒ *A a b*) *n* (*case y of* (*a*, *b*) ⇒ *B a b*)) =
(*case y of* (*a*, *b*) ⇒ *L2-seq-gets* (*A a b*) *n* (*B a b*))
by (*auto simp*: *split-def*)

lemma *L2-split-fixup-3*:
(*case* (*x*, *y*) *of* (*a*, *b*) ⇒ *P a b*) = *P x y*
by (*auto simp*: *split-def*)
lemma *L2-split-fixup-4*:
case-prod (λ*a* (*b* :: *'a* × *'b*). *P a*) = *case-prod* (λ*a*. *case-prod* (λ(*x* :: *'a*) (*y* :: *'b*). *P a*))
by (*auto simp*: *split-def*)
lemma *L2-split-fixup-f*:
(*f* (*case y of* (*a*, *b*) ⇒ *G a b*) =
(*case y of* (*a*, *b*) ⇒ *f* (*G a b*)))
by (*auto simp*: *split-def*)
lemma *L2-split-fixup-g*:
case-prod (λ*a* (*b* :: *'a* × *'b*). *P a b*) = *case-prod* (λ*a*. *case-prod* (λ(*x* :: *'a*) (*y* :: *'b*). *P a* (*x*, *y*)))
by (*auto simp*: *split-def*)

lemma *liftE-fixup*: (λ*x*. *liftE* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *liftE* (*f a b*))
by (*simp add*: *fun-eq-iff split*: *prod.splits*)

lemma *finally-fixup*: (λ*x*. *finally* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *finally* (*f a b*))
by (*simp add*: *fun-eq-iff split*: *prod.splits*)

lemma *try-fixup*: (λ*x*. *try* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *try* (*f a b*))
by (*simp add*: *fun-eq-iff split*: *prod.splits*)

lemma *L2-gets-const-split-fixup*: *L2-gets* (λ-. *case y of* (*a*, *b*) ⇒ *G a b*) = (*case y of* (*a*, *b*) ⇒ *L2-gets* (λ-. *G a b*))
by (*simp add*: *fun-eq-iff split*: *prod.splits*)

definition *STOP* :: *'a*::{} ⇒ *'a*
where *STOP P* ≡ *P*

lemma *STOP-cong*: *STOP P* ≡ *STOP P*

by *simp*

lemma *do-STOP*: $P \equiv STOP\ P$
by (*simp add*: *STOP-def*)

definition *STOP-UNBIND* :: $'a::\{\} \Rightarrow 'a$
where $STOP\text{-}UNBIND\ P\ \equiv P$

lemma *STOP-UNBIND-cong*: $STOP\text{-}UNBIND\ P \equiv STOP\text{-}UNBIND\ P$
by *simp*

lemma *do-STOP-UNBIND*: $P \equiv STOP\text{-}UNBIND\ P$
by (*simp add*: *STOP-UNBIND-def*)

definition *FUSE* :: $'a::\{\} \Rightarrow 'a$
where $FUSE\ P\ \equiv P$

lemma *FUSE-cong*: $FUSE\ P \equiv FUSE\ P$
by *simp*

lemma *do-FUSE*: $P \equiv FUSE\ P$
by (*simp add*: *FUSE-def*)

lemma *FUSE-STOP*: $X \equiv X' \Longrightarrow FUSE\ X \equiv STOP\ X'$
by (*simp add*: *FUSE-def STOP-def*)

lemma *fixup-accumulate*:
$(\lambda(x,\ a).\ (case\ a\ of\ (x1,\ x2) \Rightarrow C\ x1\ x2)\ x) = (\lambda(x,\ x1,\ x2).\ C\ x1\ x2\ x)$
by (*auto simp*: *split-def*)

lemma *case-prod-apply-distrib*:
$(case\ x\ of\ (a,\ b) \Rightarrow f\ a\ b)\ n = \ (case\ x\ of\ (a,\ b) \Rightarrow f\ a\ b\ n\ )$
by (*auto simp add*: *split-def*)

lemma *L2-guard-fixup1*: $L2\text{-}guard\ (\lambda s.\ case\ y\ of\ (a,\ b) \Rightarrow G\ a\ b\ s) = (case\ y\ of\ (a,\ b) \Rightarrow L2\text{-}guard\ (G\ a\ b))$
by (*auto simp*: *split-def*)

lemmas *L2-split-fixups-base* =

L2-split-fixup-3
L2-split-fixup-4
liftE-fixup
finally-fixup
try-fixup

L2-guard-fixup1
L2-split-fixup-f [**where** *f*=*L2-guard*]
L2-split-fixup-f [**where** *f*=*L2-gets*]

L2-split-fixup-f [**where** *f=L2-modify*]

L2-gets-const-split-fixup

L2-split-fixup-g [**where** *P=λa b. L2-gets* (*P a b*) *n* **for** *P n*]
L2-split-fixup-g [**where** *P=λa b. L2-guard* (*P a b*) **for** *P*]
L2-split-fixup-g [**where** *P=λa b. L2-modify* (*P a b*) **for** *P*]
L2-split-fixup-g [**where** *P=λa b. L2-spec* (*P a b*) **for** *P*]
L2-split-fixup-g [**where** *P=λa b. L2-assume* (*P a b*) **for** *P*]
L2-split-fixup-g [**where** *P=λa b. L2-throw* (*P a b*) *n* **for** *P n*]

L2-split-fixup-g [**where** *P=λa b. L2-seq* (*L a b*) (*R a b*) **for** *L R*]
L2-split-fixup-g [**where** *P=λa b. L2-while* (*C a b*) (*B a b*) (*I a b*) *n* **for** *C B I n*]
L2-split-fixup-g [**where** *P=λa b. L2-unknown n* **for** *n*]
L2-split-fixup-g [**where** *P=λa b. L2-catch* (*L a b*) (*R a b*) **for** *L R*]
L2-split-fixup-g [**where** *P=λa b. L2-condition* (*C a b*) (*L a b*) (*R a b*) **for** *C L R*]
L2-split-fixup-g [**where** *P=λa b. L2-call* (*M a b*) **for** *M*]
L2-split-fixup-g [**where** *P=λa b. liftE* (*M a b*) **for** *M*]
L2-split-fixup-g [**where** *P=λa b. finally* (*M a b*) **for** *M*]
L2-split-fixup-g [**where** *P=λa b. try* (*M a b*) **for** *M*]

lemmas *L2-split-fixups* =
L2-split-fixup-1
L2-split-fixup-2
L2-split-fixups-base

lemmas *L2-split-fixups′* =
L2-split-L2-seq-fixup-both
L2-split-L2-seq-gets-fixup-1
L2-split-fixup-g [**where** *P=λa b. L2-seq-gets* (*L a b*) (*R a b*) **for** *L R*]
fixup-accumulate
L2-split-fixups-base
L2-split-fixup-f [**where** *f=STOP*]
case-prod-apply-distrib

lemmas *L2-split-fixups-congs* =
prod.case-cong

named-theorems *L2opt*

lemma *monotone-nondet-monad-L2-seq-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: ⋀*r. monotone R* (≤) (*λf. Y f r*)
shows *monotone R* (≤)
(*λf.* (*L2-seq* (*X f*) (*Y f*)))
unfolding *L2-defs*

apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-seq-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: ⋀*r*. *monotone R* (≥) (λ*f*. *Y f r*)
shows *monotone R* (≥)
(λ*f*. (*L2-seq* (*X f*) (*Y f*)))
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-try-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
shows *monotone R* (≤)
(λ*f*. (*L2-try* (*X f*)))
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
done

lemma *monotone-nondet-monad-L2-try-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
shows *monotone R* (≥)
(λ*f*. (*L2-try* (*X f*)))
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
done

lemma *monotone-nondet-monad-L2-catch-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: ⋀*r*. *monotone R* (≤) (λ*f*. *Y f r*)
shows *monotone R* (≤)
(λ*f*. (*L2-catch* (*X f*) (*Y f*)))
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-catch-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: ⋀*r*. *monotone R* (≥) (λ*f*. *Y f r*)
shows *monotone R* (≥)

(λf. (L2-catch (X f) (Y f)))
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-condition-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: *monotone R* (≤) *Y*
shows *monotone R* (≤)
(λf. (L2-condition C (X f) (Y f)))
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-condition-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: *monotone R* (≥) *Y*
shows *monotone R* (≥)
(λf. (L2-condition C (X f) (Y f)))
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-X*)
apply (*rule mono-Y*)
done

lemma *monotone-nondet-monad-L2-guarded-le* [*partial-function-mono*]:
assumes *mono-X*[*partial-function-mono*]: *monotone R* (≤) *X*
shows *monotone R* (≤)
(λf. L2-guarded C (X f))
unfolding *L2-guarded-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-nondet-monad-L2-guarded-ge* [*partial-function-mono*]:
assumes *mono-X*[*partial-function-mono*]: *monotone R* (≥) *X*
shows *monotone R* (≥)
(λf. L2-guarded C (X f))
unfolding *L2-guarded-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-nondet-monad-L2-while-le* [*partial-function-mono*]:
assumes *mono-B*: ⋀r. *monotone R* (≤) (λf. *B f r*)
shows *monotone R* (≤) (λf. *L2-while C* (*B f*) *I ns*)

unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-B*)
done

lemma *monotone-nondet-monad-L2-while-ge* [*partial-function-mono*]:
assumes *mono-B*: $\bigwedge r.$ *monotone* R $(\geq)$ $(\lambda f.$ B f $r)$
shows *monotone* R $(\geq)$ $(\lambda f.$ *L2-while* C $(B$ $f)$ I $ns)$
unfolding *L2-defs*
apply (*rule partial-function-mono*)
apply (*rule mono-B*)
done

lemma *monotone-nondet-monad-map-exn-le* [*partial-function-mono*]:
assumes X[*partial-function-mono*]: *monotone* R $(\leq)$ X
shows *monotone* R $(\leq)$ $(\lambda f.$ *map-value* (*map-exn emb*) $(X$ $f))$
unfolding *map-exn-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-nondet-monad-map-exn-ge* [*partial-function-mono*]:
assumes X[*partial-function-mono*]: *monotone* R $(\geq)$ X
shows *monotone* R $(\geq)$ $(\lambda f.$ *map-value* (*map-exn emb*) $(X$ $f))$
unfolding *map-exn-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-nondet-monad-L2-call-le* [*partial-function-mono*]:
assumes X[*partial-function-mono*]: *monotone* R $(\leq)$ X
shows *monotone* R $(\leq)$
$(\lambda f.$ *L2-call* $(X$ $f)$ *emb ns*)
unfolding *L2-call-def*
apply (*rule partial-function-mono*)+
done

lemma *monotone-nondet-monad-L2-call-ge* [*partial-function-mono*]:
assumes X[*partial-function-mono*]: *monotone* R $(\geq)$ X
shows *monotone* R $(\geq)$
$(\lambda f.$ *L2-call* $(X$ $f)$ *emb ns*)
unfolding *L2-call-def*
apply (*rule partial-function-mono*)+
done

19.1 Some Relators

lemma *exists-sum-unit-eq*: $(\exists\, l'{::}unit.\ Inl\ x = Inl\ l' \wedge Inr\ y = Inr\ l') = (x{=}() \wedge y{=}())$
by *auto*

lemmas *unit-convs* = *exists-sum-unit-eq*

definition *rel-Nonlocal* = (λ*e v*. *case e of Nonlocal x* ⇒ *v* = *x* | - ⇒ *False*)

lemma *rel-Nonlocal-conv*: (*rel-Nonlocal e v*) = (*e* = *Nonlocal v*)
by (*cases e*) (*auto simp add*: *rel-Nonlocal-def*)

lemma *fun-of-rel-rel-Nonlocal*[*fun-of-rel-intros*]: *fun-of-rel rel-Nonlocal the-Nonlocal*
by (*auto simp add*: *fun-of-rel-def rel-Nonlocal-def split*: *c-exntype.splits*)

lemma *funp-rel-Nonlocal*[*funp-intros*, *corres-admissible*]: *funp rel-Nonlocal*
using *fun-of-rel-rel-Nonlocal*
by (*auto simp add*: *funp-def*)

lemma *rel-sum-rel-Nonlocal-case-InlI*: *e* = *Nonlocal v'* ⟹ *rel-sum rel-Nonlocal* (=) (*case e of Nonlocal v* ⇒ *Inl* (*Nonlocal v*) | - ⇒ *Inr x*) (*Inl v'*)
by (*simp add*: *rel-Nonlocal-def*)

lemma *rel-xval-rel-Nonlocal-case-ExnI*: *e* = *Nonlocal v'* ⟹ *rel-xval rel-Nonlocal* (=) (*case e of Nonlocal v* ⇒ *Exn* (*Nonlocal v*) | - ⇒ *Result x*) (*Exn v'*)
by (*simp add*: *rel-Nonlocal-def*)

lemma *rel-sum-rel-Nonlocal-case-InrI*: *is-local e* ⟹ *x* = *v'* ⟹ *rel-sum rel-Nonlocal* (=) (*case e of Nonlocal v* ⇒ *Inl* (*Nonlocal v*) | - ⇒ *Inr x*) (*Inr v'*)
apply (*cases e*)
apply (*auto simp add*: *rel-Nonlocal-def*)
done

lemma *rel-xval-rel-Nonlocal-case-ResultI*: *is-local e* ⟹ *x* = *v'* ⟹ *rel-xval rel-Nonlocal* (=) (*case e of Nonlocal v* ⇒ *Exn* (*Nonlocal v*) | - ⇒ *Result x*) (*Result v'*)
apply (*cases e*)
apply (*auto simp add*: *rel-Nonlocal-def*)
done

lemma *rel-sum-rel-Nonlocal-map-sumI*: *v* = (*map-sum* (λ*x*. *x*) *f* (*case e of Nonlocal x* ⇒ *Inl x* | - ⇒ *Inr x*)) ⟹
rel-sum rel-Nonlocal (=) (*map-sum Nonlocal f* (*case e of Nonlocal x* ⇒ *Inl x* | - ⇒ *Inr x*)) *v*
apply *simp*
apply (*cases e*)
apply (*auto simp add*: *rel-Nonlocal-def*)
done

lemma *rel-xval-rel-Nonlocal-map-xvalI*: *v* = (*map-xval* (λ*x*. *x*) *f* (*case e of Nonlocal x* ⇒ *Exn x* | - ⇒ *Result x*)) ⟹
rel-xval rel-Nonlocal (=) (*map-xval Nonlocal f* (*case e of Nonlocal x* ⇒ *Exn x* | - ⇒ *Result x*)) *v*
apply *simp*

apply (*cases e*)
  apply (*auto simp add*: *rel-Nonlocal-def*)
done

definition *rel-Inr v v'* ≡ (*v* = *Inr v'*)

lemma *fun-of-rel-rel-Inr*[*fun-of-rel-intros*]:
fun-of-rel rel-Inr (λ*x*. *case x of Inr v* ⇒ *v* | *Inl* - ⇒ *undefined*)
by (*auto simp add*: *rel-Inr-def fun-of-rel-def*)

lemma *funp-rel-Inr*[*funp-intros*, *corres-admissible*]: *funp rel-Inr*
using *fun-of-rel-rel-Inr* **by** (*auto simp add*: *funp-def*)

lemma *rel-InrI*: *v* = *Inr v'* ⟹ *rel-Inr v v'*
by (*simp add*: *rel-Inr-def*)

lemma *rel-Inr-apply*: *rel-Inr x y* = (*x* = *Inr y*)
by (*simp add*: *rel-Inr-def*)

lemma *rel-Inr-trivial* [*simp*]: *rel-Inr* (*Inr v*) *v*
by (*simp add*: *rel-Inr-def*)

definition *rel-liftE*:: ('*e*, '*a*) *xval* ⇒ '*a val* ⇒ *bool* **where** *rel-liftE v v'* ≡ (*v* = *Result* (*the-Res v'*))

lemma *fun-of-rel-rel-liftE*[*fun-of-rel-intros*]:
fun-of-rel rel-liftE (λ*x*. *case x of Result v* ⇒ *Result v* | *Exn* - ⇒ *undefined*)
by (*auto simp add*: *rel-liftE-def fun-of-rel-def*)

lemma *funp-rel-liftE*[*funp-intros*, *corres-admissible*]: *funp rel-liftE*
using *fun-of-rel-rel-liftE* **by** (*auto simp add*: *funp-def*)

lemma *rel-liftEI*: *v* = *Result* (*the-Res v'*) ⟹ *rel-liftE v v'*
by (*simp add*: *rel-liftE-def*)

lemma *rel-liftE-apply*: *rel-liftE x y* = (*x* = *Result* (*the-Res y*))
by (*simp add*: *rel-liftE-def*)

lemma *rel-liftE-trivial* [*simp*]: *rel-liftE* (*Result v*) (*Result v*)
by (*simp add*: *rel-liftE-def*)

lemma *rel-liftE-rel-exception-or-result-conv*: *rel-liftE* = *rel-exception-or-result* (λ*None* ⇒ λ-. *True* | *Some* - ⇒ λ-. *False*) (=)

apply (*rule ext*)+
apply (*auto simp add*: *rel-liftE-def rel-exception-or-result.simps*)
done

lemma *rel-liftE-Result-Result-iff*[*simp*]: *rel-liftE* (*Result v*) (*Result v'*) ⟷ *v* = *v'*
by (*auto simp add*: *rel-liftE-def*)

definition *rel-liftE'* *v v'* ≡ (*v* = *Inr v'*)

lemma *fun-of-rel-rel-liftE'*[*fun-of-rel-intros*]:
fun-of-rel rel-liftE' (λ*x*. *case x of Inr v* ⇒ *v* | *Inl* - ⇒ *undefined*)
by (*auto simp add*: *rel-liftE'-def fun-of-rel-def*)

lemma *funp-rel-liftE'*[*funp-intros*, *corres-admissible*]: *funp rel-liftE'*
using *fun-of-rel-rel-liftE'* **by** (*auto simp add*: *funp-def*)

lemma *rel-liftE'I*: *v* = *Inr v'* ⟹ *rel-liftE' v v'*
by (*simp add*: *rel-liftE'-def*)

lemma *rel-liftE'-apply*: *rel-liftE' x y* = (*x* = *Inr y*)
by (*simp add*: *rel-liftE'-def*)

lemma *rel-liftE'-trivial* [*simp*]: *rel-liftE'* (*Inr v*) *v*
by (*simp add*: *rel-liftE'-def*)

lemma *rel-liftE'-Inr-iff*[*simp*]: *rel-liftE'* (*Inr v*) *v'* ⟷ *v* = *v'*
by (*auto simp add*: *rel-liftE'-def*)

lemma *rel-liftE'-rel-liftE-conv*: *rel-liftE'* = *rel-map to-xval OO rel-liftE OO rel-map the-Res*
apply (*rule ext*)+
apply (*auto simp add*: *relcompp.simps rel-map-def rel-liftE-def rel-liftE'-def*)
done

lemma *rel-liftE-rel-liftE'-conv*: *rel-liftE* = *rel-map from-xval OO rel-liftE' OO rel-map Result*
apply (*rule ext*)+
apply (*auto simp add*: *relcompp.simps rel-map-def rel-liftE-def rel-liftE'-def*)
done

lemma *rel-liftE'-Inr*: *rel-liftE'* (*Inr x*) *x*
by (*simp*)

definition *rel-throwE'* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a* + '*c* ⇒ '*b* ⇒ *bool* **where**
rel-throwE' L a c ⟷ (*case a of Inl a'* ⇒ *L a' c* | *Inr* - ⇒ *False*)

lemma *rel-throwE'-iff*: *rel-throwE' L a c* ⟷ (∃ *l*. *a* = *Inl l* ∧ *L l c*)

by (*auto simp add*: *rel-throwE'-def split*: *sum.splits*)

lemma *rel-throwE'-Inl*:
L x y ⟹ *rel-throwE' L* (*Inl x*) *y*
unfolding *rel-throwE'-def*
by *auto*

lemma *rel-throwE'-Inl-iff*[*simp*]:
rel-throwE' L (*Inl x*) *y* ⟷ *L x y*
unfolding *rel-throwE'-def*
by *auto*

lemma *not-rel-throwE'-Inr*[*simp*]: ¬*rel-throwE' R* (*Inr d*) *a*
by (*auto simp*: *rel-throwE'-def*)

definition *rel-throwE* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a*, '*c*) *xval* ⇒ '*b* ⇒ *bool* **where**
rel-throwE L a c ⟷ (*case a of Exn a'* ⇒ *L a' c* | *Result* - ⇒ *False*)

lemma *rel-throwE-iff*: *rel-throwE L a c* ⟷ (∃ *e*. *a* = *Exn e* ∧ *L e c*)
by (*auto simp add*: *rel-throwE-def split*: *xval-splits*)

lemma *rel-throwE-Exn*:
L x y ⟹ *rel-throwE L* (*Exn x*) *y*
unfolding *rel-throwE-def*
by *auto*

lemma *rel-throwE-Exn-iff*[*simp*]:
rel-throwE L (*Exn x*) *y* ⟷ *L x y*
unfolding *rel-throwE-def*
by *auto*

lemma *not-rel-throwE-Result*[*simp*]: ¬*rel-throwE R* (*Result d*) *a*
by (*auto simp*: *rel-throwE-def*)

lemma *case-right-eq*: (*case v of Inl l* ⇒ *False* | *Inr r* ⇒ *P r*) = (∃ *r*. *P r* ∧ *v* = *Inr r*)
by (*auto split*: *sum.splits*)
lemma *case-left-eq*: (*case v of Inr r* ⇒ *False* | *Inl l* ⇒ *P l*) = (∃ *l*. *P l* ∧ *v* = *Inl l*)
by (*auto split*: *sum.splits*)

lemma *rel-sum-right*: *rel-sum L R* (*Inr r*) *v* = (∃ *r'*. *v* = *Inr r'* ∧ *R r r'*)
by (*auto elim*: *rel-sum.cases intro*: *rel-sum.intros split*: *sum.splits*)
lemma *rel-sum-left*: *rel-sum L R* (*Inl l*) *v* = (∃ *l'*. *v* = *Inl l'* ∧ *L l l'*)
by (*auto elim*: *rel-sum.cases intro*: *rel-sum.intros split*: *sum.splits*)

lemma *rel-sum-eq-apply*: (*rel-sum* (=) (=) *x y*) = (*x* = *y*)
by (*cases x*) (*auto elim*: *rel-sum.cases intro*: *rel-sum.intros split*: *sum.splits*)

lemma *gen-unit-eq*: $x = (y::unit)$
by *simp*

lemma *liftE-L2-while*: *L2-while c* $(\lambda r.$ *liftE* $(B\ r))\ i\ n =$ *liftE* $($*whileLoop c B i*$)$
by $($*clarsimp simp*: *L2-while-def liftE-whileLoop*$)$

lemma *liftE-L2-while-VARS*: *L2-while c* $(\lambda r.$ *liftE* $(B\ r))\ i\ n =$ *liftE* $($*L2-VARS* $($*whileLoop c B i*$)\ n)$
by $($*simp add*: *liftE-L2-while L2-VARS-def*$)$

lemma *rel-XF-True-def*: $($*rel-XF st ret-xf ex-xf* $(\lambda$- -. *True*$)) =$
$(\lambda(v,\ t)\ (r,\ s).\ s = st\ t \wedge$
(case v of Exn x $\Rightarrow$ *r = Exn (ex-xf x t)* | *Result x* $\Rightarrow$ *r = Result (ret-xf x t)*$))$
apply $($*rule ext*$)+$
apply $($*auto simp add*: *rel-XF-def rel-xval.simps split*: *xval-splits*$)$
done

lemma *refines-corresXF*: $(\forall\, t.\ P\ t \longrightarrow$ *refines C A t (st t)* $(\lambda(v,\ t)\ (r,\ s).\ s = st\ t \wedge$
(case v of Exn x $\Rightarrow$ *r = Exn (ex-xf x t)* | *Result x* $\Rightarrow$ *r = Result (ret-xf x t)*$))) \Longrightarrow$ *corresXF st ret-xf ex-xf P A C*
by $($*simp add*: *corresXF-refines-iff rel-XF-True-def*$)$

lemma *corresXF-refines*: *corresXF st ret-xf ex-xf P A C* $\Longrightarrow$
$(\forall\, t.\ P\ t \longrightarrow$ *refines C A t (st t)* $(\lambda(v,\ t)\ (r,\ s).\ s = st\ t \wedge$
(case v of Exn x $\Rightarrow$ *r = Exn (ex-xf x t)* | *Result x* $\Rightarrow$ *r = Result (ret-xf x t)*$))$
by $($*simp add*: *corresXF-refines-iff rel-XF-True-def*$)$

theorem *corresXF-refines-conv*:
$\langle$*corresXF st ret-xf ex-xf P A C* $\longleftrightarrow$
$(\forall\, t.\ P\ t \longrightarrow$ *refines C A t (st t)* $(\lambda(v,\ t)\ (r,\ s).\ s = st\ t \wedge$
(case v of Exn x $\Rightarrow$ *r = Exn (ex-xf x t)* | *Result x* $\Rightarrow$ *r = Result (ret-xf x t)*$))\rangle$
using *corresXF-refines refines-corresXF* **..**

lemma *sim-nondet-to-corresXF*: $(\bigwedge s.$ *refines f f' s s* $(=)) \Longrightarrow$
corresXF $(\lambda s.\ s)$ $(\lambda r$ -. $r)$ $(\lambda r$ - . $r)$ $(\lambda$-. *True*$)\ f'\ f$
unfolding *refines-def-old corresXF-def*
apply $($*auto split*: *xval-splits*$)$
done

named-theorems *rel-spec-monad-rewrite-simps*

locale *sim-stack-heap-state* =
abstract: *stack-heap-state* htd_a *htd-upd*$_a$ $hmem_a$ *hmem-upd*$_a$ $\mathcal{S}$ +
concrete: *stack-heap-state* htd_c *htd-upd*$_c$ $hmem_c$ *hmem-upd*$_c$ $\mathcal{S}$

```
  for
    htd_a:: 's_a ⇒ heap-typ-desc and
    htd-upd_a:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's_a ⇒ 's_a and
    hmem_a:: 's_a ⇒ heap-mem and
    hmem-upd_a:: (heap-mem ⇒ heap-mem) ⇒ 's_a ⇒ 's_a and

    htd_c:: 's_c ⇒ heap-typ-desc and
    htd-upd_c:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's_c ⇒ 's_c and
    hmem_c:: 's_c ⇒ heap-mem and
    hmem-upd_c:: (heap-mem ⇒ heap-mem) ⇒ 's_c ⇒ 's_c and
    S::addr set
begin

definition rel-stack-free where
  rel-stack-free s_c s_a ≡ stack-free (htd_c s_c) ⊆ stack-free (htd_a s_a)


lemma refines-rel-stack-free-with-fresh-stack-ptr:
  assumes s: rel-stack-free s_c s_a
  assumes init-eq: ⋀s_c s_a. I_c s_c = I_a s_a
  assumes f: ⋀s_c s_a p. rel-stack-free s_c s_a ⟹
    refines (f_c p) (f_a p) s_c s_a
      (rel-prod (rel-xval L R) rel-stack-free)
  shows refines
     (concrete.with-fresh-stack-ptr n I_c f_c)
     (abstract.with-fresh-stack-ptr n I_a f_a) s_c s_a
       (rel-prod (rel-xval L R) rel-stack-free)
 apply (simp add: concrete.with-fresh-stack-ptr-def abstract.with-fresh-stack-ptr-def)
 apply (rule refines-bind-bind-exn [where Q=(rel-prod (rel-xval (λ- -. False) (=))
rel-stack-free)])
     apply simp-all
 subgoal
   apply (clarsimp simp add: refines-def-old rel-xval.simps)
   using s init-eq
   apply (simp add: rel-stack-free-def)
   subgoal for p d vs bs
     apply (frule (1) stack-allocs-stack-free-mono)
     apply clarsimp
     subgoal for d'
       apply (rule exI[where x=hmem-upd_a  (fold (λi. heap-update-padding (p
+_p int i) (vs ! i) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs)))
[0..<length vs]) (htd-upd_a (λ-. d') s_a)])
       apply clarsimp
       apply (rule conjI)
        apply (rule exI[where x=d'])
        apply clarsimp
        apply (rule exI[where x=vs])
        apply clarsimp
        apply blast
```

```
      using stack-free-stack-allocs
      apply blast
      done
    done
  done
 apply (rule refines-rel-prod-on-exit [where S' = rel-stack-free])
  apply (rule f)
  apply assumption
  apply (simp add: rel-stack-free-def)
 apply (metis abstract.htd-hmem-upd abstract.typing.get-upd concrete.htd-hmem-upd

    concrete.typing.get-upd stack-free-stack-releases-mono')
 subgoal using stack-free-stack-releases-mono' by blast
 done

definition rel-stack-free-eq where
 rel-stack-free-eq s_c s_a  ≡ stack-free (htd_c s_c) = stack-free (htd_a s_a)

lemma refines-rel-stack-free-eq-with-fresh-stack-ptr:
 assumes s: rel-stack-free-eq s_c s_a
 assumes init-eq: ⋀s_c s_a. I_c s_c = I_a s_a
 assumes f: ⋀s_c s_a p. rel-stack-free-eq s_c s_a ⟹
   refines (f_c p) (f_a p) s_c s_a
    (rel-prod (rel-xval L R) rel-stack-free-eq)
 shows refines
    (concrete.with-fresh-stack-ptr n I_c f_c)
    (abstract.with-fresh-stack-ptr n I_a f_a) s_c s_a
      (rel-prod (rel-xval L R) rel-stack-free-eq)
 apply (simp add: concrete.with-fresh-stack-ptr-def abstract.with-fresh-stack-ptr-def)
 apply (rule refines-bind-bind-exn [where Q=(rel-prod (rel-xval (λ- -. False) (=))
rel-stack-free-eq)])
     apply simp-all
 subgoal
   apply (clarsimp simp add: refines-def-old rel-xval.simps)
   using s init-eq
   apply (simp add: rel-stack-free-eq-def)
   subgoal for p d vs bs
     apply (frule (1) stack-allocs-stack-free-eq)
     apply clarsimp
     subgoal for d'
        apply (rule exI[where x=hmem-upd_a (fold (λi. heap-update-padding (p
+_p int i) (vs ! i) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs)))
[0..<length vs]) (htd-upd_a (λ-. d') s_a)])
       apply clarsimp
       apply (rule conjI)
        apply (rule exI[where x=d'])
        apply clarsimp
        apply (rule exI[where x=vs])
        apply clarsimp
```

apply *blast*
using *stack-free-stack-allocs*
apply *blast*
done
done
done
apply (*rule refines-rel-prod-on-exit* [**where** $S' = rel\text{-}stack\text{-}free\text{-}eq$])
apply (*rule f*)
apply *assumption*
apply *clarsimp*
subgoal for *v t t' s_c s_a bs*
apply (*rule exI*[**where** *x*= $hmem\text{-}upd_a$ (*heap-update-list* (*ptr-val v*) *bs*) ($htd\text{-}upd_a$ (*stack-releases n v*) s_a)])
apply (*clarsimp simp add*: *rel-stack-free-eq-def*)
by (*metis dual-order.eq-iff stack-free-stack-releases-mono'*)
subgoal by *blast*
done

end

locale *sim-stack-heap-raw-state* =
abstract: *stack-heap-raw-state* hrs_a $hrs\text{-}upd_a$ $\mathcal{S}$ +
concrete: *stack-heap-raw-state* hrs_c $hrs\text{-}upd_c$ $\mathcal{S}$
for
hrs_a:: $'s_a \Rightarrow$ *heap-raw-state* **and**
$hrs\text{-}upd_a$:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ $'s_a \Rightarrow$ $'s_a$ **and**

hrs_c:: $'s_c \Rightarrow$ *heap-raw-state* **and**
$hrs\text{-}upd_c$:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ $'s_c \Rightarrow$ $'s_c$ **and**
$\mathcal{S}$::*addr set*
begin
sublocale *sim-stack-heap-state*
$\lambda s.\ hrs\text{-}htd\ (hrs_a\ s)\ \lambda upd.\ hrs\text{-}upd_a\ (hrs\text{-}htd\text{-}update\ upd)$
$\lambda s.\ hrs\text{-}mem\ (hrs_a\ s)\ \lambda upd.\ hrs\text{-}upd_a\ (hrs\text{-}mem\text{-}update\ upd)$
$\lambda s.\ hrs\text{-}htd\ (hrs_c\ s)\ \lambda upd.\ hrs\text{-}upd_c\ (hrs\text{-}htd\text{-}update\ upd)$
$\lambda s.\ hrs\text{-}mem\ (hrs_c\ s)\ \lambda upd.\ hrs\text{-}upd_c\ (hrs\text{-}mem\text{-}update\ upd)$
$\mathcal{S}$
by *unfold-locales*
end

named-theorems *L2corres-with-fresh-stack-ptr*

locale *L2* = *sim-stack-heap-state* htd_a $htd\text{-}upd_a$ $hmem_a$ $hmem\text{-}upd_a$ htd_c $htd\text{-}upd_c$ $hmem_c$ $hmem\text{-}upd_c$ $\mathcal{S}$
for
htd_a:: $'s_a \Rightarrow$ *heap-typ-desc* **and**
$htd\text{-}upd_a$:: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'s_a \Rightarrow$ $'s_a$ **and**
$hmem_a$:: $'s_a \Rightarrow$ *heap-mem* **and**

```
    hmem-upd_a:: (heap-mem ⇒ heap-mem) ⇒ 's_a ⇒ 's_a and

    htd_c:: 's_c ⇒ heap-typ-desc and
    htd-upd_c:: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 's_c ⇒ 's_c and
    hmem_c:: 's_c ⇒ heap-mem and
    hmem-upd_c:: (heap-mem ⇒ heap-mem) ⇒ 's_c ⇒ 's_c and
    S::addr set +
  fixes st:: 's_c ⇒ 's_a
  assumes htd-st: ⋀s. htd_a (st s) = htd_c s
  assumes htd-upd-st: ⋀s f. (htd-upd_a f (st s)) = st (htd-upd_c f s)
  assumes hmem-upd-st: ⋀s f. (hmem-upd_a f (st s)) = st (hmem-upd_c f s)

begin
lemma rel-stack-free-eq-st: rel-stack-free-eq s (st s)
  by (simp add: rel-stack-free-eq-def htd-st)

definition rel-L2 where
  rel-L2 ex-xf ret-xf ≡ (λ (r_c, s_c) (r_a, s_a).
     rel-xval (λ- a. a = ex-xf s_c) (λ- a. a = ret-xf s_c) r_c r_a ∧
     (s_a = st s_c))

lemma rel-L2-def':
  (rel-L2 ex-xf ret-xf) = (λ(v, t) (r, s).
              s = st t ∧ (case v of Exn x ⇒ r = Exn (ex-xf t) | Result x ⇒ r =
Result (ret-xf t)))
  apply  (clarsimp simp add: rel-L2-def fun-eq-iff, intro iffI)
  subgoal using rel-xval.cases by fastforce
  subgoal by(auto split: xval-splits)
  done


lemma refines-rel-L2-on-exit':
  assumes f: refines (f_c p) (f_a p) s_c (st s_c) (rel-L2 ex-xf ret-xf)
  assumes ex-xf-htd-indep: ⋀f s. ex-xf (htd-upd_c f s) = ex-xf s
  assumes ret-xf-htd-indep: ⋀f s. ret-xf (htd-upd_c f s) = ret-xf s
  shows refines
    (on-exit (f_c p) {(s,t). t = htd-upd_c (release p) s})
    (on-exit (f_a p) {(s,t). t = htd-upd_a (release p) s}) s_c (st s_c)
    (rel-L2 ex-xf ret-xf)
  unfolding on-exit-bind-exception-or-result-conv
  apply (rule refines-bind-exception-or-result)
  apply (rule refines-mono [OF - f])
  apply (clarsimp)
  subgoal for r s q t
    apply (rule refines-bind-bind-exn[where
      Q=(λ(r', s) (q', t). is-Result r' ∧ is-Result q' ∧ rel-L2 ex-xf ret-xf (r, s) (q,
t))])
    subgoal
      apply (rule refines-assert-result-and-state)
```

```
      apply (clarsimp simp add: rel-L2-def, intro conjI)
     subgoal
       using ex-xf-htd-indep ret-xf-htd-indep
       by metis
     subgoal
       using htd-upd-st by force
     by auto
   apply (auto simp: is-Result-def)
   done
 done

lemma refines-rel-L2-on-exit:
 assumes f: refines (f_c p) (f_a p) s_c (st s_c) (rel-L2 ex-xf ret-xf)
 assumes ex-xf-htd-indep: ⋀g f s. ex-xf (hmem-upd_c g (htd-upd_c f s)) = ex-xf s
 assumes ret-xf-htd-indep: ⋀g f s. ret-xf (hmem-upd_c g (htd-upd_c f s)) = ret-xf s
 assumes nonempty-cleanup_a: cleanup_a ≠ {}
 assumes cleanup-htd: ⋀s s'. (s, s') ∈ cleanup_c ⟹ ∃g f. s' = (hmem-upd_c g
(htd-upd_c f s))
 assumes stuck-sim: ⋀s. ∄s'. (s, s') ∈ cleanup_c ⟹ ∄t'. (st s, t') ∈ cleanup_a
 assumes cleanup-sim: ⋀s t. (s, t) ∈ cleanup_c ⟹ (st s, st t) ∈ cleanup_a
 shows refines
   (on-exit (f_c p) cleanup_c)
   (on-exit (f_a p) cleanup_a) s_c (st s_c)
   (rel-L2 ex-xf ret-xf)
 unfolding on-exit-bind-exception-or-result-conv
 apply (rule refines-bind-exception-or-result)
 apply (rule refines-mono [OF - f])
 apply (clarsimp)
 subgoal for r s q t
   apply (rule refines-bind-bind-exn [where
     Q=λ(r', s) (q', t). is-Result r' ∧ is-Result q' ∧ rel-L2 ex-xf ret-xf (r, s) (q,
t)])
    subgoal
     apply (rule refines-state-select)
      apply (clarsimp simp add: rel-L2-def, intro conjI)
     subgoal
       using cleanup-htd cleanup-sim
       by blast
     subgoal
       using ex-xf-htd-indep ret-xf-htd-indep cleanup-htd
       by metis
     subgoal
       using stuck-sim by (auto simp: rel-L2-def)
     done
   apply (auto simp: is-Result-def)
   done
 done
```

lemma *refines-rel-L2-with-fresh-stack-ptr*:
assumes *init-eq*: $I_c\ s_c = I_a\ (st\ s_c)$
assumes *ex-xf-htd-indep*: $\bigwedge g\ f\ s.\ ex\text{-}xf\ (hmem\text{-}upd_c\ g\ (htd\text{-}upd_c\ f\ s)) = ex\text{-}xf\ s$
assumes *ret-xf-htd-indep*: $\bigwedge g\ f\ s.\ ret\text{-}xf\ (hmem\text{-}upd_c\ g\ (htd\text{-}upd_c\ f\ s)) = ret\text{-}xf\ s$
assumes *f*: $\bigwedge(p::'a::mem\text{-}type\ ptr)\ d\ vs\ bs.\ (p,\ d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ (htd_c\ s_c) \Longrightarrow vs \in I_c\ s_c \Longrightarrow length\ vs = n \Longrightarrow$
$length\ bs = n * size\text{-}of\ TYPE('a) \Longrightarrow$
refines
$(f_c\ p)$
$(f_a\ p)$
$(hmem\text{-}upd_c\ (fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (vs\ !\ i)\ (take\ (size\text{-}of\ TYPE('a))\ (drop\ (i * size\text{-}of\ TYPE('a))\ bs)))\ [0..<length\ vs])\ ((htd\text{-}upd_c\ (\lambda\text{-}.\ d))\ s_c))$
$(st\ (hmem\text{-}upd_c\ (fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (vs\ !\ i)\ (take\ (size\text{-}of\ TYPE('a))\ (drop\ (i * size\text{-}of\ TYPE('a))\ bs)))\ [0..<length\ vs])\ ((htd\text{-}upd_c\ (\lambda\text{-}.\ d))\ s_c)))$
$(rel\text{-}L2\ ex\text{-}xf\ ret\text{-}xf)$
shows *refines*
$(concrete.with\text{-}fresh\text{-}stack\text{-}ptr\ n\ I_c\ f_c)$
$(abstract.with\text{-}fresh\text{-}stack\text{-}ptr\ n\ I_a\ f_a)\ s_c\ (st\ s_c)$
$(rel\text{-}L2\ ex\text{-}xf\ ret\text{-}xf)$
apply (*simp add*: *concrete.with-fresh-stack-ptr-def abstract.with-fresh-stack-ptr-def*)
apply (*rule refines-bind-bind-exn* [**where** $Q=(\lambda(r_c,\ t_c)\ (r_a,\ t_a).$
$(\exists\ d\ p\ vs\ bs.\ (p,\ d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ (htd_c\ s_c) \wedge r_c = Result\ p \wedge r_a = Result\ p \wedge vs \in I_c\ s_c \wedge length\ vs = n \wedge$
$length\ bs = n * size\text{-}of\ TYPE('a) \wedge$
$t_c = hmem\text{-}upd_c\ (fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (vs\ !\ i)\ (take\ (size\text{-}of\ TYPE('a))\ (drop\ (i * size\text{-}of\ TYPE('a))\ bs)))\ [0..<length\ vs])\ ((htd\text{-}upd_c\ (\lambda\text{-}.\ d))\ s_c) \wedge$
$t_a = st\ t_c))$])
apply *simp-all*
subgoal
apply (*clarsimp simp add*: *refines-def-old rel-xval.simps*)
by (*metis* (*full-types*) *hmem-upd-st htd-st htd-upd-st init-eq*)
subgoal for *p*
apply *clarsimp*
apply (*rule refines-rel-L2-on-exit*)
apply (*rule f*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply (*rule ex-xf-htd-indep*)
apply (*rule ret-xf-htd-indep*)
apply *blast*
subgoal by *auto*
subgoal by *auto*
subgoal using *htd-upd-st hmem-upd-st* **by** *auto*
done

done

lemma *L2corres-refines-rel-L2-conv*:
‹*L2corres st ret-xf ex-xf P A C* ⟷
(∀ *t*. *P t* ⟶ *refines C A t* (*st t*) (*rel-L2 ex-xf ret-xf*))›
by (*simp add*: *L2corres-def corresXF-refines-conv rel-L2-def'*)

lemma *L2corres-refines-rel-L2*:
assumes *L2*: *L2corres st ret-xf ex-xf P A C*
assumes *P*: *P* s_c
shows ‹*refines C A* s_c (*st* s_c) (*rel-L2 ex-xf ret-xf*)›
using *L2 P* **by** (*simp add*: *L2corres-refines-rel-L2-conv*)

lemma *L2corres-with-fresh-stack-ptr*[*L2corres-with-fresh-stack-ptr*]:
assumes *l2*: ⋀*p*. *L2corres st ret-xf ex-xf P* (*l2 p*) (*l1 p*)
assumes *I*: ⋀*s*. *R s* ⟹ *I-l1 s* = *I-l2* (*st s*)
assumes *P*: ⋀*s*. *R s* ⟹ *P s*
assumes *ex-xf-htd-indep*: ⋀*g f s*. *ex-xf* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *ex-xf s*
assumes *ret-xf-htd-indep*: ⋀*g f s*. *ret-xf* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *ret-xf s*
assumes *P-indep*: ⋀*f g s*. *P* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *P s*
shows *L2corres st ret-xf ex-xf R*
(*abstract.with-fresh-stack-ptr n* (*I-l2*) (*L2-VARS l2 nm*))
(*concrete.with-fresh-stack-ptr n I-l1 l1*)
apply (*clarsimp simp add*: *L2-VARS-def L2corres-refines-rel-L2-conv*)
apply (*rule refines-rel-L2-with-fresh-stack-ptr*)
apply (*rule I*, *assumption*)
apply (*rule ex-xf-htd-indep*)
apply (*rule ret-xf-htd-indep*)
apply (*rule L2corres-refines-rel-L2*)
apply (*rule l2*)
apply (*simp add*: *P-indep P*)
done

end

hide-const (**open**) *L2*

lemma *refines-rel-prod-guard-on-exit*:
assumes *f*: *refines* f_c f_a s_c s_a (*rel-prod R S'*)
assumes *cleanup*: ⋀s_c s_a t_c. *S'* s_c s_a ⟹ *grd* s_a ⟹ (s_c, t_c) ∈ *cleanup*$_c$ ⟹ ∃ t_a. (s_a, t_a) ∈ *cleanup*$_a$ ∧ *S* t_c t_a
assumes *emp*: ⋀s_c s_a. *S'* s_c s_a ⟹ ∄ t_c. (s_c, t_c) ∈ *cleanup*$_c$ ⟹ ∄ t_a. (s_a, t_a) ∈ *cleanup*$_a$
shows *refines* (*on-exit* f_c *cleanup*$_c$) (*guard-on-exit* f_a *grd cleanup*$_a$) s_c s_a (*rel-prod R S*)
unfolding *on-exit-def on-exit'-def*
apply (*rule refines-bind-exception-or-result*[*OF refines-weaken*, *OF f*])
apply (*simp add*: *bind-assoc*)

apply (*rule refines-bind-guard-right*)
apply (*rule refines-bind'*[*OF refines-state-select*])
subgoal using *cleanup* **by** *auto*
subgoal using *emp* **by** *auto*
done

lemma *refines-bind-no-throw-strong*:
assumes *no-throw* (λ-. *True*) *f*
assumes *no-throw* (λ-. *True*) *f'*
assumes *f*: *refines f f' s s' Q*
assumes *g*: ⋀*r t r' t'*. *reaches f s* (*Result r*) *t* ⟹ *reaches f' s'* (*Result r'*) *t'* ⟹
Q (*Result r*, *t*) (*Result r'*, *t'*) ⟹ *refines* (*g r*) (*g' r'*) *t t' R*
shows *refines* (*f* >>= *g*) (*f'* >>= *g'*) *s s' R*
apply (*rule refines-bind-bind-strong'* [*OF f*])
subgoal using *assms*
apply (*clarsimp simp add*: *no-throw-def runs-to-partial-def-old*)
by (*metis the-Exception-simp the-Exception-Result reaches-succeeds*)
subgoal using *assms*
apply (*clarsimp simp add*: *no-throw-def runs-to-partial-def-old*)
by (*metis the-Exception-simp the-Exception-Result reaches-succeeds*)
subgoal using *assms*
apply (*clarsimp simp add*: *no-throw-def runs-to-partial-def-old*)
by (*metis the-Exception-simp the-Exception-Result reaches-succeeds*)
subgoal using *g*
by *auto*
done

lemma *refines-bind-no-throw*:
assumes *no-throw* (λ-. *True*) *f*
assumes *no-throw* (λ-. *True*) *f'*
assumes *f*: *refines f f' s s' Q*
assumes *g*: ⋀*r t r' t'*.
Q (*Result r*, *t*) (*Result r'*, *t'*) ⟹ *refines* (*g r*) (*g' r'*) *t t' R*
shows *refines* (*f* >>= *g*) (*f'* >>= *g'*) *s s' R*
using *assms*
by (*rule refines-bind-no-throw-strong*)

lemma *no-throw-guard*[*simp*]: *no-throw P* (*guard G*)
by (*auto simp add*: *no-throw-def runs-to-partial-def-old*)

lemma *no-throw-assert-result-and-state*[*simp*]: *no-throw P* (*assert-result-and-state f*)
by (*auto simp add*: *no-throw-def runs-to-partial-def-old*)

lemma *no-throw-assume-result-and-state*[*simp*]: *no-throw P* (*assume-result-and-state f*)
by (*auto simp add*: *no-throw-def runs-to-partial-def-old*)

lemma *refines-guard-same*: *refines* (*guard P*) (*guard P*) *s s* (λ(*r*, *t*) (*r'*, *t'*). *t*=*s* ∧

$t'{=}s$)
by (*simp add*: *refines-def-old*)

lemma *refines-state-select-same*:
refines (*state-select R*) (*state-select R*) *s s* (λ(*r*, *t*) (*r'*, *t'*). *t'* = *t* ∧ (*s*, *t*) ∈ *R*)
by (*simp add*: *refines-def-old*)

lemma *refines-assert-result-and-state-same*:
refines (*assert-result-and-state R*) (*assert-result-and-state R*) *s s*
(λ(*r*, *t*) (*r'*, *t'*). ∃ *v*. (*v*, *t*) ∈ *R s* ∧ *r* = *Result v* ∧ *r'* = *Result v* ∧ *t* = *t'*)
by (*auto simp add*: *refines-def-old*)

lemma *refines-assume-result-and-state-same*:
refines (*assume-result-and-state R*) (*assume-result-and-state R*) *s s*
(λ(*r*, *t*) (*r'*, *t'*). ∃ *v*. (*v*, *t*) ∈ *R s* ∧ *r* = *Result v* ∧ *r'* = *Result v* ∧ *t* = *t'*)
by (*auto simp add*: *refines-def-old*)

lemma *refines-assume-result-and-state-same'*:
assumes *R s* = *R' s*
shows *refines* (*assume-result-and-state R*) (*assume-result-and-state R'*) *s s*
(λ(*r*, *t*) (*r'*, *t'*). ∃ *v*. (*v*, *t*) ∈ *R s* ∧ *r* = *Result v* ∧ *r'* = *Result v* ∧ *t* = *t'*)
using *assms*
by (*auto simp add*: *refines-def-old*)

lemma *refines-rel-xval-guard-on-exit*:
assumes *f*: *refines* f_c f_a *s s* (*rel-prod* (*rel-xval L R*) (=))
shows *refines*
(*guard-on-exit* f_c *P cleanup*)
(*guard-on-exit* f_a *P cleanup*) *s s*
(*rel-prod* (*rel-xval L R*) (=))
unfolding *guard-on-exit-bind-exception-or-result-conv*
apply (*rule refines-bind-exception-or-result'*)
apply (*rule f*)
apply *clarsimp*
apply (*rule refines-bind-no-throw-strong* [*OF - - refines-guard-same*])
apply *simp*
apply *simp*
apply *clarsimp*
apply (*rule refines-bind-no-throw-strong* [*OF - - refines-state-select-same*])
apply *simp*
apply *simp*
apply *clarsimp*
done

lemma *refines-rel-xval-on-exit*:
assumes *f*: *refines* f_c f_a *s s* (*rel-prod* (*rel-xval L R*) (=))
shows *refines*
(*on-exit* f_c *cleanup*)
(*on-exit* f_a *cleanup*) *s s*

```
        (rel-prod (rel-xval L R) (=))
  unfolding on-exit-bind-exception-or-result-conv
  apply (rule refines-bind-exception-or-result')
   apply (rule f)
  apply clarsimp
  apply (rule refines-bind-no-throw-strong [OF - - refines-state-select-same])
     apply simp
    apply simp
  apply clarsimp
  done


lemma refines-rel-xval-assume-on-exit:
  assumes f: refines f_c f_a s s (rel-prod (rel-xval L R) (=))
  shows refines
        (assume-on-exit f_c P cleanup)
        (assume-on-exit f_a P cleanup) s s
        (rel-prod (rel-xval L R) (=))
  unfolding assume-on-exit-bind-exception-or-result-conv
  apply (rule refines-bind-exception-or-result')
   apply (rule f)

  apply (rule refines-bind-no-throw-strong [OF - - ])
     apply simp
    apply simp
  apply clarsimp
   apply (rule refines-assume-result-and-state-same)
  apply clarsimp
  apply (rule refines-bind-no-throw-strong [OF - - refines-state-select-same])
     apply simp
   apply simp
  apply clarsimp
  done

lemma refines-state-assume-pred:
  assumes P: P t
  assumes ref: refines f g s t R
  shows refines f (do {assume-result-and-state (λs. {(x, y). (λ() s'. s' = s ∧ P s)
x y}); g }) s t R
  using P ref
  by (auto simp add: refines-def-old succeeds-bind reaches-bind)

lemma refines-rel-prod-assume-on-exit:
  assumes f: refines f_c f_a s_c s_a (rel-prod R S')
  assumes cleanup: ⋀s_c s_a t_c. S' s_c s_a ⟹ (s_c, t_c) ∈ cleanup_c ⟹ ∃ t_a. (s_a, t_a)
∈ cleanup_a ∧ S t_c t_a
  assumes emp: ⋀s_c s_a. S' s_c s_a ⟹ ∄ t_c. (s_c, t_c) ∈ cleanup_c ⟹ ∄ t_a. (s_a, t_a) ∈
cleanup_a
  assumes conseq: ⋀s_c s_a. S' s_c s_a ⟹ P s_a
```

shows *refines* (*on-exit* f_c *cleanup*$_c$) (*assume-on-exit* f_a *P cleanup*$_a$) s_c s_a (*rel-prod R S*)
unfolding *assume-on-exit-bind-exception-or-result-conv on-exit-bind-exception-or-result-conv*
apply (*rule refines-bind-exception-or-result'*)
apply (*rule f*)
apply (*rule refines-state-assume-pred*)
subgoal using *conseq* **by** *auto*
subgoal
apply (*rule refines-bind-no-throw-strong* [**where** *Q* = *rel-prod* (λ- -. *True*) *S*])
apply *simp*
apply *simp*
apply (*rule refines-state-select*)
subgoal using *cleanup* **by** *auto*
subgoal using *emp* **by** *auto*
subgoal
apply (*rule refines-yield*)
apply *auto*
done
done
done

lemma *refines-runs-to-partial-rel-prod-assume-on-exit*:
assumes *f*: *refines* f_c f_a s_c s_a (*rel-prod R S'*)
assumes *runs-to*: $f_c \bullet s_c$ *?⦃*λ*r t. P t⦄*
assumes *cleanup*: $\bigwedge s_c\ s_a\ t_c.\ S'\ s_c\ s_a \Longrightarrow (s_c,\ t_c) \in cleanup_c \Longrightarrow P\ s_c \Longrightarrow \exists\, t_a.\ (s_a,\ t_a) \in cleanup_a \wedge S\ t_c\ t_a$
assumes *emp*: $\bigwedge s_c\ s_a.\ S'\ s_c\ s_a \Longrightarrow P\ s_c \Longrightarrow \nexists\, t_c.\ (s_c,\ t_c) \in cleanup_c \Longrightarrow \nexists\, t_a.\ (s_a,\ t_a) \in cleanup_a$
assumes *conseq*: $\bigwedge s_c\ s_a.\ \ S'\ s_c\ s_a\ \Longrightarrow P\ s_c \Longrightarrow P'\ s_a$
shows *refines* (*on-exit* f_c *cleanup*$_c$) (*assume-on-exit* f_a P' *cleanup*$_a$) s_c s_a (*rel-prod R S*)
proof −
from *refines-runs-to-partial-fuse* [*OF f runs-to*]
have *refines* f_c f_a s_c s_a (λ(*r*, *t*) (*r'*, *t'*). *rel-prod R S'* (*r*, *t*) (*r'*, *t'*) ∧ *P t*) **.**
moreover have (λ(*r*, *t*) (*r'*, *t'*). *rel-prod R S'* (*r*, *t*) (*r'*, *t'*) ∧ *P t*) = *rel-prod R* (λ*t t'*. *S' t t'* ∧ *P t*)
by (*auto simp add*: *rel-prod-conv*)
ultimately have *refines* f_c f_a s_c s_a (*rel-prod R* (λ*t t'*. *S' t t'* ∧ *P t*)) **by** *simp*
then show *?thesis*
apply (*rule refines-rel-prod-assume-on-exit*)
subgoal using *cleanup* **by** *blast*
subgoal using *emp* **by** *blast*
subgoal using *conseq* **by** *auto*
done
qed

locale *L2-heap-raw-state* = *sim-stack-heap-raw-state* hrs_a *hrs-upd*$_a$ hrs_c *hrs-upd*$_c$ $\mathcal{S}$
for

hrs_a:: $'s_a \Rightarrow$ *heap-raw-state* **and**
hrs-upd$_a$:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ $'s_a \Rightarrow$ $'s_a$ **and**

hrs_c:: $'s_c \Rightarrow$ *heap-raw-state* **and**
hrs-upd$_c$:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ $'s_c \Rightarrow$ $'s_c$ **and**
$\mathcal{S}$::*addr set* +
fixes *st*:: $'s_c \Rightarrow$ $'s_a$
assumes *hrs-htd-st*: $\bigwedge s$. *hrs-htd* (*hrs*$_a$ (*st s*)) = *hrs-htd* (*hrs*$_c$ *s*)
assumes *hrs-upd-st*: $\bigwedge s\ f$. (*hrs-upd*$_a$ *f* (*st s*)) = *st* (*hrs-upd*$_c$ *f s*)

begin
sublocale *L2*
λs. *hrs-htd* (*hrs*$_a$ *s*) λupd. *hrs-upd*$_a$ (*hrs-htd-update upd*)
λs. *hrs-mem* (*hrs*$_a$ *s*) λupd. *hrs-upd*$_a$ (*hrs-mem-update upd*)
λs. *hrs-htd* (*hrs*$_c$ *s*) λupd. *hrs-upd*$_c$ (*hrs-htd-update upd*)
λs. *hrs-mem* (*hrs*$_c$ *s*) λupd. *hrs-upd*$_c$ (*hrs-mem-update upd*)
$\mathcal{S}$
apply (*unfold-locales*)
subgoal using *hrs-htd-st* **by** *simp*
subgoal using *hrs-upd-st* **by** *simp*
subgoal using *hrs-upd-st* **by** *simp*
done
end

locale *typ-heap-typing* = *pointer-lense r w* + *heap-typing-state heap-typing heap-typing-upd*

for
r:: $'t \Rightarrow$ ($'a$::*xmem-type*) *ptr* $\Rightarrow$ $'a$ **and**
w:: $'a$ *ptr* $\Rightarrow$ ($'a \Rightarrow 'a$) $\Rightarrow$ $'t \Rightarrow 't$ **and**
heap-typing :: $'t \Rightarrow$ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'t \Rightarrow 't$ **and**
$\mathcal{S}$:: *addr set*
begin

definition *stack-ptr-acquire* :: *nat* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *ptr* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ $'t \Rightarrow 't$
where *stack-ptr-acquire n vs p d s* =
(*fold* (λi. *w* (*p* +$_p$ *int i*) (λ-. (*vs* ! *i*))) [*0*..<*n*]) (*heap-typing-upd* (λ-. *d*) *s*)

definition *stack-ptr-release* :: *nat* $\Rightarrow$ $'a$ *ptr* $\Rightarrow$ $'t \Rightarrow 't$
where *stack-ptr-release n p s* =
(*heap-typing-upd* (*stack-releases n p*) *o* (*fold* (λi. *w* (*p* +$_p$ *int i*) (λ-. *c-type-class.zero*))
[*0*..<*n*])) *s*

definition *assume-stack-alloc*:: *nat* $\Rightarrow$ ($'t \Rightarrow$ ($'a$::*xmem-type*) *list set*) $\Rightarrow$ ($'e$::*default*,
$'a$ *ptr*, $'t$) *spec-monad* **where**
assume-stack-alloc n init $\equiv$ *assume-result-and-state* (λs. {(*p*, *t*).
$\exists$ *d vs*. (*p*, *d*) $\in$ *stack-allocs n* $\mathcal{S}$ *TYPE*($'a$::*xmem-type*) (*heap-typing s*) $\wedge$
vs $\in$ *init s* $\wedge$
length vs = *n* $\wedge$

$(\forall\, i \in \{0..<n\}.\ r\ s\ (p +_p\ int\ i) = ZERO('a::xmem\text{-}type)) \wedge$
$t = stack\text{-}ptr\text{-}acquire\ n\ vs\ p\ d\ s\})$

definition *guard-with-fresh-stack-ptr* :: *nat* ⇒ (*'t* ⇒ *'a list set*) ⇒ (*'a::xmem-type ptr* ⇒ (*'e::default*, *'v*, *'t*) *spec-monad*) ⇒ (*'e*, *'v*, *'t*) *spec-monad* **where**
guard-with-fresh-stack-ptr n init c ≡
do {
p ← *assume-stack-alloc n init*;
guard-on-exit (*c p*)
(λ*s*. ∀ *i* < *n*. *root-ptr-valid* (*heap-typing s*) (*p* +$_p$ *int i*))
{(*s*, *t*). *t* = *stack-ptr-release n p s*}
}

definition *assume-with-fresh-stack-ptr* :: *nat* ⇒ (*'t* ⇒ *'a list set*) ⇒ (*'a::xmem-type ptr* ⇒ (*'e::default*,*'v*, *'t*) *spec-monad*) ⇒ (*'e*, *'v*, *'t*) *spec-monad* **where**
assume-with-fresh-stack-ptr n init c ≡
do {
p ← *assume-stack-alloc n init*;
assume-on-exit (*c p*)
(λ*s*. ∀ *i* < *n*. *root-ptr-valid* (*heap-typing s*) (*p* +$_p$ *int i*))
{(*s*, *t*). *t* = *stack-ptr-release n p s*}
}

definition *with-fresh-stack-ptr* :: *nat* ⇒ (*'t* ⇒ *'a list set*) ⇒ (*'a::xmem-type ptr* ⇒ (*'e::default*,*'v*, *'t*) *spec-monad*) ⇒ (*'e*, *'v*, *'t*) *spec-monad* **where**
with-fresh-stack-ptr n init c ≡
do {
p ← *assume-stack-alloc n init*;
on-exit (*c p*)
{(*s*, *t*). *t* = *stack-ptr-release n p s*}
}

lemma *monotone-guard-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: ⋀*p*. *monotone R* (≤) (λ*f*. *c f p*)
shows *monotone R* (≤) (λ*f*. *guard-with-fresh-stack-ptr n I* (*c f*))
unfolding *guard-with-fresh-stack-ptr-def*
by (*intro partial-function-mono*)

lemma *monotone-guard-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: ⋀*p*. *monotone R* (≥) (λ*f*. *c f p*)
shows *monotone R* (≥) (λ*f*. *guard-with-fresh-stack-ptr n I* (*c f*))
unfolding *guard-with-fresh-stack-ptr-def*
by (*intro partial-function-mono*)

lemma *monotone-assume-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: ⋀*p*. *monotone R* (≤) (λ*f*. *c f p*)
shows *monotone R* (≤) (λ*f*. *assume-with-fresh-stack-ptr n I* (*c f*))
unfolding *assume-with-fresh-stack-ptr-def*
by (*intro partial-function-mono*)

lemma *monotone-assume-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone* R $(\geq)$ $(\lambda f.\ c\ f\ p)$
shows *monotone* R $(\geq)$ $(\lambda f.\ \textit{assume-with-fresh-stack-ptr}\ n\ I\ (c\ f))$
unfolding *assume-with-fresh-stack-ptr-def*
by (*intro partial-function-mono*)

lemma *monotone-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone* R $(\leq)$ $(\lambda f.\ c\ f\ p)$
shows *monotone* R $(\leq)$ $(\lambda f.\ \textit{with-fresh-stack-ptr}\ n\ I\ (c\ f))$
unfolding *with-fresh-stack-ptr-def on-exit-def*
by (*intro partial-function-mono*)

lemma *monotone-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone* R $(\geq)$ $(\lambda f.\ c\ f\ p)$
shows *monotone* R $(\geq)$ $(\lambda f.\ \textit{with-fresh-stack-ptr}\ n\ I\ (c\ f))$
unfolding *with-fresh-stack-ptr-def on-exit-def*
by (*intro partial-function-mono*)

lemma *refines-rel-xval-guard-with-fresh-stack-ptr*:
assumes *init-eq*: $\textit{init}_c\ s = \textit{init}_a\ s$
assumes *f*: $\bigwedge s\ p.$ *refines* $(f_c\ p)$ $(f_a\ p)$ $s\ s$ (*rel-prod* (*rel-xval* L R) $(=)$)
shows
refines
(*guard-with-fresh-stack-ptr* n $\textit{init}_c$ (*L2-VARS* f_c *nm*))
(*guard-with-fresh-stack-ptr* n $\textit{init}_a$ (*L2-VARS* f_a *nm*)) $s\ s$
(*rel-prod* (*rel-xval* L R) $(=)$)
unfolding *guard-with-fresh-stack-ptr-def L2-VARS-def on-exit'-def assume-stack-alloc-def*
apply (*rule refines-bind'*)
apply (*subst refines-assume-result-and-state-iff*)
apply (*subst init-eq*)
apply (*rule sim-set-refl*)
apply *clarsimp*
apply (*rule refines-bind-exception-or-result'*[*OF f*])
apply (*rule refines-bind'*)
apply (*rule refines-bind'*)
apply (*rule refines-guard*)
apply *simp-all*
apply (*rule refines-assert-result-and-state*)
apply *simp-all*
done

lemma *refines-rel-xval-assume-with-fresh-stack-ptr*:
assumes *init-eq*: $\textit{init}_c\ s = \textit{init}_a\ s$
assumes *f*: $\bigwedge s\ p.$ *refines* $(f_c\ p)$ $(f_a\ p)$ $s\ s$ (*rel-prod* (*rel-xval* L R) $(=)$)
shows
refines
(*assume-with-fresh-stack-ptr* n $\textit{init}_c$ (*L2-VARS* f_c *nm*))

```
    (assume-with-fresh-stack-ptr n init_a (L2-VARS f_a nm)) s s
    (rel-prod (rel-xval L R) (=))
  unfolding assume-with-fresh-stack-ptr-def L2-VARS-def on-exit'-def assume-stack-alloc-def
  apply (rule refines-bind')
  apply (subst refines-assume-result-and-state-iff)
  apply (subst init-eq)
  apply (rule sim-set-refl)
  apply clarsimp
  apply (rule refines-bind-exception-or-result'[OF f])
  apply (rule refines-bind')
  apply (rule refines-bind')
  apply (rule refines-assuming)
  apply simp-all
  apply (rule refines-assert-result-and-state)
  apply simp-all
  done

lemma refines-rel-xval-with-fresh-stack-ptr:
  assumes init-eq: init_c s = init_a s
  assumes f: ⋀s p. refines (f_c p) (f_a p) s s (rel-prod (rel-xval L R) (=))
  shows
    refines
      (with-fresh-stack-ptr n init_c (L2-VARS f_c nm))
      (with-fresh-stack-ptr n init_a (L2-VARS f_a nm)) s s
      (rel-prod (rel-xval L R) (=))
  unfolding with-fresh-stack-ptr-def L2-VARS-def assume-stack-alloc-def
   apply (rule refines-bind-no-throw [where Q = rel-prod (rel-xval (λ- -. False)
(=)) (=)])
  apply simp
    apply simp
  subgoal using init-eq by (auto simp add: refines-def-old rel-xval.simps)
  subgoal
    apply clarsimp
    apply (rule refines-rel-xval-on-exit)
    apply (rule f)
    done
  done


lemma write-same-ZERO:
  r s p = ZERO('a) ⟹ w p (λy. ZERO('a)) s = s
  using write-same by simp

end

lemma refines-rel-prod-eq-on-exit:
  assumes f: refines f_c f_a s s (rel-prod Q (=))
  shows refines
        (on-exit f_c cleanup)
```

```
      (on-exit f_a cleanup) s s
      (rel-prod Q (=))
  unfolding on-exit-bind-exception-or-result-conv
  apply (rule refines-bind-exception-or-result')
   apply (rule f)
  apply clarsimp
  apply (rule refines-bind-no-throw )
  apply simp
  apply simp
   apply (rule refines-state-select-same)
  apply clarsimp
  done

context stack-heap-state
begin
lemma refines-rel-prod-with-fresh-stack-ptr:
  assumes init-eq: init_c s = init_a s
  assumes f: ⋀s p. refines (f_c p) (f_a p) s s (rel-prod Q (=))
  shows
    refines
      (with-fresh-stack-ptr n init_c (L2-VARS f_c nm))
      (with-fresh-stack-ptr n init_a (L2-VARS f_a nm)) s s
      (rel-prod Q (=))
  unfolding with-fresh-stack-ptr-def L2-VARS-def
  apply (rule refines-bind-no-throw )
    apply simp
   apply simp
   apply (rule refines-assume-result-and-state-same')
   apply (simp only: init-eq)
  apply clarsimp
  apply (rule refines-rel-prod-eq-on-exit)
  apply (rule f)
  done
end

lemma h-val-coerce-ptr-coerce-packed:
  fixes p::'c::packed-type ptr
  assumes sz-eq: size-of TYPE('c) = size-of TYPE('a::packed-type)
  shows h-val h (PTR-COERCE ('c → 'a) p) = COERCE ('c → 'a) (h-val h p)
  by (simp add:  h-val-def sz-eq from-bytes-def coerce-def to-bytes-p-def to-bytes-def

      packed-type-access-ti access-ti_0-def field-access-update-same(1)
      size-of-fold sz-eq td-fafu-idem update-ti-t-def)


lemma h-val-field-ptr-coerce-from-bytes-packed:
  fixes p::'c::packed-type ptr
  assumes field-ti TYPE('a) f = Some t
  assumes export-uinfo t = export-uinfo (typ-info-t TYPE('b::mem-type))
```

assumes [*simp*]: *size-of TYPE('c) = size-of TYPE('a)*
shows *h-val h (PTR('b) &((PTR-COERCE('c::packed-type → 'a::packed-type) p)→f)) =*
from-bytes (access-ti$_0$ t (COERCE ('c → 'a) (h-val h p)))
apply (*simp add:h-val-coerce-ptr-coerce-packed*[*symmetric*])
apply (*rule h-val-field-from-bytes'* [*OF assms(1) assms(2)*])
done

lemma *packed-type-value-update-ti-explode*:
(v::'a::packed-type) = update-ti (typ-info-t TYPE('a)) (to-bytes-p v) v'
by (*simp add*: *size-of-def update-ti-s-from-bytes update-ti-update-ti-t*)

lemma *packed-type-to-bytes-to-bytes-p-conv*:
length bs = size-of TYPE('a::packed-type) ⟹ to-bytes v bs = to-bytes-p (v::'a)
apply (*simp add*: *to-bytes-p-def*)
by (*simp add*: *packed-type-access-ti to-bytes-def*)

lemma *packed-type-to-byte-p-coerce*:
assumes *sz:size-of TYPE('c::packed-type) = size-of TYPE('a::packed-type)*
shows *to-bytes-p (COERCE('a → 'c) v) = to-bytes-p v*
using *sz*
apply (*simp add*: *to-bytes-p-def coerce-def*)
by (*metis* (*mono-tags*, *lifting*) *field-access-update-same(1) field-desc.simps(2) field-desc-def*
from-bytes-def len length-replicate size-of-fold td-fafu-idem to-bytes-def
update-ti-t-def wf-fd)

lemma *to-bytes-coerce-packed*:
size-of TYPE('a) = size-of TYPE('b) ⟹ length bs = size-of TYPE('a) ⟹
to-bytes (COERCE('a::packed-type→'b::packed-type) v) bs = to-bytes v bs
by (*simp add*: *coerce-def field-access-update-same(1) from-bytes-def packed-type-access-ti*

size-of-def td-fafu-idem to-bytes-def update-ti-t-def)

lemma *heap-update-ptr-coerse-packed*:
fixes *p::'c::packed-type ptr*
assumes *cgrd*: *c-guard p*
assumes [*simp*]: *size-of TYPE('c) = size-of TYPE('a::packed-type)*
shows *heap-update (PTR-COERCE('c → 'a) p) v = heap-update p (COERCE('a →'c) v)*
apply (*rule ext*)
apply (*simp add*: *heap-update-def packed-type-to-bytes-to-bytes-p-conv packed-type-to-byte-p-coerce*)
done

lemma *c-guard-ptr-coerceI*:
size-td (typ-info-t TYPE('c)) = size-td (typ-info-t TYPE('a)) ⟹
align-td (typ-info-t TYPE('a)) ≤ align-td (typ-info-t TYPE('c)) ⟹
c-guard p ⟹

c-guard (*PTR-COERCE*('*c*::*c-type*→'*a*::*c-type*) *p*)
apply (*cases p*)
using *power-le-dvd*
by (*auto simp add*: *c-guard-def c-null-guard-def ptr-aligned-def align-of-def size-of-def*)

lemma *heap-update-field-ptr-coerce-from-bytes-packed*:
fixes *p*::'*c*::{*xmem-type*, *packed-type*} *ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*::{*xmem-type*, *packed-type*})) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*b*::{*xmem-type*})
assumes *cgrd*: *c-guard* (*PTR-COERCE*('*c*→ '*a*) *p*)
assumes *sz*[*simp*]: *size-of TYPE*('*c*) = *size-of TYPE*('*a*)
shows *heap-update* (*PTR*('*b*) &((*PTR-COERCE*('*c*→ '*a*) *p*)→*f*)) *v h* =
heap-update p (*coerce-map* (*update-ti t* (*to-bytes-p v*)) (*h-val h p*)) *h*
apply (*subst heap-update-field-root-conv'* [*OF fl match cgrd*])
apply (*simp add*: *heap-update-def h-val-coerce-ptr-coerce-packed*)
apply (*subst coerce-coerce-map-cancel-packed*[*OF sz*[*symmetric*], *symmetric*])
apply (*simp add*: *coerce-map-def*)
apply (*subst to-bytes-coerce-packed*)
apply (*simp-all*)
done

named-theorems *L2-modify-heap-update-field-root-conv*

context *heap-state*
begin
lemma *heap-update-field-root-conv*:
fixes *p*::'*a*::*xmem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
shows (*hmem-upd* (*heap-update* (*PTR*('*b*) &(*p*→*f*)) *v*)) =
(λ*s*. (*hmem-upd* (*heap-update p* (*fld-update* (λ-. *v*) (*h-val* (*hmem s*) *p*)))) *s*)
apply (*subst heap-update-field-root-conv-pointless'* [*OF fl fg-cons cgrd*])
apply (*rule ext*)
by (*metis* (*mono-tags*, *lifting*) *heap.upd-cong*)

lemma *heap-update-field-root-conv'*:
fixes *p*::'*a*::*xmem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b*::*xmem-type*)
assumes *cgrd*: *c-guard p*
shows (*hmem-upd* (*heap-update* (*PTR*('*b*) &(*p*→*f*)) *v*)) =
(λ*s*. (*hmem-upd* (*heap-update p* (*update-ti t* (*to-bytes-p v*) (*h-val* (*hmem s*) *p*)))) *s*)
using *heap-update-field-root-conv'* [*OF fl match cgrd*, *of v*]

using *heap.upd-cong* **by** *blast*

lemma *L2-modify-heap-update-field-root-conv*:
fixes *p*::*'a*::*xmem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR*(*'b*) &(*p*→*f*)) (*v s*)) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*fld-update* (λ-. *v s*) (*h-val* (*hmem s*) *p*)))) *s*)
by (*simp add*: *heap-update-field-root-conv* [*OF fl fg-cons cgrd*])

lemma *L2-modify-heap-update-field-root-conv'* [*L2-modify-heap-update-field-root-conv*]:
fixes *p*::*'a*::*xmem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*(*'b*::*xmem-type*)
assumes *cgrd*: *c-guard p*
shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR*(*'b*) &(*p*→*f*)) (*v s*)) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*update-ti t* (*to-bytes-p* (*v s*)) (*h-val* (*hmem s*) *p*)))) *s*)
by (*simp add*: *heap-update-field-root-conv'* [*OF fl match cgrd*])

lemma *L2-modify-heap-update-ptr-coerce-packed-conv'*:
fixes *p*::*'c*::*packed-type ptr*
assumes *cgrd*: *c-guard p*
assumes *sz*: *size-td* (*typ-info-t TYPE*(*'c*)) = *size-td* (*typ-info-t TYPE*(*'a*::*packed-type*))

shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR-COERCE*(*'c* → *'a*) *p*) (*COERCE*(*'c* → *'a*) (*v s*))) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*v s*))) *s*)
using *heap-update-ptr-coerse-packed* [*simplified size-of-def*, *OF cgrd sz*] *heap.upd-cong*
by (*metis assms*(2) *coerce-cancel-packed size-of-def*)

lemma *L2-modify-heap-update-ptr-coerce-packed-conv*[*L2-modify-heap-update-field-root-conv*]:
fixes *p*::*'c*::*packed-type ptr*
assumes *cgrd*: *c-guard p*
assumes *sz*: *size-td* (*typ-info-t TYPE*(*'c*)) = *size-td* (*typ-info-t TYPE*(*'a*::*packed-type*))

shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR-COERCE*(*'c* → *'a*) *p*) (*v s*)) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*COERCE*(*'a* → *'c*) (*v s*)))) *s*)
using *heap-update-ptr-coerse-packed* [*simplified size-of-def*, *OF cgrd sz*] *heap.upd-cong*
by (*metis assms*(2) *coerce-cancel-packed size-of-def*)

lemma *L2-modify-heap-update-ptr-coerce-packed-field-root-conv* [*L2-modify-heap-update-field-root-conv*]:
fixes *p*::*'c*::{*xmem-type*, *packed-type*} *ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*::{*xmem-type*, *packed-type*})) *f 0* =

Some (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b*::*xmem-type*)
assumes *cgrd*: *c-guard* (*PTR-COERCE*('*c*→ '*a*) *p*)
assumes *sz*: *size-td* (*typ-info-t TYPE*('*c*)) = *size-td* (*typ-info-t TYPE*('*a*))
shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR*('*b*) &((*PTR-COERCE*('*c*→ '*a*) *p*)→*f*)) (*v s*)) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*coerce-map* (*update-ti t* (*to-bytes-p* (*v s*))) (*h-val* (*hmem s*) *p*)))) *s*)
using *heap-update-field-ptr-coerce-from-bytes-packed* [*simplified size-of-def*, *OF fl match cgrd sz*] *heap.upd-cong*
by *meson*
end

lemma *L2-try-state-asssumeE-bindE*:
L2-try (*assume-result-and-state P* >>= *f*) = *assume-result-and-state P* >>= (λ*x*. *L2-try* (*f x*))
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff L2-defs*)
done

lemma *refines-L2-try-L2-seq-nondet*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*L2-try* (*g v*)) (*g*′ *v*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f f*′ *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*f*′ >>= *g*′) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-try-def L2-seq-def try-nested-bind-exception-or-result-conv*
using *g f*
apply (*clarsimp simp add*: *refines-def-old L2-defs succeeds-bind reaches-bind reaches-try succeeds-bind-exception-or-result*
rel-liftE-def unnest-exn-def reaches-bind-exception-or-result succeeds-bind-exception-or-result split: *xval-splits sum.splits*)
apply (*intro conjI allI impI*)
apply (*metis case-xval-simps*(*2*))
apply *clarsimp*
subgoal for *r s*′ *r*′
apply (*cases r*)
subgoal
apply (*cases r*′)
subgoal
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
by (*meson Exn-neq-Result obj-sumE*)

subgoal
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
subgoal for *s1 y r1*
apply (*cases r1*)
subgoal

```
          apply (clarsimp simp add: default-option-def Exn-def [symmetric])
        by (metis Exn-neq-Result Result-eq-Result case-xval-simps(1) old.sum.simps(5)
sum-all-ex(2))
         subgoal
           by (auto simp add: default-option-def Exn-def [symmetric])
         done
       done
     done
   subgoal
     apply (cases r')

     subgoal
       apply (clarsimp simp add: default-option-def Exn-def [symmetric])
       by (meson Exn-neq-Result sum-all-ex(2))
     subgoal
       apply (clarsimp simp add: default-option-def Exn-def [symmetric])
       subgoal for s1 r1
         apply (cases r1)
         subgoal
           apply (clarsimp simp add: default-option-def Exn-def [symmetric])
        by (metis Result-eq-Result case-xval-simps(1) old.sum.simps(6) sum-all-ex(2))
         subgoal
           by (metis case-xval-simps(2))
         done
       done
     done
   done
 done



lemma refines-L2-try-pure:
  assumes f: refines f (return f') s s (rel-prod rel-liftE (=))
  shows refines (L2-try f) (return f') s s (rel-prod rel-liftE (=))
  unfolding L2-defs L2-guarded-def using f
  by (fastforce simp add: refines-def-old reaches-try rel-liftE-def)


lemma refines-liftE-conv: refines f f' s t (rel-prod rel-liftE (=)) ⟷
 refines f (liftE f') s t (=)
 unfolding L2-defs L2-guarded-def
  by (auto simp add: refines-def-old reaches-liftE rel-liftE-def)

lemma refines-rel-liftE-L2-try-on-exit:
  assumes f: refines f_c f_a s s (rel-prod rel-liftE (=))
   shows refines (L2-try (on-exit f_c cleanup)) (on-exit f_a cleanup) s s (rel-prod
rel-liftE (=))
  using f unfolding L2-defs try-def refines-map-value-left-iff
  apply (rule refines-rel-prod-eq-on-exit[THEN refines-weaken])
```

apply (*auto simp*: *rel-liftE-def*)
done

lemma *map-value-on-exit*:
map-value g (*on-exit f cleanup*) = *on-exit* (*map-value g f*) *cleanup*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff on-exit-def on-exit'-def*)
done

lemma *L2-try-on-exit*: *L2-try* (*on-exit f cleanup*) = *on-exit* (*L2-try f*) *cleanup*
unfolding *L2-try-def try-def*
by (*simp add*: *map-value-on-exit*)

context *stack-heap-state*
begin
lemma *L2-try-with-fresh-stack-ptr*:
L2-try (*with-fresh-stack-ptr n init f*) = *with-fresh-stack-ptr n init* (λ*p*. *L2-try* (*f p*))
by (*simp add*: *with-fresh-stack-ptr-def L2-try-state-asssumeE-bindE L2-try-on-exit*)
end

lemma *map-value-guard-on-exit*:
map-value g (*guard-on-exit f P cleanup*) = *guard-on-exit* (*map-value g f*) *P cleanup*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff on-exit'-def*)
done

lemma *L2-try-guard-on-exit*:
L2-try (*guard-on-exit f P cleanup*) = *guard-on-exit* (*L2-try f*) *P cleanup*
unfolding *L2-try-def try-def*
by (*simp add*: *map-value-guard-on-exit*)

lemma *map-value-assume-on-exit*:
map-value g (*assume-on-exit f P cleanup*) = *assume-on-exit* (*map-value g f*) *P cleanup*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff on-exit'-def*)
done

lemma *L2-try-assume-on-exit*:
L2-try (*assume-on-exit f P cleanup*) = *assume-on-exit* (*L2-try f*) *P cleanup*
unfolding *L2-try-def try-def*
by (*simp add*: *map-value-assume-on-exit*)

context *typ-heap-typing*

begin
lemma *L2-try-guard-with-fresh-stack-ptr*:
L2-try (*guard-with-fresh-stack-ptr n init f*) = *guard-with-fresh-stack-ptr n init* (λ*p*. *L2-try* (*f p*))
unfolding *guard-with-fresh-stack-ptr-def L2-try-state-asssumeE-bindE assume-stack-alloc-def*

L2-try-guard-on-exit **..**

lemma *L2-try-assume-with-fresh-stack-ptr*:
L2-try (*assume-with-fresh-stack-ptr n init f*) = *assume-with-fresh-stack-ptr n init* (λ*p*. *L2-try* (*f p*))
unfolding *assume-with-fresh-stack-ptr-def L2-try-state-asssumeE-bindE assume-stack-alloc-def*

L2-try-assume-on-exit **..**

lemma *L2-try-with-fresh-stack-ptr*:
L2-try (*with-fresh-stack-ptr n init f*) = *with-fresh-stack-ptr n init* (λ*p*. *L2-try* (*f p*))
by (*simp add*: *with-fresh-stack-ptr-def L2-try-state-asssumeE-bindE assume-stack-alloc-def L2-try-on-exit*)
end

lemma *refines-L2-seq*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*. *Q* (*Exn e*, *t*) (*Exn e'*, *t'*) ⟹ *R* (*Exn e*, *t*) (*Exn e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exn e*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*throw e*) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exn e'*, *t'*) ⟹ *refines* (*g v*) (*throw e'*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*g v*) (*g' v'*) *t t' R*
shows *refines* (*L2-seq f g*) (*L2-seq f' g'*) *s s' R*
unfolding *L2-seq-def* **using** *f ll lr rl rr*
by (*rule refines-bind-bind-exn*)

lemma *rel-Nonlocal-Nonlocal*: *rel-Nonlocal* (*Nonlocal x*) *x*
by (*simp add*: *rel-Nonlocal-def*)

lemmas *rel-sum-Inl* = *rel-sum.intros*(*1*)
lemmas *rel-sum-Inr* = *rel-sum.intros*(*2*)

lemma *rel-sum-eq*: *rel-sum* (=) (=) *x x*
by (*auto simp add*: *rel-sum.simps*)

definition (**in** *heap-state*)
IO-modify-heap-padding::'*a*::*mem-type ptr* ⇒ ('*s* ⇒ '*a*) ⇒ ('*b*::*default*, *unit*, '*s*) *spec-monad* **where**

IO-modify-heap-padding p v =
state-select {(s, t). ∃ bs. length bs = size-of (TYPE('a)) ∧ t = hmem-upd (heap-update-padding p (v s) bs) s}

lemma (**in** *heap-state*) *liftE-IO-modify-heap-padding*: *liftE* (*IO-modify-heap-padding p v*) = (*IO-modify-heap-padding p v*)
unfolding *IO-modify-heap-padding-def*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

abbreviation (**in** *heap-state*) *IO-modify-heap-paddingE*::
'a::*mem-type ptr* ⇒ (*'s* ⇒ *'a*) ⇒ (*'b*, *unit*, *'s*) *exn-monad* **where**
IO-modify-heap-paddingE p v ≡ *liftE* (*IO-modify-heap-padding p v*)

lemma (**in** *heap-state*) *no-fail-IO-modify-padding*[*simp*]: *succeeds* (*IO-modify-heap-padding p v*) *s*
unfolding *IO-modify-heap-padding-def*
apply (*simp*)
using *length-to-bytes-p* **by** *blast*

lemma (**in** *heap-state*) *no-fail-IO-modify-paddingE*[*simp*]: *succeeds* (*IO-modify-heap-paddingE p v*) *s*
unfolding *IO-modify-heap-padding-def*
apply (*simp*)
using *length-to-bytes-p* **by** *blast*

end

theory *LocalVarExtract*
imports *SimplConv L2Defs*
begin

lemma *Collect-prod-inter*:
{(*s*, *t*). *P s t*} ∩ {(*s*, *t*). *Q s t*} = {(*s*, *t*). *P s t* ∧ *Q s t*}
by (*fastforce intro*: *set-eqI*)

lemma *Collect-prod-union*:
{(*s*, *t*). *P s t*} ∪ {(*s*, *t*). *Q s t*} = {(*s*, *t*). *P s t* ∨ *Q s t*}
by (*fastforce intro*: *set-eqI*)

end

theory *AutoCorresSimpset*

imports *SimplBucket*
begin

lemma *globals-surj*: *surj globals*
  apply (*rule surjI* [**where** *f*=λ*x*. *undefined*(| *globals* := *x* |)])
  apply *simp*
  done

lemma *triv-ex-apply*: ∃ *s1 s0*. *f s0* = *s1*
  by (*iprover*)

lemmas *ptr-val-ptr-add-simps* =
  ptr-add-word32
  ptr-add-word32-signed
  ptr-add-word64
  ptr-add-word64-signed

ML ‹

val AUTOCORRES-SIMPSET =
  @{*context*} *delsimps* (
    (∗ *interferes with heap-lift* ∗)
    @{*thms fun-upd-apply*}
    (∗ *affects boolean expressions* ∗)
    @ @{*thms word-neq-0-conv neq0-conv*}
    (∗ *interferes with struct-rewrite* ∗)
    @ @{*thms ptr-coerce.simps ptr-add-0-id*}
    (∗ *oversimplifies Spec sets prior to L2 stage*
      (*we will control this explicitly in L2Peephole*) ∗)
    @ @{*thms CollectPairFalse*}
    @ @{*thms unsigned-of-nat unsigned-of-int*}
    @ @{*thms map-of-default.simps*}
    @ @{*thms field-lvalue-append*}
    @ @{*thms ptr-val-ptr-add-simps*}
    @ @{*thms distinct-sets.simps*})
  addsimps (
    (∗ *Needed for L2corres-spec* ∗)
    @{*thms globals-surj*}
)
  addsimps (
    (∗ *builds up by monad-equiv tactic, and record updates* ∗)
   @{*thms triv-ex-apply*} (∗ *fixme Shouldn't this already be handled by HOL.ex-simps*
and simproc defined-Ex.∗)
)
  addsimps (@{*thms ptr-NULL-conv*})
  delsimprocs
    [@{*simproc case-prod-beta*},@{*simproc case-prod-eta*}]
  |> *simpset-of*

›

end

theory *ExceptionRewrite*
imports *L1Defs L1Peephole*
begin

definition *always-fail P A* ≡ ∀ *s*. *P s* ⟶ *run A s* = *Failure*

lemma *always-fail-fail* [*simp*]: *always-fail P fail*
  by (*clarsimp simp*: *always-fail-def top-post-state-def*)

lemma *bindE-alwaysfail-lhs*: ⟦ *always-fail* (λ-. *True*) *L* ⟧ ⟹ *always-fail* (λ-. *True*) (*L* >>= *R*)
  apply (*clarsimp simp*: *always-fail-def run-bind*)
  done

lemma *bindE-alwaysfail-rhs*: ⟦ *always-progress L*; *no-throw* (λ-. *True*) *L*; ⋀*x*. *always-fail* (λ-. *True*) (*R x*) ⟧ ⟹ *always-fail* (λ-. *True*) (*L* >>= *R*)
  apply (*clarsimp simp*: *always-fail-def no-throw-def always-progress-def*)
  apply (*clarsimp simp add*: *run-bind runs-to-partial.rep-eq*
      bind-post-state-eq-top[*unfolded top-post-state-def*])
  subgoal premises *prems* **for** *s*
    using *prems*(*1*,*2*)[*rule-format*, *of s*] *prems*(*4*)
    apply (*cases run L s*)
    subgoal by (*auto simp*: *Ball-def split*: *prod.splits exception-or-result-splits*)
    subgoal by *force*
    done
  done

lemma *handleE'-alwaysfail-lhs*: ⟦ *always-fail* (λ-. *True*) *L* ⟧ ⟹ *always-fail* (λ-. *True*) (*L* <*catch*> *R*)
  by (*clarsimp simp*: *always-fail-def run-catch*)

lemma *handleE'-alwaysfail-rhs*: ⟦ *always-progress L*; *no-return* (λ-. *True*) *L*; ⋀*r*. *always-fail* (λ-. *True*) (*R r*) ⟧ ⟹ *always-fail* (λ-. *True*) (*L* <*catch*> *R*)
  apply (*clarsimp simp*: *always-fail-def no-return-def always-progress-def*)
  subgoal premises *prems* **for** *s*
  proof (*cases run L s*)
    case *Failure*
    then show *?thesis* **by** (*auto simp add*: *run-catch*)
  next

case (*Success x2*)
then show *?thesis* **using** *prems*
apply (*clarsimp simp add*: *run-catch runs-to-partial-def-old reaches-def*)
apply (*erule-tac x=s* **in** *allE*)+
apply (*simp add*: *succeeds-def*)
apply (*clarsimp simp add*: *bot-post-state-def Exn-def default-option-def*)
using *image-iff* **by** *fastforce*
qed
done

lemma *alwaysfail-noreturn*: *always-fail P A* $\Longrightarrow$ *no-return P A*
by (*clarsimp simp*: *always-fail-def no-return-def runs-to-partial-def-old succeeds-def*

top-post-state-def)

lemma *alwaysfail-nothrow*: *always-fail P A* $\Longrightarrow$ *no-throw P A*
by (*clarsimp simp*: *always-fail-def no-throw-def runs-to-partial-def-old succeeds-def*

top-post-state-def)

lemma *no-return-Success-distrib-aux1*:
assumes $\forall (a, b) \in X.\ \exists e.\ a = Exception\ e \wedge e \neq default$
shows $X = apfst\ (Exception\ o\ the\text{-}Exception)\ `\ X \wedge (\forall a \in fst\ `\ X.\ the\text{-}Exception\ a \neq default)$
using *assms*
by (*force simp add*: *image-def apfst-def map-prod-def*)

lemma *no-return-Success-distrib-aux2*:
assumes $\forall (a, b) \in X.\ \exists e.\ a = Exception\ e \wedge e \neq default$
shows $P\ (\bigsqcup x \in X.$
case x of $(Exception\ e,\ t) \Rightarrow f\ e\ t$
$|\ (Result\ v,\ t) \Rightarrow g\ v\ t) \longleftrightarrow$
$(P\ (\bigsqcup (e,\ t) \in apfst\ (the\text{-}Exception)\ `\ X.$
$f\ e\ t))$
apply (*subst no-return-Success-distrib-aux1* [*OF assms*])
by (*smt* (*verit*, *ccfv-SIG*) *apfst-conv assms case-exception-or-result-Exception case-prodE case-prod-conv comp-apply image-cong image-image the-Exception-simp*)

lemma *no-return-Success-distrib-aux3*:
assumes $\forall (a, b) \in X.\ \exists e.\ a = Exception\ e \wedge e \neq default$
shows $P\ (\bigsqcup x \in X.$
case x of $(Exception\ e,\ t) \Rightarrow f\ e\ t$
$|\ (Result\ v,\ t) \Rightarrow g\ v\ t) \longleftrightarrow$
$(X = apfst\ (Exception\ o\ the\text{-}Exception)\ `\ X \wedge P\ (\bigsqcup (e,\ t) \in apfst\ (the\text{-}Exception)\ `\ X.$
$f\ e\ t))$
using *no-return-Success-distrib-aux1* [*OF assms*] *no-return-Success-distrib-aux2* [*OF assms*, **where** *P=P*]

apply *simp*
done

lemma *no-return-bindE*:
no-return (λ-. *True*) *A* ⟹ (*A* >>= *B*) = *A*
apply (*rule spec-monad-ext*)
subgoal for *s*
apply (*cases run A s*)
subgoal
by (*auto simp add*: *run-bind*)
subgoal for *x2*
apply (*cases x2* = {})
apply (*force simp*: *run-bind*)
apply (*clarsimp simp add*: *run-bind no-return-def runs-to-partial-def-old reaches-def succeeds-def top-post-state-def bot-post-state-def*)
apply (*erule-tac x*=*s* **in** *allE*)
apply (*clarsimp*, *safe*)
apply (*subst* (*asm*) *no-return-Success-distrib-aux2* [**where** *P* = λ*X*. *X* ≠ *Success x2*])
subgoal
apply *auto*
done
subgoal
apply (*force simp*: *image-image split-beta′ pure-post-state-def Sup-Success*)
done
done
done
done

lemma *L1-skip-always-progress* [*simp*, *L1except*, *always-progress-intros*]:
always-progress L1-skip
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-modify-always-progress* [*simp*, *L1except*, *always-progress-intros*]:
always-progress (*L1-modify m*)
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-guard-always-progress* [*simp*,*L1except*, *always-progress-intros*]:
always-progress (*L1-guard g*)
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-init-always-progress* [*simp*,*L1except*, *always-progress-intros*]:
always-progress (*L1-init v*)

unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-spec-always-progress* [*simp*,*L1except*, *always-progress-intros*]: *always-progress* (*L1-spec s*)
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-throw-always-progress* [*simp*,*L1except*,*always-progress-intros*]: *always-progress L1-throw*
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-fail-always-progress* [*simp*,*L1except*, *always-progress-intros*]: *always-progress L1-fail*
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-condition-always-progress*[*always-progress-intros*]:
⟦ *always-progress L*; *always-progress R* ⟧ ⟹ *always-progress* (*L1-condition C L R*)
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-seq-always-progress*[*always-progress-intros*]:
⟦ *always-progress L*; *always-progress R* ⟧ ⟹ *always-progress* (*L1-seq L R*)
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-catch-always-progress*[*always-progress-intros*]:
⟦ *always-progress L*; *always-progress R* ⟧ ⟹ *always-progress* (*L1-catch L R*)
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-while-always-progress*[*always-progress-intros*]:
always-progress B ⟹ *always-progress* (*L1-while C B*)
unfolding *L1-defs* **by** (*simp add*: *always-progress-intros*)

lemma *L1-call-always-progress*[*always-progress-intros*]: ⟦ *always-progress b* ⟧ ⟹ *always-progress* (*L1-call a b c d e*)
unfolding *L1-call-def* **by** (*simp add*: *always-progress-intros*)

lemma *L1-skip-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) *L1-skip*
by (*clarsimp simp*: *L1-defs no-throw-def*)

lemma *L1-modify-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-modify m*)
by (*clarsimp simp*: *L1-defs no-throw-def*; *runs-to-vcg*)

lemma *L1-guard-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-guard g*)
by (*clarsimp simp*: *L1-defs no-throw-def*; *runs-to-vcg*)

lemma *L1-init-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-init a*)

by (*clarsimp simp*: *L1-defs no-throw-def*; *runs-to-vcg*)

lemma *L1-spec-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-spec a*)
by (*clarsimp simp*: *L1-defs no-throw-def*; *runs-to-vcg*)

lemma *L1-assume-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-assume a*)
by (*clarsimp simp*: *L1-defs no-throw-def*; *runs-to-vcg*)

lemma *L1-fail-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) *L1-fail*
by (*clarsimp simp*: *L1-defs no-throw-def*; *runs-to-vcg*)

lemma *L1-while-nothrow* [*L1except*]: *no-throw* (λ-. *True*) *B* ⟹ *no-throw* (λ-. *True*) (*L1-while C B*)
apply (*clarsimp simp*: *L1-while-def*)
apply (*clarsimp simp*: *no-throw-def*)
apply (*rule runs-to-partial-whileLoop-exn* [**where** *I*=λ*r* -. *case r of Exn e* ⇒ *False* | - ⇒ *True*])
apply *simp*
apply *simp*
apply *simp*
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *L1-catch-nothrow-lhs*: ⟦ *no-throw* (λ-. *True*) *L* ⟧ ⟹ *no-throw* (λ-. *True*) (*L1-catch L R*)
apply (*clarsimp simp*: *L1-defs no-return-def no-throw-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *L1-catch-nothrow-rhs*: ⟦ *no-throw* (λ-. *True*) *R* ⟧ ⟹ *no-throw* (λ-. *True*) (*L1-catch L R*)
apply (*clarsimp simp*: *L1-defs no-return-def no-throw-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *L1-catch-nothrow-both* [*L1except*]: *no-throw* (λ-. *True*) *L* ∨ *no-throw* (λ-. *True*) *R* ⟹ *no-throw* (λ-. *True*) (*L1-catch L R*)
by (*auto simp add*: *L1-catch-nothrow-lhs L1-catch-nothrow-rhs*)

lemma *bind-nothrow-simple*: ⟦ *no-throw* (λ-. *True*) *L*; (⋀*x*. *no-throw* (λ-. *True*) (*R x*)) ⟧ ⟹ *no-throw* (λ-. *True*) (*bind L R*)
apply (*clarsimp simp*: *no-throw-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *L1-seq-nothrow* [*L1except*]: ⟦ *no-throw* (λ-. *True*) *L*; *no-throw* (λ-. *True*) *R* ⟧ ⟹ *no-throw* (λ-. *True*) (*L1-seq L R*)
apply (*clarsimp simp*: *L1-defs no-throw-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *L1-condition-nothrow* [*L1except*]:
⟦ *no-throw* (λ-. *True*) *L*; *no-throw* (λ-. *True*) *R* ⟧ ⟹ *no-throw* (λ-. *True*) (*L1-condition C L R*)
apply (*clarsimp simp*: *L1-defs no-throw-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old*)+
done

lemma *L1-call-nothrow*[*L1except*]: *no-throw* (λ-. *True*) *y* ⟹ *no-throw* (λ-. *True*) (*L1-call x y z q r*)
apply (*clarsimp simp*: *L1-call-def L1-defs no-throw-def*)
apply (*runs-to-vcg*)
by (*auto simp add*: *runs-to-partial-def-old*)
(*fastforce simp add*: *succeeds-bind reaches-bind*)+

lemma *no-throw-state-assumeE*: *no-throw* (λ-. *True*) (*assume-result-and-state p*)
apply (*clarsimp simp*: *L1-defs no-throw-def*)
apply (*runs-to-vcg*)
done

lemma *no-throw-on-exit*:
assumes *c*: *no-throw* (λ-. *True*) *c*
shows *no-throw* (λ-. *True*) (*on-exit c cleanup*)
using *c*
apply (*clarsimp simp*: *L1-defs no-throw-def on-exit′-def on-exit-def*)
apply (*runs-to-vcg*)
apply (*fastforce simp add*: *runs-to-partial-def-old succeeds-bind reaches-bind*)
done

named-theorems *no-throw-with-fresh-stack-ptr*
lemma (**in** *stack-heap-raw-state*) *no-throw-with-fresh-stack-ptr*[*no-throw-with-fresh-stack-ptr*]:
assumes *c*: ⋀*p*. *no-throw* (λ-. *True*) (*c p*)
shows *no-throw* (λ-. *True*) (*with-fresh-stack-ptr n init c*)
apply (*simp add*: *with-fresh-stack-ptr-def*)
apply (*rule bind-nothrow-simple*)
apply (*rule no-throw-state-assumeE*)
apply (*rule no-throw-on-exit*)
apply (*rule c*)
done

lemmas *L1-nothrows* =

L1-seq-nothrow
L1-skip-nothrow
L1-modify-nothrow
L1-condition-nothrow
L1-catch-nothrow-both
L1-while-nothrow
L1-spec-nothrow
L1-assume-nothrow
L1-guard-nothrow
L1-init-nothrow
L1-call-nothrow
L1-fail-nothrow

lemma *L1-throw-noreturn* [*simp*,*L1except*]: *no-return* (λ-. *True*) *L1-throw*
apply (*clarsimp simp*: *L1-defs no-return-exn-def*)
done

lemma *L1-fail-noreturn* [*simp*,*L1except*]: *no-return* (λ-. *True*) *L1-fail*
apply (*clarsimp simp*: *L1-defs no-return-exn-def*)
done

lemma *L1-seq-noreturn-lhs*: *no-return* (λ-. *True*) *L* ⟹ *no-return* (λ-. *True*) (*L1-seq L R*)
apply (*clarsimp simp*: *L1-defs no-return-exn-def*)
apply *runs-to-vcg*
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *L1-seq-noreturn-rhs*: ⟦ *no-return* (λ-. *True*) *R* ⟧ ⟹ *no-return* (λ-. *True*) (*L1-seq L R*)
apply (*clarsimp simp*: *L1-defs no-return-exn-def*)
apply *runs-to-vcg*
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *L1-catch-noreturn*: ⟦ *no-return* (λ-. *True*) *L*; *no-return* (λ-. *True*) *R* ⟧ ⟹ *no-return* (λ-. *True*) (*L1-catch L R*)
apply (*clarsimp simp*: *L1-defs no-return-exn-def*)
apply *runs-to-vcg*
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *L1-condition-noreturn*: ⟦ *no-return* (λ-. *True*) *L*; *no-return* (λ-. *True*) *R* ⟧ ⟹ *no-return* (λ-. *True*) (*L1-condition C L R*)
apply (*clarsimp simp*: *L1-defs no-return-exn-def*)
apply *runs-to-vcg*

apply (*fastforce simp add*: *runs-to-partial-def-old*)+
done

lemma *bindE-noreturn-lhs*: ⟦*no-return* (λ-. *True*) *L* ⟧ ⟹ *no-return* (λ-. *True*) (*L* >>= *R*)
apply (*clarsimp simp*: *L1-defs no-return-def*)
apply *runs-to-vcg*
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *bindE-noreturn-rhs*: ⟦⋀*x*. *no-return* (λ-. *True*) (*R x*) ⟧ ⟹ *no-return* (λ-. *True*) (*L* >>= *R*)
apply (*clarsimp simp*: *L1-defs no-return-def*)
apply *runs-to-vcg*
apply (*fastforce simp add*: *runs-to-partial-def-old*)
done

lemma *on-exit-noreturn*:
assumes *c*: *no-return* (λ-. *True*) *c*
shows *no-return* (λ-. *True*) (*on-exit c cleanup*)
using *c*
unfolding *on-exit-def on-exit'-def*
apply (*clarsimp simp add*: *no-return-def*)
apply *runs-to-vcg*
apply (*fastforce simp add*: *runs-to-partial-def-old succeeds-bind reaches-bind*)
done

named-theorems *no-return-with-fresh-stack-ptr*
lemma (**in** *stack-heap-raw-state*) *no-return-with-fresh-stack-ptr*[*no-return-with-fresh-stack-ptr*]:
assumes *c*: ⋀*p*. *no-return* (λ-. *True*) (*c p*)
shows *no-return* (λ-. *True*) (*with-fresh-stack-ptr n init c*)
apply (*simp add*: *with-fresh-stack-ptr-def*)
apply (*rule bindE-noreturn-rhs*)
apply (*rule on-exit-noreturn*)
apply (*rule c*)
done

lemma *L1-fail-alwaysfail* [*simp*,*L1except*]: *always-fail* (λ-. *True*) *L1-fail*
by (*clarsimp simp*: *L1-defs*)

lemma *L1-seq-alwaysfail-lhs*: ⟦ *always-fail* (λ-. *True*) *L* ⟧ ⟹ *always-fail* (λ-. *True*) (*L1-seq L R*)
by (*auto intro*!: *bindE-alwaysfail-lhs simp*: *L1-defs*)

lemma *L1-seq-alwaysfail-rhs*: ⟦ *always-progress L*; *no-throw* (λ-. *True*) *L*; *always-fail* (λ-. *True*) *R* ⟧ ⟹ *always-fail* (λ-. *True*) (*L1-seq L R*)
by (*auto intro*!: *bindE-alwaysfail-rhs simp*: *L1-defs*)

lemma *L1-catch-alwaysfail-lhs*: ⟦ *always-fail* (λ-. *True*) *L* ⟧ ⟹ *always-fail* (λ-. *True*) (*L1-catch L R*)
by (*clarsimp simp add*: *L1-defs always-fail-def run-catch*)

lemma *L1-catch-alwaysfail-rhs*: ⟦ *always-progress L*; *no-return* (λ-. *True*) *L*; *always-fail* (λ-. *True*) *R* ⟧
⟹ *always-fail* (λ-. *True*) (*L1-catch L R*)
apply (*clarsimp simp add*: *L1-defs always-fail-def always-progress-def no-return-def run-catch*
bind-post-state-eq-top[*unfolded top-post-state-def*] *prod-eq-iff runs-to-partial.rep-eq*
split: *prod.splits xval-splits*)
subgoal premises *prems* **for** *s*
using *prems*(*1*,*2*)[*rule-format*, *of s*] *prems*(*4*)
apply (*cases run L s*)
apply (*auto simp*: *split-beta′ Exn-def default-option-def*)
done
done

lemma *L1-condition-alwaysfail*: ⟦ *always-fail* (λ-. *True*) *L*; *always-fail* (λ-. *True*) *R* ⟧ ⟹ *always-fail* (λ-. *True*) (*L1-condition C L R*)
by (*auto simp add*: *L1-defs always-fail-def run-condition*)

lemma *L1-catch-nothrow* []:
no-throw (λ-. *True*) *A* ⟹ *L1-catch A E* = *A*
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*fastforce simp add*: *no-throw-def runs-to-partial-def-old runs-to-def-old*)+
done

lemma *L1-seq-noreturn* [*L1except*]:
no-return (λ-. *True*) *A* ⟹ *L1-seq A B* = *A*
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*fastforce simp add*: *no-return-def runs-to-partial-def-old runs-to-def-old*)+
done

lemma *L1-catch-throw* [*L1except*]:
L1-catch L1-throw E = *E*
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)

apply (*auto simp add*: *runs-to-iff*)
done

lemma *anything-to-L1-fail* [*L1except*]:
always-fail (λ-. *True*) *A* ⟹ *A* = *L1-fail*
unfolding *L1-defs*
apply (*rule spec-monad-ext*)
apply (*auto simp add*: *always-fail-def top-post-state-def*)
done

lemmas *L1-is-local-simps* [*L1except*] = *is-local-simps*

lemma *L1-catch-return* [*L1except*]: (⋀*s*. *is-local* (*get-exn* (*upd-exn s*))) ⟹
(*L1-seq*
(*L1-modify* (*upd-exn*))
(*L1-condition* (λ*s*. *is-local* (*get-exn s*))
L1-skip L1-throw))
= *L1-modify* (*upd-exn*)
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L1-catch-L1-seq-nothrow* [*L1except*]:
⟦*no-throw* (λ-. *True*) *A* ⟧ ⟹ *L1-catch* (*L1-seq A B*) *C* = *L1-seq A* (*L1-catch B C*)
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*fastforce simp add*: *no-throw-def runs-to-partial-def-old runs-to-def-old*)+
done

lemma *L1-catch-simple-seq* [*L1except*]:
L1-catch (*L1-seq L1-skip B*) *E* = (*L1-seq L1-skip* (*L1-catch B E*))
L1-catch (*L1-seq L1-fail B*) *E* = (*L1-seq L1-fail* (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-modify m*) *B*) *E* = (*L1-seq* (*L1-modify m*) (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-spec s*) *B*) *E* = (*L1-seq* (*L1-spec s*) (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-assume f*) *B*) *E* = (*L1-seq* (*L1-assume f*) (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-guard g*) *B*) *E* = (*L1-seq* (*L1-guard g*) (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-init i*) *B*) *E* = (*L1-seq* (*L1-init i*) (*L1-catch B E*))
apply (*subst L1-catch-seq-join* | *rule L1-nothrows* | *clarsimp simp*: *L1-defs* | *rule bind-nothrow-simple*)+
done

declare *L1-catch-simple-seq*(*6*) [*L1opt*]
lemma *L1-catch-L1-init-seq'*[*L1opt*, *L1except*]: *L1-catch* (*L1-seq* (*L1-seq* (*L1-init* *i*) *A*) *B*) *E* = (*L1-seq* (*L1-init* *i*) (*L1-catch* (*L1-seq* *A* *B*) *E*))
apply (*subst L1-catch-seq-join*)
apply *simp*
apply (*subst L1-seq-assoc*)
apply *simp*
done

lemma *L1-catch-call-simple-seq* [*L1except*]:
no-throw (λ-. *True*) *b* ⟹ *L1-catch* (*L1-seq* (*L1-call* *a* *b* *c* *d* *e*) *B*) *E* = (*L1-seq* (*L1-call* *a* *b* *c* *d* *e*) (*L1-catch* *B* *E*))
apply (*subst L1-catch-seq-join*)
apply (*rule L1-call-nothrow*)
apply (*assumption*)
apply *simp*
done

lemma *L1-catch-single* [*L1except*]:
L1-catch (*L1-skip*) *E* = *L1-skip*
L1-catch (*L1-fail*) *E* = *L1-fail*
L1-catch (*L1-modify* *m*) *E* = *L1-modify* *m*
L1-catch (*L1-spec* *s*) *E* = *L1-spec* *s*
L1-catch (*L1-assume* *f*) *E* = *L1-assume* *f*
L1-catch (*L1-guard* *g*) *E* = *L1-guard* *g*
L1-catch (*L1-init* *i*) *E* = *L1-init* *i*
apply (*subst L1-catch-nothrow* | *rule L1-nothrows* | *clarsimp simp*: *L1-defs* | *rule bind-nothrow-simple*)+
done

lemma *L1-catch-call-single* [*L1except*]: *no-throw* (λ-. *True*) *b* ⟹ *L1-catch* (*L1-call* *a* *b* *c* *d* *e*) *E* = *L1-call* *a* *b* *c* *d* *e*
apply (*subst L1-catch-nothrow*)
apply (*rule L1-call-nothrow*)
apply *assumption*
apply *simp*
done

lemma *L1-catch-single-while* [*L1except*]:
⟦ *no-throw* (λ-. *True*) *B* ⟧ ⟹ *L1-catch* (*L1-while* *C* *B*) *E* = *L1-while* *C* *B*
apply (*rule L1-catch-nothrow*)
apply (*rule L1-while-nothrow*)
apply *assumption*
done

lemma *L1-catch-seq-while* [*L1except*]:
⟦ *no-throw* (λ-. *True*) *B* ⟧ ⟹ *L1-catch* (*L1-seq* (*L1-while* *C* *B*) *X*) *E* = *L1-seq*

(*L1-while C B*) (*L1-catch X E*)
apply (*rule L1-catch-seq-join* [*symmetric*])
apply (*rule L1-while-nothrow*)
apply *assumption*
done

lemma *L1-catch-single-cond* [*L1except*]:
L1-catch (*L1-condition C L R*) *E* = *L1-condition C* (*L1-catch L E*) (*L1-catch R E*)
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L1-catch-cond-seq*:
L1-catch (*L1-seq* (*L1-condition C L R*) *B*) *E* = *L1-condition C* (*L1-catch* (*L1-seq L B*) *E*) (*L1-catch* (*L1-seq R B*) *E*)
apply (*subst L1-condition-distrib*)
apply (*rule L1-catch-single-cond*)
done

lemma *L1-catch-seq-cond-noreturn-ex*:
⟦ *no-return* (λ-. *True*) *E* ⟧ ⟹ (*L1-catch* (*L1-seq* (*L1-condition c A B*) *C*) *E*) = (*L1-seq* (*L1-catch* (*L1-condition c A B*) *E*) (*L1-catch C E*))
unfolding *L1-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
by (*auto simp add*: *no-return-def runs-to-partial-def-old runs-to-def-old Exn-def*)
(*metis Exception-eq-Result reaches-succeeds default-option-def*)+

lemmas *L1-catch-seq-cond-nothrow* = *L1-catch-L1-seq-nothrow* [*OF L1-condition-nothrow*]

end

19.2 Nested Exceptions

theory *L2ExceptionRewrite*
imports
L2Defs
ExceptionRewrite
begin

synthesize-rules *L2-rel-spec-monad*

19.3 Transformations from single level exceptions to nested exceptions

lemma *catch-yield-map-value-conv*: (*f* <*catch*> (λ*e*. *yield* (*g e*))) =
(*map-value* (λ*x*. *case x of Exn e* ⇒ *g e* | *Result v* ⇒ *Result v*) *f*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff elim*!: *runs-to-weaken split*: *xval-splits*)
done

lemma *rel-spec-monad-rel-xval-try-catch*:
assumes *bdy*: *rel-spec-monad Q* (*rel-xval* (λ*e e'*. (*rel-sum L R*) (*from-xval* (*f e*)) *e'*) *R*) *B B'*
shows *rel-spec-monad Q* (*rel-xval L R*) (*L2-catch B* (λ*e*. *yield* (*f e*))) (*L2-try B'*)
apply (*clarsimp simp add*: *rel-spec-monad-iff-refines*)
apply (*intro conjI*)
subgoal for *s t*
using *bdy*
apply (*clarsimp simp add*: *rel-spec-monad-iff-refines*)
apply (*erule-tac x*=*s* **in** *allE*)
apply (*erule-tac x*=*t* **in** *allE*)
apply *safe*
unfolding *L2-defs try-def catch-yield-map-value-conv*
apply (*erule refines-map-value*)
apply (*auto simp add*: *rel-xval.simps rel-sum-simps unnest-exn-def rel-sum.simps*)
done
subgoal for *s t*
using *bdy*
apply (*clarsimp simp add*: *rel-spec-monad-iff-refines*)
apply (*erule-tac x*=*s* **in** *allE*)
apply (*erule-tac x*=*t* **in** *allE*)
apply *safe*
unfolding *L2-defs try-def catch-yield-map-value-conv*
apply (*erule refines-map-value*)
apply (*auto simp add*: *rel-xval.simps rel-sum-simps unnest-exn-def rel-sum.simps*)
done
done

lemma *rel-spec-monad-L2-seq-rel-xval-result-eq*:
assumes *f-f'*: *rel-spec-monad S* (*rel-xval L* (=)) *f f'*
assumes *Res-Res*: ⋀*v*. *rel-spec-monad S* (*rel-xval L* (=)) (*g v*) (*g' v*)
shows *rel-spec-monad S* (*rel-xval L* (=)) (*L2-seq f g*) (*L2-seq f' g'*)
unfolding *L2-defs* **using** *assms*
by (*rule rel-spec-monad-rel-xval-result-eq-bind*)

lemma *rel-spec-monad-rel-xval-L2-unknown*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-unknown ns*) (*L2-unknown ns*)
unfolding *L2-unknown-def*
by (*auto simp add*: *rel-spec-monad-def rel-set-def*)

lemma *rel-spec-monad-rel-xval-L2-modify*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-modify f*) (*L2-modify f*)
unfolding *L2-modify-def*
by (*auto simp add*: *rel-spec-monad-def rel-set-def*)

lemma *rel-spec-monad-rel-xval-L2-gets*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-gets f ns*) (*L2-gets f ns*)
unfolding *L2-gets-def*
by (*auto simp add*: *rel-spec-monad-def rel-set-def*)

lemma *rel-spec-monad-eq-rel-xval-L2-condition*:
assumes *f*: *rel-spec-monad* (=) (*rel-xval L* (=)) *f f'*
and *g*: *rel-spec-monad* (=) (*rel-xval L* (=)) *g g'*
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-condition P f g*) (*L2-condition P f' g'*)
unfolding *L2-condition-def*
apply (*rule rel-spec-monad-condition* [*OF - f g*])
apply (*auto simp add*: *rel-fun-def*)
done

lemma *rel-spec-monad-L2-throw-sanitize-names*:
assumes *xy*: *R* (*Exn x*) (*Exn y*)
assumes *ns'*: *SANITIZE-NAMES y ns ns'*
shows *rel-spec-monad* (=) *R* (*L2-throw x ns*) (*L2-throw y ns'*)
unfolding *L2-throw-def*
using *xy*
by (*auto simp add*: *rel-spec-monad-def rel-set-def*)

lemma *rel-spec-monad-rel-xval-L2-spec*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-spec r*) (*L2-spec r*)
unfolding *L2-spec-def*
apply (*rule rel-spec-monad-rel-xval-bind* [**where** *P*=(=)])
subgoal by (*auto simp add*: *rel-spec-monad-def rel-set-def run-assert-result-and-state*)
subgoal by (*auto simp add*: *rel-spec-monad-def rel-set-def*)
done

lemma *rel-spec-monad-rel-xval-L2-assume*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-assume r*) (*L2-assume r*)
unfolding *L2-assume-def*
by (*auto simp add*: *rel-spec-monad-def rel-set-def split*: *prod.splits*)

lemma *rel-spec-monad-rel-xval-L2-guard*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-guard c*) (*L2-guard c*)
unfolding *L2-guard-def*
by (*auto simp add*: *rel-spec-monad-def rel-set-def run-guard*)

lemma *rel-spec-monad-rel-xval-L2-guarded*:
assumes *c-c'*: *rel-spec-monad* (=) (*rel-xval L* (=)) *c c'*
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-guarded g c*) (*L2-guarded g c'*)

unfolding *L2-guarded-def*
apply (*rule rel-spec-monad-L2-seq-rel-xval-result-eq*)
apply (*rule rel-spec-monad-rel-xval-L2-guard*)
apply (*rule c-c'*)
done

lemma *rel-spec-monad-L2-fail*:
rel-spec-monad Q R (*L2-fail*) (*L2-fail*)
unfolding *L2-fail-def*
by (*rule rel-spec-monad-fail*)

lemma *rel-spec-monad-rel-xval-L2-call*:
assumes *emb*: ⋀*e*. *L* (*emb e*) (*emb' e*)
assumes *ns'*: *SANITIZE-NAMES emb' ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-call f emb ns*) (*L2-call f emb' ns'*)
unfolding *L2-call-def*
using *emb*
by (*auto simp*: *rel-spec-monad-iff-rel-spec rel-spec-map-value-right-iff*
rel-spec-map-value-left-iff map-exn-def
split: *xval-split*
intro!: *rel-spec-refl*)

lemma *rel-fun-eq-refl*: *rel-fun* (=) (=) *f f*
by (*rule rel-funI*) *simp*

lemma *rel-spec-monad-result-exec-concrete*:
assumes *m-m'*: *rel-spec-monad* (=) *R m m'*
shows *rel-spec-monad* (=) *R* (*exec-concrete st m*) (*exec-concrete st m'*)
apply (*clarsimp simp add*: *rel-spec-monad-iff-refines*, *intro conjI*)
subgoal for *s*
using *m-m'*
by (*fastforce simp add*: *rel-spec-monad-iff-refines refines-def-old succeeds-exec-concrete-iff reaches-exec-concrete*)
subgoal for *s*
using *m-m'*
by (*fastforce simp add*: *rel-spec-monad-iff-refines refines-def-old succeeds-exec-concrete-iff reaches-exec-concrete*)
done

lemma *rel-spec-monad-result-exec-abstract*:
assumes *m-m'*: *rel-spec-monad* (=) *R m m'*
shows *rel-spec-monad* (=) *R* (*exec-abstract st m*) (*exec-abstract st m'*)
apply (*clarsimp simp add*: *rel-spec-monad-iff-refines*, *intro conjI*)
subgoal for *s*
using *m-m'*
by (*auto simp add*: *rel-spec-monad-iff-refines refines-def-old succeeds-exec-abstract-iff reaches-exec-abstract*)
subgoal for *s*

using *m-m'*
by (*auto simp add*: *rel-spec-monad-iff-refines refines-def-old succeeds-exec-abstract-iff reaches-exec-abstract*)
done

lemma *rel-spec-monad-rel-project-L2-unknown'*:
assumes *surj-prj*: *surj prj*
assumes *names*: *SANITIZE-NAMES prj ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-unknown ns*) (*L2-unknown ns'*)
unfolding *L2-unknown-def*
using *surj-prj*
by (*auto simp add*: *rel-spec-monad-def rel-set-def rel-project-def*)

Note that *prj* is the identity function on unit here

lemma *rel-spec-monad-rel-project-L2-modify*:
rel-spec-monad (=) (*rel-xval L* (*rel-project prj*)) (*L2-modify f*) (*L2-modify f*)
unfolding *L2-modify-def*
by (*auto simp add*: *rel-spec-monad-def rel-set-def rel-project-def*)

lemma *rel-spec-monad-rel-project-L2-gets'*:
assumes *g*: $\bigwedge x.\ g\ x = prj\ (f\ x)$
assumes *names*: *SANITIZE-NAMES prj ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-gets f ns*) (*L2-gets g ns'*)
unfolding *L2-gets-def g*
by (*auto simp add*: *rel-spec-monad-def rel-set-def rel-project-def*)

lemma *rel-spec-monad-rel-project-L2-condition*:
assumes *f*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) *f f'*
assumes *g*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) *g g'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-condition c f g*) (*L2-condition c f' g'*)
unfolding *L2-condition-def*
apply (*rule rel-spec-monad-condition* [*OF - f g*])
apply (*auto simp add*: *rel-fun-def*)
done

lemma *rel-spec-monad-rel-project-L2-throw*:
assumes *x-xs*: *L x y*
assumes *names*: *SANITIZE-NAMES* (*L*, *x*) *ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-throw x ns*) (*L2-throw y ns'*)
unfolding *L2-throw-def* **using** *x-xs*
by (*auto simp add*: *rel-spec-monad-def rel-set-def rel-project-def*)

lemma *rel-spec-monad-rel-project-L2-spec*:
assumes *prj-surj*: *surj prj*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-spec r*) (*L2-spec r*)

unfolding *L2-spec-def*
apply (*rule rel-spec-monad-rel-xval-bind* [**where** *P*=(=)])
subgoal by (*auto simp add*: *rel-spec-monad-def rel-set-def run-assert-result-and-state*)
subgoal using *prj-surj* **by** (*auto simp add*: *rel-spec-monad-def rel-set-def rel-project-def*)
done

lemma *rel-spec-monad-rel-project-L2-assume*:
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-assume r*) (*L2-assume r*)
unfolding *L2-assume-def*
by (*auto simp add*: *rel-spec-monad-def rel-set-def split*: *prod.splits*)

lemma *rel-spec-monad-rel-project-L2-guard*:
rel-spec-monad (=) (*rel-xval L* (*rel-project prj*)) (*L2-guard c*) (*L2-guard c*)
unfolding *L2-guard-def*
by (*auto simp add*: *rel-spec-monad-def rel-set-def run-guard rel-project-def*)

lemma *rel-spec-monad-rel-project-L2-seq*:
assumes *m-n*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj'*)) *m n*
assumes *f-g*: $\bigwedge x$. *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*f x*) (*g* (*prj' x*))
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-seq m f*) (*L2-seq n g*)
unfolding *L2-seq-def*
apply (*rule rel-spec-monad-rel-xval-bind* [*OF m-n*])
using *f-g*
by (*simp add*: *rel-project-conv*)

lemma *rel-spec-monad-rel-project-L2-guarded*:
assumes *c-c'*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) *c c'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-guarded g c*) (*L2-guarded g c'*)
unfolding *L2-guarded-def L2-seq-def*
apply (*rule rel-spec-monad-rel-xval-bind* [**where** *P*=(=)])
apply (*rule rel-spec-monad-rel-xval-L2-guard*)
apply (*rule c-c'*)
done

lemma *rel-spec-monad-rel-project-L2-try*:
assumes *fg*: *rel-spec-monad* (=) (*rel-xval* (*rel-sum L* (*rel-project prj*)) (*rel-project prj*)) *f g*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-try f*) (*L2-try g*)
unfolding *L2-try-def*
apply (*clarsimp simp add*: *rel-spec-monad-iff-refines*, *intro conjI*)
subgoal for *s*
using *fg*
apply (*clarsimp simp add*: *rel-spec-monad-iff-refines refines-def-old succeeds-try reaches-try*)
apply (*erule-tac x*=*s* **in** *allE*)
apply *clarsimp*

```
    subgoal for s' r'
      apply (erule-tac x=r' in allE)
      apply (erule-tac x=s' in allE)
      apply clarsimp
      subgoal for x
        apply (cases x)
       subgoal apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
          by (smt (verit, best) Exn-neq-Result rel-sum.simps rel-xval.simps
              unnest-exn-simps(1) unnest-exn-simps(2))
        subgoal
          apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
          by (smt (verit, best) Result-neq-Exn rel-xval.simps unnest-exn-simps(3))
        done
      done
    done
  subgoal for s
    using fg
   apply (clarsimp simp add: rel-spec-monad-iff-refines refines-def-old succeeds-try
reaches-try)
    apply (erule-tac x=s in allE)
    apply clarsimp
    subgoal for s' r'
      apply (erule-tac x=r' in allE)
      apply (erule-tac x=s' in allE)
      apply clarsimp
      subgoal for x
        apply (cases x)
       subgoal apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
          by (smt (verit, best) Exn-neq-Result rel-sum.simps rel-xval.simps
              unnest-exn-simps(1) unnest-exn-simps(2))
        subgoal
          apply (clarsimp simp add: Exn-def [symmetric] default-option-def)
          by (smt (verit, best) Result-neq-Exn rel-xval.simps unnest-exn-simps(3))
        done
      done
    done
  done
```

Tailored projection for local handler functions that may emerge in IO phase to handle exit case. These handlers are composed from liftE functions followed by rethrowing the exception. We do not attempt to optimise projections for the handlers.

```
lemma rel-spec-monad-rel-project-L2-catch:
  assumes f: rel-spec-monad (=) (rel-xval (=) (rel-project prj)) f f'
  assumes g: ⋀v. (rel-spec-monad (=) (rel-xval L (λ- -. False))) (g v) (g v)
  shows rel-spec-monad (=) (rel-xval L (rel-project prj)) (L2-catch f g) (L2-catch
f' g)
  unfolding L2-catch-def
  apply (rule rel-spec-monad-rel-xval-catch [OF f])
```

```
  using g using rel-fun-def rel-spec-monad-mono
  by (smt (verit, del-insts) Exn rel-xval.cases)


lemma rel-spec-monad-rel-project-liftE:
  assumes f: rel-spec-monad (=) (rel-map the-Res OO rel-project prj OO rel-map
Result) f f'
  shows rel-spec-monad (=) (rel-xval L (rel-project prj)) (liftE f) (liftE f')
  using assms
  apply (clarsimp simp add: rel-spec-monad-iff-refines, intro conjI)
  subgoal for s
    using f
   apply (clarsimp simp add: rel-spec-monad-iff-refines refines-def-old  reaches-liftE
)
    apply (erule-tac x=s in allE)
    apply clarsimp
    subgoal for s' r'
      apply (erule-tac x=r' in allE)
      apply (erule-tac x=s' in allE)
      by (metis rel-map-def rel-xval.Result relcomppE the-Result-simp)
    done
  subgoal for s
    using f
   apply (clarsimp simp add: rel-spec-monad-iff-refines refines-def-old  reaches-liftE
)
    apply (erule-tac x=s in allE)
    apply clarsimp
    subgoal for s' r'
      apply (erule-tac x=r' in allE)
      apply (erule-tac x=s' in allE)
      by (metis rel-map-def rel-xval.Result relcomppE the-Result-simp)
    done
  done

lemma rel-project-Res-conv:
  (rel-map the-Res OO rel-project prj OO rel-map Result) = (rel-project (Result o
prj o the-Res))
  apply (intro ext)
  apply (simp add: rel-map-def rel-project-def relcompp.simps)
  done


lemma rel-spec-monad-rel-project-id:
  shows rel-spec-monad (=) (rel-project (λv. v)) f f
  by (simp add: rel-project-id(2) rel-spec-monad-eq-conv)


lemma rel-spec-monad-rel-project-unit:
  fixes f:: (unit, 's) res-monad
```

shows *rel-spec-monad* (=) (*rel-project* (λ*v*. *Result* ())) *f f*
using *rel-spec-monad-rel-project-id*
by (*metis Result-unit-eq rel-projectI rel-spec-monad-mono*)

lemma *rel-spec-monad-rel-project-liftE-unit-id*:
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project* (*prj*::*unit* ⇒ *unit*))) (*liftE f*) (*liftE f*)
apply (*rule rel-spec-monad-rel-project-liftE*)
apply (*simp add*: *rel-project-Res-conv comp-def rel-spec-monad-rel-project-unit*)
done

lemma *rel-project-unit-eq*: (*rel-project* (*prj*::*unit* ⇒ *unit*)) = (=)
apply (*rule ext*)+
apply (*auto simp add*: *rel-project-def*)
done

lemma *rel-spec-monad-L2-seq-rel-xval-same-split*:
assumes *mn*: *rel-spec-monad R* (*rel-xval L* (=)) *m n*
and *fg*: ⋀*x*. (*rel-spec-monad R* (*rel-xval L* (=))) (*f x*) (*g x*)
shows *rel-spec-monad R* (*rel-xval L* (=)) (*L2-seq m f*) (*L2-seq n g*)
using *assms*
by (*rule rel-spec-monad-L2-seq-rel-xval-result-eq*)

lemma *rel-spec-monad-L2-while-rel-xval-same-split*:
assumes *B*: ⋀*x*. (*rel-spec-monad* (=) (*rel-xval L* (=))) (*B x*) (*B′ x*)
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-while C′ B I′ ns*) (*L2-while C′ B′ I′ ns*)
unfolding *L2-while-def*
apply (*rule rel-spec-monad-whileLoop-exn*)
apply (*auto simp add*: *B*)
done

lemma *rel-spec-monad-rel-project-L2-call-adapt-emb*:
assumes *L*: ⋀*x*. *L* (*emb x*) (*emb′ x*)
assumes *prj*: ⋀*v*. *prj v* = *v*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-call x emb ns*) (*L2-call x emb′ ns*)
unfolding *L2-call-def*
using *prj L*
apply (*auto simp*: *map-exn-def rel-project-def*
rel-spec-monad-iff-rel-spec rel-spec-map-value-left-iff rel-spec-map-value-right-iff

split: *xval-splits*
intro!: *rel-spec-refl*)
done

lemma *rel-spec-monad-liftE-id*: *rel-spec-monad* (=) (*rel-xval* (λ- -. *False*) (=)) (*liftE f*) (*liftE f*)

by (*auto simp add*: *rel-spec-monad-iff-refines refines-def-old reaches-liftE rel-xval.simps*)

lemma *rel-spec-monad-L2-seq-exit-handler*:
assumes $\bigwedge v$. *rel-spec-monad* (=) (*rel-xval L* (λ- -. *False*)) (*g v*) (*g′ v*)
shows *rel-spec-monad* (=) (*rel-xval L* (λ- -. *False*)) (*L2-seq* (*liftE f*) *g*) (*L2-seq* (*liftE f*) *g′*)
unfolding *L2-defs*
apply (*rule rel-spec-monad-bind-strong-exn* [*OF rel-spec-monad-liftE-id*])
apply (*auto simp add*: *assms*)
done

lemma *rel-spec-monad-rel-project-L2-while′*:
assumes *C-C′*: $\bigwedge v\ s$. *C v s* = *C′* (*prj v*) *s*
assumes *I′*: *I′* = *prj I*
assumes *names*: *SANITIZE-NAMES prj ns ns′*
assumes *B-B′*: $\bigwedge v$. *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*B v*) (*B′* (*prj v*))
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-while C B I ns*) (*L2-while C′ B′ I′ ns′*)
unfolding *L2-while-def*
apply (*rule rel-spec-monad-whileLoop-exn*)
subgoal using *I′* **by** (*auto simp add*: *rel-project-def*)
subgoal using *C-C′* **by** (*auto simp add*: *rel-fun-def rel-project-def*)
subgoal using *B-B′* **by** (*auto simp add*: *rel-project-def*)
done

lemma *rel-spec-monad-rel-xval-on-exit*:
assumes *c-c′*: *rel-spec-monad* (=) (*rel-xval L* (=)) *c c′*
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*on-exit c cleanup*) (*on-exit c′ cleanup*)
unfolding *on-exit-bind-exception-or-result-conv* **thm** *rel-spec-monad-bind-exception-or-result-strong*
apply (*rule rel-spec-monad-bind-exception-or-result-strong* [*OF c-c′*])
apply (*rule rel-spec-monad-bind-strong-exn* [**where** *P*=*rel-xval* (λ- -. *False*) (=)])
subgoal
apply (*subst* (*1 2*) *liftE-state-select* [*symmetric*])
apply (*rule rel-spec-monad-liftE-id*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp add*: *rel-spec-monad-yield*)
done

lemma (**in** *stack-heap-raw-state*) *rel-spec-monad-rel-xval-with-fresh-stack-ptr*:
assumes *c*: $\bigwedge p$. *rel-spec-monad* (=) (*rel-xval L* (=)) (*c p*) (*c′ p*)
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*with-fresh-stack-ptr n init* (*L2-VARS c nm*)) (*with-fresh-stack-ptr n init* (*L2-VARS c′ nm*))
unfolding *with-fresh-stack-ptr-def L2-VARS-def*
apply (*rule rel-spec-monad-bind-strong-exn* [**where** *P*=*rel-xval* (λ- -. *False*) (=)])

subgoal
apply (*subst* (*1 2*) *liftE-assume-result-and-state* [*symmetric*])
apply (*rule rel-spec-monad-liftE-id*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*simp add*: *rel-spec-monad-rel-xval-on-exit c*)
done

lemma *rel-spec-monad-rel-project-on-exit*:
assumes *c-c'*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) *c c'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*on-exit c cleanup*) (*on-exit c' cleanup*)
unfolding *on-exit-bind-exception-or-result-conv* **unfolding** *on-exit-def*
apply (*rule rel-spec-monad-bind-exception-or-result-strong* [*OF c-c'*])
apply (*rule rel-spec-monad-bind-strong-exn* [**where** *P*=*rel-xval* (λ- -. *False*) (=)])
subgoal
apply (*subst* (*1 2*) *liftE-state-select* [*symmetric*])
apply (*rule rel-spec-monad-liftE-id*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*auto simp add*: *rel-spec-monad-yield*)
done

lemma (**in** *stack-heap-raw-state*) *rel-spec-monad-rel-project-with-fresh-stack-ptr*:
assumes *c*: ⋀*p*. *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*c p*) (*c' p*)
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*with-fresh-stack-ptr n init* (*L2-VARS c nm*)) (*with-fresh-stack-ptr n init* (*L2-VARS c' nm*))
unfolding *with-fresh-stack-ptr-def L2-VARS-def*
apply (*rule rel-spec-monad-bind-strong-exn* [**where** *P*=*rel-xval* (λ- -. *False*) (=)])
subgoal
apply (*subst* (*1 2*) *liftE-assume-result-and-state* [*symmetric*])
apply (*rule rel-spec-monad-liftE-id*)
done
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*simp add*: *rel-spec-monad-rel-project-on-exit c*)
done

lemma *rel-spec-monad-L2-VARS*:
assumes *f-f'*: *rel-spec-monad P Q f f'*
shows *rel-spec-monad P Q* (*L2-VARS f ns*) (*L2-VARS f' ns*)
unfolding *L2-VARS-def*
by (*rule f-f'*)

lemma *cond-return1*: (λa.
L2-condition (λs. P a) (L2-gets (λs. f a) ns)
(L2-throw (g a) xs)) =
(λa. yield (if P a then Result (f a) else (Exn (g a))))
unfolding *L2-defs*
apply (*rule ext*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *cond-return2*: (λ(a, b).
L2-condition (λs. P a b) (L2-gets (λs. f a b) ns)
(L2-throw (g a b) xs)) =
(λ(a, b). yield (if P a b then Result (f a b) else (Exn (g a b))))
unfolding *L2-defs*
apply (*rule ext*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *rel-spec-monad-rel-xvalI*:
rel-spec-monad R (*rel-xval* (=) (=)) *f g* ⟹ *rel-spec-monad R* (=) *f g*
by (*simp add*: *rel-xval-eq*)

lemma *is-local-split*: ((*is-local a* ⟶ *e* = *Result b*) ∧ (¬ *is-local a* ⟶ *e* = *Exn* (*the-Nonlocal a*)))
=
(*case a of Nonlocal x* ⇒ *e* = *Exn x* | - ⇒ *e* = *Result b*)
by (*cases a*) *auto*

lemma *is-local-splitI* :
(*case a of Nonlocal x* ⇒ *e* = *Exn x* | - ⇒ *e* = *Result b*) ⟹
((*is-local a* ⟶ *e* = *Result b*) ∧ (¬ *is-local a* ⟶ *e* = *Exn* (*the-Nonlocal a*)))
apply (*simp add*: *is-local-split*)
done

lemma *if-is-local-cases*: (*if is-local e then f else g*) = (*case e of Nonlocal x* ⇒ *g* | - ⇒ *f*)
by (*cases e*) *auto*

lemma *if-Break-cases*: (*if e* = *Break then f else g*) = (*case e of Break* ⇒ *f* | - ⇒ *g*)
by (*cases e*) *auto*

lemma *if-Continue-cases*: (*if e = Continue then f else g*) = (*case e of Continue ⇒ f | - ⇒ g*)
by (*cases e*) *auto*

lemma *if-Return-cases*: (*if e = Return then f else g*) = (*case e of Return ⇒ f | - ⇒ g*)
by (*cases e*) *auto*

lemmas *if-c-exntype-cases* =
if-is-local-cases if-Break-cases if-Continue-cases if-Return-cases

lemmas *case-sum-c-exntype-swap* = *c-exntype.case-distrib*

lemma *ex-c-enxtype-cases-distrib*: (∃ *a*. *P a* ∧
(*case e of Break ⇒ brk a | Continue ⇒ cnt a | Return ⇒ ret a | Goto l ⇒ goto l a*
| *Nonlocal gx ⇒ nonlocal gx a*)) =
(*case e of Break ⇒* ∃ *a*. *P a* ∧ *brk a | Continue ⇒* ∃ *a*. *P a* ∧ *cnt a | Return ⇒* ∃ *a*. *P a* ∧ *ret a*
| *Goto l ⇒* ∃ *a*. *P a* ∧ *goto l a*

| *Nonlocal gx ⇒* ∃ *a*. *P a* ∧ *nonlocal gx a*)

by (*auto split*: *c-exntype.splits*)

19.4 Transformations for procedure local exceptions

Introduce constant for relations to avoid higher order patterns in term net for rules

19.4.1 Transformations for *try* and *finally*

19.5 Transformations on global exceptions

definition
lift-exit-status :: ('*e*, '*a*, '*s*) *exn-monad* ⇒ ('*e c-exntype*, '*a*, '*s*) *exn-monad*
where
lift-exit-status f ≡ *map-value* (*map-exn Nonlocal*) *f*

19.5.1 Removing unused tuple components

lemma *rel-project-eqI*: *rel-spec-monad* (=) (*rel-xval* (=) (*rel-project* (*ETA-TUPLED* (λ*v*. *v*)))) *f g* ⟹ *f* = *g*
by (*simp add*: *rel-project-id rel-xval-eq rel-spec-monad-eq-conv ETA-TUPLED-def*)

19.5.2 Setup basic rules

lemma *rel-xval-case-Nonlocal-sameI*:
assumes *L*: $\bigwedge v.\ e = Nonlocal\ v \Longrightarrow L\ (Nonlocal\ v)\ (Nonlocal\ v)$
assumes *R*: *is-local e* $\Longrightarrow$ *R v v'*
shows *rel-xval L R* (*case e of Nonlocal x* $\Rightarrow$ *Exn* (*Nonlocal x*) | - $\Rightarrow$ *Result v*)
(*case e of Nonlocal x* $\Rightarrow$ *Exn* (*Nonlocal x*) | - $\Rightarrow$ *Result v'*)
using *L R*
by (*auto split*: *c-exntype.splits*)

lemma *rel-sum-map-sum-InlI*: *L* (*l x*) *v* $\Longrightarrow$ *rel-sum L R* (*map-sum l r* (*Inl x*)) (*Inl v*)
by (*auto*)

lemma *rel-sum-map-sum-InrI*: *R* (*r x*) *v* $\Longrightarrow$ *rel-sum L R* (*map-sum l r* (*Inr x*)) (*Inr v*)
by (*auto*)

lemma *rel-map-xval-xval-ExnI*: *L* (*l x*) *v* $\Longrightarrow$ *rel-xval L R* (*map-xval l r* (*Exn x*)) (*Exn v*)
by (*auto*)

lemma *rel-map-xval-xval-ResultI*: *R* (*r x*) *v* $\Longrightarrow$ *rel-xval L R* (*map-xval l r* (*Result x*)) (*Result v*)
by (*auto*)

ML ‹
structure Rel-Spec-Monad-Synthesize-Rules =
struct

fun gen-number-bounds len [] = []
| *gen-number-bounds len* (*x*::*xs*) = ((*len*, []), *x*):: *gen-number-bounds* (*len* − *1*) *xs*;

fun number-bounds xs = gen-number-bounds (*length xs* − *1*) *xs*

exception Incompatible-Use
(∗ *Note that we only perform inner procedural analysis. Hence an global exception* (*i.e.*
∗ *all constructors are Inl and there is no final Inr*) *are irrelevant for the analysis* ∗)
datatype result = Inl of result | *Inr of int* (∗ *position in tuple, from left to right, starting with 1* ∗)
datatype uses = Unused | *Used* | *Propagated of result list*

local
fun inc-bound n (*i*, *aliases*) = (*i* + *n*, *map* (*fn i => i* + *n*) *aliases*)
fun inc-bounds n = map (*fn* (*b*, *x*) => (*inc-bound n b*, *x*))
fun add-alias i (*b*, *aliases*) = (*b*, *i*::*aliases*)

```
fun gen-number-pos n [] = []
  | gen-number-pos n (x::xs) = (n, x)::gen-number-pos (n + 1) xs
val number-pos = gen-number-pos 1

fun eq-bound (b, aliases) i = member (op =) (b::aliases) i
fun same-bound (b1, aliases1) (b2, aliases2) = not (null (inter (op =) (b1::aliases1)
(b2::aliases2)))

fun propagation (Propagated -) = true
  | propagation - = false

fun merge-result (Inr x) (Inr y) = if x = y then [Inr x] else [Inr x, Inr y]
  | merge-result (Inl x) (Inl y) = map Inl (merge-result x y)
  | merge-result x y = [x, y]

fun pop-result (Inl x) = [x]
  | pop-result - = []

fun merge-uses Unused x = x
  | merge-uses x Unused = x
  | merge-uses Used x = Used
  | merge-uses x Used = Used
  | merge-uses (Propagated rs) (Propagated rs') =
     Propagated (fold (fn res => fn rsx => flat (map (merge-result res) rsx)) rs'
rs)

fun returned (Propagated rs) = get-first (fn Inr x => SOME x | - => NONE) rs
  | returned - = NONE

fun remove-return (Propagated rs) =
     let val rs' = filter-out (fn Inr x => true  | - => false) rs
     in if null rs' then Unused else Propagated rs' end
  | remove-return x = x

fun return-to-used (Propagated rs) =
     if exists (fn Inr x => true  | - => false) rs then Used else Propagated rs
  | return-to-used x = x

fun handled (Propagated rs) = Propagated (flat (map pop-result rs))
  | handled x = x

fun analyse-result bounds constr pos propagate t =
  case t of
    (Bound i) => map (fn (b, -) =>
       if eq-bound b i
       then if propagate then Propagated [constr pos] else Used
       else Unused) bounds
  | (t1 $ t2) =>
      let
```

```
      val rs1 = analyse-result bounds constr pos false t1
      val rs2 = analyse-result bounds constr pos false t2
    in map2 merge-uses rs1 rs2 end
  | - => map (K Unused) bounds

fun analyse-used bounds = analyse-result bounds Inr 0 false

fun merge-results uss = foldl1 (uncurry (map2 merge-uses)) uss

fun merge-results-default d uss = if null uss then d else merge-results uss

fun propagate-returns off = map2 (fn (b, x) => fn r =>
   (case returned r of SOME p => inc-bound off b |> add-alias (off - p) | - =>
inc-bound off b, x))

fun merge-while-cond-bdy new-pos cond-uses bdy-uses = bdy-uses |> map (fn r =>
  case returned r of
    SOME p => merge-uses (nth cond-uses (new-pos p)) r
  | NONE => r)

datatype exn-constr = L | R
fun dest-throw-value (Const (@{const-name Inl}, -) $ x) =
      let val (constrs, bdy) = dest-throw-value x in (L :: constrs, bdy) end
  | dest-throw-value (Const (@{const-name Inr}, -) $ x) =
      let val (constrs, bdy) = dest-throw-value x in (R :: constrs, bdy) end
  | dest-throw-value x = ([], x)

fun local-exn cs = (List.last cs = R)
fun mk-exn [R] x = Inr x
  | mk-exn (c::cs) x = ((case c of L => Inl | - => raise Incompatible-Use) o mk-exn
cs) x

fun extend-bounds xs base =
  let
    fun ext x base = if not (member (fn (x, y) => same-bound (fst x) (fst y)) base
x) then base @ [x] else base
  in
    fold ext xs base
  end

exception Done of uses list

fun fold-done done f xs a =
  fold (fn x => fn a =>
    let
      val res = f x a
    in
      if done res then raise Done res else res
    end) xs a
```

```
fun fold-sup inf f xs =
  fold-done (forall (fn u => u = Used)) f xs inf

fun fold-merge inf bounds f = fold-sup (inf bounds) (fn x => fn a => merge-results
[a, f bounds x])

in
(*
analyse-uses bounds t: returns the list of usages corresponding to the list of input
bounds.
each input bound is represented as ((de bruijn index, [aliases]), (name, type))
The position in the list corresponds to the position in the tuple of the binding.
Special care has to be taken to handle L2-while as this may shuffle the tuple and
introduce
various aliases in the initalisation expression, the condition and the body.
*)
fun analyse-uses bounds t =
  let
    val sup = map (K Used) bounds
    val inf = map (K Unused)
    val done = forall (fn u => u = Used)
    fun fastpath uses = if done uses then raise Done uses else ()
    val fold-merge = fold-merge inf
    fun analyse bounds t =
        — We do not handle L2-catch only L2-try as this analysis is supposed to
happen after replacing L2-catch by L2-try. After phase IO the freshly emerging
L2-catch are only very local handler functions which we do not attempt to optimise
      case t of
        Const (@{const-name L2-gets}, -) $ Abs (-, -, bdy) $ - =>
          let
            val bounds = inc-bounds 1 bounds
            val vars = HOLogic.strip-tuple bdy |> number-pos
            val uss = map (fn (pos, t) => analyse-result bounds Inr pos true t) vars
          in
            merge-results-default (inf bounds) (* i.e. () is returned *) uss
          end
      | Const (@{const-name L2-throw}, -) $ t1 $ - =>
          let
            val (constrs, bdy) = dest-throw-value t1
            val vars = HOLogic.strip-tuple bdy |> number-pos
          val uss = map (fn (pos, t) => analyse-result bounds (mk-exn (L::constrs))
pos (local-exn (L::constrs)) t) vars
          in
            merge-results-default (inf bounds) (* e.g. if Inr () is thrown *) uss
          end
      | Const (@{const-name L2-seq}, -) $ t1 $ t2 =>
          let
```

```
      val uses1 = analyse bounds t1
      val - = fastpath uses1
      val (off, bdy) = (Synthesize-Rules.strip-abs-prod t2) |>> length
      val bounds' = propagate-returns off bounds uses1
      val uses2 = analyse bounds' bdy
    in
      merge-results [map remove-return uses1, uses2]
    end
  | Const (@{const-name L2-try}, -) $ t1 =>
    let
      val uses1 = analyse bounds t1
      val handled-uses = map handled uses1
    in
      handled-uses
    end
  | Const (@{const-name L2-while}, -) $ c $ bdy $ vars $ - =>
    let
      val n = length bounds
      val vars = HOLogic.strip-tuple vars |> number-pos
      val off = length vars
       val uses-vars = map (fn (pos, t) => analyse-result bounds Inr pos true
t) vars
        |> merge-results-default (inf bounds)
        val (cond-bounds, cond-bdy) = Synthesize-Rules.strip-abs-prod c |>>
number-bounds
          (* extend bounds to cover at least the arity of cond and body, strip in
the end *)
        val bounds' = propagate-returns off bounds uses-vars |> extend-bounds
cond-bounds
      val uses-cond = analyse bounds' cond-bdy
      val - = @{assert} (forall (not o propagation) uses-cond)
      fun new-pos p =
        let val i = nth cond-bounds (p - 1) |> #1 |> #1
        in find-index (fn (b, -) => eq-bound b i) bounds' end
      val - = fastpath uses-cond
      val (-, bdy) = Synthesize-Rules.strip-abs-prod bdy
      val uses-bdy = analyse bounds' bdy
      val cond-bdy = merge-while-cond-bdy new-pos uses-cond uses-bdy
      val res = merge-results [uses-vars, take n uses-cond, take n cond-bdy]
    in
      res
    end
  | Const (@{const-name L2-unknown}, -) $ - => inf bounds
  | Const (@{const-name L2-call}, -) $ f $ emb $ - =>
    fold-merge bounds analyse [f, emb]
  | t1 $ t2 =>
    fold-merge bounds analyse [t1, t2]
  | Abs (-, -, bdy) => analyse-uses (inc-bounds 1 bounds) bdy
  | leaf => analyse-used bounds leaf;
```

```
  in
    analyse bounds t
    handle Done - => sup
  end

fun mk-tuple [] = (HOLogic.unit, HOLogic.unitT)
  | mk-tuple [((i, -), (-, T))] = (Bound i, T)
  | mk-tuple (((i, -), (-, T))::: xs) =
    let
      val (t, T') = mk-tuple xs
    in (HOLogic.pair-const T T' $ Bound i $ t, HOLogic.mk-prodT (T, T')) end

fun result-from-rel prj-rel res =
  case (res, prj-rel) of
    (Inr p, @{term-pat rel-sum - (rel-project ?prj)}) =>
        ((nth (Synthesize-Rules.arity-from-projection prj) (p - 1))
        handle Subscript => raise TERM (prj-from-rel: cannot infer projection ^
@{make-string} (p, prj), [prj, prj-rel]))
  | (Inr -, @{term-pat rel-sum - (=)}) => true
  | (Inr -, @{term-pat (=)}) => true
  | (Inl l, @{term-pat rel-sum ?L1 -}) => result-from-rel L1 l
  | - => raise TERM (prj-from-rel: result ' ^ @{make-string} res ^ ' cannot be
extracted, [prj-rel])



fun prj-of-uses prj-rel us = us |> map (fn u =>
  case u of
    Used => true
  | Unused => false
  | Propagated rs => exists (result-from-rel prj-rel) rs) (* Value propagated until
end but still used as a result *)

local
val merge-prj = map2 (fn b1 => fn b2 => b1 orelse b2)
in
(*
A while loop in itself might have dependencies of the bound variables between the
body and
the condition. These dependencies should not be optimised away even if the value
is no longer used
after the while loop.
*)
fun constrain-projection prj-rel prj t =
  case t of
    Const (@{const-name L2-seq},-) $ t1 $ t2 => constrain-projection prj-rel prj
t2
  | Const (@{const-name L2-call},-) $ - $ - $ - => map (K true) prj
        (* As we don't recurse into body, preserve result type of call *)
```

```
  | Const (@{const-name L2-condition},-) $ - $ t1 $ t2 =>
      let
        val prj1 = constrain-projection prj-rel prj t1
        val prj2 = constrain-projection prj-rel prj t2
        val merge = merge-prj prj1 prj2
      in
       merge
      end
  | Const (@{const-name L2-while}, wT) $ c $ bdy $ vars $ names =>
       let
        val vars' = HOLogic.strip-tupleT (fastype-of vars) |> number-bounds
          |> map (fn ((i, aliases),T) => ((i, aliases), (x ^ string-of-int i,T)))
        val (bounds, -) = mk-tuple vars'
        val w = Const (@{const-name L2-while}, wT) $ c $ bdy $ bounds $ names
        val uses = analyse-uses vars' w
       val uses = if null uses (* vars is unit *) then map (K Unused) prj else uses
        val relevant-uses = map remove-return uses
        val uses-needed = prj-of-uses prj-rel relevant-uses
        val merge = merge-prj prj uses-needed
       in
        merge
       end
  | - => prj
end
end

fun analyse-uses' ctxt prj-rel t =
  let
    val (bounds, bdy) = Synthesize-Rules.strip-abs-prod t
    val uses = analyse-uses (number-bounds bounds) bdy
     val - = Utils.verbose-msg 2 ctxt (fn - => analysed uses:  ^ @{make-string}
(map fst bounds ~~ uses))
  in
    prj-of-uses prj-rel uses
  end

fun mk-case-prod (bdy, bT) [] = (bdy, bT)
  | mk-case-prod (bdy, bT) [(x,T)] = (Abs (x, T, bdy), bT)
  | mk-case-prod (bdy, bT) [(x,xT), (y,yT)] = (HOLogic.case-prod-const (xT, yT,
bT) $ (Abs (x, xT, Abs (y, yT, bdy))), HOLogic.mk-prodT (xT, yT))
  | mk-case-prod (bdy, bT) ((x,xT)::xs) =
    let
      val (bdy', bT') = mk-case-prod (bdy, bT) xs
   in (HOLogic.case-prod-const (xT, bT', bT) $ (Abs (x, xT, bdy')), HOLogic.mk-prodT
(xT, bT')) end

fun gen-mk-projection Ts prj-enc =
  let
    val n = length prj-enc
```

```
    val tagged-bounds = prj-enc |> gen-number-bounds (n - 1)
      |> map (fn ((i, aliases),t) => ((i, aliases),t, (Tuple-Tools.mk-el-name (n -
i), Tuple-Tools.mk-elT' Ts (n - i))))
     val proj-bounds = map-filter (fn (i, t, xT) => if t then SOME (i, xT) else
NONE) tagged-bounds
    val vars = map #3 tagged-bounds
    val (bdy, bT) = mk-tuple proj-bounds
    val prj = mk-case-prod (bdy, bT) vars
  in
    fst prj
  end

fun mk-projection prj-enc = gen-mk-projection (map Tuple-Tools.mk-elT (1 upto
length prj-enc)) prj-enc

val - = Tuple-Tools.assert-cterm (mk-projection [true,false,false,true])
  @{cterm λ(x1, x2, x3, x4). (x1, x4)}

val - = Tuple-Tools.assert-cterm (mk-projection [false,false])
  @{cterm λ(x1, x2). ()}

fun split-project-rule ctxt prj-var orig-names new-names rule =
  let val rule = Thm.trim-context rule
  in fn prj-arity => if null prj-arity then rule else
    let
      val orig-arity = length prj-arity
      val new-arity = prj-arity |> filter I |> length
     val name-arities = map (rpair orig-arity) orig-names @ map (rpair new-arity)
new-names
      val prj = mk-projection prj-arity
      val ctxt' = Variable.declare-term prj ctxt
      val prj = Thm.cterm-of ctxt' prj
      val [rule'] = rule
        |> Drule.infer-instantiate ctxt' [(prj-var, prj)]
        |> single |> Proof-Context.export ctxt' ctxt
    in
      Tuple-Tools.gen-split-rule ctxt name-arities rule'
    end
  end

— Tailored for case L2-seq, will it ever by used for other cases?
fun infer-projection-arity benv ctxt pattern [arity-var, constrain-var, prjL-var, prjR-var]
concl =
  case Synthesize-Rules.match-rule-vars benv ctxt pattern [arity-var, constrain-var,
prjL-var, prjR-var] concl of
    [arity-stmt, constrain-stmt, prjL, prjR] =>
      let
        val prj-rel = infer-instantiate ‹L = prjL and prj = prjR in term ‹rel-sum
L (rel-project prj)›› ctxt
```

```
      val tagged-used = analyse-uses' ctxt prj-rel arity-stmt
        val - = Utils.verbose-msg 2 ctxt (fn - => derived projection (raw): ^
@{make-string} tagged-used)
      val constrained-used = constrain-projection prj-rel tagged-used constrain-stmt
        val - = Utils.verbose-msg 2 ctxt (fn - => constrained projection: ^
@{make-string} constrained-used)
    in constrained-used end
  | - => []


fun mk-rel-spec-monad-pattern ctxt - (@{term-pat Trueprop (rel-spec-monad ?Q ?R
?f ?g)}) =
  let
    val mi = fold (curry Int.max) (map maxidx-of-term [Q, R, f, g]) 0
    val gT = Logic.incr-tvar (mi + 1) (fastype-of g) (* FIXME: do I need incr-tvar?
*)
    val pat = infer-instantiate ‹Q = Q and R = R and f = f and g = ‹Var ((-g,
mi + 1), gT)›
              in prop ‹rel-spec-monad Q R f g›› ctxt
  in
    pat
  end
  | mk-rel-spec-monad-pattern - - - = raise Match


fun add-rel-spec-monad-split-rule rules-name = Synthesize-Rules.gen-add-split-rule
rules-name {only-schematic-goal = false} mk-rel-spec-monad-pattern
fun add-rel-spec-monad-split-rules rules-name = fold-map (fn (name, priority, names,
thm) => add-rel-spec-monad-split-rule rules-name name priority names thm)

val add-rel-spec-monad-infer-project-split-rule =
 Synthesize-Rules.gen-add-infer-project-split-rule mk-rel-spec-monad-pattern infer-projection-arity

fun add-rel-spec-monad-infer-project-split-rules rules-name rs context =
   context |> fold-map (fn (name, priority, names, new-names, prj-name, con-
strain-name, prjL-name, prjR-name, thm) =>
   let
     fun get-var name = Term.add-vars (Thm.prop-of thm) [] |> filter (fn ((n, -),-)
=> n = name)
       |> distinct (op =) |> the-single |> fst
     val prj-var = get-var prj-name
    val split = split-project-rule (Context.proof-of context) prj-var names new-names
thm
   in
    add-rel-spec-monad-infer-project-split-rule split rules-name {only-schematic-goal
= false} name priority names [constrain-name, prjL-name, prjR-name] thm
   end) rs

fun add-rel-spec-monad-project-split-rule split rules-name name priority prj-name
```

```
thm context =
  Synthesize-Rules.gen-add-project-split-rule mk-rel-spec-monad-pattern split
    rules-name {only-schematic-goal = false} name priority prj-name thm context

fun add-rel-spec-monad-project-split-rules rules-name rs context =
  context |> fold-map (fn (name, priority, names, new-names, prj-name, thm) =>
    let
      val - = assert (not (null names)) add-rel-spec-monad-project-split-rules: ex-
pecting at least one variable name
     fun get-var name = Term.add-vars (Thm.prop-of thm) [] |> filter (fn ((n, -),-)
=> n = name)
       |> distinct (op =) |> the-single |> fst
      val prj-var = get-var prj-name
    val split = split-project-rule (Context.proof-of context) prj-var names new-names
thm
    in
      add-rel-spec-monad-project-split-rule split rules-name name priority prj-name
thm
    end) rs

fun add-rel-spec-monad-rule rules-name = Synthesize-Rules.gen-add-rule rules-name
{only-schematic-goal = false} NONE mk-rel-spec-monad-pattern
fun add-rel-spec-monad-rules rules-name = fold (fn (name, priority, thm) =>
add-rel-spec-monad-rule rules-name name priority thm)


(* solve trivial leftover vars like ?f x y of type unit *)
fun smash-unit-vars ctxt = SUBGOAL (fn (t, i) =>
 let
    fun make-inst (v,T) =
      let
        val (argTs, @{typ unit}) = strip-type T
        fun abs T bdy = Abs (,T, bdy)
      in ((v,T), Thm.cterm-of ctxt (fold-rev abs argTs @{term ()})) end

    val unit-vars = Term.add-vars t []
      |> filter (fn (-,T) => body-type T = @{typ unit})

    val insts = map (make-inst) unit-vars
 in
   if null insts then no-tac
   else PRIMITIVE (Thm.instantiate (TVars.empty, Vars.make insts))
 end)

local
  val ss = simpset-of (put-simpset HOL-basic-ss @{context}
            addsimps @{thms HOL.simp-thms
                  surj-def rel-project-conv rel-Nonlocal-conv Product-Type.prod.case
c-exntype.case
```

```
        rel-liftE-apply rel-sum-eq-apply
        if-True if-False
      Product-Type.prod.inject c-exntype.inject List.list.inject String.char.inject
c-exntype.distinct
        sum.map-ident  unit-convs}
      |> Simplifier.add-cong @{thm if-cong})
in
fun clarsimp-solve-tac ctxt i =
  let
   val more-simps = Named-Theorems.get ctxt @{named-theorems rel-spec-monad-rewrite-simps}
  in
    CHANGED (TRY (smash-unit-vars ctxt i)
      THEN clarsimp-tac (put-simpset ss ctxt addsimps more-simps) i
      THEN (REPEAT (resolve-tac ctxt @{thms TrueI refl gen-unit-eq conjI} i)))
  end
end

fun get-prj-sum @{term-pat rel-sum ?L ?R} @{term-pat Inl ?x} = get-prj-sum L x
  | get-prj-sum @{term-pat rel-sum ?L ?R} @{term-pat Inr ?x} = get-prj-sum R x
  | get-prj-sum @{term-pat rel-project ?prj} - _ = prj
  | get-prj-sum t - _ = t

fun get-prj @{term-pat (?R, ?x)} = get-prj-sum R x
  | get-prj t = t

fun sanitize-names-tac ctxt = SUBGOAL (fn (t, i) =>
  if not (member (op =) (Term.add-const-names t []) @{const-name SANITIZE-NAMES})
then no-tac else
  let
    fun mk-abs [] t = t
      | mk-abs (T::Ts) t = Abs (Name.uu-, T, mk-abs Ts t)

    val concl = Utils.concl-of-subgoal-open t
    val @{term-pat ‹Trueprop (SANITIZE-NAMES ?prj ?ns ?ns')›} = concl
    val (ns', -) = strip-comb ns'
  in
    if is-Var ns' then
      let
        val Var (ns',T) = ns'
        val names = Utils.decode-isa-list ns
        val prj' = get-prj prj
        val bs = Synthesize-Rules.arity-from-projection prj'

        val names' = (if length bs <= 1
              then filter-out (member (op =) [const‹global-exn-var-clocal›]) names
                else if length bs = length names then filter (fn (b, -) => b) (bs ~~
names) |> map snd
                else (warning (sanitize-names-tac: unexpected number of names in:
^ Syntax.string-of-term ctxt concl ^ \n ^
```

```
                    (bs, names): ^@{make-string} (bs, names))
                  ; names))
            |> Utils.encode-isa-list @{typ nat} |> mk-abs (binder-types T) |>
Thm.cterm-of ctxt
     in
    PRIMITIVE (Thm.instantiate (TVars.empty, Vars.make [((ns',T), names')]))
THEN
     resolve-tac ctxt @{thms sanitize-namesI} i
    end
   else
    resolve-tac ctxt @{thms sanitize-namesI} i
 end
 handle Bind => no-tac
 )

(* Avoid superflouous dependency of variable on constant terms, e.g. '?x ()'. Oth-
erwise resolution results in
 *  non eta−contracted terms which make Net.match-term fail later on.
 *)
fun norm-synthesis-var ctxt = SUBGOAL (fn (t,i) => fn st =>
 let
    val concl = Utils.concl-of-subgoal-open t
    val @{term-pat Trueprop (rel-spec-monad ?P ?Q ?f ?g)} = concl
    val (Var ((g, i), gT), args as (-::-)) = strip-comb g
    val (argTs, resT) = strip-type gT
    val n = length args
    val (appliedTs, unappliedTs) = chop n argTs
   val tagged-args = (args ~~ appliedTs) |> map (fn (arg, T) => (null (loose-bnos
arg), (arg, T)))
    val (args', appliedTs') = tagged-args |> filter-out fst |> map snd |> split-list
     val - = if length args' = n then raise Bind else () (* avoid trivial identity
instantiations *)
     val g' = Var ((g, Thm.maxidx-of st + 1), appliedTs' @ unappliedTs --->
resT)
    fun mk-inst [] g' b = g'
      | mk-inst ((tag, (t, T))::args) g' b =
          let
            val g' = if tag then g' else  (g'$Bound b)
          in Abs (, T, mk-inst args g' (b − 1)) end
     val insts = [(((g,i), gT), Thm.cterm-of ctxt (mk-inst tagged-args g' (length
tagged-args − 1)))]
 in
   if null insts then no-tac st
   else PRIMITIVE (Thm.instantiate (TVars.empty, Vars.make insts)) st
 end
 handle Bind => no-tac st
)

fun norm-resolve-split-thm rules-name ctxt =
```

```
  let
    val simp-ctxt = Simplifier.put-simpset HOL-basic-ss ctxt
        addsimps @{thms
           HOL.simp-thms if-True if-False
           c-exntype.case-distrib [where h=from-xval]
           Product-Type.prod.case-distrib[where h=from-xval]
           Product-Type.prod.case c-exntype.case from-xval-simps}
        |> Simplifier.add-cong @{thm if-weak-cong}
        |> Simplifier.add-cong @{thm c-exntype.case-cong}
  in
    TRY' (norm-synthesis-var ctxt) THEN' Synthesize-Rules.resolve-split-thm rules-name
ctxt
     THEN-ALL-NEW (Simplifier.simp-tac simp-ctxt)
 end
end
›


declare [[verbose=3]]
setup ‹
Context.theory-map (
  Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-rules @{synthesize-rules-name
L2-rel-spec-monad} [
   (@{binding L2-unknown − rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-unknown}),
   (@{binding L2-modify − rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-modify}),
    (@{binding L2-gets − rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-gets}),
   (@{binding L2-condition − rel-xval}, 10, @{thm rel-spec-monad-eq-rel-xval-L2-condition}),
   (@{binding L2-throw − rel-xval}, 10, @{thm rel-spec-monad-L2-throw-sanitize-names}),
    (@{binding L2-spec − rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-spec}),
   (@{binding L2-assume − rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-assume}),
   (@{binding L2-guard − rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-guard}),
   (@{binding L2-guarded − rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-guarded}),
    (@{binding L2-fail}, 10, @{thm rel-spec-monad-L2-fail}),
    (@{binding L2-call − rel-xval}, 10, @{thm rel-spec-monad-rel-xval-L2-call}),



    (@{binding exec-concrete}, 10, @{thm rel-spec-monad-result-exec-concrete}),
    (@{binding exec-abstract}, 10, @{thm rel-spec-monad-result-exec-abstract}),

   (@{binding L2-unknown − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-unknown'}),
   (@{binding L2-modify − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-modify}),
   (@{binding L2-gets − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-gets'}),
   (@{binding L2-condition − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-condition}),
   (@{binding L2-throw − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-throw}),
   (@{binding L2-spec − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-spec}),
   (@{binding L2-assume − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-assume}),
   (@{binding L2-guard − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-guard}),
   (@{binding L2-guarded − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-guarded}),
   (@{binding L2-try − rel-project}, 20, @{thm rel-spec-monad-rel-project-L2-try}),
```

```
  (@{binding L2-catch − rel-project}, 10, @{thm rel-spec-monad-rel-project-L2-catch}),
  (@{binding liftE unit − rel-project}, 10, @{thm rel-spec-monad-rel-project-liftE-unit-id})
   ]

  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-sum ?L ?R (Inl ?l) ?V)}))) @{binding
rel-sum-Inl} 10 @{thm rel-sum.intros(1)}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-sum ?L ?R (Inr ?r) ?V)}))) @{binding
rel-sum-Inr} 10 @{thm rel-sum.intros(2)}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false}
      NONE (K (K (K @{pattern Trueprop (rel-sum ?L ?R (map-sum ?l ?r (Inl
?x)) ?v)}))) @{binding rel-sum-map-sum-InlI} 10 @{thm rel-sum-map-sum-InlI}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-sum ?L ?R (map-sum ?l ?r (Inr ?x))
?v)}))) @{binding rel-sum-map-sum-InrI} 10 @{thm rel-sum-map-sum-InrI}

  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-xval ?L ?R (Exn ?l) ?V)}))) @{binding
rel-xval-Exn} 10 @{thm rel-xval.intros(1)}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
     (K (K (K @{pattern Trueprop (rel-xval ?L ?R (Result ?r) ?V)}))) @{binding
rel-xval-Result} 10 @{thm rel-xval.intros(2)}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false}
      NONE (K (K (K @{pattern Trueprop (rel-xval ?L ?R (map-xval ?l ?r (Exn
?x)) ?v)}))) @{binding rel-map-xval-xval-ExnI} 10 @{thm rel-map-xval-xval-ExnI}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-xval ?L ?R (map-xval ?l ?r (Result ?x))
?v)}))) @{binding rel-map-xval-xval-ResultI} 10 @{thm rel-map-xval-xval-ResultI}

  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-xval rel-Nonlocal ?R (case ?e of Nonlocal
v ⇒ Exn (Nonlocal v) | - ⇒ Result ?x) (Exn ?v'))})))
    @{binding rel-xval-rel-Nonlocal-case-ExnI} 10 @{thm rel-xval-rel-Nonlocal-case-ExnI}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-xval rel-Nonlocal ?R (case ?e of Nonlocal
v ⇒ Exn (Nonlocal v) | - ⇒ Result ?x) (Result ?v'))})))
    @{binding rel-xval-rel-Nonlocal-case-ResultI} 10 @{thm rel-xval-rel-Nonlocal-case-ResultI}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
```

```
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-xval rel-Nonlocal (=) (map-xval Nonlocal
?f (case ?e of Nonlocal x ⇒ Exn x | - ⇒ Result ?x)) ?v)})))
    @{binding rel-xval-rel-Nonlocal-map-xvalI} 10 @{thm rel-xval-rel-Nonlocal-map-xvalI}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
      (K (K (K @{pattern Trueprop (rel-xval ?L ?R (case ?e of Nonlocal x ⇒ Exn
(Nonlocal x) | - ⇒ Result ?v)
                  (case ?e of Nonlocal x ⇒ Exn (Nonlocal x) | - ⇒ Result ?v'))})))
    @{binding rel-xval-case-Nonlocal-sameI} 10 @{thm rel-xval-case-Nonlocal-sameI}
  #> Synthesize-Rules.gen-add-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false} NONE
        (K (K (K @{pattern Trueprop (rel-project ?prj ?x ?y)}))) @{binding
rel-project} 10 @{thm rel-projectI}

 #> Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-split-rules @{synthesize-rules-name
L2-rel-spec-monad} [
   (@{binding L2-seq − rel-xval}, 10, [f, g], @{thm rel-spec-monad-L2-seq-rel-xval-same-split}),
   (@{binding L2-while − rel-xval}, 10, [B, B'], @{thm rel-spec-monad-L2-while-rel-xval-same-split}),
   (@{binding L2-call − rel-project}, 10, [emb, emb'], @{thm rel-spec-monad-rel-project-L2-call-adapt-emb}),
   (@{binding L2-seq − rel-project (exit handler)}, 20, [g, g'], @{thm rel-spec-monad-L2-seq-exit-handler})
   ]
 ##>> Synthesize-Rules.gen-add-split-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false}
    (K (K (K @{pattern Trueprop (rel-spec-monad ?R (rel-xval ?L (=)) (L2-catch
?m ?f) ?g)})))
   @{binding L2-catch-try − rel-xval} 10 [f] @{thm rel-spec-monad-rel-xval-try-catch}
 ##>> Synthesize-Rules.gen-add-split-rule @{synthesize-rules-name L2-rel-spec-monad}
{only-schematic-goal = false}
    (K (K (K @{pattern Trueprop (rel-fun ?A ?B ?f ?g)})))
    @{binding rel-funI} 10 [f, g] @{thm rel-funI}
 ##>> Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-infer-project-split-rules
@{synthesize-rules-name L2-rel-spec-monad} [
   (@{binding L2-seq − rel-project}, 10, [f], [g], prj',m, L, prj, @{thm rel-spec-monad-rel-project-L2-seq})
   ]
 ##>> Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-project-split-rules @{synthesize-rules-name
L2-rel-spec-monad} [
   (@{binding L2-while − rel-project}, 10, [B,C,I], [B',C', I'] , prj, @{thm rel-spec-monad-rel-project-L2-while'})
   ]
  #> snd)
›

context stack-heap-raw-state
begin
declaration ‹
fn phi =>
  Rel-Spec-Monad-Synthesize-Rules.add-rel-spec-monad-rules @{synthesize-rules-name
L2-rel-spec-monad} [
      (@{binding with-fresh-stack-ptr − rel-sum}, 10, Morphism.thm phi @{thm
```

rel-spec-monad-rel-xval-with-fresh-stack-ptr}),
(@{*binding with-fresh-stack-ptr − rel-project*}, *10*, *Morphism.thm phi* @{*thm rel-spec-monad-rel-project-with-fresh-stack-ptr*})]
›
end
declare [[*verbose=0*]]

lemma *map-sum-right*: (*map-sum l r v* = *Inr x*) = (∃ *v'*. *v* = *Inr v'* ∧ *x* = *r v'*)
by (*cases v*) *auto*

lemma *map-sum-left*: (*map-sum l r v* = *Inl x*) = (∃ *v'*. *v* = *Inl v'* ∧ *x* = *l v'*)
by (*cases v*) *auto*

lemma *case-Nonlocal-Inr*: ((*case e of Nonlocal x* ⇒ *Inl x* | - ⇒ *Inr v*) = *Inr y*) = (*is-local e* ∧ (*v* = *y*))
by (*cases e*) *auto*

lemma *case-Nonlocal-Inl*: ((*case e of Nonlocal x* ⇒ *Inl x* | - ⇒ *Inr v*) = *Inl y*) = (*e* = *Nonlocal y*)
by (*cases e*) *auto*

method-setup *resolve-split* = ‹
Scan.succeed (*fn ctxt* =>
let
val simp-ctxt = (*ctxt* |> *Simplifier.clear-simpset*) *addsimps* @{*thms*
from-xval-simps
if-distrib [*where f* = *from-xval*]}
in
SIMPLE-METHOD'
(*Rel-Spec-Monad-Synthesize-Rules.norm-resolve-split-thm* @{*synthesize-rules-name L2-rel-spec-monad*} *ctxt*
THEN-ALL-NEW (*simp-tac simp-ctxt*))
end)›
resolve with rel-spec-monad rules of right arity

method-setup *sanitize-names* = ‹
Scan.succeed (*fn ctxt* => *SIMPLE-METHOD'* (*Rel-Spec-Monad-Synthesize-Rules.sanitize-names-tac ctxt*))›
sanitize name annotations for L2-throw

method-setup *verbose-msg* =
‹*Scan.lift* (*Parse.embedded* >>
(*fn msg* => *fn ctxt* => *SIMPLE-METHOD* (*fn st* => (*Utils.verbose-msg 1 ctxt* (*fn* - => *msg*); *all-tac st*))))›
print a message (*if flag autocorres-verbose is turned on*)

method-setup *clarsimp-solve* = ‹
Scan.succeed (*fn ctxt* => *SIMPLE-METHOD'* (*Rel-Spec-Monad-Synthesize-Rules.clarsimp-solve-tac*

```
ctxt))›
  clarsimp and then solve, instantiating variables

method-setup trace =
 ‹Scan.lift ((Parse.embedded-input -- Parse.embedded) >> (fn (method-src, msg)
=> fn ctxt =>
    let
      val (m, tok) = Method.read-closure-input ctxt method-src
        val msg-m = Method.Basic (fn ctxt => SIMPLE-METHOD (fn st =>
(Utils.verbose-msg 1 ctxt (fn - => msg); all-tac st)))
    val trace-meth = Method.Combinator (Method.no-combinator-info, Method.Then,
[m, msg-m])
    in
      Utils.timeap-method 3 ctxt (fn - => msg) (Method.evaluate trace-meth ctxt)
    end))›
  trace a method application (if flag autocorres-verbose is turned on)

method rel-spec-monad-L2-step uses more-intro-thms =
  ((rule-tac more-intro-thms, verbose-msg applied: rule-tac) |
   trace resolve-split applied: resolve-split |
   trace assumption applied: assumption |
   trace sanitize-names applied: sanitize-names |
   trace clarsimp-solve applied: clarsimp-solve
 )

method rel-spec-monad-L2-rewrite =
  (use in ‹rel-spec-monad-L2-step more-intro-thms: method-facts›)+


ML ‹
structure L2-Exception-Rewrite =
struct

fun rhs-conv conv eq-thm =
  Conv.fconv-rule (Conv.arg-conv conv) eq-thm

 val rel-spec-monad-L2-rewrite-tac = UMM-Tools.tactic-from-src @{context} ‹rel-spec-monad-L2-rewrite›



fun abstract-try-catch ctxt t =
  let
    val goal = infer-instantiate ‹t = t in prop (schematic) ‹t = a›› ctxt
    val thm = Goal.prove ctxt [] [] goal (fn {prems, context,...} =>
          simp-tac ((Simplifier.add-cong @{thm c-exntype.case-cong} (put-simpset
HOL-basic-ss context))
       addsimps @{thms cond-return1 cond-return2 if-c-exntype-cases rel-spec-monad-eq-conv
[symmetric]}) 1 THEN
       resolve-tac context @{thms rel-spec-monad-rel-xvalI} 1 THEN
```

```
      rel-spec-monad-L2-rewrite-tac context [] THEN
      print-unsolved-tac abstract-try-catch: unfinished goal ctxt
    )
  in
    SOME thm
  end
  handle ERROR - => raise TERM (abstract-try-catch failed, [t])

fun abstract-try-catch-conv ctxt ct =
  case abstract-try-catch ctxt (Thm.term-of ct) of
     SOME eq => mk-meta-eq (Simplifier.simplify (Simplifier.clear-simpset ctxt
addsimps @{thms from-xval-simps}) eq)
  | NONE => (warning (abstract-try-catch-conv: failed to convert to nested excep-
tions. ^
    @{make-string} ct); Conv.all-conv ct)

fun rel-spec-monad-conv trace-unfinshed eq-intro ctxt lhs =
  let
    val ([lhs'], ctxt') = Variable.import-terms false [lhs] ctxt
    val goal = infer-instantiate ‹lhs = lhs' in prop (schematic) ‹lhs = A›› ctxt'
    val thm = Goal.prove ctxt' [] [] goal (fn {context, ...} =>
      resolve-tac context [eq-intro] 1 THEN
    simp-tac (Simplifier.clear-simpset context addsimprocs [@{simproc ETA-TUPLED}])
1 THEN
      rel-spec-monad-L2-rewrite-tac context [] THEN
      trace-unfinshed context)
    val [eq] = Variable.export ctxt' ctxt [mk-meta-eq thm]
    val eq = Thm.instantiate (Thm.match (Thm.lhs-of eq, Thm.cterm-of ctxt lhs))
eq
  in
    SOME eq
  end
  handle ERROR x => NONE

fun project-used-components-conv ctxt ct =
  let
    val trace = print-unsolved-tac project-used-components-conv: unfinished goal
  in
    case rel-spec-monad-conv trace @{thm rel-project-eqI} ctxt (Thm.term-of ct) of
      SOME eq => eq |> rhs-conv (Simplifier.rewrite (put-simpset HOL-basic-ss
ctxt addsimps
       (Utils.get-rules ctxt @{named-theorems L2opt}))) — get rid of now unused
result values, e.g. L2-unknown
   | NONE => (warning (project-used-components failed on:  ^ @{make-string}
ct); Conv.all-conv ct)
  end

end
›
```

end

19.6 Peep-hole optimisations applied to L2

theory *L2Peephole*
imports *L2Defs Tuple-Tools*
begin

lemmas *id-def* [*L2opt*]

lemma *L2-seq-skip* [*L2opt*]:
  L2-seq (*L2-gets* (λ-. ()) *n*) *X* = (*X* ())
  L2-seq Y (λ-. (*L2-gets* (λ-. ()) *n*)) = *Y*
  apply (*clarsimp simp*: *L2-seq-def L2-gets-def gets-return*)+
  done

lemma *L2-seq-L2-gets* [*L2opt*]: *L2-seq X* (λ*x*. *L2-gets* (λ-. *x*) *n*) = *X*
  apply (*clarsimp simp*: *L2-defs gets-return*)
  done

lemma *L2-seq-L2-gets-unit*[*L2opt*]: *L2-seq* (*L2-gets g ns*) (λ*x*:: *unit*. *f x*) = *f* ()
  apply (*clarsimp simp*: *L2-defs*)
  by (*rule spec-monad-eqI*) (*auto simp add*: *runs-to-iff*)

lemma *L2-seq-L2-gets-const*: *L2-seq* (*L2-gets* (λ-. *c*) *n*) *X* = *X c*
  apply (*clarsimp simp*: *L2-defs liftE-def gets-return*)
  done

lemma *const-propagation-cong*: *X c* = *X*′ *c* ⟹ (*L2-seq* (*L2-gets* (λ-. *c*) *n*) *X*) = (*L2-seq* (*L2-gets* (λ-. *c*) *n*) *X*′)
  by (*clarsimp simp*: *L2-seq-L2-gets-const*)

lemma *L2-seq-const*′:
  assumes *bdy-eq*: *f c* ≡ *g c*
  shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *L2-seq* (*L2-gets* (λ-. *c*) *n*) *g*
  apply (*rule eq-reflection*)
  unfolding *L2-defs*
  apply (*rule spec-monad-eqI*)
  apply (*auto simp add*: *bdy-eq runs-to-iff*)
  done

lemma *L2-seq-const*:
  assumes *bdy-eq*: *f*′ ≡ *g*′
  assumes *f-app*: *f c* ≡ *f*′
  assumes *g-app*: *g c* ≡ *g*′

shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *L2-seq* (*L2-gets* (λ-. *c*) *n*) *g*
proof −
from *bdy-eq f-app g-app* **have** *f c* ≡ *g c*
by *simp*
then show *PROP ?thesis*
by (*rule L2-seq-const′*)
qed

lemma *L2-seq-const-stop*:
assumes *bdy-eq*: *f′* ≡ *g′*
assumes *f-app*: *f c* ≡ *f′*
assumes *g-app*: *g c* ≡ *g′*
shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *STOP* (*L2-seq* (*L2-gets* (λ-. *c*) *n*) *g*)
unfolding *STOP-def* **using** *bdy-eq f-app g-app*
by (*rule L2-seq-const*)

lemma *L2-seq-const-stop′*:
assumes *bdy-eq*: *f c* ≡ *g′*
assumes *g-app*: *g c* ≡ *g′*
shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *STOP* (*L2-seq* (*L2-gets* (λ-. *c*) *n*) *g*)
apply (*rule L2-seq-const-stop*)
apply (*rule bdy-eq*)
apply *simp*
apply (*rule g-app*)
done

lemma *L2-seq-const-stop″*:
assumes *bdy-eq*: *f c* ≡ *g c*
shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *STOP* (*L2-seq* (*L2-gets* (λ-. *c*) *n*) *g*)
apply (*rule L2-seq-const-stop′*)
apply (*rule bdy-eq*)
apply *simp*
done

lemma *L2-seq-const-stop‴*:
assumes *bdy-eq*: *f c* ≡ *g*
shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *STOP* (*L2-seq* (*L2-gets* (λ-. *c*) *n*) (λ-. *g*))
apply (*rule L2-seq-const-stop′*)
apply (*rule bdy-eq*)
apply *simp*
done

lemma *L2-marked-seq-gets-cong*:
c=*c′* ⟹ *L2-seq-gets c n A* ≡ *L2-seq-gets c′ n A*
by *simp*

lemma *L2-marked-seq-gets-stop*:
assumes *bdy-eq*: *f c* ≡ *g*
shows *L2-seq-gets c n f* ≡ *STOP* (*L2-seq-gets c n* (λ-. *g*))

unfolding *L2-seq-gets-def* **using** *bdy-eq*
by (*rule L2-seq-const-stop'''*)

lemma *L2-marked-seq-gets-stop'*:
assumes *bdy-eq*: $f\ c \equiv g\ c$
shows *L2-seq-gets c n f* $\equiv$ *STOP* (*L2-seq-gets c n g*)
unfolding *L2-seq-gets-def STOP-def*
by (*simp add*: *L2-gets-bind bdy-eq*)

lemma *L2-marked-seq-gets-stop''*:
assumes *bdy-eta*: $f \equiv f'$
assumes *bdy-eq*: $\bigwedge v.\ v = c \Longrightarrow f'\ v \equiv g\ v$
shows *L2-seq-gets c n f* $\equiv$ *STOP* (*L2-seq-gets c n g*)
unfolding *L2-seq-gets-def* **using** *bdy-eq bdy-eta*
by (*simp add*: *L2-gets-bind STOP-def*)

lemma *L2-guarded-block-cong*: *L2-guarded g c* = *L2-guarded g c*
by *simp*

lemma *L2-guarded-cong-stop'*:
assumes *guard-eq*: $\bigwedge s.\ g\ s \equiv g'\ s$
assumes *bdy-eq*: $\bigwedge s.\ g'\ s \Longrightarrow run\ c\ s \equiv run\ c'\ s$
shows *L2-guarded g c* $\equiv$ *STOP* (*L2-guarded g' c'*)
unfolding *L2-guarded-def L2-defs STOP-def*
apply (*rule eq-reflection*)
apply (*rule spec-monad-ext*)
using *guard-eq bdy-eq*
apply (*auto simp add*: *run-guard run-bind*)
done

lemma *L2-seq-guard-cong-stop0*:
assumes *guard-eq*: $\bigwedge s.\ g\ s \equiv g'\ s$
assumes *bdy-eq*: $\bigwedge s.\ g'\ s \Longrightarrow run\ (c\ ())\ s \equiv run\ c'\ s$
shows *L2-seq-guard g c* $\equiv$ *STOP* (*L2-seq-guard g'* (λ-. *c'*))
unfolding *L2-seq-guard-def STOP-def L2-defs*
apply (*rule eq-reflection*)
apply (*rule spec-monad-ext*)
using *guard-eq bdy-eq*
apply (*auto simp add*: *run-guard run-bind*)
done

lemma *L2-guarded-cong-stop*:
assumes *guard-eq*: $g \equiv g'$
assumes *bdy-eq*: $\bigwedge s.\ g'\ s \Longrightarrow run\ c\ s \equiv run\ c'\ s$
shows *L2-guarded g c* $\equiv$ *STOP* (*L2-guarded g' c'*)
unfolding *L2-guarded-def L2-defs STOP-def*
apply (*rule eq-reflection*)
apply (*rule spec-monad-ext*)

using *guard-eq bdy-eq*
apply (*auto simp add*: *run-guard run-bind*)
done

lemma *L2-seq-assoc* :
L2-seq (*L2-seq A* (λ*x*. *B x*)) *C* = *L2-seq A* (λ*x*. *L2-seq* (*B x*) *C*)
apply (*clarsimp simp*: *L2-seq-def bind-assoc*)
done

lemma *L2-seq-assoc′* [*L2opt*]:
L2-seq (*L2-seq A B*) *C* = *L2-seq A* (λ*x*. *L2-seq* (*B x*) *C*)
apply (*clarsimp simp*: *L2-seq-def bind-assoc*)
done

lemma *L2-seq-rev-assoc′*:
L2-seq A (λ*x*. (*L2-seq* (*B x*) (*C x*))) =
L2-seq (*L2-seq A* (λ*x*. *L2-seq* (*B x*) (λ*y*. *L2-gets* (λ-. (*x*, *y*)) *ns*))) (λ(*x*, *y*). *C x y*)
apply (*clarsimp simp*: *L2-seq-def L2-gets-def bind-assoc*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-rev-assoc-accumulated′*:
L2-seq A (λ(*a*, *x*). (*L2-seq* (*B a x*) (*C a x*))) =
L2-seq (*L2-seq A* (λ(*a*, *x*). *L2-seq* (*B a x*) (λ*y*. *L2-gets* (λ-. (*a*, *x*, *y*)) *ns*))) (λ(*a*, *x*, *y*). *C a x y*)
apply (*clarsimp simp*: *L2-seq-def L2-gets-def bind-assoc*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-rev-assoc*:
L2-seq A (λ*a*. (*L2-seq* (*B a*) (*C a*))) =
L2-seq (*L2-seq A* (λ*a*. *L2-seq* (*B a*) (λ*x*. *L2-gets* (λ-. (*x*, *a*)) *ns*))) (λ(*x*, *a*). *C a x*)
apply (*clarsimp simp*: *L2-seq-def L2-gets-def bind-assoc*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-rev-assoc-accumulated*:
L2-seq A (λ(*x*, *a*). (*L2-seq* (*B a x*) (*C a x*))) =
L2-seq (*L2-seq A* (λ(*x*, *a*). *L2-seq* (*B a x*) (λ*y*. *L2-gets* (λ-. (*y*, *x*, *a*)) *ns*))) (λ(*y*, *x*, *a*). *C a x y*)
apply (*clarsimp simp*: *L2-seq-def L2-gets-def bind-assoc*)
apply (*rule spec-monad-eqI*)

apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-trim-after-throw* [*L2opt*]:
L2-seq (*L2-throw x n*) *Z* = (*L2-throw x n*)
apply (*clarsimp simp*: *L2-seq-def L2-throw-def*)
done

lemma *L2-guard-true* [*L2opt*]: *L2-guard* (λ-. *True*) = *L2-gets* (λ-. ()) []
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-guard*)
done

This rule can be too expensive in simplification as it might invoke the arithmetic solver

lemma *L2-guard-solve-true*: ⟦ ⋀*s*. *P s* ⟧ ⟹ *L2-guard P* = *L2-gets* (λ-. ()) []
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-guard*)
done

lemma *L2-guard-false* [*L2opt*]: *L2-guard* (λ-. *False*) = *L2-fail*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-guard*)
done

This rule can be too expensive in simplification as it might invoke the arithmetic solver

lemma *L2-guard-solve-false*: ⟦ ⋀*s*. ¬ *P s* ⟧ ⟹ *L2-guard P* = *L2-fail*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-guard*)
done

lemma *L2-spec-empty* [*L2opt*]:

L2-spec {} = *L2-fail*
⟦ ⋀*s t*. ¬ *C s t* ⟧ ⟹ *L2-spec* {(*s*, *t*). *C s t*} = *L2-fail*
unfolding *L2-defs*
apply (*rule spec-monad-ext*; *simp add*: *run-bind run-assert-result-and-state*)+
done

lemma *L2-unknown-bind-unbound*[*L2opt*]:
L2-seq (*L2-unknown ns*) (λ*x*. *f*) = *f*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)

done

lemma *L2-unknown-bind-unbound'*:
$f = (\lambda\text{-}.\ g) \Longrightarrow$ *L2-seq* (*L2-unknown ns*) $f = g$
by (*simp add*: *L2-unknown-bind-unbound*)

lemma *L2-seq-L2-unknown-unit*[*L2opt*]: *L2-seq* (*L2-unknown ns*) (λx:: *unit*. *f x*) = *f* ()
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

The following rule can cause some exponential blowup, especially when rewriting is not successful. Note that *f* is duplicated in the premise. The more restricted *L2-seq* (*L2-unknown ?ns*) (λx. *?f*) = *?f* should also do the job, in case of properly initialised variables. Maybe we should remove it entirely from "L2opt".

lemma *L2-unknown-bind* []:
($\bigwedge a\ b$. *f a* = *f b*) $\Longrightarrow$ (*L2-seq* (*L2-unknown ns*) *f*) = *f undefined*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*clarsimp simp add*: *runs-to-def-old*)
apply *metis*
done

lemma *L2-seq-guard-cong*:
⟦ *P* = *P'*; $\bigwedge s$. *P' s* $\Longrightarrow$ *run X s* = *run X' s* ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) = *L2-seq* (*L2-guard P'*) (λ-. *X'*)
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *L2-seq-guard-cong'*:
⟦ *P* ≡ *P'*; $\bigwedge s$. *P' s* $\Longrightarrow$ *run X s* ≡ *run X' s* ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) ≡ *L2-seq* (*L2-guard P'*) (λ-. *X'*)
apply (*rule eq-reflection*)
apply (*rule L2-seq-guard-cong*)
by *auto*

lemma *L2-seq-guard-cong-stop*:
⟦ *P* = *P'*; $\bigwedge s$. *P' s* $\Longrightarrow$ *run X s* = *run X' s* ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) = *STOP* (*L2-seq* (*L2-guard P'*) (λ-. *X'*))
unfolding *STOP-def*

by (*rule L2-seq-guard-cong*, *auto*)

lemma *L2-seq-guard-cong-stop'*:
⟦ $P \equiv P'$; $\bigwedge s.\ P'\ s \Longrightarrow run\ X\ s \equiv run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) ≡ *STOP* (*L2-seq* (*L2-guard P'*) (λ-. *X'*))
apply (*rule eq-reflection*)
apply (*rule L2-seq-guard-cong-stop*)
by *auto*

lemma *L2-seq-guard-cong-stop''*:
⟦$\bigwedge s.\ P\ s \Longrightarrow run\ X\ s \equiv run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) ≡ *STOP* (*L2-seq* (*L2-guard P*) (λ-. *X'*))
apply (*rule eq-reflection*)
apply (*rule L2-seq-guard-cong-stop*)
by *auto*

lemma *L2-seq-guard-cong-stop'''*:
⟦$\bigwedge s.\ P\ s \Longrightarrow run\ (X\ ())\ s \equiv run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) *X* ≡ *STOP* (*L2-seq* (*L2-guard P*) (λ-. *X'*))
unfolding *STOP-def L2-defs*
apply (*rule eq-reflection*)
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *L2-marked-seq-guard-block-cong*:
L2-seq-guard P X = *L2-seq-guard P X*
by (*rule refl*)

lemma *L2-marked-seq-guard-cong*:
⟦$P = P'$; $\bigwedge s.\ P'\ s \Longrightarrow run\ (X\ ())\ s = run\ X'\ s$⟧ $\Longrightarrow$ *L2-seq-guard P X* = *L2-seq-guard P'* (λ-. *X'*)
unfolding *L2-seq-guard-def L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *L2-gaurded-keep-guard-cong*:
($\bigwedge s.\ g\ s \Longrightarrow run\ c\ s = run\ c'\ s$) $\Longrightarrow$ *L2-guarded g c* = *L2-guarded g c'*
unfolding *L2-guarded-def L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *gets-guard-move-before* [*L2opt*]:
L2-seq (*L2-gets f ns*) (λ*r*. *L2-seq* (*L2-guard P*) (λ-. *X r*)) =
L2-seq (*L2-guard P*) (λ-. *L2-seq* (*L2-gets f ns*) *X*)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)

apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-guard-cong''*:
⟦ ⋀*s*. *P s* = *P' s*; ⋀*s*. *P' s* ⟹ *run X s* = *run X' s* ⟧ ⟹ *L2-seq* (*L2-guard P*) (λ-. *X*) = *L2-seq* (*L2-guard P'*) (λ-. *X'*)
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *guard-triv* [*L2opt*]: *L2-seq* (*L2-guard* (λ*s*. *True*)) (λ-. *X*) = *X*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *L2-condition-cong*:
⟦ *C* = *C'*; ⋀*s*. *C' s* ⟹ *run A s* = *run A' s*;⋀*s*. ¬ *C' s* ⟹ *run B s* = *run B' s* ⟧ ⟹ *L2-condition C A B* = *L2-condition C' A' B'*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-condition simp-implies-def*)
done

lemma *L2-condition-cong'*:
⟦ ⋀*s*. *C s* = *C' s*; ⋀*s*. *C' s* ⟹ *run A s* = *run A' s*;⋀*s*. ¬ *C' s* ⟹ *run B s* = *run B' s* ⟧ ⟹ *L2-condition C A B* = *L2-condition C' A' B'*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-condition*)
done

lemma *L2-condition-true* [*L2opt*]: ⟦ ⋀*s*. *C s* ⟧ ⟹ *L2-condition C A B* = *A*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-condition*)
done

lemma *L2-condition-false* [*L2opt*]: ⟦ ⋀*s*. ¬ *C s* ⟧ ⟹ *L2-condition C A B* = *B*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-condition*)
done

lemma *L2-condition-true'* [*simp*]: *L2-condition* (λ*s*. *True*) *A B* = *A*
unfolding *L2-defs*
by *simp*

lemma *L2-condition-false'* [*simp*]: *L2-condition* (λ*s*. *False*) *A B* = *B*
unfolding *L2-defs*
by *simp*

lemma *L2-condition-same* [*L2opt*]: *L2-condition C A A* = *A*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-condition*)
done

lemma *L2-fail-seq* [*L2opt*]: *L2-seq L2-fail X* = *L2-fail*
unfolding *L2-defs*
by *simp*

lemma *bind-fail-propagates*: ⟦ *no-throw* (λ-. *True*) *A*; *always-progress A* ⟧ ⟹ *A* >>= (λ-. *fail*) = *fail*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*clarsimp simp add*: *no-throw-def runs-to-def-old runs-to-partial-def-old*)
by (*meson always-progressD*)

lemma *state-select-select-fail*:
state-select S ⟜ (λ-. *select UNIV*) ⟜ (λ-. *fail*) = *fail*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-fail-propagates* [*L2opt*]:
L2-seq (*L2-gets V n*) (λ-. *L2-fail*) = *L2-fail*
L2-seq (*L2-modify M*) (λ-. *L2-fail*) = *L2-fail*
L2-seq (*L2-spec S*) (λ-. *L2-fail*) = *L2-fail*
L2-seq (*L2-guard G*) (λ-. *L2-fail*) = *L2-fail*
L2-seq (*L2-unknown ns*) (λ-. *L2-fail*) = *L2-fail*
L2-seq L2-fail (λ-. *L2-fail*) = *L2-fail*
unfolding *L2-defs*
apply (*simp-all add*: *bind-fail-propagates always-progress-intros state-select-select-fail*)
done

lemma *L2-condition-distrib*:
L2-seq (*L2-condition C L R*) *X* = *L2-condition C* (*L2-seq L X*) (*L2-seq R X*)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemmas *L2-fail-propagate-condition* [*L2opt*] = *L2-condition-distrib* [**where** *X*=λ-. *L2-fail*]

lemma *L2-seq-condition-skip-throw* [*L2opt*]: *L2-seq*
(*L2-condition c* (*L2-gets* (λ-. ()) *ns*)
(*L2-throw x ms*))
(λ*r*. *y*) =
(*L2-condition c y*
(*L2-throw x ms*))
by (*simp add*: *L2-condition-distrib L2-seq-skip L2-trim-after-throw*)

lemma *L2-fail-propagate-catch* [*L2opt*]:
(*L2-seq* (*L2-catch L R*) (λ-. *L2-fail*)) = (*L2-catch* (*L2-seq L* (λ-. *L2-fail*)) (λ*e*. *L2-seq* (*R e*) (λ-. *L2-fail*)))
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff default-option-def Exn-def*, *safe*)
apply (*auto simp add*: *runs-to-def-old*)
done

lemma *L2-condition-fail-lhs* [*L2opt*]:
L2-condition C L2-fail A = *L2-seq* (*L2-guard* (λ*s*. ¬ *C s*)) (λ-. *A*)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff default-option-def Exn-def*)
done

lemma *L2-condition-fail-rhs* [*L2opt*]:
L2-condition C A L2-fail = *L2-seq* (*L2-guard* (λ*s*. *C s*)) (λ-. *A*)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff default-option-def Exn-def*)
done

lemma *L2-catch-fail* [*L2opt*]: *L2-catch L2-fail A* = *L2-fail*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff default-option-def Exn-def*)
done

lemma *L2-try-fail* [*L2opt*]: *L2-try L2-fail* = *L2-fail*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff default-option-def Exn-def*)
done

lemma *L2-while-fail* [*L2opt*]: *L2-while C* (λ-. *L2-fail*) *i n* = (*L2-seq* (*L2-guard* (λ*s*. ¬ *C i s*)) (λ-. *L2-gets* (λ*s*. *i*) *n*))
unfolding *L2-defs*

apply (*rule spec-monad-ext*)
apply (*subst whileLoop-unroll*)
apply (*auto simp add*: *run-condition run-bind run-guard*)
done

lemma *unit-bind*: $(\lambda x.\ f\ (x::unit)) = (\lambda\text{-}.\ f\ ())$
apply (*rule ext*)
subgoal for *x* **by** (*cases x*, *auto*)
done

lemma *unit-bind'*: $f \equiv (\lambda\text{-}.\ f\ ())$
by (*simp add*: *unit-bind*)

lemma *L2-while-cong*:
assumes *c-eq*: $\bigwedge r\ s.\ c\ r\ s = c'\ r\ s$
assumes *bdy-eq*: $\bigwedge r\ s.\ c'\ r\ s \Longrightarrow run\ (A\ r)\ s = run\ (A'\ r)\ s$
shows *L2-while c A* $=$ *L2-while c' A'*
apply (*rule ext*)+
unfolding *L2-while-def*
using *whileLoop-cong c-eq bdy-eq*
apply *metis*
done

lemma *L2-while-simp-cong*:
assumes *c-eq*: $\bigwedge r\ s.\ c\ r\ s = c'\ r\ s$
assumes *bdy-eq*[*simplified simp-implies-def*]: $\bigwedge r\ s.\ c'\ r\ s \Longrightarrow run\ (A\ r)\ s = run\ (A'\ r)\ s$
shows *L2-while c A* $=$ *L2-while c' A'*
apply (*rule L2-while-cong*)
apply (*simp add*: *c-eq*)
apply (*simp add*: *bdy-eq*)
done

lemma *L2-while-cong'*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: $\bigwedge r\ s.\ c'\ r\ s \Longrightarrow run\ (A\ r)\ s = run\ (A'\ r)\ s$
shows *L2-while c A* $=$ *L2-while c' A'*
apply (*rule L2-while-cong*)
apply (*simp add*: *c-eq*)
apply (*simp add*: *bdy-eq*)
done

lemma *L2-while-simp-cong'*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*[*simplified simp-implies-def*]: $\bigwedge r\ s.\ c'\ r\ s \Longrightarrow run\ (A\ r)\ s = run\ (A'\ r)\ s$

shows *L2-while c A = L2-while c′ A′*
apply (*rule L2-while-cong*)
apply (*simp add*: *c-eq*)
apply (*simp add*: *bdy-eq*)
done

lemma *L2-while-cong-split*:
assumes *c-eq*: *c = c′*
assumes *bdy-eq*: *PROP SPLIT* (⋀*r s*. *c′ r s* ⟹*run* (*A r*) *s* = *run* (*A′ r*) *s*)
shows *L2-while c A = L2-while c′ A′*
apply (*rule L2-while-simp-cong′* [*OF c-eq*])
using *bdy-eq*
by (*simp add*: *SPLIT-def simp-implies-def*)

lemma *L2-while-cong-simp-split*:
assumes *c-eq*: *c = c′*
assumes *bdy-eq*: *PROP SPLIT* (⋀*r s*. *c′ r s* =*simp*=> *run* (*A r*) *s* = *run* (*A′ r*) *s*)
shows *L2-while c A = L2-while c′* (*ETA-TUPLED A′*)
apply (*rule L2-while-simp-cong′* [*OF c-eq*])
using *bdy-eq*
by (*simp add*: *SPLIT-def simp-implies-def ETA-TUPLED-def*)

lemma *L2-while-cong-split-stop*:
assumes *c-eq*: *c = c′*
assumes *bdy-eq*: *PROP SPLIT* (⋀*r s*. *c′ r s* ⟹ *run* (*A r*) *s* = *run* (*A′ r*) *s*)
shows *L2-while c A = STOP* (*L2-while c′ A′*)
apply (*simp add*: *STOP-def*)
apply (*rule L2-while-simp-cong′* [*OF c-eq*])
using *bdy-eq*
by (*simp add*: *SPLIT-def simp-implies-def*)

lemma *L2-while-cong-simp-split-stop*:
assumes *c-eq*: *c = c′*
assumes *bdy-eq*: *PROP SPLIT* (⋀*r s*. *c′ r s* =*simp*=> *run* (*A r*) *s* = *run* (*A′ r*) *s*)
shows *L2-while c A = STOP* (*L2-while c′ A′*)
apply (*simp add*: *STOP-def*)
apply (*rule L2-while-simp-cong′* [*OF c-eq*])
using *bdy-eq*
by (*simp add*: *SPLIT-def simp-implies-def*)

lemma *L2-while-cong″*:
assumes *c-eq*: *c = c′*
assumes *bdy-eq*: ⋀*r s*. *c′ r s* ⟹ *run* (*A r*) *s* = *run* (*A′ r*) *s*
assumes *i-eq*: *i = i′*
shows *L2-while c A i ns = L2-while c′ A′ i′ ns*
apply (*simp add*: *i-eq*)

using *c-eq bdy-eq L2-while-cong*
by *metis*

lemma *L2-while-cong-block*: *L2-while c b = L2-while c b*
by *simp*
lemma *L2-while-unbind-STOP*: $c \equiv c' \Longrightarrow b \equiv b' \Longrightarrow$ *L2-while c b i ns* $\equiv$ *STOP-UNBIND* (*L2-while c' b' i ns*)
by (*simp add*: *STOP-UNBIND-def*)

lemma *L2-returncall-trivial* [*L2opt*]:
$[\![\bigwedge s\ v.\ f\ v\ s = v\ ]\!] \Longrightarrow$ *L2-returncall x f = L2-call x*
unfolding *L2-defs L2-call-def L2-returncall-def*
apply (*rule ext*)+
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
by (*auto simp add*: *runs-to-def-old map-exn-def Exn-def default-option-def split*:*xval-splits*
)

lemma *L2-gets-unused*:
$[\![\bigwedge x\ y\ s.$ *run* (*B x*) *s = run* (*B y*) *s* $]\!] \Longrightarrow$ *L2-seq* (*L2-gets A n*) *B* = (*B undefined*)
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)
done

lemma *L2-gets-unbound*[*L2opt*]:
L2-seq (*L2-gets A n*) ($\lambda x.$ *f*) = *f*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)
done

lemma *L2-gets-bind*:
L2-seq (*L2-gets* (λ-. *x* :: *'var-type*) *n*) *f* = *f x*
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)
done

lemma *L2-gets-bind-stop-cong*:
L2-seq (*L2-gets* (λ-. *x*) *n*) *f* = *L2-seq* (*L2-gets* (λ-. *x*) *n*) *f*
by *simp*

lemma *L2-seq-stop-cong*:
L2-seq x y = *L2-seq x y*
by *simp*

lemma *L2-marked-seq-gets-apply*:
L2-seq-gets c n A ≡ A c
unfolding *L2-seq-gets-def*
apply (*rule eq-reflection*)
by (*rule L2-gets-bind*)

lemma *split-seq-assoc* [*L2opt*]:
(λ*x*. *L2-seq* (*case x of* (*a*, *b*) ⇒ *B a b*) (*G x*)) = (λ*x*. *case x of* (*a*, *b*) ⇒ (*L2-seq* (*B a b*) (*G x*)))
by (*rule ext*) *clarsimp*

lemma *whileLoop-succeeds-terminates-infinite′*:
assumes *run* (*whileLoop* (λ-. *C*) (λ*x*. *gets* (λ*s*. *B s x*)) *i*) *s* ≠ ⊤
shows *C s* ⟹ *False*
using *assms*
by (*induct rule*: *whileLoop-ne-top-induct*) (*auto simp*: *runs-to-iff*)

lemma *run-whileLoop-infinite′*: *run* (*whileLoop* (λ*i*. *C*)
(λ*x*. *gets* (λ*s*. *B s x*))
i)
s =
run (*guard* (λ*s*. ¬ *C s*) ≫= (λ-. *gets* (λ-. *i*))) *s*
proof (*cases C s*)
case *True*
show *?thesis*
apply (*rule antisym*)
subgoal
apply (*subst whileLoop-unroll*)
apply (*simp add*: *run-guard True run-bind*)
done
subgoal
proof −
have ¬ *run* (*whileLoop* (λ-. *C*) (λ*x*. *gets* (λ*s*. *B s x*)) *i*) *s* ≠ *top*
using *True whileLoop-succeeds-terminates-infinite′*[*of C B i s*] **by** *auto*
hence ¬ *succeeds* (*whileLoop* (λ-. *C*) (λ*x*. *gets* (λ*s*. *B s x*)) *i*) *s*
by (*simp add*: *succeeds-def*)
then show *?thesis*
by (*simp add*: *run-guard run-bind succeeds-def True top-post-state-def*)
qed
done
next
case *False*
then show *?thesis* **apply** (*subst whileLoop-unroll*) **by** (*simp add*: *run-bind run-guard*)
qed

lemma *whileLoop-infinite′*:

```
  whileLoop (λi. C)
              (λx. gets (λs. B s x))
             i
          =
        guard (λs. ¬ C s) >>= (λ-. gets (λ-. i))
  apply (rule spec-monad-ext)
  apply (rule run-whileLoop-infinite')
  done

lemma L2-while-infinite [L2opt]:
       L2-while (λi s. C s) (λx. L2-gets (λs. B s x) n') i n
              = (L2-seq (L2-guard (λs. ¬ C s)) (λ-. L2-gets (λ-. i) n))
  unfolding L2-defs
  by (rule whileLoop-infinite')



lemmas L2-gets-bind-c-exntype [L2opt] = L2-gets-bind [where 'var-type='gx c-exntype]

lemmas L2-gets-bind-unit [L2opt] = L2-gets-bind [where 'var-type=unit]

declare L2-voidcall-def [L2opt]
declare L2-modifycall-def [L2opt]
declare ucast-id [L2opt]
declare scast-id [L2opt]


declare singleton-iff [L2opt]



lemma L2-gets-L2-seq-if-P-1-0 [L2opt]:
    L2-seq (L2-gets (λs. if P s then 1 else 0) n) (λx. Q x)
       = (L2-seq (L2-gets P n) (λx. Q (if x then 1 else 0)))
  unfolding L2-defs
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done




lemma L2-guard-join-simple [L2opt]:
  L2-seq (L2-guard A) (λ-. L2-guard B) = L2-guard (λs. A s ∧ B s)
  unfolding L2-defs
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done
```

lemma *L2-guard-join-nested* [*L2opt*]:
L2-seq (*L2-guard A*) (λ-. *L2-seq* (*L2-guard B*) (λ-. *C*))
= *L2-seq* (*L2-guard* (λ*s*. *A s* ∧ *B s*)) (λ-. *C*)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-guarded-L2-gets*[*L2opt*]:
L2-guarded (λ-. *P*) (*L2-seq* (*L2-gets f ns*) *g*) = *L2-seq* (*L2-gets f ns*) (λ*x*. *L2-guarded* (λ-. *P*) (*g x*))
unfolding *L2-guarded-def L2-defs*
by (*rule spec-monad-eqI*) (*auto simp add*: *runs-to-iff*)
lemma *L2-guarded-L2-guard*[*L2opt*]:
L2-guarded (λ-. *P*) (*L2-seq* (*L2-guard Q*) *g*) = *L2-seq* (*L2-guard Q*) (λ*x*. *L2-guarded* (λ-. *P*) (*g x*))
unfolding *L2-guarded-def L2-defs*
by (*rule spec-monad-eqI*) (*auto simp add*: *runs-to-iff*)

lemma *unbind*:
assumes *eq*: ⋀*x*. *f x* = *g*
shows *f* = (λ-. *f ZERO*('*a*::*c-type*))
apply (*rule ext*)
apply (*simp add*: *eq*)
done

lemma *L2-seq-unknown-block-cong*: *L2-seq-unknown ns X* = *L2-seq-unknown ns X*
by (*rule refl*)

lemma *L2-seq-unknown-STOP*: *f* ≡ *f'* ⟹ *L2-seq-unknown ns f* ≡ *STOP* (*L2-seq* (*L2-unknown ns*) *f'*)
by (*simp add*: *STOP-def L2-seq-unknown-def*)

lemma *L2-seq-unknown-unfold-STOP*: *f* ≡ (λ-. *g*) ⟹ *L2-seq-unknown ns f* ≡ *STOP g*
by (*simp add*: *L2-seq-unknown-def STOP-def L2-unknown-bind-unbound*)

lemma *L2-seq-gets-unfold*: *L2-seq-gets c ns f* = *f c*
by (*simp add*: *L2-seq-gets-def L2-gets-bind*)

lemma *L2-condition-L2-seq-distrib*:
L2-seq (*L2-condition C L R*) (λ*x*. *L2-seq* (*X x*) *Y*) ≡
L2-seq
(*L2-condition C*
(*L2-seq L X*)
(*L2-seq R X*)) *Y*
by (*simp add*: *L2-condition-distrib L2-seq-assoc*)

lemma *L2-condition-L2-seq-gets-distrib*:
L2-seq (*L2-condition C L R*) (λ*x*. *STOP* (*L2-seq-gets* (*X x*) *n Y*)) ≡
L2-seq
(*L2-condition C*
(*L2-seq L* (λ*x*. *L2-gets* (λ-. *X x*) *n*))
(*L2-seq R* (λ*x*. *L2-gets* (λ-. *X x*) *n*))) *Y*
unfolding *L2-seq-gets-def STOP-def*
by (*rule L2-condition-L2-seq-distrib*)

lemma *L2-condition-L2-seq-gets-distrib′*:
assumes *asm*: ⋀*x*. *STOP* (*L2-seq-gets* (*X x*) *n* (*Y x*)) ≡ *L2-seq* (*A x*) *B*
shows *L2-seq* (*L2-condition C L R*) (λ*x*. *STOP* (*L2-seq-gets* (*X x*) *n* (*Y x*))) ≡

L2-seq
(*L2-condition C*
(*FUSE* (*L2-seq L A*))
(*FUSE* (*L2-seq R A*))) *B*
unfolding *FUSE-def*
apply (*simp add*: *asm*)
by (*simp add*: *L2-condition-distrib L2-seq-assoc*)

lemma *L2-seq-condition-block-cong*: *L2-seq-condition c L R X* = *L2-seq-condition c L R X*
by (*rule refl*)

lemma *L2-seq-condition-unfold-STOP*:
L2-seq L X ≡ *L′* ⟹ *L2-seq R X* ≡ *R′* ⟹
L2-seq-condition c L R X ≡ *STOP* (*L2-condition c L′ R′*)
by (*simp add*: *L2-seq-condition-def L2-condition-distrib STOP-def*)

thm *L2-split-fixups*

thm *L2-condition-L2-seq-distrib* [*split-tuple X arity* : *3*, *simplified L2-split-fixups*]
thm *L2-condition-L2-seq-gets-distrib* [*split-tuple X arity* : *3*, *simplified L2-split-fixups*]
thm *L2-condition-L2-seq-gets-distrib′* [*split-tuple X* **and** *A arity* : *2*, *simplified L2-split-fixups*]
thm *L2-seq-assoc*
end

Chapter 20

IO phase: In/Out Parameters

20.1 Heap Typing for Split Heap

theory *TypHeapSimple*
imports
AutoCorres-Base
begin

20.2 Valid root footprint

lemma *c-null-guard-size-of-limit*:
c-null-guard (*p*::'*a*::*c-type ptr*) ⟹ *size-td* (*typ-uinfo-t* (*TYPE*('*a*))) < 2 ^ *len-of* *TYPE*(*addr-bitsize*)
unfolding *c-null-guard-def size-of-def*
by (*metis* (*no-types*, *opaque-lifting*) *add.commute add-diff-cancel-left' add-lessD1*

cancel-comm-monoid-add-class.diff-cancel first-in-intvl nat-less-le nat-neq-iff
not-add-less1 typ-uinfo-size unat-lt2p unsigned-eq-0-iff zero-not-in-intvl-no-overflow)

lemma *root-ptr-valid-legacy-def*: *root-ptr-valid d p* ⟷
valid-root-footprint d (*ptr-val* (*p*::'*a ptr*)) (*typ-uinfo-t TYPE*('*a*)) ∧
valid-simple-footprint d (*ptr-val* (*p*::'*a*::*c-type ptr*)) (*typ-uinfo-t TYPE*('*a*)) ∧
d, *c-guard* ⊨*t p*
using *c-null-guard-size-of-limit* [*of p*]
by (*auto simp add*: *root-ptr-valid-def c-guard-def addr-card-len-of-conv*
h-t-valid-def valid-root-footprint-valid-footprint intro!: *valid-root-footprint-valid-simple-footprint*)

definition
simple-lift :: *heap-raw-state* ⇒ ('*a*::*c-type*) *ptr* ⇒ '*a option*
where
simple-lift s p = (
if (*root-ptr-valid* (*hrs-htd s*) *p*) *then*

```
    (Some (h-val (hrs-mem s) p))
   else
    None)

lemma simple-lift-root-ptr-valid:
  simple-lift s p = Some x ⟹ root-ptr-valid (hrs-htd s) p
  apply (clarsimp simp: simple-lift-def split: if-split-asm)
  done

lemma simple-lift-h-t-valid:
  simple-lift s p = Some x ⟹ (hrs-htd s), c-guard ⊨ₜ p
  apply (clarsimp simp: simple-lift-def root-ptr-valid-legacy-def split: if-split-asm)
  done

lemma simple-lift-valid-root-footprint:
   simple-lift s (p::'a::c-type ptr) = Some x ⟹ valid-root-footprint (hrs-htd s)
(ptr-val p) (typ-uinfo-t (TYPE('a)))
  apply (clarsimp simp: simple-lift-def root-ptr-valid-def split: if-split-asm)
  done

lemma simple-lift-c-guard:
  assumes lift: simple-lift s p = Some x
  shows c-guard p
  using simple-lift-h-t-valid [OF lift]
  by (simp add: h-t-valid-def)




lemma root-ptr-valid-heap-update-other:
  assumes val-p: root-ptr-valid d (p::'a::mem-type ptr)
      and val-q: root-ptr-valid d (q::'b::c-type ptr)
      and neq: ptr-val p ≠ ptr-val q
  shows h-val (heap-update p v h) q = h-val h q
  apply (clarsimp simp: h-val-def heap-update-def)
  apply (subst heap-list-update-disjoint-same)
   apply simp
   apply (rule root-ptr-valid-neq-disjoint [OF val-p val-q neq])
  apply simp
  done


lemma root-ptr-valid-heap-update-other-typ:
  assumes val-p: root-ptr-valid d (p::'a::mem-type ptr)
      and val-q: root-ptr-valid d (q::'b::c-type ptr)
      and neq: typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b)
  shows h-val (heap-update p v h) q = h-val h q
  apply (clarsimp simp: h-val-def heap-update-def)
```

apply (*subst heap-list-update-disjoint-same*)
apply *simp*
apply (*rule root-ptr-valid-type-neq-disjoint* [*OF val-p val-q neq*])
apply *simp*
done

lemma *simple-lift-heap-update*:
⟦ *root-ptr-valid* (*hrs-htd h*) *p* ⟧ ⟹
simple-lift (*hrs-mem-update* (*heap-update p v*) *h*)
= (*simple-lift h*)(*p* := *Some* (*v*::*'a*::*mem-type*))
apply (*rule ext*)
apply (*clarsimp simp*: *simple-lift-def hrs-mem-update h-val-heap-update*)
apply (*fastforce simp*: *root-ptr-valid-heap-update-other*)
done

lemma *simple-lift-heap-update-other*:
⟦ *root-ptr-valid* (*hrs-htd d*) (*p*::*'b*::*mem-type ptr*);
typ-uinfo-t TYPE(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*) ⟧ ⟹
simple-lift (*hrs-mem-update* (*heap-update p v*) *d*) = ((*simple-lift d*)::*'a*::*c-type typ-heap*)
apply (*rule ext*)+
apply (*clarsimp simp*: *simple-lift-def h-val-heap-update hrs-mem-update*)
apply (*auto intro*: *root-ptr-valid-heap-update-other-typ*)
done

lemma *h-val-simple-lift*:
simple-lift h p = *Some v* ⟹ *h-val* (*hrs-mem h*) *p* = *v*
apply (*clarsimp simp*: *simple-lift-def split*: *if-split-asm*)
done

lemma *the-simple-lift-h-val-conv*:
root-ptr-valid (*hrs-htd h*) *p* ⟹ *the* (*simple-lift h p*) = *h-val* (*hrs-mem h*) *p*
apply (*clarsimp simp*: *simple-lift-def split*: *if-split-asm*)
done

lemma *slift-clift-Some-same*:
assumes *slift*: *simple-lift s p* = *Some x*
assumes *clift*: *clift s p* = *Some y*
shows *x* = *y*
using *h-val-simple-lift* [*OF slift*] *h-val-clift'* [*OF clift*]
by *simp*

lemma *simple-lift-Some-clift-Some*:
assumes *slift*: *simple-lift s p* = *Some x*
shows *clift s p* = *Some x*

using *simple-lift-h-t-valid* [*OF slift*] *h-t-valid-clift-Some-iff slift-clift-Some-same* [*OF slift*]
by (*cases s*) *auto*

lemma *h-val-field-simple-lift*:
⟦ *simple-lift h* (*pa* :: *'a ptr*) = *Some* (*v*::*'a*::*mem-type*);
field-ti TYPE('*a*) *f* = *Some t*;
export-uinfo (*the* (*field-ti TYPE*('*a*) *f*)) = *export-uinfo* (*typ-info-t TYPE*('*b* :: *mem-type*)) ⟧ ⟹
h-val (*hrs-mem h*) (*Ptr* &(*pa*→*f*) :: *'b* :: *mem-type ptr*) = *from-bytes* (*access-ti*$_0$ (*the* (*field-ti TYPE*('*a*) *f*)) *v*)
apply (*clarsimp simp*: *simple-lift-def split*: *if-split-asm*)
apply (*clarsimp simp*: *h-val-field-from-bytes*)
done

lemma *simple-lift-heap-update'*:
simple-lift h p = *Some v'* ⟹
simple-lift (*hrs-mem-update* (*heap-update* (*p*::(*'a*::{*mem-type*}) *ptr*) *v*) *h*)
= (*simple-lift h*)(*p* := *Some v*)
apply (*rule simple-lift-heap-update*)
apply (*erule simple-lift-root-ptr-valid*)
done

lemma *simple-lift-hrs-mem-update-None* [*simp*]:
(*simple-lift* (*hrs-mem-update a hp*) *x* = *None*) = (*simple-lift hp x* = *None*)
apply (*clarsimp simp*: *simple-lift-def*)
done

lemma *simple-lift-hrs-mem-update-Some*: (∃ *z*. *simple-lift* (*hrs-mem-update upd h*) *x* = *Some z*)
⟷ (∃ *z*. *simple-lift h x* = *Some z*)

apply (*cases simple-lift h x*)
apply *simp*
apply (*cases simple-lift* (*hrs-mem-update upd h*) *x*)
apply (*auto*)
done

lemma *clift-hrs-mem-update-None* [*simp*]:
(*clift* (*hrs-mem-update a hp*) *x* = *None*) = (*clift hp x* = *None*)
using *h-t-valid-clift-Some-iff*
apply (*cases hp*)
apply (*clarsimp simp add*: *hrs-mem-update*)
apply (*metis hrs-htd-def hrs-htd-mem-update lift-t-if option.distinct*(*1*) *prod.collapse*)+
done

lemma *clift-hrs-mem-update-Some*:
(∃ *z*. *clift* (*hrs-mem-update a hp*) *x* = *Some z*) = (∃ *z*. *clift hp x* = *Some z*)

apply (*cases clift hp x*)
apply *simp*
apply (*cases clift* (*hrs-mem-update a hp*) *x*)
apply (*auto*)
done

lemma *simple-lift-data-eq*:
⟦ *h-val* (*hrs-mem h*) *p* = *h-val* (*hrs-mem h'*) *p'*;
root-ptr-valid (*hrs-htd h*) *p* = *root-ptr-valid* (*hrs-htd h'*) *p'* ⟧ ⟹
simple-lift h p = *simple-lift h' p'*
apply (*clarsimp simp*: *simple-lift-def*)
done

lemma *h-val-heap-update-disjoint*:
⟦ {*ptr-val p* ..+ *size-of TYPE*('*a*::*c-type*)}
∩ {*ptr-val q* ..+ *size-of TYPE*('*b*::*mem-type*)} = {} ⟧ ⟹
h-val (*heap-update* (*q* :: '*b ptr*) *r h*) (*p* :: '*a ptr*) = *h-val h p*
apply (*clarsimp simp*: *h-val-def*)
apply (*clarsimp simp*: *heap-update-def*)
apply (*subst heap-list-update-disjoint-same*)
apply *clarsimp*
apply *blast*
apply *clarsimp*
done

lemma *update-ti-t-valid-size*:
size-of TYPE('*b*) = *size-td t* ⟹
update-ti-t t (*to-bytes-p* (*val*::'*b*::*mem-type*)) *obj* = *update-ti t* (*to-bytes-p val*) *obj*
apply (*clarsimp simp*: *update-ti-t-def to-bytes-p-def*)
done

lemma *h-val-field-from-bytes'*:
⟦ *field-ti TYPE*('*a*::{*mem-type*}) *f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*('*b*::{*mem-type*})) ⟧ ⟹
h-val h (*Ptr* &(*pa*→*f*) :: '*b ptr*) = *from-bytes* (*access-ti*$_0$ *t* (*h-val h pa*))
apply (*insert h-val-field-from-bytes*[**where** *f*=*f* **and** *pa*=*pa* **and** *t*=*t* **and** *h*=(*h*,*x*)
and '*a*='*a* **and** '*b*='*b* **for** *x*])
apply (*clarsimp simp*: *hrs-mem-def*)
done

lemma *h-val-field-from-root*:
fixes *p*::'*a*:: *mem-type ptr*
assumes *fl*:
field-lookup (*typ-info-t TYPE*('*a*::{*mem-type*})) *f* 0 =
Some (*adjust-ti* (*typ-info-t TYPE*('*b*::*mem-type*)) *fld fld-update*, *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update*)
shows *h-val h* (*PTR*('*b*) &(*p*→*f*)) = *fld* (*h-val h p*)

proof −
from *fl*
have *fl*: *field-ti TYPE('a) f = Some (adjust-ti (typ-info-t TYPE('b::mem-type)) fld fld-update)*
using *field-lookup-field-ti* **by** *blast*
have *exp*: *export-uinfo (adjust-ti (typ-info-t TYPE('b)) fld fld-update) = export-uinfo (typ-info-t TYPE('b))*
using *fg-cons*
by *simp*
from *h-val-field-from-bytes'* [*OF fl exp*]
show *?thesis*
by *simp*
qed

lemma *simple-lift-super-field-update-lookup*:
fixes *dummy* :: *'b* :: *mem-type*
assumes *field-lookup (typ-info-t TYPE('b::mem-type)) f 0 = Some (s,n)*
and *typ-uinfo-t TYPE('a) = export-uinfo s*
and *simple-lift h p = Some v'*
shows (*super-field-update-t (Ptr (&(p→f))) (v::'a::mem-type) ((simple-lift h)::'b ptr ⇒ 'b option)) =*
((simple-lift h)(p ↦ field-update (field-desc s) (to-bytes-p v) v'))
proof −
from *assms* **have** [*simp*]: *unat (of-nat n :: addr) = n*
apply (*subst unat-of-nat*)
apply (*subst mod-less*)
apply (*drule td-set-field-lookupD*)+
apply (*drule td-set-offset-size*)+
apply (*subst len-of-addr-card*)
apply (*subst (asm) size-of-def* [*symmetric*, **where** *t=TYPE('b)*])+
apply (*subgoal-tac size-of TYPE('b) < addr-card*)
apply *arith*
apply *simp*
apply *simp*
done
from *assms* **show** *?thesis*
supply *unsigned-of-nat* [*simp del*]
apply (*clarsimp simp*: *super-field-update-t-def*)
apply (*rule ext*)
apply (*clarsimp simp*: *field-lvalue-def split*: *option.splits*)
apply (*safe*, *simp-all*)
apply (*frule-tac v=v* **and** *v'=v'* **in** *update-field-update*)
apply (*clarsimp simp*: *field-of-t-def field-of-def typ-uinfo-t-def*)
apply (*frule-tac m=0* **in** *field-names-SomeD2*)
apply *simp*
apply *clarsimp*
apply (*simp add*: *field-typ-def field-typ-untyped-def*)
apply (*frule field-lookup-export-uinfo-Some*)
apply (*frule-tac s=k* **in** *field-lookup-export-uinfo-Some*)

```
    apply simp
    apply (drule (1) field-lookup-inject)
     apply (subst typ-uinfo-t-def [symmetric, where t=TYPE('b)])
     apply simp
    apply simp
   apply (drule field-of-t-mem)+
   apply (cases h)
   apply (clarsimp simp: simple-lift-def split: if-split-asm)
   apply (drule (1) root-ptr-valid-neq-disjoint)
    apply simp
   apply fast
  apply (clarsimp simp: field-of-t-def field-of-def)
  apply (subst (asm) td-set-field-lookup)
   apply simp
  apply simp
  apply (frule field-lookup-export-uinfo-Some)
  apply (simp add: typ-uinfo-t-def)
 apply (clarsimp simp: field-of-t-def field-of-def)
 apply (subst (asm) td-set-field-lookup)
  apply simp
 apply simp
 apply (frule field-lookup-export-uinfo-Some)
 apply (simp add: typ-uinfo-t-def)
 done
qed

lemma field-offset-addr-card:
  ∃x. field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some x
    ⟹ field-offset TYPE('a) f < addr-card
  apply (clarsimp simp: field-offset-def field-offset-untyped-def typ-uinfo-t-def)
  apply (subst field-lookup-export-uinfo-Some)
   apply assumption
  apply (frule td-set-field-lookupD)
  apply (drule td-set-offset-size)
  apply (insert max-size [where ?'a='a])
  apply (clarsimp simp: size-of-def)
  done

lemma unat-of-nat-field-offset:
  ∃x. field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some x ⟹
      unat (of-nat (field-offset TYPE('a) f)  :: addr) = field-offset TYPE('a) f
  apply (subst word-unat.Abs-inverse)
   apply (clarsimp simp: unats-def)
   apply (insert field-offset-addr-card [where f=f and ?'a='a])[1]
   apply (fastforce simp: addr-card)
  apply simp
  done

lemma field-of-t-field-lookup:
```

```
  assumes a: field-lookup (typ-info-t TYPE('a::mem-type)) f 0 = Some (s, n)
  assumes b: export-uinfo s = typ-uinfo-t TYPE('b::mem-type)
  assumes n: n = field-offset TYPE('a) f
  shows field-of-t (Ptr &(ptr→f) :: ('b ptr)) (ptr :: 'a ptr)
  supply unsigned-of-nat [simp del]
  apply (clarsimp simp del: field-lookup-offset-eq
      simp: field-of-t-def field-of-def)
  apply (subst td-set-field-lookup)
   apply (rule wf-desc-typ-tag)
  apply (rule exI [where x=f])
  using a[simplified n] b
  apply (clarsimp simp: typ-uinfo-t-def)
  apply (subst field-lookup-export-uinfo-Some)
   apply assumption
  apply (clarsimp simp del: field-lookup-offset-eq
      simp: field-lvalue-def unat-of-nat-field-offset)
  done

lemma simple-lift-field-update':
  fixes val :: 'b :: mem-type and ptr :: 'a :: mem-type ptr
  assumes   fl: field-lookup (typ-info-t TYPE('a)) f 0 =
                    Some (adjust-ti (typ-info-t TYPE('b)) xf xfu, n)
  and    xf-xfu: fg-cons xf xfu
  and        cl: simple-lift hp ptr = Some z
  shows (simple-lift (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) =
              (simple-lift hp)(ptr ↦ xfu val z)
    (is ?LHS = ?RHS)
proof (rule ext)
  fix p

  have eui: typ-uinfo-t TYPE('b) =
       export-uinfo (adjust-ti (typ-info-t TYPE('b)) xf xfu)
    using xf-xfu
    apply (subst export-tag-adjust-ti2 [OF - wf-lf wf-desc])
    apply (simp add: fg-cons-def )
    apply (rule meta-eq-to-obj-eq [OF typ-uinfo-t-def])
    done

  have n-is-field-offset: n = field-offset TYPE('a) f
    apply (insert field-lookup-offset-eq [OF fl])
    apply (clarsimp)
    done

  have equal-case: ?LHS ptr = ?RHS ptr
    supply unsigned-of-nat [simp del]
    apply (insert cl)
    apply (clarsimp simp: simple-lift-def split: if-split-asm)
    apply (clarsimp simp: hrs-mem-update)
    apply (subst h-val-super-update-bs)
```

```
    apply (rule field-of-t-field-lookup [OF fl])
     apply (clarsimp simp: eui)
    apply (clarsimp simp: n-is-field-offset)
   apply clarsimp
   apply (unfold from-bytes-def)
  apply (subst fi-fu-consistentD [where f=f and s=adjust-ti (typ-info-t TYPE('b))
xf xfu])
       apply (clarsimp simp: fl)
       apply (clarsimp simp: n-is-field-offset field-lvalue-def)
       apply (metis unat-of-nat-field-offset fl)
      apply clarsimp
     apply (clarsimp simp: size-of-def)
    apply (clarsimp simp: size-of-def)
   apply clarsimp
   apply (subst update-ti-s-from-bytes)
    apply clarsimp
   apply (subst update-ti-t-adjust-ti)
    apply (rule xf-xfu)
   apply (subst update-ti-s-from-bytes)
    apply clarsimp
   apply clarsimp
   apply (clarsimp simp: h-val-def)
   done

 show ?LHS p = ?RHS p
   apply (cases p = ptr)
    apply (erule ssubst)
    apply (rule equal-case)
   apply (insert cl)
   apply (clarsimp simp: simple-lift-def hrs-mem-update split: if-split-asm)
   apply (rule h-val-heap-update-disjoint)
   apply (insert field-tag-sub [OF fl, where p=ptr])
   apply (clarsimp simp: size-of-def)
   apply (clarsimp simp: root-ptr-valid-legacy-def)
   apply (frule (1) valid-simple-footprint-neq-disjoint, fastforce)
   apply clarsimp
   apply blast
   done
qed

lemma simple-lift-field-update:
 fixes val :: 'b :: mem-type and ptr :: 'a :: mem-type ptr
 assumes   fl: field-ti TYPE('a) f =
                  Some (adjust-ti (typ-info-t TYPE('b)) xf (xfu o (λx -. x)))
 and    xf-xfu: fg-cons xf (xfu o (λx -. x))
 and        cl: simple-lift hp ptr = Some z
 shows (simple-lift (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) =
            (simple-lift hp)(ptr ↦ xfu (λ-. val) z)
   (is ?LHS = ?RHS)
```

apply (*insert fl* [*unfolded field-ti-def*])
apply (*clarsimp split*: *option.splits*)
apply (*subst simple-lift-field-update'* [**where** *xf*=*xf* **and** *xfu*=*xfu o* (λ*x* -. *x*) **and** *z*=*z*])
apply (*clarsimp simp*: *o-def split*: *option.splits*)
apply (*rule refl*)
apply (*rule xf-xfu*)
apply (*rule cl*)
apply *clarsimp*
done

lemma *simple-heap-diff-types-impl-diff-ptrs*:
⟦ *root-ptr-valid h* (*p*::('*a*::*c-type*) *ptr*);
root-ptr-valid h (*q*::('*b*::*c-type*) *ptr*);
typ-uinfo-t TYPE('*a*) ≠ *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
ptr-val p ≠ *ptr-val q*
apply (*clarsimp simp*: *root-ptr-valid-legacy-def*)
apply (*clarsimp simp*: *valid-simple-footprint-def*)
done

lemma *h-val-update-regions-disjoint*:
⟦ { *ptr-val p* ..+ *size-of TYPE*('*a*) } ∩ { *ptr-val x* ..+ *size-of TYPE*('*b*)} = {} ⟧
⟹
h-val (*heap-update p* (*v*::('*a*::*mem-type*)) *h*) *x* = *h-val h* (*x*::('*b*::*c-type*) *ptr*)
apply (*clarsimp simp*: *heap-update-def*)
apply (*clarsimp simp*: *h-val-def*)
apply (*subst heap-list-update-disjoint-same*)
apply *clarsimp*
apply *clarsimp*
done

lemma *h-val-heap-update-padding-disjoint*:
fixes *p*::'*a*::*mem-type ptr*
fixes *q*::'*b*::*c-type ptr*
shows ⟦ *ptr-span p* ∩ *ptr-span q* = {}; *length bs* = *size-of TYPE*('*a*) ⟧ ⟹
h-val (*heap-update-padding p v bs h*) *q* = *h-val h q*
apply (*clarsimp simp*: *heap-update-padding-def*)
apply (*clarsimp simp*: *h-val-def*)
apply (*subst heap-list-update-disjoint-same*)
apply *clarsimp*
apply *clarsimp*
done

lemma *simple-lift-field-update-t*:
fixes *val* :: '*b* :: *mem-type* **and** *ptr* :: '*a* :: *mem-type ptr*
assumes *fl*: *field-ti TYPE*('*a*) *f* = *Some t*
and *diff*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*c* :: *c-type*)
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*('*b*))

and *cl*: *simple-lift hp ptr* = *Some z*
shows ((*simple-lift* (*hrs-mem-update* (*heap-update* (*Ptr* &(*ptr*→*f*)) *val*) *hp*)) :: *'c ptr* ⇒ *'c option*) =
simple-lift hp
apply (*rule ext*)
subgoal for *x*
apply (*cases simple-lift hp x*)
apply *clarsimp*
apply (*cases ptr-val x* = *ptr-val ptr*)
apply *clarsimp*
apply (*clarsimp simp*: *simple-lift-def hrs-mem-update split*: *if-split-asm*)
apply (*cut-tac simple-lift-root-ptr-valid* [*OF cl*])
apply (*drule* (*1*) *simple-heap-diff-types-impl-diff-ptrs* [*OF - - diff*])
apply *simp*
apply (*clarsimp simp*: *simple-lift-def hrs-mem-update split*: *if-split-asm*)
apply (*rule field-ti-field-lookupE* [*OF ft*])
apply (*frule-tac p*=*ptr* **in** *field-tag-sub*)
apply (*clarsimp simp*: *h-val-def heap-update-def*)
apply (*subst heap-list-update-disjoint-same*)
apply *clarsimp*
apply (*cut-tac simple-lift-root-ptr-valid* [*OF cl*])
apply (*drule* (*2*) *root-ptr-valid-neq-disjoint*)
apply (*clarsimp simp*: *export-size-of* [*unfolded typ-uinfo-t-def*, *OF eu*])
apply *blast*
apply *simp*
done
done

lemma *simple-lift-heap-update-other'*:
⟦ *simple-lift h* (*p*::*'b*::*mem-type ptr*) = *Some v'*;
typ-uinfo-t TYPE(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*) ⟧ ⟹
simple-lift (*hrs-mem-update* (*heap-update p v*) *h*) = ((*simple-lift h*)::*'a*::*c-type typ-heap*)
apply (*rule simple-lift-heap-update-other*)
apply (*erule simple-lift-root-ptr-valid*)
apply *simp*
done

lemma *simple-lift-heap-update-bytes-in-other*:
⟦ *simple-lift h* (*p*::*'b*::*mem-type ptr*) = *Some v'*;
typ-uinfo-t TYPE(*'b*) ≠ *typ-uinfo-t TYPE*(*'c*);
{ *ptr-val q* ..+ *size-of TYPE*(*'a*)} ⊆ {*ptr-val p* ..+ *size-of TYPE*(*'b*) } ⟧ ⟹
simple-lift (*hrs-mem-update* (*heap-update* (*q*::*'a*::*mem-type ptr*) *v*) *h*) = ((*simple-lift h*)::*'c*::*mem-type typ-heap*)
apply (*rule ext*)

apply (*clarsimp simp*: *simple-lift-def split*: *if-split-asm*)
apply (*drule* (*1*) *root-ptr-valid-type-neq-disjoint*, *simp*)
apply (*clarsimp simp*: *hrs-mem-update*)
apply (*rule h-val-heap-update-disjoint*)
apply *blast*
done

lemma *typ-name-neq*:
typ-name (*export-uinfo* (*typ-info-t TYPE*('*a*::*c-type*)))
≠ *typ-name* (*export-uinfo* (*typ-info-t TYPE*('*b*::*c-type*)))
⟹ *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
apply (*metis typ-uinfo-t-def*)
done

lemma *of-nat-mod-div-decomp*:
of-nat k
= *of-nat* (*k div size-of TYPE*('*b*)) ∗ *of-nat* (*size-of TYPE*('*b*::*mem-type*)) +
of-nat (*k mod size-of TYPE*('*b*))
by (*metis mod-div-decomp of-nat-add of-nat-mult*)

lemma *c-guard-array-c-guard*:
⟦ ⋀*x*. *x* < *CARD*('*a*) ⟹ *c-guard* (*ptr-coerce p* +$_p$ *int x* :: '*b ptr*) ⟧ ⟹ *c-guard*
(*p* :: ('*b* :: *mem-type*, '*a* :: *finite*) *array ptr*)
apply *atomize*
apply (*clarsimp simp*: *c-guard-def*)
apply (*rule conjI*)
apply (*drule-tac x=0* **in** *spec*)
apply (*clarsimp simp*: *ptr-aligned-def align-of-def align-td-array*)
apply (*simp add*: *c-null-guard-def*)
apply (*clarsimp simp*: *intvl-def*)
apply (*drule-tac x=k div size-of TYPE*('*b*) **in** *spec*)
apply (*erule impE*)
apply (*metis* (*full-types*) *less-nat-zero-code mult-is-0 neq0-conv td-gal-lt*)
apply *clarsimp*
apply (*drule-tac x=k mod size-of TYPE*('*b*) **in** *spec*)
apply (*clarsimp simp*: *CTypesDefs.ptr-add-def*)
apply (*subst* (*asm*) *add.assoc*)
apply (*subst* (*asm*) *of-nat-mod-div-decomp* [*symmetric*])
apply *clarsimp*
done

lemma *heap-list-update-list'*:
⟦ *n* + *x* ≤ *length v*; *length v* < *addr-card*; *q* = (*p* + *of-nat x*) ⟧ ⟹
heap-list (*heap-update-list p v h*) *n q* = *take n* (*drop x v*)
by (*metis heap-list-update-list*)

lemma *outside-intvl-range*:

$p \notin \{a\ ..+\ b\} \Longrightarrow p < a \vee p \geq a + of\text{-}nat\ b$
apply (*clarsimp simp*: *intvl-def not-le not-less*)
apply (*drule-tac x*=*unat* (*p*−*a*) **in** *spec*)
apply *clarsimp*
apply (*metis add-diff-cancel2 le-less-linear le-unat-uoi mpl-lem not-add-less2 unat-mono word-less-minus-mono-left*)
done

lemma *root-ptr-valid-intersect-array*:
$[\![\forall j < n.\ root\text{-}ptr\text{-}valid\ htd\ (p +_p int\ j);$
$root\text{-}ptr\text{-}valid\ htd\ (q :: ('a :: c\text{-}type)\ ptr)\ ]\!]$
$\Longrightarrow (\exists m < n.\ q = (p +_p int\ m))$
$\vee\ (\{ptr\text{-}val\ p\ ..+\ size\text{-}of\ TYPE\ ('a) * n\} \cap \{ptr\text{-}val\ q\ ..+\ size\text{-}of\ TYPE\ ('a :: c\text{-}type)\} = \{\})$
apply (*induct n*)
apply *clarsimp*
subgoal for *n*
apply *atomize*
apply *simp*
apply (*cases n* = *0*)
apply *clarsimp*
apply (*metis root-ptr-valid-neq-disjoint ptr-val-inj*)
apply (*erule disjE*)
apply (*metis less-Suc-eq*)
apply (*cases q* = *p* +*p int n*)
apply *force*
apply (*frule-tac x*=*n* **in** *spec*)
apply (*erule impE*, *simp*)
apply (*drule* (*1*) *root-ptr-valid-neq-disjoint*)
apply *simp*
apply (*simp add*: *CTypesDefs.ptr-add-def*)
apply (*rule disjI2*)
apply (*cut-tac a*= *of-nat n* ∗ *of-nat* (*size-of TYPE*('*a*))
and *p*=*ptr-val p* **and** *n*=*n* ∗ *size-of TYPE*('*a*) + *size-of TYPE*('*a*) **in** *intvl-split*)
apply *clarsimp*
apply (*clarsimp simp*: *field-simps Int-Un-distrib2*)
apply (*metis IntI emptyE intvl-empty intvl-inter intvl-self neq0-conv*)
done
done

lemmas *simple-lift-simps* =
typ-name-neq
simple-lift-c-guard
h-val-simple-lift
simple-lift-heap-update
simple-lift-heap-update-other

c-guard-field
h-val-field-simple-lift
simple-lift-field-update
simple-lift-field-update-t
c-guard-array-field

lemmas *typ-simple-heap-simps* = *simple-lift-simps*

lemma *valid-footprint-overlap-cases*:
assumes *valid-p*: *valid-footprint d* (*ptr-val* (*p*::'*a*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*b*))
assumes *overlap*: *ptr-span p* ∩ *ptr-span q* ≠ {}
shows *TYPE*('*a*) ≤$_\tau$ *TYPE*('*b*) ∨ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
proof (*cases typ-uinfo-t* (*TYPE*('*a*)) = *typ-uinfo-t* (*TYPE*('*b*)))
case *True*
thus *?thesis* **by** (*simp add*: *sub-typ-def*)
next
case *False*
note *not-eq* = *False*
show *?thesis*
proof (*cases TYPE*('*a*) ⊥$_\tau$ *TYPE*('*b*))
case *False*
thus *?thesis* **by** (*simp add*: *tag-disj-typ-def tag-disj-def sub-typ-def*)
next
case *True*
note *disj-types* = *True*

with *disj-types not-eq* **have** *ne-le-b-a*: ¬ *typ-uinfo-t* (*TYPE*('*b*)) < *typ-uinfo-t* (*TYPE*('*a*))
apply (*simp add*: *tag-disj-typ-def tag-disj-def*)
by (*meson le-less*)

have *not-field-of-p-q*: ¬ *field-of* (*ptr-val p* − *ptr-val q*) (*typ-uinfo-t* (*TYPE*('*a*))) (*typ-uinfo-t* (*TYPE*('*b*)))
apply (*rule ccontr*, *simp*)
apply (*drule field-of-sub*)
using *disj-types*
apply (*simp add*: *tag-disj-typ-def tag-disj-def*)
done

from *valid-footprint-neq-disjoint* [*OF valid-p valid-q ne-le-b-a not-field-of-p-q*]
have *ptr-span p* ∩ *ptr-span q* = {}
by (*simp add*: *size-of-def*)
with *overlap* **show** *?thesis* **by** *auto*

qed
qed

lemma *valid-root-footprint-valid-footprint-overlap-case*:
assumes *valid-p*: *valid-root-footprint d* (*ptr-val* (*p*::'*a*:: *mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*b*)))
assumes *overlap*: *ptr-span p* ∩ *ptr-span q* ≠ {}
shows *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
proof −
from *overlap* [*simplified size-of-def*, *folded typ-uinfo-t-def*]
have {*ptr-val p*..+*size-td* (*typ-uinfo-t TYPE*('*a*))} ∩ {*ptr-val q*..+*size-td* (*typ-uinfo-t TYPE*('*b*))} ≠ {}
by *simp*
from *valid-root-footprint-overlap-sub-typ* [*OF valid-p valid-q this*]
have *typ-uinfo-t TYPE*('*b*) ≤ *typ-uinfo-t TYPE*('*a*)**.**
thus *?thesis*
by (*simp add*: *sub-typ-def*)
qed

lemma *valid-root-footprint-overlap-contained*:
assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: {*x* ..+ *size-td t*} ∩ {*y* ..+ *size-td s*} ≠ {}
shows *y* ∈ {*x* ..+ *size-td t*}
using *valid-root-x overlap valid-y*
apply (*clarsimp simp add*: *valid-root-footprint-def valid-footprint-def Let-def typ-tag-le-def*)
by (*metis* (*no-types*, *opaque-lifting*) *intvl-inter intvl-inter-le le-less-trans less-irrefl overlap*
valid-footprint-sub2 valid-root-footprint-overlap-sub-typ valid-root-footprint-valid-footprint valid-root-x valid-y zero-le)

lemma *valid-footprint-field-of-contained*:
assumes *valid-x*: *valid-footprint d x t*
assumes *field*: *field-of off s t*
shows {*x* + *off* ..+ *size-td s*} ⊆ {*x* ..+ *size-td t*}
proof −
from *field* **have** (*s*, *unat off*) ∈ *td-set t 0*
by (*simp add*: *field-of-def*)
from *td-set-offset-size* [*OF this*] **have** *size-td s* + *unat off* ≤ *size-td t***.**
thus *?thesis*
by (*simp add*: *intvl-sub-offset*)
qed

lemma *valid-root-footprint-overlap-field-of*:

assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *y*: $y \in \{x ..+ \textit{size-td } t\}$
shows *field-of* $(y - x)$ *s t*
proof −
from *y* **have** $\{x ..+ \textit{size-td } t\} \cap \{y ..+ \textit{size-td } s\} \neq \{\}$
by (*meson disjoint-iff intvl-self valid-footprint-def valid-y*)
from *valid-root-footprint-overlap-sub-typ* [*OF valid-root-x valid-y this*]
have $s \leq t$ **.**
with *y* **show** *?thesis*
by (*meson le-less-trans less-irrefl valid-footprint-sub valid-root-footprint-valid-footprint valid-root-x valid-y*)
qed

lemma *valid-root-footprint-overlap-contained′*:
assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: $\{x ..+ \textit{size-td } t\} \cap \{y ..+ \textit{size-td } s\} \neq \{\}$
shows $\{y ..+ \textit{size-td } s\} \subseteq \{x ..+ \textit{size-td } t\}$
proof −
from *valid-root-footprint-overlap-contained* [*OF valid-root-x valid-y overlap*]
have $y \in \{x..+\textit{size-td } t\}$ **.**
from *valid-root-footprint-overlap-field-of* [*OF valid-root-x valid-y this*]
have *field-of* $(y - x)$ *s t* **.**
from *valid-footprint-field-of-contained* [*OF valid-root-footprint-valid-footprint* [*OF valid-root-x*] *this*]
have $\{x + (y - x)..+\textit{size-td } s\} \subseteq \{x..+\textit{size-td } t\}$ **.**
then show *?thesis*
by *auto*
qed

lemma *valid-footprint-sub-cases*:
assumes *valid-p*: *valid-footprint d p s*
assumes *valid-q*: *valid-footprint d q t*
assumes *sub*: $\neg\ t < s$
shows $\{p..+\textit{size-td } s\} \cap \{q..+\textit{size-td } t\} = \{\} \vee$ *field-of* $(p - q)$ (s) (t)
using *valid-footprint-neq-disjoint valid-p valid-q sub* **by** *blast*

lemma *valid-root-footprint-dist-type-cases*:
assumes *valid-p*: *valid-root-footprint d* (*ptr-val* (*p*::′*a*:: *mem-type ptr*)) (*typ-uinfo-t* (*TYPE*(′*a*)))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::′*b*::*mem-type ptr*)) (*typ-uinfo-t* (*TYPE*(′*b*)))
assumes *dist-type*: *typ-uinfo-t* (*TYPE*(′*a*)) ≠ *typ-uinfo-t* (*TYPE*(′*b*))
assumes *nested-case*: $\bigwedge f.\ f \in$ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*(′*a*))) (*typ-uinfo-t* (*TYPE*(′*b*)))) ⟹
field-lookup (*typ-uinfo-t TYPE*(′*a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*(′*b*), *unat* (*ptr-val q* − *ptr-val p*)) ⟹

field-of-t (*PTR*('*b*) &(*p*→*f*)) *p* ⟹
ptr-val q = &(*p*→*f*) ⟹
ptr-span q ⊆ *ptr-span p* ⟹ *P*
assumes *disj-case*: *ptr-span p* ∩ *ptr-span q* = {} ⟹ *P*
shows *P*
proof (*cases ptr-span p* ∩ *ptr-span q* = {})
case *True* **thus** *?thesis* **by** (*rule disj-case*)
next
case *False*
from *valid-root-footprint-valid-footprint-overlap-case* [*OF valid-p valid-q False*]
have *le-b-a*: *typ-uinfo-t TYPE*('*b*) ≤ *typ-uinfo-t TYPE*('*a*) **by** (*simp add*: *sub-typ-def*)

from *sub-typ-field-names-u-nonempty* [*OF this*]
obtain *f* **where** *f* ∈ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*('*a*))) (*typ-uinfo-t* (*TYPE*('*b*))))
by *fastforce*

from *False* **have** {*ptr-val p..+size-td* (*typ-uinfo-t TYPE*('*a*))} ∩ {*ptr-val q..+size-td* (*typ-uinfo-t TYPE*('*b*))} ≠ {}
by (*simp add*: *size-of-def*)
from *valid-root-footprint-overlap-contained'* [*OF valid-p valid-q this*]
have {*ptr-val q..+size-td* (*typ-uinfo-t TYPE*('*b*))} ⊆ {*ptr-val p..+size-td* (*typ-uinfo-t TYPE*('*a*))} .
hence *ptr-val q* ∈ {*ptr-val p..+size-td* (*typ-uinfo-t TYPE*('*a*))}
by (*simp add*: *size-of-tag subsetD*)
from *valid-root-footprint-overlap-field-of* [*OF valid-p valid-q this*]
have *field-of* (*ptr-val q* − *ptr-val p*) (*typ-uinfo-t TYPE*('*b*)) (*typ-uinfo-t TYPE*('*a*))
.
from *field-of-lookup-info* [*OF this*, *of p*]
obtain *f* **where**
f ∈ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*('*a*))) (*typ-uinfo-t* (*TYPE*('*b*))))
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f* 0 =
Some (*typ-uinfo-t TYPE*('*b*), *unat* (*ptr-val q* − *ptr-val p*))
field-of-t (*PTR*('*b*) &(*p*→*f*)) *p*
ptr-val q = &(*p*→*f*)
ptr-span q ⊆ *ptr-span p*
by (*auto simp add*: *size-of-def*)
from *nested-case* [*OF this*]
show *?thesis* .
qed

lemma *cvalid-field-pres*:
assumes *lookup*: *field-lookup* (*typ-uinfo-t TYPE*('*a*::*mem-type*)) *f* 0 = *Some* (*typ-uinfo-t TYPE*('*b*::*mem-type*), *n*)
assumes *valid*: *d*,*c-guard* ⊨$_t$ (*p*::'*a*:: *mem-type ptr*)
shows *d*, *c-guard* ⊨$_t$ *PTR*('*b*) &(*p*→*f*)
using *field-lookup-export-uinfo-Some-rev* [*OF lookup* [*simplified typ-uinfo-t-def*]]
apply (*clarsimp*)

apply (*rule h-t-valid-mono* [*rule-format*, *OF - - c-guard-mono valid*])
apply (*auto simp add*: *typ-uinfo-t-def*)
done

lemma *cvalid-field-pres'*:
assumes *ptr-val-eq*: *ptr-val q* = &(*p*→*f*)
assumes *lookup*: *field-lookup* (*typ-uinfo-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*::*mem-type*), *n*)
assumes *valid*: *d*,*c-guard* ⊨$_t$ (*p*::'*a*:: *mem-type ptr*)
shows *d*, *c-guard* ⊨$_t$ (*q*::'*b ptr*)
proof −
from *ptr-val-eq* **have** *q* = *PTR*('*b*) (&(*p*→*f*))
by (*cases q*) (*auto simp add*: *ptr-val-def*)
with *cvalid-field-pres* [*OF lookup valid*]
show *?thesis*
by *simp*
qed

lemma *cvalid-field-pres''*:
assumes *ptr-val-eq*: *ptr-val q* = &(*p*→*f*)
assumes *lookup*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*, *n*)
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*::*mem-type*)
assumes *valid*: *d*,*c-guard* ⊨$_t$ (*p*::'*a*:: *mem-type ptr*)
shows *d*, *c-guard* ⊨$_t$ (*q*::'*b ptr*)
proof −
from *cvalid-field-pres'* [*OF ptr-val-eq - valid*] *lookup match*
show *?thesis*
by (*simp add*: *UMM.field-lookup-typ-uinfo-t-Some*)
qed

lemma *cvalid-field-pres'''*:
assumes *lookup*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b*::*mem-type*)
assumes *valid*: *d*,*c-guard* ⊨$_t$ (*p*::'*a*:: *mem-type ptr*)
shows *d*, *c-guard* ⊨$_t$ *PTR*('*b*) &(*p*→*f*)
using *lookup match valid*
using *c-guard-mono h-t-valid-mono* **by** *blast*

lemma *the-clift-eq-h-val-eq*:
assumes *h-val-eq*: *hrs-htd hp* ⊨$_t$ *p* ⟹ *h-val* (*hrs-mem* (*hrs-mem-update a hp*)) *p* = *h-val* (*hrs-mem hp*) *p*
shows *the* (*clift* (*hrs-mem-update a hp*) *p*) = *the* (*clift hp p*)
proof (*cases clift hp p*)
case *None*
with *clift-hrs-mem-update-None* **show** *?thesis* **by** *metis*
next
case (*Some v*)

with *clift-hrs-mem-update-None* **obtain** v' **where** *clift* (*hrs-mem-update a hp*) *p* = *Some* v'
by (*cases clift* (*hrs-mem-update a hp*) *p*) *auto*
with *Some h-val-clift' h-val-eq* [*OF h-t-valid-clift* [*OF Some*]]
show *?thesis* **by** *metis*
qed

lemma *field-lvalue-same-conv*: &(p::'a:: *c-type ptr*→*f*) = &(q::'a:: *c-type ptr*→*f*) ⟹ *p* = *q*
by (*simp add*: *field-lvalue-def*)

lemma *ptr-val-field-convD*: *ptr-val p* = &(*q*→*f*) ⟹ *p* = *Ptr* &(*q*→*f*)
by (*cases p*) *auto*

lemma *ptr-val-field-conv*: *ptr-val p* = &(*q*→*f*) ⟷ *p* = *Ptr* &(*q*→*f*)
by (*cases p*) *auto*

lemma *ptr-val-array-index-convD*:
ptr-val p = *ptr-val* (*array-ptr-index q False j*) ⟹ *p* = *array-ptr-index q False j*
by (*cases p*) *auto*

lemma *ptr-val-conv'*: *ptr-coerce p* = *Ptr q* ⟷ *ptr-val p* = *q*
by (*cases p*) *simp*

lemma *ptr-val-conv*: *p* = *q* ⟷ *ptr-val p* = *ptr-val q*
by *auto*

lemma *ptr-val-neq-conv*: *p* ≠ *q* ⟷ *ptr-val p* ≠ *ptr-val q*
by *auto*

lemma *the-simple-lift-strong-eqI*:
fixes p::'a::*mem-type ptr*
fixes q::'b::*mem-type ptr*

assumes *eq*: ⋀*x1 x2*. *root-ptr-valid* (*hrs-htd s*) *q* ⟹
simple-lift s q = *Some x1* ⟹
(*simple-lift* (*hrs-mem-update* (*heap-update p v*) *s*) *q*) = *Some x2* ⟹
x1 = *x2*
shows *the* (*simple-lift*
(*hrs-mem-update* (*heap-update p v*) *s*) *q*) =
the (*simple-lift s q*)
proof (*cases root-ptr-valid* (*hrs-htd s*) *q*)
case *True*
with *eq* **show** *?thesis* **by** (*fastforce simp add*: *simple-lift-def*)
next
case *False*
hence *simple-lift s q* = *None*
by (*simp add*: *simple-lift-def*)
with *simple-lift-hrs-mem-update-None*

show *?thesis* **by** *metis*
qed

lemma *the-simple-lift-hval-eqI*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*

assumes *eq*: *root-ptr-valid* (*hrs-htd s*) *q* ⟹
(*h-val* ((*heap-update p v*) (*hrs-mem s*)) *q*) = *h-val* (*hrs-mem s*) *q*

shows *the* (*simple-lift*
(*hrs-mem-update* (*heap-update p v*) *s*) *q*) =
the (*simple-lift s q*)
apply (*rule the-simple-lift-strong-eqI*)
apply (*drule h-val-simple-lift*)
apply (*drule h-val-simple-lift*)
apply (*auto simp add*: *hrs-mem-update eq*)
done

lemma *h-t-valid-hrs-mem-update-pres*: *hrs-htd s*,*g* ⊨$_t$ *q* ⟹
hrs-htd ((*hrs-mem-update* (*heap-update p v*) *s*)), *g*⊨$_t$ *q*
by (*cases s*) (*auto simp add*: *hrs-htd-def hrs-mem-update-def*)

lemma *field-the-clift-hval-eqI*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*
assumes *eq*: *hrs-htd s*, *c-guard* ⊨$_t$ *q* ⟹
f (*h-val* ((*heap-update p v*) (*hrs-mem s*)) *q*) = *f* (*h-val* (*hrs-mem s*) *q*)
shows *f* (*the* (*clift*
(*hrs-mem-update* (*heap-update p v*) *s*) *q*)) =
f (*the* (*clift s q*))
proof (*cases* (*clift s q*))
case *None*
then show *?thesis* **using** *clift-hrs-mem-update-None* **by** *metis*
next
case (*Some x*)
from *h-t-valid-clift* [*OF this*] **have** *valid*: *hrs-htd s* ⊨$_t$ *q* **.**
hence *hrs-htd* ((*hrs-mem-update* (*heap-update p v*) *s*))⊨$_t$ *q*
by (*rule h-t-valid-hrs-mem-update-pres*)
with *h-t-valid-clift-Some-iff*
obtain *y* **where** *y*: *clift* (*hrs-mem-update* (*heap-update p v*) *s*) *q* = *Some y*
by *blast*
from *h-val-clift'* [*OF Some*] *h-val-clift'* [*OF this*] *eq* [*OF valid*]
show *?thesis* **by** (*auto simp add*: *Some y hrs-mem-update*)
qed

lemma *the-clift-hval-eqI*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*

assumes *eq*: *hrs-htd s*, *c-guard* $\models_t$ *q* $\Longrightarrow$
(*h-val* ((*heap-update p v*) (*hrs-mem s*)) *q*) = (*h-val* (*hrs-mem s*) *q*)
shows *the* (*clift*
(*hrs-mem-update* (*heap-update p v*) *s*) *q*) =
(*the* (*clift s q*))
apply (*rule field-the-clift-hval-eqI*)
by (*rule eq*)

lemma *valid-root-footprint-same-type-cases*:
assumes *valid-p*: *valid-root-footprint d* (*ptr-val* (*p*::'*a*:: *mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::'*a*::*mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))

assumes *eq-case*: *p* = *q* $\Longrightarrow$ *P*
assumes *disj-case*: *ptr-span p* $\cap$ *ptr-span q* = {} $\Longrightarrow$ *P*
shows *P*
proof (*cases ptr-span p* $\cap$ *ptr-span q* = {})
case *True*
then show *?thesis* **by** (*simp add*: *disj-case*)
next
case *False*
from *valid-root-footprint-overlap-contained* [*OF valid-p valid-q*] *False*
have *ptr-val q* $\in$ *ptr-span p*
by (*simp add*: *typ-uinfo-t-def size-of-def*)
hence *p* = *q*
by (*metis ptr-val-conv size-of-tag valid-footprint-neq-nmem valid-p valid-q valid-root-footprint-valid-footprint*)
thus *?thesis* **by** (*simp add*: *eq-case*)
qed

lemma *valid-footprint-overlap-contained*:
assumes *valid-root-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: {*x* ..+ *size-td t*} $\cap$ {*y* ..+ *size-td s*} $\neq$ {}
shows *y* $\in$ {*x* ..+ *size-td t*} $\vee$ *x* $\in$ {*y* ..+ *size-td s*}
using *valid-root-x overlap valid-y*
apply (*clarsimp simp add*: *valid-footprint-def Let-def typ-tag-le-def*)
by (*meson intvl-inter overlap*)

lemma *valid-footprint-same-type-cases*:
assumes *valid-p*: *valid-footprint d* (*ptr-val* (*p*::'*a*:: *mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::'*a*::*mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))

assumes *eq-case*: *p* = *q* $\Longrightarrow$ *P*
assumes *disj-case*: *ptr-span p* $\cap$ *ptr-span q* = {} $\Longrightarrow$ *P*
shows *P*
proof (*cases ptr-span p* $\cap$ *ptr-span q* = {})

```
  case True
  then show ?thesis by (simp add: disj-case)
next
  case False
  from valid-footprint-overlap-contained [OF valid-p valid-q ] False
  have ptr-val q ∈ ptr-span p ∨ ptr-val p ∈ ptr-span q
    by (simp add: typ-uinfo-t-def size-of-def)
  hence p = q
    by (metis ptr-val-conv size-of-tag valid-footprint-neq-nmem valid-p valid-q )
  thus ?thesis by (simp add: eq-case)
qed

theorem subfield-deref-append:
  fixes q::'a::mem-type ptr
  fixes p::'b::mem-type ptr
  assumes base: ptr-val p = &(q→g)
 assumes g: g ∈ set (field-names-u (typ-uinfo-t (TYPE('a))) (typ-uinfo-t (TYPE('b))))
  assumes f: f ∈ set (all-field-names (typ-uinfo-t (TYPE('b))))
  shows &(p→f) = &(q→(g@f))
proof −
  have wf-a: wf-desc (typ-info-t (TYPE('a))) by simp
  have wf-b: wf-desc (typ-info-t (TYPE('b))) by simp
 from g have g ∈ set (field-names (typ-info-t (TYPE('a))) (typ-uinfo-t (TYPE('b))))
    by (simp add: typ-uinfo-t-def field-names-u-field-names-export-uinfo-conv(1))
  from field-names-Some2(1) [rule-format, OF wf-a this] obtain n s where
    fl-g: field-lookup (typ-info-t TYPE('a)) g 0 = Some (s, n) and
    s: export-uinfo s = typ-uinfo-t TYPE('b)
    by blast

  from fl-g have off-g: field-offset TYPE('a) g = n
    by (simp)

  from all-field-names-exists-field-names-u(1) [OF f] obtain t
    where f ∈ set (field-names-u (typ-uinfo-t TYPE('b)) t) by blast
  hence f ∈ set (field-names (typ-info-t TYPE('b)) t)
    by (simp add: field-names-u-field-names-export-uinfo-conv(1) typ-uinfo-t-def)
  from field-names-Some2(1) [rule-format, OF wf-b  this] obtain t' m where
    fl-f: field-lookup (typ-info-t TYPE('b)) f 0 = Some (t', m) and
    t': export-uinfo t' = t
    by blast

  from fl-f have off-f: field-offset TYPE('b) f = m
    by simp

  from field-lookup-prefix-Some''(1) [rule-format, OF fl-g wf-a, of f]
  have fl-g-f: field-lookup (typ-info-t TYPE('a)) (g @ f) 0 = field-lookup s f n .

  from field-lookup-export-uinfo-Some [OF fl-f] t'
  have field-lookup (typ-uinfo-t TYPE('b)) f 0 = Some (t, m)
```

by (*simp add*: *typ-uinfo-t-def*)

from *field-lookup-offset-shift'* [*OF this*, *of n*]
have *field-lookup* (*typ-uinfo-t TYPE*('*b*)) *f n* = *Some* (*t*, *m* + *n*)
by *simp*

with *s* **have** *field-lookup* (*export-uinfo s*) *f n* = *Some* (*t*, *m* + *n*)
by *simp*
from *field-lookup-export-uinfo-Some-rev* [*OF this*] **obtain** *t''* **where**
fl-s-f: *field-lookup s f n* = *Some* (*t''*, *m* + *n*) **and**
t'': *t* = *export-uinfo t''*
by *blast*
from *fl-g-f fl-s-f* **have** *off-g-f*: *field-offset TYPE*('*a*) (*g*@*f*) = *m* + *n*
by (*simp*)

show *?thesis*
using *base*
by (*simp add*: *field-lvalue-def off-g off-f off-g-f*)
qed

lemmas *root-ptr-valid-same-type-cases* = *valid-root-footprint-same-type-cases* [*OF root-ptr-valid-valid-root-footprint h-t-valid-valid-footprint*]
lemmas *ptr-valid-same-type-cases* = *valid-footprint-same-type-cases* [*OF h-t-valid-valid-footprint h-t-valid-valid-footprint*]

theorem *root-ptr-valid-heap-update-other'*:
assumes *val-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
assumes *not-subtype*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
apply (*clarsimp simp*: *h-val-def heap-update-def*)
apply (*subst heap-list-update-disjoint-same*)
apply *simp*
apply (*rule root-ptr-valid-not-subtype-disjoint* [*OF val-p val-q not-subtype*])
apply *simp*
done

theorem *root-ptr-valid-heap-update-field-other*:
assumes *val-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*c*::*mem-type*)
assumes *not-subtype-b-c*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*c*)
assumes *not-subtype-c-b*: ¬ *TYPE*('*c*) ≤$_\tau$ *TYPE*('*b*)
shows *h-val* (*heap-update* (*PTR*('*c*) &(*p*→*f*)) *v h*) *q* = *h-val h q*

proof (*cases ptr-span* (*PTR*('*c*) &(*p*→*f*)) ∩ *ptr-span q* = {})
case *True*
then show *?thesis* **by** (*simp add*: *h-val-update-regions-disjoint*)
next
case *False*
from *valid-footprint-overlap-cases* [*OF - - False*] *val-p val-q field-lookup-typ-uinfo-t-Some* [*OF fl*]
not-subtype-b-c not-subtype-c-b match
have *False*
by (*smt* (*verit*) *cvalid-field-pres h-t-valid-valid-footprint root-ptr-valid-legacy-def*)

thus *?thesis*
by *simp*
qed

theorem *root-ptr-valid-heap-update-field-other'*:
assumes *val-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*b*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*c*::*mem-type*)
assumes *not-subtype-b-a*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
shows *h-val* (*heap-update p v h*) (*PTR* ('*c*) &(*q*→*f*)) = *h-val h* (*PTR* ('*c*) &(*q*→*f*))
proof (*cases ptr-span p* ∩ *ptr-span q* = {})
case *True*
from *fl val-q match* **have** *subset*: *ptr-span* (*PTR*('*c*) &(*q*→*f*)) ⊆ *ptr-span q*
by (*metis export-uinfo-size field-tag-sub ptr-val.ptr-val-def size-of-tag*)
show *?thesis*
apply (*rule h-val-update-regions-disjoint*)
using *True subset*
by *auto*
next
case *False*
from *valid-root-footprint-valid-footprint-overlap-case* [*OF - - False*] *val-p val-q*
have *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
by (*meson False cvalid-field-pres'' fl match ptr-val.ptr-val-def root-ptr-valid-not-subtype-disjoint*)
with *not-subtype-b-a* **have** *False* **by** *simp*
thus *?thesis*
by *simp*
qed

lemmas *root-ptr-valid-dist-type-cases* = *valid-root-footprint-dist-type-cases* [*OF root-ptr-valid-valid-root-footpri*
h-t-valid-valid-footprint,
consumes 3, *case-names Field Disjoint-Spans*]

theorem *ptr-valid-heap-update-field-disj*:

assumes *val-p*: $d \models_t (p::'a::mem\text{-}type\ ptr)$
assumes *val-q*: $d \models_t (q::'b::mem\text{-}type\ ptr)$
assumes *subtype*: $TYPE('b) \leq_\tau TYPE('a)$
assumes *disj* [*rule-format*] :
$\forall f.\ f \in set\ (field\text{-}names\text{-}u\ (typ\text{-}uinfo\text{-}t\ (TYPE('a)))\ (typ\text{-}uinfo\text{-}t\ (TYPE('b))))$
$\longrightarrow q \neq Ptr\ (\&(p{\rightarrow}f))$
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
proof (*cases ptr-span p* ∩ *ptr-span q* = {})
case *True*
then show *?thesis*
by (*simp add*: *h-val-update-regions-disjoint*)
next
case *False*
from *subtype* **have** *typ-uinfo-t TYPE*('*b*) ≤ *typ-uinfo-t TYPE*('*a*) **by** (*simp add*: *sub-typ-def*)

from *this False valid-footprint-sub-cases* [*OF h-t-valid-valid-footprint* [*OF val-q*] *h-t-valid-valid-footprint* [*OF val-p*]]
have *field-of*: *field-of* (*ptr-val q* − *ptr-val p*) (*typ-uinfo-t TYPE*('*b*)) (*typ-uinfo-t TYPE*('*a*))
apply (*simp add*: *size-of-def*)
by (*metis Int-commute le-less-trans less-irrefl*)
then obtain *f* **where**
fl: *field-lookup* (*typ-uinfo-t* (*TYPE*('*a*))) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*), *unat* (*ptr-val q* − *ptr-val p*))
using *field-of-lookup-info* **by** *blast*

then obtain *s* **where**
fl': *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*s*, *unat* (*ptr-val q* − *ptr-val p*)) **and**
s: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*)
using *field-lookup-uinfo-Some-rev* **by** *blast*

from *fl* **have** *f*: *f* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))
by (*metis field-names-Some3*(*1*) *field-names-u-field-names-export-uinfo-conv*(*1*) *fl' s typ-uinfo-t-def*)

from *fl'*
have *ptr-val-q*: *ptr-val q* = &(*p*→*f*)
by (*simp add*: *field-lvalue-def*)

from *disj* [*OF f*] *ptr-val-q*
show *?thesis*
by (*auto simp add*: *ptr-val-field-convD*)
qed

lemma *is-padding-tag-access-ti-eq*: *is-padding-tag s* ⟹ *access-ti s x bs* = *access-ti*

s y bs
by (*clarsimp simp add*: *is-padding-tag-def padding-tag-def from-bytes-def access-ti$_0$-def padding-desc-def*)

lemma *is-padding-tag-access-ti$_0$*: *is-padding-tag s* $\Longrightarrow$ *access-ti$_0$ s x* = *replicate* (*size-td s*) *0*
by (*clarsimp simp add*: *is-padding-tag-def padding-tag-def from-bytes-def access-ti$_0$-def padding-desc-def*)

lemma *is-padding-tag-from-bytes-eq*: *is-padding-tag s* $\Longrightarrow$ *from-bytes* (*access-ti$_0$ s x*) = *from-bytes* (*access-ti$_0$ s y*)
by (*simp add*: *from-bytes-def is-padding-tag-access-ti$_0$*)

theorem *ptr-valid-heap-update-field-disj'*:
assumes *val-p*: *d* $\models_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* $\models_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*('*b*) $\leq_\tau$ *TYPE*('*a*)
assumes *disj* [*rule-format*] :
$\forall f.\ f \in$ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t* (*TYPE*('*b*))))) $\longrightarrow$ *q* $\neq$ *Ptr* (&(*p*→*f*))
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
proof (*cases ptr-span p* $\cap$ *ptr-span q* = {})
case *True*
then show *?thesis*
by (*simp add*: *h-val-update-regions-disjoint*)
next
case *False*
from *subtype* **have** *typ-uinfo-t TYPE*('*b*) $\leq$ *typ-uinfo-t TYPE*('*a*) **by** (*simp add*: *sub-typ-def*)

from *this False valid-footprint-sub-cases* [*OF h-t-valid-valid-footprint* [*OF val-q*] *h-t-valid-valid-footprint* [*OF val-p*]]
have *field-of*: *field-of* (*ptr-val q* − *ptr-val p*) (*typ-uinfo-t TYPE*('*b*)) (*typ-uinfo-t TYPE*('*a*))
apply (*simp add*: *size-of-def*)
by (*metis Int-commute le-less-trans less-irrefl*)
then obtain *f* **where**
fl: *field-lookup* (*typ-uinfo-t* (*TYPE*('*a*))) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*), *unat* (*ptr-val q* − *ptr-val p*))
using *field-of-lookup-info* **by** *blast*

then obtain *s* **where**
fl': *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*s*, *unat* (*ptr-val q* − *ptr-val p*)) **and**
s: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*)
using *field-lookup-uinfo-Some-rev* **by** *blast*

from *fl'* **have** *fl''*: *field-ti* (*TYPE*('*a*)) *f* = *Some s*

by (*simp add*: *field-ti-def*)

from *fl'* **have** *f*: *f* ∈ *set* (*field-names* (*typ-info-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))

by (*metis field-names-Some3*(*1*) *fl' s*)

from *fl'*
have *ptr-val-q*: *ptr-val q* = &(*p*→*f*)
by (*simp add*: *field-lvalue-def*)
then have *q*: *q* = *PTR* ('*b*) &(*p*→*f*)
by (*cases q*) *auto*

show *?thesis*
proof (*cases qualified-padding-field-name f*)
case *True*
from *field-lookup-qualified-padding-field-name*(*1*) [*OF fl' True*]
have *padding-tag-s*: *is-padding-tag s*
by (*simp add*: *wf-padding*)
note *h-val* = *h-val-field-from-bytes'* [**where** *pa*=*p*,*OF fl'' s* [*simplified typ-uinfo-t-def*]]
show *?thesis* **apply** (*simp add*: *q*)
apply (*simp add*: *h-val*)
apply (*simp add*: *is-padding-tag-from-bytes-eq* [*OF padding-tag-s*])
done
next
case *False*
with *f fl' s* **have** *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*)))
(*export-uinfo* (*typ-info-t* (*TYPE*('*b*)))))
by (*simp add*: *set-field-names-no-padding-all-field-names-no-padding-conv*
all-field-names-no-padding-def
fold-typ-uinfo-t set-field-names-all-field-names-conv)
with *disj* [*OF this*] *ptr-val-q* **show** *?thesis* **by** (*auto simp add*: *ptr-val-field-convD*)
qed

qed

lemma *is-padding-tag-update-ti-id*: *is-padding-tag s* ⟹ *update-ti s bs v* = *v*
by (*auto simp add*: *is-padding-tag-def padding-tag-def padding-desc-def*)

theorem *ptr-valid-heap-update-field-disj''*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *fl-g*: *field-lookup* (*typ-info-t TYPE*('*a*)) *g 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*c*::*xmem-type*)
assumes *disj* [*rule-format*] :
∀*f*. *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t*
(*TYPE*('*b*)))))
⟶ *ptr-span* (*PTR*('*b*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*c*) &(*p*→*g*)) = {}
shows *h-val* (*heap-update q v h*) (*PTR*('*c*) &(*p*→*g*)) = *h-val h* (*PTR*('*c*) &(*p*→*g*))

proof (*cases ptr-span q* ∩ *ptr-span* (*PTR*('*c*) &(*p*→*g*)) = {})
case *True*
then show *?thesis* **by** (*simp add*: *h-val-update-regions-disjoint*)
next
case *False*
from *subtype* **have** *st*: *typ-uinfo-t TYPE*('*b*) ≤ *typ-uinfo-t TYPE*('*a*) **by** (*simp add*: *sub-typ-def*)
from *fl-g match* **have** *ptr-span* (*PTR*('*c*) &(*p*→*g*)) ⊆ *ptr-span p*
by (*metis export-uinfo-size field-tag-sub ptr-val.ptr-val-def size-of-tag*)
with *False* **have** *ptr-span q* ∩ *ptr-span p* ≠ {}
by *auto*

from *this st valid-footprint-sub-cases* [*OF h-t-valid-valid-footprint* [*OF val-q*] *h-t-valid-valid-footprint* [*OF val-p*]]
have *field-of*: *field-of* (*ptr-val q* − *ptr-val p*) (*typ-uinfo-t TYPE*('*b*)) (*typ-uinfo-t TYPE*('*a*))
by (*simp add*: *size-of-def*)

then obtain *f* **where**
fl: *field-lookup* (*typ-uinfo-t* (*TYPE*('*a*))) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*), *unat* (*ptr-val q* − *ptr-val p*))
using *field-of-lookup-info* **by** *blast*

then obtain *s* **where**
fl': *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*s*, *unat* (*ptr-val q* − *ptr-val p*)) **and**
s: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*)
using *field-lookup-uinfo-Some-rev* **by** *blast*

from *fl'* **have** *fl''*: *field-ti* (*TYPE*('*a*)) *f* = *Some s*
by (*simp add*: *field-ti-def*)

from *fl'* **have** *f*: *f* ∈ *set* (*field-names* (*typ-info-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))

by (*metis field-names-Some3*(*1*) *fl' s*)

from *val-p* **have** *cgrd-p*: *c-guard p*
by (*simp add*: *h-t-valid-c-guard*)
from *fl'*
have *ptr-val-q*: *ptr-val q* = &(*p*→*f*)
by (*simp add*: *field-lvalue-def*)
then have *q*: *q* = *PTR* ('*b*) &(*p*→*f*)
by (*cases q*) *auto*

show *?thesis*
proof (*cases qualified-padding-field-name f*)
case *True*
from *field-lookup-qualified-padding-field-name*(*1*) [*OF fl' True*]
have *padding-tag-s*: *is-padding-tag s*

by (*simp add*: *wf-padding*)
note *h-upd* = *heap-update-field-root-conv'''* [*OF fl''* *cgrd-p s*]

show *?thesis*
apply (*simp add*: *q*)
by (*simp add*: *h-upd is-padding-tag-update-ti-id* [*OF padding-tag-s*] *xmem-type-class.heap-update-id*)
next
case *False*
with *f fl' s* **have** *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t* (*TYPE*('*b*)))))
by (*simp add*: *set-field-names-no-padding-all-field-names-no-padding-conv all-field-names-no-padding-def fold-typ-uinfo-t set-field-names-all-field-names-conv*)
from *disj* [*OF this*] **have** *ptr-span* (*PTR*('*b*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*c*) &(*p*→*g*)) = {} **.**
then show *?thesis*
by (*simp add*: *q h-val-update-regions-disjoint*)
qed
qed

theorem *ptr-valid-heap-update-field-access-ti-disj*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *disj* [*rule-format*] :
∀*f*. *f* ∈ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*('*a*))) (*typ-uinfo-t* (*TYPE*('*b*))))
⟶
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t* *TYPE*('*a*)) *f* 0)))
v
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t* *TYPE*('*a*)) *f* 0)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
proof (*cases ptr-span p* ∩ *ptr-span q* = {})
case *True*
then show *?thesis*
by (*simp add*: *h-val-update-regions-disjoint*)
next
case *False*

from *subtype* **have** *typ-uinfo-t TYPE*('*b*) ≤ *typ-uinfo-t TYPE*('*a*) **by** (*simp add*: *sub-typ-def*)

from *this False valid-footprint-sub-cases* [*OF h-t-valid-valid-footprint* [*OF val-q*] *h-t-valid-valid-footprint* [*OF val-p*]]
have *field-of*: *field-of* (*ptr-val q* − *ptr-val p*) (*typ-uinfo-t TYPE*('*b*)) (*typ-uinfo-t TYPE*('*a*))
apply (*simp add*: *size-of-def*)

by (*metis Int-commute le-less-trans less-irrefl*)
then obtain *f* **where**
fl: *field-lookup* (*typ-uinfo-t* (*TYPE*('*a*))) *f* 0 = *Some* (*typ-uinfo-t TYPE*('*b*), *unat* (*ptr-val q* − *ptr-val p*))
using *field-of-lookup-info* **by** *blast*

then obtain *s* **where**
fl': *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f* 0 = *Some* (*s*, *unat* (*ptr-val q* − *ptr-val p*)) **and**
s: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*)
using *field-lookup-uinfo-Some-rev* **by** *blast*

from *fl* **have** *f*: *f* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))
by (*metis field-names-Some3*(*1*) *field-names-u-field-names-export-uinfo-conv*(*1*) *fl' s typ-uinfo-t-def*)

from *fl'*
have *ptr-val-q*: *ptr-val q* = &(*p*→*f*)
by (*simp add*: *field-lvalue-def*)

from *field-of*
have *contained*: *ptr-span q* ⊆ *ptr-span p*
by (*metis* (*no-types*, *opaque-lifting*)
add-diff-cancel-left' export-uinfo-size field-lvalue-def field-of-lookup-info ptr-val-q size-of-def typ-uinfo-t-def)

from *disj* [*OF f*] *fl'* **have** *from-bytes-eq*:
(*access-ti s v* (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti s* (*h-val h p*) (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) **by** *simp*

from *ptr-val-q* **have** *q*: *ptr-val q* = *ptr-val p* + *word-of-nat* (*unat* (*ptr-val q* − *ptr-val p*))
by (*simp add*: *field-lvalue-def field-offset-def field-offset-untyped-def fl*)

from *fl' s*
have *sz-bound*: *size-of TYPE*('*b*) + *unat* (*ptr-val q* − *ptr-val p*)
≤ *length* (*to-bytes v* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)))
by (*metis export-uinfo-size field-lookup-offset-size heap-list-length len size-of-def typ-uinfo-size*)

have *lheap-list*: *length* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)) = *size-of* (*TYPE*('*a*))
by *simp*
from *s*
have *sz-s*: *size-td s* = *size-of TYPE*('*b*)
by (*simp add*: *export-size-of*)

from *val-p*
have *p-no-overflow*: *unat* (*ptr-val p*) + *size-of TYPE*('*a*) ≤ *addr-card*
by (*meson c-guard-no-wrap' h-t-valid-def*)

from *contained q*
have *q-bound*: *unat* (*ptr-val q* − *ptr-val p*) + *size-of TYPE*('*b*) ≤ *size-of TYPE*('*a*)
using *sz-bound* **by** *auto*

from *heap-list-take-drop'* [*OF p-no-overflow q-bound*]
have *take-drop-eq*:
take (*size-of TYPE*('*b*))
(*drop* (*unat* (*ptr-val q* − *ptr-val p*))
(*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))) =
heap-list h (*size-of TYPE*('*b*)) (*ptr-val q*)
by (*simp add*: *q* [*symmetric*])

note *acc-eq* = *mem-type-field-lookup-access-ti-take-drop* [*OF fl' lheap-list*, *simplified sz-s*, *of v*, *simplified take-drop-eq*]

note *acc-eq'* = *mem-type-field-lookup-access-ti-take-drop* [*OF fl' lheap-list*, *simplified sz-s*, *of h-val h p*, *simplified take-drop-eq*]

have *acc-eq''*:
(*access-ti* (*typ-info-t TYPE*('*a*)) (*h-val h p*)
(*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))) = (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))
apply (*simp add*: *h-val-def*)
apply (*simp add*: *to-bytes-def* [*symmetric*])
by (*simp add*: *to-bytes-from-bytes-id*)

show *?thesis*
apply (*simp add*: *heap-update-def h-val-def*)
apply (*subst* (*1 2*) *q*)
apply (*subst heap-list-update-list* [*OF sz-bound*])
apply *simp*
apply (*simp add*: *to-bytes-def*)
apply (*subst acc-eq* [*symmetric*])
apply (*subst from-bytes-eq*)
apply (*subst acc-eq'*)
apply (*subst acc-eq''*)
apply (*simp add*: *take-drop-eq*)
done
qed

lemma *access-ti-field-names-no-padding-to-field-names-u*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)

```
  assumes disj [rule-format]:
  ∀f. f ∈ set (field-names-no-padding (typ-info-t (TYPE('a))) (export-uinfo (typ-info-t
(TYPE('b))))) ⟶
     (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
        v
        (heap-list h (size-of TYPE('b)) (ptr-val q))) =
     (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
       (h-val h p)
       (heap-list h (size-of TYPE('b)) (ptr-val q)))
 assumes f-in: f ∈ set (field-names-u (typ-uinfo-t (TYPE('a))) (typ-uinfo-t (TYPE('b))))
 shows (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
        v
        (heap-list h (size-of TYPE('b)) (ptr-val q))) =
     (access-ti (fst (the (field-lookup (typ-info-t TYPE('a)) f 0)))
       (h-val h p)
       (heap-list h (size-of TYPE('b)) (ptr-val q)))
proof −
  from f-in
  obtain n s where
    fl: field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n) and
    match: export-uinfo s = export-uinfo (typ-info-t (TYPE('b)))
   by (smt (verit) field-names-Some2(1) fold-typ-uinfo-t set-field-names-all-field-names-conv

        set-field-names-u-all-field-names-conv wf-desc)

  show ?thesis
  proof (cases qualified-padding-field-name f)
    case True
    from field-lookup-qualified-padding-field-name(1) [OF fl True]
    have padding-tag-s: is-padding-tag s
      by (simp add: wf-padding)
    show ?thesis by (simp add: fl is-padding-tag-access-ti-eq [OF padding-tag-s])
  next
    case False
    with fl match field-lookup-all-field-names(1) [OF fl]
     have f ∈ set (field-names-no-padding (typ-info-t (TYPE('a))) (export-uinfo
(typ-info-t (TYPE('b)))))
        by (simp add: set-field-names-no-padding-all-field-names-no-padding-conv
all-field-names-no-padding-def)

    from disj [rule-format, OF this]
    show ?thesis .
  qed
qed


theorem ptr-valid-heap-update-field-access-ti-disj':
  assumes val-p: d ⊨t (p::'a::xmem-type ptr)
  assumes val-q: d ⊨t (q::'b::xmem-type ptr)
```

assumes *subtype*: $TYPE('b) \leq_\tau TYPE('a)$
assumes *disj* :
$\forall f.\ f \in set\ (field\text{-}names\text{-}no\text{-}padding\ (typ\text{-}info\text{-}t\ (TYPE('a)))\ (export\text{-}uinfo\ (typ\text{-}info\text{-}t\ (TYPE('b)))))) \longrightarrow$
$(access\text{-}ti\ (fst\ (the\ (field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a))\ f\ 0)))\ v\ (heap\text{-}list\ h\ (size\text{-}of\ TYPE('b))\ (ptr\text{-}val\ q))) =$
$(access\text{-}ti\ (fst\ (the\ (field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a))\ f\ 0)))\ (h\text{-}val\ h\ p)\ (heap\text{-}list\ h\ (size\text{-}of\ TYPE('b))\ (ptr\text{-}val\ q)))$
shows $h\text{-}val\ (heap\text{-}update\ p\ v\ h)\ q = h\text{-}val\ h\ q$
apply (*rule ptr-valid-heap-update-field-access-ti-disj* [*OF val-p val-q subtype*])
apply *clarify*
apply (*rule access-ti-field-names-no-padding-to-field-names-u* [*OF val-p val-q subtype disj*])
apply *assumption*
done

theorem *ptr-valid-heap-update-subtype-field-access-ti-disj*:
assumes *val-p*: $d \models_t (p::'a::xmem\text{-}type\ ptr)$
assumes *val-q*: $d \models_t (q::'b::xmem\text{-}type\ ptr)$
assumes *subtype-b-a*: $TYPE('b) \leq_\tau TYPE('a)$
assumes *fl-g*: $field\text{-}lookup\ (typ\text{-}uinfo\text{-}t\ TYPE('b))\ g\ 0 = Some\ (typ\text{-}uinfo\text{-}t\ TYPE('c::xmem\text{-}type), n)$
assumes *disj* [*rule-format*] :
$\forall f.\ f \in set\ (field\text{-}names\text{-}u\ (typ\text{-}uinfo\text{-}t\ (TYPE('a)))\ (typ\text{-}uinfo\text{-}t\ (TYPE('b))))) \longrightarrow$
$(access\text{-}ti\ (fst\ (the\ (field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a))\ f\ 0)))\ v\ (heap\text{-}list\ h\ (size\text{-}of\ TYPE('b))\ (ptr\text{-}val\ q))) =$
$(access\text{-}ti\ (fst\ (the\ (field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a))\ f\ 0)))\ (h\text{-}val\ h\ p)\ (heap\text{-}list\ h\ (size\text{-}of\ TYPE('b))\ (ptr\text{-}val\ q)))$
shows $h\text{-}val\ (heap\text{-}update\ p\ v\ h)\ (PTR('c)\ (\&(q \rightarrow g))) = h\text{-}val\ h\ (PTR('c)\ (\&(q \rightarrow g)))$
proof (*cases ptr-span p* $\cap$ *ptr-span q* $= \{\}$)
case *True*
from *fl-g* **have** $ptr\text{-}span\ (PTR('c)\ (\&(q \rightarrow g))) \subseteq ptr\text{-}span\ q$
by (*metis export-uinfo-size field-lookup-uinfo-Some-rev field-tag-sub ptr-val.ptr-val-def size-of-tag*)
with *True* **have** $ptr\text{-}span\ p \cap ptr\text{-}span\ (PTR('c)\ (\&(q \rightarrow g))) = \{\}$ **by** *blast*
then show *?thesis*
by (*simp add*: *h-val-update-regions-disjoint*)
next
case *False*
from *subtype-b-a* **have** $typ\text{-}uinfo\text{-}t\ TYPE('b) \leq typ\text{-}uinfo\text{-}t\ TYPE('a)$ **by** (*simp add*: *sub-typ-def*)

from *this False valid-footprint-sub-cases* [*OF h-t-valid-valid-footprint* [*OF val-q*]

h-t-valid-valid-footprint [*OF val-p*]]
have *field-of*: *field-of* (*ptr-val q* − *ptr-val p*) (*typ-uinfo-t TYPE*('*b*)) (*typ-uinfo-t TYPE*('*a*))
apply (*simp add*: *size-of-def*)
by (*metis Int-commute le-less-trans less-irrefl*)
then obtain *f* **where**
fl: *field-lookup* (*typ-uinfo-t* (*TYPE*('*a*))) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*), *unat* (*ptr-val q* − *ptr-val p*))
using *field-of-lookup-info* **by** *blast*

then obtain *s* **where**
fl': *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*s*, *unat* (*ptr-val q* − *ptr-val p*)) **and**
s: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*)
using *field-lookup-uinfo-Some-rev* **by** *blast*

from *fl* **have** *f*: *f* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))
by (*metis field-names-Some3*(*1*) *field-names-u-field-names-export-uinfo-conv*(*1*) *fl' s typ-uinfo-t-def*)

from *fl'*
have *ptr-val-q*: *ptr-val q* = &(*p*→*f*)
by (*simp add*: *field-lvalue-def*)

from *field-of*
have *contained*: *ptr-span q* ⊆ *ptr-span p*
by (*metis* (*no-types*, *opaque-lifting*)
add-diff-cancel-left' export-uinfo-size field-lvalue-def field-of-lookup-info ptr-val-q size-of-def typ-uinfo-t-def)

from *fl-g* **have** *contained'*: *ptr-span* (*PTR*('*c*) (&(*q*→*g*))) ⊆ *ptr-span q*
by (*metis export-uinfo-size field-lookup-uinfo-Some-rev field-tag-sub ptr-val.ptr-val-def size-of-tag*)

from *disj* [*OF f*] *fl'* **have** *from-bytes-eq*:
(*access-ti s v* (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti s* (*h-val h p*) (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) **by** *simp*

from *ptr-val-q* **have** *q*: *ptr-val q* = *ptr-val p* + *word-of-nat* (*unat* (*ptr-val q* − *ptr-val p*))
by (*simp add*: *field-lvalue-def field-offset-def field-offset-untyped-def fl*)

from *fl-g* **have** *q-g*: &(*q*→*g*) = *ptr-val q* + *word-of-nat n*
by (*simp add*: *field-lvalue-def field-offset-def field-offset-untyped-def*)
with *q* **have** *q-g'*: &(*q*→*g*) = *ptr-val p* + *word-of-nat* (*unat* (*ptr-val q* − *ptr-val p*) + *n*)
by *simp*

from *fl' s*
have *sz-bound*: *size-of TYPE('b) + unat (ptr-val q − ptr-val p)*
≤ length (to-bytes v (heap-list h (size-of TYPE('a)) (ptr-val p)))
by (*metis export-uinfo-size field-lookup-offset-size heap-list-length len size-of-def typ-uinfo-size*)

from *fl-g* **have** *subtype-c-b*: *TYPE('c::xmem-type) ≤τ TYPE('b)*
by (*meson sub-typ-def td-set-field-lookupD typ-tag-le-def*)

from *contained' subtype-c-b fl-g*
have *sz-c-n*: *size-of TYPE('c) + n ≤ size-of TYPE('b)*
by (*metis size-of-def td-set-field-lookupD td-set-offset-size typ-uinfo-size*)

from *sz-c-n sz-bound*
have *sz-bound'*: *size-of TYPE('c) + (unat (ptr-val q − ptr-val p) + n)*
≤ length (to-bytes v (heap-list h (size-of TYPE('a)) (ptr-val p)))
by *simp*
have *lheap-list*: *length (heap-list h (size-of TYPE('a)) (ptr-val p)) = size-of (TYPE('a))*
by *simp*
from *s*
have *sz-s*: *size-td s = size-of TYPE('b)*
by (*simp add*: *export-size-of*)

from *val-p*
have *p-no-overflow*: *unat (ptr-val p) + size-of TYPE('a) ≤ addr-card*
by (*meson c-guard-no-wrap' h-t-valid-def*)

from *contained q*
have *q-bound*: *unat (ptr-val q − ptr-val p) + size-of TYPE('b) ≤ size-of TYPE('a)*
using *sz-bound* **by** *auto*

from *fl-g s* **obtain** *s'* **where**
fl-s-g: *field-lookup s g 0 = Some (s', n)* **and**
s': *export-uinfo s' = typ-uinfo-t TYPE('c)*
by (*metis CTypes.field-lookup-export-uinfo-Some-rev*)

from *s'*
have *sz-s'*: *size-td s' = size-of TYPE('c)*
by (*simp add*: *export-size-of*)

from *sz-c-n q-bound*
have *q-bound'*: *unat (ptr-val q − ptr-val p) + n + size-of TYPE('c) ≤ size-of TYPE('a)*
by *simp*
have *take-drop-eq*:
(*take (size-of TYPE('c))*
(*drop (unat (ptr-val q − ptr-val p) + n)*
(*heap-list h (size-of TYPE('a)) (ptr-val p)*))) =

heap-list h (size-of TYPE('c)) (&(q→g))
apply *(subst q-g')*
apply *(rule heap-list-take-drop' [OF p-no-overflow q-bound'])*
done

from *fl-s-g field-lookup-prefix-Some''(1) [rule-format, OF fl' wf-desc,* **where** *g=g*
]
have *fl-f-g: field-lookup (typ-info-t TYPE('a)) (f @ g) 0 = Some (s', unat (ptr-val q − ptr-val p) + n)*
by *(simp add: field-lookup-offset)*

note *acc-eq = mem-type-field-lookup-access-ti-take-drop [OF fl-f-g lheap-list, simplified sz-s', of v, simplified take-drop-eq]*
note *acc-eq' = mem-type-field-lookup-access-ti-take-drop [OF fl-f-g lheap-list, simplified sz-s', of h-val h p, simplified take-drop-eq]*

from *s fl'* **have** *wf-fd-s: wf-fd s* **by** *(meson wf-fd wf-fd-field-lookupD)*
from *s fl'* **have** *wf-desc-s: wf-desc s* **by** *(meson field-lookup-wf-desc-pres(1) wf-desc)*
from *s fl'* **have** *wf-sz-s: wf-size-desc s* **by** *(meson field-lookup-wf-size-desc-pres(1) wf-size-desc)*

from *s* **have** *lheap-list': length (heap-list h (size-of TYPE('b)) (ptr-val q)) = size-td s*
using *sz-s* **by** *simp*

from *from-bytes-eq*
field-lookup-access-ti-take-drop [OF fl-s-g wf-fd-s wf-desc-s wf-sz-s lheap-list', of v]
field-lookup-access-ti-take-drop [OF fl-s-g wf-fd-s wf-desc-s wf-sz-s lheap-list', of h-val h p]
have *from-bytes-eq':*
(access-ti s' v (heap-list h (size-of TYPE('c)) (&(q→g)))) =
(access-ti s' (h-val h p) (heap-list h (size-of TYPE('c)) (&(q→g))))
by *(metis add-leD2 add-le-imp-le-diff drop-heap-list-le q-g sz-c-n sz-s' take-heap-list-le)*

have *acc-eq'':*
(access-ti (typ-info-t TYPE('a)) (h-val h p)
(heap-list h (size-of TYPE('a)) (ptr-val p))) = (heap-list h (size-of TYPE('a)) (ptr-val p))
apply *(simp add: h-val-def)*
apply *(simp add: to-bytes-def [symmetric])*
by *(simp add: to-bytes-from-bytes-id)*

show *?thesis*
apply *(simp add: heap-update-def h-val-def)*
apply *(subst (1 2) q-g')*

apply (*subst heap-list-update-list* [*OF sz-bound'*])
apply *simp*
apply (*simp add*: *to-bytes-def*)
apply (*subst acc-eq* [*symmetric*])

apply (*subst from-bytes-eq'*)
apply (*subst acc-eq'*)
apply (*subst acc-eq''*)
apply (*simp add*: *take-drop-eq q-g'*)
done
qed

corollary *ptr-valid-heap-update-subtype-field-access-ti-disj'*:
assumes *val-p*: $d \models_t (p::'a::xmem\text{-}type\ ptr)$
assumes *val-q*: $d \models_t (q::'b::xmem\text{-}type\ ptr)$
assumes *subtype-b-a*: $TYPE('b) \leq_\tau TYPE('a)$
assumes *fl-g*: *field-lookup* (*typ-info-t* $TYPE('b)$) *g 0* = *Some* (*s*, *n*)
assumes *match*: *export-uinfo s* = *typ-uinfo-t* $TYPE('c::xmem\text{-}type)$
assumes *disj* :
$\forall f.\ f \in set$ (*field-names-no-padding* (*typ-info-t* ($TYPE('a)$)) (*export-uinfo* (*typ-info-t* ($TYPE('b)$))))) $\longrightarrow$
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t* $TYPE('a)$) *f 0*)))
v
(*heap-list h* (*size-of* $TYPE('b)$) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t* $TYPE('a)$) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of* $TYPE('b)$) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) ($PTR('c)$ (&(*q*→*g*))) = *h-val h* ($PTR('c)$ (&(*q*→*g*)))
apply (*rule ptr-valid-heap-update-subtype-field-access-ti-disj* [*OF val-p val-q subtype-b-a*])
apply (*simp add*: *match* [*symmetric*] *field-lookup-typ-uinfo-t-Some* [*OF fl-g*])
apply *clarify*
apply (*rule access-ti-field-names-no-padding-to-field-names-u* [*OF val-p val-q subtype-b-a disj*])
apply *assumption*
done

lemma *array-ptr-index-field-lvalue-conv*:
fixes *p*:: ($'a::c\text{-}type['b::finite]$) *ptr*
assumes *i-bound*: $i < CARD('b)$
shows (*array-ptr-index p False i*) = ($PTR('a)$ &(*p*→[*replicate i CHR ''1''*]))
proof −
from *field-lookup-array* [*OF i-bound*, *of 0*, *simplified*]
have *field-lookup* (*typ-info-t* $TYPE('a['b])$) [*replicate i CHR ''1''*] *0* =
Some (*adjust-ti* (*typ-info-t* $TYPE('a)$) (λ*x*. *x*.[*i*]) (λ*x f*. *Arrays.update f i x*),
i ∗ *size-of* $TYPE('a)$) .

from *field-lookup-offset-eq* [*OF this*]
have *field-offset TYPE('a['b]) [replicate i CHR ''1''] = i * size-of TYPE('a)* **.**
then show *?thesis*
by (*simp add*: *field-lvalue-def array-ptr-index-def ptr-add-def*)
qed

lemma *field-lvalue-array-index*:
fixes *p*:: ('*a*::*c-type*['*b*::*finite*]) *ptr*
shows *i < CARD('b) ⟹ &(p→[replicate i CHR ''1'']) =*
ptr-val (*PTR-COERCE*('*a*['*b*] → '*a*) *p* +$_p$ *int i*)
using *array-ptr-index-field-lvalue-conv*[**where** *i=i* **and** *'b='b* **and** *'a='a* **and** *p=p*]
by (*simp add*: *array-ptr-index-def*)

lemma *field-lvalue-array-index'*:
fixes *p*:: ('*a*::*c-type*['*b*::*finite*]) *ptr*
shows *i < CARD('b) ⟹*
PTR('a) &(p→[replicate i CHR ''1'']) =
(*PTR-COERCE*('*a*['*b*] → '*a*) *p* +$_p$ *int i*)
by (*simp add*: *field-lvalue-array-index*)

thm *field-lvalue-append*
corollary *ptr-valid-heap-update-subtype-array-access-ti-disj*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::('*b*::*array-outer-max-size*['*c*::*array-max-count*]) *ptr*)
assumes *subtype-b-a*: *TYPE('b['c]) ≤$_\tau$ TYPE('a)*
assumes *i-bound*: *i < CARD('c)*
assumes *disj* :
∀*f*. *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t* (*TYPE*('*b*['*c*])))))) ⟶
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE('a)*) *f 0*)))
v
(*heap-list h* (*size-of TYPE('b['c])*) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE('a)*) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE('b['c])*) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) (*array-ptr-index q False i*) = *h-val h* (*array-ptr-index q False i*)
proof −
note *fl* = *field-lookup-array* [*OF i-bound*, *of 0*]

have *export-uinfo*
(*adjust-ti* (*typ-info-t TYPE('b)*) (λ*x*. *x*.[*i*]) (λ*x f*. *Arrays.update* (*f*::'*b*['*c*]) *i x*)) = *typ-uinfo-t TYPE('b)*
using *i-bound*
by (*simp*)

from *ptr-valid-heap-update-subtype-field-access-ti-disj'* [*OF val-p val-q subtype-b-a fl this disj*]

have *h-val* (*heap-update p v h*) (*PTR*(*'b*) &(*q*→[*replicate i CHR ''1''*])) =
h-val h (*PTR*(*'b*) &(*q*→[*replicate i CHR ''1''*])).
then show *?thesis*
by (*simp add*: *array-ptr-index-field-lvalue-conv i-bound*)
qed

theorem *ptr-valid-heap-update-subtype-field-access-ti-indep*:
assumes *val-p*: *d* ⊨$_t$ (*p*::*'a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::*'a*::*xmem-type ptr*)
assumes *fl-f*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*(*'b*::*xmem-type*))
assumes *disj*:
(*access-ti t v* (*heap-list h* (*size-of TYPE*(*'b*)) (*ptr-val q*))) =
(*access-ti t* (*h-val h p*) (*heap-list h* (*size-of TYPE*(*'b*)) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) (*PTR*(*'b*) (&(*q*→*f*))) = *h-val h* (*PTR*(*'b*) (&(*q*→*f*)))
proof (*cases ptr-span p* ∩ *ptr-span q* = {})
case *True*
from *fl-f* **have** *ptr-span* (*PTR*(*'b*) (&(*q*→*f*))) ⊆ *ptr-span q*
by (*metis export-uinfo-size field-tag-sub match ptr-val.ptr-val-def size-of-def typ-uinfo-t-def*)
with *True* **have** *ptr-span p* ∩ *ptr-span* (*PTR*(*'b*) (&(*q*→*f*))) = {} **by** *blast*
then show *?thesis*
by (*simp add*: *h-val-update-regions-disjoint*)
next
case *False*
from *val-p val-q False* **have** *pq*: *p* = *q*
by (*meson ptr-valid-same-type-cases*)

from *fl-f* **have** *fl-f'*: *field-ti TYPE*(*'a*) *f* = *Some t*
using *field-lookup-field-ti* **by** *blast*

from *h-val-field-from-bytes'* [*OF fl-f' match* [*simplified typ-uinfo-t-def*], *of h p*]
have *eq1*: *h-val h* (*PTR*(*'b*) &(*p*→*f*)) = *from-bytes* (*access-ti*$_0$ *t* (*h-val h p*)) .

from *h-val-field-from-bytes'* [*OF fl-f' match* [*simplified typ-uinfo-t-def*], *of heap-update p v h p*]
have *eq2*: *h-val* (*heap-update p v h*) (*PTR*(*'b*) &(*p*→*f*)) = *from-bytes* (*access-ti*$_0$ *t* (*h-val* (*heap-update p v h*) *p*)).

thm *field-lookup-access-ti-to-bytes-field-conv* [*OF fl-f match*]

have *eq-upto1*: *eq-upto-padding* (*export-uinfo t*)
(*access-ti t v* (*replicate* (*size-td t*) *0*))
((*access-ti t v* (*replicate* (*size-td t*) *0*)))
by (*metis access-ti*$_0$ *eq-upto-padding-refl export-uinfo-size fd-cons fd-cons-length-p fd-consistentD fl-f*)

have *eq-pad-zeros*: *eq-padding* (*export-uinfo t*) (*replicate* (*size-td t*) *0*) (*replicate* (*size-td t*) *0*)

by *simp*

from *field-lookup-access-ti-eq-upto-padding* [*OF fl-f match*, *of* (*replicate* (*size-td t*) *0*)]
have *eq-upto-padding* (*export-uinfo t*)
(*access-ti t v* (*replicate* (*size-td t*) *0*))
(*access-ti t v* (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
by *simp*

moreover

have *length* (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)) = *size-td t*
by (*simp add*: *export-size-of match*)
from *field-lookup-access-ti-eq-upto-padding* [*OF fl-f match this*]
have *eq-upto-padding* (*export-uinfo t*)
(*access-ti t* (*h-val h p*) (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
(*access-ti t* (*h-val h p*) (*replicate* (*size-td t*) *0*))
by *simp*

ultimately
have *eq-upto*: *eq-upto-padding* (*export-uinfo t*)
(*access-ti t v* (*replicate* (*size-td t*) *0*))
(*access-ti t* (*h-val h p*) (*replicate* (*size-td t*) *0*))
using *disj*
by (*simp add*: *eq-upto-padding-def*)

from *field-lookup-access-ti-eq-padding-value*[*OF fl-f match*, *of* (*replicate* (*size-td t*) *0*)]
have *eq-padding* (*export-uinfo t*)
(*access-ti t v* (*replicate* (*size-td t*) *0*))
(*access-ti t* (*h-val h p*) (*replicate* (*size-td t*) *0*))
by *simp*

from *this eq-upto*
have *acc-eq*: *access-ti t v* (*replicate* (*size-td t*) *0*) =
access-ti t (*h-val h p*) (*replicate* (*size-td t*) *0*)
by (*rule eq-padding-eq-upto-padding-eq*)

show *?thesis*
apply (*simp add*: *pq* [*symmetric*] *eq1 eq2*)
apply (*simp add*: *access-ti*$_0$*-def h-val-heap-update*)

apply (*simp add*: *from-bytes-def*)
apply (*simp add*: *acc-eq*)
done
qed

lemmas *clift-field'* = *clift-field* [*simplified fold-typ-uinfo-t*]

lemma *field-lookup-cons-Some*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
wf-desc t ⟹ *field-lookup t* (*f*#*fs*) *n* = *Some* (*s*, *m*) ⟹
∃ *w k*. *field-lookup t* [*f*] *n* = *Some* (*w*, *k*) ∧ *field-lookup w fs k* = *Some* (*s*, *m*)
wf-desc-struct st ⟹ *field-lookup-struct st* (*f* # *fs*) *n* = *Some* (*s*, *m*) ⟹
∃ *w k*. *field-lookup-struct st* [*f*] *n* = *Some* (*w*, *k*) ∧ *field-lookup w fs k* = *Some* (*s*, *m*)
wf-desc-list ts ⟹ *field-lookup-list ts* (*f*#*fs*) *n* = *Some* (*s*, *m*) ⟹
∃ *w k*. *field-lookup-list ts* [*f*] *n* = *Some* (*w*, *k*) ∧ *field-lookup w fs k* = *Some* (*s*, *m*)
wf-desc-tuple x ⟹ *field-lookup-tuple x* (*f* # *fs*) *n* = *Some* (*s*, *m*) ⟹
f = *dt-snd x* ∧
field-lookup (*dt-fst x*) *fs n* = *Some* (*s*, *m*)
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *n s m f fs* **and** *n s m f fs* **and** *n s m f fs* **and** *n s m f fs*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*

next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc t ts*)
thus *?case* **apply** *clarsimp*
by (*metis Cons-typ-desc.prems*(*1*) *append-Cons append-Nil field-lookup-list-None field-lookup-list-Some*
field-lookup-prefix-Some''(*3*) *fl4 not-Some-eq-tuple*)

next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** (*auto split*: *if-split-asm*)
qed

lemma *field-offset-append*:
assumes *f*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f* 0 = *Some* (*typ-uinfo-t TYPE*('*b*), *n*)
assumes *g*: *field-lookup* (*typ-uinfo-t TYPE*('*b*)) *g* 0 = *Some x*
shows *field-offset TYPE*('*a*::*mem-type*) (*f* @ *g*) =

field-offset (*TYPE*('*a*::*mem-type*)) *f* + *field-offset* (*TYPE*('*b*::*mem-type*)) *g*
proof −
from *field-lookup-prefix-Some''*(*1*) [*rule-format* , *OF f*]
have *field-lookup* (*typ-uinfo-t TYPE*('*a*)) (*f* @ *g*) *0* = *field-lookup* (*typ-uinfo-t TYPE*('*b*)) *g n*
by *simp*
with *f g field-lookup-offset-non-zero* **show** *?thesis*
apply (*simp add*: *field-offset-def field-offset-untyped-def*)
by (*smt* (*verit*, *ccfv-threshold*) *eq-snd-iff field-lookup-offset option.sel*)
qed

lemma *field-lookup-struct-offset-shift*:
field-lookup-struct st f m = *Some* (*s*, *k*) ⟹ *field-lookup-struct st f n* = *Some* (*s*, *k* + *n* − *m*)
by (*metis* (*no-types*, *lifting*) *add.commute field-lookup-offset'*(*2*) *field-lookup-offset-le*(*2*)
le-add-diff-inverse ordered-cancel-comm-monoid-diff-class.diff-add-assoc2)

lemma *field-lookup-list-offset-shift*:
field-lookup-list ts f m = *Some* (*s*, *k*) ⟹ *field-lookup-list ts f n* = *Some* (*s*, *k* + *n* − *m*)
by (*metis Nat.add-diff-assoc2 add.commute field-lookup-list-offset field-lookup-list-offset2 field-lookup-offset-le*(*3*))

lemma *field-lookup-tuple-offset-shift*:
field-lookup-tuple x f m = *Some* (*s*, *k*) ⟹ *field-lookup-tuple x f n* = *Some* (*s*, *k* + *n* − *m*)
by (*metis* (*no-types*, *lifting*) *add.commute field-lookup-offset'*(*4*) *field-lookup-offset-le*(*4*) *ordered-cancel-comm-monoid-diff-class.add-diff-assoc2*
ordered-cancel-comm-monoid-diff-class.add-diff-inverse)

lemma *h-val-field-update*:
fixes *p*::'*a*::*mem-type ptr*
and *q*::'*a*::*mem-type ptr*
and *v*:: '*b*::*mem-type*
assumes *fl*: *field-ti TYPE*('*a*) *f* = *Some t*
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b*)
and *valid-p*: *hrs-htd h* ⊨$_t$ *p*
and *valid-q*: *hrs-htd h* ⊨$_t$ *q*
shows (*h-val* (*heap-update* (*Ptr* &(*p*→*f*)) *v* (*hrs-mem h*)) *q*) =
(*if p* = *q then field-update* (*field-desc t*) (*to-bytes-p v*) (*h-val* (*hrs-mem h*) *p*) *else* (*h-val* (*hrs-mem h*) *q*))
proof −
from *clift-Some-eq-valid valid-p* **obtain** *z* **where** *z*: *clift h p* = *Some z* **by** *metis*

from *clift-field-update* [*OF fl match* [*simplified typ-uinfo-t-def*] *this*]
show *?thesis*
apply *simp*

by (*smt* (*verit*) *fun-upd-apply h-val-clift'* *hrs-htd-def hrs-mem-update lift-t-if prod.collapse valid-q z*)
qed

theorem *root-ptr-valid-field-disj-type-h-val-heap-update*:
assumes *valid-p*: *root-ptr-valid* (*hrs-htd h*) (*p*::'*b*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid* (*hrs-htd h*) (*q*::'*a*::*mem-type ptr*)
assumes *neq*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
assumes *f*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*s*, *n*)
assumes *match*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*c*))
shows *h-val* (*heap-update* (*PTR*('*c*::*mem-type*) &(*q*→*f*)) *v* (*hrs-mem h*)) *p* =
h-val (*hrs-mem h*) *p*
proof −
from *root-ptr-valid-type-neq-disjoint* [*OF valid-q valid-p neq*]
have *disj*: *ptr-span q* ∩ *ptr-span p* = {} **.**

from *export-uinfo-size* [*of s*] **have** *size-td s* = *size-td* (*typ-uinfo-t TYPE*('*c*)) **by** (*simp add*: *match*)
moreover have *size-td* (*typ-info-t TYPE*('*c*)) = *size-td* (*typ-uinfo-t* (*TYPE*('*c*)))
by (*simp add*: *typ-uinfo-t-def*)
ultimately have *sz*: *size-td s* = *size-of* (*TYPE*('*c*))
unfolding *size-of-def*
by *metis*

from *field-tag-sub* [*OF f*] **have** {&(*q*→*f*)..+*size-td s*} ⊆ *ptr-span q* **.**

with *disj sz*
have *ptr-span* (*PTR*('*c*::*mem-type*) &(*q*→*f*)) ∩ *ptr-span p* = {}
apply (*simp*)
by *blast*
from *h-val-update-regions-disjoint* [*OF this*]
show *?thesis* **.**
qed

lemma *ptr-empty-qualified-field-name-conv*:
fixes *p*::'*a*::*c-type ptr*
shows *p* = *PTR* ('*a*) &(*p*→[])
by (*simp add*: *field-lvalue-def*)

theorem *root-ptr-valid-field-disj-h-val-heap-update-eq*:
assumes *valid-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid d* (*q*::'*b*::*mem-type ptr*)
assumes *f*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*t*, *n*)
assumes *g*: *field-lookup* (*typ-info-t* (*TYPE*('*b*))) *g 0* = *Some* (*s*, *m*)
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*c*))

assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*d*))
assumes *disj*: *typ-uinfo-t* (*TYPE*('*a*)) = (*typ-uinfo-t* (*TYPE*('*b*))) ⟹
ptr-val p = *ptr-val q* ⟹
ptr-span (*PTR*('*c*::*mem-type*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*d*::*mem-type*) &(*q*→*g*)) = {}
shows *h-val* (*heap-update* (*PTR*('*c*::*mem-type*) &(*p*→*f*)) *v h*) (*PTR*('*d*::*mem-type*) &(*q*→*g*)) =
h-val h (*PTR*('*d*::*mem-type*) &(*q*→*g*))
proof −
from *f match-t*
have *sub-f-p*: *ptr-span* (*PTR*('*c*::*mem-type*) &(*p*→*f*)) ⊆ *ptr-span p*
using *export-size-of field-tag-sub* **by** *fastforce*
from *g match-s*
have *sub-g-q*: *ptr-span* (*PTR*('*d*::*mem-type*) &(*q*→*g*)) ⊆ *ptr-span q*
using *export-size-of field-tag-sub* **by** *fastforce*
show *?thesis*
proof (*cases typ-uinfo-t* (*TYPE*('*a*)) = *typ-uinfo-t* (*TYPE*('*b*)))
case *True*
note *typ-eq* = *this*
show *?thesis*
proof (*cases ptr-val p* = *ptr-val q*)
case *True*
from *disj* [*OF typ-eq True*]
have *ptr-span* (*PTR*('*c*::*mem-type*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*d*::*mem-type*) &(*q*→*g*)) = {} **by** *blast*
then show *?thesis*
by (*simp add*: *h-val-update-regions-disjoint*)
next
case *False*
from *root-ptr-valid-neq-disjoint* [*OF valid-p valid-q False*]
have *ptr-span p* ∩ *ptr-span q* = {}.
with *sub-f-p sub-g-q*
have *ptr-span* (*PTR*('*c*::*mem-type*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*d*::*mem-type*) &(*q*→*g*)) = {} **by** *blast*
then show *?thesis*
by (*simp add*: *h-val-update-regions-disjoint*)
qed
next
case *False*
from *root-ptr-valid-type-neq-disjoint* [*OF valid-p valid-q False*]
have *ptr-span p* ∩ *ptr-span q* = {}.
with *sub-f-p sub-g-q*
have *ptr-span* (*PTR*('*c*::*mem-type*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*d*::*mem-type*) &(*q*→*g*)) = {} **by** *blast*
then show *?thesis*
by (*simp add*: *h-val-update-regions-disjoint*)
qed
qed

definition *PTR-MATCH p q* = (*p* = *q*)

lemma *PTR-MATCH-field*:
PTR-MATCH (*PTR*('*a*::*c-type*) &(*p*→*f*)) (*PTR*('*a*) &(*p*→*f*))
by (*simp add*: *PTR-MATCH-def*)

lemma *PTR-MATCH-dummy*:
PTR-MATCH (*p*::'*a*::*c-type ptr*) (*PTR*('*a*) &(*p*→[]))
by (*simp add*: *PTR-MATCH-def*)

theorem *root-ptr-valid-field-disj-h-val-heap-update-eq'*:
assumes *p'*: *PTR-MATCH p'* (*PTR*('*c*) &(*p*→*f*))
assumes *q'*: *PTR-MATCH q'* (*PTR*('*d*) &(*q*→*g*))
assumes *valid-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid d* (*q*::'*b*::*mem-type ptr*)
assumes *disj*: *typ-uinfo-t* (*TYPE*('*a*)) = (*typ-uinfo-t* (*TYPE*('*b*))) ⟹
ptr-val p = *ptr-val q* ⟹
ptr-span (*PTR*('*c*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*d*) &(*q*→*g*)) = {}
assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*d*::*mem-type*))
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*c*::*mem-type*))
assumes *g*: *field-lookup* (*typ-info-t* (*TYPE*('*b*))) *g 0* = *Some* (*s*, *n*)
assumes *f*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*t*, *m*)
shows *h-val* (*heap-update p' v h*) *q'* = *h-val h q'*
using *root-ptr-valid-field-disj-h-val-heap-update-eq* [*OF valid-p valid-q f g match-t match-s disj*] *p' q'*
by (*simp add*: *PTR-MATCH-def*)

lemma *field-lookup-single-field-disj'*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
field-lookup t [*f*] *n* = *Some* (*s*, *n* + *m*) ⟹
field-lookup t [*g*] *n* = *Some* (*s'*, *n* + *k*) ⟹ *f* ≠ *g* ⟹
{*m* ..< *m* + *size-td s*} ∩ {*k* ..<*k* + *size-td s'*} = {}
field-lookup-struct st [*f*] *n* = *Some* (*s*, *n* + *m*) ⟹
field-lookup-struct st [*g*] *n* = *Some* (*s'*, *n* + *k*) ⟹ *f* ≠ *g* ⟹
{*m* ..< *m* + *size-td s*} ∩ {*k* ..< *k* + *size-td s'*} = {}
field-lookup-list ts [*f*] *n* = *Some* (*s*, *n* + *m*) ⟹
field-lookup-list ts [*g*] *n* = *Some* (*s'*, *n* + *k*) ⟹ *f* ≠ *g* ⟹
{*m* ..< *m* + *size-td s*} ∩ {*k* ..< *k* + *size-td s'*} = {}
field-lookup-tuple x [*f*] *n* = *Some* (*s*, *n* + *m*) ⟹
field-lookup-tuple x [*g*] *n* = *Some* (*s'*, *n* + *k*) ⟹ *f* ≠ *g* ⟹
{*m* ..< *m* + *size-td s*} ∩ {*k* ..< *k* + *size-td s'*} = {}

```
proof (induct t and st and ts and x arbitrary: n s s' m f g k and n s s' m f g k
and n s s' m f g k and n s s' m f g k)
  case (TypDesc nat typ-struct list)
then show ?case by auto
next
  case (TypScalar nat1 nat2 a)
  then show ?case by auto
next
  case (TypAggregate list)
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by auto
next
  case (Cons-typ-desc x fs)
  obtain d nm y where x: x = DTuple d nm y by (cases x) auto
  show ?case
  proof (cases nm=f)
    case True
    from True Cons-typ-desc.prems  obtain
      fl-g: field-lookup-list fs [g] (n + size-td s) = Some (s', n + k) and
      neq: f ≠ g and
      d-s: d = s and m: m = 0
      by (auto simp add: x split: if-split-asm)
    from field-lookup-offset-le(3) [OF fl-g] have n + size-td s ≤ n + k .
    hence size-td s ≤ k by simp
    then show ?thesis by (simp add: m)
  next
    case False
    note not-nm-f = this
    show ?thesis
    proof (cases nm = g)
      case True
      from True Cons-typ-desc.prems  obtain
        fl-f: field-lookup-list fs [f] (n + size-td s') = Some (s, n + m) and
        neq: f ≠ g
        d = s' and
        k: k = 0
        by  (auto simp add: x split: if-split-asm)
      from field-lookup-offset-le(3) [OF fl-f] have n + size-td s' ≤ n + m .
      hence size-td s' ≤ m by simp
      then show ?thesis by (simp add: k)

    next
      case False
        from False not-nm-f Cons-typ-desc.prems obtain
          fl-f: field-lookup-list fs [f] (n + size-td d) = Some (s, n + m) and
          fl-g: field-lookup-list fs [g] (n + size-td d) = Some (s', n + k) and
          neq: f ≠ g
```

```
      by (auto simp add: x split: if-split-asm)
    from field-lookup-list-offset-shift [OF fl-f, of n, simplified]
    have fl-f': field-lookup-list fs [f] n = Some (s, m + n − size-td d) .
    from field-lookup-offset-le(3)[OF fl-f]
    have bounds1: n + size-td d ≤ n + m.
    then have eq1: m + n − size-td d = n + (m − size-td d)
      by simp

    from field-lookup-list-offset-shift [OF fl-g, of n, simplified]
    have fl-g': field-lookup-list fs [g] n = Some (s', k + n − size-td d) .
    from field-lookup-offset-le(3)[OF fl-g]
    have bounds2: n + size-td d ≤ n + k.
    then have eq2: k + n − size-td d = n + (k − size-td d)
      by simp
     from Cons-typ-desc.hyps(2) [OF fl-f' [simplified eq1] fl-g' [simplified eq2]
neq ]
    have {m − size-td d..<m − size-td d + size-td s} ∩
          {k − size-td d..<k − size-td d + size-td s'} = {} .
    then show ?thesis using bounds1 bounds2 by auto
  qed
 qed

next
 case (DTuple-typ-desc typ-desc list b)
 then show ?case by (auto split: option.splits if-split-asm)
qed

lemma field-lookup-single-field-disj:
 assumes fl-f: field-lookup t [f] 0 = Some (s, m)
 assumes fl-g: field-lookup t [g] 0 = Some (s', k)
 assumes neq: f ≠ g
 shows {m ..<m + size-td s} ∩ {k ..<k + size-td s'} = {}
 using field-lookup-single-field-disj'(1) [where n=0, simplified, OF fl-f fl-g neq ]
 by simp



lemma field-lookup-non-prefix-disj:
 assumes wf: wf-desc t
 assumes f: field-lookup t f 0 = Some (tf, n)
 assumes g: field-lookup t g 0 = Some (tg, m)
 assumes g-f:¬ prefix g f
 assumes f-g: ¬ prefix f g
 shows {n ..<n + size-td tf} ∩ {m ..<m + size-td tg} = {}
 using wf f g g-f f-g
 proof (induct f arbitrary: t tf n g tg m  )
  case Nil
  then show ?case by simp
 next
```

case (*Cons f1 fs*)
from *Cons.prems* **obtain**
wf: *wf-desc t* **and**
fl-f: *field-lookup t* (*f1* # *fs*) *0* = *Some* (*tf*, *n*) **and**
fl-g: *field-lookup t g 0* = *Some* (*tg*, *m*) **and**
not-prefix-g-f: ¬ *prefix g* (*f1* # *fs*) **and**
not-prefix-f-g: ¬ *prefix* (*f1* # *fs*) *g* **by** *simp*

from *field-lookup-append-Some* [*OF wf*, **where** *f*=[*f1*] **and** *g*=*fs*, *simplified*, *OF fl-f*]
obtain *w k* **where** *fl-f1*: *field-lookup t* [*f1*] *0* = *Some* (*w*, *k*) **and** *fl-fs*: *field-lookup w fs k* = *Some* (*tf*, *n*)
by *blast*

from *field-lookup-wf-desc-pres*(*1*) [*OF wf fl-f1*] **have** *wf-w*: *wf-desc w*.
show *?case*
proof (*cases g*)
case *Nil*
with *Cons.prems* **show** *?thesis* **by** *simp*
next
case (*Cons g1 gs*)
from *field-lookup-append-Some* [*OF wf*, **where** *f*=[*g1*] **and** *g*=*gs*, *simplified*, *OF fl-g* [*simplified Cons*]]
obtain *v l* **where** *fl-g1*: *field-lookup t* [*g1*] *0* = *Some* (*v*, *l*) **and** *fl-gs*: *field-lookup v gs l* = *Some* (*tg*, *m*)
by *blast*
show *?thesis*
proof (*cases f1* = *g1*)
case *True*
with *fl-f1 fl-g1 fl-gs* **obtain**
v-w: *v*=*w* **and** *k-l*: *k*=*l* **and**
fl-g1': *field-lookup t* [*g1*] *0* = *Some* (*w*, *k*) **and** *fl-gs'*: *field-lookup w gs k* = *Some* (*tg*, *m*)
by *auto*
from *not-prefix-g-f Cons True* **have** *not-prefix-gs-fs*: ¬ *prefix gs fs* **by** *auto*
from *not-prefix-f-g Cons True* **have** *not-prefix-fs-gs*: ¬ *prefix fs gs* **by** *auto*

from *field-lookup-offset-le*(*1*) [*OF fl-fs*] **have** *le-k-n*: *k* ≤ *n*.
from *field-lookup-offset-shift'* [*OF fl-fs*, *of 0*, *simplified*]
have *fl-fs'*: *field-lookup w fs 0* = *Some* (*tf*, *n* − *k*).

from *field-lookup-offset-le*(*1*) [*OF fl-gs'*] **have** *le-k-m*: *k* ≤ *m*.
from *field-lookup-offset-shift'* [*OF fl-gs'*, *of 0*, *simplified*]
have *fl-gs'*: *field-lookup w gs 0* = *Some* (*tg*, *m* − *k*).

from *Cons.hyps* [*OF wf-w fl-fs' fl-gs' not-prefix-gs-fs not-prefix-fs-gs*]
have {*n* − *k*..<*n* − *k* + *size-td tf*} ∩ {*m* − *k*..<*m* − *k* + *size-td tg*} = {}

.
      with *le-k-n le-k-m* **show** *?thesis*
        by *auto*
    next
      case *False*
      from *field-lookup-single-field-disj* [*OF fl-f1 fl-g1 False*]
      have *disj*: $\{k..<k + \textit{size-td } w\} \cap \{l..<l + \textit{size-td } v\} = \{\}$ **.**
      from *field-lookup-offset-le*(*1*) [*OF fl-fs*] **have** *k-n*: $k \leq n$ **.**
      from *field-lookup-offset-le*(*1*) [*OF fl-gs*] **have** *l-m*: $l \leq m$ **.**
      from *field-lookup-offset-shift'* [*OF fl-fs*, *of 0*, *simplified*]
      have *fl-fs'*: *field-lookup w fs 0* = *Some* (*tf*, $n - k$) **.**
      from *field-lookup-offset-shift'* [*OF fl-gs*, *of 0*, *simplified*]
      have *fl-gs'*: *field-lookup v gs 0* = *Some* (*tg*, $m - l$) **.**
       from *field-lookup-offset-size'* [*OF fl-fs'*] **have** *sz1*: *size-td tf* + $(n - k) \leq$ *size-td w* **.**
      from *field-lookup-offset-size'* [*OF fl-gs'*] **have** *sz2*: *size-td tg* + $(m - l) \leq$ *size-td v* **.**

      show *?thesis*
        using *disj k-n l-m sz1 sz2*
        by *auto*
    qed
  qed
qed

theorem *root-ptr-valid-non-prefix-disj*:
  assumes *valid-p*: *root-ptr-valid d* (*p*::*'a*::*mem-type ptr*)
  assumes *f*: *field-lookup* (*typ-info-t* (*TYPE*(*'a*::*mem-type*))) *f 0* = *Some* (*t*, *n*)
  assumes *g*: *field-lookup* (*typ-info-t* (*TYPE*(*'a*::*mem-type*))) *g 0* = *Some* (*s*, *m*)
  assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*(*'c*))
  assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*(*'d*))
  assumes *non-prefix-f-g*: ¬ *prefix f g*
  assumes *non-prefix-g-f*: ¬ *prefix g f*
  shows *ptr-span* (*PTR*(*'c*::*mem-type*) *&*(*p*→*f*)) ∩ *ptr-span* (*PTR*(*'d*::*mem-type*) *&*(*p*→*g*)) = {}
proof −
  from *field-lookup-non-prefix-disj* [*OF wf-desc f g non-prefix-g-f non-prefix-f-g*]
  have *disj*: $\{n..<n + \textit{size-td } t\} \cap \{m..<m + \textit{size-td } s\} = \{\}$ **.**

  from *match-t* **have** *sz-t*: *size-td t* = *size-of TYPE*(*'c*)
    using *export-size-of* **by** *force*
  from *match-s* **have** *sz-s*: *size-td s* = *size-of TYPE*(*'d*)
    using *export-size-of* **by** *force*
  from *f* **have** *off-f*: *field-offset TYPE*(*'a*) *f* = *n*
    using *field-lookup-offset-eq* **by** *blast*
  from *g* **have** *off-g*: *field-offset TYPE*(*'a*) *g* = *m*
    using *field-lookup-offset-eq* **by** *blast*
  have *le-addr-card*: *size-td* (*typ-info-t TYPE*(*'a*)) < *addr-card*

```
    by (metis max-size size-of-def)
  from field-lookup-offset-size' [OF f] have t-bound: size-td t + n ≤ size-td (typ-info-t
TYPE('a)).
   with le-addr-card have t-bound': size-td t + n < addr-card
    by simp
   from field-lookup-offset-size' [OF g] have s-bound: size-td s + m ≤ size-td
(typ-info-t TYPE('a)).
   with le-addr-card have s-bound': size-td s + m < addr-card
    by simp

  show ?thesis
   using disj t-bound' s-bound'
   apply (simp add: sz-t sz-s field-lvalue-def off-f off-g intvl-def)
   apply (safe; clarsimp simp add: unat-of-nat-eq)
   subgoal premises prems for k ka
   proof −
    from prems have n + k = m + ka
     by (metis (no-types, opaque-lifting) add.commute len-of-addr-card
        less-trans nat-add-left-cancel-less of-nat-add sz-t t-bound' unat-of-nat-eq)
    with prems show False by simp
   qed
   subgoal premises prems for k ka
   proof −
    from prems have n + k = m + ka
     by (metis (no-types, opaque-lifting) add.commute len-of-addr-card
       less-trans nat-add-left-cancel-less of-nat-add s-bound' sz-s unat-of-nat-eq)
    with prems show ?thesis by simp
   qed
   done
 qed

theorem root-ptr-valid-non-prefix-h-val-heap-update-eq':

  assumes p': PTR-MATCH p' (PTR('c) &(p→f))
  assumes q': PTR-MATCH q' (PTR('d) &(q→g))
  assumes valid-p: root-ptr-valid d (p::'a::mem-type ptr)
  assumes valid-q: root-ptr-valid d (q::'b::mem-type ptr)
  assumes disj: typ-uinfo-t (TYPE('a)) = (typ-uinfo-t (TYPE('b))) ⟹
              ptr-val p = ptr-val q ⟹
              ¬ prefix f g ∧ ¬ prefix g f
  assumes match-s: export-uinfo s = typ-uinfo-t (TYPE('d::mem-type))
  assumes match-t: export-uinfo t = typ-uinfo-t (TYPE('c::mem-type))
  assumes g: field-lookup (typ-info-t (TYPE('b))) g 0 = Some (s, n)
  assumes f: field-lookup (typ-info-t (TYPE('a))) f 0 = Some (t, m)
  shows h-val (heap-update p' v h) q' = h-val h q'
proof −
  {
   assume typ-eq: typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b)
   assume ptr-val-eq: ptr-val p = ptr-val q
```

```
  have ptr-span (PTR('c) &(p→f)) ∩ ptr-span (PTR('d) &(q→g)) = {}
  proof −
    from disj [OF typ-eq ptr-val-eq] obtain
      non-prefix-f-g: ¬ prefix f g  and non-prefix-g-f: ¬ prefix g f by blast

    from typ-eq ptr-val-eq g match-s ptr-val-eq
    have span-eq: (PTR('d) &(q→g)) = (PTR('d) &(p→g))
      by (simp add: field-lvalue-def field-offset-def)

    from field-lookup-typ-uinfo-t-Some [OF g] typ-eq
    have fl-u: field-lookup (typ-uinfo-t TYPE('a)) g 0 = Some (export-uinfo s, n)
      by simp
    from field-lookup-export-uinfo-Some-rev [OF fl-u [simplified typ-uinfo-t-def]]
     match-s
    obtain s' where fl-g': field-lookup (typ-info-t TYPE('a)) g 0 = Some (s', n)
and
       match-s': export-uinfo s' = typ-uinfo-t TYPE('d)
      by auto

   from root-ptr-valid-non-prefix-disj [OF valid-p f fl-g' match-t match-s' non-prefix-f-g
non-prefix-g-f]
    show ?thesis by (simp add: span-eq)
  qed
 } note disj = this
  from root-ptr-valid-field-disj-h-val-heap-update-eq' [OF p' q' valid-p valid-q disj
match-s match-t g f]
 show ?thesis by simp
qed


lemma field-tag-sub':
 assumes fl: field-lookup (typ-uinfo-t TYPE('a::mem-type)) f 0 = Some (t, n)
 shows {&(p::'a ptr→f)..+size-td t} ⊆ ptr-span p
proof −
 from field-lookup-uinfo-Some-rev [OF fl] obtain s where
   fl': field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n) and s-t:  export-uinfo
s = t
   by auto
 from field-tag-sub [OF fl']
 have {&(p→f)..+size-td s} ⊆ ptr-span p .
 with export-uinfo-size s-t
 show {&(p→f)..+size-td t} ⊆ ptr-span p by auto
qed


lemma all-field-names-field-lookup:
 assumes f ∈ set (all-field-names (typ-uinfo-t TYPE('a::mem-type)))
 shows ∃ t n. field-lookup (typ-uinfo-t TYPE('a::mem-type)) f 0 = Some (t, n)
 using all-field-names-union-field-names-u-conv field-names-u-field-names-export-uinfo-conv(1)
```

field-names-SomeD assms
by (*metis all-field-names-exists-field-names-u*(*1*) *field-lookup-export-uinfo-Some not-Some-eq-tuple typ-uinfo-t-def*)

lemma *field-lvalue-same-type-conv*:
assumes *valid-p*: *d*,*g* ⊨$_t$ (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *d*,*g* ⊨$_t$ (*q*::'*a*::*mem-type ptr*)
assumes *f1*: *f1* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))
shows (&(*p*→*f1*) = &(*q*→*f2*)) ⟷ (*p* = *q* ∧ *field-offset* (*TYPE*('*a*)) *f1* = *field-offset TYPE*('*a*) *f2*)
proof (*cases rule*: *ptr-valid-same-type-cases* [*OF valid-p valid-q*, *case-names Same Disj*])
case *Same*
with *f1 f2* **show** *?thesis*
apply (*simp add*: *field-lvalue-def*)
apply (*metis all-field-names-exists-field-names-u*(*1*) *field-names-SomeD2 field-names-u-field-names-export-uinfo-conv*(*1*) *typ-uinfo-t-def unat-of-nat-field-offset wf-desc*)
done
next
case *Disj*
from *all-field-names-field-lookup* [*OF f1*]
obtain *t1 n1* **where** *fl1*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f1 0* = *Some* (*t1*, *n1*)
by *blast*
from *all-field-names-field-lookup* [*OF f2*]
obtain *t2 n2* **where** *fl2*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f2 0* = *Some* (*t2*, *n2*)
by *blast*

from *field-tag-sub'* [*OF fl1*, *of p*] *field-tag-sub'* [*OF fl2*, *of q*] *Disj*
have {&(*p*→*f1*)..+*size-td t1*} ∩ {&(*q*→*f2*)..+*size-td t2*} = {}
by *blast*
moreover
from *field-lookup-wf-size-desc-gt* [*OF fl1*] **have** *0* < *size-td t1* **by** *simp*
then have &(*p*→*f1*) ∈ {&(*p*→*f1*)..+*size-td t1*}
using *first-in-intvl* **by** *blast*
moreover
from *field-lookup-wf-size-desc-gt* [*OF fl2*] **have** *0* < *size-td t2* **by** *simp*
then have &(*q*→*f2*) ∈ {&(*q*→*f2*)..+*size-td t2*}
using *first-in-intvl* **by** *blast*
ultimately
have &(*p*→*f1*) ≠ &(*q*→*f2*)
by *auto*
thus *?thesis*
by (*auto simp add*: *field-lvalue-def field-offset-def field-offset-untyped-def size-of-def fl1 fl2*)
qed

lemma *field-lvalue-same-root-conv*:
assumes *f1*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f1 0* = *Some*(*t1*, *n1*)

assumes *f2*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f2 0* = *Some*(*t2*, *n2*)

shows (&(*p*::'*a ptr*→*f1*) = &(*p*→*f2*)) ⟷ (*n1* = *n2*)
proof −
have *max*: *size-of TYPE*('*a*) < *addr-card* **by** (*rule max-size*)
from *f1*
have *n1-bound*: *n1* < *addr-card*
using *field-lookup-offset-eq field-offset-addr-card* **by** *blast*

from *f2*
have *n2-bound*: *n2* < *addr-card*
using *field-lookup-offset-eq field-offset-addr-card* **by** *blast*
from *n1-bound n2-bound field-lookup-offset-size* [*OF f1*] *field-lookup-offset-size* [*OF f2*]
show *?thesis*
apply (*simp add*: *field-lvalue-def field-lookup-offset-eq* [*OF f1*] *field-lookup-offset-eq* [*OF f2*])
by (*metis len-of-addr-card unat-of-nat-eq*)
qed

lemma *field-lvalue-same-type-conv'*:
assumes *valid-p*: *d*,*g* ⊨$_t$ (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *d*,*g* ⊨$_t$ (*q*::'*a*::*mem-type ptr*)
assumes *f1*: *f1* ∈ *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*)))
shows (&(*p*→*f1*) = &(*q*→*f2*)) ⟷ (*p* = *q* ∧ *field-offset* (*TYPE*('*a*)) *f1* = *field-offset TYPE*('*a*) *f2*)
proof −
have *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*))) ⊆ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))
using *subset-all-field-names-no-padding-all-field-names*
all-field-names-typ-uinfo-t-conv
by *metis*
then
show *?thesis* **using** *field-lvalue-same-type-conv* [*OF valid-p valid-q*] *f1 f2*
by *blast*
qed

lemma *field-lvalue-same-root-conv'*:
assumes *f1*: *f1* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*::*mem-type*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))
shows (&(*p*::'*a ptr*→*f1*) = &(*p*→*f2*)) ⟷ (*field-offset* (*TYPE*('*a*)) *f1* = *field-offset TYPE*('*a*) *f2*)
proof −
from *all-field-names-field-lookup* [*OF f1*]

obtain *t1 n1* **where** *fl1*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f1 0* = *Some* (*t1*, *n1*)
by *blast*
from *all-field-names-field-lookup* [*OF f2*]
obtain *t2 n2* **where** *fl2*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f2 0* = *Some* (*t2*, *n2*)
by *blast*

have *max*: *size-of TYPE*('*a*) < *addr-card* **by** (*rule max-size*)

from *field-lookup-offset-size'* [*OF fl1*]
have *n1* < *size-of TYPE*('*a*)
apply (*simp add*: *size-of-def*)
by (*metis add-diff-cancel-right' add-leD2 cancel-comm-monoid-add-class.diff-cancel field-lookup-wf-size-desc-gt*
less-le local.fl1 not-add-less2 wf-size-desc-typ-uinfo-t-simp)
with *max* **have** *n1* < *addr-card* **by** *arith*

moreover
from *field-lookup-offset-size'* [*OF fl2*]
have *n2* < *size-of TYPE*('*a*)
apply (*simp add*: *size-of-def*)
by (*metis add-diff-cancel-right' add-leD2 cancel-comm-monoid-add-class.diff-cancel field-lookup-wf-size-desc-gt*
le-imp-less-or-eq less-not-refl2 local.fl2 not-add-less2 wf-size-desc-typ-uinfo-t-simp)
with *max* **have** *n2* < *addr-card* **by** *arith*

ultimately

show *?thesis*
apply (*simp add*: *field-offset-def field-offset-untyped-def fl1 fl2 field-lvalue-def size-of-def*)
by (*metis len-of-addr-card unat-of-nat-eq*)

qed

lemma *field-lvalue-same-root-conv''*:
assumes *f1*: *f1* ∈ *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*::*mem-type*)))

assumes *f2*: *f2* ∈ *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*)))
shows (&(*p*::'*a ptr*→*f1*) = &(*p*→*f2*)) ⟷ (*field-offset* (*TYPE*('*a*)) *f1* = *field-offset TYPE*('*a*) *f2*)
proof −
have *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*::*mem-type*))) ⊆
set (*all-field-names* (*typ-uinfo-t TYPE*('*a*::*mem-type*)))
using *subset-all-field-names-no-padding-all-field-names*
all-field-names-typ-uinfo-t-conv

by *metis*
with *field-lvalue-same-root-conv'* *f1* *f2* **show** *?thesis*
by *blast*
qed

lemma *disj-ptr-span-field-neq*:
assumes *disj*: *ptr-span* (*p*::'*a*::*mem-type ptr*) ∩ *ptr-span* (*q*::'*b*::*mem-type ptr*) = {}
assumes *fl1*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f1 0* = *Some* (*t1*, *n1*)
assumes *fl2*: *field-lookup* (*typ-info-t TYPE*('*b*)) *f2 0* = *Some* (*t2*, *n2*)
shows &(*p*→*f1*) ≠ &(*q*→*f2*)
proof −
from *field-tag-sub* [*OF fl1*, *of p*] **have** {&(*p*→*f1*)..+*size-td t1*} ⊆ *ptr-span p* .
moreover
from *field-tag-sub* [*OF fl2*, *of q*] **have** {&(*q*→*f2*)..+*size-td t2*} ⊆ *ptr-span q* .
moreover
from *field-lookup-wf-size-desc-gt* [*OF fl1*] **have** *0* < *size-td t1* **by** *simp*
then have &(*p*→*f1*) ∈ {&(*p*→*f1*)..+*size-td t1*}
using *first-in-intvl* **by** *blast*
moreover
from *field-lookup-wf-size-desc-gt* [*OF fl2*] **have** *0* < *size-td t2* **by** *simp*
then have &(*q*→*f2*) ∈ {&(*q*→*f2*)..+*size-td t2*}
using *first-in-intvl* **by** *blast*
ultimately
show &(*p*→*f1*) ≠ &(*q*→*f2*)
using *disj*
by *force*
qed

lemma *disj-ptr-span-field-neq''*:
assumes *disj*: *ptr-span* (*p*::'*a*::*mem-type ptr*) ∩ *ptr-span* (*q*::'*b*::*mem-type ptr*) = {}
assumes *f1*: *f1* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*b*)))
shows &(*p*→*f1*) ≠ &(*q*→*f2*)
proof −
from *all-field-names-field-lookup* [*OF f1*]
obtain *t1 n1* **where** *fl1*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f1 0* = *Some* (*t1*, *n1*)
by *blast*
from *all-field-names-field-lookup* [*OF f2*]
obtain *t2 n2* **where** *fl2*: *field-lookup* (*typ-uinfo-t TYPE*('*b*)) *f2 0* = *Some* (*t2*, *n2*)
by *blast*
from *field-tag-sub'* [*OF fl1*, *of p*] **have** {&(*p*→*f1*)..+*size-td t1*} ⊆ *ptr-span p* .
moreover
from *field-tag-sub'* [*OF fl2*, *of q*] **have** {&(*q*→*f2*)..+*size-td t2*} ⊆ *ptr-span q* .
moreover

from *field-lookup-wf-size-desc-gt* [*OF fl1*] **have** *0 < size-td t1* **by** *simp*
then have *&(p→f1) ∈ {&(p→f1)..+size-td t1}*
using *first-in-intvl* **by** *blast*
moreover
from *field-lookup-wf-size-desc-gt* [*OF fl2*] **have** *0 < size-td t2* **by** *simp*
then have *&(q→f2) ∈ {&(q→f2)..+size-td t2}*
using *first-in-intvl* **by** *blast*
ultimately
show *&(p→f1) ≠ &(q→f2)*
using *disj*
by *force*
qed

lemma *disj-ptr-span-field-neq'*:
assumes *disj*: *ptr-span (p::'a::mem-type ptr) ∩ ptr-span (q::'b::mem-type ptr) = {}*
assumes *f1*: *f1 ∈ set (all-field-names-no-padding (typ-info-t TYPE('a)))*
assumes *f2*: *f2 ∈ set (all-field-names-no-padding (typ-info-t TYPE('b)))*
shows *&(p→f1) ≠ &(q→f2)*
proof −
have *set (all-field-names-no-padding (typ-info-t TYPE('a))) ⊆ set (all-field-names (typ-uinfo-t TYPE('a)))*
using *subset-all-field-names-no-padding-all-field-names all-field-names-typ-uinfo-t-conv*
by *metis*
moreover
have *set (all-field-names-no-padding (typ-info-t TYPE('b))) ⊆ set (all-field-names (typ-uinfo-t TYPE('b)))*
using *subset-all-field-names-no-padding-all-field-names all-field-names-typ-uinfo-t-conv*
by *metis*
ultimately
show *?thesis* **using** *disj f1 f2 disj-ptr-span-field-neq''* **by** *blast*
qed

lemma *disj-ptr-span-field-disj-ptr-span*:
assumes *disj*: *ptr-span (p::'a::mem-type ptr) ∩ ptr-span (q::'b::mem-type ptr) = {}*
assumes *f1*: *f1 ∈ set (field-names-u (typ-uinfo-t TYPE('a)) (typ-uinfo-t (TYPE('c::mem-type))))*
assumes *f2*: *f2 ∈ set (field-names-u (typ-uinfo-t TYPE('b)) (typ-uinfo-t (TYPE('d::mem-type))))*
shows *ptr-span (PTR('c) &(p→f1)) ∩ ptr-span (PTR('d) (&(q→f2))) = {}*
proof −
from *f1*
obtain *n1* **where** *fl1*: *field-lookup (typ-uinfo-t TYPE('a)) f1 0 = Some (typ-uinfo-t (TYPE('c::mem-type)), n1)*
apply (*simp add*: *typ-uinfo-t-def*)
by (*smt* (*verit*) *field-lookup-export-uinfo-Some field-names-SomeD2 field-names-u-field-names-export-uinfo-c wf-desc*)
from *f2*

obtain *n2* **where** *fl2*: *field-lookup* (*typ-uinfo-t TYPE*('*b*)) *f2 0* = *Some* (*typ-uinfo-t* (*TYPE*('*d*::*mem-type*)), *n2*)
apply (*simp add*: *typ-uinfo-t-def*)
by (*smt* (*verit*) *field-lookup-export-uinfo-Some field-names-SomeD2 field-names-u-field-names-export-uinfo-c wf-desc*)

from *field-tag-sub'* [*OF fl1*, *of p*] **have** {&(*p*→*f1*)..+*size-of TYPE*('*c*)} ⊆ *ptr-span p*
by (*simp add*: *size-of-tag*)
moreover
from *field-tag-sub'* [*OF fl2*, *of q*] **have** {&(*q*→*f2*)..+*size-of TYPE*('*d*)} ⊆ *ptr-span q*
by (*simp add*: *size-of-tag*)
ultimately show *?thesis*
using *disj*
by *auto*
qed

lemma *disj-ptr-span-field-disj-ptr-span'*:
assumes *disj*: *ptr-span* (*p*::'*a*::*mem-type ptr*) ∩ *ptr-span* (*q*::'*b*::*mem-type ptr*) = {}
assumes *f1*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f1 0* = *Some* (*s*, *n*)
assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*c*::*mem-type*))
assumes *f2*: *field-lookup* (*typ-info-t* (*TYPE*('*b*))) *f2 0* = *Some* (*t*, *m*)
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*d*::*mem-type*))
shows {&(*p*→*f1*)..+*size-of TYPE*('*c*)} ∩ {&(*q*→*f2*)..+*size-of TYPE*('*d*)} = {}
proof −
from *f1 match-s*
have *f1'*: *f1* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t* (*TYPE*('*c*::*mem-type*))))
by (*simp add*: *field-lookup-all-field-names*(*1*) *field-lookup-field-ti set-field-names-u-all-field-names-conv*)
from *f2 match-t*
have *f2'*: *f2* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*b*)) (*typ-uinfo-t* (*TYPE*('*d*::*mem-type*))))
by (*simp add*: *field-lookup-all-field-names*(*1*) *field-lookup-field-ti set-field-names-u-all-field-names-conv*)
from *disj-ptr-span-field-disj-ptr-span* [*OF disj f1' f2'*]
show *?thesis*
by *simp*
qed

lemma *disj-ptr-span-field-disj-ptr-span''*:
assumes *disj*: *ptr-span* (*p*::'*a*::*mem-type ptr*) ∩ *ptr-span* (*q*::'*b*::*mem-type ptr*) ≡ {}
assumes *f1*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f1 0* = *Some* (*s*, *n*)
assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*c*::*mem-type*))
assumes *f2*: *field-lookup* (*typ-info-t* (*TYPE*('*b*))) *f2 0* = *Some* (*t*, *m*)
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*d*::*mem-type*))
shows {&(*p*→*f1*)..+*size-of TYPE*('*c*)} ∩ {&(*q*→*f2*)..+*size-of TYPE*('*d*)} = {}
using *disj-ptr-span-field-disj-ptr-span'* [*OF - f1 match-s f2 match-t*]
by (*simp add*: *disj*)

lemma *field-lvalue-empty*:
 $\&(p\rightarrow[]) = ptr\text{-}val\ p$
 by (*simp add*: *field-lvalue-def field-offset-def field-offset-untyped-def*)

lemma *disj-ptr-span-field-disj-ptr-span-root1*:
 assumes *disj*: $ptr\text{-}span\ (p::'a::mem\text{-}type\ ptr) \cap ptr\text{-}span\ (q::'b::mem\text{-}type\ ptr) \equiv \{\}$
 assumes *f1*: $field\text{-}lookup\ (typ\text{-}info\text{-}t\ (TYPE('a)))\ f1\ 0 = Some\ (s,\ n)$
 assumes *match-s*: $export\text{-}uinfo\ s = typ\text{-}uinfo\text{-}t\ (TYPE('c::mem\text{-}type))$
 shows $\{\&(p\rightarrow f1)..+size\text{-}of\ TYPE('c)\} \cap ptr\text{-}span\ q \equiv \{\}$
  using *disj-ptr-span-field-disj-ptr-span''* [*OF disj f1 match-s, of* [] *typ-info-t TYPE('b) 0*,
    simplified typ-uinfo-t-def, *simplified*, *OF refl*]
 by (*simp add*: *field-lvalue-empty*)

lemma *disj-ptr-span-field-disj-ptr-span-root2*:
 assumes *disj*: $ptr\text{-}span\ (p::'a::mem\text{-}type\ ptr) \cap ptr\text{-}span\ (q::'b::mem\text{-}type\ ptr) \equiv \{\}$
 assumes *f2*: $field\text{-}lookup\ (typ\text{-}info\text{-}t\ (TYPE('b)))\ f2\ 0 = Some\ (t,\ m)$
 assumes *match-t*: $export\text{-}uinfo\ t = typ\text{-}uinfo\text{-}t\ (TYPE('d::mem\text{-}type))$
 shows $ptr\text{-}span\ p \cap \{\&(q\rightarrow f2)..+size\text{-}of\ TYPE('d)\} = \{\}$
 using *disj-ptr-span-field-disj-ptr-span''* [*OF disj - - f2 match-t, of* [] *typ-info-t TYPE('a) 0*,
    simplified typ-uinfo-t-def, *simplified*, *OF refl*]
 by (*simp add*: *field-lvalue-empty*)

lemma *root-ptr-valid-disj-field-lvalue-conv*:
 assumes *valid-p*: $root\text{-}ptr\text{-}valid\ d\ (p::'a::mem\text{-}type\ ptr)$
 assumes *valid-q*: $root\text{-}ptr\text{-}valid\ d\ (q::'b::mem\text{-}type\ ptr)$
 assumes *neq*: $typ\text{-}uinfo\text{-}t\ TYPE('a) \neq typ\text{-}uinfo\text{-}t\ TYPE('b)$
 assumes *f1*: $f1 \in set\ (all\text{-}field\text{-}names\ (typ\text{-}uinfo\text{-}t\ TYPE('a)))$
 assumes *f2*: $f2 \in set\ (all\text{-}field\text{-}names\ (typ\text{-}uinfo\text{-}t\ TYPE('b)))$
 shows $\&(p\rightarrow f1) = \&(q\rightarrow f2) = False$
 using *root-ptr-valid-type-neq-disjoint* [*OF valid-p valid-q neq*] *f1 f2*
 by (*simp add*: *disj-ptr-span-field-neq''*)

lemma *root-ptr-valid-disj-field-lvalue-conv'*:
 assumes *valid-p*: $root\text{-}ptr\text{-}valid\ d\ (p::'a::mem\text{-}type\ ptr)$
 assumes *valid-q*: $root\text{-}ptr\text{-}valid\ d\ (q::'b::mem\text{-}type\ ptr)$
 assumes *neq*: $typ\text{-}uinfo\text{-}t\ TYPE('a) \neq typ\text{-}uinfo\text{-}t\ TYPE('b)$
 assumes *f1*: $f1 \in set\ (all\text{-}field\text{-}names\ (typ\text{-}info\text{-}t\ TYPE('a)))$
 assumes *f2*: $f2 \in set\ (all\text{-}field\text{-}names\ (typ\text{-}info\text{-}t\ TYPE('b)))$
 shows $\&(p\rightarrow f1) = \&(q\rightarrow f2) = False$
proof −
 have $set\ (all\text{-}field\text{-}names\ (typ\text{-}info\text{-}t\ TYPE('a))) = set\ (all\text{-}field\text{-}names\ (typ\text{-}uinfo\text{-}t\ TYPE('a)))$
  by (*simp add*: *all-field-names-typ-uinfo-t-conv*)
 moreover
 have $set\ (all\text{-}field\text{-}names\ (typ\text{-}info\text{-}t\ TYPE('b))) = set\ (all\text{-}field\text{-}names\ (typ\text{-}uinfo\text{-}t$

TYPE('*b*)))
by (*simp add*: *all-field-names-typ-uinfo-t-conv*)
ultimately show *?thesis* **using** *root-ptr-valid-disj-field-lvalue-conv* [*OF valid-p valid-q neq*] *f1 f2*
by *simp*
qed

lemma *intvl-non-zero-non-empty*: $0 < n \Longrightarrow \{p..+n\} \neq \{\}$
by (*metis empty-iff first-in-intvl less-numeral-extra*(*3*))

lemma *disj-inter-swap*: $A \cap B = \{\} \Longrightarrow B \cap A = \{\}$
by *blast*

lemma *h-val-clift-field*:
clift hp (*Ptr* &(*p*→*f*)) = *Some v* ⟹ *v* = *h-val* (*hrs-mem hp*) (*Ptr* &(*p*→*f*))
by (*simp add*: *h-val-clift'*)

lemma *valid-root-footprints-no-overlap*:
assumes *valid-root-footprint d* (*ptr-val* (*p*::'*a*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
assumes *valid-root-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*b*))
assumes *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
shows *ptr-span p* ∩ *ptr-span q* = {}
by (*metis antisym-conv assms*(*1*) *assms*(*2*) *assms*(*3*) *disj-inter-swap*
export-uinfo-size size-of-def typ-uinfo-t-def valid-root-footprint-overlap-sub-typ valid-root-footprint-valid-footprint)

lemma *valid-root-footprints-overlap*:
assumes *valid-root-footprint d* (*ptr-val* (*p*::'*a*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
assumes *valid-root-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*b*))
assumes *ptr-span p* ∩ *ptr-span q* ≠ {}
shows (*typ-uinfo-t TYPE*('*a*)) = (*typ-uinfo-t TYPE*('*b*))
by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *valid-root-footprints-no-overlap*)

lemma *field-lookup-same-type-empty*:
field-lookup t f n = *Some* (*s*, *m*) ⟹ *s* = *t* ⟷ *f* = []
field-lookup-struct st f n = *Some* (*s*, *m*) ⟹ *f* ≠ []
field-lookup-list ts f n = *Some* (*s*, *m*) ⟹ *f* ≠ []
field-lookup-tuple td f n = *Some* (*s*, *m*) ⟹ *f* ≠ []
apply (*induct t* **and** *st* **and** *ts* **and** *td arbitrary*: *f n s m* **and** *f n s m* **and** *f n s m* **and** *f n s m*)
apply (*auto split*: *if-split-asm dest*!: *td-set-struct-field-lookup-structD td-set-struct-size-lte*)
done

lemma *root-ptr-valid-root-only*:
assumes *valid*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *non-empty*: *f* ≠ []
assumes *f*: *f* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))

shows *root-ptr-valid d* (*PTR*('*b*::*mem-type*) &(*p*→*f*)) = *False*
proof −
{
assume *root-ptr-valid d* (*PTR*('*b*::*mem-type*) &(*p*→*f*))
hence *valid-root*: *valid-root-footprint d* (*ptr-val* (*PTR*('*b*::*mem-type*) &(*p*→*f*)))
(*typ-uinfo-t TYPE*('*b*))
by (*simp add*: *root-ptr-valid-def*)

have *wf*: *wf-desc* (*typ-info-t TYPE*('*a*)) **by** *simp*
from *valid* **have** *valid-p*: *valid-root-footprint d* (*ptr-val p*) (*typ-uinfo-t TYPE*('*a*))
by (*simp add*: *root-ptr-valid-def*)

from *f* **have** *f* ∈ *set* (*field-names* (*typ-info-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))
by (*simp add*: *field-names-u-field-names-export-uinfo-conv*(*1*) *typ-uinfo-t-def*)

from *field-names-SomeD2* [*OF this wf*] **obtain** *s n* **where**
fl: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*) **and** *s*: *export-uinfo*
s = *typ-uinfo-t TYPE*('*b*)
by *blast*

from *field-lookup-export-uinfo-Some* [*OF fl*] *s*
have *fl'*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*),
n) **by** (*simp add*: *typ-uinfo-t-def*)

from *field-lookup-same-type-empty*(*1*) [*OF this*] *non-empty*
have *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*) **by** *simp*
from *valid-root-footprints-no-overlap* [*OF valid-p valid-root this*]
have *disjnt* (*ptr-span p*) (*ptr-span* (*PTR*('*b*) &(*p*→*f*))) **by** (*simp add*: *disjnt-def*)
moreover
from *field-tag-sub'* [*OF fl'*] **have** {&(*p*→*f*)..+*size-td* (*typ-uinfo-t TYPE*('*b*))}
⊆ *ptr-span p* **.**
ultimately
have *False*
by (*metis Int-absorb Int-absorb1 disj-ptr-span-ptr-neq disjnt-def ptr-val.ptr-val-def*
size-of-tag)
} **thus** *?thesis* **by** *blast*
qed

lemma *root-ptr-valid-root-only'*:
assumes *valid*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *non-empty*: *f* ≠ []
assumes *f*: *f* ∈ *set* (*field-names* (*typ-info-t TYPE*('*a*)) (*export-uinfo* (*typ-info-t*
TYPE('*b*))))
shows *root-ptr-valid d* (*PTR*('*b*::*mem-type*) &(*p*→*f*)) = *False*
proof −
have *set* (*field-names* (*typ-info-t TYPE*('*a*)) (*export-uinfo* (*typ-info-t TYPE*('*b*))))
=

set (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))
by (*simp add*: *fold-typ-uinfo-t set-field-names-all-field-names-conv set-field-names-u-all-field-names-conv*)
with *root-ptr-valid-root-only* [*OF valid non-empty*] *f*
show *?thesis*
by *simp*
qed

lemma *disj-subset*:
assumes $A \cap B = \{\}$
assumes $A' \subseteq A$
assumes $B' \subseteq B$
shows $A' \cap B' = \{\}$
using *assms*
by *blast*

lemma *disj-intvl-subset*:
assumes *disj*: $\{p..+n1\} \cap \{q..+m1\} = \{\}$
assumes *le1*: *n2* + *off1* ≤ *n1*
assumes *le2*: *m2* + *off2* ≤ *m1*
shows $\{p + of\text{-}nat\ off1..+n2\} \cap \{q + of\text{-}nat\ off2..+m2\} = \{\}$
apply (*rule disj-subset* [*OF disj*])
apply (*rule intvl-le* [*OF le1*])
apply (*rule intvl-le* [*OF le2*])
done

lemma *disj-intvl-field'*:
assumes *disj*: $\{ptr\text{-}val\ p..+n1\} \cap \{ptr\text{-}val\ q..+m1\} = \{\}$
assumes *f*: $\&(p \rightarrow f) = ptr\text{-}val\ p + (of\text{-}nat\ off1)$
assumes *g*: $\&(q \rightarrow g) = ptr\text{-}val\ q + (of\text{-}nat\ off2)$
assumes *le1*: *n2* + *off1* ≤ *n1*
assumes *le2*: *m2* + *off2* ≤ *m1*
shows $\{\&(p \rightarrow f)..+n2\} \cap \{\&(q \rightarrow g)..+m2\} = \{\}$
apply (*simp add*: *f g*)
apply (*rule disj-intvl-subset* [*OF disj le1 le2*])
done

lemma *disj-intvl-field*:
assumes *disj*: $\{ptr\text{-}val\ (p::'a::mem\text{-}type\ ptr)..+n1\} \cap \{ptr\text{-}val\ (q::'b::mem\text{-}type\ ptr)..+m1\} = \{\}$
assumes *le1*: *n2* + (*field-offset TYPE*('*a*) *f*) ≤ *n1*
assumes *le2*: *m2* + (*field-offset TYPE*('*b*) *g*) ≤ *m1*
shows $\{\&(p \rightarrow f)..+n2\} \cap \{\&(q \rightarrow g)..+m2\} = \{\}$
apply (*simp add*: *field-lvalue-def*)
apply (*rule disj-intvl-subset* [*OF disj le1 le2*])
done

lemma *intvl-fields-disj1*:

assumes *f*: &(*p*→*f*) = *ptr-val p* + *off1*
assumes *g*: &(*p*→*g*) = *ptr-val p* + *off2*
assumes *n-sz*: *unat off1* + *n* ≤ *addr-card*
assumes *m-sz*: *unat off2* + *m* ≤ *addr-card*
assumes *le*: *unat off1* ≤ *unat off2*
assumes *disj*: *unat off1* + *n* ≤ *unat off2*
shows{&(*p*→*f*)..+*n*} ∩ {&(*p*→*g*)..+*m*} = {}
proof −
 {
  fix *k k'*
  assume *k-n*: *k* < *n*
  assume *k'-m*: *k'* < *m*
  assume *eq*: *off1* + *word-of-nat k* = *off2* + *word-of-nat k'*
  have *False*
  proof −
   from *eq n-sz m-sz k-n k'-m* **have** *unat off1* + *k* = *unat off2* + *k'*
    by (*metis* (*no-types*, *opaque-lifting*) *word-bits-def add.right-neutral*
        len-of-addr-card less-le-trans nat-add-left-cancel-less of-nat-add unat-0
unat-of-nat-eq word-arith-nat-add)
   with *disj k-n* **show** *False* **by** *arith*
  qed
 } **note** *lem* = *this*
 show *?thesis*

  apply (*simp add*: *f g*)
  apply (*clarsimp simp add*: *intvl-def*)
  using *lem*
  by *auto*
qed

lemma *intvl-fields-disj*:
assumes *f*: &(*p*→*f*) = *ptr-val p* + *off1*
assumes *g*: &(*p*→*g*) = *ptr-val p* + *off2*
assumes *n-sz*: *unat off1* + *n* ≤ *addr-card*
assumes *m-sz*: *unat off2* + *m* ≤ *addr-card*
assumes *disj*: *if unat off1* ≤ *unat off2 then unat off1* + *n* ≤ *unat off2 else unat*
off2 + *m* ≤ *unat off1*
shows{&(*p*→*f*)..+*n*} ∩ {&(*p*→*g*)..+*m*} = {}
proof (*cases unat off1* ≤ *unat off2*)
 case *True*
 show *?thesis*
  apply (*rule intvl-fields-disj1* [*OF f g n-sz m-sz*])
  using *disj True*
  by *simp-all*
next
 case *False*
 have {&(*p*→*g*)..+*m*} ∩ {&(*p*→*f*)..+*n*} = {}
  apply (*rule intvl-fields-disj1* [*OF g f m-sz n-sz*])
  using *disj False*

by *simp-all*
thus *?thesis* **by** *blast*
qed

lemma *intvl-fields-disj-calculation*:
assumes *f*: *&(p→f) = ptr-val p + off1*
assumes *g*: *&(p→g) = ptr-val p + off2*
assumes *uoff1*: *unat off1 = uoff1*
assumes *uoff2*: *unat off2 = uoff2*
assumes *n-sz*: *uoff1 + n ≤ addr-card*
assumes *m-sz*: *uoff2 + m ≤ addr-card*
assumes *disj*: *if uoff1 ≤ uoff2 then uoff1 + n ≤ uoff2 else uoff2 + m ≤ uoff1*
shows *{&(p→f)..+n} ∩ {&(p→g)..+m} = {}*
using *intvl-fields-disj [OF f g] uoff1 uoff2 n-sz m-sz disj* **by** *simp*

lemma *disjoint-field-lvalue-propagation-right*:
fixes *q*::*'a*::*mem-type ptr*
assumes *disj*: *A ∩ {ptr-val q..+m} = {}*
assumes *fl*: *field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, k)*
assumes *m*: *m = size-of TYPE('a)*
assumes *n*: *n = size-td t*
shows *A ∩ {&(q→f)..+n} = {}*
using *disj fl m n*
by (*meson disjoint-iff field-tag-sub subsetD*)

lemma *disjoint-field-lvalue-propagation-left*:
fixes *q*::*'a*::*mem-type ptr*
assumes *disj*: *{ptr-val q..+m} ∩ A = {}*
assumes *fl*: *field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, k)*
assumes *m*: *m = size-of TYPE('a)*
assumes *n*: *n = size-td t*
shows *{&(q→f)..+n} ∩ A = {}*
using *disj fl m n*
by (*meson disjoint-iff field-tag-sub subsetD*)

lemma *disjoint-field-lvalue-propagation-both*:
fixes *q*::*'a*::*mem-type ptr*
fixes *p*::*'b*::*mem-type ptr*
assumes *disj*: *{ptr-val q..+m1} ∩ {ptr-val p..+m2} = {}*
assumes *fl1*: *field-lookup (typ-info-t TYPE('a)) f1 0 = Some (t1, k1)*
assumes *fl2*: *field-lookup (typ-info-t TYPE('b)) f2 0 = Some (t2, k2)*
assumes *m1*: *m1 = size-of TYPE('a)*
assumes *n1*: *n1 = size-td t1*
assumes *m2*: *m2 = size-of TYPE('b)*
assumes *n2*: *n2 = size-td t2*
shows *{&(q→f1)..+n1} ∩ {&(p→f2)..+n2} = {}*
using *disj fl1 fl2 m1 n1 m2 n2*

by (*meson disjoint-iff field-tag-sub subsetD*)

lemmas *disjoint-field-lvalue-propagation* =
disjoint-field-lvalue-propagation-left
disjoint-field-lvalue-propagation-right
disjoint-field-lvalue-propagation-both

lemma *disjoint-field-lvalue-neq*:
fixes *q*::*'a*::*mem-type ptr*
fixes *p*::*'b*::*mem-type ptr*
assumes *disj*: {*ptr-val q..+m1*} ∩ {*ptr-val p..+m2*} = {}
assumes *fl1*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f1 0* = *Some* (*t1*, *k1*)
assumes *fl2*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f2 0* = *Some* (*t2*, *k2*)
assumes *m1*: *m1* = *size-of TYPE*(*'a*)
assumes *n1*: *n1* = *size-td t1*
assumes *m2*: *m2* = *size-of TYPE*(*'b*)
assumes *n2*: *n2* = *size-td t2*
shows &(*q*→*f1*) = &(*p*→*f2*) = *False*
using *disjoint-field-lvalue-propagation-both* [*OF disj fl1 fl2 m1 n1 m2 n2*] *n1 n2*
by (*metis field-lookup-wf-size-desc-gt intvl-start-inter local.fl1 local.fl2 wf-size-desc*)

lemma *overlap-field-disj-contradiction*:
fixes *q*::*'a*::*mem-type ptr*
assumes *overlap*: *A* ∩ {&(*q*→*f*)..+*n*} ≠ {}
assumes *disj*: *A* ∩ {*ptr-val q..+m*} = {}
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *k*)
assumes *m*: *m* = *size-of TYPE*(*'a*)
assumes *n*: *n* = *size-td t*
shows *False*
using *overlap disj fl m n*
by (*meson disjoint-iff field-tag-sub subsetD*)

thm *root-ptr-valid-non-prefix-disj*

lemma *overlap-field-prefix-left*:
fixes *p*::*'a*::*mem-type ptr*
assumes *overlap*: {&(*p*→*f*) ..+*n1*} ∩ {&(*p*→*g*) ..+*n2*} ≠ {}
assumes *less*: *length f* < *length g*
assumes *f*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t1*, *k1*)
assumes *g*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *g 0* = *Some* (*t2*, *k2*)
assumes *n1*: *n1* = *size-td t1*
assumes *n2*: *n2* = *size-td t2*
shows *prefix f g*
proof (*cases prefix f g*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *less* **have** *not-prefix*: ¬ *prefix g f*

```
    by (simp add: More-Sublist.not-prefix-longer)
  have wf:wf-desc (typ-info-t TYPE('a)) by simp
  have le-addr-card: size-td (typ-info-t TYPE('a)) < addr-card
    by (metis max-size size-of-def)
  from field-lookup-offset-size' [OF f] have t1-bound: size-td t1 + k1 ≤ size-td
(typ-info-t TYPE('a)).
  with le-addr-card have t1-bound': size-td t1 + k1 < addr-card
    by simp
  from field-lookup-offset-size' [OF g] have t2-bound: size-td t2 + k2 ≤ size-td
(typ-info-t TYPE('a)).
  with le-addr-card have t2-bound': size-td t2 + k2 < addr-card
    by simp

  from field-lookup-non-prefix-disj [OF wf f g not-prefix False] overlap n1 n2
  have False
    using f g
    apply (simp add: field-lvalue-def intvl-def)
    apply (safe; clarsimp simp add: unat-of-nat-eq)
    subgoal premises prems for k ka
    proof −
      from prems  have k1 + k = k2 + ka
      by (smt (verit, ccfv-threshold) Abs-fnat-hom-add add.commute add-mono-thms-linordered-field(2)

            of-nat-lt-size-of order-less-le-trans size-of-def t1-bound t2-bound)
      with prems show False by simp
    qed
    subgoal premises prems for k ka
    proof −
      from prems  have k1 + k = k2 + ka
      by (smt (verit, ccfv-threshold) Abs-fnat-hom-add add.commute add-mono-thms-linordered-field(2)

            of-nat-lt-size-of order-less-le-trans size-of-def t1-bound t2-bound)
      with prems show False by simp
    qed
    done
  thus ?thesis by simp
qed

lemma overlap-field-prefix-right:
  fixes p::'a::mem-type ptr
  assumes overlap: {&(p→g) ..+n2} ∩ {&(p→f) ..+n1} ≠ {}
  assumes less: length f < length g
  assumes f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t1, k1)
  assumes g: field-lookup (typ-info-t TYPE('a)) g 0 = Some (t2, k2)
  assumes n1: n1 = size-td t1
  assumes n2: n2 = size-td t2
  shows prefix f g
  using overlap-field-prefix-left
  using f g less n1 n2 overlap by blast
```

lemma *field-lvalue-eq-non-prefix*:
fixes *p*::*'a*::*mem-type ptr*
assumes *fld-eq*: &(*p*→*f*) = &(*p*→*g*)
assumes *f*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t1*, *k1*)
assumes *g*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *g 0* = *Some* (*t2*, *k2*)
assumes *f-g*: ¬ *prefix f g*
assumes *g-f*: ¬ *prefix g f*
shows *f* = *g*
proof −
have *wf*:*wf-desc* (*typ-info-t TYPE*(*'a*)) **by** *simp*
show *?thesis* **using** *field-lookup-non-prefix-disj* [*OF wf f g g-f f-g*] *fld-eq f g*
apply (*simp add*: *field-lvalue-def*)
by (*metis field-lookup-wf-size-desc-pres*(*1*) *field-lvalue-same-root-conv fld-eq*
less-add-same-cancel1 nat-neq-iff wf-size-desc wf-size-desc-gt(*1*))
qed

lemma *field-lvalue-eq-non-prefix-conv*:
fixes *p*::*'a*::*mem-type ptr*
assumes *f*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t1*, *k1*)
assumes *g*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *g 0* = *Some* (*t2*, *k2*)
assumes *f-g*: ¬ *prefix f g*
assumes *g-f*: ¬ *prefix g f*
shows &(*p*→*f*) = &(*p*→*g*) ⟷ *f* = *g*
apply *standard*
apply (*drule field-lvalue-eq-non-prefix* [*OF - f g f-g g-f*])
apply *simp*
apply *simp*
done

thm *cvalid-field-pres''*

lemma *hrs-mem-update-comp*: *hrs-mem-update f o hrs-mem-update g* = *hrs-mem-update* (*f o g*)
apply (*rule ext*)
apply (*auto simp add*: *hrs-mem-update-def*)
done

lemma *hrs-mem-update-comp'*: *hrs-mem-update f* (*hrs-mem-update g s*) = *hrs-mem-update* (*f o g*) *s*
using *hrs-mem-update-comp*
by (*metis comp-eq-dest-lhs*)

named-theorems *plift-defs* **and**
plift-cond-simps **and**
plift-iff **and**
plift-eqI **and**
ptr-valid-h-t-valid **and**

plift-heap-update **and**
ptr-valid-intros **and**
h-val-field-plift **and**
the-plift-h-val-conv

lemma *map-option-the-conv*: $f\ (the\ (map\text{-}option\ g\ x)) = f\ (the\ x) \longleftrightarrow (\forall v.\ x = Some\ v \longrightarrow f\ (g\ v) = f\ v)$
by (*metis handy-if-lemma option.exhaust-sel option.map-sel option.simps*(8))

definition *the-default*:: $'a \Rightarrow 'a\ option \Rightarrow 'a$ **where**
the-default x v = (*case v of Some z* ⇒ *z* | *None* ⇒ *x*)

lemma *the-default-simps*[*simp*]:
the-default x None = *x*
the-default x (*Some z*) = *z*
by (*auto simp add*: *the-default-def*)

locale *wf-ptr-valid* =
fixes *ptr-valid*::*heap-typ-desc* ⇒ 'a::*mem-type ptr-guard*
assumes *ptr-valid-h-t-valid*[*ptr-valid-h-t-valid*]: $\bigwedge d.\ ptr\text{-}valid\ d\ p \Longrightarrow d \models_t p$
begin

definition *plift*:: *heap-raw-state* ⇒ 'a *typ-heap*
where [*plift-defs*]:*plift h* = *lift-t* (*ptr-valid* (*hrs-htd h*)) *h*

lemma *plift-Some-iff* [*plift-iff*]:
shows $(\exists v.\ plift\ h\ p = Some\ v) \longleftrightarrow ptr\text{-}valid\ (hrs\text{-}htd\ h)\ p$
using *ptr-valid-h-t-valid*
apply (*cases h*)
apply (*auto simp add*: *plift-def TypHeap.lift-t-if hrs-htd-def h-t-valid-def*)
done

lemma *plift-None-iff* [*plift-iff*]:
shows $(plift\ h\ p = None) \longleftrightarrow \neg\ ptr\text{-}valid\ (hrs\text{-}htd\ h)\ p$
using *ptr-valid-h-t-valid*
apply (*cases h*)
apply (*auto simp add*: *plift-def TypHeap.lift-t-if hrs-htd-def h-t-valid-def*)
done

lemma *plift-None*: $\neg\ ptr\text{-}valid\ (hrs\text{-}htd\ h)\ p \Longrightarrow plift\ h\ p = None$
by (*simp add*: *plift-None-iff*)

lemma *ptr-valid-c-guard*: $ptr\text{-}valid\ d\ p \Longrightarrow c\text{-}guard\ p$
by (*auto intro*: *ptr-valid-h-t-valid h-t-valid-c-guard*)

lemma *plift-Some-ptr-valid*[*plift-cond-simps*]: $plift\ h\ p = Some\ v \Longrightarrow ptr\text{-}valid\ (hrs\text{-}htd\ h)\ p$
using *plift-Some-iff* **by** *blast*

lemma *plift-Some-h-t-valid*[*plift-cond-simps*]: *plift h p = Some v ⟹ hrs-htd h ⊨$_t$ p*
using *plift-Some-ptr-valid ptr-valid-h-t-valid*
by *blast*

lemma *plift-Some-c-guard*[*plift-cond-simps*]: *plift h p = Some v ⟹ c-guard p*
using *plift-Some-h-t-valid h-t-valid-c-guard*
by *blast*

lemma *ptr-valid-heap-update-conv*[*simp*]:
ptr-valid (hrs-htd ((hrs-mem-update (heap-update p v) s))) q ⟷ ptr-valid (hrs-htd s) q
by (*cases s*) (*auto simp add*: *hrs-htd-def hrs-mem-update-def*)

lemma *ptr-valid-heap-update-padding-conv*[*simp*]:
ptr-valid (hrs-htd ((hrs-mem-update (heap-update-padding p v bs) s))) q ⟷ ptr-valid (hrs-htd s) q
by (*cases s*) (*auto simp add*: *hrs-htd-def hrs-mem-update-def*)

lemma *ptr-valid-heap-update-pres*: *ptr-valid (hrs-htd s) q ⟹*
ptr-valid (hrs-htd ((hrs-mem-update (heap-update p v)s))) q
by (*simp add*: *ptr-valid-heap-update-conv*)

lemma *ptr-valid-heap-update-padding-pres*: *ptr-valid (hrs-htd s) q ⟹*
ptr-valid (hrs-htd ((hrs-mem-update (heap-update-padding p v bs)s))) q
by (*simp add*: *ptr-valid-heap-update-conv*)

lemma *plift-Some-h-val*:
assumes *Some*: *plift h p = Some v*
shows *h-val (hrs-mem h) p = v*
using *plift-Some-c-guard* [*OF Some*] *plift-Some-ptr-valid* [*OF Some*] *Some h-val-clift′*
apply (*simp add*: *plift-def*)
by (*smt* (*verit*, *del-insts*) *c-type-class.lift-def hrs-mem-def prod.collapse typ-simps*(*8*))

lemma *ptr-valid-plift-Some-hval*:
assumes *ptr-valid (hrs-htd h) p*
shows *plift h p = Some (h-val (hrs-mem h) p)*
using *assms plift-Some-h-val plift-Some-iff* **by** *blast*

lemma *the-plift-hval-conv*: *ptr-valid (hrs-htd h) p ⟹ the (plift h p) = h-val (hrs-mem h) p*
by (*metis option.collapse plift-None-iff wf-ptr-valid.plift-Some-h-val wf-ptr-valid-axioms*)

lemma *the-default-plift-hval-conv*[*the-plift-h-val-conv*]:
ptr-valid (hrs-htd h) p ⟹ the-default c-type-class.zero (plift h p) = h-val (hrs-mem

h) *p*
by (*simp add*: *ptr-valid-plift-Some-hval*)

lemma *valid-h-val-the-plift-conv*:
assumes *valid*: *ptr-valid* (*hrs-htd h*) *p*
shows *h-val* (*hrs-mem h*) *p* = *the* (*plift h p*)
using *valid plift-Some-h-val plift-iff*
by (*cases plift h p*) *auto*

lemma *valid-h-val-the-default-plift-conv*:
assumes *valid*: *ptr-valid* (*hrs-htd h*) *p*
shows *h-val* (*hrs-mem h*) *p* = *the-default c-type-class.zero* (*plift h p*)
using *valid plift-Some-h-val plift-iff*
by (*cases plift h p*) *auto*

lemma *plift-Some-clift-Some*:
assumes *Some*: *plift h p* = *Some v*
shows *clift h p* = *Some v*
using *plift-Some-h-t-valid* [*OF Some*]
h-t-valid-clift-Some-iff [*of h p*]
h-val-clift' [*of h p*] *plift-Some-h-val* [*OF Some*]
by *auto*

lemma *clift-None-plift-None*:
assumes *clift h p* = *None*
shows *plift h p* = *None*
using *assms*
by (*metis plift-None-iff plift-Some-iff wf-ptr-valid.plift-Some-clift-Some wf-ptr-valid-axioms*)

lemma *plift-clift-conv*:
assumes *ptr-valid* (*hrs-htd h*) *p*
shows *plift h p* = *clift h p*
by (*metis assms plift-Some-clift-Some plift-Some-iff*)

lemma *the-plift-hval-fun-upd-eqI* [*plift-eqI*]:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *ptr-valid* (*hrs-htd h*) *p*
shows (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update p v*) *h*) *q*)) =
((λ*p*. *the-default c-type-class.zero* (*plift h p*))(*p* := *v*)) *q*
by (*smt* (*verit*, *best*) *assms clift-heap-update fun-upd-apply hrs-htd-mem-update*
local.ptr-valid-h-t-valid plift-clift-conv the-default-simps(*2*) *wf-ptr-valid.plift-None-iff*
wf-ptr-valid-axioms)

lemma *the-plift-hval-fun-upd-padding-eqI* [*plift-eqI*]:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *ptr-valid* (*hrs-htd h*) *p*

assumes *length bs = size-of TYPE('a)*
shows *(the-default c-type-class.zero (plift*
(hrs-mem-update (heap-update-padding p v bs) h) q)) =
((λp. the-default c-type-class.zero (plift h p))(p := v)) q
by *(smt (verit, ccfv-SIG) CTypes.mem-type-simps(2) assms(1) assms(2) fun-upd-apply h-val-def*
h-val-heap-update-padding heap-list-update-disjoint-same heap-update-padding-def

hrs-htd-mem-update hrs-mem-update ptr-valid-same-type-cases the-default-plift-hval-conv

wf-ptr-valid.plift-None-iff wf-ptr-valid.ptr-valid-h-t-valid wf-ptr-valid-axioms)

lemma *field-the-plift-hval-fun-upd-eqI [plift-eqI]:*
fixes *p::'a::mem-type ptr*
fixes *q::'a::mem-type ptr*
assumes *ptr-valid (hrs-htd h) p ∧ f x = v*
shows *f (the-default c-type-class.zero (plift*
(hrs-mem-update (heap-update p x) h) q)) =
((λp. f (the-default c-type-class.zero (plift h p)))(p := v)) q
by *(simp add: assms(1) the-plift-hval-fun-upd-eqI)*

lemma *field-the-plift-hval-fun-upd-padding-eqI [plift-eqI]:*
fixes *p::'a::mem-type ptr*
fixes *q::'a::mem-type ptr*
assumes *ptr-valid (hrs-htd h) p ∧ f x = v*
assumes *length bs = size-of TYPE('a)*
shows *f (the-default c-type-class.zero (plift*
(hrs-mem-update (heap-update-padding p x bs) h) q)) =
((λp. f (the-default c-type-class.zero (plift h p)))(p := v)) q
by *(simp add: assms the-plift-hval-fun-upd-padding-eqI)*

lemma *the-plift-hval-eqI [plift-eqI]:*
fixes *p::'b::c-type ptr*
fixes *q::'a::mem-type ptr*
assumes *eq: ptr-valid (hrs-htd h) q ⟹ hrs-htd h ⊨ₜ q ⟹*
(h-val ((heap-update p v) (hrs-mem h)) q) = (h-val (hrs-mem h) q)
shows *(the-default c-type-class.zero (plift*
(hrs-mem-update (heap-update p v) h) q)) =
(the-default c-type-class.zero (plift h q))
proof *(cases (plift h q))*
case *None*
then show *?thesis* **using** *plift-None-iff*
by *(metis hrs-htd-mem-update)*
next
case *(Some x)*
from *plift-Some-ptr-valid [OF Some]*
have *ptr-valid: ptr-valid (hrs-htd h) q.*
from *plift-Some-h-t-valid [OF Some]*

have *h-t-valid*: *hrs-htd h* $\models_t$ *q*.
from *eq* [*OF ptr-valid h-t-valid*]
have *h-val-eq*: (*h-val* (*heap-update p v* (*hrs-mem h*)) *q*) = (*h-val* (*hrs-mem h*) *q*).

with *ptr-valid-heap-update-conv plift-Some-iff ptr-valid eq plift-Some-h-val*
show *?thesis*
by (*smt* (*verit*) *hrs-mem-update option.sel*)
qed

lemma *the-plift-hval-padding-eqI* [*plift-eqI*]:
fixes *p*::'*b*::*c-type ptr*
fixes *q*::'*a*::*mem-type ptr*
assumes *lbs*: *length bs* = *size-of TYPE*('*b*)
assumes *eq*: *ptr-valid* (*hrs-htd h*) *q* ⟹ *hrs-htd h* $\models_t$ *q* ⟹
(*h-val* ((*heap-update-padding p v bs*) (*hrs-mem h*)) *q*) = (*h-val* (*hrs-mem h*) *q*)

shows (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q*)) =
(*the-default c-type-class.zero* (*plift h q*))
proof (*cases* (*plift h q*))
case *None*
then show *?thesis* **using** *plift-None-iff*
by (*metis hrs-htd-mem-update*)
next
case (*Some x*)
from *plift-Some-ptr-valid* [*OF Some*]
have *ptr-valid*: *ptr-valid* (*hrs-htd h*) *q*.
from *plift-Some-h-t-valid* [*OF Some*]
have *h-t-valid*: *hrs-htd h* $\models_t$ *q*.
from *eq* [*OF ptr-valid h-t-valid*]
have *h-val-eq*: (*h-val* (*heap-update-padding p v bs* (*hrs-mem h*)) *q*) = (*h-val* (*hrs-mem h*) *q*).

with *ptr-valid-heap-update-padding-conv plift-Some-iff ptr-valid eq plift-Some-h-val*
show *?thesis*
by (*smt* (*verit*) *hrs-mem-update option.sel*)
qed

lemma *field-the-plift-hval-eqI* [*plift-eqI*]:
fixes *p*::'*b*::*c-type ptr*
fixes *q*::'*a*::*mem-type ptr*
assumes *eq*: *ptr-valid* (*hrs-htd h*) *q* ⟹ *hrs-htd h* $\models_t$ *q* ⟹
f (*h-val* ((*heap-update p v*) (*hrs-mem h*)) *q*) = *f* (*h-val* (*hrs-mem h*) *q*)
shows *f* (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update p v*) *h*) *q*)) =
f (*the-default c-type-class.zero* (*plift h q*))
proof (*cases* (*plift h q*))
case *None*
then show *?thesis* **using** *plift-None-iff*

by (*metis hrs-htd-mem-update*)
next
case (*Some x*)
from *plift-Some-ptr-valid* [*OF Some*]
have *ptr-valid*: *ptr-valid* (*hrs-htd h*) *q*.
from *plift-Some-h-t-valid* [*OF Some*]
have *h-t-valid*: *hrs-htd h* $\models_t$ *q*.
from *eq* [*OF ptr-valid h-t-valid*]
have *h-val-eq*: *f* (*h-val* (*heap-update p v* (*hrs-mem h*)) *q*) = *f* (*h-val* (*hrs-mem h*) *q*).

with *ptr-valid-heap-update-conv plift-Some-iff ptr-valid eq plift-Some-h-val*
show *?thesis*
by (*metis hrs-mem-f the-default-simps*(*2*))
qed

lemma *field-the-plift-hval-padding-eqI* [*plift-eqI*]:
fixes *p*::*'b*::*c-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *lbs*: *length bs* = *size-of TYPE*(*'b*)
assumes *eq*: *ptr-valid* (*hrs-htd h*) *q* ⟹ *hrs-htd h* $\models_t$ *q* ⟹
f (*h-val* ((*heap-update-padding p v bs*) (*hrs-mem h*)) *q*) = *f* (*h-val* (*hrs-mem h*) *q*)
shows *f* (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q*)) =
f (*the-default c-type-class.zero* (*plift h q*))
proof (*cases* (*plift h q*))
case *None*
then show *?thesis* **using** *plift-None-iff*
by (*metis hrs-htd-mem-update*)
next
case (*Some x*)
from *plift-Some-ptr-valid* [*OF Some*]
have *ptr-valid*: *ptr-valid* (*hrs-htd h*) *q*.
from *plift-Some-h-t-valid* [*OF Some*]
have *h-t-valid*: *hrs-htd h* $\models_t$ *q*.
from *eq* [*OF ptr-valid h-t-valid*]
have *h-val-eq*: *f* (*h-val* (*heap-update-padding p v bs* (*hrs-mem h*)) *q*) = *f* (*h-val* (*hrs-mem h*) *q*).

with *ptr-valid-heap-update-padding-conv plift-Some-iff ptr-valid eq plift-Some-h-val*
show *?thesis*
by (*metis hrs-mem-f the-default-simps*(*2*))
qed

lemma *plift-heap-update* [*plift-heap-update*]:
⟦ *ptr-valid* (*hrs-htd h*) *p* ⟧ ⟹
plift (*hrs-mem-update* (*heap-update p v*) *h*)
= (*plift h*)(*p* := *Some* (*v*::*'a*::*mem-type*))

using *plift-Some-clift-Some ptr-valid-h-t-valid*
by (*metis* (*no-types*, *opaque-lifting*) *h-t-valid-guard-subst hrs-htd-def hrs-htd-mem-update*

hrs-mem-def hrs-mem-update lift-t-heap-update plift-def prod.collapse)

lemma *plift-heap-update-padding* [*plift-heap-update*]:
⟦ *ptr-valid* (*hrs-htd h*) *p*; *length bs* = *size-of TYPE*('*a*) ⟧ ⟹
plift (*hrs-mem-update* (*heap-update-padding p v bs*) *h*)
= (*plift h*)(*p* := *Some* (*v*::'*a*::*mem-type*))
apply (*rule ext*)
apply (*clarsimp simp add*: *fun-upd-def*, *intro conjI*)
subgoal by (*simp add*: *h-val-heap-update-padding hrs-mem-update wf-ptr-valid.ptr-valid-plift-Some-hval wf-ptr-valid-axioms*)
using *plift-Some-clift-Some ptr-valid-h-t-valid*
by (*smt* (*verit*, *best*) *fun-upd-apply hrs-htd-mem-update the-default-plift-hval-conv*

the-plift-hval-fun-upd-padding-eqI wf-ptr-valid.plift-None-iff wf-ptr-valid.ptr-valid-plift-Some-hval wf-ptr-valid-axioms)

lemma *h-val-field-plift* [*h-val-field-plift*]:
fixes *pa* :: '*a* :: *mem-type ptr*
assumes *cl*: *plift hp pa* = *Some v*
and *fl*: *field-ti TYPE*('*a*) *f* = *Some t*
and *eu*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b* :: *mem-type*)
shows *h-val* (*hrs-mem hp*) (*Ptr* &(*pa*→*f*) :: '*b* :: *mem-type ptr*) = *from-bytes* (*access-ti*$_0$ *t v*)
using *cl fl eu*
using *h-val-field-clift' plift-Some-clift-Some* **by** *blast*

lemma *plift-disjoint-region*:
fixes *p*:: '*a* :: *mem-type ptr*
fixes *q*:: '*b* :: *mem-type ptr*
assumes *disj*: *ptr-span q* ∩ *ptr-span p* = {}
shows *plift* (*hrs-mem-update* (*heap-update q v*) *m*) *p* = *plift m p*
using *disj*
by (*metis Int-commute h-val-update-regions-disjoint hrs-htd-mem-update hrs-mem-heap-update option.collapse plift-None-iff the-plift-hval-conv*)

lemma *plift-disjoint-region-padding*:
fixes *p*:: '*a* :: *mem-type ptr*
fixes *q*:: '*b* :: *mem-type ptr*
assumes *disj*: *ptr-span q* ∩ *ptr-span p* = {}
assumes *lbs*: *length bs* = *size-of TYPE*('*b*)
shows *plift* (*hrs-mem-update* (*heap-update-padding q v bs*) *m*) *p* = *plift m p*
using *disj lbs*
by (*metis* (*no-types*, *lifting*) *CTypes.mem-type-simps*(2) *disj h-val-def heap-list-update-disjoint-same heap-update-padding-def hrs-htd-mem-update hrs-mem-update wf-ptr-valid.plift-None-iff wf-ptr-valid.ptr-valid-plift-Some-hval*

wf-ptr-valid-axioms)

lemma *map-option-the-plift-h-val-conv*:
  f (*the* (*map-option* g (*plift* h p))) = f (*the* (*plift* h p)) $\longleftrightarrow$ (*ptr-valid* (*hrs-htd* h) p $\longrightarrow$ f (g (*h-val* (*hrs-mem* h) p)) = f (*h-val* (*hrs-mem* h) p))
  apply (*subst map-option-the-conv*)
  apply *standard*
  using *ptr-valid-plift-Some-hval* **apply** *blast*
  using *plift-Some-h-val plift-Some-ptr-valid* **by** *blast*

lemma *invalid-plift-heap-update-skip*:
  assumes *invalid-p*: ¬ *ptr-valid* (*hrs-htd* h) p
  assumes *typed-p*: *hrs-htd* h $\models_t$ p
shows *plift* (*hrs-mem-update* (*heap-update* p v) h) q = *plift* h q
proof (*cases* p = q)
  case *True*
  with *invalid-p* **have** *invalid-q*: ¬ *ptr-valid* (*hrs-htd* h) q **by** *simp*
  from *invalid-q* **have** *plift* h q = *None*
    by (*simp add*: *plift-None*)
  moreover
  from *invalid-q* **have** *plift* (*hrs-mem-update* (*heap-update* p v) h) q = *None*
    by (*simp add*: *plift-None*)
  ultimately show *?thesis* **by** *simp*
next
  case *False*
  note *neq-p-q* = *this*
  show *?thesis*
  proof (*cases ptr-valid* (*hrs-htd* h) q)
    case *True*
    from *ptr-valid-h-t-valid* [*OF True*]
    have *hrs-htd* h $\models_t$ q **.**
    from *ptr-valid-same-type-cases* [*OF typed-p this*] *neq-p-q*
    have *ptr-span* p ∩ *ptr-span* q = {} **by** *blast*
    then show *?thesis*
      by (*simp add*: *plift-disjoint-region*)
  next
    case *False*
    from *False* **have** *plift* h q = *None*
      by (*simp add*: *plift-None*)
    moreover
    from *False* **have** *plift* (*hrs-mem-update* (*heap-update* p v) h) q = *None*
      by (*simp add*: *plift-None*)
    ultimately show *?thesis* **by** *simp*
  qed
qed

lemma *invalid-plift-heap-update-padding-skip*:
  assumes *invalid-p*: ¬ *ptr-valid* (*hrs-htd* h) p
  assumes *typed-p*: *hrs-htd* h $\models_t$ p

assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *plift* (*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q* = *plift h q*
proof (*cases p* = *q*)
case *True*
with *invalid-p* **have** *invalid-q*: ¬ *ptr-valid* (*hrs-htd h*) *q* **by** *simp*
from *invalid-q* **have** *plift h q* = *None*
by (*simp add*: *plift-None*)
moreover
from *invalid-q* **have** *plift* (*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q* = *None*
by (*simp add*: *plift-None*)
ultimately show *?thesis* **by** *simp*
next
case *False*
note *neq-p-q* = *this*
show *?thesis*
proof (*cases ptr-valid* (*hrs-htd h*) *q*)
case *True*
from *ptr-valid-h-t-valid* [*OF True*]
have *hrs-htd h* ⊨$_t$ *q* **.**
from *ptr-valid-same-type-cases* [*OF typed-p this*] *neq-p-q*
have *ptr-span p* ∩ *ptr-span q* = {} **by** *blast*
then show *?thesis*
using *lbs*
by (*simp add*: *plift-disjoint-region-padding*)
next
case *False*
from *False* **have** *plift h q* = *None*
by (*simp add*: *plift-None*)
moreover
from *False* **have** *plift* (*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q* = *None*
by (*simp add*: *plift-None*)
ultimately show *?thesis* **by** *simp*
qed
qed

end

global-interpretation *simple-lift*: *wf-ptr-valid root-ptr-valid* **rewrites** *simple-lift.plift* = *simple-lift*
apply (*unfold-locales*)
apply (*simp add*: *root-ptr-valid-h-t-valid*)
apply (*rule ext*, *rule ext*)
by (*metis root-ptr-valid-h-t-valid hrs-mem-def lift-t-if prod.collapse simple-lift-def*

wf-ptr-valid.intro wf-ptr-valid.plift-None-iff wf-ptr-valid.plift-def)

named-theorems *ptr-valid-stack-alloc* **and** *ptr-valid-stack-release* **and** *plift-stack-alloc* **and** *plift-stack-release*

locale *ptr-valid-stack-alloc* = *wf-ptr-valid* +
  assumes *alloc*[*ptr-valid-stack-alloc*]:
    $\bigwedge d\ d'\ p\ q.\ (p,\ d') \in \mathit{stack\text{-}alloc}\ \mathcal{S}\ \mathit{TYPE}('a{::}\mathit{mem\text{-}type})\ d \Longrightarrow$
      $\mathit{ptr\text{-}valid}\ d'\ q \longleftrightarrow (q = p) \vee \mathit{ptr\text{-}valid}\ d\ q$
begin

lemma *plift-stack-alloc-update*:
  assumes *alloc*: $(p,\ d) \in \mathit{stack\text{-}alloc}\ \mathcal{S}\ \mathit{TYPE}('a{::}\mathit{mem\text{-}type})\ (\mathit{hrs\text{-}htd}\ m)$
  shows $(\mathit{plift}\ (\mathit{hrs\text{-}htd\text{-}update}\ (\lambda\text{-}.\ d)\ m))(p{\mapsto}x) = (\lambda q.\ \mathit{if}\ p = q\ \mathit{then}\ \mathit{Some}\ x\ \mathit{else}\ \mathit{plift}\ m\ q)$
  apply (*rule ext*)
  apply *clarsimp*
  using *ptr-valid-stack-alloc* [*OF alloc*]
  by (*metis hrs-htd-update hrs-mem-htd-update option.collapse plift-None-iff wf-ptr-valid.valid-h-val-the-plift-conv wf-ptr-valid-axioms*)

lemma *plift-stack-alloc-sim*[*plift-stack-alloc*]:
  assumes *alloc*: $(p,\ d) \in \mathit{stack\text{-}alloc}\ \mathcal{S}\ \mathit{TYPE}('a{::}\mathit{mem\text{-}type})\ (\mathit{hrs\text{-}htd}\ m)$
  shows $(\mathit{plift}\ (\mathit{hrs\text{-}mem\text{-}update}\ (\mathit{heap\text{-}update}\ p\ x)\ (\mathit{hrs\text{-}htd\text{-}update}\ (\lambda\text{-}.\ d)\ m))) =$
      $(\lambda q.\ \mathit{if}\ p = q\ \mathit{then}\ \mathit{Some}\ x\ \mathit{else}\ \mathit{plift}\ m\ q)$
  using *plift-stack-alloc-update* [*OF alloc*]
  by (*simp add*: *hrs-htd-update plift-heap-update ptr-valid-stack-alloc* [*OF alloc*])

end

locale *ptr-valid-stack-release* = *wf-ptr-valid* +
  assumes *release*[*ptr-valid-stack-release*]:
   $\bigwedge d\ p\ q.\ \mathit{root\text{-}ptr\text{-}valid}\ d\ p \Longrightarrow \mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}('a) \neq \mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}(\mathit{stack\text{-}byte}) \Longrightarrow$
     $\mathit{ptr\text{-}valid}\ (\mathit{stack\text{-}release}\ p\ d)\ q \longleftrightarrow ((q \neq p) \wedge \mathit{ptr\text{-}valid}\ d\ q)$
begin

lemma *plift-stack-release*[*plift-stack-release*]:
  assumes *root-valid*: *root-ptr-valid* (*hrs-htd m*) ($p{::}'a{::}\mathit{mem\text{-}type}\ \mathit{ptr}$)
  assumes *p*: $\mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}('a) \neq \mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}(\mathit{stack\text{-}byte})$
  shows *plift* (*hrs-htd-update* (*stack-release p*) *m*) *q* =
      (*if p* = *q then None else plift m q*)
 using *ptr-valid-stack-release* [*OF root-valid p*]
  by (*metis* (*no-types*, *lifting*) *hrs-htd-update hrs-mem-htd-update option.collapse valid-h-val-the-plift-conv wf-ptr-valid.plift-None-iff wf-ptr-valid-axioms*)
end

locale *ptr-valid-stack* = *ptr-valid-stack-alloc* + *ptr-valid-stack-release*

global-interpretation *simple-lift*: *ptr-valid-stack root-ptr-valid*

apply (*unfold-locales*)
apply (*smt* (*verit*) *disjoint-iff ptr-force-type-same-cleared-region root-ptr-valid-def*

root-ptr-valid-h-t-valid s-footprintD s-footprintI2 size-of-def stack-alloc-cases
stack-alloc-preserves-root-ptr-valid stack-release-other
stack-release-stack-alloc-inverse typ-uinfo-size valid-root-footprint-def)
apply (*smt* (*verit*) *disjoint-iff in-ptr-span-itself intvlI intvl-inter root-ptr-valid-def*

root-ptr-valid-h-t-valid root-ptr-valid-neq-disjoint root-ptr-valid-same-type-cases

simple-heap-diff-types-impl-diff-ptrs size-of-def stack-release-other
stack-release-root-ptr-valid-footprint typ-uinfo-size valid-root-footprint-def)
done

lemma *valid-root-footprint-domain*:
assumes *valid-root-footprint d a t*
assumes $\bigwedge x.\ x \in \{a\ ..+\ size\text{-}td\ t\} \Longrightarrow d'\ x = d\ x$
shows *valid-root-footprint d' a t*
by (*metis assms*(*1*) *assms*(*2*) *intvlI valid-root-footprint-def*)

lemma *valid-root-footprint-cases*:
assumes *valid-a*: *valid-root-footprint d a T*
assumes *root-p*: *valid-root-footprint d b S*
shows $(a = b \wedge S = T) \vee \{b\ ..+\ size\text{-}td\ S\} \cap \{a\ ..+\ size\text{-}td\ T\} = \{\}$
proof (*cases* $\{b\ ..+\ size\text{-}td\ S\} \cap \{a\ ..+\ size\text{-}td\ T\} = \{\}$)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
note *overlap* = *this*

show *?thesis*
proof (*cases S* = *T*)
case *True*
from *valid-root-footprint-neq-disjoint* [*OF valid-a root-p*] *overlap*
have *a* = *b*
by (*metis inf-commute size-of-tag*)
then show *?thesis* **by** (*simp add*: *True*)
next
case *False*
with *valid-root-footprint-type-neq-disjoint* [*OF root-p valid-a False*] *overlap*
have *False*
by (*simp add*: *size-of-def*)
thus *?thesis*
by *simp*
qed
qed

lemma *valid-root-footprint-domain-cases*:
assumes *valid-a*: *valid-root-footprint d′ a t*
assumes *other-eq*: $\bigwedge x.\ x \notin \{a\ ..+\ size\text{-}td\ t\} \Longrightarrow d'\ x = d\ x$
assumes *root-p*: *valid-root-footprint d′ b s*
shows
$(a = b \wedge s = t \vee$
valid-root-footprint d b s)
proof (*cases* $\{b\ ..+\ size\text{-}td\ s\} \cap \{a\ ..+\ size\text{-}td\ t\} = \{\}$)
case *True*
with *other-eq* **have** $\bigwedge x.\ x \in \{b\ ..+\ size\text{-}td\ s\} \Longrightarrow\ d'\ x = d\ x$ **by** *auto*
with *valid-root-footprint-domain* [*OF root-p*]
have *valid-root-footprint d b s*
by *auto*
thus *?thesis* **by** *blast*
next
case *False*
note *overlap* = *this*

show *?thesis*
proof (*cases s* = *t*)
case *True*
from *valid-root-footprint-neq-disjoint* [*OF valid-a root-p*] *overlap*
have *a* = *b*
by (*metis inf-commute size-of-tag*)
then show *?thesis* **by** (*simp add*: *True*)
next
case *False*
with *valid-root-footprint-type-neq-disjoint* [*OF root-p valid-a False*] *overlap*
have *False*
by (*simp add*: *size-of-def*)
thus *?thesis*
by *simp*
qed
qed

lemma *h-t-valid-ptr-span-preservation*:
fixes *p*::′*a*::*mem-type ptr*
assumes *valid*: *d* $\models_t$ *p*
assumes *eq*: $\bigwedge a.\ a \in ptr\text{-}span\ p \Longrightarrow d'\ a = d\ a$
shows *d′* $\models_t$ *p*
using *valid eq*
by (*auto simp add*: *h-t-valid-def valid-footprint-def Let-def intvlI size-of-def*)

lemma
fixes *p*::′*a*::*mem-type ptr*
fixes *q*::′*a*::*mem-type ptr*
assumes *root-ptr-valid d p*
assumes *d* $\models_t$ *q*
shows *p* = *q* ∨ *ptr-span p* ∩ *ptr-span q* = {}

by (*meson assms*(*1*) *assms*(*2*) *root-ptr-valid-same-type-cases*)

lemma *stack-alloc-typing-other*:
fixes *q*::*'b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*(*'a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*(*'b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *other*: ¬ *TYPE*(*'b*) ≤$_\tau$ (*TYPE*(*'a*))
shows *d'* ⊨$_t$ *q* = *d* ⊨$_t$ *q*
proof
assume *valid-d'-q*: *d'* ⊨$_t$ *q*
show *d* ⊨$_t$ *q*
proof (*cases d* ⊨$_t$ *q*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have *invalid-d-q*: ¬ *d* ⊨$_t$ *q* **using** *False* **.**

from *stack-alloc* **obtain**
d': *d'* = *ptr-force-type p d* **and**
typ-uinfo-t (*TYPE*(*'a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) **and**
ptr-span p ⊆ $\mathcal{S}$ **and**
no-stack-d: ∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*) **and**
ptr-aligned p **and** *c-guard p* **and** *root-p*: *root-ptr-valid d' p*
by (*cases rule*: *stack-alloc-cases*)
from *valid-d'-q* **obtain**
footprint-d'-q: *valid-footprint d'* (*ptr-val q*) (*typ-uinfo-t TYPE*(*'b*)) **and**
c-guard-q: *c-guard q*
by (*auto simp add*: *h-t-valid-def*)
with *invalid-d-q*
have *no-footprint-d-q*: ¬ (*valid-footprint d* (*ptr-val q*) (*typ-uinfo-t TYPE*(*'b*)))
by (*simp add*: *d' h-t-valid-def*)
from *root-p valid-d'-q other* **have** *ptr-span q* ∩ *ptr-span p* = {}
by (*meson disj-inter-swap root-ptr-valid-not-subtype-disjoint*)
with *no-footprint-d-q ptr-force-type-valid-footprint-disjoint2* [*OF footprint-d'-q* [*simplified d'*]]
have *False*
by (*simp add*: *size-of-tag*)
then show *?thesis* **by** *simp*
qed
next
assume *d* ⊨$_t$ *q* **from** *stack-alloc-preserves-typing* [*OF stack-alloc no-stack this*]
show *d'* ⊨$_t$ *q* **.**
qed

lemma *TYPE*(*'b*::*c-type*) ≤$_\tau$ (*TYPE*(*'a*::*c-type*)) ⟹ *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*'a*)
⟹ ¬*TYPE*(*'a*) ≤$_\tau$ (*TYPE*(*'b*))

using *typ-uinfo-eq-sub-typ-conv*(*3*) **by** *blast*

named-theorems *stack-alloc-ptr-valid-parent* **and** *stack-release-ptr-valid-parent*

lemma *stack-alloc-root-ptr-valid-other*:
fixes *q*::*'b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*(*'a*::*mem-type*)) *d*
assumes *neq*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*'a*)
assumes *sub*: *TYPE*(*'a*) ≤$_\tau$ (*TYPE*(*'b*))
shows *root-ptr-valid d'* (*q*::*'b*::*mem-type ptr*) = *root-ptr-valid d q*
proof
assume *valid-d'*: *root-ptr-valid d' q*
show *root-ptr-valid d q*
using *neq stack-alloc stack-alloc-root-ptr-valid-new-cases valid-d'* **by** *blast*
next
assume *root-ptr-valid d q*
then show *root-ptr-valid d' q*
by (*smt* (*verit*, *best*) *sub stack-alloc stack-alloc-cases stack-release-root-ptr-valid-cases*

stack-release-stack-alloc-inverse sub-typ-def sub-typ-stack-byte)
qed

lemma *stack-release-root-ptr-valid-other*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*
assumes *root*: *root-ptr-valid d p*
assumes *non-stack*: *typ-uinfo-t* (*TYPE*(*'a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *neq*: *typ-uinfo-t TYPE*(*'b*::*mem-type*) ≠ *typ-uinfo-t TYPE*(*'a*)
assumes *sub*: *TYPE*(*'a*) ≤$_\tau$ (*TYPE*(*'b*))
shows *root-ptr-valid* (*stack-release p d*) (*q*::*'b*::*mem-type ptr*) = *root-ptr-valid d q*
by (*metis neq non-stack root root-ptr-valid-type-neq-disjoint*
stack-release-root-ptr-valid1 stack-release-root-ptr-valid2 sub sub-typ-def sub-typ-stack-byte)

lemma *bex-impl-forall-conv*: ((∃ *x*∈*A*. *P x*) ⟶ *Q*) ⟷ (∀ *x*. *x* ∈ *A* ⟶ *P x* ⟶ *Q*)
by *auto*

end

20.3 More Stack Typing

theory *Stack-Typing*
imports *L2Defs Runs-To-VCG*
begin

definition *equal-upto*:: *'a set* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool* **where**

equal-upto S f g = (∀ *x*. *x* ∉ *S* ⟶ *f x* = *g x*)

lemma *equal-upto-refl*[*simp*, *intro*]: *equal-upto S f f*
by (*simp add*: *equal-upto-def*)

lemma *symp-equal-upto*: *symp* (*equal-upto S*)
by (*simp add*: *equal-upto-def symp-def*)

lemma *equal-upto-commute*: *equal-upto S f g* ⟷ *equal-upto S g f*
using *symp-equal-upto*
by (*metis sympE*)

lemma *equal-upto-trans*[*trans*]: *equal-upto S f g* ⟹ *equal-upto S g h* ⟹ *equal-upto S f h*
by (*simp add*: *equal-upto-def*)

lemma *equal-upto-mono*: *S* ⊆ *T* ⟹ *equal-upto S f g* ⟹ *equal-upto T f g*
by (*meson equal-upto-def subset-iff*)

definition *equal-on*:: *'a set* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool* **where**
equal-on S f g = (∀ *x*. *x* ∈ *S* ⟶ *f x* = *g x*)

lemma *equal-on-UNIV-iff*[*simp*]: *equal-on UNIV f g* ⟷ *f* = *g*
by (*auto simp add*: *equal-on-def*)

lemma *equal-on-refl* [*simp*, *intro*]: *equal-on S f f*
by (*auto simp add*: *equal-on-def*)

lemma *symp-equal-on*: *symp* (*equal-on S*)
by (*simp add*: *equal-on-def symp-def*)

lemma *equal-on-commute*: *equal-on S f g* ⟷ *equal-on S g f*
using *symp-equal-on*
by (*metis sympE*)

lemma *equal-on-trans*[*trans*]: *equal-on S f g* ⟹ *equal-on S g h* ⟹ *equal-on S f h*
by (*simp add*: *equal-on-def*)

lemma *equal-on-mono*: *S* ⊆ *T* ⟹ *equal-on T f g* ⟹ *equal-on S f g*
by (*meson equal-on-def subset-iff*)

lemma *equal-on-equal-upto-eq*: *equal-on S f g* ⟹ *equal-upto S f g* ⟹ *f* = *g*
by (*auto simp add*: *equal-on-def equal-upto-def*)

named-theorems *unchanged-typing-on-simps* **and** *unchanged-typing*

declare *mex-def* [*unchanged-typing-on-simps*]
declare *meq-def* [*unchanged-typing-on-simps*]

ML ‹
structure Unchanged-Typing =
struct
fun unchanged-typing-tac splitter ctxt =
let
val unchanged-typing-on-simps = Named-Theorems.get ctxt @{named-theorems unchanged-typing-on-simps}
val unchanged-typing = Named-Theorems.get ctxt @{named-theorems unchanged-typing}
val vcg-attrs = map (Attrib.attribute ctxt) @{attributes [runs-to-vcg]}
val (-, ctxt) = ctxt |> fold-map (Thm.proof-attributes vcg-attrs) unchanged-typing
val ctxt = ctxt addsimps unchanged-typing-on-simps

val trace-tac = Runs-To-VCG.no-trace-tac
fun solver-tac ctxt = ALLGOALS (asm-full-simp-tac ctxt)
in
Runs-To-VCG.runs-to-vcg-tac splitter (~1) trace-tac false
{do-nosplit = false, no-unsafe-hyp-subst = false} solver-tac ctxt
end
end
›

lemma *stack-allocs-releases-equal-on-stack*:
(p, d2) ∈ stack-allocs n 𝒮 TYPE('a::mem-type) d1 ⟹
equal-on 𝒮 d2 d3 ⟹ equal-on 𝒮 d1 (stack-releases n p d3)
by *(smt (verit, best) equal-on-def stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse)*

lemma *stack-allocs-releases-equal-on-typing*:
(p, d2) ∈ stack-allocs n 𝒮 TYPE('a::mem-type) d1 ⟹
equal-on X d2 d3 ⟹ equal-on X d1 (stack-releases n p d3)
by *(smt (verit, best) equal-on-def stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse)*

lemma *stack-allocs-releases-equal-on-typing'*:
(p, d2) ∈ stack-allocs n 𝒮 TYPE('a::mem-type) d1 ⟹
equal-on X' d2 d3 ⟹ X ⊆ X' ⟹ equal-on X d1 (stack-releases n p d3)
using *stack-allocs-releases-equal-on-typing*
using *equal-on-mono* **by** *blast*

context *heap-typing-state*
begin

definition *unchanged-typing-on*:: *addr set ⇒ 's ⇒ 's ⇒ bool* **where**
unchanged-typing-on S s t = equal-on S (htd s) (htd t)

lemma *unchanged-typing-on-UNIV-iff*: *unchanged-typing-on UNIV s t ⟷ htd s = htd t*
by *(auto simp add: unchanged-typing-on-def)*

lemma *unchanged-typing-on-UNIV-I*: *htd s = htd t ⟹ unchanged-typing-on UNIV s t*
by (*simp add*: *unchanged-typing-on-UNIV-iff*)

lemma *typing-eq-unchanged-typing-on*: *htd s = htd t ⟹ unchanged-typing-on S s t*
by (*auto simp add*: *unchanged-typing-on-def equal-on-def*)

lemma *unchanged-typing-on-relf*[*simp*]: *unchanged-typing-on S s s*
by (*auto simp add*: *unchanged-typing-on-def equal-on-def*)

lemma *unchanged-typing-on-conv*[*unchanged-typing-on-simps*]: *unchanged-typing-on S s t ⟷ (∀ a ∈ S. htd s a = htd t a)*
by (*auto simp add*: *unchanged-typing-on-def equal-on-def*)

lemma *unchanged-typing-on-cong*: *(⋀a. a ∈ S ⟹ htd s a = htd s' a) ⟹*
(⋀a. a ∈ S ⟹ htd t a = htd t' a)
⟹ unchanged-typing-on S s t ⟷ unchanged-typing-on S s' t
by (*simp add*: *unchanged-typing-on-def equal-on-def*)

lemma *typing-eq-left-unchanged-typing-on*: *htd s1 = htd s2 ⟹*
unchanged-typing-on S s1 t ⟷ unchanged-typing-on S s2 t
by (*auto simp add*: *unchanged-typing-on-def equal-on-def*)

lemma *symp-unchanged-typing-on*: *symp* (*unchanged-typing-on S*)
using *symp-equal-on*
by (*metis symp-def unchanged-typing-on-def*)

lemma *unchanged-typing-on-commute*: *unchanged-typing-on S s t ⟷ unchanged-typing-on S t s*
using *symp-unchanged-typing-on*
by (*metis sympE*)

lemma *unchanged-typing-on-trans*[*trans*]:
assumes *s-t*: *unchanged-typing-on S s t*
assumes *t-u*: *unchanged-typing-on S t u*
shows *unchanged-typing-on S s u*
by (*meson equal-on-trans s-t t-u unchanged-typing-on-def*)

lemma *unchanged-typing-on-mono*: *S ⊆ T ⟹ unchanged-typing-on T s t ⟹ unchanged-typing-on S s t*
using *equal-on-mono*
by (*smt* (*verit*) *unchanged-typing-on-def*)

lemma *unchanged-typing-on-root-ptr-valid-preservation*:
fixes *p*::*'a*::*mem-type ptr*
assumes *unchanged-typing-on S s t*

assumes *ptr-span p* $\subseteq$ *S*
assumes *root-ptr-valid* (*htd s*) *p*
shows *root-ptr-valid* (*htd t*) *p*
using *assms*
by (*simp add*: *equal-on-def root-ptr-valid-domain subset-iff unchanged-typing-on-def*)

simproc-setup *unchanged-typing-nondet* (‹*c* · *s* ?{*λr t. unchanged-typing-on* $\mathcal{S}$ *s t**}›) = ‹
let
val prop-to-meta-eq = @{*lemma* ‹*P* $\Longrightarrow$ (*P* $\equiv$ *True*)› *for P by auto*}
fun try-prove ctxt prop = *try* (*Goal.prove-internal ctxt* [] *prop*)
(*fn - => Unchanged-Typing.unchanged-typing-tac NONE ctxt*)
in
fn phi => fn ctxt => fn ct =>
let
val - = Utils.verbose-msg 3 ctxt (*fn - => unchanged-typing-nondet invoked on*: ^@{*make-string*} *ct*)
in
try-prove ctxt ***instantiate*** ‹*P* = *ct in cterm* ‹*Trueprop P*› *for P*›
|> *Option.map* (*fn thm => prop-to-meta-eq OF* [*thm*])
end

end
›
declare [[*simproc del*: *unchanged-typing-nondet*]]
end

context *stack-heap-state*
begin
lemma *with-fresh-stack-ptr-unchanged-typing*[*unchanged-typing*]:
assumes *f*[*runs-to-vcg*]:$\bigwedge$*p s.* (*f p*) · *s* ?{*λr t. typing.unchanged-typing-on* $\mathcal{S}$ *s t**}
shows (*with-fresh-stack-ptr n I* (*L2-VARS f nm*)) · *s* ?{*λr t. typing.unchanged-typing-on* $\mathcal{S}$ *s t**}
unfolding *with-fresh-stack-ptr-def on-exit-def on-exit'-def L2-VARS-def*
apply (*runs-to-vcg*)
subgoal for *x d vs r t*
thm *typing.typing-eq-left-unchanged-typing-on*
thm *typing.typing-eq-left-unchanged-typing-on* [*of* - (*htd-upd* (λ-. *d*) *s*)]
apply (*subst* (*asm*) *typing.typing-eq-left-unchanged-typing-on* [*of* - (*htd-upd* (λ-. *d*) *s*)])
apply *simp*
apply (*simp add*: *typing.unchanged-typing-on-def stack-allocs-releases-equal-on-stack*)
done
done
end

lemma *override-on-merge*: *override-on* (*override-on f g B*) *g A* = *override-on f g* (*A* $\cup$ *B*)

by (*auto simp add*: *override-on-def fun-eq-iff*)

lemma *override-on-id* [*simp*]: *override-on f f A* = *f*
by (*auto simp add*: *override-on-def*)

lemma *override-cancel-snd* [*simp*]: *override-on f* (*override-on g1 g0 A*) *A* = *override-on f g0 A*
by (*auto simp add*: *override-on-def*)

lemma *override-cancel-subset-snd* : *A* ⊆ *B* ⟹ *override-on f* (*override-on g1 g0 B*) *A* = *override-on f g0 A*
by (*auto simp add*: *override-on-def fun-eq-iff*)

lemma *override-cancel-fst*[*simp*] : *override-on* (*override-on f1 f0 A*) *g A* = *override-on f1 g A*
by (*auto simp add*: *override-on-def fun-eq-iff*)

lemma *override-cancel-subset-fst* : *A* ⊆ *B* ⟹ *override-on* (*override-on f1 f0 A*) *g B* = *override-on f1 g B*
by (*auto simp add*: *override-on-def fun-eq-iff*)

lemma *equal-on-stack-free-preservation*:
assumes *eq*: *equal-on S d2 d1*
assumes *stack-free d1* ⊆ *S*
assumes *stack-free d2* ⊆ *S*
shows *stack-free d2* = *stack-free d1*
using *assms*
by (*smt* (*verit*, *best*) *Abs-fnat-hom-0 Collect-cong One-nat-def add.right-neutral equal-on-def less-Suc0 mem-Collect-eq ptr-val.simps*
root-ptr-valid-def size-of-stack-byte(*3*) *stack-free-def subsetD valid-root-footprint-def*)

lemma *equal-upto-disjoint-h-val*: *equal-upto P h*′ *h* ⟹
ptr-span p ∩ *P* = {} ⟹
h-val h′ (*p*::′*a*::*mem-type ptr*) = *h-val h p*
by (*smt* (*verit*, *best*) *disjoint-iff equal-upto-def h-val-def heap-list-h-eq2*)

context *heap-state*
begin

definition *equal-upto-heap-on*:: *addr set* ⇒ ′*s* ⇒ ′*s* ⇒ *bool* **where**
equal-upto-heap-on S s t = (∃ *T H*.
t = *hmem-upd* (λ-. *H*) (*htd-upd* (λ-. *T*) *s*) ∧
equal-upto S (*hmem s*) *H* ∧
equal-upto S (*htd s*) *T*)

definition *zero-heap-on*:: *addr set* ⇒ ′*s* ⇒ ′*s* **where**
zero-heap-on A s = *hmem-upd* (λ*h*. *override-on h zero-heap A*) (*htd-upd* (λ*htd*.

override-on htd stack-byte-typing A) s)

lemma *equal-upto-heap-on-equal-upto-htd*: *equal-upto-heap-on S s t ⟹ equal-upto S (htd s) (htd t)*
by (*auto simp add*: *equal-upto-heap-on-def*)

lemma *equal-upto-heap-on-equal-upto-hmem*: *equal-upto-heap-on S s t ⟹ equal-upto S (hmem s) (hmem t)*
by (*auto simp add*: *equal-upto-heap-on-def*)

lemma *zero-heap-on-empty*[*simp*]: *zero-heap-on {} s = s*
by (*auto simp add*: *zero-heap-on-def*)

lemma *equal-upto-heap-on-zero-heap-on-subset*: *A ⊆ B ⟹ equal-upto-heap-on B s (zero-heap-on A s)*
apply (*clarsimp simp add*: *zero-heap-on-def equal-upto-heap-on-def override-on-def*)
by (*smt* (*verit*, *best*) *equal-upto-def equal-upto-mono heap.upd-cong hmem-htd-upd typing.upd-cong*)

lemma *equal-upto-heap-on-zero-heap-on*[*simp*]: *equal-upto-heap-on A s (zero-heap-on A s)*
by (*simp add*: *equal-upto-heap-on-zero-heap-on-subset*)

lemma *equal-upto-heap-on-refl*[*simp*, *intro*]: *equal-upto-heap-on S s s*
by (*force simp add*: *equal-upto-heap-on-def*)

lemma *override-on-heap-update-other*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p ⊆ A ⟹ override-on (heap-update p v h) f A = override-on h f A*
apply (*clarsimp simp add*: *override-on-def fun-eq-iff*)
by (*smt* (*verit*, *ccfv-SIG*) *CTypes.mem-type-simps*(*2*) *heap-list-length heap-update-def heap-update-nmem-same subsetD*)

lemma *override-on-heap-update-commute*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p ∩ A = {} ⟹*
override-on (heap-update p v h) f A = heap-update p v (override-on h f A)
apply (*clarsimp simp add*: *override-on-def fun-eq-iff heap-update-def heap-update-nmem-same orthD2*)
by (*smt* (*verit*, *best*) *disjoint-iff heap-list-h-eq2 heap-list-length heap-update-list-value len max-size*)

lemma *override-on-heap-update-padding-other*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p ⊆ A ⟹ length bs = size-of TYPE('a) ⟹ override-on (heap-update-padding p v bs h) f A = override-on h f A*
apply (*clarsimp simp add*: *override-on-def fun-eq-iff*)
by (*smt* (*verit*) *CTypes.mem-type-simps*(*2*) *heap-update-nmem-same heap-update-padding-def*

subsetD)

lemma *zero-heap-on-heap-update-other*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ⊆ *A* ⟹ *zero-heap-on A* (*hmem-upd* (*heap-update p v*) *s*) = *zero-heap-on A s*
by (*simp add*: *zero-heap-on-def heap-commute comp-def override-on-heap-update-other*)

lemma *override-on-stack-byte-typing-stack-allocs*:
assumes *alloc*: (*p*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*::*mem-type*) *d0*
assumes *subset*: *stack-free d0* ⊆ *A*
shows *override-on d stack-byte-typing A* = *override-on d0 stack-byte-typing A*
using *alloc subset*
apply (*simp add*: *override-on-def fun-eq-iff*)
by (*meson in-mono stack-allocs-other stack-allocs-stack-subset-stack-free*)

lemma *zero-heap-on-stack-allocs*:
assumes *alloc*: (*p*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*::*mem-type*) (*htd s*)
assumes *subset*: *stack-free* (*htd s*) ⊆ *A*
shows *zero-heap-on A* (*htd-upd* (λ-. *d*) *s*) = *zero-heap-on A s*
apply (*simp add*: *zero-heap-on-def comp-def*)
using *override-on-stack-byte-typing-stack-allocs* [*OF alloc subset*]
by (*metis* (*no-types*, *lifting*) *typing.upd-cong*)

lemma *zero-heap-on-stack-releases*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *subset*: {*ptr-val p*..+*n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ⊆ *A*
shows *zero-heap-on A* (*htd-upd* (*stack-releases n p*) *s*) = *zero-heap-on A s*
using *subset*
by (*simp add*: *zero-heap-on-def comp-def stack-releases-def override-on-merge Un-absorb2*)

lemma *zero-heap-on-stack-releases'*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *guard*: (⋀*i*. *i* < *n* ⟹ *c-null-guard* (*p* +$_p$ *int i*))
shows *zero-heap-on* (*stack-free* (*stack-releases n p* (*htd s*)) ∪ *A*) (*htd-upd* (*stack-releases n p*) *s*) =
zero-heap-on (*stack-free* (*htd s*) ∪ {*ptr-val p*..+*n* ∗ *size-of TYPE*(*'a*)} ∪ *A*) *s*
apply (*subst stack-free-stack-releases*)
apply (*simp add*: *guard*)
apply (*subst zero-heap-on-stack-releases*)
apply *blast*
apply (*simp add*: *sup-commute*)
done

We want to express that certain portions of the heap (typing and values) are 'irrelevant', in particular regarding (free) stack space. The notion of 'irrelevant' is a bit vague, it means that the behaviour of the program does not depend on those locations and also that it does not modify those locations. Moreover, we prefer an abstraction function rather than an relation between states to avoid admissibility issues for refinement.

definition *frame*:: *addr set* ⇒ *'s* ⇒ *'s* ⇒ *'s* **where**
frame A s_0 *s* =
hmem-upd (λ*h*. *override-on h* (*hmem* s_0) (*A* ∪ *stack-free* (*htd s*)))
(*htd-upd* (λ*d*. *override-on d* (*htd* s_0) (*A* − *stack-free* (*htd s*))) *s*)

The standard use case we have in mind is $A \cap$ *stack-free* (*htd s*) = {}, hence $A -$ *stack-free* (*htd s*) = *A*, but nevertheless the intuition is:

- stack free typing from s is preserved, the framed sate has at least as many stack free addresses as the original one. So we can simulate any stack allocation.
- heap values for stack free and A are taken from reference state s_0, this captures that we do not depend on the original values in s for those addresses.
- typing for allocations in A is taken from reference state s_0, this captures that we do not depend on the original typing in s for those addresses.

  By taking the same reference state s_0 to frame two states s and s', we can express that the 'irrelevant' parts of the heap did not change in the respective framed states *frame A* s_0 *s* and *frame A* s_0 s'.

definition *raw-frame*:: *addr set* ⇒ *'s* ⇒ *'s* ⇒ *'s* **where**
raw-frame A s_0 *s* =
hmem-upd (λ*h*. *override-on h* (*hmem* s_0) *A*)
(*htd-upd* (λ*d*. *override-on d* (*htd* s_0) *A*) *s*)

lemma *frame-heap-independent-selector*:
(⋀*f s*. *sel* (*hmem-upd f s*) = *sel s*) ⟹ (⋀*f s*. *sel* (*htd-upd f s*) = *sel s*) ⟹
sel (*frame A* s_0 *s*) = *sel s*
by (*auto simp add*: *frame-def*)

lemma *frame-id*[*simp*]: *frame A s s* = *s*
unfolding *frame-def*
by (*simp add*: *heap.upd-same typing.upd-same*)

lemma *frame-hmem-UNIV*[*simp*]: *hmem* (*frame UNIV* s_0 *s*) = *hmem* s_0
unfolding *frame-def*
by *simp*

lemma *hmem-frame*: *hmem* (*frame A* s_0 *s*) = (λ*a*. *if a* ∈ *A* ∪ *stack-free* (*htd s*) *then hmem* s_0 *a else hmem s a*)
unfolding *frame-def*
by (*auto simp add*: *fun-eq-iff*)

lemma *htd-frame*: *htd* (*frame A* s_0 *s*) = (λ*a*. *if a* ∈ (*A* − *stack-free* (*htd s*)) *then htd* s_0 *a else htd s a*)
unfolding *frame-def*
by (*auto simp add*: *fun-eq-iff*)

lemma *stack-free-htd-frame*: *stack-free* (*htd s*) ⊆ *stack-free* (*htd* (*frame A* s_0 *s*))
apply (*clarsimp simp add*: *htd-frame stack-free-def root-ptr-valid-def*)
by (*smt* (*verit*) *One-nat-def add.right-neutral less-Suc0 mem-Collect-eq orthD2 semiring-1-class.of-nat-0 size-of-stack-byte*(*3*) *valid-root-footprint-def*)

lemma *stack-free-htd-frame'*: *stack-free* (*htd* s_0) ∩ *A* = {} ⟹ *stack-free* (*htd* (*frame A* s_0 *s*)) = *stack-free* (*htd s*)
by (*auto simp add*: *htd-frame stack-free-def root-ptr-valid-def valid-root-footprint-def*)

lemma *equal-on-hmem-frame*: *equal-on* (*A* ∪ *stack-free* (*htd s*)) (*hmem* (*frame A* s_0 *s*)) (*hmem* s_0)
by (*auto simp add*: *equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-upto-hmem-frame*: *equal-upto* (*A* ∪ *stack-free* (*htd s*)) (*hmem* (*frame A* s_0 *s*)) (*hmem s*)
by (*auto simp add*: *equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-on-htd-frame*: *stack-free* (*htd s*) ∩ *A* = {} ⟹ *equal-on A* (*htd* (*frame A* s_0 *s*)) (*htd* s_0)
by (*auto simp add*: *equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-on-htd-stack-free-frame*: *stack-free* (*htd s*) ∩ *A* = {} ⟹
equal-on (*stack-free* (*htd s*)) (*htd* (*frame A* s_0 *s*)) (*htd s*)
by (*auto simp add*: *equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-upto-htd-frame*: *equal-upto A* (*htd* (*frame A* s_0 *s*)) (*htd s*)
by (*auto simp add*: *equal-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)

lemma *equal-upto-heap-on-frame*: *equal-upto-heap-on* (*A* ∪ *stack-free* (*htd s*)) (*frame A* s_0 *s*) *s*
apply (*clarsimp simp add*: *equal-upto-heap-on-def frame-def equal-upto-def override-on-def fun-eq-iff*)
by (*smt* (*verit*, *ccfv-threshold*) *K-record-comp heap.upd-compose heap.upd-same heap-commute typing.upd-compose typing.upd-same*)

lemma *frame-heap-update*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ⊆ *A* ⟹ *frame A* s_0 (*hmem-upd* (*heap-update p v*) *s*) = *frame*

A s_0 *s*
apply (*clarsimp simp add*: *frame-def*)
apply (*subst heap-commute* [*symmetric*])
apply (*simp add*: *comp-def*)
apply (*subst override-on-heap-update-other*)
apply *auto*
done

lemma *frame-heap-update-disjoint*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ∩ *A* = {} ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {} ⟹
frame A s_0 (*hmem-upd* (*heap-update p v*) *s*) = *hmem-upd* (*heap-update p v*) (*frame A* s_0 *s*)
apply (*clarsimp simp add*: *frame-def*)
apply (*subst heap-commute* [*symmetric*])
apply (*simp add*: *comp-def*)
apply (*subst override-on-heap-update-commute*)
apply *blast*
apply *blast*
done

lemma *h-val-frame-disjoint'*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ∩ *A* = {} ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {} ⟹
ptr-span p = *ptr-span p'* ⟹
h-val (*hmem* (*frame A* s_0 *s*)) *p'* = *h-val* (*hmem s*) *p'*
apply (*clarsimp simp add*: *frame-def override-on-def*)
by (*smt* (*verit*, *del-insts*) *disjoint-iff h-val-def heap-list-h-eq2 hmem-htd-upd*)

lemma *h-val-frame-disjoint*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ∩ *A* = {} ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {} ⟹
h-val (*hmem* (*frame A* s_0 *s*)) *p* = *h-val* (*hmem s*) *p*
apply (*erule* (*1*) *h-val-frame-disjoint'*)
by (*rule refl*)

lemma *h-val-frame-disjoint-globals*:
fixes *p*::*'a*::*mem-type ptr*
assumes *G* ∩ *A* = {} *G* ∩ *stack-free* (*htd s*) = {}
assumes *ptr-span p* ⊆ *G*
shows *h-val* (*hmem* (*frame A* s_0 *s*)) *p* = *h-val* (*hmem s*) *p*
apply (*rule h-val-frame-disjoint*)
subgoal using *assms* **by** *blast*
subgoal using *assms* **by** *blast*
done

lemma *frame-heap-update-padding*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ⊆ *A* ⟹ *length bs* = *size-of TYPE*(*'a*) ==> *frame A* s_0

(*hmem-upd* (*heap-update-padding p v bs*) *s*) = *frame A* s_0 *s*
apply (*clarsimp simp add*: *frame-def*)
apply (*subst heap-commute* [*symmetric*])
apply (*simp add*: *comp-def*)
apply (*subst override-on-heap-update-padding-other*)
apply *auto*
done

lemma *frame-stack-alloc*:
fixes *p*::'*a*::*mem-type ptr*
assumes *subset*: *ptr-span p* ⊆ *stack-free* (*htd s*)
assumes *disjoint*: *stack-free* (*htd s*) ∩ *A* = {}
assumes *alloc*: *stack-free* (*ptr-force-type p* (*htd s*)) = *stack-free* (*htd s*) − *ptr-span p*
shows
frame (*ptr-span p* ∪ *A*) (*htd-upd* (λ*d*. *override-on d stack-byte-typing* (*ptr-span p*)) t_0)
(*htd-upd* (λ-. *ptr-force-type p* (*htd s*)) *s*) = *frame A* t_0 *s*
proof −
from *alloc subset disjoint*
have ∗: *ptr-span p* ∪ *A* ∪ *stack-free* (*ptr-force-type p* (*htd s*)) = *A* ∪ *stack-free* (*htd s*)
by *auto*
from *subset disjoint alloc*
have ∗∗:*override-on* (*ptr-force-type p* (*htd s*))
(*override-on* (*htd* t_0) *stack-byte-typing* (*ptr-span p*)) (*ptr-span p* ∪ *A* − *stack-free* (*ptr-force-type p* (*htd s*))) = *override-on* (*htd s*) (*htd* t_0) (*A* − *stack-free* (*htd s*))
apply (*clarsimp simp add*: *override-on-def fun-eq-iff ptr-force-type-d*)
by (*smt* (*verit*, *ccfv-threshold*) *add.right-neutral mem-Collect-eq old.prod.exhaust*

prod.sel(*1*) *prod.sel*(*2*) *ptr-val.ptr-val-def root-ptr-valid-def semiring-1-class.of-nat-0 stack-byte-typing-footprint stack-free-def subset-iff valid-root-footprint-def*)
show *?thesis*
apply (*clarsimp simp add*: *frame-def comp-def* ∗ ∗∗)
by (*metis* (*no-types*, *lifting*) *typing.upd-cong*)
qed

lemma *frame-stack-release*:
fixes *p*::'*a*::*mem-type ptr*
assumes *disjoint-free*: *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disjoint-old*: *ptr-span p* ∩ *A* = {}
assumes *c-null-guard*: *c-null-guard p*
assumes *lbs*: *length bs* = *size-of TYPE* ('*a*)
shows *frame* (*ptr-span p* ∪ *A*) (*htd-upd* (λ*d*. *override-on d stack-byte-typing* (*ptr-span p*)) t_0) *s* =
frame A t_0 (*hmem-upd* (*heap-update-list* (*ptr-val p*) *bs*) (*htd-upd* (*stack-releases* (*Suc 0*) *p*) *s*))

proof −
from *disjoint-free c-null-guard*
have ∗: (*ptr-span p* ∪ *A* ∪ *stack-free* (*htd s*)) = *A* ∪ *stack-free* (*stack-releases* (*Suc 0*) *p* (*htd s*))
by (*metis* (*no-types*, *opaque-lifting*) *Un-assoc add.right-neutral mult-0 stack-free-stack-release*
stack-release-def stack-releases-def sup-commute times-nat.simps(*2*))

from *disjoint-free disjoint-old c-null-guard*
have ∗∗: *override-on* (*htd s*) (*override-on* (*htd* t_0) *stack-byte-typing* (*ptr-span p*)) (*ptr-span p* ∪ *A* − *stack-free* (*htd s*)) =
override-on (*stack-releases* (*Suc 0*) *p* (*htd s*)) (*htd* t_0) (*A* − *stack-free* (*stack-releases* (*Suc 0*) *p* (*htd s*)))
apply (*clarsimp simp add*: *override-on-def fun-eq-iff stack-byte-typing-footprint stack-releases-footprint stack-releases-other*,
intro conjI impI)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*metis UnE add.right-neutral mult-is-0 stack-free-stack-release stack-release-def stack-releases-def times-nat.simps*(*2*))
using *stack-free-stack-releases-mono* **by** *blast*

have ∗∗∗: *override-on* (*heap-update-list* (*ptr-val p*) *bs* (*hmem s*)) (*hmem* t_0) (*A* ∪ *stack-free* (*stack-releases* (*Suc 0*) *p* (*htd s*)))
=
override-on (*hmem s*) (*hmem* t_0) (*A* ∪ *stack-free* (*stack-releases* (*Suc 0*) *p* (*htd s*)))
using *disjoint-old lbs* ∗
apply (*clarsimp simp add*: *override-on-def fun-eq-iff stack-byte-typing-footprint stack-releases-footprint stack-releases-other*)
by (*metis Un-iff heap-update-nmem-same lbs*)

show *?thesis*
apply (*simp add*: *frame-def comp-def* ∗)
apply (*simp add*: *heap-commute*)
using ∗∗ ∗∗∗
by (*metis* (*no-types*, *lifting*) *heap.get-upd heap.upd-compose heap.upd-cong htd-hmem-upd*
typing.get-upd typing.upd-compose typing.upd-cong)
qed

lemma *frame-stack-release-keep*:
fixes *p*::′*a*::*mem-type ptr*
assumes *disjoint-free*: *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disjoint-old*: *ptr-span p* ∩ *A* = {}
assumes *c-null-guard*: *c-null-guard p*
assumes *lbs*: *length bs* = *size-of TYPE*(′*a*)

shows *hmem-upd* (*heap-update-list* (*ptr-val p*) (*heap-list* (*hmem* t_0) (*size-of TYPE*('*a*)) (*ptr-val p*)))
(*htd-upd* (*stack-releases* (*Suc 0*) *p*) (*frame A* t_0 *s*)) =
frame A t_0
(*hmem-upd* (*heap-update-list* (*ptr-val p*) *bs*)
(*htd-upd* (*stack-releases* (*Suc 0*) *p*) *s*))
proof −
from *disjoint-free c-null-guard*
have ∗: (*ptr-span p* ∪ *A* ∪ *stack-free* (*htd s*)) = *A* ∪ *stack-free* (*stack-releases* (*Suc 0*) *p* (*htd s*))
by (*metis* (*no-types*, *opaque-lifting*) *Un-assoc add.right-neutral mult-0 stack-free-stack-release*
stack-release-def stack-releases-def sup-commute times-nat.simps(*2*))

from *disjoint-free disjoint-old c-null-guard*
have ∗∗: *stack-releases* (*Suc 0*) *p* (*override-on* (*htd s*) (*htd* t_0) (*A* − *stack-free* (*htd s*))) =
override-on (*stack-releases* (*Suc 0*) *p* (*htd s*)) (*htd* t_0) (*A* − *stack-free* (*stack-releases* (*Suc 0*) *p* (*htd s*)))
apply (*clarsimp simp add*: *override-on-def fun-eq-iff stack-byte-typing-footprint stack-releases-footprint stack-releases-other*,
intro conjI impI)
subgoal by (*smt* (*verit*, *del-insts*) *Un-iff add.right-neutral mult-is-0 stack-free-stack-release*
stack-release-def stack-release-other stack-releases-def times-nat.simps(*2*))
subgoal by (*smt* (*verit*, *best*) *UnE less-Suc0 ptr-add-0-id semiring-1-class.of-nat-0 stack-free-stack-releases*
stack-releases-footprint stack-releases-other)
done

have ∗∗∗: *heap-update-list* (*ptr-val p*) (*heap-list* (*hmem* t_0) (*size-of TYPE*('*a*)) (*ptr-val p*))
(*override-on* (*hmem s*) (*hmem* t_0) (*A* ∪ *stack-free* (*htd s*))) =
override-on (*heap-update-list* (*ptr-val p*) *bs* (*hmem s*)) (*hmem* t_0)
(*A* ∪ *stack-free* (*stack-releases* (*Suc 0*) *p* (*htd s*)))
using *lbs disjoint-old disjoint-free* ∗
apply (*clarsimp simp add*: *override-on-def fun-eq-iff heap-update-nmem-same orthD2 heap-update-list-value*)
apply (*smt* (*verit*, *ccfv-SIG*) *Un-iff heap-list-length heap-update-list-id*′ *heap-update-list-value max-size*)
done
show *?thesis*
apply (*simp add*: *frame-def comp-def* ∗ *heap-commute*)
using ∗∗ ∗∗∗
by (*metis* (*no-types*, *lifting*) *heap.upd-cong htd-hmem-upd lense.upd-cong typing.lense-axioms*)
qed

lemma *symp-equal-upto-heap-on*: *symp* (*equal-upto-heap-on S*)
proof
  fix *s t*
  assume *equal-upto-heap-on S s t*
  then obtain *T H* **where**
    t: *t* = *hmem-upd* (λ-. *H*) (*htd-upd* (λ-. *T*) *s*) **and**
    eq-H: *equal-upto S* (*hmem s*) *H* **and**
    eq-T: *equal-upto S* (*htd s*) *T*
    by (*auto simp add*: *equal-upto-heap-on-def*)
  show *equal-upto-heap-on S t s*
  proof −
    define *H′* **where** *H′* = *hmem s*
    define *T′* **where** *T′* = *htd s*
    have *s* = *hmem-upd* (λ-. *H′*) (*htd-upd* (λ-. *T′*) *t*)
      unfolding *H′-def T′-def*
      apply (*subst t*)
      apply (*simp add*: *heap-commute heap.upd-same typing.upd-same*)
      done
    moreover
    from *eq-H* **have** *equal-upto S* (*hmem t*) *H′*
      by (*simp add*: *H′-def equal-upto-commute t*)
    moreover
    from *eq-T* **have** *equal-upto S* (*htd t*) *T′*
      by (*simp add*: *T′-def equal-upto-commute t*)
    ultimately show *?thesis*
      by (*auto simp add*: *equal-upto-heap-on-def*)
  qed
qed

lemma *equal-upto-heap-on-commute*: *equal-upto-heap-on S s t* ⟷ *equal-upto-heap-on S s t*
  using *symp-equal-upto-heap-on*
  by (*metis sympE*)

lemma *equal-upto-heap-on-trans*[*trans*]:
  assumes *s-t*: *equal-upto-heap-on S s t*
  assumes *t-u*: *equal-upto-heap-on S t u*
  shows *equal-upto-heap-on S s u*
proof −
  from *s-t* **obtain** *T H* **where**
    t: *t* = *hmem-upd* (λ-. *H*) (*htd-upd* (λ-. *T*) *s*) **and**
    eq-H: *equal-upto S* (*hmem s*) *H* **and**
    eq-T: *equal-upto S* (*htd s*) *T*
     by (*auto simp add*: *equal-upto-heap-on-def*)

  from *t-u* **obtain** *T′ H′* **where**
    u: *u* = *hmem-upd* (λ-. *H′*) (*htd-upd* (λ-. *T′*) *t*) **and**
    eq-H′: *equal-upto S* (*hmem t*) *H′* **and**
    eq-T′: *equal-upto S* (*htd t*) *T′*

by (*auto simp add*: *equal-upto-heap-on-def*)

have $u = hmem\text{-}upd\ (\lambda\text{-}.\ H')\ (htd\text{-}upd\ (\lambda\text{-}.\ T')\ s)$
apply (*subst u*)
apply (*subst t*)
apply (*simp add*: *heap-commute comp-def*)
done

moreover
from *eq-H'*
have *equal-upto S* (*hmem s*) *H'*
apply −
apply (*subst* (*asm*) *t*)
apply *simp*
using *eq-H equal-upto-trans* **by** *blast*

moreover
from *eq-T'*
have *equal-upto S* (*htd s*) *T'*
apply −
apply (*subst* (*asm*) *t*)
apply *simp*
using *eq-T equal-upto-trans* **by** *blast*

ultimately show *?thesis*
by (*auto simp add*: *equal-upto-heap-on-def*)
qed

lemma *equal-upto-heap-on-mono*: $S \subseteq T \Longrightarrow$ *equal-upto-heap-on S s t* $\Longrightarrow$ *equal-upto-heap-on T s t*
using *equal-upto-mono*
by (*smt* (*verit*) *equal-upto-heap-on-def*)

end

lemma *runs-to-partial-L2-modify*[*runs-to-vcg*]:
(*L2-modify f*) · *s* ?⦃$\lambda r\ t.\ r = Result\ ()\ \wedge\ t = f\ s$⦄
unfolding *L2-defs*
apply *runs-to-vcg*
done

lemma *runs-to-partial-L2-unknown*[*runs-to-vcg*]:
($\bigwedge x.\ P\ (Result\ x)\ s$) $\Longrightarrow$ (*L2-unknown ns*) · *s* ?⦃*P*⦄
unfolding *L2-defs*
apply *runs-to-vcg*
apply *simp*
done

lemma *runs-to-partial-L2-seq*[*runs-to-vcg*]:
  $f \cdot s$?{| $\lambda r\ t.\ (\forall a.\ r = Result\ a \longrightarrow g\ a \cdot t$?{| Q |}$) \wedge (\forall e.\ r = Exn\ e \longrightarrow Q\ (Exn\ e)\ t)$ |} $\Longrightarrow$
    $(L2\text{-}seq\ f\ g) \cdot s$?{| Q |}
  unfolding *L2-defs*
  apply *runs-to-vcg*
  done

lemma (**in** *heap-typing-state*) *runs-to-partial-L2-seq*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
  assumes [*runs-to-vcg*]: $\bigwedge r\ s.\ (g\ r) \cdot s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
  shows $(L2\text{-}seq\ f\ g) \cdot s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
  by (*runs-to-vcg*) (*auto simp add*: *unchanged-typing-on-simps*)

lemma *runs-to-partial-L2-gets*[*runs-to-vcg*]:
  $P\ (Result\ (f\ s))\ s \Longrightarrow (L2\text{-}gets\ f\ ns) \cdot s$?{| P |}
  unfolding *L2-defs*
  apply *runs-to-vcg*
  done

lemma *runs-to-partial-L2-condition*[*runs-to-vcg*]:
  $(P\ s \Longrightarrow f \cdot s$?{|Q|}$) \Longrightarrow (\neg\ P\ s \Longrightarrow g \cdot s$?{|Q|}$) \Longrightarrow (L2\text{-}condition\ P\ f\ g) \cdot s$?{|Q|}
  unfolding *L2-defs*
  apply *runs-to-vcg*
   apply *auto*
  done

lemma *runs-to-partial-L2-catch*[*runs-to-vcg*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{| $\lambda r\ t.\ (\forall e.\ r = Exn\ e \longrightarrow ((g\ e) \cdot t$?{| P |}$)) \wedge (\forall v'.\ r = Result\ v' \longrightarrow P\ (Result\ v')\ t)$ |}
  shows $(L2\text{-}catch\ f\ g) \cdot s$?{|P |}
  unfolding *L2-defs*
  using *assms*
  apply *runs-to-vcg*
  done

lemma (**in** *heap-typing-state*) *runs-to-partial-L2-catch*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
  assumes [*runs-to-vcg*]: $\bigwedge r\ s.\ (g\ r) \cdot s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
  shows $(L2\text{-}catch\ f\ g) \cdot s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
  by (*runs-to-vcg*) (*auto simp add*: *unchanged-typing-on-simps*)

lemma *runs-to-partial-L2-try*[*runs-to-vcg*]:
  assumes [*runs-to-vcg*]:
    $f \cdot s$?{| $\lambda r.\ Q\ (unnest\text{-}exn\ r)$ |}
  shows $(L2\text{-}try\ f) \cdot s$?{| Q |}
  unfolding *L2-defs*

apply *runs-to-vcg*
done

lemma (**in** *heap-typing-state*) *runs-to-partial-L2-try*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: $f \cdot s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
shows (*L2-try* f) $\cdot\ s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
by (*runs-to-vcg*)

lemma *runs-to-partial-L2-while*:
assumes B: $\bigwedge r\ s.\ I\ (\mathit{Result}\ r)\ s \Longrightarrow C\ r\ s \Longrightarrow (B\ r) \cdot s$?{|I|}
assumes Qr: $\bigwedge r\ s.\ I\ (\mathit{Result}\ r)\ s \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ (\mathit{Result}\ r)\ s$
assumes Ql: $\bigwedge e\ s.\ I\ (\mathit{Exn}\ e)\ s \Longrightarrow Q\ (\mathit{Exn}\ e)\ s$
assumes I: $I\ (\mathit{Result}\ r)\ s$
shows (*L2-while* $C\ B\ r\ ns$) $\cdot\ s$?{| Q |}
unfolding *L2-defs*
by (*rule runs-to-partial-whileLoop-exn* [**where** $I{=}I$, *OF* $I\ Qr\ Ql\ B$])

lemma *runs-to-partial-L2-while-same-post*:
assumes B: $\bigwedge r\ s.\ Q\ (\mathit{Result}\ r)\ s \Longrightarrow C\ r\ s \Longrightarrow (B\ r) \cdot s$?{|Q|}
assumes I: $Q\ (\mathit{Result}\ r)\ s$
shows (*L2-while* $C\ B\ r\ ns$) $\cdot\ s$?{| Q |}
using *assms*
by $-$ (*rule runs-to-partial-L2-while*)

lemma (**in** *heap-typing-state*) *runs-to-partial-L2-while-unchanged-typing*[*unchanged-typing*]:
assumes B[*runs-to-vcg*]: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow (B\ r) \cdot s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
shows (*L2-while* $C\ B\ r\ ns$) $\cdot\ s$?{|$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$|}
supply *runs-to-partial-L2-while-same-post* [*runs-to-vcg*]
apply (*runs-to-vcg*)
apply (*auto simp add*: *unchanged-typing-on-simps*)
done

lemma *runs-to-partial-L2-throw*[*runs-to-vcg*]:
assumes P (*Exn* e) s
shows (*L2-throw* e ns) $\cdot\ s$?{| P |}
unfolding *L2-defs*
using *assms*
by (*rule runs-to-partial-yield*)

lemma *runs-to-partial-L2-spec*[*runs-to-vcg*]: (*L2-spec* R) $\cdot\ s$?{|$\lambda r\ t.\ (s,\ t) \in R$|}
unfolding *L2-defs*
by *runs-to-vcg*

lemma *runs-to-partial-state-L2-assume*[*runs-to-vcg*]:
(*L2-assume* R) $\cdot\ s$?{|$\lambda r\ t.\ \exists x.\ r = \mathit{Result}\ x \wedge (x,\ t) \in R\ s$|}

unfolding *L2-defs*
apply (*runs-to-vcg*)
done

lemma *runs-to-partial-L2-guard*[*runs-to-vcg*]:
shows (*L2-guard P*) · *s* ?{| λ*r t*. *r* = *Result* () ∧ *s* = *t* ∧ *P s*|}
unfolding *L2-defs*
apply (*runs-to-vcg*)
done

definition *GUARDED-ASSM* :: *bool* ⇒ *bool* **where** [*remove-ASSMs*]: *GUARDED-ASSM P* ⟷ *P*
lemma *GUARDED-ASSM-D*: *GUARDED-ASSM P* ⟹ *P* **by** (*simp add*: *GUARDED-ASSM-def*)

lemma *runs-to-partial-L2-guarded*[*runs-to-vcg*]:
(*GUARDED-ASSM* (*P s*) ⟹ *c* · *s* ?{|*Q*|}) ⟹ (*L2-guarded P c*) · *s* ?{|*Q*|}
unfolding *L2-guarded-def GUARDED-ASSM-def*
by *runs-to-vcg*

lemma *runs-to-partial-L2-fail-conv*[*simp*]: *L2-fail* · *s* ?{| *R* |} ⟷ *True*
by (*simp add*: *L2-fail-def*)

lemma *runs-to-partial-L2-fail*[*runs-to-vcg*]: *L2-fail* · *s* ?{| *R* |}
by *simp*

lemma *runs-to-partial-L2-call*[*runs-to-vcg*]:
assumes [*runs-to-vcg*]: *m* · *s* ?{| λ*r*. *Q* (*case r of Exn e* ⇒ *Exn* (*f e*) | *Result r* ⇒ *Result r*) |}
shows (*L2-call m f ns*) · *s* ?{| *Q* |}
unfolding *L2-call-def*
apply *runs-to-vcg*
apply (*simp add*: *map-exn-def*)
done

end

20.4 In Out Parameter Refinement

theory *In-Out-Parameters*
imports
L2ExceptionRewrite
L2Peephole
TypHeapSimple
Stack-Typing
begin

lemma *map-exn-catch-conv*: *map-value* (*map-exn f*) *m* = (*m* <*catch*> (λ*r*. *throw* (*f r*)))

apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*auto simp add*: *runs-to-def-old map-exn-def split*: *xval-splits*)
done

abbreviation *L2-return x ns* ≡ *liftE* (*L2-VARS* (*return x*) *ns*)

lemma *L2-return-L2-gets-conv*: *L2-return x ns* = *L2-gets* (λ-. *x*) *ns*
unfolding *L2-defs L2-VARS-def*
by (*simp add*: *gets-return*)

lemma *return-L2-gets-conv*: (*return x*) = *L2-gets* (λ-. *x*) []
unfolding *L2-defs L2-VARS-def*
by (*simp add*: *gets-return*)

named-theorems *refines-right-eq*

lemma (**in** *heap-state*) *IO-modify-heap-paddingE-root-refines'*:
fixes *p*::*'a*::*xmem-type ptr*
fixes *fld-update*::(*'b*::*xmem-type* ⇒ *'b*) ⇒ *'a* ⇒ *'a*
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *cgrd*: *c-guard p*
shows *refines*
(*IO-modify-heap-paddingE* (*PTR*(*'b*) &(*p*→*f*)) *v*)
(*IO-modify-heap-paddingE p* (λ*s*. (*fld-update* (λ-. *v s*)) (*h-val* (*hmem s*) *p*)))
s s ((=))
proof −
{
fix *r t*
assume *exec*: *reaches* (*IO-modify-heap-paddingE* (*PTR*(*'b*) &(*p*→*f*)) *v*) *s r t*
have *reaches* (*IO-modify-heap-paddingE p* (λ*s*. *fld-update* (λ-. *v s*) (*h-val* (*hmem s*) *p*))) *s r t*
proof −
from *exec* **obtain** *bs* **where**
r: *r* = *Result* () **and** *bs*: *length bs* = *size-of TYPE*(*'b*) **and**
t: *t* = *hmem-upd* (*heap-update-padding* (*PTR*(*'b*) &(*p*→*f*)) (*v s*) *bs*) *s*
unfolding *liftE-IO-modify-heap-padding*
by (*auto simp add*: *IO-modify-heap-padding-def*)
note *root-conv* = *heap-update-padding-field-root-conv''* [*OF fl fg-cons cgrd bs*, **where** *v*=*v s* **and** *hp* = *hmem s*]
from *bs cgrd fl* **have** ∗: *length* (*super-update-bs bs* (*heap-list* (*hmem s*) (*size-of TYPE*(*'a*)) (*ptr-val p*)) *n*) = *size-of TYPE*(*'a*)
apply (*subst length-super-update-bs*)
subgoal
by (*metis add.commute field-lookup-offset-size heap-list-length size-of-tag*

typ-desc-size-update-ti typ-uinfo-size)
subgoal
using *heap-list-length* **by** *blast*
done
show *?thesis*
unfolding *liftE-IO-modify-heap-padding*
apply (*clarsimp simp add*: *IO-modify-heap-padding-def r*)
using *root-conv heap.upd-cong* ∗ *t*
by *blast*
qed
} **note** ∗ = *this*
show *?thesis*
using ∗
by (*auto simp add*: *refines-def-old*)
qed

lemma (**in** *heap-state*) *IO-modify-heap-paddingE-root-refines''*:
fixes *p*::*'a*::*xmem-type ptr*
fixes *fld-update*::(*'b*::*xmem-type* ⇒ *'b*) ⇒ *'a* ⇒ *'a*
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *fl*: *field-ti TYPE*(*'a*) *f* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*))
fld (*fld-update* ∘ (λ*x* -. *x*)))
assumes *cgrd*: *c-guard p*
shows *refines*
(*IO-modify-heap-paddingE* (*PTR*(*'b*) &(*p*→*f*)) *v*)
(*IO-modify-heap-paddingE p* (λ*s*. (*fld-update* (λ-. *v s*)) (*h-val* (*hmem s*)
p)))
s s ((=))
proof −
from *fl* **obtain** *n* **where**
field-lookup (*typ-info-t TYPE*(*'a*)) *f* 0 = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*))
fld (*fld-update* ∘ (λ*x* -. *x*)), *n*)
using *field-ti-field-lookupE* **by** *blast*
from *IO-modify-heap-paddingE-root-refines'* [*OF fg-cons this cgrd*]
show *?thesis* **.**
qed

lemma (**in** *heap-state*) *IO-modify-heap-paddingE-root-refines* [*refines-right-eq*]:
fixes *p*::*'a*::*xmem-type ptr*
fixes *fld-update*::(*'b*::*xmem-type* ⇒ *'b*) ⇒ *'a* ⇒ *'a*
assumes *v'*: ⋀*s*. *v' s* = ((*fld-update* (λ-. *v s*)) (*h-val* (*hmem s*) *p*))
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *fl*: *field-ti TYPE*(*'a*) *f* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*))
fld (*fld-update* ∘ (λ*x* -. *x*)))
assumes *cgrd*: *c-guard p*
shows *refines*
(*IO-modify-heap-paddingE* (*PTR*(*'b*) &(*p*→*f*)) *v*)
(*IO-modify-heap-paddingE p v'*)
s s ((=))

unfolding v'
by (*rule IO-modify-heap-paddingE-root-refines''* [*OF fg-cons fl cgrd*])

lemma *refines-subst-right*:
assumes *f-g*: *refines f g s t Q*
assumes *refines-eq*: *refines g g' t t* ((=))
shows *refines f g' s t Q*
using *f-g refines-eq*
by (*force simp add*: *refines-def-old*)

lemma *refines-right-eq-id*: *refines f f s s* ((=))
by (*force simp add*: *refines-def-old*)

lemma *refines-subst-left*:
assumes *f-g*: *refines f g s t Q*
assumes *f-eq*: *run f s* = *run f' s*
shows *refines f' g s t Q*
using *f-g f-eq*
apply (*auto simp add*: *refines-def-old reaches-def succeeds-def*)
done

lemma *refines-right-eq-L2-seq* [*refines-right-eq*]:
assumes *f1*: *refines f1 g1 s s* ((=))
assumes *f2*: $\bigwedge v\ t$. *refines* (*f2 v*) (*g2 v*) *t t* ((=))
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s s* ((=))
unfolding *L2-defs*
apply (*rule refines-bind-bind-exn* [*OF f1*])
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal using *f2* **by** *auto*
done

lemma *refines-right-eq-L2-guard'*:
assumes *Q s* $\Longrightarrow$ *P s*
assumes *Q s* $\Longrightarrow$ *refines X X' s s* (=)
shows *refines* (*L2-seq* (*L2-guard P*) (λ-. *X*)) (*L2-seq* (*L2-guard Q*) (λ-. *X'*)) *s s* ((=))
unfolding *L2-defs*
apply (*rule refines-bind'*[*OF*
refines-guard[*THEN refines-strengthen*[*OF - runs-to-partial-guard runs-to-partial-guard*]]])
apply (*auto simp*: *assms*)
done

lemma *refines-right-eq-L2-guard*:
assumes *Q* $\Longrightarrow$ *P*
assumes *Q* $\Longrightarrow$ *refines X X' s s* (=)

shows *refines* (*L2-seq* (*L2-guard* (λ-. *P*)) (λ-. *X*)) (*L2-seq* (*L2-guard* (λ-. *Q*)) (λ-. *X'*)) *s s* ((=))
using *assms*
by (*rule refines-right-eq-L2-guard'*)

lemma *refines-L2-guarded*:
assumes *grd*: *g' t* ⟹ *g s*
assumes *X-X'*: *g s* ⟹ *g' t* ⟹ *refines X X' s t Q*
shows *refines* (*L2-guarded g X*) (*L2-guarded g' X'*) *s t Q*
using *grd X-X'*
apply (*simp add*: *L2-guarded-def L2-seq-def L2-guard-def*)
apply (*rule refines-bind'*)
apply (*rule refines-guard*)
apply *simp*
apply *clarsimp*
done

lemma *select-UNIV-L2-unknown-conv*: (*select UNIV*) = *L2-unknown ns*
unfolding *L2-defs*
by *simp*

lemma *select-singleton-conv*: ((*select* ({*x*})) >>= *g*) = *g x*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *hd-UNIV*: *hd* ‘ *UNIV* ⊆ *UNIV*
by (*rule subset-UNIV*)

lemma *hd-singleton*: *hd* ‘ {[*x*]} ⊆ {*x*}
by *simp*

lemma (**in** *heap-state*) *refines-exec-IO-modify-heap-padding-step-right*:
fixes *p*:: *'a*::*mem-type ptr*
assumes
refines (*return x*) *g s*
(*hmem-upd* (*heap-update-padding p v* (*heap-list* (*hmem s*) (*size-of TYPE*('*a*)) (*ptr-val p*))) *t*) *Q*
shows *refines* (*return x*) (*do* { - <− *IO-modify-heap-paddingE p* (λ-. *v*); *g* }) *s t Q*
using *assms* **unfolding** *liftE-IO-modify-heap-padding*
apply (*clarsimp simp add*: *refines-def-old IO-modify-heap-padding-def reaches-bind succeeds-bind*)
apply (*metis heap-list-length*)
done

lemma (**in** *heap-state*) *refines-exec-IO-modify-heap-padding-single-step-right*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *Q* (*Result* (), *s*)
(*Result* (), *hmem-upd* (*heap-update-padding p* (*v t*) (*heap-list* (*hmem s*) (*size-of TYPE*(*'a*)) (*ptr-val p*))) *t*)
shows *refines* (*return* ())
(*IO-modify-heap-paddingE p v*) *s t Q*
using *assms* **unfolding** *liftE-IO-modify-heap-padding*
apply (*clarsimp simp add*: *refines-def-old IO-modify-heap-padding-def reaches-bind succeeds-bind*)
by (*metis heap-list-length*)

lemma (**in** *heap-state*) *refines-exec-IO-modify-heap-padding-single-step-right-InL*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *Q* (*Exn e*, *s*) (*Exn e'*, *hmem-upd* (*heap-update-padding p* (*v t*) (*heap-list* (*hmem s*) (*size-of TYPE*(*'a*)) (*ptr-val p*))) *t*)
shows *refines* (*throw e*)
(*do* { - <− *IO-modify-heap-paddingE p v*;
L2-throw e' ns
})
s t Q
unfolding *L2-defs* **unfolding** *liftE-IO-modify-heap-padding*
using *assms*
apply (*clarsimp simp add*: *refines-def-old IO-modify-heap-padding-def reaches-bind succeeds-bind Exn-def* [*symmetric*])
apply (*metis heap-list-length*)
done

lemma *refines-exec-gets-right*:
assumes *Q* (*Result x*, *s*) (*Result* (*g t*), *t*)
shows *refines* (*return x*) (*gets g*) *s t Q*
using *assms*
by (*auto simp add*: *refines-def-old*)

lemma *refines-exec-L2-return-right*:
assumes *Q* (*Result x*, *s*) (*Result w*, *t*)
shows *refines* (*return* (*x*)) (*L2-return w ns*) *s t Q*
using *assms* **unfolding** *L2-VARS-def*
by (*auto simp add*: *refines-def-old*)

lemma *refines-L2-catch-right*:
assumes *f*: *refines f g s t Q*
assumes *Res-Res*: ⋀*v v' s' t'*. *Q* (*Result v*, *s'*) (*Result v'*, *t'*) ⟹ *R* (*Result v*, *s'*) (*Result v'*, *t'*)
assumes *Res-Exn*: ⋀*v e' s' t'*. *Q* (*Result v*, *s'*) (*Exn e'*, *t'*) ⟹ *refines* (*return v*) (*h e'*) *s' t' R*
assumes *Exn-Res*: ⋀*e v' s' t'*. *Q* (*Exn e*, *s'*) (*Result v'*, *t'*) ⟹ *R* (*Exn e*, *s'*)

(*Result* v', t')
assumes *Exn-Exn*: $\bigwedge e\ e'\ s'\ t'.\ Q\ (Exn\ e,\ s')\ (Exn\ e',\ t') \Longrightarrow$ *refines* (*throw* *e*) (*h* e') $s'\ t'\ R$
shows *refines f* (*L2-catch g h*) *s t R* **unfolding** *L2-defs*
apply (*subst f-catch-throw*[*symmetric*])
apply (*rule refines-catch* [*OF f*])
subgoal using *Exn-Exn* **by** *auto*
subgoal using *Exn-Res* **by** (*auto simp add*: *refines-def-old Exn-def* [*symmetric*])
subgoal using *Res-Exn* **by** (*auto simp add*: *refines-def-old Exn-def* [*symmetric*])
subgoal using *Res-Res* **by** *auto*
done

lemma *L2-seq-gets-app-conv*: *run* (*L2-seq* (*L2-gets f ns*) *g*) *s* = *run* (*g* (*f s*)) *s*
unfolding *L2-defs*
apply (*auto simp add*: *run-bind*)
done

lemma *refines-project-right*:
assumes *f-g*: *refines f g s t Q*
assumes *run g t* = *run* (g' (*prj t*)) *t*
shows *refines f* (*L2-seq* (*L2-gets prj ns*) g') *s t Q*
unfolding *L2-defs*
using *assms*
apply (*clarsimp simp add*: *refines-def-old reaches-bind succeeds-bind*)
apply (*auto simp add*: *succeeds-def reaches-def*)
done

lemma *refines-project-guard-right*:
assumes *f-g*: *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t Q*
assumes *P t* $\Longrightarrow$ *run g t* = *run* (g' (*prj t*)) *t*
shows *refines f* (*L2-seq* (*L2-guard P*) (λ-. (*L2-seq* (*L2-gets prj ns*) g'))) *s t Q*
using *assms* **unfolding** *L2-defs*
apply (*clarsimp simp add*: *refines-def-old reaches-bind succeeds-bind*)
apply (*auto simp add*: *succeeds-def reaches-def*)
done

named-rules *cguard-assms* **and** *alloc-assms* **and** *modifies-assms* **and** *disjoint-assms* **and**
disjoint-alloc **and** *disjoint-stack-free* **and** *stack-ptr* **and** *h-val-globals-frame-eq* **and**
rel-alloc-independent-globals **and** *keep-non-stack-ptr-eqs*
synthesize-rules *refines-in-out*
add-synthesize-pattern *refines-in-out* =
‹
[(*Binding.make* (*concrete-function*, ***here***), *fn ctxt* => *fn i* => *fn t* =>
(*case t of*

@{*term-pat Trueprop* (*refines* (*L2-modify* (λ*s*. *?glob-upd* - -)) - - - -)} => *glob-upd*
| @{*term-pat Trueprop* (*refines* (*L2-modify* (*?glob-upd* -)) - - - -)} => *glob-upd*
| @{*term-pat Trueprop* (*refines ?f* - - - -)} => *f*
| *t* => *t*))]
›

lemma *refines-yield-both*[*simp*]: *refines* (*return a*) (*return b*) *s t R* ⟷ *R* (*Result a*, *s*) (*Result b*, *t*)
by (*auto simp add*: *refines-def-old*)

lemma *disjoint-union-distrib*: $A \cap (B \cup C) = \{\} \longleftrightarrow A \cap B = \{\} \wedge A \cap C = \{\}$
by *blast*

lemma *inter-commute*: $A \cap B = B \cap A$ **by** *blast*
lemma *disjoint-symmetric*: $A \cap B = \{\} \Longrightarrow B \cap A = \{\}$
by *auto*
lemma *disjoint-symmetric′*: $A \cap B \equiv \{\} \Longrightarrow B \cap A = \{\}$
by *auto*

definition *IOcorres*::
(′*s* ⇒ *bool*) ⇒
(′*s* ⇒ (′*e*,′*a*) *xval* ⇒ ′*s* ⇒ *bool*) ⇒
(′*s* ⇒ ′*t*) ⇒
(′*a* ⇒ ′*s* ⇒ ′*b*) ⇒
(′*e* ⇒ ′*s* ⇒ ′*e1*) ⇒
(′*e1*, ′*b*, ′*t*) *exn-monad* =>
(′*e*, ′*a*, ′*s*) *exn-monad* => *bool* **where**
IOcorres P Q st rx ex f_a f_c ≡ *corresXF-post st rx ex P Q* f_a f_c

lemma *IOcorres-id*: *IOcorres* (λ-. *True*) (λ - - -. *True*) (λ*s*. *s*) (λ*r* -. *r*) (λ*r* -. *r*) *f f*
by (*auto simp add*: *IOcorres-def corresXF-post-def split*: *xval-splits prod.splits*)
lemmas *IOcorres-skip* = *IOcorres-id*

lemma *IOcorres-trivial-from-local-var-extract*:
L2corres st rx ex P A C ⟹ *IOcorres* (λ-. *True*) (λ - - -. *True*) (λ*s*. *s*) (λ*r* -. *r*) (λ*r* -. *r*) *A A*
by (*rule IOcorres-id*)

lemma *admissible-IOcorres* [*corres-admissible*]:
ccpo.admissible Inf (≥) (λf_a. *IOcorres P Q st rx ex* f_a f_c)
unfolding *IOcorres-def*
by (*rule admissible-nondet-ord-corresXF-post*)

lemma *IOcorres-top* [*corres-top*]: *IOcorres P Q st rx ex* ⊤ f_c
unfolding *IOcorres-def*
by (*rule corresXF-post-top*)

lemma *distinct-addresses-ptr-val-lemma*:
n < *addr-card* ⟹ *ptr-val p* + *word-of-nat n* ∉ (λ*x*. *ptr-val p* + *word-of-nat x*) ‘ {*0*..<*n*}
by *auto* (*metis addr-card-len-of-conv nless-le order-le-less-trans unat-of-nat-eq*)

lemma *distinct-addresses-ptr-lemma*:
assumes *bound*: *n* ≤ *addr-card*
shows *distinct* (*map* (λ*i*. *ptr-val p* + *of-nat i*) [*0*..<*n*])
using *bound*
proof (*induct n arbitrary*: *p*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc* **have** *hyp*: *distinct* (*map* (λ*i*. *ptr-val p* + *word-of-nat i*) [*0*..<*n*]) **by** *auto*
show *?case*
apply (*subst upt-Suc-snoc*)
apply (*subst map-append*)
apply (*subst distinct-append*)
apply (*simp add*: *hyp*)
using *Suc.prems*
apply (*simp add*: *distinct-addresses-ptr-val-lemma*)
done
qed

lemma *array-intvl-Suc-split*:{*a*..+*Suc n* ∗ *m*} = {*a*..+*n* ∗ *m*} ∪ {*a* + (*word-of-nat* (*n*∗*m*))..+*m*}
by (*metis add-diff-cancel-right′ intvl-split le-add2 mult-Suc*)

definition *rel-singleton-stack* :: *′a*::*c-type ptr* ⇒ *heap-mem* ⇒ *unit* ⇒ *′a* ⇒ *bool*
where
rel-singleton-stack p h = (λ(-::*unit*) *v*.
v = *h-val h p*)

lemma *domain-rel-singleton-stack*:
equal-on (*ptr-span p*) *h h′* ⟹ *rel-singleton-stack p h* = *rel-singleton-stack p h′*
apply (*clarsimp simp add*: *fun-eq-iff rel-singleton-stack-def*)
apply (*metis* (*mono-tags*, *lifting*) *equal-on-def h-val-def heap-list-h-eq2*)+
done

named-theorems *rel-stack-intros* **and** *rel-stack-simps* **and** *rel-entail*

lemma *rel-singleton-stack-simp* [*rel-stack-simps*]:
rel-singleton-stack p h x v $\longleftrightarrow$ *v = h-val h p*
by (*auto simp add*: *rel-singleton-stack-def*)

lemma *rel-stack-refl* [*rel-stack-intros*]: (λ-. (=)) *h x x*
by *auto*

lemma *rel-singleton-stackI* [*rel-stack-intros*]: *rel-singleton-stack p h x* (*h-val h p*)
by (*auto simp add*: *rel-singleton-stack-def*)

lemma *rel-singleton-stack-condI* [*rel-stack-intros*]: *h-val h p = y* $\Longrightarrow$ *rel-singleton-stack p h x y*
by (*auto simp add*: *rel-singleton-stack-def*)

lemma *fun-of-rel-singleton-stack*[*fun-of-rel-intros*]: *fun-of-rel* (*rel-singleton-stack p h*) (λ-. (*h-val h p*))
by (*auto simp add*: *fun-of-rel-def rel-singleton-stack-def*)

lemma *funp-rel-singleton-stack*[*funp-intros*, *corres-admissible*]: *funp* (*rel-singleton-stack p h*)
by (*auto simp add*: *rel-singleton-stack-def funp-def fun-of-rel-def*)

definition *rel-push* ::
'a::*c-type ptr* ⇒ (*heap-mem* ⇒ *'b* ⇒ *'c* ⇒ *bool*) ⇒ *heap-mem* ⇒
'b ⇒ *'a* × *'c* ⇒ *bool*
where
rel-push p R h = (λ*r* (*v*, *x*).
R h r x ∧
v = h-val h p)

lemma *rel-singleton-stack-rel-push-conv*: *rel-singleton-stack p* = (λ*h x y*. *rel-push p* (λ- - -. *True*) *h x* (*y*, ()))
by (*auto simp add*: *rel-singleton-stack-def rel-push-def*)

lemma *rel-push-simp*[*rel-stack-simps*]: *rel-push p R h r* (*v*, *x*) $\longleftrightarrow$
R h r x ∧ *v = h-val h p*
by (*auto simp add*: *rel-push-def*)

lemma *fun-of-rel-puhsh* [*fun-of-rel-intros*]:
fun-of-rel (*R h*) *f* $\Longrightarrow$ *fun-of-rel* (*rel-push p R h*) (λ*x*. (*h-val h p*, *f x*))
by (*auto simp add*: *fun-of-rel-def rel-push-def*)

lemma *funp-rel-push*[*funp-intros*, *corres-admissible*]: *funp* (*R h*) $\Longrightarrow$ *funp* (*rel-push p R h*)
by (*force simp add*: *funp-def rel-push-def fun-of-rel-def*)

lemma *rel-push-stackI* [*rel-stack-intros*]: $Q\ h\ x\ y \Longrightarrow$ *rel-push* $p\ Q\ h\ x\ ((h\text{-}val\ h\ p),\ y)$
by (*auto simp add*: *rel-push-def*)

lemma *rel-push-stack-condI* [*rel-stack-intros*]: $h\text{-}val\ h\ p = v \Longrightarrow Q\ h\ x\ y \Longrightarrow$ *rel-push* $p\ Q\ h\ x\ (v,\ y)$
by (*auto simp add*: *rel-push-def*)

definition *rel-sum-stack* $L\ R\ h =$ *rel-sum* $(L\ h)\ (R\ h)$
definition *rel-xval-stack* $L\ R\ h =$ *rel-xval* $(L\ h)\ (R\ h)$

lemma *rel-sum-stack-expand-sum-eq*:
(*rel-sum-stack* $(\lambda\text{-}.\ (=))\ R) =$ (*rel-sum-stack* (*rel-sum-stack* $(\lambda\text{-}.\ (=))\ (\lambda\text{-}.\ (=)))$ R)
by (*auto simp add*: *rel-sum-stack-def fun-eq-iff rel-sum.simps split*: *sum.splits*)

lemma *rel-xval-stack-expand-sum-eq*:
(*rel-xval-stack* $(\lambda\text{-}.\ (=))\ R) =$ (*rel-xval-stack* (*rel-sum-stack* $(\lambda\text{-}.\ (=))\ (\lambda\text{-}.\ (=)))$ R)
by (*auto simp add*: *rel-sum-stack-def rel-xval-stack-def fun-eq-iff rel-sum.simps sum.rel-eq*
split: *xval-splits sum.splits*)

lemma *rel-sum-stack-expand-sum-bot*:
$(\lambda\text{- - -}.\ False) =$ (*rel-sum-stack* $(\lambda\text{- - -}.\ False)\ (\lambda\text{- - - }.\ False)$)
by (*auto simp add*: *rel-sum-stack-def fun-eq-iff rel-sum.simps split*: *sum.splits*)

lemma *rel-xval-stack-expand-xval-bot*:
$(\lambda\text{- - -}.\ False) =$ (*rel-xval-stack* $(\lambda\text{- - -}.\ False)\ (\lambda\text{- - - }.\ False)$)
by (*auto simp add*: *rel-xval-stack-def fun-eq-iff rel-xval.simps split*: *xval-splits*)

lemma *rel-sum-stack-entail*:
assumes L: $\bigwedge v\ v'.\ L'\ h\ v\ v' \Longrightarrow L\ h\ v\ v'$
assumes R: $\bigwedge v\ v'.\ R'\ h\ v\ v' \Longrightarrow R\ h\ v\ v'$
assumes *rel-sum-stack* $L'\ R'\ h\ x\ y$
shows *rel-sum-stack* $L\ R\ h\ x\ y$
using *assms*
by (*auto simp add*: *rel-sum-stack-def rel-sum.simps split*: *sum.splits*)

lemma *rel-xval-stack-entail*:
assumes L: $\bigwedge v\ v'.\ L'\ h\ v\ v' \Longrightarrow L\ h\ v\ v'$
assumes R: $\bigwedge v\ v'.\ R'\ h\ v\ v' \Longrightarrow R\ h\ v\ v'$
assumes *rel-xval-stack* $L'\ R'\ h\ x\ y$
shows *rel-xval-stack* $L\ R\ h\ x\ y$
using *assms*
by (*auto simp add*: *rel-xval-stack-def rel-xval.simps*)

lemma *rel-sum-stack-intros*:
L h l1 l2 $\Longrightarrow$ *rel-sum-stack L R h* (*Inl l1*) (*Inl l2*)
R h r1 r2 $\Longrightarrow$ *rel-sum-stack L R h* (*Inr r1*) (*Inr r2*)
by (*auto simp add*: *rel-sum-stack-def*)

lemma *rel-xval-stack-intros*:
L h l1 l2 $\Longrightarrow$ *rel-xval-stack L R h* (*Exn l1*) (*Exn l2*)
R h r1 r2 $\Longrightarrow$ *rel-xval-stack L R h* (*Result r1*) (*Result r2*)
by (*auto simp add*: *rel-xval-stack-def*)

lemma *fun-of-rel-sum-stack*[*fun-of-rel-intros*]:
fun-of-rel (*L h*) *f-l* $\Longrightarrow$ *fun-of-rel* (*R h*) *f-r* $\Longrightarrow$ *fun-of-rel* (*rel-sum-stack L R h*) (*sum-map f-l f-r*)
unfolding *rel-sum-stack-def*
by (*auto intro*: *fun-of-rel-intros*)

lemma *fun-of-rel-xval-stack*[*fun-of-rel-intros*]:
fun-of-rel (*L h*) *f-l* $\Longrightarrow$ *fun-of-rel* (*R h*) *f-r* $\Longrightarrow$ *fun-of-rel* (*rel-xval-stack L R h*) (*map-xval f-l f-r*)
unfolding *rel-xval-stack-def*
by (*auto intro*: *fun-of-rel-intros*)

lemma *funp-rel-sum-stack*[*funp-intros*, *corres-admissible*]: *funp* (*L h*) $\Longrightarrow$ *funp* (*R h*) $\Longrightarrow$ *funp* (*rel-sum-stack L R h*)
unfolding *rel-sum-stack-def* **by** (*auto intro*: *funp-intros*)

lemma *funp-rel-xval-stack*[*funp-intros*, *corres-admissible*]: *funp* (*L h*) $\Longrightarrow$ *funp* (*R h*) $\Longrightarrow$ *funp* (*rel-xval-stack L R h*)
unfolding *rel-xval-stack-def* **by** (*auto intro*: *funp-intros*)

lemma *rel-sum-stack-cases*:
rel-sum-stack L R h x y =
$((\exists\, v\ w.\ x = Inl\ v \wedge y = Inl\ w \wedge L\ h\ v\ w) \vee$
$(\exists\, v\ w.\ x = Inr\ v \wedge y = Inr\ w \wedge R\ h\ v\ w))$
by (*auto simp add*: *rel-sum-stack-def rel-sum.simps*)

lemma *rel-xval-stack-cases*:
rel-xval-stack L R h x y =
$((\exists\, v\ w.\ x = Exn\ v \wedge y = Exn\ w \wedge L\ h\ v\ w) \vee$
$(\exists\, v\ w.\ x = Result\ v \wedge y = Result\ w \wedge R\ h\ v\ w))$
by (*auto simp add*: *rel-xval-stack-def rel-xval.simps*)

lemma *rel-sum-stack-simps* [*simp*]:
rel-sum-stack L R h (*Inl* l_1) (*Inl* l_2) = *L h* l_1 l_2
rel-sum-stack L R h (*Inr* r_1) (*Inr* r_2) = *R h* r_1 r_2

rel-sum-stack L R h (Inl l₁) (Inr r₂) = False
rel-sum-stack L R h (Inr r₁) (Inl l₂) = False
rel-sum-stack L R h (Inl l₁) y = (∃ w. y = Inl w ∧ L h l₁ w)
rel-sum-stack L R h (Inr r₁) y = (∃ w. y = Inr w ∧ R h r₁ w)
rel-sum-stack L R h x (Inl l₂) = (∃ v. x = Inl v ∧ L h v l₂)
rel-sum-stack L R h x (Inr r₂) = (∃ v. x = Inr v ∧ R h v r₂)
by (*auto simp add*: *rel-sum-stack-def rel-sum.simps*)

lemma *rel-xval-stack-simps* [*simp*]:
rel-xval-stack L R h (Exn l₁) (Exn l₂) = L h l₁ l₂
rel-xval-stack L R h (Result r₁) (Result r₂) = R h r₁ r₂
rel-xval-stack L R h (Exn l₁) (Result r₂) = False
rel-xval-stack L R h (Result r₁) (Exn l₂) = False
rel-xval-stack L R h (Exn l₁) y = (∃ w. y = Exn w ∧ L h l₁ w)
rel-xval-stack L R h (Result r₁) y = (∃ w. y = Result w ∧ R h r₁ w)
rel-xval-stack L R h x (Exn l₂) = (∃ v. x = Exn v ∧ L h v l₂)
rel-xval-stack L R h x (Result r₂) = (∃ v. x = Result v ∧ R h v r₂)
by (*auto simp add*: *rel-xval-stack-def rel-xval.simps*)

lemma *rel-sum-stack-eq-collapse*: (*rel-sum-stack* (λ-. (=)) (λ-. (=))) = ((λ-. (=)))

by (*auto simp add*: *rel-sum-stack-cases fun-eq-iff*)

lemma *rel-xval-stack-eq-collapse*: (*rel-xval-stack* (λ-. (=)) (λ-. (=))) = ((λ-. (=)))

apply (*clarsimp simp add*: *rel-xval-stack-cases fun-eq-iff*)
by (*metis Exn-def default-option-def exception-or-result-cases not-Some-eq*)

lemma *rel-sum-stack-InlI*: *L h l1 l2* ⟹ *rel-sum-stack L R h* (*Inl l1*) (*Inl l2*)
by (*simp*)

lemma *rel-xval-stack-ExnI*: *L h l1 l2* ⟹ *rel-xval-stack L R h* (*Exn l1*) (*Exn l2*)
by (*simp*)

lemma *rel-sum-stack-InrI*: *R h r1 r2* ⟹ *rel-sum-stack L R h* (*Inr r1*) (*Inr r2*)
by (*simp*)

lemma *rel-xval-stack-ResultI*: *R h r1 r2* ⟹ *rel-xval-stack L R h* (*Result r1*) (*Result r2*)
by (*simp*)

definition *rel-exit Q h* = (λ*v w*. ∃ *x*. *v* = *Nonlocal x* ∧ *Q h x w*)

lemma *rel-exit-simps*[*simp*]:
rel-exit Q h (*Nonlocal x*) *y* = *Q h x y*
rel-exit Q h Break *y* = *False*
rel-exit Q h Continue *y* = *False*
rel-exit Q h Return *y* = *False*

rel-exit Q h (Goto l) y = False
by (*auto simp add*: *rel-exit-def*)

lemma *rel-exit-intro*: *Q h x y* ⟹ *rel-exit Q h (Nonlocal x) y*
by (*auto simp add*: *rel-exit-def*)

lemma *rel-exit-entail*[*rel-entail*]:
(⋀*x y*. *R h x y* ⟹ *R′ h x y*) ⟹
rel-exit R h x y ⟹ *rel-exit R′ h x y*
by (*auto simp add*: *rel-exit-def*)

lemma *rel-xval-stack-rel-exit-intro*:
assumes ⋀*x*. *rel-xval-stack (rel-exit Q) R h (Exn (Nonlocal (the-Nonlocal x))) (Exn x)*
assumes *rel-exit Q h e e′*
shows *rel-xval-stack (rel-exit Q) R h (Exn (Nonlocal (the-Nonlocal e))) (Exn e′)*
using *assms*
by (*auto simp add*: *rel-exit-def*)

lemma *rel-xval-stack-rel-exit-intro′*:
assumes ⋀*x*. *rel-xval-stack (rel-exit Q) R h (Exn (Nonlocal x)) (Exn x)*
assumes *Q h e e′*
shows *rel-xval-stack (rel-exit Q) R h (Exn (Nonlocal e)) (Exn e′)*
using *assms*
by (*auto simp add*: *rel-exit-def*)

lemma *rel-exit-False-conv* [*simp*]: *rel-exit* (λ- - -. *False*) *h e e′* ⟷ *False*
by (*auto simp add*: *rel-exit-def*)

lemma *rel-exit-FalseE*: *rel-exit* (λ- - -. *False*) *h e e′* ⟹ *L*
by *simp*

lemma *rel-sum-stack-generalise-left*:
rel-sum-stack L R h v w ⟹ (⋀*v w*. *L h v w* ⟹ *L′ h v w*) ⟹ *rel-sum-stack L′ R h v w*
by (*auto simp add*: *rel-sum-stack-def rel-sum.simps*)

lemma *rel-xval-stack-generalise-left*:
rel-xval-stack L R h v w ⟹ (⋀*v w*. *L h v w* ⟹ *L′ h v w*) ⟹ *rel-xval-stack L′ R h v w*
by (*auto simp add*: *rel-xval-stack-def rel-xval.simps*)

lemma *rel-sum-stack-generalise-both*:
rel-sum-stack L R h v w ⟹ (⋀*v w*. *L h v w* ⟹ *L′ h v w*) ⟹ (⋀*v w*. *R h v w* ⟹ *R′ h v w*) ⟹

rel-sum-stack L′ R′ h v w
by (*auto simp add*: *rel-sum-stack-def rel-sum.simps*)

lemmas *generalise-unreachable-exitE* =
rel-exit-FalseE
FalseE
rel-sum-stack-generalise-both
rel-xval-stack-generalise-left

lemma *fun-of-rel-rel-exit*: *fun-of-rel* (*L h*) *f-l* ⟹ *fun-of-rel* (*rel-exit L h*) (λ*v*. *case v of Nonlocal x* ⇒ *f-l x* | - ⇒ *undefined*)
by (*auto simp add*: *fun-of-rel-def split*: *c-exntype.splits*)

lemma *funp-rel-exit* [*funp-intros*, *corres-admissible*]: *funp* (*L h*) ⟹ *funp* (*rel-exit L h*)
using *fun-of-rel-rel-exit*
by (*metis funp-def*)

named-theorems *equal-upto-simps*
named-theorems *refines-stack-intros*

lemma *equal-uptoI*:
assumes *eq*: ⋀*x*. *x* ∉ *A* ⟹ *f x* = *g x*
shows *equal-upto A f g*
using *eq*
by (*auto simp add*: *equal-upto-def*)

lemma *equal-upto-heap-update*[*equal-upto-simps*]:
fixes *p*:: *'a*::*mem-type ptr*
assumes (*ptr-span p*) ⊆ *A*
shows *equal-upto A* (*heap-update p v h1*) *h2* = *equal-upto A h1 h2*
using *assms*
by (*smt* (*verit*, *best*) *CTypes.mem-type-simps*(*2*) *equal-upto-def heap-list-length heap-update-def heap-update-nmem-same subset-iff*)

lemma *equal-upto-complement*: *equal-upto B f g* = *equal-on* (− *B*) *f g*
by (*simp add*: *equal-on-def equal-upto-def*)

lemma *equal-upto-update-left-equalize*:
equal-on (− *F*) (*f s*) *s* ⟹ *equal-on F* (*f s*) *t* ⟹ *equal-upto* (*F* ∪ *A*) *s t* = *equal-upto A* (*f s*) *t*
by (*smt* (*verit*) *Compl-iff Un-iff equal-on-def equal-upto-def*)

lemma *admissible-const-snd*:
assumes *admissible-fst*: *ccpo.admissible* ⋂ (λ*x y*. *y* ⊆ *x*) (λ*X*. ∃ *w* ∈ *X*. *Q w*)
shows *ccpo.admissible* ⋂ (λ*x y*. *y* ⊆ *x*) (λ*X*. ∃ *w*. (*w*, *v*) ∈ *X* ∧ *Q w*)
proof (*rule ccpo.admissibleI*[*rule-format*])
fix *C*::(*'a* × *'b*) *set set*

assume *chain*: *Complete-Partial-Order.chain* $(\lambda x\ y.\ y \subseteq x)\ C$
assume *nonempty*: $C \neq \{\}$
assume *chain-prop*: $\bigwedge X.\ X \in C \Longrightarrow \exists\, w.\ (w,\ v) \in X \wedge Q\ w$
show $\exists\, w.\ (w,\ v) \in \bigcap\ C \wedge Q\ w$
proof −
define C' **where** $C' = Set.filter\ (\lambda(r,\ t).\ t =\ v)\ `\ C$
have *subset*: $\bigcap\ C' \subseteq \bigcap\ C$
using C'*-def* **by** *fastforce*
moreover
have *snd-C'*: $\bigwedge X\ t.\ X \in C' \Longrightarrow\ t \in snd\ `\ X \Longrightarrow t = v$
using C'*-def*
by *auto*
moreover
from *chain* **have** *chain-C'*: *Complete-Partial-Order.chain* $(\lambda x\ y.\ y \subseteq x)\ C'$
using *chain-prop* C'*-def*
by (*simp add*: *chain-imageI subset-iff*)
from *nonempty chain-prop* **have** *nonempty-C'*: $C' \neq \{\}$
using C'*-def*
by *blast*

from *chain-C'* **have** *result-chain*: *Complete-Partial-Order.chain* $(\lambda x\ y.\ y \subseteq x)$ $((\lambda X.\ fst\ `\ X)\ `\ C')$
by (*simp add*: *chain-imageI image-mono*)
from *nonempty-C'* **have** *nonempty-result-chain*: $((\lambda X.\ fst\ `\ X)\ `\ C') \neq \{\}$ **by** *auto*
{
fix X::$'a$ *set*
assume $X \in (`)\ fst\ `\ C'$
then obtain *X0* **where** *X*: $X = fst\ `\ Set.filter\ (\lambda(r,\ t).\ t =\ v)\ X0$ **and** *X0*: $X0 \in C$
by (*auto simp add*: C'*-def*)
from *chain-prop* [*OF X0*] **obtain** w **where** w: $(w,\ v) \in X0$ **and** Q: $Q\ w$ **by** *blast*
from w X **have** $w \in X$
apply *clarsimp*
by (*metis* (*mono-tags*, *lifting*) *case-prodI fst-conv image-iff member-filter*)
with Q
have $\exists\, w{\in}X.\ Q\ w$ **..**
} **note** *result-chain-prop* = *this*
from *ccpo.admissibleD* [*OF admissible-fst result-chain nonempty-result-chain result-chain-prop*]
obtain r' **where**
r': $r' \in\ \bigcap\ ((\lambda X.\ fst\ `\ X)\ `\ C')$ **and** Q: $Q\ r'$
by *auto*
from r' *snd-C'* **have** $(r',\ v) \in \bigcap\ C'$
by *fastforce*
with Q *subset* **show** *?thesis* **by** *blast*
qed
qed

lemma *admissible-funp*: *funp Q* $\Longrightarrow$ *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists\, w \in X.\ Q\ v\ w)$
apply (*clarsimp simp add*: *funp-def fun-of-rel-def*)
by (*smt* (*verit*, *del-insts*) *InterI admissible-const ccpo.admissible-def*)

lemma *admissible-funp-conj*: *funp Q* $\Longrightarrow$ *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.$ $\exists\, w \in X.\ Q\ v\ w \wedge P\ w)$
apply (*clarsimp simp add*: *funp-def fun-of-rel-def*)
by (*smt* (*verit*, *del-insts*) *InterI admissible-const ccpo.admissible-def*)

lemma *override-on-frame-raw-frame-lemma1*:
assumes *disj-A-B*: $A \cap B = \{\}$
assumes *sf*: *stack-free d* $\cap$ $B = \{\}$
shows
override-on d (*override-on d' d B*) $(A \cup B -$ *stack-free d*$)$ $=$
override-on d d' $(A -$ *stack-free d*$)$
using *disj-A-B sf*
by (*auto simp add*: *override-on-def fun-eq-iff*)

lemma *override-on-frame-raw-frame-lemma2*:
assumes *disj-A-B*: $A \cap B = \{\}$
assumes *sf*: *stack-free d* $\cap$ $B = \{\}$
shows
override-on h' (*override-on h h' B*) $(A \cup B \cup$ *stack-free d*$) =$
override-on h' h $(A \cup$ *stack-free d*$)$
using *disj-A-B sf*
by (*auto simp add*: *override-on-def fun-eq-iff*)

lemma *disjoint-stack-free-equal-upto-trans*:
equal-upto A d' d $\Longrightarrow$
$P \cap A = \{\} \Longrightarrow$ *stack-free d* $\cap$ $P = \{\} \Longrightarrow$
stack-free d' $\cap$ $P = \{\}$
apply (*clarsimp simp add*: *equal-upto-def*)
using *root-ptr-valid-def stack-free-def valid-root-footprint-def* **by** *fastforce*

lemma *disjoint-stack-free-equal-on-trans*:
equal-on S d' d $\Longrightarrow$
$P \subseteq S \Longrightarrow$ *stack-free d* $\cap$ $P = \{\} \Longrightarrow$
stack-free d' $\cap$ $P = \{\}$
apply (*clarsimp simp add*: *equal-on-def*)
using *root-ptr-valid-def stack-free-def valid-root-footprint-def* **by** *fastforce*

lemma *refines-L2-guard-right-unconditional*:
assumes *refines f g s t Q*
shows *refines f* (*L2-seq* (*L2-guard P*) $(\lambda$-. *g*$)$) *s t Q*
using *assms* **unfolding** *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right-unconditional'*:
assumes *refines f g s t Q*
shows *refines f (L2-seq (L2-guard (λ-. P)) (λ-. g)) s t Q*
using *assms* **unfolding** *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right'*:
assumes *P t ⟹ refines f g s t Q*
shows *refines f (L2-seq (L2-guard P) (λ-. g)) s t Q*
using *assms* **unfolding** *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right''*:
assumes *P t ⟹ refines f (L2-seq (L2-guard P) (λ-. g)) s t Q*
shows *refines f (L2-seq (L2-guard P) (λ-. g)) s t Q*
using *assms* **unfolding** *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right'''*:
assumes *P t ⟹ refines f (L2-seq (L2-seq (L2-guard P) (λ-. g)) X) s t Q*
shows *refines f (L2-seq (L2-seq (L2-guard P) (λ-. g)) X) s t Q*
using *assms* **unfolding** *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right''''*:
assumes *P t ⟹*
refines f
(*L2-seq*
(*L2-catch* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *X*)
Y) *s t Q*
shows *refines f*
(*L2-seq*
(*L2-catch* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *X*)
Y) *s t Q*
using *assms* **unfolding** *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind reaches-catch succeeds-catch*)

lemma *refines-L2-guard-rightE*:
assumes *P t*
assumes *refines f (L2-seq (L2-guard P) (λ-. g)) s t Q*
shows *refines f g s t Q*
using *assms* **unfolding** *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-rightE'*:
assumes *P t*

assumes *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t Q*
shows *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t Q*
using *assms* **unfolding** *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)

lemma *refines-L2-guard-right*:
(*P* ⟹ *refines f g s t Q*) ⟹ *refines f* (*L2-seq* (*L2-guard* (λ-. *P*)) (λ-. *g*)) *s t Q*
unfolding *L2-defs*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)

context *heap-state*
begin

lemma *equal-upto-heap-on-heap-update*[*equal-upto-simps*]:
fixes *p*:: *'a*::*mem-type ptr*
assumes *ptr-span p* ⊆ *A*
shows *equal-upto-heap-on A* (*hmem-upd* (*heap-update p v*) *s*) *t* = *equal-upto-heap-on A s t*
using *assms equal-upto-heap-update*
by (*smt* (*verit*, *ccfv-threshold*) *equal-upto-heap-on-def equal-upto-heap-on-refl equal-upto-heap-on-trans heap.get-upd heap.upd-cong heap.upd-same hmem-htd-upd sympD symp-equal-upto-heap-on*)

lemma *equal-upto-heap-stack-release'*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *ptr-span p* ⊆ *A*
shows *equal-upto-heap-on A* (*htd-upd* (*stack-releases* (*Suc 0*) *p*) *s*) *s*
by (*metis* (*no-types*, *lifting*) *One-nat-def add-mult-distrib add-right-cancel assms equal-upto-heap-on-zero-heap-on heap-state.equal-upto-heap-on-trans heap-state-axioms plus-1-eq-Suc sympD symp-equal-upto-heap-on times-nat.simps*(*2*) *zero-heap-on-stack-releases*)

lemma *equal-upto-heap-stack-release*[*equal-upto-simps*]:
fixes *p*:: *'a*::*mem-type ptr*
assumes *ptr-span p* ⊆ *A*
shows *equal-upto-heap-on A* (*htd-upd* (*stack-releases* (*Suc 0*) *p*) *s*) *t* = *equal-upto-heap-on A s t*
using *equal-upto-heap-stack-release'*
by (*metis assms equal-upto-heap-on-trans sympD symp-equal-upto-heap-on*)

lemma *equal-upto-heap-on-htd-upd*:
equal-upto-heap-on A s t ⟹
equal-upto A (*f* (*htd s*)) (*htd t*) ⟹
equal-upto-heap-on A (*htd-upd f s*) *t*
by (*smt* (*verit*, *best*) *equal-upto-heap-on-def hmem-htd-upd htd-hmem-upd lense.upd-compose typing.get-upd typing.lense-axioms typing.upd-cong*)

lemma *equal-upto-heap-on-hmem*: *equal-upto-heap-on A s t* $\Longrightarrow$ *equal-upto A* (*hmem s*) (*hmem t*)
by (*auto simp add*: *equal-upto-heap-on-def*)

lemma *equal-upto-heap-on-sym*: *equal-upto-heap-on A s t* = *equal-upto-heap-on A t s*
by (*metis heap-state.symp-equal-upto-heap-on heap-state-axioms sympD*)

lemma *equal-upto-heap-stack-alloc'*:
fixes *p*:: *'a*::*mem-type ptr*
shows *equal-upto-heap-on* (*ptr-span p*)
(*hmem-upd* (*heap-update p v*) (*htd-upd* (λ-. *ptr-force-type p* (*htd s*)) *s*))
s
by (*simp add*: *equal-upto-def equal-upto-heap-on-heap-update*
equal-upto-heap-on-htd-upd ptr-force-type-d sympD symp-equal-upto-heap-on)

lemma *equal-upto-heap-stack-alloc*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *length bs* = *size-of TYPE*('*a*)
shows *equal-upto-heap-on* (*ptr-span p*)
(*hmem-upd* (*heap-update-padding p v bs*) (*htd-upd* (λ-. *ptr-force-type p* (*htd s*)) *s*))
s
apply (*subst equal-upto-heap-on-sym*)
using *assms*
apply (*clarsimp simp add*: *equal-upto-def equal-upto-heap-on-def*)
by (*smt* (*verit*, *del-insts*) *CTypes.mem-type-simps*(*2*) *heap.lense-axioms heap-update-nmem-same*

heap-update-padding-def hmem-htd-upd lense.upd-cong ptr-force-type-d typing.lense-axioms)

definition
rel-alloc:: *addr set* ⇒ *addr set* ⇒ *addr set* ⇒ *'s* ⇒ *'s* ⇒ *'s* ⇒ *bool*
where
rel-alloc S M A t_0 ≡ λ*s t*.
stack-free (*htd s*) ⊆ *S* ∧
stack-free (*htd s*) ∩ *A* = {} ∧
stack-free (*htd s*) ∩ *M* = {} ∧
t = *frame A* t_0 *s*

lemma *rel-alloc-fold-frame*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$ *frame A* t_0 *s* = *t*
by (*auto simp add*: *rel-alloc-def*)

lemma *rel-alloc-modifies-antimono*: *rel-alloc S M2 A* t_0 *s t* $\Longrightarrow$ *M1* ⊆ *M2* $\Longrightarrow$ *rel-alloc S M1 A* t_0 *s t*
by (*auto simp add*: *rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint*:

rel-alloc S M A t_0 *s t* $\Longrightarrow$ *ptr-span p* $\subseteq$ *A* $\Longrightarrow$ *ptr-span p* $\cap$ *stack-free* (*htd s*) = {}
by (*auto simp add*: *rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint-trans*:
rel-alloc S M A t_0 *s'* *t* $\Longrightarrow$ *htd s'* = *htd s* $\Longrightarrow$ *ptr-span p* $\subseteq$ *A* $\Longrightarrow$ *ptr-span p* $\cap$ *stack-free* (*htd s*) = {}
by (*auto simp add*: *rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint'*:
rel-alloc S M A t_0 *s t* $\Longrightarrow$ *ptr-span p* $\subseteq$ *A* $\Longrightarrow$ *stack-free* (*htd s*) $\cap$ *ptr-span p* = {}
by (*auto simp add*: *rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint-trans'*:
rel-alloc S M A t_0 *s'* *t* $\Longrightarrow$ *htd s'* = *htd s* $\Longrightarrow$ *ptr-span p* $\subseteq$ *A* $\Longrightarrow$ *stack-free* (*htd s*) $\cap$ *ptr-span p* = {}
by (*auto simp add*: *rel-alloc-def*)

lemma *rel-alloc-disj-G-S*: *G* $\cap$ *S* = {} $\Longrightarrow$ *rel-alloc S M A* t_0 *s t* $\Longrightarrow$ *G* $\cap$ *stack-free* (*htd s*) = {}
by (*auto simp add*: *rel-alloc-def*)

lemma *rel-alloc-stack-free-disjoint-field-lvalue*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *rel-alloc S M A* t_0 *s t ptr-span p* $\subseteq$ *A*
assumes *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*T*, *n*)
assumes *export-uinfo T* = *typ-uinfo-t* (*TYPE*(*'b*))
shows {&(*p*→*f*)..+*size-of TYPE*(*'b*::*c-type*)} $\cap$ *stack-free* (*htd s*) = {}
using *rel-alloc-stack-free-disjoint field-tag-sub assms export-size-of*
by *fastforce*

lemma *rel-alloc-stack-free-disjoint-field-lvalue-trans*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *rel-alloc S M A* t_0 *s' t htd s'* = *htd s ptr-span p* $\subseteq$ *A*
assumes *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*T*, *n*)
assumes *export-uinfo T* = *typ-uinfo-t* (*TYPE*(*'b*))
shows {&(*p*→*f*)..+*size-of TYPE*(*'b*::*c-type*)} $\cap$ *stack-free* (*htd s*) = {}
using *rel-alloc-stack-free-disjoint field-tag-sub assms export-size-of*
by *fastforce*

lemma *rel-alloc-stack-free-disjoint-field-lvalue'*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *rel-alloc S M A* t_0 *s t ptr-span p* $\subseteq$ *A*
assumes *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*T*, *n*)
assumes *export-uinfo T* = *typ-uinfo-t* (*TYPE*(*'b*))
shows *stack-free* (*htd s*) $\cap$ {&(*p*→*f*)..+*size-of TYPE*(*'b*::*c-type*)} = {}
using *rel-alloc-stack-free-disjoint-field-lvalue* [*OF assms*] **by** *blast*

lemma *rel-alloc-stack-free-disjoint-field-lvalue-trans'*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *rel-alloc S M A t*$_0$ *s' t htd s' = htd s ptr-span p* ⊆ *A*
assumes *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0 = Some* (*T*, *n*)
assumes *export-uinfo T = typ-uinfo-t* (*TYPE*(*'b*))
shows *stack-free* (*htd s*) ∩ {&(*p*→*f*)..+*size-of TYPE*(*'b*::*c-type*)} = {}
using *rel-alloc-stack-free-disjoint-field-lvalue-trans* [*OF assms*] **by** *blast*

lemma *h-val-rel-alloc-disjoint*:
fixes *p*::*'a*::*mem-type ptr*
shows *rel-alloc S M A t*$_0$ *s t* ⟹ *ptr-span p* ∩ *A* = {} ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {} ⟹ *h-val* (*hmem t*) *p* = *h-val* (*hmem s*) *p*
apply (*simp add*: *rel-alloc-def h-val-frame-disjoint*)
done

definition *rel-stack*::
addr set ⇒ *addr set* ⇒ *addr set* ⇒ *'s* ⇒ *'s* ⇒ (*heap-mem* ⇒ *'b* ⇒ *'c* ⇒ *bool*) ⇒
(*'b* × *'s*) ⇒ (*'c* × *'s*) ⇒ *bool*
where
rel-stack S M A s t$_0$ *R* = (λ(*v*, *s'*) (*w*, *t'*).
R (*hmem s'*) *v w* ∧
rel-alloc S M A t$_0$ *s' t'* ∧
equal-upto (*M* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ∧
htd s' = *htd s*)

lemma *rel-stack-unchanged-typing*: *rel-alloc S M' A t*$_0$ *s t* ⟹ *rel-stack S M A s t*$_0$ *R* (*v*, *s'*) (*w*, *t'*) ⟹
equal-on S (*htd t'*) (*htd t*)
by (*auto simp add*: *rel-alloc-def rel-stack-def Diff-triv equal-on-def htd-frame inf-commute*)

lemma *rel-stack-unchanged-heap*: *rel-alloc S M' A t*$_0$ *s t* ⟹ *rel-stack S M A s t*$_0$ *R* (*v*, *s'*) (*w*, *t'*) ⟹
equal-upto (*M* ∪ *stack-free* (*htd s'*)) (*hmem t'*) (*hmem t*)
apply (*clarsimp simp add*: *rel-alloc-def rel-stack-def Diff-triv equal-on-def htd-frame inf-commute*)
by (*smt* (*verit*, *ccfv-threshold*) *equal-on-def equal-on-stack-free-preservation equal-upto-def hmem-frame*)

lemma *rel-stack-unchanged-stack-free*:
rel-alloc S M' A t$_0$ *s t* ⟹ *rel-stack S M A s t*$_0$ *R* (*v*, *s'*) (*w*, *t'*) ⟹
stack-free (*htd s'*) = *stack-free* (*htd s*)
apply (*auto simp add*: *rel-alloc-def rel-stack-def Diff-triv htd-frame inf-commute*)
done

lemma *rel-stack-unchanged-stack-free'*:
assumes *stack*: *rel-alloc S M' A t*$_0$ *s t*

assumes *rel-stack*: *rel-stack S M A s* t_0 *R* (*v*, *s'*) (*w*, *t'*)
shows *stack-free* (*htd t'*) = *stack-free* (*htd t*)
using *stack rel-stack*
apply (*clarsimp simp add*: *rel-alloc-def rel-stack-def Diff-triv htd-frame inf-commute*)
subgoal
using *rel-stack-unchanged-stack-free* [*OF stack rel-stack*] *equal-on-stack-free-preservation*
by (*auto simp add*: *frame-def stack-free-def root-ptr-valid-def valid-root-footprint-def*)
done

lemma *rel-stack-unchanged-heap'*:
assumes *alloc*: *rel-alloc S* {} *A* t_0 *s t*
assumes *stack*: *rel-stack S M A s* t_0 *R* (*v*, *s'*) (*w*, *t'*)
shows *equal-upto* (*M* ∪ *stack-free* (*htd t'*)) (*hmem t'*) (*hmem t*)
apply (*rule equal-upto-mono* [*OF - rel-stack-unchanged-heap* [*OF alloc stack*]])
using *stack-free-htd-frame stack*
apply (*auto simp add*: *rel-stack-def rel-alloc-def subset-iff*)
done

lemma *rel-stack-Exn*[*simp*]:
rel-stack S M A s t_0 (*rel-xval-stack L R*) (*Exn v*, *s'*) (*Exn w*, *t'*) = *rel-stack S M A s* t_0 *L* (*v*, *s'*) (*w*, *t'*)
by (*simp add*: *rel-stack-def*)
lemma *rel-stack-Exn-Result*[*simp*]:
rel-stack S M A s t_0 (*rel-xval-stack L R*) (*Exn v*, *s'*) (*Result w*, *t'*) = *False*
by (*simp add*: *rel-stack-def*)
lemma *rel-stack-Result*[*simp*]:
rel-stack S M A s t_0 (*rel-xval-stack L R*) (*Result v*, *s'*) (*Result w*, *t'*) = *rel-stack S M A s* t_0 *R* (*v*, *s'*) (*w*, *t'*)
by (*simp add*: *rel-stack-def*)
lemma *rel-zero-stack-Result-Exn*[*simp*]:
rel-stack S M A s t_0 (*rel-xval-stack L R*) (*Result v*, *s'*) (*Exn w*, *t'*) = *False*
by (*simp add*: *rel-stack-def*)

lemma *admissible-fail*: *ccpo.admissible Inf* (λ*x y*. *y* ≤ *x*) (λ*A*. ¬ *succeeds A t*)
apply (*rule ccpo.admissibleI*)
apply (*clarsimp simp add*: *ccpo.admissible-def chain-def succeeds-def reaches-def split*: *prod.splits xval-splits*)
apply *transfer*
by (*auto simp add*: *Inf-post-state-def top-post-state-def*)

lemma *fun-lub-lem*: (λ(*A*::('*f* ⇒ (('*d*, '*e*) *exception-or-result* × '*f*) *post-state*) *set*) *x*::'*f*.
⋂*f*::'*f* ⇒ (('*d*, '*e*) *exception-or-result* × '*f*) *post-state*∈*A*. *f x*) = *fun-lub* (*Inf*)
apply (*clarsimp simp add*: *fun-lub-def* [*abs-def*])
by (*simp add*: *image-def*)

lemma *fun-ord-lem*:

```
   (λ(a::'f ⇒ (('d, 'e) exception-or-result × 'f) post-state)
           b::'f ⇒ (('d, 'e) exception-or-result × 'f) post-state. ∀ x::'f. b x ≤ a x)
= fun-ord (≥)
  apply (simp add: fun-ord-def [abs-def])
  done



lemma admissible-refines-funp:
  assumes ∗: funp R
  shows ccpo.admissible Inf (≥) (λA. refines C A s t R)
    apply (rule ccpo.admissibleI)

  apply (clarsimp simp add: ccpo.admissible-def chain-def
        refines-def-old reaches-def succeeds-def)
  apply (intro allI impI conjI)
  subgoal
    apply transfer
    by (auto simp add: Inf-post-state-def top-post-state-def split: if-split-asm)
  subgoal
    using ∗
    apply transfer
   apply (clarsimp simp add: funp-def fun-of-rel-def Inf-post-state-def top-post-state-def
image-def vimage-def
        split: if-split-asm)
    by (metis post-state.distinct(1) outcomes.simps(2))
  done

lemma fun-of-rel-stack:
  assumes f: ⋀h. fun-of-rel (Q h) (f h)
  shows fun-of-rel (rel-stack S M A s t0 Q) (λ(r, s). (f (hmem s) r, frame A t0
s))
  using f
  by (auto simp add: fun-of-rel-def rel-stack-def rel-alloc-def)

lemma funp-rel-stack:
  assumes funp: ⋀h. funp (Q h)
  shows funp (rel-stack S M A s t0 Q)
proof −
  from funp obtain f where f: ⋀h. fun-of-rel (Q h) (f h)
    apply (clarsimp simp add: funp-def )
    by metis
  from fun-of-rel-stack [OF f]
  have fun-of-rel (rel-stack S M A s t0 Q) (λ(r, s). (f (hmem s) r, frame A t0 s)).
  then show ?thesis
    unfolding funp-def
    by blast
qed
```

theorem *admissible-refines-rel-stack*[*corres-admissible*]:
assumes *funp*: $\bigwedge h.\ funp\ (Q\ h)$
shows *ccpo.admissible Inf* $(\geq)$ $(\lambda g.\ refines\ f\ g\ s'\ t'\ (rel\text{-}stack\ S\ M\ A\ s\ t_0\ Q))$
apply (*rule admissible-refines-funp*)
apply (*rule funp-rel-stack*)
apply (*rule funp*)
done

lemma *admissible-rel-stack-eq*: *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ (\lambda\text{-}.\ (=))\ h\ v\ w)$
by (*auto simp add*: *ccpo.admissible-def*)

lemma *admissible-rel-singleton-stack*:
shows *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ (rel\text{-}singleton\text{-}stack\ p)\ h\ v\ w)$
by (*auto simp add*: *ccpo.admissible-def rel-singleton-stack-def*)

lemma *admissible-rel-push*:
assumes *admiss*: $\bigwedge h\ v.$ *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ Q\ h\ v\ w)$
shows *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ (rel\text{-}push\ p\ Q)\ h\ v\ w)$
proof −
have *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w.\ (h\text{-}val\ h\ p,\ w) \in X \wedge Q\ h\ v\ w)$
proof (*rule ccpo.admissibleI*[*rule-format*])
fix C::$('c \times 'b)\ set\ set$
assume *chain*: *Complete-Partial-Order.chain* $(\lambda x\ y.\ y \subseteq x)\ C$
assume *nonempty*: $C \neq \{\}$
assume *chain-prop*: $(\bigwedge X.\ X \in C \Longrightarrow \exists w.\ (h\text{-}val\ h\ p,\ w) \in X \wedge Q\ h\ v\ w)$
show $\exists w.\ (h\text{-}val\ h\ p,\ w) \in \bigcap\ C \wedge Q\ h\ v\ w$
proof −
define C' **where** $C' = Set.filter\ (\lambda(v,\ w).\ v = h\text{-}val\ h\ p)\ `\ C$
have *subset*: $\bigcap\ C' \subseteq \bigcap\ C$
using C'*-def* **by** *fastforce*
from *chain* **have** *chain-C'*: *Complete-Partial-Order.chain* $(\lambda x\ y.\ y \subseteq x)\ C'$
using *chain-prop* C'*-def*
by (*simp add*: *chain-imageI subset-iff*)
have *fst-C'*: $\bigwedge X\ t.\ X \in C' \Longrightarrow\ t \in fst\ `\ X \Longrightarrow t = h\text{-}val\ h\ p$
using C'*-def*
by *auto*
from *nonempty chain-prop* **have** *nonempty-C'*: $C' \neq \{\}$
using C'*-def*
by *blast*
from *chain-C'* **have** *result-chain*: *Complete-Partial-Order.chain* $(\lambda x\ y.\ y \subseteq x)\ ((\lambda X.\ snd\ `\ X)\ `\ C')$
by (*simp add*: *chain-imageI image-mono*)
from *nonempty-C'* **have** *nonempty-result-chain*: $((\lambda X.\ snd\ `\ X)\ `\ C') \neq \{\}$ **by** *auto*
from *chain-prop* **have** *result-chain-prop*: $(\bigwedge X.\ X \in (`)\ snd\ `\ C' \Longrightarrow \exists w{\in}X.\ Q\ h\ v\ w)$

```
      using C'-def
      by auto (metis (mono-tags, lifting) case-prodI member-filter snd-conv)
    from ccpo.admissibleD [OF admiss [of h v] result-chain nonempty-result-chain
result-chain-prop]
    obtain w where
      w: w ∈ ⋂ ((λX. snd ` X) ` C') and Q: Q h v w
      by auto
    from w fst-C' have (h-val h p, w) ∈ ⋂ C'
      by fastforce
    with Q subset show ?thesis by blast
   qed
  qed
  then show ?thesis
   by (clarsimp simp add: rel-push-def  split-paired-Bex)
qed

lemma admissible-rel-sum-stack:
  assumes admiss-L: ⋀h v. ccpo.admissible ⋂ (λx y. y ⊆ x) (λX. ∃ w ∈ X. L h v
w)
  assumes admiss-R: ⋀h v. ccpo.admissible ⋂ (λx y. y ⊆ x) (λX. ∃ w ∈ X. R h
v w)
  shows ccpo.admissible ⋂ (λx y. y ⊆ x) (λX. ∃ w ∈ X. (rel-sum-stack L R) h v
w)
proof −
  have ccpo.admissible ⋂ (λx y. y ⊆ x) (λX. ∃ w∈X. rel-sum (L h) (R h) v w)
  proof (rule ccpo.admissibleI[rule-format])
    fix C::('c + 'e) set set
    assume chain: Complete-Partial-Order.chain (λx y. y ⊆ x) C
    assume nonempty: C ≠ {}
    assume chain-prop: ⋀X. X ∈ C ⟹ ∃ w∈X. rel-sum (L h) (R h) v w
    show ∃ w∈⋂ C. rel-sum (L h) (R h) v w
    proof (cases v)
      case (Inl l)
      define C' where C' = Set.filter (Sum-Type.isl) ` C
      have subset: ⋂ C' ⊆ ⋂ C
        using C'-def by fastforce
      from chain have chain-C': Complete-Partial-Order.chain (λx y. y ⊆ x) C'
        using chain-prop C'-def
        by (simp add: chain-imageI subset-iff)
      from nonempty chain-prop have nonempty-C': C' ≠ {}
        using C'-def
        by blast
      have prop-C': ⋀X t. X ∈ C' ⟹  t ∈ X ⟹ Sum-Type.isl t
        using C'-def
        by auto
      from Inl chain-prop  have chain-prop':
        ⋀X. X ∈ C' ⟹ ∃ w∈X. ∃ l'. w = Inl l' ∧ L h l l'
        by (fastforce simp add: rel-sum.simps C'-def)
      have ∃ w∈⋂ C. ∃ l'. w = Inl l' ∧ L h l l'
```

```
    proof −
      from chain-C' have result-chain: Complete-Partial-Order.chain (λx y. y ⊆
x) ((λX. Sum-Type.projl ‘ X) ‘ C')
        by (simp add: chain-imageI image-mono)
      from nonempty-C' have nonempty-result-chain: ((λX. Sum-Type.projl ‘ X)
‘ C') ≠ {} by auto

      from chain-prop' have result-chain-prop: ⋀X. X ∈(λX. Sum-Type.projl ‘
X) ‘ C' ⟹ ∃ w∈X. L h l w
        using image-iff by fastforce
    from ccpo.admissibleD [OF admiss-L [of h l] result-chain nonempty-result-chain
result-chain-prop]
      obtain w where
        w: w∈⋂ ((‘) projl ‘ C') and L: L h l w
        by auto
      from w prop-C' have Inl w ∈ ⋂ C' by fastforce
      with L subset show ?thesis by blast
    qed
    then show ?thesis by (simp add: Inl rel-sum.simps)
  next
    case (Inr r)
    define C' where C' = Set.filter (Not o Sum-Type.isl) ‘ C
    have subset: ⋂ C' ⊆ ⋂ C
      using C'-def by fastforce
    from chain have chain-C': Complete-Partial-Order.chain (λx y. y ⊆ x) C'
      using chain-prop C'-def
      by (simp add: chain-imageI subset-iff)
    from nonempty chain-prop have nonempty-C': C' ≠ {}
      using C'-def
      by blast
    have prop-C': ⋀X t. X ∈ C' ⟹ t ∈ X ⟹ ¬ (Sum-Type.isl t)
      using C'-def
      by auto
    from Inr chain-prop have chain-prop':
      ⋀X. X ∈ C' ⟹ ∃ w∈X. ∃ r'. w = Inr r' ∧ R h r r'
      by (fastforce simp add: rel-sum.simps C'-def)
    have ∃ w∈⋂ C. ∃ r'. w = Inr r' ∧ R h r r'
    proof −
      from chain-C' have result-chain: Complete-Partial-Order.chain (λx y. y ⊆
x) ((λX. Sum-Type.projr ‘ X) ‘ C')
        by (simp add: chain-imageI image-mono)
      from nonempty-C' have nonempty-result-chain: ((λX. Sum-Type.projr ‘ X)
‘ C') ≠ {} by auto

      from chain-prop' have result-chain-prop: ⋀X. X ∈(λX. Sum-Type.projr ‘
X) ‘ C' ⟹ ∃ w∈X. R h r w
        using image-iff by fastforce
    from ccpo.admissibleD [OF admiss-R [of h r] result-chain nonempty-result-chain
result-chain-prop]
```

```
      obtain w where
        w: w∈⋂ ((`) projr ` C') and L: R h r w
        by auto
      from w prop-C' have Inr w ∈ ⋂ C' by fastforce
      with L subset show ?thesis by blast
    qed
    then show ?thesis by (simp add: Inr rel-sum.simps)
  qed
 qed
 then show ?thesis by (simp add: rel-sum-stack-def)
qed

lemma IOcorres-refines-conv:
 assumes rel-to-prj: ⋀h. fun-of-rel (R h) (prj h)
 assumes rel-to-prjE: ⋀h. fun-of-rel (L h) (prjE h)
 shows IOcorres
        (λs. rel-alloc S M A t₀ s (frame A t₀ s) ∧ P s)
        (λs r s'. stack-free (htd s') ⊆ S ∧ stack-free (htd s') ∩ A = {} ∧ stack-free
(htd s') ∩ M1 = {} ∧
                  equal-upto (M1 ∪ stack-free (htd s')) (hmem s') (hmem s) ∧
                  htd s' = htd s ∧
                    (case r of Exn l ⇒ ∃e. l = Nonlocal e ∧ L (hmem s') e (prjE
(hmem s') e)| Result x ⇒ R (hmem s') x (prj (hmem s') x)))
        (frame A t₀)
        (λr s. prj (hmem s) r)
        (λr s. prjE (hmem s) (the-Nonlocal r))
        g f
       ⟷
       (∀ s t. rel-alloc S M A t₀ s t ⟶ P s ⟶ refines f g s t (rel-stack S M1 A
s t₀ (rel-xval-stack (rel-exit L) R)))
 apply standard
 subgoal
  apply (clarsimp simp add: IOcorres-def corresXF-post-def)
  apply (rule refinesI)
  subgoal
   using rel-alloc-fold-frame by blast


  subgoal for s t v s'
   apply (erule-tac x=s in allE)
   apply (subst (asm) (1 2) rel-alloc-def)
   apply safe
   apply (clarsimp)
   apply (erule-tac x=v in allE)
   apply (erule-tac x=s' in allE)
   subgoal
    apply (clarsimp split: xval-splits)
    subgoal for x
     apply (rule exI[where x=Exn (prjE (hmem s') x)])
```

apply (*rule exI*[**where** *x*=*frame A* t_0 *s'*])
apply (*clarsimp simp add*: *rel-stack-def rel-alloc-def fun-of-rel-def*)
done
subgoal for *x*
apply (*rule exI*[**where** *x*=*Result* (*prj* (*hmem s'*) *x*)])
apply (*rule exI*[**where** *x*=*frame A* t_0 *s'*])
apply (*clarsimp simp add*: *rel-stack-def rel-alloc-def fun-of-rel-def*)
done
done
done
done
subgoal
apply (*clarsimp simp add*: *IOcorres-def corresXF-post-def*)
subgoal for *s*
apply (*rule conjI*)
subgoal
apply *clarsimp*
subgoal for *v s'*
apply (*erule-tac x*=*s* **in** *allE*)
apply (*erule-tac x*=(*frame A* t_0 *s*) **in** *allE*)
apply (*clarsimp simp add*: *rel-alloc-def refines-def-old*)
apply (*erule-tac x*=*v* **in** *allE*)
apply (*erule-tac x*=*s'* **in** *allE*)
subgoal
apply (*cases v*)
subgoal
using *rel-to-prjE* **by** (*clarsimp simp add*: *rel-stack-def rel-alloc-def fun-of-rel-def rel-exit-def default-option-def Exn-def* [*symmetric*])
subgoal
using *rel-to-prj* **by** (*clarsimp simp add*: *rel-stack-def rel-alloc-def fun-of-rel-def default-option-def Exn-def* [*symmetric*])
done
done
done
subgoal
by (*auto simp add*: *refines-def-old*)
done
done
done

lemmas *IOcorres-to-refines* = *iffD1* [*OF IOcorres-refines-conv*, *rule-format*]
lemmas *refines-to-IOcorres* = *iffD2* [*OF IOcorres-refines-conv*, *rule-format*]

lemma *refines-rel-xval-stack-generalise-exit*:
refines f g s t (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R*)) $\Longrightarrow$ ($\bigwedge$*h v w*. *L h v w* $\Longrightarrow$ *L' h v w*) $\Longrightarrow$
refines f g s t (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L' R*))

by (*erule refines-weaken*) (*auto simp add*: *rel-stack-def rel-xval-stack-def rel-xval.simps*)

lemma *L2-gets-rel-stack'*:
assumes *R* (*hmem s*) (*e s*) (*e'* (*frame A* t_0 *s*))
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-gets e ns*) (*L2-gets e' ns*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L R*))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-gets-rel-stack*:
assumes *R* (*hmem s*) (*e s*) (*e'* (*frame A* t_0 *s*))
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-gets e ns*) (*L2-gets e' ns'*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L R*))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-gets-rel-stack-guarded*:
assumes $G \Longrightarrow$ *R* (*hmem s*) (*e s*) (*e'* (*frame A* t_0 *s*))
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-gets e ns*) (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. (*L2-gets e' ns'*))) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L R*))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def succeeds-bind reaches-bind*)

lemma *L2-gets-constant-trivial-rel-stack*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-gets* (λ-. *c*) *ns*) (*L2-gets* (λ-. *c*) *ns*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-gets-constant-rel-stack*:
assumes *R* (*hmem s*) *c c'*
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-gets* (λ-. *c*) *ns*) (*L2-gets* (λ-. *c'*) *ns*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L R*))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-throw-rel-stack*:
assumes *L* (*hmem s*) *c c'*
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-throw c ns*) (*L2-throw c' ns'*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L R*))
using *assms*

by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def Exn-def* [*symmetric*])

lemma *L2-try-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-sum-stack L R*) *R*))
shows *refines* (*L2-try f*) (*L2-try g*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
using *assms*
apply (*clarsimp simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def reaches-try*)
subgoal for s' r'
apply (*cases* r')
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
subgoal for *y*
apply (*cases y*)
subgoal by *fastforce*
subgoal by *fastforce*
done
subgoal
apply *fastforce*
done
done
done

lemma *L2-try-rel-stack-merge1*:
assumes $\bigwedge h\ v\ v'.\ R'\ h\ v\ v' \Longrightarrow\ R\ h\ v\ v'$
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-sum-stack L R'*) *R*))
shows *refines* (*L2-try f*) (*L2-try g*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
using *assms*
apply (*clarsimp simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def reaches-try*)
subgoal for s' r'
apply (*cases* r')
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
subgoal for *y*
apply (*cases y*)
subgoal by *fastforce*
subgoal by *fastforce*
done
subgoal
apply *fastforce*
done
done
done

lemma *L2-try-rel-stack-merge2*:
assumes $\bigwedge h\ v\ v'.\ R'\ h\ v\ v' \Longrightarrow R\ h\ v\ v'$

assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-sum-stack L R*) *R'*))
shows *refines* (*L2-try f*) (*L2-try g*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
using *assms*
apply (*clarsimp simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def reaches-try*)
subgoal for *s' r'*
apply (*cases r'*)
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
subgoal for *y*
apply (*cases y*)
subgoal by *fastforce*
subgoal by *fastforce*
done
subgoal
apply *fastforce*
done
done
done

lemma *L2-guard-rel-stack*:
assumes *e'* (*frame A* t_0 *s*) $\Longrightarrow$ *e s*
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-guard e*) (*L2-guard e'*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def*)

lemma *L2-modify-heap-update-rel-stack'*:
fixes *p*::'*a*:: *mem-type ptr*
assumes *R* (*heap-update p v* (*hmem s*)) () (*e'* (*frame A* t_0 *s*))
assumes *ptr-span p* ⊆ *A*
assumes *rel-alloc* $\mathcal{S}$ (*ptr-span p*) *A* t_0 *s t*
shows *refines*
(*L2-modify* (*hmem-upd* (*heap-update p v*)))
(*L2-gets e' ns*)
s t
(*rel-stack* $\mathcal{S}$ (*ptr-span p*) *A s* t_0 (*rel-xval-stack L R*))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def frame-heap-update equal-upto-heap-update*)

lemma *L2-modify-heap-update-rel-stack*:
fixes *p*::'*a*:: *mem-type ptr*
assumes *R* (*heap-update p* (*v s*) (*hmem s*)) () (*e'* (*frame A* t_0 *s*))
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *ptr-span p* ⊆ *A*
assumes *ptr-span p* ⊆ *M*

shows *refines*
(*L2-modify* (λ*s*. *hmem-upd* (*heap-update p* (*v s*)) *s*))
(*L2-gets e' ns*)
s t
(*rel-stack* $\mathcal{S}$ (*ptr-span p*) *A s* t_0 (*rel-xval-stack L R*))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def frame-heap-update equal-upto-heap-update*)

lemma *L2-modify-keep-heap-update-rel-stack*:
fixes *p*::'*a*:: *mem-type ptr*
assumes *v s* = *v'* (*frame A* t_0 *s*)
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *ptr-span p* ∩ *A* = {}
assumes *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *ptr-span p* ⊆ *M*
shows *refines*
(*L2-modify* (λ*s*. *hmem-upd* (*heap-update p* (*v s*)) *s*))
(*L2-modify* (λ*s*. *hmem-upd* (*heap-update p* (*v' s*)) *s*))
s t
(*rel-stack* $\mathcal{S}$ (*ptr-span p*) *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
using *assms*
by (*auto simp add*: *refines-def-old L2-defs rel-stack-def rel-alloc-def frame-heap-update disjoint-subset2 equal-upto-heap-update frame-heap-update-disjoint*)

lemma *L2-modify-keep-heap-update-rel-stack-guarded*:
fixes *p*::'*a*:: *mem-type ptr*
assumes *G* ⟹ *v s* = *v'* (*frame A* t_0 *s*)
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *G* ⟹ *ptr-span p* ∩ *A* = {}
assumes *G* ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *G* ⟹ *ptr-span p* ⊆ *M*
shows *refines*
(*L2-modify* (λ*s*. *hmem-upd* (*heap-update p* (*v s*)) *s*))
(*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. (*L2-modify* (λ*s*. *hmem-upd* (*heap-update p* (*v' s*)) *s*))))
s t
(*rel-stack* $\mathcal{S}$ (*ptr-span p*) *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
apply (*rule refines-L2-guard-right*)
apply (*rule L2-modify-keep-heap-update-rel-stack*)
using *assms* **by** *auto*

lemma *L2-call-rel-stack'*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *L'*: ⋀*e e' s*. *L* (*hmem s*) *e e'* ⟹ *L'* (*hmem s*) (*emb e*) (*emb' e'*)
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))

shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb′ ns′*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L′ R*))
using *assms*
apply (*clarsimp simp add*: *L2-call-def refines-def-old reaches-map-value*)
subgoal for *s′ r′*
apply (*cases r′*)
subgoal
by (*fastforce simp add*: *default-option-def Exn-def* [*symmetric*]
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps)
subgoal
by (*fastforce simp add*: *rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps*)
done
done

lemma *L2-call-rel-stack′′*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *L′*: $\bigwedge$*e e′ s. L* (*hmem s*) *e e′* $\Longrightarrow$ *rel-sum-stack L R′* (*hmem s*) (*emb e*) (*emb′ e′*)
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb′ ns′*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-sum-stack L R′*) *R*))
apply (*rule L2-call-rel-stack′* [*OF assms*(*1*) *- f*])
by (*rule L′*)

lemma *L2-call-rel-stack*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *L′*: $\bigwedge$*e e′ s. L* (*hmem s*) *e e′* $\Longrightarrow$ *L′* (*hmem s*) (*emb e*) (*emb e′*)
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb ns′*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L′ R*))
apply (*rule L2-call-rel-stack′* [*OF assms* (*1*) *- f*])
by (*rule L′*)

Currently exceptions on the function level are terminal and are propagated to the toplevel. So the result relation for *Inl* is equality.

lemma *L2-call-rel-stack-eq-InL*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*

assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (λ-. (=)) *R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb ns'*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (λ-. (=)) *R*))
apply (*rule L2-call-rel-stack'* [*OF assms*(*1*) - *f*])
by *simp*

lemma *L2-call-rel-stack-bot-InL*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (λ- - -. *False*) *R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb ns'*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (λ- - -. *False*) *R*))
apply (*rule L2-call-rel-stack'* [*OF assms*(*1*) - *f*])
by *simp*

lemma *L2-seq-rel-stack''*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R1*))
assumes *f2-g2*: $\bigwedge$*s' t' v w. R1* (*hmem s'*) *v w* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* ⟹
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ⟹
htd s' = *htd s* ⟹

refines (*f2 v*) (*g2' s' w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M2 A s'* t_0 (*rel-xval-stack L R*))
assumes *g2'-g2*: $\bigwedge$*s' t' v w. R1* (*hmem s'*) *v w* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* ⟹
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ⟹
htd s' = *htd s* ⟹
g2' s' w = *g2 w*

assumes *M1*: *M1* ⊆ *M*
assumes *M2*: *M2* ⊆ *M*
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ (*M1* ∪ *M2*) *A s* t_0 (*rel-xval-stack L R*))
proof −
from *f2-g2 g2'-g2*
have *f2-g2*: $\bigwedge$*s' t' v w. R1* (*hmem s'*) *v w* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* ⟹
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ⟹
htd s' = *htd s* ⟹

refines (*f2 v*) (*g2 w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M2 A s'* t_0 (*rel-xval-stack L R*))
by *metis*
show *?thesis*

apply (*rule refines-L2-seq*)
apply (*rule f1-g1*)
apply (*simp-all*)
subgoal
using *s-t M1 M2*
apply (*clarsimp simp add*: *rel-stack-def*, *intro conjI*)
subgoal
by (*smt* (*verit*, *best*) *Un-commute disjoint-subset2 disjoint-union-distrib equal-on-stack-free-preservation rel-alloc-def sup.coboundedI2*)
subgoal
using *equal-upto-mono*
by (*metis inf-sup-ord*(*3*) *order-le-less sup-mono*)
done
apply (*rule refines-mono* [*OF - f2-g2*])
subgoal
by (*auto simp add*: *rel-stack-def rel-alloc-def equal-upto-def*)
subgoal
by (*auto simp add*: *rel-stack-def rel-alloc-def*)
subgoal
using *M1 M2 s-t*
by (*auto simp add*: *rel-stack-def rel-alloc-def*)
subgoal
by (*auto simp add*: *rel-stack-def*)
subgoal
by (*auto simp add*: *rel-stack-def*)
done
qed

lemma *L2-seq-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R1*))
assumes *f2-g2*: $\bigwedge s'\ t'\ v\ w.$ *R1* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd s' = *htd s* $\Longrightarrow$

refines (*f2 v*) (*g2' w s'*) *s' t'* (*rel-stack* $\mathcal{S}$ *M2 A s'* t_0 (*rel-xval-stack L R*))
assumes *g2'-g2*: $\bigwedge s'\ t'\ v\ w.$ *R1* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd s' = *htd s* $\Longrightarrow$
g2' w s' = *g2 w*

assumes *M1*: *M1* $\subseteq$ *M*
assumes *M2*: *M2* $\subseteq$ *M*
assumes *M'*: *M'* = *M1* $\cup$ *M2*
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))
proof −
from *f2-g2 g2'-g2*
have *f2-g2*: $\bigwedge s'\ t'\ v\ w.$ *R1* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$

```
                equal-upto (M1 ∪ stack-free (htd s')) (hmem s') (hmem s) ⟹
                htd s' = htd s ⟹

              refines (f2 v) (g2 w) s' t' (rel-stack 𝒮 M2 A s' t₀ (rel-xval-stack L R))
    by metis
  show ?thesis
  apply (rule refines-L2-seq)
     apply (rule f1-g1)
    apply (simp-all)
  subgoal
    using s-t M1 M2 M'
    apply (clarsimp simp add: rel-stack-def, intro conjI)
    subgoal
      by (metis (no-types, lifting) Int-Un-distrib rel-alloc-def sup.absorb-iff1)

    subgoal
      using equal-upto-mono
      by (metis inf-sup-ord(3) order-le-less sup-mono)
    done
  apply (rule refines-mono [OF - f2-g2 ])
  subgoal
    apply (clarsimp simp add: rel-stack-def rel-alloc-def, intro conjI)
   subgoal by auto (metis M1 M2 M' UnE disjoint-iff equal-on-stack-free-preservation)
   subgoal by (smt (verit, best) M1 M2 M' Un-iff equal-on-stack-free-preservation
equal-upto-def)
    done
  subgoal
    by (auto simp add: rel-stack-def rel-alloc-def)
  subgoal
    using M1 M2 s-t
    apply (auto simp add: rel-stack-def rel-alloc-def)
    done
  subgoal
    by (auto simp add: rel-stack-def)
  subgoal
    by (auto simp add: rel-stack-def)
  done
qed

lemma L2-seq-rel-stack-g2-normalised:
  assumes s-t: rel-alloc 𝒮 M A t₀ s t
  assumes f1-g1: refines f1 g1 s t (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack L R1))
  assumes f2-g2: ⋀s' t' v w. R1 (hmem s') v w ⟹ rel-alloc 𝒮 M A t₀ s' t' ⟹
                equal-upto (M1 ∪ stack-free (htd s')) (hmem s') (hmem s) ⟹
                htd s' = htd s ⟹
              refines (f2 v) (g2 w) s' t' (rel-stack 𝒮 M2 A s' t₀ (rel-xval-stack L R))
  assumes M1: M1 ⊆ M
  assumes M2: M2 ⊆ M
  assumes M': M' = M1 ∪ M2
```

shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))
by (*rule L2-seq-rel-stack* [*OF s-t f1-g1 f2-g2 - M1 M2 M'*]) *auto*

lemma *L2-seq-rel-stack'*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R1*))
assumes *f2-g2*: $\bigwedge s'\ t'\ v\ w.$ *R1* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
refines (*f2 v*) (*g2 w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M2 A s'* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* $\subseteq$ *M*
assumes *M2*: *M2* $\subseteq$ *M*
assumes *M'*: *M'* = *M1* $\cup$ *M2*
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))
proof −
show *?thesis*
apply (*rule refines-L2-seq*)
apply (*rule f1-g1*)
apply (*simp-all*)
subgoal
using *s-t M1 M2 M'*
apply (*clarsimp simp add*: *rel-stack-def*, *intro conjI*)
subgoal
by (*metis* (*no-types*, *lifting*) *Int-Un-distrib rel-alloc-def sup.absorb-iff1*)

subgoal
using *equal-upto-mono*
by (*metis inf-sup-ord*(*3*) *order-le-less sup-mono*)
done
apply (*rule refines-mono* [*OF - f2-g2*])
subgoal
apply (*clarsimp simp add*: *rel-stack-def rel-alloc-def*, *intro conjI*)
subgoal by *auto* (*metis M1 M2 M' UnE disjoint-iff equal-on-stack-free-preservation*)
subgoal by (*smt* (*verit*, *best*) *M1 M2 M' Un-iff equal-on-stack-free-preservation equal-upto-def*)
done
subgoal
by (*auto simp add*: *rel-stack-def rel-alloc-def*)
subgoal
using *M1 M2 s-t*
apply (*auto simp add*: *rel-stack-def rel-alloc-def*)
done
done
qed

lemma *frame-raw-frame-union*:
assumes *disj-A-B*: *A* $\cap$ *B* = {}
assumes *sf*: *stack-free* (*htd s*) $\cap$ *B* = {}

shows *frame* $(A \cup B)$ (*raw-frame* $B\ s\ t_0$) s = *frame* $A\ t_0\ s$
apply (*clarsimp simp add*: *frame-def raw-frame-def*)
using *override-on-frame-raw-frame-lemma1* [*OF disj-A-B sf*]
override-on-frame-raw-frame-lemma2 [*OF disj-A-B sf*]
by (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-htd-upd typing.upd-cong*)

lemma *refines-narrow-frame'*:
assumes *disj-B-M*: $B \cap M = \{\}$
assumes *disj-A-B*: $A \cap B = \{\}$
assumes *spec*: $\bigwedge t_0\ s\ t.$ *rel-alloc* $\mathcal{S}$ *M1* $(A \cup B)\ t_0\ s\ t \Longrightarrow$
refines f g s t (*rel-stack* $\mathcal{S}$ M $(A \cup B)\ s\ t_0$ Q)
assumes *sf*: *stack-free* (*htd s*) $\cap B = \{\}$
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M1* $A\ t_0\ s\ t$
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ $M\ A\ s\ t_0\ Q$)
proof −
from *rel-alloc* **have** *t*: t = *frame* $A\ t_0\ s$ **by** (*auto simp add*: *rel-alloc-def*)
define t_0' **where** t_0' = *raw-frame* $B\ s\ t_0$

have t = *frame* $(A \cup B)\ t_0'\ s$
using *frame-raw-frame-union* [*OF disj-A-B sf*]
by (*simp add*: t_0'*-def t*)

with *rel-alloc sf* **have** *rel-alloc'*: *rel-alloc* $\mathcal{S}$ *M1* $(A \cup B)\ t_0'\ s\ t$
by (*auto simp add*: t_0'*-def rel-alloc-def*)
from *spec* [*OF rel-alloc'*] **have** *refines'*: *refines f g s t* (*rel-stack* $\mathcal{S}$ M $(A \cup B)\ s\ t_0'\ Q$) **.**
show *?thesis*
proof (*rule refinesI*)
show *succeeds g t* $\Longrightarrow$ *succeeds f s* **using** *refines'*
by (*auto simp add*: *refines-def-old*)
next
fix $v\ s'$
assume *succeeds*: *succeeds g t succeeds f s*
assume *result*: *reaches f s v s'*
show $\exists w\ t'.$ *reaches g t w t'* $\wedge$ *rel-stack* $\mathcal{S}$ $M\ A\ s\ t_0\ Q$ $(v,\ s')$ $(w,\ t')$
proof −
from *succeeds result refines'* **obtain** $w\ t'$
where
res: *reaches g t w t'* **and**
stack: *rel-stack* $\mathcal{S}$ M $(A \cup B)\ s\ t_0'\ Q$ $(v,\ s')$ $(w,\ t')$
by (*fastforce simp add*: *refines-def-old*)
from *stack* **obtain**
mem-eq: *equal-upto* $(M \cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) **and**
htd-eq1: *equal-upto* M (*htd s'*) (*htd s*) **and**
htd-eq2: *equal-on* $\mathcal{S}$ (*htd s'*) (*htd s*)
by (*auto simp add*: *rel-stack-def*)

from *stack*
have *sf-eq*: *stack-free* (*htd* s') = *stack-free* (*htd* s)
using *disj-A-B* **by** (*auto simp add*: *rel-stack-def*)

have *htd-eq*:
override-on (*htd* s') (*override-on* (*htd* t_0) (*htd* s) B) ($A \cup B$ − *stack-free* (*htd* s)) =
override-on (*htd* s') (*htd* t_0) (A − *stack-free* (*htd* s))
using *disj-A-B disj-B-M sf sf-eq htd-eq1*
apply (*clarsimp simp add*: *override-on-def fun-eq-iff*)
by (*auto simp add*: *equal-upto-def orthD1*)

have *hmem-eq*:
override-on (*hmem* s') (*override-on* (*hmem* t_0) (*hmem* s) B) ($A \cup B \cup$ *stack-free* (*htd* s)) =
override-on (*hmem* s') (*hmem* t_0) ($A \cup$ *stack-free* (*htd* s))
using *disj-A-B disj-B-M sf sf-eq mem-eq*
by (*auto simp add*: *override-on-def fun-eq-iff disjoint-iff equal-upto-def*)

from *stack* **have** *rel-stack'*: *rel-stack* $\mathcal{S}$ M A s t_0 Q (v, s') (w, t')
apply (*simp add*: t_0'-*def*)
apply (*simp add*: *rel-stack-def rel-alloc-def*)
apply (*clarsimp simp add*: *frame-def raw-frame-def sf-eq*)
using *htd-eq mem-eq*
by *auto* (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-eq hmem-htd-upd typing.upd-cong*)

then show *?thesis* **using** *res*
by *auto*
qed
qed
qed

lemma *refines-narrow-frame*:
assumes *subset*: $B \subseteq A$
assumes *disj-B-M*: $B \cap M = \{\}$
assumes *spec*: $\bigwedge t_0\ s\ t$. *rel-alloc* $\mathcal{S}$ *M1* A t_0 s $t \Longrightarrow$ *refines* f g s t (*rel-stack* $\mathcal{S}$ M A s t_0 Q)
assumes *sf*: *stack-free* (*htd* s) $\cap$ $B = \{\}$
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M1* (A − B) t_0 s t
shows *refines* f g s t (*rel-stack* $\mathcal{S}$ M (A − B) s t_0 Q)
proof −
from *subset* **obtain** *A1* **where** *A*: $A = A1 \cup B$ **and** *disj*: $A1 \cap B = \{\}$ **and** *A1*: $A1 = A - B$
by (*metis Diff-Diff-Int Diff-disjoint Un-Diff-Int inter-sub*)
from *refines-narrow-frame'*[*OF disj-B-M disj spec* [*simplified A*] *sf*] *rel-alloc*
show *?thesis*

by (*auto simp add*: *A1*)
qed

lemma *refines-widen-modifies*:
assumes *rel-alloc S M A* t_0 *s t*
assumes *refines f g s t* (*rel-stack S M' A s* t_0 *Q*)
assumes $M' \subseteq M$
shows *refines f g s t* (*rel-stack S M A s* t_0 *Q*)
using *assms*
apply (*clarsimp simp add*: *refines-def-old rel-stack-def rel-alloc-def split*: *prod.splits xval-splits*)
by (*smt* (*verit*) *UnI2 Un-assoc equal-on-stack-free-preservation equal-upto-def sup.absorb-iff1*)

lemma *refines-widen-modifies'*:
assumes *rel-alloc S M A* t_0 *s t* $\Longrightarrow$ *refines f g s t* (*rel-stack S M' A s* t_0 *Q*)
assumes $M' \subseteq M$
assumes *rel-alloc S M A* t_0 *s t*
shows *refines f g s t* (*rel-stack S M A s* t_0 *Q*)
using *refines-widen-modifies assms* **by** *blast*

lemma *refines-widen-modifies''*:
assumes *rel-alloc S M A* t_0 *s t*
assumes *refines f g s t* (*rel-stack S M1 A s* t_0 *Q*)
assumes *M1* $\subseteq$ *M2*
assumes *M2* $\subseteq$ *M*
shows *refines f g s t* (*rel-stack S M2 A s* t_0 *Q*)
using *refines-widen-modifies assms rel-alloc-modifies-antimono*
by *blast*

lemma *refines-widen-modifies-weaken*:
assumes *alloc*: *rel-alloc S M A* t_0 *s t*
assumes *f*: *refines f g s t* (*rel-stack S M1 A s* t_0 *Q'*)
assumes *M1-M2*: *M1* $\subseteq$ *M2*
assumes *M2-M*: *M2* $\subseteq$ *M*
assumes *weaken*: $\bigwedge h\ r\ r'.\ Q'\ h\ r\ r' \Longrightarrow Q\ h\ r\ r'$
shows *refines f g s t* (*rel-stack S M2 A s* t_0 *Q*)
proof −
have *refines f g s t* (*rel-stack S M1 A s* t_0 *Q*)
apply (*rule refines-weaken* [*OF f*])
using *weaken* **by** (*auto simp add*: *rel-stack-def*)
with *alloc M1-M2 M2-M refines-widen-modifies rel-alloc-modifies-antimono refines-weaken*
show *?thesis*
by *blast*
qed

lemma *L2-guarded-rel-stack*:

assumes *e*: g' (*frame A* t_0 *s*) $\Longrightarrow$ *g s*
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *X-X'*: *g s* $\Longrightarrow$ g' *t* $\Longrightarrow$ *refines X X' s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))
assumes *M*: $M' \subseteq M$
shows *refines* (*L2-guarded g X*) (*L2-guarded* g' *X'*) *s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))
apply (*rule refines-L2-guarded*)
subgoal using *e rel-alloc*
apply (*auto simp add*: *rel-alloc-def*)
done
subgoal
apply (*rule X-X'*)
apply (*simp-all add*: *M*)
done
done

lemma *L2-condition-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *c-c'*: *c s* = c' (*frame A* t_0 *s*)
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *f2-g2*: *refines f2 g2 s t* (*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* $\subseteq$ *M*
assumes *M2*: *M2* $\subseteq$ *M*
assumes *M'*: *M'* = *M1* $\cup$ *M2*
shows *refines* (*L2-condition c f1 f2*) (*L2-condition* c' *g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))
unfolding *L2-condition-def*
apply (*rule refines-condition*)
subgoal using *s-t c-c'* **by** (*auto simp add*: *rel-alloc-def*)
subgoal apply (*rule refines-widen-modifies* [*OF - f1-g1*]) **using** *s-t M1 M2 M' rel-alloc-modifies-antimono* **by** *auto*
subgoal apply (*rule refines-widen-modifies* [*OF - f2-g2*]) **using** *s-t M1 M2 M' rel-alloc-modifies-antimono* **by** *auto*
done

lemma *L2-while-rel-stack''*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i'*
assumes *refines-condition*: $\bigwedge$*s v v'*. *R* (*hmem s*) *v v'* $\Longrightarrow$ c' *v'* (*frame A* t_0 *s*) = *c v s*
assumes *bdy*: $\bigwedge$*s t v v'*. *R* (*hmem s*) *v v'* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M1 A* t_0 *s t* $\Longrightarrow$ *c v s* $\Longrightarrow$ c' *v' t* $\Longrightarrow$
refines (*f v*) (*g v'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* $\subseteq$ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while* c' *g i' ns'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
unfolding *L2-while-def*

apply (*rule refines-whileLoop-exn*)
subgoal by (*auto simp add*: *rel-stack-def refines-condition rel-alloc-def*)
subgoal for *v s′ w t′*
apply *clarsimp*
apply (*rule refines-mono* [*OF - bdy*])
subgoal
apply (*clarsimp simp add*: *rel-stack-def*)
subgoal by (*metis equal-upto-trans*)
done
apply (*auto simp add*: *rel-stack-def*)
done
apply (*auto simp add*: *s-t R-i rel-stack-def rel-alloc-modifies-antimono* [*OF - M1*])
done

lemma *L2-while-rel-stack‴*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i′*
assumes *refines-condition*: $\bigwedge$*s v v′*. *R* (*hmem s*) *v v′* $\Longrightarrow$ *c′ v′* (*frame A* t_0 *s*) = *c v s*
assumes *bdy*: $\bigwedge$*s t v v′*. *R* (*hmem s*) *v v′* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$ *c v s* $\Longrightarrow$ *c′ v′ t* $\Longrightarrow$
refines (*f v*) (*g v′*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* $\subseteq$ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c′ g i′ ns′*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
unfolding *L2-while-def*
apply (*rule refines-whileLoop-exn*)
subgoal by (*auto simp add*: *rel-stack-def refines-condition rel-alloc-def*)
subgoal for *v s′ w t′*
apply *clarsimp*
apply (*rule refines-mono* [*OF - bdy*])
apply (*clarsimp simp add*: *rel-stack-def*)
apply (*metis equal-upto-trans*)
using *s-t* **by** (*auto simp add*: *rel-stack-def rel-alloc-def*)
apply (*auto simp add*: *s-t R-i rel-stack-def rel-alloc-modifies-antimono* [*OF - M1*])
done

lemma *L2-while-rel-stack′*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *refines-condition*: $\bigwedge$*s v v′*. *R* (*hmem s*) *v v′* $\Longrightarrow$ *c′ v′* (*frame A* t_0 *s*) = *c v s*
assumes *R-i*: *R* (*hmem s*) *i i′*
assumes *bdy*: $\bigwedge$*s t v v′*. *R* (*hmem s*) *v v′* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines (*f v*) (*g v′*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* $\subseteq$ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c′ g i′ ns′*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))

unfolding *L2-while-def*
apply (*rule refines-whileLoop-exn*)
subgoal by (*auto simp add*: *rel-stack-def refines-condition rel-alloc-def*)
subgoal for *v s′ w t′*
apply *clarsimp*
apply (*rule refines-mono* [*OF - bdy*])
apply (*clarsimp simp add*: *rel-stack-def*)
apply (*metis equal-upto-trans*)
subgoal by (*auto simp add*: *rel-stack-def*)
subgoal using *s-t* **by** (*simp add*: *rel-stack-def*) (*metis rel-alloc-def*)
done
apply (*auto simp add*: *s-t R-i rel-stack-def rel-alloc-modifies-antimono* [*OF - M1*])
done

lemma *L2-while-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i′*
assumes *bdy*: $\bigwedge$*s′ t′ v w*. *R* (*hmem s′*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s′ t′* $\Longrightarrow$ *c v s′* $\Longrightarrow$ *c′ w t′* $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s′*)) (*hmem s′*) (*hmem s*) $\Longrightarrow$
htd s′ = *htd s* $\Longrightarrow$
refines (*f v*) (*g′ w s′*) *s′ t′* (*rel-stack* $\mathcal{S}$ *M1 A s′* t_0 (*rel-xval-stack L R*))
assumes *refines-condition*: $\bigwedge$*s′ t′ v w*. *R* (*hmem s′*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s′ t′* $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s′*)) (*hmem s′*) (*hmem s*) $\Longrightarrow$ *htd s′* = *htd s* $\Longrightarrow$
c′ w t′ = *c v s′*
assumes *g′-g*: $\bigwedge$*s′ t′ v w*. *R* (*hmem s′*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s′ t′* $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s′*)) (*hmem s′*) (*hmem s*) $\Longrightarrow$
htd s′ = *htd s* $\Longrightarrow$
g′ w s′ = *g w*
assumes *M1*: *M1* $\subseteq$ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c′ g i′ ns′*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
unfolding *L2-while-def*
apply (*rule refines-mono* [**where** *R* = λ(*r*, *s′*) (*w*, *t′*).
equal-upto (*M1* $\cup$ *stack-free* (*htd s′*)) (*hmem s′*) (*hmem s*) $\wedge$
htd s′ = *htd s* $\wedge$
(*rel-stack* $\mathcal{S}$ *M1 A s′* t_0 (*rel-xval-stack L R*)) (*r*, *s′*) (*w*, *t′*)])
subgoal
by (*auto simp add*: *rel-stack-def*)
apply (*rule refines-whileLoop-exn*)
subgoal using *s-t* **by** (*auto simp add*: *rel-stack-def refines-condition rel-alloc-def*)
subgoal for *v s′ w t′*
apply *clarsimp*
apply (*rule refines-weaken* [**where** *R*=(*rel-stack* $\mathcal{S}$ *M1 A s′* t_0 (*rel-xval-stack L R*))])
apply (*subst g′-g* [*of s′ v w t′*, *symmetric*])

apply (*simp add*: *rel-stack-def*)

apply (*simp add*: *rel-stack-def*)
using *rel-alloc-def s-t*
apply *force*
apply *simp*
apply *simp*
apply (*rule bdy*)
apply (*simp add*: *rel-stack-def*)
apply (*simp add*: *rel-stack-def*)
using *s-t* **apply** (*metis equal-on-stack-free-preservation rel-alloc-def*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply (*clarsimp simp add*: *rel-stack-def*)
apply (*metis equal-upto-trans*)
done
apply (*smt* (*verit*) *L2-split-fixup-3 M1 R-i equal-upto-refl heap-state.rel-stack-def heap-state-axioms rel-alloc-modifies-antimono rel-xval-stack-simps*(*2*) *s-t*)
apply (*auto simp add*: *s-t R-i rel-stack-def rel-alloc-modifies-antimono* [*OF* - *M1*])
done

lemma *L2-while-rel-stack-g-normalised*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i'*
assumes *bdy*: $\bigwedge s'\ t'\ v\ w.$ *R* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$ *c v s'* $\Longrightarrow$ *c' w t'* $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd s' = *htd s* $\Longrightarrow$
refines (*f v*) (*g w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M1 A s'* t_0 (*rel-xval-stack L R*))
assumes *refines-condition*: $\bigwedge s'\ t'\ \ v\ w.$ *R* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$ *htd s'* = *htd s* $\Longrightarrow$
c' w t' = *c v s'*
assumes *M1*: *M1* $\subseteq$ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c' g i' ns'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
apply (*rule L2-while-rel-stack* [*OF s-t R-i bdy refines-condition* - *M1*])
by *auto*

lemma *L2-while-rel-stack-g-normalised-guarded*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*

assumes *R-i*: $R\ (hmem\ s)\ i\ i'$
assumes *bdy*: $\bigwedge s'\ t'\ v\ w.\ R\ (hmem\ s')\ v\ w \Longrightarrow rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s'\ t' \Longrightarrow c\ v\ s' \Longrightarrow c'\ w\ t' \Longrightarrow$
$equal\text{-}upto\ (M1 \cup stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \Longrightarrow$
$htd\ s' = htd\ s \Longrightarrow$
$refines\ (f\ v)\ (g\ w)\ s'\ t'\ (rel\text{-}stack\ \mathcal{S}\ M1\ A\ s'\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))$
assumes *refines-condition*: $\bigwedge s'\ t'\ \ v\ w.\ R\ (hmem\ s')\ v\ w \Longrightarrow rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s'\ t' \Longrightarrow$
$equal\text{-}upto\ (M1 \cup stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \Longrightarrow htd\ s' = htd\ s \Longrightarrow G'\ w\ t' \Longrightarrow$
$c'\ w\ t' = c\ v\ s'$
assumes *M1*: $M1 \subseteq M$
assumes *G-G'*: $G\ t = G'\ i'\ t$
shows *refines* $(L2\text{-}while\ c\ f\ i\ ns)$
$(L2\text{-}seq\ (L2\text{-}guard\ G)$
$(\lambda\text{-}.\ (L2\text{-}while\ c'\ (\lambda v.\ L2\text{-}seq\ (g\ v)\ (\lambda res.\ L2\text{-}seq\ (L2\text{-}guard\ (G'\ res))\ (\lambda\text{-}.\ L2\text{-}gets\ (\lambda\text{-}.\ res)\ ns')))\ i'\ ns'))\ )\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ M1\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))$
apply (*rule refines-mono* [**where** $R = \lambda(r,\ s')\ (w,\ t').$
$equal\text{-}upto\ (M1 \cup stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \wedge$
$htd\ s' = htd\ s \wedge$
$(rel\text{-}stack\ \mathcal{S}\ M1\ A\ s'\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))\ (r,\ s')\ (w,\ t')$])
subgoal
by (*auto simp add*: *rel-stack-def*)
unfolding *L2-defs gets-return*
apply (*rule refines-whileLoop-guard-right*)
subgoal using *s-t* **by** (*auto simp add*: *rel-stack-def refines-condition rel-alloc-def*)
subgoal for $v\ s'\ w\ t'$
apply *clarsimp*
apply (*rule refines-weaken* [**where** *R*=(*rel-stack* $\mathcal{S}$ *M1 A* $s'\ t_0$ (*rel-xval-stack L R*))])
apply (*rule bdy*)
subgoal by (*auto simp add*: *rel-stack-def*)
subgoal using *rel-alloc-def s-t* **by** (*auto simp add*: *rel-stack-def*)
subgoal by *simp*
subgoal by *simp*
subgoal using *s-t* **by** *simp*
subgoal by *simp*
subgoal
apply (*clarsimp simp add*: *rel-stack-def*)
apply (*metis equal-upto-trans*)
done
done
subgoal by (*auto simp add*: *rel-stack-def*)
subgoal using *R-i s-t* **by** (*auto simp add*: *rel-stack-def rel-alloc-modifies-antimono* [*OF - M1*])
subgoal using *G-G'* **by** *simp*
done

lemma *L2-unknown-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-unknown ns*) (*L2-unknown ns*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
using *s-t*
by (*auto simp add*: *L2-unknown-def refines-def-old rel-stack-def rel-alloc-modifies-antimono*)

lemma *L2-fail-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* (*L2-fail*) (*L2-fail*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L R*))
using *s-t*
by (*auto simp add*: *L2-fail-def refines-def-old rel-stack-def rel-alloc-modifies-antimono*)

lemma *L2-undefined-function-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines* ((*L2-guard* (λ*s*. *UNDEFINED-FUNCTION*))) *L2-fail s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L R*))
using *s-t L2-fail-rel-stack*
by (*auto simp add*: *L2-guard-false UNDEFINED-FUNCTION-def*)

lemma *L2-skip-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *refines L2-skip L2-skip s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
using *s-t*
by (*auto simp add*: *L2-gets-def refines-def-old rel-stack-def rel-alloc-modifies-antimono*)

lemma *L2-spec-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes $\bigwedge t'$. (*frame A* t_0 *s*, t') $\in r' \Longrightarrow \exists\, s'$. (*s*, s') $\in r$
assumes $\bigwedge s'$. (*s*, s') $\in r \Longrightarrow$ (*frame A* t_0 *s*, *frame A* t_0 s') $\in r'$
assumes $\bigwedge s'$. (*s*, s') $\in r \Longrightarrow$ *equal-upto* (*M* ∪ *stack-free* (*htd* s')) (*hmem* s') (*hmem s*)
assumes $\bigwedge s'$. (*s*, s') $\in r \Longrightarrow$ *htd* s' = *htd s*
assumes $\bigwedge s'$. (*s*, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') $\subseteq \mathcal{S}$
assumes $\bigwedge s'$. (*s*, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') ∩ *A* = {}
assumes $\bigwedge s'$. (*s*, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') ∩ *M* = {}
shows *refines* (*L2-spec r*) (*L2-spec r'*) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L* (λ-. (=))))
using *assms*
by (*auto simp add*: *L2-spec-def refines-def-old rel-stack-def rel-alloc-def succeeds-bind reaches-bind*)

lemma *L2-spec-rel-stack'*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*

assumes $\bigwedge t'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (*frame* A t_0 s, t') $\in r' \Longrightarrow \exists s'$. ($s$, s') $\in r$
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ (*frame* A t_0 s, *frame* A t_0 s') $\in r'$
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *equal-upto* ($M \cup$ *stack-free* (*htd* s')) (*hmem* s') (*hmem* s)
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *htd* s' = *htd* s
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') $\subseteq \mathcal{S}$
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') $\cap$ A = {}
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') $\cap$ M = {}
shows *refines* (*L2-spec* r) (*L2-spec* r') s t (*rel-stack* $\mathcal{S}$ M A s t_0 (*rel-xval-stack* L (λ-. (=))))
using *s-t assms*(2−8) [*OF s-t*] **by** (*rule L2-spec-rel-stack*)

lemma *L2-spec-rel-stack-same*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ M A t_0 s t
assumes $\bigwedge t'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (*frame* A t_0 s, t') $\in r \Longrightarrow \exists s'$. ($s$, s') $\in r$
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ (*frame* A t_0 s, *frame* A t_0 s') $\in r$
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *equal-upto* ($M \cup$ *stack-free* (*htd* s')) (*hmem* s') (*hmem* s)
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *htd* s' = *htd* s
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') $\subseteq \mathcal{S}$
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') $\cap$ A = {}
assumes $\bigwedge s'$. *rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ (s, s') $\in r \Longrightarrow$ *stack-free* (*htd* s') $\cap$ M = {}
shows *refines* (*L2-spec* r) (*L2-spec* r) s t (*rel-stack* $\mathcal{S}$ M A s t_0 (*rel-xval-stack* L (λ-. (=))))
using *assms*
by (*rule L2-spec-rel-stack'*)

lemma *L2-spec-rel-stack-heap-agnostic*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ M A t_0 s t
assumes *hmem-unchanged*: $\bigwedge s\ s'$. (s, s') $\in r \Longrightarrow$ *hmem* s' = *hmem* s
assumes *htd-unchanged*: $\bigwedge s\ s'$. (s, s') $\in r \Longrightarrow$ *htd* s' = *htd* s
assumes *heap-irrelevant*: $\bigwedge s\ s'\ f\ g$. (s, s') $\in r \Longrightarrow$ (*hmem-upd* g (*htd-upd* f s), *hmem-upd* g (*htd-upd* f s')) $\in r$
shows *refines* (*L2-spec* r) (*L2-spec* r) s t (*rel-stack* $\mathcal{S}$ {} A s t_0 (*rel-xval-stack* L (λ-. (=))))
proof −
have *refines* (*L2-spec* r) (*L2-spec* r) s t (*rel-stack* $\mathcal{S}$ M A s t_0 (*rel-xval-stack* L (λ-. (=))))
apply (*rule L2-spec-rel-stack-same* [*OF s-t*])
subgoal premises *prems* **for** t'
proof −

from *prems* **have** *r*: (*frame A* t_0 *s*, t') ∈ *r* **by** *simp*
have *s-eq*: *s* = *hmem-upd* (λ-. *hmem s*) ((*htd-upd* (λ-. *htd s*) (*frame A* t_0 *s*)))
apply (*simp add*: *frame-def*)
by (*metis equal-upto-heap-on-def equal-upto-heap-on-frame frame-def heap.get-upd htd-hmem-upd typing.get-upd*)
show *?thesis*
apply (*subst s-eq*)
apply (*rule exI*)
apply (*rule heap-irrelevant*)
apply (*rule r*)
done
qed
subgoal
apply (*simp add*: *frame-def*)
using *heap-irrelevant htd-unchanged*
by *simp*
subgoal
using *hmem-unchanged* **by** *simp*
subgoal
using *htd-unchanged* **by** *simp*
subgoal
using *htd-unchanged* **by** (*simp add*: *rel-alloc-def*)
subgoal
using *htd-unchanged* **by** (*simp add*: *rel-alloc-def*)
subgoal
using *htd-unchanged* **by** (*simp add*: *rel-alloc-def*)
done
then show *?thesis*
using *hmem-unchanged htd-unchanged*
apply (*clarsimp simp add*: *L2-spec-def refines-def-old rel-stack-def rel-alloc-def reaches-bind succeeds-bind*)
apply (*meson UNIV-I image-eqI*)
apply *force*+
done
qed

lemma *L2-assume-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes ⋀*v s'*. (*v*, *s'*) ∈ *f s* ⟹
∃ *w*. (*w*, *frame A* t_0 *s'*) ∈ *g* (*frame A* t_0 *s*) ∧ *rel-stack* $\mathcal{S}$ *M A s* t_0 *R* (*v*, *s'*) (*w*, *frame A* t_0 *s'*)
shows *refines* (*L2-assume f*) (*L2-assume g*) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R*))
using *assms*
apply (*clarsimp simp add*: *L2-assume-def refines-def-old rel-alloc-def rel-stack-def*)

by *metis*

lemma *L2-assume-rel-stack-heap-agnostic*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *hmem-unchanged*: $\bigwedge a\ s\ s'.\ (a,\ s') \in f\ s \Longrightarrow hmem\ s' = hmem\ s$
assumes *htd-unchanged*: $\bigwedge a\ s\ s'.\ (a,\ s') \in f\ s \Longrightarrow htd\ s' = htd\ s$
assumes *heap-irrelevant*: $\bigwedge a\ s\ s'\ g\ h.\ (a,\ s') \in f\ s \Longrightarrow (a,\ hmem\text{-}upd\ h\ (htd\text{-}upd\ g\ s')) \in f\ (hmem\text{-}upd\ h\ (htd\text{-}upd\ g\ s))$
shows *refines* (*L2-assume f*) (*L2-assume f*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
apply (*rule L2-assume-rel-stack*)
subgoal
using *s-t*
by (*auto simp add*: *L2-assume-def refines-def-old rel-stack-def rel-alloc-def*)
subgoal for *v s'*
apply (*rule exI*[**where** *x*=*v*])
apply (*intro conjI*)
subgoal
using *frame-def heap-irrelevant htd-unchanged* **by** *presburger*
subgoal
apply (*clarsimp simp*: *rel-stack-def frame-def*)
apply (*intro conjI*)
subgoal using ‹*rel-alloc* $\mathcal{S}$ {} *A* t_0 *s t*› *frame-def htd-unchanged rel-alloc-def* **by** *force*
subgoal using *hmem-unchanged* **by** *force*
subgoal using *htd-unchanged* **by** *blast*
done
done
done

lemma *refines-rel-stack-embed-result'*:
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *R1*: $\bigwedge v\ s'\ w\ t'.$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$ *R* (*hmem s'*) *v w* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd s' = *htd s* $\Longrightarrow$
R1 (*hmem s'*) *v* (*emb w*)
assumes *M2-M*: *M2* ⊆ *M*
assumes *M1-M2*: *M1* ⊆ *M2*
shows *refines f* (*L2-seq g* (*ETA-TUPLED* (λ*x*. *L2-gets* (λ-. *emb x*) *ns*))) *s t* (*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack L R1*))
unfolding *L2-defs ETA-TUPLED-def*
apply (*subst bind-return* [*symmetric*, *of f*])
apply (*rule refines-bind-bind-exn* [*OF f*])
subgoal
using *M1-M2 M2-M rel-alloc*
apply (*clarsimp simp add*: *rel-stack-def rel-alloc-def*)
by *auto* (*meson Un-mono equal-upto-mono equalityE*)

subgoal by *auto*
subgoal by *auto*
subgoal
apply (*clarsimp simp add*: *gets-return rel-stack-def*)
apply (*intro conjI*)
subgoal using *R1 rel-alloc* **by** (*auto simp add*: *rel-alloc-def*)
subgoal using *M2-M rel-alloc* **by** (*auto simp add*: *rel-alloc-def*)
subgoal using *M1-M2* **by** (*meson equal-upto-mono equalityD2 sup.mono*)
done
done

lemma *refines-rel-stack-root-upd-result*:
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: *P t* $\Longrightarrow$ *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *R1*: $\bigwedge$*v s′ w t′*. *rel-alloc* $\mathcal{S}$ *M A* t_0 *s′ t′* $\Longrightarrow$ *R* (*hmem s′*) *v w* $\Longrightarrow$ *P t* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s′*)) (*hmem s′*) (*hmem s*) $\Longrightarrow$
htd s′ = *htd s* $\Longrightarrow$
R1 (*hmem s′*) *v* (*emb w*)
assumes *M2-M*: *P t* $\Longrightarrow$ *M2* ⊆ *M*
assumes *M1-M2*: *P t* $\Longrightarrow$ *M1* ⊆ *M2*
shows *refines f* (*L2-seq* (*L2-guard P*) (λ-. *L2-seq g* (*ETA-TUPLED* (λ*x*. *L2-gets* (λ-. *emb x*) *ns*)))) *s t* (*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack L R1*))
using *refines-rel-stack-embed-result′*[*OF assms*(*1*) - - *assms*(*4*,*5*), *of f g L R R1 emb*]
using *assms*(*2*,*3*)
by (*auto simp add*: *refines-bind-guard-right-iff L2-defs ETA-TUPLED-def*)

lemma *refines-rel-stack-embed-exit*:
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *L1*: $\bigwedge$*v s′ w t′*. *rel-alloc* $\mathcal{S}$ *M A* t_0 *s′ t′* $\Longrightarrow$ *L* (*hmem s′*) *v w* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s′*)) (*hmem s′*) (*hmem s*) $\Longrightarrow$
htd s′ = *htd s* $\Longrightarrow$
L1 (*hmem s′*) *v* (*emb w*)
assumes *M2-M*: *M2* ⊆ *M*
assumes *M1-M2*: *M1* ⊆ *M2*
shows *refines*
f (*L2-catch g* (*ETA-TUPLED* (λ*x*. *L2-throw* (*emb x*) *ns*))) *s t*
(*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack L1 R*))
unfolding *L2-defs ETA-TUPLED-def*
apply (*subst f-catch-throw* [*symmetric*, *of f*])
apply (*rule refines-catch* [*OF assms*(*2*)])
subgoal
apply (*clarsimp simp add*: *rel-stack-def*, *intro conjI*)
subgoal using *L1 rel-alloc* **by** (*auto simp add*: *rel-alloc-def*)
subgoal using *M2-M rel-alloc* **by** (*auto simp add*: *rel-alloc-def*)
subgoal using *M1-M2* **by** (*meson equal-upto-mono equalityD2 sup.mono*)

```
    done
  subgoal by auto
  subgoal by auto
  subgoal
    apply (simp add: rel-stack-def)
    apply (intro conjI)
    subgoal using M2-M rel-alloc by (auto simp add: rel-alloc-def)
   subgoal using M1-M2 by auto (meson equal-upto-mono equalityD2 sup.mono)
    done
  done

lemma refines-rel-stack-embed-both:
  assumes stack: rel-alloc 𝒮 M A t₀ s t
  assumes f-g: refines f g s t (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack L R))
  assumes L: ⋀v s' w t'. rel-alloc 𝒮 M A t₀ s' t' ⟹ L (hmem s') v w ⟹
            equal-upto (M1 ∪ stack-free (htd s')) (hmem s') (hmem s) ⟹
            htd s' = htd s ⟹
             L1 (hmem s') v (embL w)
  assumes R: ⋀v s' w t'. rel-alloc 𝒮 M A t₀ s' t' ⟹ R (hmem s') v w ⟹
            equal-upto (M1 ∪ stack-free (htd s')) (hmem s') (hmem s) ⟹
            htd s' = htd s ⟹
             R1 (hmem s') v (embR w)
  assumes M2-M: M2 ⊆ M
  assumes M1-M2: M1 ⊆ M2
  shows refines
  f
  (L2-seq
    (L2-catch g (ETA-TUPLED (λx. L2-throw (embL x) ns)))
    (ETA-TUPLED (λx. L2-gets (λ-. embR x) ns))) s t
  (rel-stack 𝒮 M2 A s t₀ (rel-xval-stack L1 R1))
  apply (rule refines-rel-stack-embed-result' [OF stack -  - M2-M M1-M2, where
R=R])
  subgoal
    apply (rule refines-rel-stack-embed-exit [OF stack f-g ])
    subgoal
      by (rule L)
    subgoal using M1-M2 M2-M by auto
    subgoal by auto
    done
  subgoal
    by (rule R)
  done
end


lemma refines-assume-result-and-state-left:
  assumes ⋀s' v. (v, s') ∈ A s ⟹ refines (f v) g s' t Q
  shows refines (do { v <- assume-result-and-state A; f v }) g s t Q
  using assms
```

apply (*auto simp add*: *refines-def-old succeeds-bind reaches-bind split*: *xval-splits*)
done

lemma *refines-assume-result-and-state-right*:
assumes $A\ t \neq \{\}$
assumes $\bigwedge t'\ v.\ (v,\ t') \in A\ t \Longrightarrow$ *refines* $f\ (g\ v)\ s\ t'\ Q$
shows *refines* f (*do* { v <− *assume-result-and-state* A; $g\ v$ }) $s\ t\ Q$
using *assms*
apply (*clarsimp simp add*: *refines-def-old succeeds-bind reaches-bind split*: *exception-or-result-splits*)
by *auto* (*metis* (*no-types*, *opaque-lifting*) *Result-eq-Result is-Result-simps*(*1*) *is-Result-simps*(*2*))

lemma *refines-assume-result-and-state-both*:
assumes $B\ t = \{\} \Longrightarrow A\ s = \{\}$
assumes $\bigwedge s'\ v\ t'\ w.\ (v,\ s') \in A\ s \Longrightarrow (w,\ t') \in B\ t \Longrightarrow$ *refines* $(f\ v)\ (g\ w)\ s'\ t'$ Q
shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) (*do* {w <− *assume-result-and-state* B; $g\ w$ }) $s\ t\ Q$
apply (*rule refines-bind'*)
apply (*simp add*: *refines-assume-result-and-state-iff*)
using *assms*
by (*force simp add*: *sim-set-def*)

lemma *refines-assume-result-and-state-both-same-val'*:
assumes $\bigwedge s'\ v.\ (v,\ s') \in A\ s \Longrightarrow \exists\, t'.\ (v,\ t') \in B\ t \wedge$ *refines* $(f\ v)\ (g\ v)\ s'\ t'\ Q$
shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) (*do* { v <− *assume-result-and-state* B; $g\ v$ }) $s\ t\ Q$
apply (*rule refines-bind'*)
apply (*simp add*: *refines-assume-result-and-state-iff*)
using *assms*
by (*force simp add*: *sim-set-def*)

lemma *refines-assume-result-and-state-both-same-val*:
assumes $\bigwedge v\ s'.\ (v,\ s') \in A\ s \Longrightarrow \exists\, t'.\ (v,\ t') \in B\ t$
assumes $\bigwedge s'\ v\ t'.\ (v,\ s') \in A\ s \Longrightarrow (v,\ t') \in B\ t \Longrightarrow$ *refines* $(f\ v)\ (g\ v)\ s'\ t'\ Q$
shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) (*do* { v <− *assume-result-and-state* B; $g\ v$ }) $s\ t\ Q$
apply (*rule refines-bind'*)
apply (*simp add*: *refines-assume-result-and-state-iff*)
using *assms*
by (*force simp add*: *sim-set-def*)

lemma *refines-assume-result-and-state-both-same-val-frame*:
assumes *f*: $t = frm\ s$
assumes $\bigwedge s'\ v.\ (v,\ s') \in A\ s \Longrightarrow (v,\ frm\ s') \in B\ t$
assumes $\bigwedge s'\ v.\ (v,\ s') \in A\ s \Longrightarrow$ *refines* $(f\ v)\ (g\ v)\ s'\ (frm\ s')\ Q$
shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) (*do* { v <− *as-*

sume-result-and-state B; *g v* }) *s t Q*
apply (*rule refines-bind'*)
apply (*simp add*: *refines-assume-result-and-state-iff*)
using *assms*
by (*force simp add*: *sim-set-def*)

lemma *refines-on-exit-left*:
assumes *f-g*: *refines f g s0 t Q'*
assumes *1*: $\bigwedge s.\ \exists\, s'.\ (s,\ s') \in cleanup$
assumes *2*: $\bigwedge s\ v\ t\ w\ s'.\ Q'\ (v,\ s)\ (w,\ t) \Longrightarrow (s,\ s') \in cleanup \Longrightarrow Q\ (v,\ s')\ (w,\ t)$
shows *refines* (*on-exit f cleanup*) *g s0 t Q*
unfolding *on-exit-def*
apply (*subst on-exit'-skip*[*of g*, *symmetric*])
apply (*rule refines-on-exit'*[*OF f-g*[*THEN refines-weaken*]])
apply (*auto simp*: *refines-yield-right-iff runs-to-iff 1 2*)
done

lemma *refines-on-exit-same-cleanup*:
assumes *f-g*: *refines f g s0 t Q'*
assumes *1*: $\bigwedge s.\ \exists\, s'.\ (s,\ s') \in cleanup$
assumes *2*: $\bigwedge s\ v\ t\ w\ s'\ t'.\ Q'\ (v,\ s)\ (w,\ t) \Longrightarrow (s,\ s') \in cleanup \Longrightarrow (t,\ t') \in cleanup \Longrightarrow Q\ (v,\ s')\ (w,\ t')$
shows *refines* (*on-exit f cleanup*) (*on-exit g cleanup*) *s0 t Q*
unfolding *on-exit-def*
apply (*rule refines-on-exit'*[*OF f-g*[*THEN refines-weaken*]])
using *1 2*
apply *clarsimp*
apply (*rule refines-state-select*)
apply *force*
apply *force*
done

lemma *refines-on-exit-same-cleanup-choice*:
assumes *f-g*: *refines f g s0 t Q'*
assumes *1*: $\bigwedge s.\ \exists\, s'.\ (s,\ s') \in cleanup$
assumes *2*: $\bigwedge s\ v\ t\ w\ s'.\ Q'\ (v,\ s)\ (w,\ t) \Longrightarrow (s,\ s') \in cleanup \Longrightarrow \exists\, t'.\ (t,\ t') \in cleanup \wedge\ Q\ (v,\ s')\ (w,\ t')$
shows *refines* (*on-exit f cleanup*) (*on-exit g cleanup*) *s0 t Q*
unfolding *on-exit-def*
apply (*rule refines-on-exit'*[*OF f-g*[*THEN refines-weaken*]])
using *1 2*
apply *clarsimp*
apply (*rule refines-state-select*)
apply *auto*
done

lemma *refines-runs-to-partial-fuse-both*:
assumes *sim*: *refines f f' s s' Q*

assumes *runs-to-f*: $f \cdot s$?⦇*P1*⦈
assumes *runs-to-f'*: $f' \cdot s'$?⦇*P2*⦈
shows *refines f f' s s'* (λ(*r*,*t*) (*r'*,*t'*). *Q* (*r*,*t*) (*r'*,*t'*) ∧ *P1 r t* ∧ *P2 r' t'*)
using *sim runs-to-f runs-to-f'*
apply (*force simp add*: *refines-def-old runs-to-partial-def-old*)
done

definition *domain-bound A Q* = (∀ *h h'*. *equal-on A h h'* ⟶ *Q h* = *Q h'*)

lemma *domain-bound-equal-on*: *domain-bound A Q* ⟹ *equal-on A h h'* ⟹ *Q h* = *Q h'*
by (*auto simp add*: *domain-bound-def*)

lemma *domain-bound-equal-on-subset*: *domain-bound A Q* ⟹ *A* ⊆ *A'* ⟹ *equal-on A' h h'* ⟹ *Q h* = *Q h'*
using *domain-bound-equal-on equal-on-mono* **by** *blast*

lemma *domain-bound-heap-update-list*:
fixes *p*::'*a*::*mem-type ptr*
assumes *domain-bound A Q*
assumes *length bs* = *size-of TYPE*('*a*)
assumes *ptr-span p* ∩ *A* = {}
shows *Q* (*heap-update-list* (*ptr-val p*) *bs h*) = *Q h*
using *assms domain-bound-equal-on*
by (*smt* (*verit*, *best*) *equal-on-def heap-update-nmem-same orthD2*)

named-theorems *domain-bound-intros*

lemma *domain-bound-mono*: *A* ⊆ *A'* ⟹ *domain-bound A Q* ⟹ *domain-bound A' Q*
by (*auto simp add*: *domain-bound-def equal-on-mono*)

lemma *domain-bound-eq*: *domain-bound* {} (λ-. (=))
by (*auto simp add*: *domain-bound-def*)

lemma *domain-bound-rel-sum-stack*[*domain-bound-intros*]: *domain-bound A L* ⟹ *domain-bound A R* ⟹ *domain-bound A* (*rel-sum-stack L R*)
by (*auto simp add*: *domain-bound-def rel-sum-stack-def*)

lemma *domain-bound-rel-xval-stack*[*domain-bound-intros*]: *domain-bound A L* ⟹ *domain-bound A R* ⟹ *domain-bound A* (*rel-xval-stack L R*)
by (*auto simp add*: *domain-bound-def rel-xval-stack-def*)

lemma *domain-bound-unreachable-exit* [*domain-bound-intros*]:
domain-bound A (*rel-exit* (λ- - -. *False*))
by (*auto simp add*: *domain-bound-def rel-exit-def*)

lemma *domain-bound-bot*[*domain-bound-intros*]: *domain-bound A* (λ- - -. *False*)

by (*auto simp add*: *domain-bound-def*)

lemma *domain-bound-eq'*[*domain-bound-intros*]: *domain-bound A* (λ-. (=))
using *domain-bound-eq domain-bound-mono* **by** *blast*

lemma *domain-bound-top*[*domain-bound-intros*]: *domain-bound A* (λ- - -. *True*)
by (*auto simp add*: *domain-bound-def*)

lemma *domain-bound-rel-singleton-stack*: *domain-bound* (*ptr-span p*) (*rel-singleton-stack p*)
using *domain-rel-singleton-stack* **by** (*auto simp add*: *domain-bound-def*)

lemma *domain-bound-rel-exit*[*domain-bound-intros*]:
domain-bound A Q ⟹ *domain-bound A* (*rel-exit Q*)
by (*auto simp add*: *rel-exit-def domain-bound-def*)

lemma *domain-bound-rel-singleton-stack'*[*domain-bound-intros*]:
ptr-span p ⊆ *A* ⟹ *domain-bound A* (*rel-singleton-stack p*)
using *domain-bound-rel-singleton-stack domain-bound-mono* **by** *blast*

lemma *domain-rel-push*: *domain-bound A Q* ⟹ *equal-on* (*ptr-span p* ∪ *A*) *h h'* ⟹ *rel-push p Q h* = *rel-push p Q h'*
apply (*simp add*: *domain-bound-def rel-push-def fun-eq-iff*)
by (*metis Un-upper1 domain-rel-singleton-stack equal-on-mono rel-singleton-stack-def sup-commute*)

lemma *domain-bound-rel-push*: *domain-bound A Q* ⟹ *domain-bound* (*ptr-span p* ∪ *A*) (*rel-push p Q*)
using *domain-rel-push domain-bound-def* **by** *blast*

lemma *domain-bound-rel-push'*[*domain-bound-intros*]: *ptr-span p* ⊆ *A* ⟹ *domain-bound A Q* ⟹ *domain-bound A* (*rel-push p Q*)
using *domain-bound-rel-push*
by (*metis* (*no-types*, *lifting*) *subset-Un-eq*)

context *stack-heap-state*
begin

lemma *with-fresh-stack-ptr-rel-stack''*:
assumes *stack*: *rel-alloc* 𝒮 *M A* t_0 *s t*
assumes *domain-bound*: *domain-bound A Q*
assumes *f*: ⋀*p s'* t_0.
⟦*ptr-span p* ⊆ 𝒮; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;

```
    [h-val (hmem s') p] ∈ I s;
    equal-upto-heap-on (ptr-span p) s' s;
    stack-free (htd s') ⊆ stack-free (htd s);
    rel-alloc S (ptr-span p ∪ M) (ptr-span p ∪ A) t0 s' t⟧
   ⟹ refines (f p) g s' t (rel-stack S (ptr-span p ∪ M1) (ptr-span p ∪ A) s' t0 Q)
  assumes M1-M: M1 ⊆ M
  shows refines (with-fresh-stack-ptr (Suc 0) I f) g s t (rel-stack S M1 A s t0 Q)
  unfolding with-fresh-stack-ptr-def
  apply (rule refines-assume-result-and-state-left)
  apply clarsimp
  subgoal for p d vs bs
    apply (rule refines-on-exit-left)
      apply (rule f [of - - htd-upd (λd. override-on d stack-byte-typing (ptr-span
p)) t0])
    subgoal
      by (erule stack-allocs-cases) (auto)
    subgoal
      using stack-allocs-stack-subset-stack-free by fastforce
    subgoal
      apply (erule stack-allocs-cases)
      using stack
      by (auto simp add: rel-alloc-def stack-free-def)
    subgoal
      apply (erule stack-allocs-cases)
      using stack
      by (auto simp add: rel-alloc-def stack-free-def)
    subgoal
      apply (erule stack-allocs-cases)
      using stack
      by (auto simp add: rel-alloc-def)
    subgoal
      apply (erule stack-allocs-cases)
      using stack
      by (metis h-val-heap-update-padding heap.get-upd length-1-conv nth-Cons-0)
    subgoal
      apply (erule stack-allocs-cases)
      apply (auto simp add: equal-upto-heap-stack-alloc)

      done
    subgoal
      by (simp add: stack-free-stack-allocs)
    subgoal
      apply (frule stack-free-stack-allocs)
      apply (erule stack-allocs-cases)
      using stack M1-M
      apply (simp add: rel-alloc-def)
      apply (simp add: frame-heap-update-padding)
      using frame-stack-alloc
      apply (subst frame-stack-alloc)
```

apply (*auto simp add*: *stack-free-def*)
done
subgoal
by *blast*
subgoal
apply *clarsimp*
apply (*clarsimp simp add*: *rel-stack-def*)
apply (*clarsimp simp add*: *rel-alloc-def*)
apply (*intro conjI*)
subgoal
apply (*erule stack-allocs-cases*)
using *domain-bound stack domain-bound-heap-update-list*
apply (*simp add*: *rel-alloc-def*)
by (*simp add*: *disjoint-subset domain-bound-heap-update-list*)
subgoal
by (*metis* (*no-types*, *lifting*) *Int-Un-eq*(*1*) *distrib-inf-le equal-on-stack-free-preservation*

in-mono inf-idem rel-alloc-def stack stack-free-stack-allocs stack-free-stack-releases-mono'

stack-releases-stack-allocs-inverse subset-drop-Diff-strg)
subgoal
by (*metis* (*no-types*, *lifting*) *Int-Un-eq*(*1*) *distinct-element distrib-inf-le*
equal-on-stack-free-preservation inf.idem order-antisym-conv rel-alloc-def
stack stack-free-stack-allocs stack-free-stack-releases-mono' stack-releases-stack-allocs-inverse
subset-drop-Diff-strg)
subgoal
apply (*erule stack-allocs-cases*)
using *M1-M*
by (*smt* (*verit*, *best*) *UnE c-guard-def disj-subset disjoint-union-distrib*
inter-commute
orthD2 rel-alloc-def stack stack-free-stack-releases)

subgoal
apply (*erule stack-allocs-cases*)
apply (*subst frame-stack-release*)
subgoal by *auto*
subgoal by *auto* (*metis IntI empty-iff mem-Collect-eq rel-alloc-def stack*
stack-free-def)
subgoal by *auto* (*meson c-null-guard-def*)
apply *auto*
done
subgoal
apply (*subst stack-free-stack-releases*)
subgoal
by (*meson c-guard-def stack-allocs-cases*)
subgoal
apply *clarsimp*
by (*simp add*: *equal-upto-def heap-update-nmem-same heap-update-padding-def*)
done

```
      subgoal
        using stack-allocs-releases-equal-on-stack
        by (smt (verit, del-insts) UnE add.right-neutral equal-upto-def mult-is-0
         stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse
times-nat.simps(2))
      done
    done
  done

lemma gen-with-fresh-stack-ptr-rel-stack:
  assumes I': hd ‘ I s ⊆ I' — There are two cases of local variables: I s = (λ-.
UNIV) (unitialized) and I s = (λ-. {v}) (initialized)
  assumes stack: rel-alloc S M A t₀ s t
  assumes domain-bound: domain-bound A Q
  assumes f: ⋀p s' t₀ v.
    ⟦ptr-span p ⊆ S; ptr-span p ⊆ stack-free (htd s);
     ptr-span p ∩ A = {}; ptr-span p ∩ M = {};
     root-ptr-valid (htd s') p;
     h-val (hmem s') p = v;
     [v] ∈ I s;
     equal-upto-heap-on (ptr-span p) s' s;
     stack-free (htd s') ⊆ stack-free (htd s);
     rel-alloc S (ptr-span p ∪ M) (ptr-span p ∪ A) t₀ s' t⟧
    ⟹ refines (f p) (g v) s' t (rel-stack S (ptr-span p ∪ M1) (ptr-span p ∪ A) s'
t₀ Q)
  assumes M1-M: M1 ⊆ M
  shows refines (with-fresh-stack-ptr (Suc 0) I f) (select I' >>= g) s t (rel-stack
S M1 A s t₀ Q)
  unfolding with-fresh-stack-ptr-def
  apply (rule refines-assume-result-and-state-left)
  apply clarsimp
  subgoal for p d vs bs
    apply (rule refines-on-exit-left)
      apply (rule refines-select-right-witness [where x=hd vs])
      subgoal using I'
        by auto
      apply (rule f [of - - - htd-upd (λd. override-on d stack-byte-typing (ptr-span
p)) t₀])
    subgoal
      by (erule stack-allocs-cases) (auto)
    subgoal
      using stack-allocs-stack-subset-stack-free by fastforce
    subgoal
      apply (erule stack-allocs-cases)
      using stack
      by (auto simp add: rel-alloc-def stack-free-def)
    subgoal
      apply (erule stack-allocs-cases)
      using stack
```

by (*auto simp add*: *rel-alloc-def stack-free-def*)
subgoal
apply (*erule stack-allocs-cases*)
using *stack*
by (*auto simp add*: *rel-alloc-def*)
subgoal
apply (*erule stack-allocs-cases*)
apply (*cases vs*)
apply (*auto simp add*: *h-val-heap-update-padding*)
done
subgoal
by (*cases vs*) *auto*
subgoal
apply (*erule stack-allocs-cases*)
apply (*auto simp add*: *equal-upto-heap-stack-alloc*)
done
subgoal
by (*simp add*: *stack-free-stack-allocs*)
subgoal
apply (*frule stack-free-stack-allocs*)
apply (*erule stack-allocs-cases*)
using *stack M1-M*
apply (*simp add*: *rel-alloc-def*)
apply (*simp add*: *frame-heap-update-padding*)
using *frame-stack-alloc*
apply (*subst frame-stack-alloc*)
apply (*auto simp add*: *stack-free-def*)
done
subgoal
by *blast*
subgoal
apply *clarsimp*
apply (*clarsimp simp add*: *rel-stack-def*)
apply (*clarsimp simp add*: *rel-alloc-def*)
apply (*intro conjI*)
subgoal
apply (*erule stack-allocs-cases*)
using *domain-bound stack domain-bound-heap-update-list*
apply (*simp add*: *rel-alloc-def*)
by (*simp add*: *disjoint-subset domain-bound-heap-update-list*)
subgoal
by (*metis* (*no-types*, *lifting*) *Int-Un-eq*(*1*) *distrib-inf-le equal-on-stack-free-preservation*

in-mono inf-idem rel-alloc-def stack stack-free-stack-allocs stack-free-stack-releases-mono′

stack-releases-stack-allocs-inverse subset-drop-Diff-strg)
subgoal
by (*metis* (*no-types*, *lifting*) *Int-Un-eq*(*1*) *distinct-element distrib-inf-le*
equal-on-stack-free-preservation inf.idem order-antisym-conv rel-alloc-def

stack stack-free-stack-allocs stack-free-stack-releases-mono′ stack-releases-stack-allocs-inverse subset-drop-Diff-strg)
subgoal
apply (*erule stack-allocs-cases*)
using *M1-M*
by (*smt* (*verit*, *best*) *UnE c-guard-def disj-subset disjoint-union-distrib inter-commute*
orthD2 rel-alloc-def stack stack-free-stack-releases)

subgoal
apply (*erule stack-allocs-cases*)
apply (*subst frame-stack-release*)
subgoal by *auto*
subgoal by *auto* (*metis IntI empty-iff mem-Collect-eq rel-alloc-def stack stack-free-def*)
subgoal by *auto* (*meson c-null-guard-def*)
apply *auto*
done
subgoal
apply (*subst stack-free-stack-releases*)
subgoal
by (*meson c-guard-def stack-allocs-cases*)
subgoal
apply *clarsimp*
by (*simp add*: *equal-upto-def heap-update-nmem-same heap-update-padding-def*)
done
subgoal
using *stack-allocs-releases-equal-on-stack*
by (*smt* (*verit*, *del-insts*) *UnE add.right-neutral equal-upto-def mult-is-0*
stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse times-nat.simps(*2*))
done
done
done

lemma *with-fresh-stack-ptr-rel-stack-uninitialized′*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0\ v.$
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s′*) *p*;
h-val (*hmem s′*) *p* = *v*;
equal-upto-heap-on (*ptr-span p*) *s′ s*;
stack-free (*htd s′*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s′ t*⟧
⟹ *refines* (*f p*) (*g v*) *s′ t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s′* t_0 *Q*)
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*

shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. *UNIV*) (*L2-VARS f ns*)) (*L2-seq* (*L2-unknown ns*) *g*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
unfolding *L2-seq-def L2-unknown-def L2-VARS-def*
by (*rule gen-with-fresh-stack-ptr-rel-stack* [*OF hd-UNIV stack domain-bound f M1-M*])

lemma *with-fresh-stack-ptr-rel-stack-uninitialized*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: ⋀*p s*′ t_0.
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s*′) *p*;
equal-upto-heap-on (*ptr-span p*) *s*′ *s*;
stack-free (*htd s*′) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s*′ *t*⟧
⟹ *refines* (*f p*) (*g*′ *s*′ *p*) *s*′ *t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s*′ t_0 *Q*)
assumes *g*: ⋀*s*′ *p*. *ptr-span p* ∩ *A* = {} ⟹ *equal-upto* (*ptr-span p*) (*hmem s*′) (*hmem s*) ⟹ *g*′ *s*′ *p* = *g* (*h-val* (*hmem s*′) *p*)
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. *UNIV*) (*L2-VARS f ns*)) (*L2-seq* (*L2-unknown ns*) *g*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
using *with-fresh-stack-ptr-rel-stack-uninitialized*′ [*OF stack - M1-M domain-bound*] *f g*
by (*smt* (*verit*, *best*) *equal-upto-heap-on-hmem*)

lemma *with-fresh-stack-ptr-rel-stack-uninitialized-g-normalised*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: ⋀*p s*′ t_0.
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s*′) *p*;
equal-upto-heap-on (*ptr-span p*) *s*′ *s*;
stack-free (*htd s*′) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s*′ *t*⟧
⟹ *refines* (*f p*) (*g* (*h-val* (*hmem s*′) *p*)) *s*′ *t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s*′ t_0 *Q*)
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. *UNIV*) (*L2-VARS f ns*)) (*L2-seq* (*L2-unknown ns*) *g*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
apply (*rule with-fresh-stack-ptr-rel-stack-uninitialized* [*OF stack f - M1-M domain-bound*])
apply *auto*
done

lemma *with-fresh-stack-ptr-rel-stack-initialized*′:
fixes *g*:: (′*d*, ′*e*, ′*s*) *exn-monad*

assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0$.
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
h-val (*hmem s'*) *p* = *v*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s' t*⟧
⟹ *refines* (*f p*) *g s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s'* t_0 *Q*)
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. {[*v*]}) (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
unfolding *L2-VARS-def*
apply (*rule gen-with-fresh-stack-ptr-rel-stack* [**where** *I*= (λ-. {[*v*]}), *OF hd-singleton*, *simplified select-singleton-conv*, *OF stack domain-bound f M1-M*])
apply *auto*
done

lemma *with-fresh-stack-ptr-rel-stack-initialized*:
fixes *g*:: (*'d*, *'e*, *'s*) *exn-monad*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0$.
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
h-val (*hmem s'*) *p* = *v*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s' t*⟧
⟹ *refines* (*f p*) (*g' s' p*) *s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s'* t_0 *Q*)
assumes *g*: $\bigwedge s'\ p$. *ptr-span p* ∩ *A* = {} ⟹ *equal-upto* (*ptr-span p*) (*hmem s'*) (*hmem s*) ⟹ *h-val* (*hmem s'*) *p* = *v* ⟹ *g' s' p* = *g*
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. {[*v*]}) (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
using *with-fresh-stack-ptr-rel-stack-initialized'* [*OF stack - M1-M domain-bound*] *g*
equal-upto-heap-on-hmem f **by** *auto*

lemma *with-fresh-stack-ptr-rel-stack-fix-initialized*:
fixes *g*:: (*'d*, *'e*, *'s*) *exn-monad*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0$.
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};

root-ptr-valid (*htd* s') p;
h-val (*hmem* s') $p = v$;
equal-upto-heap-on (*ptr-span* p) s' s;
stack-free (*htd* s') $\subseteq$ *stack-free* (*htd* s);
rel-alloc $\mathcal{S}$ (*ptr-span* $p \cup M$) (*ptr-span* $p \cup A$) t_0 s' t⟧
$\Longrightarrow$ *refines* (*f p*) (*g'* *s'* *p*) *s'* *t* (*rel-stack* $\mathcal{S}$ (*ptr-span* $p \cup M1$) (*ptr-span* $p \cup A$) s' t_0 Q)
assumes *g*: $\bigwedge s'$ *p*. *ptr-span* $p \cap A = \{\} \Longrightarrow$ *equal-upto* (*ptr-span p*) (*hmem* s') (*hmem s*) $\Longrightarrow$ *h-val* (*hmem* s') $p = v \Longrightarrow$ *g'* *s'* *p* = *g*
assumes *M1-M*: $M1 \subseteq M$
assumes *I*: $I\ s = \{[v]\}$
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
proof −
from *I* **have** *eq*: *run* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. {[*v*]}) (*L2-VARS f ns*)) *s* = *run* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) *s*
unfolding *with-fresh-stack-ptr-def*
by (*simp add*: *run-bind*)
from *with-fresh-stack-ptr-rel-stack-initialized* [*OF stack f g M1-M domain-bound*]
have *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. {[*v*]}) (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*) **.**
from *refines-subst-left* [*OF this*] *eq*
show *?thesis*
by *blast*
qed

lemma *with-fresh-stack-ptr-rel-stack-fix-initialized-g-normalised*:
fixes *g*:: ('*d*, '*e*, '*s*) *exn-monad*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0$.
⟦*ptr-span p* $\subseteq \mathcal{S}$; *ptr-span p* $\subseteq$ *stack-free* (*htd s*);
ptr-span p $\cap$ *A* = {}; *ptr-span p* $\cap$ *M* = {};
root-ptr-valid (*htd* s') *p*;
h-val (*hmem* s') $p = v$;
equal-upto-heap-on (*ptr-span p*) s' *s*;
stack-free (*htd* s') $\subseteq$ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span* $p \cup M$) (*ptr-span* $p \cup A$) t_0 s' t⟧
$\Longrightarrow$ *refines* (*f p*) *g s'* *t* (*rel-stack* $\mathcal{S}$ (*ptr-span* $p \cup M1$) (*ptr-span* $p \cup A$) s' t_0 Q)
assumes *M1-M*: $M1 \subseteq M$
assumes *I*: $I\ s = \{[v]\}$
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
apply (*rule with-fresh-stack-ptr-rel-stack-fix-initialized* [*OF stack f - M1-M - domain-bound*])
using *I*
apply *auto*
done

lemma *with-fresh-stack-ptr-rel-stack*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0$.
$[\![$
ptr-span p $\subseteq \mathcal{S}$; *ptr-span p* $\subseteq$ *stack-free* (*htd s*);
ptr-span p $\cap$ *A* = {}; *ptr-span p* $\cap$ *M* = {};
root-ptr-valid (*htd s'*) *p*;
[*h-val* (*hmem s'*) *p*] $\in$ *I s*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) $\subseteq$ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* $\cup$ *M*) (*ptr-span p* $\cup$ *A*) t_0 *s' t*$]\!]$
$\Longrightarrow$ *refines* (*f p*) (*g' s' p*) *s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* $\cup$ *M1*) (*ptr-span p* $\cup$ *A*) *s'* t_0 *Q*)
assumes *g*: $\bigwedge s'\ p$. *ptr-span p* $\cap$ *A* = {} $\Longrightarrow$ *equal-upto* (*ptr-span p*) (*hmem s'*) (*hmem s*) $\Longrightarrow$ [*h-val* (*hmem s'*) *p*] $\in$ *I s* $\Longrightarrow$ *g' s' p* = *g*
assumes *M1-M*: *M1* $\subseteq$ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
unfolding *L2-VARS-def*
using *with-fresh-stack-ptr-rel-stack''* [*OF stack domain-bound*] *M1-M f g equal-upto-heap-on-hmem*
by *auto*

lemma *refines-rel-stack-project-result*:
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R*))
assumes $\bigwedge h\ x\ y$. *R h x y* $\Longrightarrow$ *R' h x* (*prj y*)
shows *refines f* (*L2-seq g* (*ETA-TUPLED* (λx. *L2-gets* (λ-. *prj x*) *ns*))) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R'*))
using *assms*
apply (*clarsimp simp add*: *refines-def-old L2-defs rel-stack-def rel-xval-stack-def ETA-TUPLED-def*
reaches-bind succeeds-bind
split: *prod.splits sum.splits*)
subgoal for *r s'*
apply (*cases r*)
subgoal
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
by (*smt* (*verit*, *del-insts*) *Exn-def Result-neq-Exn case-exception-or-result-Exn rel-xval.simps*)
subgoal
apply *clarsimp*
by (*smt* (*verit*, *ccfv-SIG*) *Result-neq-Exn case-exception-or-result-Result rel-xval.cases rel-xval-simps*(*2*))
done
done

lemma *refines-rel-stack-adjust-result*:

assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R*))
assumes $\bigwedge s'\ t'\ v\ w$. *R* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M* $\cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
equal-upto M (*htd s'*) (*htd s*) $\Longrightarrow$
equal-on $\mathcal{S}$ (*htd s'*) (*htd s*) $\Longrightarrow$ *R'* (*hmem s'*) *v* (*adj w*)
shows *refines f* (*L2-seq g* (*ETA-TUPLED* (λx. *L2-gets* (λ-. *adj x*) *ns*))) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R'*))
using *assms*
apply (*clarsimp simp add*: *refines-def-old L2-defs rel-stack-def rel-xval-stack-def ETA-TUPLED-def*
reaches-bind succeeds-bind
split: *prod.splits sum.splits*)
subgoal for *r s'*
apply (*cases r*)
subgoal
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
by (*smt* (*verit*, *del-insts*) *Exn-def Result-neq-Exn case-exception-or-result-Exn rel-xval.simps*)
subgoal for *v*
apply (*erule-tac x*=*Result v* **in** *allE*)
apply (*erule-tac x*=*s'* **in** *allE*)
apply (*fastforce simp add*: *rel-xval.simps*)
done
done
done

lemma *stack-allocs-frame*:
assumes *alloc*: (*p*, *d*) $\in$ *stack-allocs* (*Suc 0*) $\mathcal{S}$ *TYPE*('*a*::*mem-type*) (*htd s*)
shows $\exists$ *d*. (*p*, *d*) $\in$ *stack-allocs* (*Suc 0*) $\mathcal{S}$ *TYPE*('*a*::*mem-type*) (*htd* (*frame A* t_0 *s*))
proof −
have *stack-free* (*htd s*) $\subseteq$ *stack-free* (*htd* (*frame A* t_0 *s*))
by (*rule stack-free-htd-frame*)
from *stack-allocs-stack-free-mono* [*OF this alloc*]
show *?thesis* **.**
qed

lemma *heap-list-update-nth*:
$\bigwedge h\ p$. *length v* $\leq$ *addr-card* $\Longrightarrow$
i < *length v* $\Longrightarrow$
(*heap-update-list p v h*) (*p* + *of-nat i*) = *v*!*i*
proof (*induct v arbitrary*: *i*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
thus *?case*

by (*metis heap-list-nth heap-list-update*)
qed

lemma *stack-alloc-simulation-aux*:
fixes *p*::*'a*::*mem-type ptr*
assumes *neq-stack-byte*: *typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE(stack-byte)*
assumes *disjnt*: *stack-free (htd s) ∩ A = {}*
assumes *stack-free*: *∀ a∈ptr-span p. root-ptr-valid (htd s) (PTR(stack-byte) a)*
assumes *not-null*: *0 ∉ ptr-span p*
assumes *lbs*: *length bs = size-of TYPE('a)*
assumes *root-ptr*: *root-ptr-valid (ptr-force-type p (htd s)) p*
shows
htd-upd (λx. override-on (ptr-force-type p (htd s)) (htd t₀) (A − stack-free (ptr-force-type p (htd s))))
(hmem-upd (λx. override-on (heap-update-padding p (vs ! 0) bs x) (hmem t₀) (A ∪ stack-free (ptr-force-type p (htd s)))) s) =
htd-upd (λ-. ptr-force-type p (override-on (htd s) (htd t₀) (A − stack-free (htd s))))
(hmem-upd (λx. heap-update-padding p (vs ! 0) bs (override-on x (hmem t₀) (A ∪ stack-free (htd s)))) s)
proof −
from *stack-free* **have** *stack-free'*: *ptr-span p ⊆ stack-free (htd s)*
by (*simp add*: *stack-free-def subsetI*)
with *root-ptr neq-stack-byte*
have *sf-eq*: *stack-free (ptr-force-type p (htd s)) = stack-free (htd s) − ptr-span p*
apply (*clarsimp simp add*: *root-ptr-valid-def stack-free-def*, *intro set-eqI iffI*)
apply (*clarsimp*, *intro conjI*)
apply (*smt* (*verit*, *best*) *ptr-force-type-d size-of-def typ-uinfo-size valid-root-footprint-domain-cases*)

apply (*metis* (*mono-tags*, *lifting*) *size-of-tag valid-root-footprint-type-neq*)
by (*simp add*: *ptr-force-type-d valid-root-footprint-def*)
have *heap-eq*: *override-on (heap-update-padding p (vs ! 0) bs (hmem s)) (hmem t₀) (A ∪ (stack-free (htd s) − ptr-span p))*
=
heap-update-padding p (vs ! 0) bs (override-on (hmem s) (hmem t₀) (A ∪ stack-free (htd s)))
using *disjnt stack-free' lbs*
apply (*clarsimp simp add*: *override-on-def fun-eq-iff*, *intro conjI impI*)
subgoal by (*smt* (*verit*) *CTypes.mem-type-simps*(*2*) *disjoint-subset heap-update-nmem-same heap-update-padding-def orthD2*)
subgoal by (*clarsimp simp add*: *heap-update-nmem-same heap-update-padding-def*)
by (*auto simp add*: *heap-update-mem-same heap-update-padding-def*)
(*smt* (*verit*, *best*) *CTypes.mem-type-simps*(*2*) *heap-update-nmem-same heap-update-padding-def subset-iff*)

have *htd-eq*: *override-on (ptr-force-type p (htd s)) (htd t₀) (A − stack-free (ptr-force-type p (htd s)))*
=

```
    ptr-force-type p (override-on (htd s) (htd t₀) (A − stack-free (htd s)))

    apply (simp add: sf-eq)
    using disjnt stack-free'
    by (auto simp add: override-on-def fun-eq-iff ptr-force-type-split)
  show ?thesis
    apply (simp add: sf-eq)
    using heap-eq htd-eq
    by (metis (no-types, lifting) heap.upd-cong sf-eq typing.upd-cong)
qed

lemma keep-with-fresh-stack-ptr-rel-stack':
  assumes stack: rel-alloc 𝒮 M A t₀ s t
  assumes I: I (frame A t₀ s) = I s
  assumes f: ⋀p s' t₀ t. — I could use the fixed t₀
    ⟦ptr-span p ⊆ 𝒮; ptr-span p ⊆ stack-free (htd s);
     ptr-span p ∩ A = {}; ptr-span p ∩ M = {};
     root-ptr-valid (htd s') p;
     [h-val (hmem s') p] ∈ I s;
     equal-upto-heap-on (ptr-span p) s' s;
     stack-free (htd s') ⊆ stack-free (htd s);
     rel-alloc 𝒮 (ptr-span p ∪ M) A t₀ s' t;
     ptr-span p ∩ stack-free (htd s') = {}⟧
    ⟹ refines (f p) (g p) s' t (rel-stack 𝒮 (ptr-span p ∪ M1) A s' t₀ Q)
  assumes M1-M: M1 ⊆ M
  assumes domain-bound: domain-bound A Q
  shows refines (with-fresh-stack-ptr (Suc 0) I f) (with-fresh-stack-ptr (Suc 0) I
g) s t (rel-stack 𝒮 M1 A s t₀ Q)
  unfolding with-fresh-stack-ptr-def
  apply (rule refines-assume-result-and-state-both-same-val-frame [where frm =
frame A t₀])
  subgoal using stack by (auto simp add: rel-alloc-def)
  subgoal for s' p
    using stack I
    apply (clarsimp simp add: rel-alloc-def)
    apply (frule stack-allocs-frame [where A=A and t₀=t₀])
    apply clarsimp
    apply (erule stack-allocs-cases)
    subgoal for d vs bs d'
      apply (rule exI[where x=d'])
      apply clarsimp
      apply (rule exI[where x=vs])
      apply (simp add: I)
      apply (erule stack-allocs-cases)
      apply clarsimp
      apply (rule exI[where x=bs])
      using stack

      apply (clarsimp simp add: frame-def heap-commute comp-def )
```

apply (*rule stack-alloc-simulation-aux*)
apply *auto*
done
done
subgoal for s' p
apply *clarsimp*
apply (*rule refines-on-exit-same-cleanup-choice*)

apply (*rule f* [*rule-format*, **where** t_0=t_0])
subgoal
by (*erule stack-allocs-cases*) *auto*
subgoal
apply (*erule stack-allocs-cases*)
by (*simp add*: *stack-free-def subset-iff*)
subgoal
using *stack rel-alloc-def stack-allocs-stack-subset-stack-free'* **by** *fastforce*
subgoal using *stack*
by (*metis* (*no-types*, *lifting*) *add.right-neutral inf.orderE inf-assoc inf-bot-right mult-is-0*
rel-alloc-def stack-allocs-stack-subset-stack-free times-nat.simps(*2*))
subgoal
apply (*erule stack-allocs-cases*)
by (*metis htd-hmem-upd typing.get-upd*)
subgoal
by (*metis h-val-heap-update-padding heap.get-upd length-1-conv nth-Cons-0*)
subgoal
by (*smt* (*verit*, *best*) *append.simps*(*1*) *dual-order.refl fold.simps*(*1*) *fold.simps*(*2*)
heap-state.equal-upto-heap-stack-alloc heap-state-axioms id-comp lense.upd-cong ptr-add-0-id
semiring-1-class.of-nat-0 stack-allocs-cases typing.lense-axioms upt.simps(*1*) *upt.simps*(*2*))
subgoal
by (*simp add*: *stack-free-stack-allocs*)
subgoal using *stack M1-M*
by (*smt* (*verit*) *Diff-Diff-Int Diff-eq-empty-iff Int-Diff Int-Un-distrib One-nat-def Un-Int-eq*(*3*)
add-mult-distrib add-right-cancel htd-hmem-upd inf.absorb-iff2 inf.orderE plus-1-eq-Suc
rel-alloc-def stack-allocs-stack-subset-stack-free stack-free-stack-allocs times-nat.simps(*2*) *typing.get-upd*)

subgoal
apply (*erule stack-allocs-cases*)
by (*smt* (*verit*, *ccfv-threshold*) *disjoint-iff htd-hmem-upd in-ptr-span-itself mem-Collect-eq*
root-ptr-valid-casesE stack-free-def typing.get-upd)
subgoal by *blast*
subgoal for *d vs bs s1 v t w s2*

apply *clarsimp*
subgoal for *bs1*
apply (*rule exI*[**where** *x*= *hmem-upd*
(*heap-update-list* (*ptr-val p*)
(*heap-list* (*hmem* t_0) (*size-of TYPE*('*a*)) (*ptr-val p*)))
(*htd-upd* (*stack-releases* (*Suc 0*) *p*) *t*)])

apply (*rule conjI*)
subgoal using *heap-list-length* **by** *blast*
using *stack*
apply (*clarsimp simp add*: *rel-stack-def*)
apply (*intro conjI*)
subgoal
apply (*erule stack-allocs-cases*)
using *domain-bound*
by (*simp add*: *disjoint-subset domain-bound-heap-update-list rel-alloc-def*)
subgoal
apply (*clarsimp simp add*: *rel-alloc-def*, *intro conjI*)
apply (*metis* (*no-types*, *lifting*) *Int-Un-eq*(*1*)
in-mono inf-idem stack-free-stack-allocs stack-free-stack-releases-mono'
stack-releases-stack-allocs-inverse subset-drop-Diff-strg sup.absorb-iff1)
apply (*metis* (*no-types*, *lifting*) *Int-absorb2 Int-iff Un-Int-eq*(*4*)
boolean-algebra.conj-disj-distrib distrib-inf-le orthD1
stack-free-stack-allocs stack-free-stack-releases-mono' stack-releases-stack-allocs-inverse subset-drop-Diff-strg)
using *M1-M*
apply (*smt* (*verit*) *c-guard-def disjoint-subset disjoint-union-distrib inter-commute orthD2*
stack-allocs-cases stack-free-stack-releases)

apply (*erule stack-allocs-cases*)
apply (*subst frame-stack-release-keep* [**where** *bs*=*bs1*])
subgoal
by (*metis disjoint-union-distrib inter-commute*)
subgoal
by (*metis add.right-neutral disjoint-subset mult-Suc mult-is-0*)
subgoal
by (*meson c-guard-def*)
subgoal
by *assumption*
subgoal by *simp*
done
subgoal
apply (*subst stack-free-stack-releases*)
subgoal
by (*meson c-guard-def stack-allocs-cases*)
subgoal
apply *clarsimp*
by (*simp add*: *equal-upto-def heap-update-nmem-same heap-update-padding-def*)

done
subgoal
using *stack-allocs-releases-equal-on-stack*
by (*smt* (*verit*, *del-insts*) *UnE add.right-neutral equal-upto-def mult-is-0*
stack-releases-footprint stack-releases-other stack-releases-stack-allocs-inverse times-nat.simps(*2*))
done
done
done
done

lemma *keep-with-fresh-stack-ptr-rel-stack*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *I*: *I s* = *I* (*frame A* t_0 *s*) — FIXME: make refinement I' similar to condition in *L2-condition* / *L2-while*?
assumes *f*: $\bigwedge p\ s'\ t_0\ t$. — I could use the fixed t_0
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
[*h-val* (*hmem s'*) *p*] ∈ *I s*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) *A* t_0 *s' t*;
ptr-span p ∩ *stack-free* (*htd s'*) = {}⟧
⟹ *refines* (*f p*) (*g' s' p*) *s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) *A s'* t_0 *Q*)
assumes *g*: $\bigwedge s'$ *p*. *ptr-span p* ∩ *A* = {} ⟹ *equal-upto* (*ptr-span p*) (*hmem s'*) (*hmem s*) ⟹ [*h-val* (*hmem s'*) *p*] ∈ *I s*
⟹ *g' s' p* = *g p*
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS g ns*)) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
unfolding *L2-VARS-def*
using *keep-with-fresh-stack-ptr-rel-stack'* [*OF stack I* [*symmetric*] - *M1-M domain-bound*] *f g*
using *equal-upto-heap-on-hmem* **by** *force*

lemma *keep-with-fresh-stack-ptr-rel-stack-g-normalised*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *I*: *I s* = *I* (*frame A* t_0 *s*) — FIXME: make refinement I' similar to condition in *L2-condition* / *L2-while*?
assumes *f*: $\bigwedge p\ s'\ t_0\ t$. — I could use the fixed t_0
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
[*h-val* (*hmem s'*) *p*] ∈ *I s*;
equal-upto-heap-on (*ptr-span p*) *s' s*;

stack-free (*htd* s') $\subseteq$ *stack-free* (*htd* s);
rel-alloc $\mathcal{S}$ (*ptr-span* $p \cup M$) A t_0 s' t;
ptr-span $p \cap$ *stack-free* (*htd* s') $= \{\}$⟧
$\Longrightarrow$ *refines* (*f p*) (*g p*) s' t (*rel-stack* $\mathcal{S}$ (*ptr-span* $p \cup M1$) A s' t_0 Q)
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: *domain-bound* A Q
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) I (*L2-VARS f ns*)) (*with-fresh-stack-ptr* (*Suc 0*) I (*L2-VARS g ns*)) s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 Q)
apply (*rule keep-with-fresh-stack-ptr-rel-stack* [*OF stack I f - M1-M domain-bound*])
apply *auto*
done

lemma *refines-rel-stack-adapt-right*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ M A t_0 s t
assumes *M1-M*: $M1 \subseteq M$
assumes *f-g*: *refines* f g s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* L R))
assumes *h*: $\bigwedge s'$ v t' w.
R (*hmem* s') v w $\Longrightarrow$
rel-alloc $\mathcal{S}$ *M1* A t_0 s' t' $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd* s')) (*hmem* s') (*hmem* s) $\Longrightarrow$
htd s' = *htd* s $\Longrightarrow$
refines (*L2-gets* (λ-. v) *ns*) (*h w*) s' t' (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* L R'))
shows *refines* f (*L2-seq g h*) s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* L R'))
using *stack M1-M f-g h*
apply (*clarsimp simp add*: *refines-def-old L2-defs*
rel-stack-def rel-alloc-def succeeds-bind reaches-bind)
subgoal for r s'
apply (*cases* r)
subgoal for e
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
subgoal for y
apply (*erule-tac x=Exn y* **in** *allE*)
apply (*erule-tac x=*s' **in** *allE*)
by (*metis* (*mono-tags*, *lifting*) *Exn-def case-exception-or-result-Exn rel-xval-stack-simps*(*5*))
done
subgoal for v
apply (*erule-tac x=Result v* **in** *allE*)
apply (*erule-tac x=*s' **in** *allE*)
apply (*fastforce simp add*: *rel-xval.simps*)
done
done
done

lemma *refines-handleE'-both-sides*:
assumes *refines f g s t Q*
assumes $\bigwedge v$ v' s' t'. Q (*Result* v, s') (*Result* v', t') $\Longrightarrow$ R (*Result* v, s') (*Result* v', t')

assumes $\bigwedge v\ e'\ s'\ t'.\ Q\ (Result\ v,\ s')\ (Exn\ e',\ t') \Longrightarrow refines\ (return\ v)\ (h'\ e')\ s'\ t'\ R$
assumes $\bigwedge e\ v'\ s'\ t'.\ Q\ (Exn\ e,\ s')\ (Result\ v',\ t') \Longrightarrow refines\ (h\ e)\ (return\ v')\ s'\ t'\ R$
assumes $\bigwedge e\ e'\ s'\ t'.\ Q\ (Exn\ e,\ s')\ (Exn\ e',\ t') \Longrightarrow refines\ (h\ e)\ (h'\ e')\ s'\ t'\ R$
shows $refines\ (f <catch> h)\ (g <catch> h')\ s\ t\ R$
by (*rule Spec-Monad.refines-catch* [*OF assms*(*1*) *assms*(*5*) *assms*(*4*) *assms*(*3*) *assms*(*2*)])

lemma *refines-rel-stack-map-exn*:
assumes *f-g*: $refines\ f\ g\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ M\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))$
assumes *L*: $\bigwedge e\ e'\ s'.\ L\ (hmem\ s')\ e\ e' \Longrightarrow L'\ (hmem\ s')\ (emb\ e)\ (emb\ e')$
shows $refines\ (map\text{-}value\ (map\text{-}exn\ emb)\ f)\ (map\text{-}value\ (map\text{-}exn\ emb)\ g)\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ M\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L'\ R))$
using *assms*
apply (*clarsimp simp add*: *refines-def-old L2-defs*
rel-stack-def rel-alloc-def reaches-map-value)
subgoal for $s'\ r$
apply (*cases r*)
subgoal for *e*
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
subgoal for *y*
apply (*erule-tac x*=*Exn y* **in** *allE*)
apply (*erule-tac x*=s' **in** *allE*)
by *auto*
done
subgoal for *v*
apply (*erule-tac x*=*Result v* **in** *allE*)
apply (*erule-tac x*=s' **in** *allE*)
apply (*auto*)
done
done
done

lemma *refines-rel-stack-map-exn-exit*:
assumes *f-g*: $refines\ f\ g\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ M\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ (rel\text{-}exit\ L)\ R))$
assumes *L*: $\bigwedge e\ e'\ h.\ L\ h\ e\ e' \Longrightarrow L'\ h\ (emb\ e)\ (emb'\ e')$
shows $refines\ (map\text{-}value\ (map\text{-}exn\ (emb\ o\ the\text{-}Nonlocal))\ f)\ (map\text{-}value\ (map\text{-}exn\ (emb'))\ g)\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ M\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L'\ R))$
using *assms*
apply (*clarsimp simp add*: *refines-def-old L2-defs*
rel-stack-def rel-alloc-def reaches-map-value)
subgoal for $s'\ r$
apply (*cases r*)
subgoal for *e*
apply (*clarsimp simp add*: *default-option-def Exn-def* [*symmetric*])
subgoal for *y*

```
      apply (erule-tac x=Exn y in allE)
      apply (erule-tac x=s' in allE)
      apply (auto simp add: rel-exit-def)
      done
    done
  subgoal for v
    apply (erule-tac x=Result v in allE)
    apply (erule-tac x=s' in allE)
    apply (auto)
    done
  done
done


lemma throw-L2-throw-conv: throw x = L2-throw x []
  by (simp add: L2-throw-def)

lemma L2-catch-join-exn-conv:
  (L2-catch
      (L2-seq
        (L2-catch g (λx. L2-seq (liftE (h x)) (λ-. L2-throw (prj x) ns1)))
        (λs. liftE (g1 s)))
      (λr. L2-throw (emb r) ns2)) =
    (L2-seq
      (L2-catch g (λx. L2-seq (liftE (h x)) (λ-. L2-throw (emb (prj x)) ns2)))
      (λs. liftE (g1 s)))
  unfolding L2-defs

  apply (rule spec-monad-eqI)
  apply (clarsimp simp add: runs-to-iff)
  apply (auto simp add: runs-to-def-old default-option-def Exn-def)
  done

lemma L2-call-rel-stack-embed-exit:
  assumes L: ⋀e e' h. L h e e' ⟹ L' h (emb e) (emb' e')
  assumes f-g: refines
    f
    (L2-seq
      (L2-catch g (λx. (L2-seq (liftE (h x)) (λ-. L2-throw (prj x) ns'))))
      (λs. liftE (g1 s)))
    s t
    (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack (rel-exit L) R))

  shows refines
    (L2-call f (emb o the-Nonlocal) ns)
    (L2-seq
      (L2-catch g (λx. (L2-seq (liftE (h x)) (λ-. L2-throw (emb' (prj x)) ns'))))
      (λs. liftE (g1 s)))
    s t
```

```
    (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack L′ R))
proof −
  from refines-rel-stack-map-exn-exit [where L′ = L′, OF f-g L ]
  have refines
        (map-value (map-exn (emb o the-Nonlocal)) f)
        (map-value (map-exn (emb′))
           (L2-seq
              (L2-catch g (λx. L2-seq (liftE (h x)) (λ-. L2-throw (prj x) ns′)))
              (λs. liftE (g1 s))))
        s t (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack L′ R)) .
  then show ?thesis
   apply (simp add: map-exn-catch-conv L2-catch-def [symmetric] throw-L2-throw-conv)
    apply (simp add:  L2-catch-join-exn-conv)
     apply (simp add: L2-call-def map-exn-catch-conv L2-catch-def [symmetric]
L2-throw-def)
    done
qed

lemma L2-call-rel-stack-nest-exit-guarded:
  assumes f-g: P t ⟹ refines
    f
    (L2-seq (L2-guard P) (λ-.
      (L2-seq
        (L2-catch g (λx. (L2-seq (liftE (h x)) (λ-. L2-throw (prj x) ns′))))
        (λs. liftE (g1 s)))))
    s t
    (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack (rel-exit L) R))
  assumes L: ⋀e e′ h. P t ⟹ L h e e′ ⟹ L′ h (emb e) (emb′ e′)
  shows refines
    (L2-call f (emb o the-Nonlocal) ns)
    (L2-seq (L2-guard P) (λ-.
      (L2-seq
        (L2-catch g (λx. (L2-seq (liftE (h x)) (λ-. L2-throw (emb′ (prj x)) ns′))))
        (λs. liftE (g1 s)))))
    s t
    (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack L′ R))
  apply (rule refines-L2-guard-right′)
  apply (rule L2-call-rel-stack-embed-exit [where L=L and emb=emb and emb′=
emb′])
   apply (rule L, assumption+)
  using f-g refines-L2-guard-rightE by fastforce



lemma bind-catch-liftE-assoc:
  (do {v ← (g <catch> (λx. do {
                                  y ← liftE (h x);
                                  throw (exn x y)
                                }));
```

```
      liftE (g1 v)
    }) =
    (do {v ← g; liftE (g1 v)} <catch> (λx. do {
                                  y ← liftE (h x);
                                  throw (exn x y)
                                }))
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff Exn-def[symmetric] elim!: runs-to-weaken)
  done


lemma bind-catch-liftE-split-catch: (do {
              s ← g;
              liftE (g1 s)
            } <catch> (λx. do {
                         - ← liftE (h x);
                         throw (emb (prj x))
                       })) =
((do {
              s ← g;
              liftE (g1 s)
            } <catch> (λx. do {
                         - ← liftE (h x);
                         throw (prj x) }))
           <catch> (λx. throw (emb x)))
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff Exn-def[symmetric] elim!: runs-to-weaken)
  done

lemma refines-catch-right-trans:
  fixes f:: ('e, 'a, 's) exn-monad
  assumes f-g: refines f g s t Q
  assumes R: ⋀ r s' e t'. Q (r, s') (Exn e, t') ⟹ refines (yield r) (h e) s' t' R
  assumes Exn-Res: ⋀e s' v' t'. Q (Exn e, s') (Result v', t') ⟹ R (Exn e, s')
(Result v', t')
  assumes Res-Res: ⋀v v' s' t'. Q (Result v, s') (Result v', t') ⟹ R (Result v,
s') (Result v', t')
  shows refines f (g <catch> h) s t R
  apply (subst f-catch-throw[symmetric, of f])
  apply (rule refines-catch [OF f-g])
  subgoal premises prems using R [OF prems(1)] by simp
  subgoal using Exn-Res by (auto simp add: refines-yield-right-iff)
  subgoal premises prems using R [OF prems(1)] by simp
  subgoal using Res-Res by (auto simp add: refines-yield-right-iff)
  done

lemma L2-call-rel-stack-embellish-exit:
  assumes s-t: rel-alloc 𝒮 M A t₀ s t
  assumes f-g: P t ⟹ refines
```

```
    f
    (L2-seq (L2-guard P) (λ-.
      (L2-seq
        (L2-catch g (λx. (L2-seq (liftE (h x)) (λ-. L2-throw (prj x) ns))))
        (λs. liftE (g1 s)))))
    s t
    (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack (rel-exit L) R))
  assumes L: ⋀s' t' e e'. L (hmem s') e e' ⟹ P t ⟹ rel-alloc 𝒮 M A t₀ s' t'
⟹
            equal-upto (M1 ∪ stack-free (htd s')) (hmem s') (hmem s) ⟹ htd s' =
htd s ⟹
           L' (hmem s') e (emb e')
  assumes M1: P t ⟹ M1 ⊆ M
  shows refines
    f
    (L2-seq (L2-guard P) (λ-.
      (L2-seq
       (L2-catch g (λx. (L2-seq (liftE (h x)) (λ-. L2-throw (emb (prj x)) ns-exit))))

        (λs. liftE (g1 s)))))
    s t
    (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack (rel-exit L') R))
  unfolding L2-defs
  apply (clarsimp simp add: refines-bind-guard-right-iff bind-catch-liftE-assoc)
  apply (simp add: bind-catch-liftE-split-catch)
  apply (rule refines-catch-right-trans)
   apply (rule f-g [simplified L2-defs refines-bind-guard-right-iff bind-catch-liftE-assoc,
rule-format])
     apply assumption
    apply simp
  subgoal
    apply (clarsimp simp add: rel-stack-def rel-exit-def)
    using L s-t by (auto simp add: rel-alloc-def)
  subgoal
    by (auto simp add: rel-stack-def)
  subgoal
    by (auto simp add: rel-stack-def)
  done

definition override-heap-on P t1 t2 ≡
  hmem-upd (λh. override-on h (hmem t2) P)
    (htd-upd (λd. override-on d (htd t2) P) t1)

lemma override-heap-on-empty [simp]: override-heap-on {} t1 t2 = t1
  by (simp add: override-heap-on-def)

lemma refines-rel-stack-override-heap-on-exit:
  fixes p:: 'a::xmem-type ptr
  assumes stack: rel-alloc 𝒮 M A t₀ s t
```

assumes *f-g*: $\bigwedge s\ t\ t_0$. *rel-alloc* $\mathcal{S}$ *M* $(P \cup A)$ t_0 *s t* $\Longrightarrow$
refines f (*g s*) *s t* (*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* t_0 *Q*)
assumes *disj-A*: $P \cap A = \{\}$
assumes *M1-M*: $M1 \subseteq M$
assumes *disj-stack-free-s*: $P \cap$ *stack-free* (*htd s*) $= \{\}$
shows *refines f* (*g s*) *s t*
(*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* (*override-heap-on P* t_0 *t*) *Q*)
proof −
from *stack* **have** *sf-s-t*: *stack-free* (*htd s*) $\subseteq$ *stack-free* (*htd t*)
using *rel-alloc-def stack-free-htd-frame* **by** *auto*

from *stack* **have** *t*: *t* = *frame A* t_0 *s*
by (*auto simp add*: *rel-alloc-def*)

define *t0′* **where** *t0′* = *override-heap-on P* t_0 *t*

have *eq-htd*:
override-on (*htd s*) (*htd* t_0) (A − *stack-free* (*htd s*)) =
override-on (*htd s*) (*htd t0′*) (($P \cup A$) − *stack-free* (*htd s*))
using *disj-stack-free-s disj-A t*
by (*auto simp add*: *t0′-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

have *eq-hmem*:
override-on (*hmem s*) (*hmem* t_0) ($A \cup$ *stack-free* (*htd s*)) =
override-on (*hmem s*) (*hmem t0′*) (($P \cup A$) ∪ *stack-free* (*htd s*))
using *disj-stack-free-s disj-A t*
by (*auto simp add*: *t0′-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

from *t* **have** *t-t0′*: *t* = *frame* (($P \cup A$)) *t0′ s*
using *eq-htd eq-hmem*
apply (*simp add*: *frame-def*)
by (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-htd-upd typing.upd-cong*)

have *stack′*: *rel-alloc* $\mathcal{S}$ *M* (($P \cup A$)) *t0′ s t*
using *stack disj-stack-free-s disj-stack-free-s t t-t0′*
by (*auto simp add*: *rel-alloc-def*)

from *disj-stack-free-s disj-stack-free-s stack*
have *stack-free′*: *stack-free* (*htd s*) ∩ ($P \cup A$) $= \{\}$
by (*auto simp add*: *rel-alloc-def*)

from *f-g* [*OF stack′*]
have *f-g′*: *refines f* (*g s*) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (($P \cup A$)) *s t0′ Q*) **.**
then show *?thesis*
by (*simp add*: *t0′-def*)
qed

lemma *refines-rel-stack-override-heap-on-exit-guarded*:

fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: $\bigwedge s\ t\ t_0.\ G \Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M* $(P \cup A)$ t_0 *s t* $\Longrightarrow$
refines f (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t* (*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* t_0 *Q*)
assumes *disj-A*: $G \Longrightarrow P \cap A = \{\}$
assumes *M1-M*: $G \Longrightarrow M1 \subseteq M$
assumes *disj-stack-free-s*: $G \Longrightarrow P \cap$ *stack-free* (*htd s*) $= \{\}$
shows *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* (*override-heap-on P* t_0 *t*) *Q*)
proof −
{assume *guard*: *G*
have *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t* (*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* (*override-heap-on P* t_0 *t*) *Q*)
using *refines-rel-stack-override-heap-on-exit* [*OF stack f-g* [*OF guard*] *disj-A* [*OF guard*]
M1-M [*OF guard*] *disj-stack-free-s* [*OF guard*]]**.}**
then show *?thesis*
by (*rule refines-L2-guard-right''*)
qed

lemma *override-heap-on-merge*:
(*override-heap-on* (*P1* ∪ *P2*) t_0 *t*) = *override-heap-on P2* (*override-heap-on P1* t_0 *t*) *t*
unfolding *override-heap-on-def*
using *override-on-merge*
by (*smt* (*verit*, *ccfv-threshold*) *heap.get-upd heap.lense-axioms*
heap-commute lense.upd-compose lense.upd-cong sup-aci(*1*) *typing.get-upd typing.lense-axioms*)

lemma *refines-rel-stack-override-heap-on-unmodified''*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* ((*P1* ∪ *P2*) ∪ *A*) *s* (*override-heap-on* (*P1* ∪ *P2*) t_0 *t*) *Q*)
assumes *disj-A*: (*P1* ∪ *P2*) ∩ *A* = {}
assumes *unmodified*: *P1* ∩ *M1* = {}
assumes *M1-M*: *M1* ⊆ *M*
assumes *disj-stack-free-s*: (*P1* ∪ *P2*) ∩ *stack-free* (*htd s*) = {}
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1* (*P2* ∪ *A*) *s* (*override-heap-on P2* t_0 *t*) *Q*)
using *f-g*
apply (*auto simp add*: *refines-def-old rel-stack-def*)
subgoal for *r s'*
apply (*erule allE* [**where** *x*=*r*])
apply (*erule allE* [**where** *x*=*s'*])
apply *auto*
subgoal for *x t'*
apply (*rule exI* [**where** *x* = *x*])

apply (*rule exI* [**where** $x = t'$])
apply *auto*
using *disj-A unmodified M1-M stack*
apply (*auto simp add*: *rel-alloc-def*)
subgoal premises *prems*
proof −
have *eq1*:
(λ*a*. *if* (*a* ∈ *P1* ∨ *a* ∈ *P2* ∨ *a* ∈ *A*) ∧ *a* ∉ *stack-free* (*htd s*)
then if a ∈ *P1* ∨ *a* ∈ *P2 then if a* ∈ *A* ∧ *a* ∉ *stack-free* (*htd s*) *then htd* t_0 *a else htd s a*
else htd t_0 *a*
else htd s′ *a*) =
(λ*a*. *if* (*a* ∈ *P2* ∨ *a* ∈ *A*) ∧ *a* ∉ *stack-free* (*htd s*)
then if a ∈ *P2 then if a* ∈ *A* ∧ *a* ∉ *stack-free* (*htd s*) *then htd* t_0 *a else htd s a else htd* t_0 *a*
else htd s′ *a*)
using *prems*
by *fastforce*

have *eq2*:
(λ*a*. *if a* ∈ *P1* ∨ *a* ∈ *P2* ∨ *a* ∈ *A* ∨ *a* ∈ *stack-free* (*htd s*)
then if a ∈ *P1* ∨ *a* ∈ *P2 then if a* ∈ *A* ∨ *a* ∈ *stack-free* (*htd s*) *then hmem* t_0 *a else hmem s a*
else hmem t_0 *a*
else hmem s′ *a*) =
(λ*a*. *if a* ∈ *P2* ∨ *a* ∈ *A* ∨ *a* ∈ *stack-free* (*htd s*)
then if a ∈ *P2 then if a* ∈ *A* ∨ *a* ∈ *stack-free* (*htd s*) *then hmem* t_0 *a else hmem s a else hmem* t_0 *a*
else hmem s′ *a*)
apply (*auto simp add*: *fun-eq-iff*)
using *prems*
by (*simp add*: *equal-upto-def orthD1*)

have *eq3*: ⋀*f g*. *hmem-upd f* (*htd-upd g s*′) = (*hmem-upd* (λ-. *f* (*hmem s*′)) (*htd-upd g s*′))
by (*metis heap.upd-cong hmem-htd-upd*)

have ∗:
hmem-upd
(λ*h a*. *if a* ∈ *P1* ∨ *a* ∈ *P2* ∨ *a* ∈ *A* ∨ *a* ∈ *stack-free* (*htd s*)
then if a ∈ *P1* ∨ *a* ∈ *P2 then if a* ∈ *A* ∨ *a* ∈ *stack-free* (*htd s*) *then hmem* t_0 *a else hmem s a*
else hmem t_0 *a*
else h a)
(*htd-upd*
(λ*d a*. *if* (*a* ∈ *P1* ∨ *a* ∈ *P2* ∨ *a* ∈ *A*) ∧ *a* ∉ *stack-free* (*htd s*)
then if a ∈ *P1* ∨ *a* ∈ *P2 then if a* ∈ *A* ∧ *a* ∉ *stack-free* (*htd s*) *then htd* t_0 *a else htd s a*
else htd t_0 *a*

```
            else d a)
          s') =
         hmem-upd
          (λh a. if a ∈ P2 ∨ a ∈ A ∨ a ∈ stack-free (htd s)
            then if a ∈ P2 then if a ∈ A ∨ a ∈ stack-free (htd s) then hmem t₀ a
else hmem s a else hmem t₀ a
             else h a)
          (htd-upd
            (λd a. if (a ∈ P2 ∨ a ∈ A) ∧ a ∉ stack-free (htd s)
              then if a ∈ P2 then if a ∈ A ∧ a ∉ stack-free (htd s) then htd t₀ a
else htd s a else htd t₀ a
              else d a)
           s')
        apply (subst typing.upd-cong )
         apply (rule eq1)
        apply (subst eq3)
        apply (subst eq2)
        apply (subst (2) eq3)
        apply simp
        done
      show ?thesis
        apply (simp add: override-heap-on-def frame-def)
        apply (auto simp add: override-on-def split: if-splits)
        using ∗
        using prems(7) by auto
    qed
    done
  done
 done

lemma refines-rel-stack-override-heap-on-unmodified':
 fixes p:: 'a::xmem-type ptr
 assumes stack: rel-alloc 𝒮 M A t₀ s t
 assumes f-g: refines f g s t
   (rel-stack 𝒮 M1 (P ∪ A) s (override-heap-on P t₀ t) Q)

 assumes disj-A: P ∩ A = {}
 assumes unmodified: P ∩ M1 = {}
 assumes M1-M: M1 ⊆ M
 assumes disj-stack-free-s: P ∩ stack-free (htd s) = {}
 shows refines f g s t (rel-stack 𝒮 M1 A s t₀ Q)
  using refines-rel-stack-override-heap-on-unmodified'' [where ?P1.0= P and
?P2.0={}] assms
 by simp

lemma refines-rel-stack-override-heap-on-unmodified:
 fixes p:: 'a::xmem-type ptr
 assumes stack: rel-alloc 𝒮 M A t₀ s t
 assumes f-g: refines f g s t
```

(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) *Q*)
assumes *P*: *P* = (*P1* ∪ *P2*)
assumes *disj-A*: *P* ∩ *A* = {}
assumes *unmodified*: *P1* ∩ *M1* = {}
assumes *M1-M*: *M1* ⊆ *M*
assumes *disj-stack-free-s*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1* (*P2* ∪ *A*) *s* (*override-heap-on P2* t_0 *t*) *Q*)
using *refines-rel-stack-override-heap-on-unmodified''* *assms*
by *auto*

lemma *refines-rel-stack-override-heap-on-exit-unmodified*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: ⋀*s t* t_0. *rel-alloc* $\mathcal{S}$ *M* (*P* ∪ *A*) t_0 *s t* ⟹
refines f (*g s*) *s t* (*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* t_0 *Q*)
assumes *P*: *P* = (*P1* ∪ *P2*)
assumes *disj-A*: *P* ∩ *A* = {}
assumes *unmodified*: *P1* ∩ *M1* = {}
assumes *M1-M*: *M1* ⊆ *M*
assumes *disj-stack-free-s*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines f* (*g s*) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P2* ∪ *A*) *s* (*override-heap-on P2* t_0 *t*) *Q*)
proof −
from *refines-rel-stack-override-heap-on-exit* [*OF stack f-g disj-A M1-M disj-stack-free-s*]
have *refines f* (*g s*) *s t* (*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) *Q*).
from *refines-rel-stack-override-heap-on-unmodified* [*OF stack this P disj-A unmodified M1-M disj-stack-free-s*]
show *?thesis* .
qed

lemma *refines-rel-stack-override-heap-on-exit-guarded-unmodified*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: ⋀*s t* t_0. *G* ⟹ *rel-alloc* $\mathcal{S}$ *M* (*P* ∪ *A*) t_0 *s t* ⟹
refines f (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t* (*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* t_0 *Q*)
assumes *P*: *G* ⟹ *P* = (*P1* ∪ *P2*)
assumes *disj-A*: *G* ⟹ *P* ∩ *A* = {}
assumes *unmodified*: *G* ⟹ *P1* ∩ *M1* = {}
assumes *M1-M*: *G* ⟹ *M1* ⊆ *M*
assumes *disj-stack-free-s*: *G* ⟹ *P* ∩ *stack-free* (*htd s*) = {}
shows *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P2* ∪ *A*) *s* (*override-heap-on P2* t_0 *t*) *Q*)
proof −
{assume *guard*: *G*
have *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t* (*rel-stack* $\mathcal{S}$ *M1* (*P2* ∪ *A*) *s* (*override-heap-on P2* t_0 *t*) *Q*)
using *refines-rel-stack-override-heap-on-exit-unmodified* [*OF stack f-g* [*OF*

guard] *P* [*OF guard*]
disj-A [*OF guard*] *unmodified* [*OF guard*]
M1-M [*OF guard*] *disj-stack-free-s* [*OF guard*]]**.}**
then show *?thesis*
by (*rule refines-L2-guard-right''*)
qed

lemma *refines-rel-stack-override-heap-emptyI*:
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) *Q*)
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ *M* (*P* ∪ {} ∪ *A*) *s* (*override-heap-on* (*P* ∪ {}) t_0 *t*) *Q*)
using *assms*
by *simp*

lemma *refines-rel-stack-pop-heap-both*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (*rel-push p L*)) (*rel-push p R*)))
assumes *disj-P-A*: *P* ∩ *A* = {}
assumes *disj-p-A*: *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq*
(*L2-catch g* (λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-throw* (*snd x*) *ns-exit*))))
(λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-return* (*snd x*) *ns*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit L*) *R*))
proof −
from *stack* **have** *sf-s-t*: *stack-free* (*htd s*) ⊆ *stack-free* (*htd t*)
using *rel-alloc-def stack-free-htd-frame* **by** *auto*

from *stack* **have** *t*: *t* = *frame A* t_0 *s*
by (*auto simp add*: *rel-alloc-def*)

define *t0'* **where** *t0'* = *override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*

have *eq-htd*:
override-on (*htd s*) (*htd* t_0) (*A* − *stack-free* (*htd s*)) =
override-on (*htd s*) (*htd t0'*) ((*ptr-span p* ∪ *P* ∪ *A*) − *stack-free* (*htd s*))
using *disj-stack-free-s-p disj-stack-free-s-P disj-P-A disj-p-A t*

by (*auto simp add*: *t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

have *eq-hmem*:
override-on (*hmem s*) (*hmem* t_0) ($A \cup$ *stack-free* (*htd s*)) =
override-on (*hmem s*) (*hmem t0'*) ((*ptr-span p* $\cup$ *P* $\cup$ *A*) $\cup$ *stack-free* (*htd s*))
using *disj-stack-free-s-p disj-stack-free-s-P disj-p-A disj-P-A t*
by (*auto simp add*: *t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

from *t* **have** *t-t0'*: *t* = *frame* ((*ptr-span p* $\cup$ *P* $\cup$ *A*)) *t0' s*
using *eq-htd eq-hmem*
apply (*simp add*: *frame-def*)
by (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-htd-upd typing.upd-cong*)

have *stack'*: *rel-alloc* $\mathcal{S}$ *M* ((*ptr-span p* $\cup$ *P* $\cup$ *A*)) *t0' s t*
using *stack disj-stack-free-s-P disj-stack-free-s-p t t-t0'*
by (*auto simp add*: *rel-alloc-def*)

from *disj-stack-free-s-p disj-stack-free-s-P disj-stack-free-s-P stack*
have *stack-free'*: *stack-free* (*htd s*) $\cap$ (*P* $\cup$ *A*) = {}
by (*auto simp add*: *rel-alloc-def*)

show *?thesis*
unfolding *L2-seq-def*
apply (*rule refines-bindE-right'* [**where** *Q* = λ(*v*, *s'*) (*w*, *t'*).
case v of
Exn e $\Rightarrow$ $\exists$ *w'*. *w* = *Exn w'* $\wedge$ *rel-stack* $\mathcal{S}$ *M1* (*P* $\cup$ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit L*) (λ- - -. *False*)) (*Exn e*, *s'*) (*Exn w'*, *t'*)
| *Result v* $\Rightarrow$ $\exists$ *w'*. *w* = *Result w'* $\wedge$ *rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* $\cup$ *P* $\cup$ *A*) *s* (*override-heap-on* (*ptr-span p* $\cup$ *P*) t_0 *t*) (*rel-xval-stack* (λ- - -. *False*) (*rel-push p R*)) (*Result v*, *s'*) (*Result w'*, *t'*)])
apply (*rule refines-L2-catch-right*)
apply (*rule f-g*)
subgoal by (*clarsimp simp add*: *rel-stack-def t0'-def rel-alloc-def*)
subgoal by *clarsimp*
subgoal by *clarsimp*
subgoal for *e e' s' t'*
apply (*frule rel-stack-unchanged-stack-free* [*OF stack'*, *unfolded t0'-def*])
apply (*frule rel-stack-unchanged-stack-free'* [*OF stack'*, *unfolded t0'-def*])
apply (*subst* (*asm*) *rel-stack-def*)
apply (*clarsimp simp add*: *rel-exit-def*)
apply (*subst* (*asm*) *rel-push-def*)
apply *clarsimp*
apply (*rule refines-exec-IO-modify-heap-padding-single-step-right-InL*)
using *stack*
apply (*clarsimp simp add*: *stack-free' rel-alloc-def rel-stack-def t0'-def* [*symmetric*] *t* [*symmetric*] *t-t0'* [*symmetric*])
subgoal premises *prems* **for** *w x*

proof −
from *prems* **obtain**
$e = Nonlocal\ x$
$e' = (h\text{-}val\ (hmem\ s')\ p,\ w)$
$L\ (hmem\ s')\ x\ w$ **and**
sf-s': $stack\text{-}free\ (htd\ s') = stack\text{-}free\ (htd\ s)$ **and**
sf-t': $stack\text{-}free\ (htd\ (frame\ (ptr\text{-}span\ p \cup P \cup A)\ t0'\ s')) = stack\text{-}free\ (htd\ t)$ **and**
$stack\text{-}free\ (htd\ s) \subseteq \mathcal{S}$
$stack\text{-}free\ (htd\ s) \cap (ptr\text{-}span\ p \cup P \cup A) = \{\}$
$stack\text{-}free\ (htd\ s) \cap M1 = \{\}$
$t' = frame\ (ptr\text{-}span\ p \cup P \cup A)\ t0'\ s'$
$equal\text{-}upto\ (M1 \cup stack\text{-}free\ (htd\ s))\ (hmem\ s')\ (hmem\ s)$ **and**
htd-eq: $htd\ s' = htd\ s$ **by** (*simp*)
have *eq-typing*:
$override\text{-}on\ (htd\ s')\ (htd\ t0')\ (ptr\text{-}span\ p \cup P \cup A - stack\text{-}free\ (htd\ s')) =$
$override\text{-}on\ (htd\ s')\ (htd\ (override\text{-}heap\text{-}on\ P\ t_0\ t))\ (P \cup A - stack\text{-}free\ (htd\ s'))$
using *disj-P-A disj-p-A*
by (*auto simp add*: *override-on-def fun-eq-iff t0'-def htd-eq t htd-frame override-heap-on-def*)

have *eq-heap*: $heap\text{-}update\text{-}padding\ p\ (h\text{-}val\ (hmem\ s')\ p)$
$(heap\text{-}list\ (hmem\ s')\ (size\text{-}of\ TYPE('a))\ (ptr\text{-}val\ p))$
$(override\text{-}on\ (hmem\ s')\ (hmem\ t0')\ (ptr\text{-}span\ p \cup P \cup A \cup stack\text{-}free\ (htd\ s'))) =$
$override\text{-}on\ (hmem\ s')\ (hmem\ (override\text{-}heap\text{-}on\ P\ t_0\ t))\ (P \cup A \cup stack\text{-}free\ (htd\ s'))$
using *disj-P-A disj-p-A*
apply (*simp add*: *override-on-def override-heap-on-def fun-eq-iff t0'-def*)
by (*smt* (*verit*, *best*) *disj-p-P disj-stack-free-s-p h-val-def heap-list-length heap-update-list-id' heap-update-list-value heap-update-padding-def hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id*)

show *hmem-upd*
$(heap\text{-}update\text{-}padding\ p\ (h\text{-}val\ (hmem\ s')\ p)$
$(heap\text{-}list\ (hmem\ s')\ (size\text{-}of\ TYPE('a))\ (ptr\text{-}val\ p)))$
$(frame\ (ptr\text{-}span\ p \cup P \cup A)\ t0'\ s') =$
$frame\ (P \cup A)\ (override\text{-}heap\text{-}on\ P\ t_0\ t)\ s'$
apply (*simp add*: *frame-def comp-def*)
using *eq-typing eq-heap*
by (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-htd-upd typing.upd-cong*)
qed
done
subgoal by *clarsimp*
subgoal by *clarsimp*
subgoal by *clarsimp*
subgoal for $v\ v'\ s'\ t'$

apply (*clarsimp*)
apply (*frule rel-stack-unchanged-stack-free* [*OF stack'*, *unfolded t0'-def*])
apply (*frule rel-stack-unchanged-stack-free'* [*OF stack'*, *unfolded t0'-def*])
apply (*subst* (*asm*) *rel-stack-def*)
apply (*clarsimp simp add*: *rel-exit-def*)
apply (*subst* (*asm*) *rel-push-def*)
apply *clarsimp*
apply (*rule refines-exec-IO-modify-heap-padding-step-right*)
apply (*rule refines-exec-L2-return-right*)
using *stack*
apply (*clarsimp simp add*: *stack-free' rel-alloc-def rel-stack-def t0'-def* [*symmetric*] *t* [*symmetric*] *t-t0'* [*symmetric*])
subgoal premises *prems* **for** *w*
proof −
from *prems* **obtain**
$v' = (h\text{-}val\ (hmem\ s')\ p,\ w)$
$R\ (hmem\ s')\ v\ w$ **and**
sf-s': $stack\text{-}free\ (htd\ s') = stack\text{-}free\ (htd\ s)$ **and**
sf-t': $stack\text{-}free\ (htd\ (frame\ (ptr\text{-}span\ p \cup P \cup A)\ t0'\ s')) = stack\text{-}free\ (htd\ t)$ **and**
$stack\text{-}free\ (htd\ s) \subseteq \mathcal{S}$
$stack\text{-}free\ (htd\ s) \cap (ptr\text{-}span\ p \cup P \cup A) = \{\}$
$stack\text{-}free\ (htd\ s) \cap M1 = \{\}$
$t' = frame\ (ptr\text{-}span\ p \cup P \cup A)\ t0'\ s'$
$equal\text{-}upto\ (M1 \cup stack\text{-}free\ (htd\ s))\ (hmem\ s')\ (hmem\ s)$ **and**
htd-eq: $htd\ s' = htd\ s$ **by** *simp*
have *eq-typing*:
$override\text{-}on\ (htd\ s')\ (htd\ t0')\ (ptr\text{-}span\ p \cup P \cup A - stack\text{-}free\ (htd\ s')) =$
$override\text{-}on\ (htd\ s')\ (htd\ (override\text{-}heap\text{-}on\ P\ t_0\ t))\ (P \cup A - stack\text{-}free\ (htd\ s'))$
using *disj-P-A disj-p-A*
by (*auto simp add*: *override-on-def fun-eq-iff t0'-def htd-eq t htd-frame override-heap-on-def*)

have *eq-heap*: $heap\text{-}update\text{-}padding\ p\ (h\text{-}val\ (hmem\ s')\ p)$
$(heap\text{-}list\ (hmem\ s')\ (size\text{-}of\ TYPE('a))\ (ptr\text{-}val\ p))$
$(override\text{-}on\ (hmem\ s')\ (hmem\ t0')\ (ptr\text{-}span\ p \cup P \cup A \cup stack\text{-}free\ (htd\ s'))) =$
$override\text{-}on\ (hmem\ s')\ (hmem\ (override\text{-}heap\text{-}on\ P\ t_0\ t))\ (P \cup A \cup stack\text{-}free\ (htd\ s'))$
using *disj-P-A disj-p-A*
apply (*simp add*: *override-on-def override-heap-on-def fun-eq-iff t0'-def*)
by (*smt* (*verit*, *best*) *disj-p-P disj-stack-free-s-p h-val-def heap-list-length heap-update-list-id' heap-update-list-value heap-update-padding-def hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id*)

show *hmem-upd*
$(heap\text{-}update\text{-}padding\ p\ (h\text{-}val\ (hmem\ s')\ p)$

```
            (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p)))
           (frame (ptr-span p ∪ P ∪ A) t0' s') =
          frame (P ∪ A) (override-heap-on P t₀ t) s'
       apply (simp add: frame-def comp-def)
       using eq-typing eq-heap
      by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)
    qed
    done
  done
qed

lemma refines-rel-stack-pop-heap-both-guarded:
  fixes p:: 'a::xmem-type ptr
  assumes stack: rel-alloc 𝒮 M A t₀ s t
  assumes f-g: refines f (L2-seq (L2-guard (λ-. G)) (λ-. g)) s t
           (rel-stack 𝒮 M1 (ptr-span p ∪ P ∪ A) s
            (override-heap-on (ptr-span p ∪ P) t₀ t)
            (rel-xval-stack (rel-exit (rel-push p L)) (rel-push p R)))
  assumes disj-P-A: G ⟹ P ∩ A = {}
  assumes disj-p-A: G ⟹ ptr-span p ∩ A = {}
  assumes disj-p-P: G ⟹ ptr-span p ∩ P = {}
  assumes disj-stack-free-s-p: G ⟹ ptr-span p ∩ stack-free (htd s) = {}
  assumes disj-stack-free-s-P: G ⟹ P ∩ stack-free (htd s) = {}
  shows refines
   f
   (L2-seq
    (L2-catch (L2-seq (L2-guard (λ-. G)) (λ-. g))
       (λx. (L2-seq (IO-modify-heap-paddingE p (λ-. (fst x))) (λ-. L2-throw (snd
x) ns-exit))))
     (λx. (L2-seq (IO-modify-heap-paddingE p (λ-. (fst x))) (λ-. L2-return (snd x)
ns)))) s t
   (rel-stack 𝒮 M1 (P ∪ A) s (override-heap-on P t₀ t) (rel-xval-stack (rel-exit L)
R))
proof −
  {
   assume grd: G
   have ?thesis
    by (rule refines-rel-stack-pop-heap-both [OF stack f-g disj-P-A [OF grd] disj-p-A
[OF grd] disj-p-P [OF grd]
          disj-stack-free-s-p [OF grd] disj-stack-free-s-P [OF grd]])
  } then show ?thesis
   by (rule refines-L2-guard-right'''')
qed

lemma refines-rel-stack-pop-heap-both-singleton:
  fixes p:: 'a::xmem-type ptr
  assumes stack: rel-alloc 𝒮 M A t₀ s t
  assumes f-g: refines f g s t
           (rel-stack 𝒮 M1 (ptr-span p ∪ P ∪ A) s
```

(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (*rel-push p L*)) (*rel-singleton-stack p*)))
assumes *disj-P-A*: *P* ∩ *A* = {}
assumes *disj-p-A*: *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq*
(*L2-catch g* (λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-throw* (*snd x*) *ns-exit*))))
(λ*x*. (*IO-modify-heap-paddingE p* (λ-. *x*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit L*) (λ-. (=))))
proof −
from *stack* **have** *sf-s-t*: *stack-free* (*htd s*) ⊆ *stack-free* (*htd t*)
using *rel-alloc-def stack-free-htd-frame* **by** *auto*

from *stack* **have** *t*: *t* = *frame A* t_0 *s*
by (*auto simp add*: *rel-alloc-def*)

define *t0′* **where** *t0′* = *override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*

have *eq-htd*:
override-on (*htd s*) (*htd* t_0) (*A* − *stack-free* (*htd s*)) =
override-on (*htd s*) (*htd t0′*) ((*ptr-span p* ∪ *P* ∪ *A*) − *stack-free* (*htd s*))
using *disj-stack-free-s-p disj-stack-free-s-P disj-P-A disj-p-A t*
by (*auto simp add*: *t0′-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

have *eq-hmem*:
override-on (*hmem s*) (*hmem* t_0) (*A* ∪ *stack-free* (*htd s*)) =
override-on (*hmem s*) (*hmem t0′*) ((*ptr-span p* ∪ *P* ∪ *A*) ∪ *stack-free* (*htd s*))
using *disj-stack-free-s-p disj-stack-free-s-P disj-p-A disj-P-A t*
by (*auto simp add*: *t0′-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

from *t* **have** *t-t0′*: *t* = *frame* ((*ptr-span p* ∪ *P* ∪ *A*)) *t0′ s*
using *eq-htd eq-hmem*
apply (*simp add*: *frame-def*)
by (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-htd-upd typing.upd-cong*)

have *stack′*: *rel-alloc* $\mathcal{S}$ *M* ((*ptr-span p* ∪ *P* ∪ *A*)) *t0′ s t*
using *stack disj-stack-free-s-P disj-stack-free-s-p t t-t0′*
by (*auto simp add*: *rel-alloc-def*)

from *disj-stack-free-s-p disj-stack-free-s-P disj-stack-free-s-P stack*
have *stack-free′*: *stack-free* (*htd s*) ∩ (*P* ∪ *A*) = {}
by (*auto simp add*: *rel-alloc-def*)

show *?thesis*
unfolding *L2-seq-def*
apply (*rule refines-bindE-right'* [**where** $Q = \lambda(v, s')\ (w, t').$
case v of
Exn e $\Rightarrow \exists w'.$ $w =$ *Exn w'* $\wedge$ *rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit L*) (λ- - -. *False*)) (*Exn e*, *s'*) (*Exn w'*, *t'*)
| *Result v* $\Rightarrow \exists w'.$ $w =$ *Result w'* $\wedge$ *rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* $\cup$ *P* $\cup$ *A*) *s* (*override-heap-on* (*ptr-span p* $\cup$ *P*) t_0 *t*) (*rel-xval-stack* (λ- - -. *False*) (*rel-singleton-stack p*)) (*Result v*, *s'*) (*Result w'*, *t'*)])
apply (*rule refines-L2-catch-right*)
apply (*rule f-g*)
subgoal by (*clarsimp simp add*: *rel-stack-def t0'-def rel-alloc-def*)
subgoal by *clarsimp*
subgoal by *clarsimp*
subgoal for *e e' s' t'*
apply (*frule rel-stack-unchanged-stack-free* [*OF stack'*, *unfolded t0'-def*])
apply (*frule rel-stack-unchanged-stack-free'* [*OF stack'*, *unfolded t0'-def*])
apply (*subst* (*asm*) *rel-stack-def*)
apply (*clarsimp simp add*: *rel-exit-def*)
apply (*subst* (*asm*) *rel-push-def*)
apply *clarsimp*
apply (*rule refines-exec-IO-modify-heap-padding-single-step-right-InL*)
using *stack*
apply (*clarsimp simp add*: *stack-free' rel-alloc-def rel-stack-def t0'-def* [*symmetric*] *t* [*symmetric*] *t-t0'* [*symmetric*])
subgoal premises *prems* **for** *w x*
proof −
from *prems* **obtain**
$e =$ *Nonlocal x*
$e' =$ (*h-val* (*hmem s'*) *p*, *w*)
L (*hmem s'*) *x w* **and**
sf-s': *stack-free* (*htd s'*) = *stack-free* (*htd s*) **and**
sf-t': *stack-free* (*htd* (*frame* (*ptr-span p* $\cup$ *P* $\cup$ *A*) *t0' s'*)) = *stack-free* (*htd t*) **and**
stack-free (*htd s*) $\subseteq \mathcal{S}$
stack-free (*htd s*) $\cap$ (*ptr-span p* $\cup$ *P* $\cup$ *A*) = {}
stack-free (*htd s*) $\cap$ *M1* = {}
t' = *frame* (*ptr-span p* $\cup$ *P* $\cup$ *A*) *t0' s'*
equal-upto (*M1* $\cup$ *stack-free* (*htd s*)) (*hmem s'*) (*hmem s*) **and**
htd-eq: *htd s'* = *htd s* **by** (*simp*)
have *eq-typing*:
override-on (*htd s'*) (*htd t0'*) (*ptr-span p* $\cup$ *P* $\cup$ *A* − *stack-free* (*htd s'*)) =
override-on (*htd s'*) (*htd* (*override-heap-on P* t_0 *t*)) (*P* $\cup$ *A* − *stack-free* (*htd s'*))
using *disj-P-A disj-p-A*
by (*auto simp add*: *override-on-def fun-eq-iff t0'-def htd-eq t htd-frame override-heap-on-def*)

have *eq-heap*: *heap-update-padding p* (*h-val* (*hmem s'*) *p*)
(*heap-list* (*hmem s'*) (*size-of TYPE*('*a*)) (*ptr-val p*))
(*override-on* (*hmem s'*) (*hmem t0'*) (*ptr-span p* ∪ *P* ∪ *A* ∪ *stack-free* (*htd s'*))) =
override-on (*hmem s'*) (*hmem* (*override-heap-on P* t_0 *t*)) (*P* ∪ *A* ∪ *stack-free* (*htd s'*))
using *disj-P-A disj-p-A*
apply (*simp add*: *override-on-def override-heap-on-def fun-eq-iff t0'-def*)
by (*smt* (*verit*, *best*) *disj-p-P disj-stack-free-s-p h-val-def heap-list-length heap-update-list-id' heap-update-list-value heap-update-padding-def hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id*)

show *hmem-upd*
(*heap-update-padding p* (*h-val* (*hmem s'*) *p*)
(*heap-list* (*hmem s'*) (*size-of TYPE*('*a*)) (*ptr-val p*)))
(*frame* (*ptr-span p* ∪ *P* ∪ *A*) *t0' s'*) =
frame (*P* ∪ *A*) (*override-heap-on P* t_0 *t*) *s'*
apply (*simp add*: *frame-def comp-def*)
using *eq-typing eq-heap*
by (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-htd-upd typing.upd-cong*)
qed
done
subgoal by *clarsimp*
subgoal by *clarsimp*
subgoal by *clarsimp*
subgoal for *v v' s' t'*
apply (*clarsimp*)
apply (*frule rel-stack-unchanged-stack-free* [*OF stack'*, *unfolded t0'-def*])
apply (*frule rel-stack-unchanged-stack-free'* [*OF stack'*, *unfolded t0'-def*])
apply (*subst* (*asm*) *rel-stack-def*)
apply (*clarsimp simp add*: *rel-exit-def*)
apply (*subst* (*asm*) *rel-singleton-stack-def*)
apply *clarsimp*
apply (*rule refines-exec-IO-modify-heap-padding-single-step-right*)
using *stack*
apply (*clarsimp simp add*: *stack-free' rel-alloc-def rel-stack-def t0'-def* [*symmetric*] *t* [*symmetric*] *t-t0'* [*symmetric*])
subgoal premises *prems* **proof** −
from *prems* **obtain**
v' = *h-val* (*hmem s'*) *p* **and**
sf-s': *stack-free* (*htd s'*) = *stack-free* (*htd s*) **and**
sf-t': *stack-free* (*htd* (*frame* (*ptr-span p* ∪ *P* ∪ *A*) *t0' s'*)) = *stack-free* (*htd t*) **and**
stack-free (*htd s*) ⊆ $\mathcal{S}$
stack-free (*htd s*) ∩ (*ptr-span p* ∪ *P* ∪ *A*) = {}
stack-free (*htd s*) ∩ *M1* = {}
t' = *frame* (*ptr-span p* ∪ *P* ∪ *A*) *t0' s'*

equal-upto (*M1* ∪ *stack-free* (*htd s*)) (*hmem s′*) (*hmem s*) **and**
htd-eq: *htd s′* = *htd s* **by** *simp*
have *eq-typing*:
override-on (*htd s′*) (*htd t0′*) (*ptr-span p* ∪ *P* ∪ *A* − *stack-free* (*htd s′*)) =
override-on (*htd s′*) (*htd* (*override-heap-on P* t_0 *t*)) (*P* ∪ *A* − *stack-free* (*htd s′*))
using *disj-P-A disj-p-A*
by (*auto simp add*: *override-on-def fun-eq-iff t0′-def htd-eq t htd-frame override-heap-on-def*)

have *eq-heap*: *heap-update-padding p* (*h-val* (*hmem s′*) *p*)
(*heap-list* (*hmem s′*) (*size-of TYPE*('*a*)) (*ptr-val p*))
(*override-on* (*hmem s′*) (*hmem t0′*) (*ptr-span p* ∪ *P* ∪ *A* ∪ *stack-free* (*htd s′*))) =
override-on (*hmem s′*) (*hmem* (*override-heap-on P* t_0 *t*)) (*P* ∪ *A* ∪ *stack-free* (*htd s′*))
using *disj-P-A disj-p-A*
apply (*simp add*: *override-on-def override-heap-on-def fun-eq-iff t0′-def*)
by (*smt* (*verit*, *best*) *disj-p-P disj-stack-free-s-p h-val-def heap-list-length heap-update-list-id′ heap-update-list-value heap-update-padding-def hmem-htd-upd max-size orthD2 sf-s′ to-bytes-heap-list-id*)

show *hmem-upd*
(*heap-update-padding p* (*h-val* (*hmem s′*) *p*)
(*heap-list* (*hmem s′*) (*size-of TYPE*('*a*)) (*ptr-val p*)))
(*frame* (*ptr-span p* ∪ *P* ∪ *A*) *t0′ s′*) =
frame (*P* ∪ *A*) (*override-heap-on P* t_0 *t*) *s′*
apply (*simp add*: *frame-def comp-def*)
using *eq-typing eq-heap*
by (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-htd-upd typing.upd-cong*)
qed
done
done
qed

lemma *refines-rel-stack-pop-heap-both-singleton-guarded*:
fixes *p*:: '*a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (*rel-push p L*)) (*rel-singleton-stack p*)))
assumes *disj-P-A*: *G* ⟹ *P* ∩ *A* = {}
assumes *disj-p-A*: *G* ⟹ *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *G* ⟹ *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *G* ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *G* ⟹ *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*

```
    f
    (L2-seq
      (L2-catch (L2-seq (L2-guard (λ-. G)) (λ-. g))
          (λx. (L2-seq (IO-modify-heap-paddingE p (λ-. (fst x))) (λ-. L2-throw (snd
x) ns-exit))))
      (λx. (IO-modify-heap-paddingE p (λ-. x)))) s t
    (rel-stack S M1 (P ∪ A) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit L)
(λ-. (=))))
proof −
  {
    assume grd: G
    have ?thesis
      by (rule refines-rel-stack-pop-heap-both-singleton [OF stack f-g disj-P-A [OF
grd] disj-p-A [OF grd] disj-p-P [OF grd]
          disj-stack-free-s-p [OF grd] disj-stack-free-s-P [OF grd]])
  } then show ?thesis
    by (rule refines-L2-guard-right'''')
qed

lemma refines-rel-stack-pop-heap-no-exit:
  fixes p:: 'a::xmem-type ptr
  assumes stack: rel-alloc S M A t0 s t
  assumes f-g: refines f g s t
            (rel-stack S M1 (ptr-span p ∪ P ∪ A) s
              (override-heap-on (ptr-span p ∪ P) t0 t)
              (rel-xval-stack (rel-exit (λ- - -. False)) (rel-push p R)))
  assumes disj-P-A: P ∩ A = {}
  assumes disj-p-A: ptr-span p ∩ A = {}
  assumes disj-p-P: ptr-span p ∩ P = {}
  assumes disj-stack-free-s-p: ptr-span p ∩ stack-free (htd s) = {}
  assumes disj-stack-free-s-P: P ∩ stack-free (htd s) = {}
  shows refines
    f
    (L2-seq g (λx. (L2-seq (IO-modify-heap-paddingE p (λ-. (fst x))) (λ-. L2-return
(snd x) ns)))) s t
    (rel-stack S M1 (P ∪ A) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit (λ-
- -. False)) R))
proof −
  from stack have sf-s-t: stack-free (htd s) ⊆ stack-free (htd t)
    using rel-alloc-def stack-free-htd-frame by auto

  from stack have t: t = frame A t0 s
    by (auto simp add: rel-alloc-def)

  define t0' where t0' = override-heap-on (ptr-span p ∪ P) t0 t

  have eq-htd:
    override-on (htd s) (htd t0) (A − stack-free (htd s)) =
        override-on (htd s) (htd t0') ((ptr-span p ∪ P ∪ A) − stack-free (htd s))
```

```
    using disj-stack-free-s-p disj-stack-free-s-P disj-P-A disj-p-A t
   by (auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

  have eq-hmem:
    override-on (hmem s) (hmem t_0) (A ∪ stack-free (htd s)) =
        override-on (hmem s) (hmem t0') ((ptr-span p ∪ P ∪ A) ∪ stack-free (htd
s))
    using disj-stack-free-s-p disj-stack-free-s-P disj-p-A disj-P-A t
   by (auto simp add: t0'-def override-heap-on-def override-on-def fun-eq-iff frame-def)

  from t have t-t0': t = frame ((ptr-span p ∪ P ∪ A)) t0' s
    using eq-htd eq-hmem
    apply (simp add: frame-def)
    by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)

  have stack': rel-alloc 𝒮 M ((ptr-span p ∪ P ∪ A)) t0' s t
    using stack disj-stack-free-s-P  disj-stack-free-s-p t t-t0'
    by (auto simp add: rel-alloc-def)

  from disj-stack-free-s-p disj-stack-free-s-P disj-stack-free-s-P stack
  have stack-free': stack-free (htd s) ∩ (P ∪ A) = {}
    by (auto simp add: rel-alloc-def)

  show ?thesis
    unfolding L2-seq-def
    apply (rule refines-bindE-right')
       apply (rule f-g)
      apply simp-all
    subgoal by (simp add: rel-stack-def)
    subgoal for v v' s' t'
      apply (frule rel-stack-unchanged-stack-free [OF stack', unfolded t0'-def])
      apply (frule rel-stack-unchanged-stack-free' [OF stack', unfolded t0'-def])
      apply (subst (asm) rel-stack-def)
      apply clarsimp
      apply (subst (asm) rel-push-def)
      apply clarsimp
      apply (rule refines-exec-IO-modify-heap-padding-step-right)
      apply (rule refines-exec-L2-return-right)
      using stack
      apply (clarsimp simp add: rel-stack-def rel-push-def rel-alloc-def stack-free')
      unfolding t0'-def [symmetric] t [symmetric]
      subgoal premises prems for w
      proof −
        from prems obtain
          v' = (h-val (hmem s') p, w)
          R (hmem s') v w and
          sf-s': stack-free (htd s') = stack-free (htd s) and
         sf-t': stack-free (htd (frame (ptr-span p ∪ P ∪ A) t0' s')) = stack-free (htd
t) and
```

```
      stack-free (htd s) ⊆ 𝒮
      stack-free (htd s) ∩ (ptr-span p ∪ P ∪ A) = {}
      stack-free (htd s) ∩ M1 = {}
      t' = frame (ptr-span p ∪ P ∪ A) t0' s'
      equal-upto (M1 ∪ stack-free (htd s)) (hmem s') (hmem s) and
      htd-eq: htd s' = htd s  by simp

    have eq-typing:
     override-on (htd s') (htd t0') (ptr-span p ∪ P ∪ A − stack-free (htd s')) =
                override-on (htd s') (htd (override-heap-on P t0 t)) (P ∪ A −
stack-free (htd s'))
      using disj-P-A disj-p-A
       by (auto simp add: override-on-def fun-eq-iff t0'-def htd-eq t htd-frame
override-heap-on-def)


    have eq-heap: heap-update-padding p (h-val (hmem s') p)
                (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p))
              (override-on (hmem s') (hmem t0') (ptr-span p ∪ P ∪ A ∪ stack-free
(htd s'))) =
               override-on (hmem s') (hmem (override-heap-on P t0 t)) (P ∪ A ∪
stack-free (htd s'))
      using disj-P-A disj-p-A
      apply (simp add: override-on-def override-heap-on-def fun-eq-iff t0'-def)
      by (smt (verit, best) disj-p-P disj-stack-free-s-p h-val-def heap-list-length
          heap-update-list-id' heap-update-list-value heap-update-padding-def
          hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id)

    show hmem-upd
          (heap-update-padding p (h-val (hmem s') p)
            (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p)))
          (frame (ptr-span p ∪ P ∪ A) t0' s') =
          frame (P ∪ A) (override-heap-on P t0 t) s'
      apply (simp add: frame-def comp-def)
      using eq-typing eq-heap
     by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)
  qed
  done
 done
qed


lemma refines-rel-stack-pop-heap-no-exit-guarded:
 fixes p:: 'a::xmem-type ptr
 assumes stack: rel-alloc 𝒮 M A t0 s t
 assumes f-g: refines f (L2-seq (L2-guard (λ-. G)) (λ-. g)) s t
         (rel-stack 𝒮 M1 (ptr-span p ∪ P ∪ A) s
           (override-heap-on (ptr-span p ∪ P) t0 t)
           (rel-xval-stack (rel-exit (λ- - -. False)) (rel-push p R)))
```

```
  assumes disj-P-A: G ⟹ P ∩ A = {}
  assumes disj-p-A: G ⟹ ptr-span p ∩ A = {}
  assumes disj-p-P: G ⟹ ptr-span p ∩ P = {}
  assumes disj-stack-free-s-p: G ⟹ ptr-span p ∩ stack-free (htd s) = {}
  assumes disj-stack-free-s-P: G ⟹ P ∩ stack-free (htd s) = {}
  shows refines
    f
   (L2-seq (L2-seq (L2-guard (λ-. G)) (λ-. g)) (λx. (L2-seq (IO-modify-heap-paddingE
p (λ-. (fst x))) (λ-. L2-return (snd x) ns)))) s t
   (rel-stack S M1 (P ∪ A) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit (λ-
- -. False)) R))
proof −
  {
    assume grd: G
    have ?thesis
      by (rule refines-rel-stack-pop-heap-no-exit [OF stack f-g disj-P-A [OF grd]
disj-p-A [OF grd] disj-p-P [OF grd]
          disj-stack-free-s-p [OF grd] disj-stack-free-s-P [OF grd]])
  } then show ?thesis
    by (rule refines-L2-guard-right''')
qed

lemma refines-rel-stack-pop-heap-no-exit-singleton:
  fixes p:: 'a::xmem-type ptr
  assumes stack: rel-alloc S M A t0 s t
  assumes f-g: refines f g s t
            (rel-stack S M1 (ptr-span p ∪ P ∪ A) s
              (override-heap-on (ptr-span p ∪ P) t0 t)
              (rel-xval-stack (rel-exit (λ- - -. False)) (rel-singleton-stack p)))
  assumes disj-P-A: P ∩ A = {}
  assumes disj-p-A: ptr-span p ∩ A = {}
  assumes disj-p-P: ptr-span p ∩ P = {}
  assumes disj-stack-free-s-p: ptr-span p ∩ stack-free (htd s) = {}
  assumes disj-stack-free-s-P: P ∩ stack-free (htd s) = {}
  shows refines
    f
    (L2-seq g (λx. (IO-modify-heap-paddingE p (λ-. x)))) s t
   (rel-stack S M1 (P ∪ A) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit (λ-
- -. False)) (λ-. (=))))
proof −
  from stack have sf-s-t: stack-free (htd s) ⊆ stack-free (htd t)
    using rel-alloc-def stack-free-htd-frame by auto

  from stack have t: t = frame A t0 s
    by (auto simp add: rel-alloc-def)

  define t0' where t0' = override-heap-on (ptr-span p ∪ P) t0 t

  have eq-htd:
```

override-on (*htd s*) (*htd* t_0) (*A* − *stack-free* (*htd s*)) =
override-on (*htd s*) (*htd t0′*) ((*ptr-span p* ∪ *P* ∪ *A*) − *stack-free* (*htd s*))
using *disj-stack-free-s-p disj-stack-free-s-P disj-P-A disj-p-A t*
by (*auto simp add*: *t0′-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

have *eq-hmem*:
override-on (*hmem s*) (*hmem* t_0) (*A* ∪ *stack-free* (*htd s*)) =
override-on (*hmem s*) (*hmem t0′*) ((*ptr-span p* ∪ *P* ∪ *A*) ∪ *stack-free* (*htd s*))
using *disj-stack-free-s-p disj-stack-free-s-P disj-p-A disj-P-A t*
by (*auto simp add*: *t0′-def override-heap-on-def override-on-def fun-eq-iff frame-def*)

from *t* **have** *t-t0′*: *t* = *frame* ((*ptr-span p* ∪ *P* ∪ *A*)) *t0′ s*
using *eq-htd eq-hmem*
apply (*simp add*: *frame-def*)
by (*metis* (*no-types*, *lifting*) *heap.upd-cong hmem-htd-upd typing.upd-cong*)

have *stack′*: *rel-alloc* 𝒮 *M* ((*ptr-span p* ∪ *P* ∪ *A*)) *t0′ s t*
using *stack disj-stack-free-s-P disj-stack-free-s-p t t-t0′*
by (*auto simp add*: *rel-alloc-def*)

from *disj-stack-free-s-p disj-stack-free-s-P disj-stack-free-s-P stack*
have *stack-free′*: *stack-free* (*htd s*) ∩ (*P* ∪ *A*) = {}
by (*auto simp add*: *rel-alloc-def*)

show *?thesis*
unfolding *L2-seq-def*
apply (*rule refines-bindE-right′*)
apply (*rule f-g*)
apply *simp-all*
subgoal by (*simp add*: *rel-stack-def*)
subgoal for *v′ s′ t′*
apply (*frule rel-stack-unchanged-stack-free* [*OF stack′*, *unfolded t0′-def*])
apply (*frule rel-stack-unchanged-stack-free′* [*OF stack′*, *unfolded t0′-def*])
apply (*subst* (*asm*) *rel-stack-def*)
apply *clarsimp*
apply (*subst* (*asm*) *rel-singleton-stack-def*)
apply *clarsimp*
apply (*rule refines-exec-IO-modify-heap-padding-single-step-right*)
using *stack*
apply (*clarsimp simp add*: *rel-stack-def rel-push-def rel-alloc-def stack-free′*)
unfolding *t0′-def* [*symmetric*] *t* [*symmetric*]
subgoal premises *prems*
proof −
from *prems* **obtain**
v′ = *h-val* (*hmem s′*) *p* **and**
sf-s′: *stack-free* (*htd s′*) = *stack-free* (*htd s*) **and**
sf-t′: *stack-free* (*htd* (*frame* (*ptr-span p* ∪ *P* ∪ *A*) *t0′ s′*)) = *stack-free* (*htd t*) **and**

```
        stack-free (htd s) ⊆ 𝒮
        stack-free (htd s) ∩ (ptr-span p ∪ P ∪ A) = {}
        stack-free (htd s) ∩ M1 = {}
        t' = frame (ptr-span p ∪ P ∪ A) t0' s'
        equal-upto (M1 ∪ stack-free (htd s)) (hmem s') (hmem s) and
        htd-eq: htd s' = htd s  by simp

      have eq-typing:
       override-on (htd s') (htd t0') (ptr-span p ∪ P ∪ A − stack-free (htd s')) =
                override-on (htd s') (htd (override-heap-on P t0 t)) (P ∪ A −
stack-free (htd s'))
        using disj-P-A disj-p-A
         by (auto simp add: override-on-def fun-eq-iff t0'-def htd-eq t htd-frame
override-heap-on-def)


      have eq-heap: heap-update-padding p (h-val (hmem s') p)
                 (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p))
               (override-on (hmem s') (hmem t0') (ptr-span p ∪ P ∪ A ∪ stack-free
(htd s'))) =
                override-on (hmem s') (hmem (override-heap-on P t0 t)) (P ∪ A ∪
stack-free (htd s'))
        using disj-P-A disj-p-A
        apply (simp add: override-on-def override-heap-on-def fun-eq-iff t0'-def)
        by (smt (verit, best) disj-p-P disj-stack-free-s-p h-val-def heap-list-length
            heap-update-list-id' heap-update-list-value heap-update-padding-def
            hmem-htd-upd max-size orthD2 sf-s' to-bytes-heap-list-id)

      show hmem-upd
            (heap-update-padding p (h-val (hmem s') p)
              (heap-list (hmem s') (size-of TYPE('a)) (ptr-val p)))
            (frame (ptr-span p ∪ P ∪ A) t0' s') =
           frame (P ∪ A) (override-heap-on P t0 t) s'
        apply (simp add: frame-def comp-def)
        using eq-typing eq-heap
       by (metis (no-types, lifting) heap.upd-cong hmem-htd-upd typing.upd-cong)
    qed
    done
  done
qed

lemma refines-rel-stack-pop-heap-no-exit-singleton-guarded:
  fixes p:: 'a::xmem-type ptr
  assumes stack: rel-alloc 𝒮 M A t0 s t
  assumes f-g: refines f (L2-seq (L2-guard (λ-. G)) (λ-. g)) s t
           (rel-stack 𝒮 M1 (ptr-span p ∪ P ∪ A) s
             (override-heap-on (ptr-span p ∪ P) t0 t)
             (rel-xval-stack (rel-exit (λ- - -. False)) (rel-singleton-stack p)))
  assumes disj-P-A: G ⟹ P ∩ A = {}
```

```
  assumes disj-p-A: G ⟹ ptr-span p ∩ A = {}
  assumes disj-p-P: G ⟹ ptr-span p ∩ P = {}
  assumes disj-stack-free-s-p: G ⟹ ptr-span p ∩ stack-free (htd s) = {}
  assumes disj-stack-free-s-P: G ⟹ P ∩ stack-free (htd s) = {}
  shows refines
    f
     (L2-seq (L2-seq (L2-guard (λ-. G)) (λ-. g)) (λx. (IO-modify-heap-paddingE p
(λ-. x)))) s t
    (rel-stack S M1 (P ∪ A) s (override-heap-on P t0 t) (rel-xval-stack (rel-exit (λ-
- -. False)) (λ-. (=))))
proof −
  {
    assume grd: G
    have ?thesis
     by (rule refines-rel-stack-pop-heap-no-exit-singleton [OF stack f-g disj-P-A [OF
grd] disj-p-A [OF grd] disj-p-P [OF grd]
           disj-stack-free-s-p [OF grd] disj-stack-free-s-P [OF grd]])
  } then show ?thesis
    by (rule refines-L2-guard-right''')
qed

lemma refines-rel-stack-shuffle-both:
  assumes refines f g s t (rel-stack S M A s t0 (rel-xval-stack (rel-exit L) R))
  assumes ⋀h e e'. L h e e' ⟹ L' h e (shuffle-exit e')
  assumes ⋀h v v'. R h v v' ⟹ R' h v (shuffle v')
  shows refines
    f
    (L2-seq
      (L2-catch g (λe. L2-throw (shuffle-exit e) ns-exit))
      (λx. L2-return (shuffle x) ns))
    s t
    (rel-stack S M A s t0 (rel-xval-stack (rel-exit L') R'))
  using assms
 apply (clarsimp simp add: reaches-bind reaches-catch succeeds-bind succeeds-catch
L2-defs refines-def-old L2-VARS-def
    rel-stack-def rel-alloc-def rel-xval-stack-def rel-exit-def rel-xval.simps default-option-def
Exn-def[symmetric]
     split: xval-splits prod.splits)
  subgoal for r s'
  apply (cases r)
    subgoal
      apply (clarsimp simp add: default-option-def  Exn-def[symmetric])
      subgoal for y

        apply (erule-tac x=Exn y in allE)
        apply (erule-tac x=s' in allE)
        apply clarsimp
      by (smt (verit) Exn-def Exn-eq-Exn Result-neq-Exn case-exception-or-result-Exn)
      done
```

```
    subgoal for v
       apply (erule-tac x=Result v in allE)
       apply (erule-tac x=s' in allE)
       apply clarsimp
      by (smt (z3) Result-eq-Result Result-neq-Exn case-exception-or-result-Result)
    done
  done

lemma refines-rel-stack-shuffle-no-exit:
   assumes refines f g s t (rel-stack S M A s t₀ (rel-xval-stack (rel-exit (λ- - -. False)) R))
  assumes ⋀h v v'. R h v v' ⟹ R' h v (shuffle v')
  shows refines
    f
    (L2-seq
      g
      (λx. L2-return (shuffle x) ns))
    s t
    (rel-stack S M A s t₀ (rel-xval-stack (rel-exit (λ- - -. False)) R'))
  using assms
   apply (clarsimp simp add: succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def
        rel-stack-def rel-alloc-def rel-sum-stack-def rel-exit-def rel-sum.simps split: sum.splits prod.splits)
  subgoal for r s'
  apply (cases r)
    subgoal
      apply (clarsimp simp add: default-option-def  Exn-def[symmetric])
      subgoal for y

        apply (erule-tac x=Exn y in allE)
        apply (erule-tac x=s' in allE)
        apply auto
        done
      done
    subgoal for v
       apply (erule-tac x=Result v in allE)
       apply (erule-tac x=s' in allE)
       apply clarsimp
      by (smt (z3) Result-eq-Result Result-neq-Exn case-exception-or-result-Result)
    done

lemma L2-call-rel-stack-bare':
  assumes f: refines f g s t
              (rel-stack S M1 A s t₀ (rel-xval-stack L R))
  shows refines
          f
          (L2-call g (λx. x) ns')
```

s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
using *assms*
by (*auto simp add*: *L2-call-def refines-def-old reaches-map-value*)

lemma *L2-call-rel-stack-bare-retype-unreachable-exit'*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
shows *refines*
f
(*L2-call g emb ns'*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
using *assms*
apply (*clarsimp simp add*: *L2-call-def reaches-map-value succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def*
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
by *fastforce*

lemma *L2-call-rel-stack-bare-retype-unreachable-exit''*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g emb ns*)
(λ*x*. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λ*v*. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
using *assms*
apply (*clarsimp simp add*: *L2-call-def reaches-map-value succeeds-catch reaches-catch succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def*
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
by *fastforce*

lemma *L2-call-rel-stack-bare-retype-unreachable-exit*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g undefined ns*)
(λ*x*. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λ*v*. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
by (*rule L2-call-rel-stack-bare-retype-unreachable-exit''* [*OF f*])

lemma *L2-call-rel-stack-bare-retype-unreachable-exit-extend-modifies'*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
assumes *P* ⟹ *stack-free* (*htd s*) ∩ *M2* = {}
assumes *M1-M2*: *P* ⟹ *M1* ⊆ *M2*
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g emb ns*)
(λ*x*. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λ*v*. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
using *assms*
apply (*clarsimp simp add*: *L2-call-def reaches-map-value succeeds-catch reaches-catch succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def*
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
subgoal for *r s'*
apply (*cases r*)
subgoal
apply (*clarsimp simp add*: *Exn-def* [*symmetric*] *default-option-def*)
by (*metis Result-neq-Exn*)
subgoal for *v*
apply (*erule-tac x*=*Result v* **in** *allE*)
apply (*erule-tac x*=*s'* **in** *allE*)
apply (*clarsimp simp add*: *Exn-def* [*symmetric*] *default-option-def*)
by (*smt* (*z3*) *case-xval-simps*(*2*) *equal-upto-mono map-exn-simps*(*2*)
order-eq-refl sup-mono)
done
done

lemma *L2-call-rel-stack-bare-retype-unreachable-exit-extend-modifies*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
assumes *sf*: *P* ⟹ *stack-free* (*htd s*) ∩ *M2* = {}
assumes *M1-M2*: *P* ⟹ *M1* ⊆ *M2*
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g undefined ns*)
(λ*x*. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λ*v*. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
by (*rule L2-call-rel-stack-bare-retype-unreachable-exit-extend-modifies'* [*OF f sf M1-M2*])

lemma *L2-call-rel-stack-bare*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g* (λ*x*. *x*) *ns*)
(λ*x*. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λ*v*. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
using *assms*
by (*clarsimp simp add*: *L2-call-def reaches-map-value succeeds-catch reaches-catch succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def*)

lemma *L2-call-rel-stack-bare-extend-modifies*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
assumes *sf*: *P* ⟹ *stack-free* (*htd s*) ∩ *M2* = {}
assumes *M1-M2*: *P* ⟹ *M1* ⊆ *M2*
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g* (λ*x*. *x*) *ns*)
(λ*x*. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λ*v*. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
unfolding *L2-defs L2-VARS-def liftE-return*
apply (*clarsimp simp add*: *f-catch-throw L2-call-def refines-bind-guard-right-iff*)
apply (*rule refines-weaken* [*OF f*])
using *sf M1-M2*
apply (*clarsimp simp add*: *rel-stack-def rel-alloc-def*)
by (*meson Un-subset-iff Un-upper2 equal-upto-mono sup.coboundedI1*)

lemma *refines-rel-stack-extend-modifies*:
assumes *f*: *refines f* (*L2-seq* (*L2-guard* (λ-. *P*)) (λ-. *g*)) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
assumes *sf*: *P* ⟹ *M2* ∩ *stack-free* (*htd s*) = {}
assumes *M1-M2*: *P* ⟹ *M1* ⊆ *M2*
shows *refines f* (*L2-seq* (*L2-guard* (λ-. *P*)) (λ-. *g*)) *s t* (*rel-stack* $\mathcal{S}$ *M2 A s* t_0 *Q*)
using *assms*
apply (*clarsimp simp add*: *L2-call-def reaches-map-value succeeds-catch reaches-catch succeeds-bind reaches-bind L2-defs refines-def-old L2-VARS-def rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def*)
by (*smt* (*verit*, *best*) *dual-order.refl equal-upto-mono inf-aci*(1) *sup.mono*)

lemma *refines-rel-sum-stack-pop-exit'*:
assumes *f-g*:
refines f g s t
(*rel-stack* $\mathcal{S}$ *M A s* t_0
(*rel-xval-stack*
(*rel-exit* (*rel-push p L*))
R))
shows
refines f (*L2-catch g* (λ(*x*, *p*). *L2-throw p ns*)) *s t*
(*rel-stack* $\mathcal{S}$ *M A s* t_0
(*rel-xval-stack*
(*rel-exit L*)
R))
apply (*rule refines-L2-catch-right*)
apply (*rule f-g*)
apply *simp-all*
apply (*auto simp add*: *L2-call-def reaches-map-value succeeds-catch reaches-catch*

succeeds-bind reaches-bind L2-defs refines-def-old default-option-def Exn-def [*symmetric*] *L2-VARS-def*
rel-push-def
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
done

lemma *refines-rel-sum-stack-pop-exit*:
assumes *f-g*:
refines f g s t
(*rel-stack* $\mathcal{S}$ *M A s* t_0
(*rel-xval-stack*
(*rel-exit* (*rel-push p L*))
R))
shows
refines f (*L2-catch g* (λ*x*. *L2-throw* (*snd x*) *ns*)) *s t*
(*rel-stack* $\mathcal{S}$ *M A s* t_0
(*rel-xval-stack*
(*rel-exit L*)
R))
apply (*rule refines-L2-catch-right*)
apply (*rule f-g*)
apply *simp-all*
apply (*auto simp add*: *L2-call-def reaches-map-value succeeds-catch reaches-catch*

succeeds-bind reaches-bind L2-defs refines-def-old default-option-def Exn-def [*symmetric*] *L2-VARS-def*
rel-push-def
rel-stack-def rel-alloc-def rel-xval-stack-def rel-xval.simps rel-exit-def)
done

end

lemma *L2-call-fuse-handlers1*:
L2-seq
(*L2-catch*
(*L2-seq*
(*L2-catch g* (λ*x*. *L2-seq* (*liftE* (*h1 x*)) (λ-. *L2-throw* (*prj1 x*) *ns1*)))
(λ*x*. *liftE* (*g1 x*)))
(λ*x*. *L2-seq* (*liftE* (*h2 x*)) (λ-. *L2-throw* (*prj2 x*) *ns2*)))
(λ*x*. *liftE* (*g2 x*)) =
(*L2-seq*
(*L2-catch g* (λ*x*. *L2-seq* (*L2-seq* (*liftE* (*h1 x*)) (λ-. *liftE* (*h2* (*prj1 x*)))) (λ-. *L2-throw* (*prj2* (*prj1 x*)) *ns2*)))
(λ*x*. *L2-seq* (*liftE* (*g1 x*)) (λ*x*. *liftE* (*g2 x*))))
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-call-fuse-handlers2*:
(*L2-catch*
(*L2-seq*
(*L2-catch g* (λ*x*. *L2-seq* (*liftE* (*h1 x*)) (λ-. *L2-throw* (*prj1 x*) *ns1*)))
(λ*x*. *liftE* (*g1 x*)))
(λ*x*. *L2-throw* (*prj2 x*) *ns2*))
=
L2-seq
(*L2-catch g* (λ*x*. *L2-seq* (*liftE* (*h1 x*)) (λ-. *L2-throw* (*prj2* (*prj1 x*)) *ns2*)))
(λ*x*. (*liftE* (*g1 x*)))
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*auto simp add*: *runs-to-def-old default-option-def Exn-def split*: *xval-splits*)
done

lemma *L2-call-triv-conv*: *L2-call f* (λ*x*. *x*) *ns* = *f*
unfolding *L2-call-def*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-catch-L2-call-conv*:
L2-catch
(*L2-call f* (λ*e*. *e*) *ns*)
(λ*e*. *L2-throw* (*emb e*) *ns*′) =
L2-call f (*ETA-TUPLED emb*) *ns*
apply (*simp add*: *L2-call-triv-conv ETA-TUPLED-def*)

apply (*simp add*: *L2-call-def map-exn-catch-conv L2-catch-def L2-throw-def comp-def*)
done

lemma *L2-seq-return-conv*:
L2-seq (*liftE* (*return x*)) (λ*x*. *liftE* (*return* (*f x*))) = *liftE* (*return* (*f x*))
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-return-unused-conv*:
L2-seq (*liftE* (*return x*)) (λ-. *g*) = *g*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-L2-return-unused-conv*:
L2-seq (*L2-return x ns*) (λ-. *g*) = *g*
unfolding *L2-defs L2-VARS-def*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-return-id-conv*:
L2-seq f (λ*x*. *liftE* (*return x*)) = *f*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-L2-return-id-conv*:
L2-seq f (λ*x*. *L2-return x ns*) = *f*
unfolding *L2-defs L2-VARS-def*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-catch-L2-guard-out-conv*: *L2-catch* (*L2-seq* (*L2-guard P*) (λ-. *X*)) *Y* = *L2-seq* (*L2-guard P*) (λ-. *L2-catch X Y*)
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *L2-seq-guard-out-conv*:

L2-seq (*L2-seq* (*L2-guard P*) (λ-. *X*)) *Y* = (*L2-seq* (*L2-guard P*) (λ-. *L2-seq X Y*))
by (*simp add*: *L2-seq-assoc*)

lemma *L2-seq-L2-catch-assoc*:
L2-seq (*L2-seq* (*L2-catch X Y*) *Z1*) *Z2* = *L2-seq* (*L2-catch X Y*) (λ*x*. *L2-seq* (*Z1 x*) *Z2*)
by (*rule L2-seq-assoc*)

lemma *L2-catch-throw-id*: *L2-catch f* (λ*x*. *L2-throw x ns*) = *f*
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
done

lemma *L2-catch-call-throw*:
(*L2-catch*
(*L2-call f emb ns*)
(λ*x*. *L2-throw* (*g x*) *ns-exit*)) =
L2-call f (*ETA-TUPLED* (λ*x*. (*g* (*emb x*)))) *ns*
unfolding *L2-defs L2-call-def ETA-TUPLED-def*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*auto simp add*: *runs-to-def-old map-exn-def split*: *xval-splits*)
done

lemma *liftE-bind-L2-seq*: (*liftE* (*A* >>= *B*)) = *L2-seq* (*liftE A*) (*ETA-TUPLED* (λ*x*. *liftE* (*B x*)))
unfolding *L2-defs L2-call-def ETA-TUPLED-def*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *liftE-L2-seq*: *L2-seq* (*liftE A*) (λ*x*. *liftE* (*B x*)) = (*liftE* (*A* >>= *B*))
unfolding *L2-defs*
by (*simp add*: *liftE-bind*)

lemma *liftE-return-L2-gets-conv*: *liftE* (*return x*) = *L2-gets* (λ-. *x*) []
by (*simp add*: *return-L2-gets-conv*)

lemma *L2-call-shuffle-conv*:
L2-call (*L2-seq* (*L2-catch f A*) *B*) (λ*x*. *x*) *ns* =
(*L2-seq* (*L2-catch* (*L2-call f* (λ*x*. *x*) *ns*) *A*) *B*)
unfolding *L2-call-def L2-defs*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemmas *L2-call-canonical-convs* =
L2-call-shuffle-conv
L2-seq-return-conv
L2-call-fuse-handlers2
L2-call-fuse-handlers1
L2-guard-join-nested
liftE-L2-seq

L2-seq-L2-catch-assoc
L2-catch-L2-guard-out-conv
L2-seq-guard-out-conv

bind-assoc

lemmas *L2-call-pre-final-convs* =
L2-seq-return-unused-conv
L2-seq-L2-return-unused-conv
L2-seq-return-id-conv
L2-seq-L2-return-id-conv
L2-catch-call-throw

lemmas *L2-call-final-convs* =
L2-seq-return-unused-conv
L2-seq-L2-return-unused-conv
L2-seq-return-id-conv
L2-seq-L2-return-id-conv
L2-catch-L2-call-conv

liftE-bind-L2-seq
L2-gets-unbound
L2-catch-throw-id
L2-return-L2-gets-conv
liftE-return-L2-gets-conv

L2-seq-L2-catch-assoc
L2-catch-L2-guard-out-conv
L2-seq-guard-out-conv

lemma *L2-call-ETA-TUPLED1*: *L2-seq* (*L2-catch g h*) *g1* ≡ *L2-seq* (*L2-catch g* (*ETA-TUPLED h*)) (*ETA-TUPLED g1*)
unfolding *ETA-TUPLED-def* **by** *simp*

lemma *L2-call-ETA-TUPLED2*: *L2-seq* (*L2-call g emb ns*) *g1* ≡ *L2-seq* (*L2-call g emb ns*) (*ETA-TUPLED g1*)
unfolding *ETA-TUPLED-def* **by** *simp*

lemma *distinct-sets-consI*: *distinct-sets ps* ⟹ *distinct-sets* (*p*#*ps*) ⟷ *p* ∩ ⋃ (*set ps*) = {}

by (*simp add*: *distinct-sets.simps*)

lemma *disjoint-subset-simps*:
ASSUMPTION $(A \cap B = \{\}) \Longrightarrow A' \subseteq A \Longrightarrow B \cap A' = \{\} \longleftrightarrow True$
ASSUMPTION $(B \cap A = \{\}) \Longrightarrow A' \subseteq A \Longrightarrow B \cap A' = \{\} \longleftrightarrow True$
ASSUMPTION $(A \cap B = \{\}) \Longrightarrow B' \subseteq B \Longrightarrow B' \cap A = \{\} \longleftrightarrow True$
ASSUMPTION $(B \cap A = \{\}) \Longrightarrow B' \subseteq B \Longrightarrow B' \cap A = \{\} \longleftrightarrow True$
by (*auto simp add*: *ASSUMPTION-def*)

lemma *disjoint-subset-simps'*:
$(B \cap A = \{\}) \Longrightarrow A' \subseteq A \Longrightarrow B \cap A' = \{\} \longleftrightarrow True$
$(A \cap B = \{\}) \Longrightarrow B' \subseteq B \Longrightarrow B' \cap A = \{\} \longleftrightarrow True$
$(B \cap A = \{\}) \Longrightarrow B' \subseteq B \Longrightarrow B' \cap A = \{\} \longleftrightarrow True$
$(A \cap B = \{\}) \Longrightarrow A' \subseteq A \Longrightarrow B \cap A' = \{\} \longleftrightarrow True$
by *auto*

lemma *field-lvalue-ptr-span-trans*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*)
assumes *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*b*))
assumes *ptr-span p* ⊆ *A*
shows {&(*p*→*f*)..+*size-of TYPE*('*b*::*c-type*)} ⊆ *A*
using *field-tag-sub assms export-size-of* **by** *fastforce*

lemma *field-lvalue-ptr-span-root-contained*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*)
assumes *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*b*))
shows {&(*p*→*f*)..+*size-of TYPE*('*b*::*c-type*)} ⊆ *ptr-span p*
using *field-tag-sub assms export-size-of* **by** *fastforce*

lemma *field-lvalue-disjoint-fields-same-root*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *f*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*T1*, *n*)
assumes *exp1*: *export-uinfo T1* = *typ-uinfo-t* (*TYPE*('*b*::*c-type*))
assumes *g*: *field-lookup* (*typ-info-t TYPE*('*a*)) *g 0* = *Some* (*T2*, *m*)
assumes *exp2*: *export-uinfo T2* = *typ-uinfo-t* (*TYPE*('*c*::*c-type*))
assumes *prfx1*: ¬ *prefix f g*
assumes *prfx2*: ¬ *prefix g f*
shows {&(*p*→*f*)..+*size-of TYPE*('*b*::*c-type*)} ∩ {&(*p*→*g*)..+*size-of TYPE*('*c*::*c-type*)}
= {}
using *field-lookup-non-prefix-disj* [*OF - f g prfx2 prfx1*]
field-lookup-offset-eq [*OF f*] *field-lookup-offset-eq* [*OF g*] *exp1 exp2*
export-size-of [*OF exp1*] *export-size-of* [*OF exp2*]
apply *simp*
by (*smt* (*verit*, *best*) *add.commute add-diff-cancel-left'*
diff-is-0-eq dual-order.trans export-size-of f field-lookup-offset-size
field-lookup-wf-size-desc-gt field-lvalue-def fold-typ-uinfo-t g intvl-fields-disj
linorder-not-less max-size nat-less-le unat-of-nat-field-offset wf-size-desc)

lemma *ptr-coerce-index-array-ptr-index-conv*:
ptr-coerce p +$_p$ uint i = array-ptr-index p False (nat (uint i))
by (*auto simp add*: *array-ptr-index-def*)

lemma *ptr-coerce-index-array-ptr-index-numeral-conv*:
ptr-coerce p +$_p$ (numeral i) = array-ptr-index p False (numeral i)
by (*auto simp add*: *array-ptr-index-def*)

lemma *ptr-coerce-index-array-ptr-index-numeral-conv'*:
fixes *p*:: ((*'a* :: *c-type*)[*'b* :: *finite*]) *ptr*
assumes *sz-eq*: *size-of TYPE('c*::*c-type*) = *size-of TYPE('a)*
shows *PTR-COERCE('a['b] → 'c*::*c-type) p +$_p$ (numeral n) = PTR-COERCE('a → 'c) (array-ptr-index p False (numeral n))*
using *assms*
by (*simp add*: *array-ptr-index-simps*(*1*) *c-type-class.ptr-add-def*)

lemma *ptr-coerce-index-array-ptr-index-numeral-conv''*:
fixes *p*:: ((*'a* :: *c-type*)[*'b* :: *finite*]) *ptr*
assumes *sz-eq*: *size-of TYPE('c*::*c-type*) = *size-of TYPE('a)*
assumes *bound*: *numeral n < CARD('b)*
shows *PTR-COERCE('a['b] → 'c*::*c-type) p +$_p$ (numeral n) = PTR('c) &(p→[replicate (numeral n) CHR ''1''])*
proof −
have *eq1*: *PTR('c) &(p→[replicate (numeral n) CHR ''1'']) = PTR-COERCE('a → 'c) (PTR('a) &(p→[replicate (numeral n) CHR ''1'']))*
by *simp*
note *replicate-numeral* [*simp del*]
show *?thesis*
apply (*subst eq1*)
apply (*subst array-ptr-index-field-lvalue-conv* [*symmetric*, *OF bound*])
apply (*rule ptr-coerce-index-array-ptr-index-numeral-conv'* [*OF sz-eq*])
done
qed

lemma *ptr-coerce-index-array-ptr-index-0-conv*:
ptr-coerce p +$_p$ 0 = array-ptr-index p False 0
by (*auto simp add*: *array-ptr-index-def*)

lemma *ptr-coerce-index-array-ptr-index-1-conv*:
ptr-coerce p +$_p$ 1 = array-ptr-index p False 1
by (*auto simp add*: *array-ptr-index-def*)

lemma *ptr-coerce-index-array-ptr-index-sint-conv*:
0 ≤s i ⟹ ptr-coerce p +$_p$ sint i = array-ptr-index p False (nat (sint i))
by (*auto simp add*: *array-ptr-index-def word-sle-def*)

lemma *heap-access-Array-element''*:
fixes *p* :: (*'a*::*mem-type*[*'b*::*finite*]) *ptr*

assumes *less*: $n < CARD('b)$
shows
$index\ (h\text{-}val\ hp\ p)\ n$
$= h\text{-}val\ hp\ (array\text{-}ptr\text{-}index\ p\ False\ n)$
using *heap-access-Array-element' less*
by *auto*

lemma *index-fupdate-split*: $i < CARD('n) \Longrightarrow j < CARD('n) \Longrightarrow$
$index\ (fupdate\ i\ f\ (x::'a['n::finite]))\ j = (if\ i \neq j\ then\ (x.[j])\ else\ f\ (index\ x\ i))$
by (*cases i=j*) (*auto simp add*: *arr-fupdate-same arr-fupdate-other*)

lemma *root-disjoint-field-lvalue-disjoint1*:
fixes $p::'a::mem\text{-}type\ ptr$
assumes *field-lookup*: $field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a))\ path\ 0 = Some\ (t,\ n)$
assumes *match*: $export\text{-}uinfo\ t = typ\text{-}uinfo\text{-}t\ TYPE('f::c\text{-}type)$
assumes $ptr\text{-}span\ p \cap A = \{\}$
shows $A \cap \{\&(p\rightarrow path)..+size\text{-}of\ TYPE('f)\} = \{\}$
using *assms*
by (*simp add*: *disjoint-field-lvalue-propagation-right export-size-of inter-commute*)

lemma *root-disjoint-field-lvalue-disjoint2*:
fixes $p::'a::mem\text{-}type\ ptr$
assumes *field-lookup*: $field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a))\ path\ 0 = Some\ (t,\ n)$
assumes *match*: $export\text{-}uinfo\ t = typ\text{-}uinfo\text{-}t\ TYPE('f::c\text{-}type)$
assumes $ptr\text{-}span\ p \cap A = \{\}$
shows $\{\&(p\rightarrow path)..+size\text{-}of\ TYPE('f)\} \cap A = \{\}$
using *assms*
by (*simp add*: *disjoint-field-lvalue-propagation-right export-size-of inter-commute*)

context *heap-state-global*
begin

lemma *global-update-frame-commute*:
shows $glob\text{-}upd\ g\ (frame\ A\ t_0\ s) =$
$frame\ A\ t_0\ (glob\text{-}upd\ g\ s)$
proof −
from *glob-htd-commute glob-hmem-commute*
show *?thesis*
apply (*clarsimp simp add*: *frame-def*)
by (*smt* (*verit*) *heap.upd-cong typing.get-upd typing.upd-cong typing.upd-get*)
qed

lemma *L2-modify-no-heap-update-rel-stack*[*synthesize-rule refines-in-out*]:
assumes $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
assumes $v\ s = v'\ t$
shows *refines*
$(L2\text{-}modify\ (\lambda s.\ glob\text{-}upd\ (\lambda\text{-}.\ v\ s)\ s))$
$(L2\text{-}modify\ (\lambda s.\ glob\text{-}upd\ (\lambda\text{-}.\ v'\ s)\ s))$

s t
(*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
proof −
from *glob-hmem-commute* **have** *no-heap1*: $\bigwedge g\ s.$ *hmem* (*glob-upd g s*) = *hmem s*
by (*metis heap.get-upd heap.upd-get*)

from *glob-htd-commute* **have** *no-htd1*: $\bigwedge g\ s.$ *htd* (*glob-upd g s*) = *htd s*
by (*metis typing.get-upd typing.upd-get*)

from *assms no-heap1 no-htd1 global-update-frame-commute*
show *?thesis*
by (*auto simp add*: *refines-def-old rel-stack-def rel-alloc-def L2-defs*)
qed

lemma *rel-alloc-glabal-independent-eq* [*rel-alloc-independent-globals*]:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
shows *glob t* = *glob s*
by (*metis assms get-upd global-update-frame-commute rel-alloc-fold-frame upd-get*)

end

lemma *L2-seq-guard-cong-stateless*:
$(\bigwedge s.\ P\ s = P'\ s) \Longrightarrow (\bigwedge s.\ P'\ s \Longrightarrow X = X') \Longrightarrow$
L2-seq (*L2-guard P*) (λ-. *X*) = *L2-seq* (*L2-guard P'*) (λ-. *X'*)
using *L2-seq-guard-cong''*
by *metis*

context *globals-stack-heap-state*
begin
lemma *globals-subset-trans*: *NO-MATCH* $\mathcal{G}$ *X* $\Longrightarrow$ *NO-MATCH* $\mathcal{G}$ *Y* $\Longrightarrow X \subseteq \mathcal{G} \Longrightarrow \mathcal{G} \subseteq Y \Longrightarrow X \subseteq Y$
by *blast*

lemma *globals-disjoint-subset-left*: *NO-MATCH* $\mathcal{G}$ *X* $\Longrightarrow X \subseteq \mathcal{G} \Longrightarrow \mathcal{G} \cap A = \{\} \Longrightarrow X \cap A = \{\}$
by *blast*

lemma *globals-disjoint-subset-right*: *NO-MATCH* $\mathcal{G}$ *X* $\Longrightarrow X \subseteq \mathcal{G} \Longrightarrow A \cap \mathcal{G} = \{\} \Longrightarrow A \cap X = \{\}$
by *blast*

end

lemma *Union-Diff-right-conv'*: $X \cup Y = X \cup (Y - X)$
by *blast*

lemma *Union-Diff-right-conv*: $(Y - X) \equiv Z \Longrightarrow X \cup Y \equiv X \cup Z$
apply (*subst Union-Diff-right-conv'*)

apply *simp*
done

lemma *Union-assoc*: $X \cup Y \cup Z = X \cup (Y \cup Z)$
by (*simp add*: *sup-assoc*)

simproc-setup *Union-Diff-conv* (‹*ptr-span x* ∪ *Y*›) = ‹
let
val cterm-eq = *is-equal o Thm.fast-term-ord*
fun dest-union-right ct = *ct* |> *Match-Cterm.switch* [
@{*cterm-match* ‹*?X* ∪ *?Y*›} #> (*fn* {*X*, *Y*, ...} => *X*::*dest-union-right Y*),
fn - => [*ct*]]
in
fn - => *fn ctxt* => *fn ct* =>
let
val {*X*, *Y*, ...} = @{*cterm-match ?X* ∪ *?Y*} *ct*
val ys = *dest-union-right Y*
val - = *if member cterm-eq ys X then* () *else raise Match*
val lhs = ***infer-instantiate*** ‹*X* = *X and Y* = *Y in cterm Y* − *X*› *ctxt*
val eq = *lhs* |> *Simplifier.asm-full-rewrite* (*ctxt addsimps* @{*thms Un-Diff Diff-triv Union-assoc*})
val rhs = *Thm.rhs-of eq*
in
if cterm-eq (*lhs*, *rhs*) *then*
NONE
else
let val eq′ = @{*thm Union-Diff-right-conv*} *OF* [*eq*]
val - = *Utils.verbose-msg 1* (∗ *FIXME* ∗) *ctxt* (*fn* - => *Union-Diff-conv*: ^ *Thm.string-of-thm ctxt eq′*)
in SOME eq′ end
end
handle Match => *NONE*
end›
declare [[*simproc del*: *Union-Diff-conv*]]

lemma
ptr-span q ∩ *ptr-span p* = {} ⟹ *ptr-span x* ∩ *ptr-span p* = {} ⟹
(*ptr-span p* ∪ (*ptr-span q* ∪ (*ptr-span x* ∪ *ptr-span p*))) = *ptr-span p* ∪ (*ptr-span q* ∪ *ptr-span x*)
supply [[*simproc add*: *Union-Diff-conv*]]
apply *simp*
done

lemma
ptr-span q ∩ *ptr-span p* = {} ⟹ *ptr-span x* ∩ *ptr-span p* = {} ⟹
((*ptr-span p* ∪ *ptr-span q*) ∪ (*ptr-span x* ∪ *ptr-span p*)) = *ptr-span p* ∪ (*ptr-span q* ∪ *ptr-span x*)
supply [[*simproc add*: *Union-Diff-conv*]]
apply (*simp add*: *Union-assoc*)

done

lemma
assumes *disj*: *ptr-span q* ∩ *ptr-span p* = {}
ptr-span p ∩ *ptr-span q* = {}
ptr-span x ∩ *ptr-span p* = {}
ptr-span x ∩ *ptr-span q* = {}
shows ((*ptr-span p* ∪ *ptr-span q*) ∪ ((*ptr-span p* ∪ *ptr-span q*) ∪ *ptr-span x* ∪ *ptr-span p*)) = *ptr-span p* ∪ (*ptr-span q* ∪ *ptr-span x*)
supply [[*simproc add*: *Union-Diff-conv*]]
apply (*simp add*: *Union-assoc disj*)
done

lemma *subset-union-left*: $X \subseteq L \Longrightarrow X \subseteq L \cup R$
by *blast*

lemma *subset-union-right*: $X \subseteq R \Longrightarrow X \subseteq L \cup R$
by *blast*

lemma *disjoint-globals-stack*: $G \cap S = \{\} \Longrightarrow x \subseteq S \Longrightarrow G \cap x = \{\}$
by *blast*
end

Chapter 21

HL phase: Heap Lifting / Split Heap

theory *Split-Heap*
  imports
    TypHeapSimple
begin

ML-file *ac-names.ML*

definition *array-fields* :: *nat* ⇒ *qualified-field-name list* **where**
  array-fields n = *map* (λ*n*. [*replicate n CHR ''1''*]) [*0*..<*n*]

lemma *Nil-nmem-array-fields*[*simp*]: [] ∉ *set* (*array-fields n*)
  by (*auto simp add*: *array-fields-def*)

lemma *distinct-prop-disj-fn-array-fields*[*simp*]: *distinct-prop disj-fn* (*array-fields n*)
  by (*simp add*: *array-fields-def distinct-prop-iff-nth*)

lemma *field-lookup-field-ti'*:
  field-lookup (*typ-info-t TYPE*('*a* :: *c-type*)) *f 0* = *Some* (*a*, *b*) ⟹ *field-ti TYPE*('*a*) *f* = *Some a*
  unfolding *field-ti-def* **by** *simp*

lemma *field-lookup-append-add*:
  wf-desc t ⟹
    field-lookup t (*f* @ *g*) *n* =
      Option.bind (*field-lookup t f n*) (λ(*t'*, *m*).
        Option.bind (*field-lookup t' g 0*) (λ(*t''*, *m'*). *Some* (*t''*, *m* + *m'*)))
  by (*subst field-lookup-append-eq*)
    (*auto simp*: *field-lookup-offset intro*: *field-lookup-offset-None*[*THEN iffD2*]
        split: *bind-split*)

lemma *nth-array-fields*: *i* < *n* ⟹ *array-fields n* ! *i* = [*replicate i CHR ''1''*]
  by (*simp add*: *array-fields-def*)

lemma *array-fields-Suc*: *array-fields* (*Suc n*) = *array-fields n* @ [[*replicate n CHR ''1''*]]
by (*simp add*: *array-fields-def*)

lemma *find-append*: *find P* (*xs* @ *ys*) = (*case find P xs of None* ⇒ *find P ys* | *Some x* => *Some x*)
by (*induct xs*) *auto*

lemma *length-array-fields*: *length* (*array-fields n*) = *n*
by (*simp add*: *array-fields-def*)

lemma *set-array-fields*: *set* (*array-fields n*) = (⋃ *i*<*n*. {[*replicate i CHR ''1''*]})
by (*induct n*) (*auto simp add*: *array-fields-def*)
lemma *find-array-fields-Some*:
find P (*array-fields n*) = *Some y* ⟷
(∃ *i* < *n*. *y* = [*replicate i CHR ''1''*] ∧ *P y* ∧ (∀ *j*<*i*. ¬ *P* ([*replicate j CHR ''1''*])))
proof (*induct n arbitrary*: *y*)
case *0*
then show *?case*
by (*simp add*: *array-fields-def*)
next
case (*Suc n*)
show *?case*
proof (*cases find P* (*array-fields n*))
case *None*
then show *?thesis*
apply (*simp add*: *array-fields-Suc find-append*)
using *less-Suc-eq*
by (*auto dest*!: *findNoneD simp add*: *set-array-fields*)
next
case (*Some a*)
then show *?thesis*
apply (*simp add*: *array-fields-Suc find-append*)
apply (*simp add*: *Suc.hyps*)
by (*metis less-Suc-eq linorder-neqE-nat*)
qed
qed

lemma *Bex-intvl-conv*: (∃ *x* ∈ {*0*..<*n*::*nat*}. *P x*) ⟷ (∃ *i*. *i* < *n* ∧ *P i*)
by *auto*

lemma *Bex-union-conv*: (∃ *x* ∈ *A* ∪ *B*. *P x*) ⟷ ((∃ *x* ∈ *A*. *P x*) ∨ (∃ *x* ∈ *B*. *P x*))
by *auto*

lemma *ex-intvl-conj-distribR*: ((∃ *x*∈{*0*..<*n*}. *P x*) ∧ *Q*) ⟷ (∃ *x*∈{*0*..<*n*}. *P x* ∧ *Q*)
by *blast*

lemma *ex-less-conj-distribR*: $((\exists\, i{<}n{::}nat.\ P\ i) \wedge Q) \longleftrightarrow (\exists\, i{<}n.\ P\ i \wedge Q)$
by *blast*

lemma *map-filter-map-Some-conv*:
assumes *all-Some*: $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = Some\ (g\ x)$
shows *List.map-filter f xs* = *map g xs*
using *all-Some*
apply (*induct xs*)
apply (*auto simp add*: *List.map-filter-def*)
done

lemma *map-filter-map-compose*:
List.map-filter f (*map g xs*) = *List.map-filter* (*f o g*) *xs*
by (*induct xs*) (*auto simp add*: *List.map-filter-def*)

lemma *map-filter-fun-eq-conv*:
assumes *all-same*: $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x$
shows *List.map-filter f xs* = *List.map-filter g xs*
using *all-same*
apply (*induct xs*)
apply (*auto simp add*: *List.map-filter-def*)
done

lemma *map-filter-empty*[*simp*]: *List.map-filter Map.empty xs* = []
by (*simp add*: *List.map-filter-def*)

lemma *map-filter-Some*[*simp*]: *List.map-filter* (λ*x*. *Some* (*f x*)) *xs* = *map f xs*
by (*simp add*: *List.map-filter-def*)

lemma *list-all-field-lookup*[*simp*]:
CARD('*n*) = *m* ⟹
list-all
(λ*f*. ∃ *a b*. *field-lookup* (*typ-uinfo-t TYPE*('*a*::*c-type*['*n*::*finite*])) *f 0* = *Some* (*a*, *b*))
(*map* (λ*n*. [*replicate n CHR* ''*1*'']) [*0*..<*m*])
using *field-lookup-array*[**where** '*b*='*n* **and** '*a*='*a*]
apply (*simp add*: *list-all-iff atLeast0LessThan typ-uinfo-t-def flip*: *All-less-Ball*)
using *field-lookup-export-uinfo-Some* **by** *blast*

lemma *ptr-span-with-stack-byte-type-subset-field*[*simp*]:
∀ *a*∈*ptr-span* (*p*::'*a*::*mem-type ptr*). *root-ptr-valid h* (*PTR*(*stack-byte*) *a*) ⟹
∃ *u*. *field-ti TYPE*('*a*::*mem-type*) *f* = *Some u* ∧ *size-td u* = *sz* ⟹
(∀ *a*∈{&(*p*→*f*)..+*sz*}. *root-ptr-valid h* (*PTR*(*stack-byte*) *a*)) ⟷ *True*
by (*auto simp*: *dest*!: *mem-type-class.field-tag-sub field-tiD*)

lemma (**in** *pointer-lense*) *pointer-lense-field-lvalue*:
assumes *f*: *field-ti TYPE*('*c*::*mem-type*) *f* = *Some u*
and *u*: *size-td u* = *size-of TYPE*('*a*)

shows *pointer-lense* (λh (p::$'c$ *ptr*). r h (*PTR*($'a$) &(p→f))) (λp. w (*PTR*($'a$) &(p→f)))
apply *unfold-locales*
apply (*simp-all add*: *read-write-same write-same*)
apply (*subst write-other-commute*)
subgoal for p q
using *field-tag-sub*[*OF field-tiD*, *OF f*, *of p*]
using *field-tag-sub*[*OF field-tiD*, *OF f*, *of q*]
apply (*simp add*: *u*)
apply (*rule disjnt-subset1*, *erule disjnt-subset2*)
by *auto*
apply *simp*
done

lemma *exists-nat-numeral*: $\exists x$::*nat*. $x <$ *numeral k*
apply (*rule exI*[**where** *x=0*])
apply (*rule Num.rel-simps*)
done

lemma *fun-upd-eq-cases*: $f(p{:=}x) = g \longleftrightarrow (g\ p = x \wedge (\forall q.\ p{\neq}q \longrightarrow f\ q = g\ q))$
by (*auto simp add*: *fun-upd-def*)

lemma *fun-upd-apply-eq-cases*: $(f(p{:=}x))\ q = g\ q \longleftrightarrow ((p = q \longrightarrow g\ q = x) \wedge (p \neq q \longrightarrow f\ q = g\ q))$
by (*auto simp add*: *fun-upd-def*)

lemma *comp-fun-upd-same-fuse*:
$(\lambda f.\ f(p := x))\ o\ (\lambda f.\ f(p := y)) = (\lambda f.\ f(p{:=}x))$
by *auto*

lemma *comp-fun-upd-other-commute*:
$p \neq q \Longrightarrow (\lambda f.\ f(q := y))\ o\ (\lambda f.\ f(p := x)) = (\lambda f.\ f(p := x))\ o\ (\lambda f.\ f(q := y))$
by (*auto simp add*: *comp-def fun-upd-def fun-eq-iff*)

lemma *map-td-compose*:
fixes *t*::($'a$, $'b$) *typ-desc*
and *st*::($'a$, $'b$) *typ-struct*
and *ts*::($'a$, $'b$) *typ-tuple list*
and *x*::($'a$, $'b$) *typ-tuple*
shows
map-td f1 g1 (*map-td f2 g2 t*) = *map-td* (λn *algn*. *f1 n algn o f2 n algn*) (*g1 o g2*) *t* **and**
map-td-struct f1 g1 (*map-td-struct f2 g2 st*) = *map-td-struct* (λn *algn*. *f1 n algn o f2 n algn*) (*g1 o g2*) *st* **and**
map-td-list f1 g1 (*map-td-list f2 g2 ts*) = *map-td-list* (λn *algn*. *f1 n algn o f2 n algn*) (*g1 o g2*) *ts* **and**
map-td-tuple f1 g1 (*map-td-tuple f2 g2 x*) = *map-td-tuple* (λn *algn*. *f1 n algn o f2 n algn*) (*g1 o g2*) *x*
proof (*induct t* **and** *st* **and** *ts* **and** *x*)

case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** *auto*
qed

lemma (**in** *mem-type*) *distinct-all-field-names*:
distinct (*all-field-names* (*typ-info-t TYPE*('*a*)))
using *distinct-all-field-names wf-desc* **by** *blast*

lemma *field-lookup-same-type-disjoint*:
⟦*field-lookup t f m* = *Some* (*d*,*n*);
field-lookup t f' *m* = *Some* (*d'*,*n'*); *f* ≠ *f'*; *export-uinfo d* = *export-uinfo d'*;
wf-desc t; *size-td t* < *addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} = {} **and**

⟦*field-lookup-struct st f m* = *Some* (*d*,*n*);
field-lookup-struct st f' *m* = *Some* (*d'*,*n'*); *f* ≠ *f'*; *export-uinfo d* = *export-uinfo d'*;
wf-desc-struct st; *size-td-struct st* < *addr-card* ⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} = {} **and**

⟦*field-lookup-list ts f m* = *Some* (*d*,*n*);
field-lookup-list ts f' *m* = *Some* (*d'*,*n'*); *f* ≠ *f'*; *export-uinfo d* = *export-uinfo d'*;
wf-desc-list ts; *size-td-list ts* < *addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} = {} **and**
⟦*field-lookup-tuple x f m* = *Some* (*d*,*n*) ;
field-lookup-tuple x f' *m* = *Some* (*d'*,*n'*); *f* ≠ *f'*; *export-uinfo d* = *export-uinfo d'*;
wf-desc-tuple x; *size-td-tuple x* < *addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} =

{}
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m d n f' m d' n'* **and** *f m d n f' m d' n'* **and**
f m d n f' m d' n' **and** *f m d n f' m d' n'*)
case (*TypDesc algn st nm*)
then show *?case*
by (*metis field-lookup.simps field-lookup-export-uinfo-Some field-lookup-same-type-empty*(*1*) *size-td.simps wf-desc.simps*)
next
case (*TypScalar n algn x*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *simp*
next
case *Nil-typ-desc*
then show *?case* **by** *simp*
next
case (*Cons-typ-desc x xs*)
from *Cons-typ-desc.prems* **obtain**
f: *field-lookup-list* (*x* # *xs*) *f m* = *Some* (*d*, *n*) **and**
f': *field-lookup-list* (*x* # *xs*) *f' m* = *Some* (*d'*, *n'*) **and**
neq-f-f': *f* ≠ *f'* **and**
eq-d-d': *export-uinfo d* = *export-uinfo d'* **and**
sz: *size-td-tuple x* + *size-td-list xs* < *addr-card* **and**
wf-x: *wf-desc-tuple x* **and**
distinct: *dt-snd x* ∉ *dt-snd* ' *set xs* **and**
wf-xs: *wf-desc-list xs* **by** *clarsimp*
from *sz* **have** *sz-xs*: *size-td-list xs* < *addr-card* **by** *auto*
from *sz* **have** *sz-x*: *size-td-tuple x* < *addr-card* **by** *auto*
show *?case*
proof (*cases field-lookup-tuple x f m*)
case *None*
note *f-None* = *this*
from *f None* **have** *f-xs*: *field-lookup-list xs f* (*m* + *size-td* (*dt-fst x*)) = *Some* (*d*, *n*) **by** *simp*
from *td-set-list-field-lookup-listD* [*OF f-xs*] **have** (*d*, *n*) ∈ *td-set-list xs* (*m* + *size-td* (*dt-fst x*)) **.**
from *td-set-list-intvl-sub* [*OF this*]
have *contained-f*: {*word-of-nat n..+size-td d*} ⊆ {*word-of-nat* (*m* + *size-td* (*dt-fst x*))*..+size-td-list xs*} **.**
show *?thesis*
proof (*cases field-lookup-tuple x f' m*)
case *None*
from *f' None* **have** *f'-xs*: *field-lookup-list xs f'* (*m* + *size-td* (*dt-fst x*)) = *Some* (*d'*, *n'*) **by** *simp*
from *Cons-typ-desc.hyps*(*2*) [*OF f-xs f'-xs neq-f-f' eq-d-d' wf-xs sz-xs*]
show *?thesis* **.**
next

```
      case (Some -)
      with f' have f'-x: field-lookup-tuple x f' m = Some (d', n') by simp
      from td-set-tuple-field-lookup-tupleD [OF f'-x] have (d', n') ∈ td-set-tuple x
m .
      from td-set-tuple-intvl-sub [OF this]
      have {word-of-nat n'..+size-td d'} ⊆ {word-of-nat m..+size-td-tuple x} .
      with contained-f sz show ?thesis
     by (smt (verit, ccfv-SIG) disj-subset init-intvl-disj of-nat-add size-td-tuple-dt-fst)
    qed
  next
    case (Some -)
    from f Some have f-x: field-lookup-tuple x f m = Some (d, n) by simp
    from td-set-tuple-field-lookup-tupleD [OF f-x] have (d, n) ∈ td-set-tuple x m .
    from td-set-tuple-intvl-sub [OF this]
   have contained-f: {word-of-nat n..+size-td d} ⊆ {word-of-nat m..+size-td-tuple
x} .
    show ?thesis
    proof (cases field-lookup-tuple x f' m)
      case None
       from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d', n') by simp
      from td-set-list-field-lookup-listD [OF f'-xs]
      have (d', n') ∈ td-set-list xs (m + size-td (dt-fst x)) .
      from td-set-list-intvl-sub [OF this]
        have {word-of-nat n'..+size-td d'} ⊆ {word-of-nat (m + size-td (dt-fst
x))..+size-td-list xs} .
      with contained-f sz show ?thesis
        by (smt (verit, ccfv-threshold) Int-commute disjoint-subset init-intvl-disj
            of-nat-add size-td-tuple-dt-fst)
    next
      case (Some -)
      with f' have f'-x: field-lookup-tuple x f' m = Some (d', n') by simp
      from Cons-typ-desc.hyps(1) [OF f-x f'-x neq-f-f' eq-d-d' wf-x sz-x]
      show ?thesis .
    qed
  qed
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
   by (metis fl5 list.expand option.simps(3) size-td-tuple.simps wf-desc-tuple.simps)
qed


lemma field-lookup-same-type-u-disjoint:
  ⟦field-lookup t f m = Some (d,n);
      field-lookup t f' m = Some (d,n'); f ≠ f';
      wf-desc t; size-td t < addr-card⟧ ⟹
      {(of-nat n)::addr..+size-td d} ∩ {of-nat n'..+size-td d} = {} and
```

⟦*field-lookup-struct st f m* = *Some* (*d*,*n*);
field-lookup-struct st f′ m = *Some* (*d*,*n′*); *f* ≠ *f′*;
wf-desc-struct st; *size-td-struct st* < *addr-card* ⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td d*} ∩ {*of-nat n′*..+*size-td d*} = {} **and**

⟦*field-lookup-list ts f m* = *Some* (*d*,*n*);
field-lookup-list ts f′ m = *Some* (*d*,*n′*); *f* ≠ *f′*;
wf-desc-list ts; *size-td-list ts* < *addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td d*} ∩ {*of-nat n′*..+*size-td d*} = {} **and**
⟦*field-lookup-tuple x f m* = *Some* (*d*,*n*) ;
field-lookup-tuple x f′ m = *Some* (*d*,*n′*); *f* ≠ *f′*;
wf-desc-tuple x; *size-td-tuple x* < *addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td d*} ∩ {*of-nat n′*..+*size-td d*} = {}
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m d n f′ m n′* **and** *f m d n f′ m n′* **and**
f m d n f′ m n′ **and** *f m d n f′ m n′*)
case (*TypDesc algn st nm*)
then show *?case*
by (*metis field-lookup.simps field-lookup-same-type-empty*(*1*)
size-td.simps wf-desc.simps)
next
case (*TypScalar n algn x*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)
then show *?case* **by** *simp*
next
case *Nil-typ-desc*
then show *?case* **by** *simp*
next
case (*Cons-typ-desc x xs*)
from *Cons-typ-desc.prems* **obtain**
f: *field-lookup-list* (*x* # *xs*) *f m* = *Some* (*d*, *n*) **and**
f′: *field-lookup-list* (*x* # *xs*) *f′ m* = *Some* (*d*, *n′*) **and**
neq-f-f′: *f* ≠ *f′* **and**
sz: *size-td-tuple x* + *size-td-list xs* < *addr-card* **and**
wf-x: *wf-desc-tuple x* **and**
distinct: *dt-snd x* ∉ *dt-snd* ‘ *set xs* **and**
wf-xs: *wf-desc-list xs* **by** *clarsimp*
from *sz* **have** *sz-xs*: *size-td-list xs* < *addr-card* **by** *auto*
from *sz* **have** *sz-x*: *size-td-tuple x* < *addr-card* **by** *auto*
show *?case*
proof (*cases field-lookup-tuple x f m*)
case *None*
note *f-None* = *this*
from *f None* **have** *f-xs*: *field-lookup-list xs f* (*m* + *size-td* (*dt-fst x*)) = *Some* (*d*, *n*) **by** *simp*
from *td-set-list-field-lookup-listD* [*OF f-xs*] **have** (*d*, *n*) ∈ *td-set-list xs* (*m* + *size-td* (*dt-fst x*)) **.**

```
    from td-set-list-intvl-sub [OF this]
     have contained-f: {word-of-nat n..+size-td d} ⊆ {word-of-nat (m + size-td
(dt-fst x))..+size-td-list xs} .
    show ?thesis
    proof (cases field-lookup-tuple x f' m)
      case None
       from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d, n') by simp
      from Cons-typ-desc.hyps(2) [OF f-xs f'-xs neq-f-f' wf-xs sz-xs]
      show ?thesis .
    next
      case (Some -)
      with f' have f'-x: field-lookup-tuple x f' m = Some (d, n') by simp
       from td-set-tuple-field-lookup-tupleD [OF f'-x] have (d, n') ∈ td-set-tuple x
m .
      from td-set-tuple-intvl-sub [OF this]
      have {word-of-nat n'..+size-td d} ⊆ {word-of-nat m..+size-td-tuple x} .
      with contained-f sz show ?thesis
      by (smt (verit, ccfv-SIG) disj-subset init-intvl-disj of-nat-add size-td-tuple-dt-fst)
    qed
  next
    case (Some -)
    from f Some have f-x: field-lookup-tuple x f m = Some (d, n) by simp
    from td-set-tuple-field-lookup-tupleD [OF f-x] have (d, n) ∈ td-set-tuple x m .
    from td-set-tuple-intvl-sub [OF this]
   have contained-f: {word-of-nat n..+size-td d} ⊆ {word-of-nat m..+size-td-tuple
x} .
    show ?thesis
    proof (cases field-lookup-tuple x f' m)
      case None
       from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d, n') by simp
      from td-set-list-field-lookup-listD [OF f'-xs]
      have (d, n') ∈ td-set-list xs (m + size-td (dt-fst x)) .
      from td-set-list-intvl-sub [OF this]
        have {word-of-nat n'..+size-td d} ⊆ {word-of-nat (m + size-td (dt-fst
x))..+size-td-list xs} .
      with contained-f sz show ?thesis
       by (smt (verit, ccfv-threshold) Int-commute disjoint-subset init-intvl-disj
          of-nat-add size-td-tuple-dt-fst)
    next
      case (Some -)
      with f' have f'-x: field-lookup-tuple x f' m = Some (d, n') by simp
      from Cons-typ-desc.hyps(1) [OF f-x f'-x neq-f-f' wf-x sz-x]
      show ?thesis .
    qed
  qed
next
  case (DTuple-typ-desc typ-desc list b)
```

then show *?case*
by (*metis fl5 list.expand option.simps*(*3*) *size-td-tuple.simps wf-desc-tuple.simps*)
qed

lemma *td-set-list-intvl-sub-nat*:
(*d*,*n*) ∈ *td-set-list t m* ⟹ {*n*..< *n* +*size-td d*} ⊆ {*m*..< *m* +*size-td-list t*}
apply(*frule td-set-list-offset-le*)
apply(*drule td-set-list-offset-size-m*)
apply *auto*
done

lemma *td-set-tuple-intvl-sub-nat*:
(*d*,*n*) ∈ *td-set-tuple t m* ⟹ {*n*..<*n*+*size-td d*} ⊆ {*m*..<*m*+*size-td-tuple t*}
apply(*frule td-set-tuple-offset-le*)
apply(*drule td-set-tuple-offset-size-m*)
apply *auto*
done

lemma *field-lookup-same-type-u-disjoint-nat*:
⟦*field-lookup t f m* = *Some* (*d*,*n*);
field-lookup t f′ m = *Some* (*d*,*n′*); *f* ≠ *f′*;
wf-desc t⟧ ⟹
{*n*..<*n* + *size-td d*} ∩ {*n′*..<*n′* + *size-td d*} = {} **and**

⟦*field-lookup-struct st f m* = *Some* (*d*,*n*);
field-lookup-struct st f′ m = *Some* (*d*,*n′*); *f* ≠ *f′*;
wf-desc-struct st ⟧ ⟹
{*n*..< *n* + *size-td d*} ∩ {*n′*..< *n′* + *size-td d*} = {} **and**

⟦*field-lookup-list ts f m* = *Some* (*d*,*n*);
field-lookup-list ts f′ m = *Some* (*d*,*n′*); *f* ≠ *f′*;
wf-desc-list ts⟧ ⟹
{*n*..<*n* + *size-td d*} ∩ {*n′*..<*n′* + *size-td d*} = {} **and**
⟦*field-lookup-tuple x f m* = *Some* (*d*,*n*) ;
field-lookup-tuple x f′ m = *Some* (*d*,*n′*); *f* ≠ *f′*;
wf-desc-tuple x⟧ ⟹
{*n*..<*n* + *size-td d*} ∩ {*n′*..<*n′* +*size-td d*} = {}
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f m d n f′ m n′* **and** *f m d n f′ m n′* **and**
f m d n f′ m n′ **and** *f m d n f′ m n′*)
case (*TypDesc align st nm*)
then show *?case*
by (*metis field-lookup.simps field-lookup-same-type-empty*(*1*)
wf-desc.simps)
next
case (*TypScalar n align x*)
then show *?case* **by** *auto*
next
case (*TypAggregate ts*)

```
  then show ?case by auto
next
  case Nil-typ-desc
  then show ?case by simp
next
  case (Cons-typ-desc x xs)

  from Cons-typ-desc.prems obtain
    f: field-lookup-list (x # xs) f m = Some (d, n) and
    f': field-lookup-list (x # xs) f' m = Some (d, n') and
    neq-f-f': f ≠ f' and
    wf-x: wf-desc-tuple x and
    distinct: dt-snd x ∉ dt-snd ‘ set xs and
    wf-xs: wf-desc-list xs by clarsimp

  show ?case
  proof (cases field-lookup-tuple x f m)
    case None
    note f-None = this
    from f None have f-xs: field-lookup-list xs f (m + size-td (dt-fst x)) = Some
(d, n) by simp
    from td-set-list-field-lookup-listD [OF f-xs] have (d, n) ∈ td-set-list xs (m +
size-td (dt-fst x)) .
    from td-set-list-intvl-sub-nat [OF this]
   have contained-f: {n..<n+size-td d} ⊆ {(m + size-td (dt-fst x))..<(m + size-td
(dt-fst x)) +size-td-list xs} .
    show ?thesis
    proof (cases field-lookup-tuple x f' m)
      case None
       from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d, n') by simp
      from Cons-typ-desc.hyps(2) [OF f-xs f'-xs neq-f-f' wf-xs ]
      show ?thesis .
    next
      case (Some -)
      with f' have f'-x: field-lookup-tuple x f' m = Some (d, n') by simp
       from td-set-tuple-field-lookup-tupleD [OF f'-x] have (d, n') ∈ td-set-tuple x
m .
      from td-set-tuple-intvl-sub-nat[OF this]
      have {n'..<n'+size-td d} ⊆ {m..<m+size-td-tuple x} .
      with contained-f show ?thesis
     by (smt (verit, best) disj-inter-swap disj-subset ivl-disj-int-two(3) size-td-tuple-dt-fst)
    qed

  next
    case (Some -)
    from f Some have f-x: field-lookup-tuple x f m = Some (d, n) by simp
    from td-set-tuple-field-lookup-tupleD [OF f-x] have (d, n) ∈ td-set-tuple x m .
    from td-set-tuple-intvl-sub-nat [OF this]
```

```
    have contained-f: {n..<n+size-td d} ⊆ {m..<m+size-td-tuple x} .
    show ?thesis
    proof (cases field-lookup-tuple x f' m)
      case None
       from f' None have f'-xs: field-lookup-list xs f' (m + size-td (dt-fst x)) =
Some (d, n') by simp
      from td-set-list-field-lookup-listD [OF f'-xs]
      have (d, n') ∈ td-set-list xs (m + size-td (dt-fst x)) .
      from td-set-list-intvl-sub-nat [OF this]
       have {n'..<n'+size-td d} ⊆ {(m + size-td (dt-fst x))..<(m + size-td (dt-fst
x))+size-td-list xs} .
      with contained-f show ?thesis
       by (metis Int-commute disjoint-subset ivl-disj-int-two(3) size-td-tuple-dt-fst)
    next
      case (Some -)
      with f' have f'-x: field-lookup-tuple x f' m = Some (d, n') by simp
      from Cons-typ-desc.hyps(1) [OF f-x f'-x neq-f-f'  wf-x ]
      show ?thesis .
    qed
  qed
next
  case (DTuple-typ-desc typ-desc list b)
  then show ?case
    by (metis fl5 list.expand option.simps(3) wf-desc-tuple.simps)
qed

lemma (in mem-type) field-lookup-same-type-disjoint:
  assumes f: field-lookup (typ-info-t TYPE('a)) f m = Some (t, n)
  assumes f': field-lookup (typ-info-t TYPE('a)) f' m = Some (t', n')
  assumes neq: f ≠ f'
  assumes same: export-uinfo t = export-uinfo t'
  shows {(of-nat n)::addr..+size-td t} ∩ {of-nat n'..+size-td t'} = {}
proof −
  have size-td (typ-info-t TYPE('a)) < addr-card
    by (simp add: size-of-fold)

  from field-lookup-same-type-disjoint(1) [OF f f' neq same wf-desc this]
  show ?thesis
    by simp
qed

lemma (in mem-type) field-lookup-same-type-ptr-span-disjoint:
  fixes p::'a ptr
  assumes f: field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)
  assumes f': field-lookup (typ-info-t TYPE('a)) f' 0 = Some (t', n')
  assumes neq: f ≠ f'
  assumes t: export-uinfo t = typ-uinfo-t (TYPE('b::c-type))
  assumes t': export-uinfo t' = typ-uinfo-t (TYPE('b::c-type))
  shows ptr-span (PTR('b) &(p→f)) ∩ ptr-span (PTR('b) &(p→f')) = {}
```

proof −
from *t t′* **have** *export-uinfo t* = *export-uinfo t′* **by** *simp*
from *field-lookup-same-type-disjoint* [*OF f f′ neq this*]
have {*word-of-nat n*::*addr*..+*size-td t*} ∩ {*word-of-nat n′*..+*size-td t′*} = {} **.**
moreover
from *t* **have** *size-td t* = *size-of TYPE*(′*b*)
by (*simp add*: *export-size-of*)
moreover
from *t′* **have** *size-td t′* = *size-of TYPE*(′*b*)
by (*simp add*: *export-size-of*)
ultimately
show *?thesis*
using *f f′*
by (*simp add*: *field-lvalue-def intvl-disj-offset*)
qed

lemma *sub-type-valid-field-lvalue-overlap*:
fixes *p*::′*a*::*mem-type ptr*
fixes *q*::′*b*::*mem-type ptr*
assumes *subtype*: *TYPE*(′*b*) ≤$_\tau$ *TYPE*(′*a*)
assumes *valid-p*: *d* ⊨$_t$ *p*
assumes *overlap*: *ptr-span p* ∩ *ptr-span q* ≠ {}
shows *d* ⊨$_t$ *q* ⟷
(∃*f t n*. *field-lookup* (*typ-info-t TYPE*(′*a*)) *f 0* = *Some* (*t*, *n*) ∧
export-uinfo t = *typ-uinfo-t TYPE*(′*b*) ∧
q = *PTR*(′*b*) (&(*p*→*f*)))
proof
assume *valid-q*: *d* ⊨$_t$ *q*
show ∃*f t n*. *field-lookup* (*typ-info-t TYPE*(′*a*)) *f 0* = *Some* (*t*, *n*) ∧
export-uinfo t = *typ-uinfo-t TYPE*(′*b*) ∧ *q* = *PTR*(′*b*) (&(*p*→*f*))
proof −

from *subtype* **have** *typ-uinfo-t TYPE*(′*b*) ≤ *typ-uinfo-t TYPE*(′*a*) **by** (*simp add*: *sub-typ-def*)

from *this overlap valid-footprint-sub-cases* [*OF h-t-valid-valid-footprint* [*OF valid-q*] *h-t-valid-valid-footprint* [*OF valid-p*]]
have *field-of*: *field-of* (*ptr-val q* − *ptr-val p*) (*typ-uinfo-t TYPE*(′*b*)) (*typ-uinfo-t TYPE*(′*a*))
apply (*simp add*: *size-of-def*)
by (*metis Int-commute le-less-trans less-irrefl*)
then obtain *f* **where**
fl: *field-lookup* (*typ-uinfo-t* (*TYPE*(′*a*))) *f 0* = *Some* (*typ-uinfo-t TYPE*(′*b*), *unat* (*ptr-val q* − *ptr-val p*))
using *field-of-lookup-info* **by** *blast*

then obtain *t* **where**

fl': *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*t*, *unat* (*ptr-val q* − *ptr-val p*)) **and**
t: *export-uinfo t* = *typ-uinfo-t TYPE*('*b*)
using *field-lookup-uinfo-Some-rev* **by** *blast*

from *fl'*
have *ptr-val-q*: *ptr-val q* = &(*p*→*f*)
by (*simp add*: *field-lvalue-def*)

with *fl' t* **show** *?thesis*
by (*metis Ptr-ptr-val*)
qed
next
assume ∃*f t n*.
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*) ∧ *export-uinfo t* = *typ-uinfo-t TYPE*('*b*) ∧
q = *PTR*('*b*) &(*p*→*f*)
then show *d* ⊨*t q*
using *valid-p*
using *c-guard-mono h-t-valid-mono* **by** *blast*
qed

lemma *field-lookup-intvl-contained-left*:
fixes *p*::'*a*::*mem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *k*)
assumes *n*: *n* = *size-td t*
assumes *m*: *m* = *size-of TYPE*('*a*)
shows {&(*p*→*f*)..+*n*} ∩ {*ptr-val p*..+*m*} ≠ {}
proof −
from *field-tag-sub* [*OF fl*] **have** {&(*p*→*f*)..+*size-td t*} ⊆ *ptr-span p* **.**
moreover from *fl* **have** *0* < *size-td t*
by (*simp add*: *field-lookup-wf-size-desc-gt*)
ultimately show *?thesis* **using** *n m*
by (*simp add*: *intvl-non-zero-non-empty le-iff-inf*)
qed

lemma *field-lookup-intvl-contained-right*:
fixes *p*::'*a*::*mem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *k*)
assumes *n*: *n* = *size-td t*
assumes *m*: *m* = *size-of TYPE*('*a*)
shows {*ptr-val p*..+*m*} ∩ {&(*p*→*f*)..+*n*} ≠ {}
using *field-lookup-intvl-contained-left* [*OF fl n m*] **by** *blast*

lemma *field-overlap-right*:
fixes *p*::'*a*::*mem-type ptr*
assumes *field-lookup*: *field-lookup* (*typ-info-t TYPE*('*a*)) *path 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*f*)
shows (*ptr-span p*) ∩ *ptr-span* (*PTR*('*f*::*c-type*) &(*p*→*path*)) ≠ {}

using *field-lookup match*
by (*simp add*: *export-size-of field-lookup-intvl-contained-right*)

locale *nested-field'* =
fixes *t* :: *'a::mem-type xtyp-info*
fixes *path* :: *string list*
fixes *sel* :: *'a::mem-type ⇒ 'f::mem-type*
fixes *upd* :: *'f ⇒ 'a ⇒ 'a*

assumes *field-ti*: *field-ti TYPE('a) path = Some t*
assumes *field-typ-match*: *export-uinfo t = typ-uinfo-t TYPE('f)*

assumes *sel-def*: *sel ≡ from-bytes o access-ti$_0$ t*
assumes *upd-def*: *upd ≡ update-ti t o to-bytes-p*
begin

lemma *sub-typ*: *TYPE('f) ≤$_\tau$ TYPE('a)*
using *field-ti field-typ-match*
using *field-ti-sub-typ sub-typ-def* **by** *blast*

lemma *h-val-field*:
shows *h-val h (PTR('f) &(p→path)) = sel (h-val h p)*
using *TypHeap.h-val-field-from-bytes' field-ti field-typ-match sel-def typ-uinfo-t-def*
by *fastforce*

lemma *clift-field-update*:
assumes *typed*: *hrs-htd h ⊨$_t$ p*
shows *clift (hrs-mem-update (heap-update (PTR('f) &(p→path)) x) h) =*
(clift h)(p↦ upd x (h-val (hrs-mem h) p))
proof −
from *h-t-valid-clift-Some-iff typed* **obtain** *v* **where** *clift*: *clift h p = Some v* **by** *blast*
from *h-val-clift' [OF this]*
have *v*: *v = h-val (hrs-mem h) p* **by** *simp*
from *clift-field-update [OF field-ti - clift [simplified v]*, **where** *val=x,OF field-typ-match [simplified typ-uinfo-t-def]]*
show *?thesis*
by (*simp add*: *upd-def export-size-of field-typ-match update-ti-t-def*)
qed

lemma *clift-field-update-padding*:
assumes *typed*: *hrs-htd h ⊨$_t$ p*
assumes *lbs*: *length bs = size-of TYPE('f)*
shows *clift (hrs-mem-update (heap-update-padding (PTR('f) &(p→path)) x bs) h) =*
(clift h)(p↦ upd x (h-val (hrs-mem h) p))
proof −
from *h-t-valid-clift-Some-iff typed* **obtain** *v* **where** *clift*: *clift h p = Some v* **by** *blast*

from *h-val-clift'* [*OF this*]
have *v*: *v* = *h-val* (*hrs-mem h*) *p* **by** *simp*
from *clift-field-update-padding* [*OF field-ti - clift* [*simplified v*] *lbs*, **where** *val=x,OF field-typ-match* [*simplified typ-uinfo-t-def*]]
show *?thesis*
by (*simp add*: *upd-def export-size-of field-typ-match update-ti-t-def*)
qed

lemma *h-val-field-lvalue-update*: *d* ⊨$_t$ *p* ⟹ *d* ⊨$_t$ *q* ⟹
h-val (*heap-update* (*PTR*('*f*) &(*p*→*path*)) *x h*) *q* = ((*h-val h*)(*p* := *upd x* (*h-val h p*))) *q*
by (*smt* (*verit*, *best*) *c-type-class.lift-def clift-Some-eq-valid h-val-clift' hrs-htd-def hrs-mem-def hrs-mem-update lift-heap-update lift-t-heap-update nested-field'.clift-field-update nested-field'-axioms prod.sel*(*1*) *prod.sel*(*2*))

lemma *h-val-field-lvalue-update-padding*: *d* ⊨$_t$ *p* ⟹ *d* ⊨$_t$ *q* ⟹ *length bs* = *size-of TYPE*('*f*) ⟹
h-val (*heap-update-padding* (*PTR*('*f*) &(*p*→*path*)) *x bs h*) *q* = ((*h-val h*)(*p* := *upd x* (*h-val h p*))) *q*
by (*smt* (*verit*, *ccfv-threshold*) *clift-Some-eq-valid fst-conv h-val-clift hrs-htd-def hrs-htd-mem-update hrs-mem-def hrs-mem-update clift-field-update-padding clift-field-update h-val-field-lvalue-update prod.collapse snd-conv*)

end

lemma *align-addr-card*:
assumes *wf-size-desc*: *wf-size-desc t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size-of*: *2* ^ *align-td t dvd size-td t*
shows *2* ^ *align-td t dvd addr-card*
proof −
from *wf-size-desc* **have** *sz-nzero*: *0* < *size-td t*
by (*simp add*: *wf-size-desc-gt*(*1*))
with *align-size-of max-size*
have *align-bound*: *align-td t* < *addr-bitsize*
by (*metis addr-card linorder-not-le nat-dvd-not-less power-le-dvd*)
then have *bound*: *2* ^ *align-td t* < *addr-card*
by (*metis addr-card one-less-numeral-iff power-strict-increasing semiring-norm*(*76*))
from *bound align-bound*
show *?thesis*
apply (*clarsimp simp*: *dvd-def*)
apply (*rule exI*[**where** *x=2*^(*len-of TYPE*(*addr-bitsize*) − *align-td t*)])
apply *clarsimp*
apply(*simp add*: *addr-card flip*: *power-add*)
done
qed

lemma *ptr-aligned-u-field-lookup*:
assumes *wf-size-desc*: *wf-size-desc t*
assumes *wf-align*: *wf-align t*
assumes *align-field*: *align-field t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size-of*: *2* ^ *align-td t dvd size-td t*
assumes *align-size-of-u*: *2* ^ *align-td u dvd size-td u* — does not follow from a well-formedness condition on t
assumes *fl*: *field-lookup t path 0* = *Some* (*u*, *n*)
assumes *ptr-aligned-u*: *ptr-aligned-u t a*
shows *ptr-aligned-u u* (*a* + *of-nat n*)
proof −

from *wf-size-desc fl* **have** *wf-size-desc-u*: *wf-size-desc u*
using *field-lookup-wf-size-desc-pres*(*1*) **by** *blast*

from *fl* **have** *u-n-bound*: *size-td u* + *n* ≤ *size-td t*
by (*simp add*: *field-lookup-offset-size'*)
with *max-size* **have** *n-bound*: *n* < *addr-card*
by *simp*
from *u-n-bound max-size* **have** *sz-u*: *size-td u* < *addr-card*
by *simp*

from *align-td-field-lookup*(*1*) [*OF wf-align fl*] **have** *align-u-t*: *align-td u* ≤ *align-td t* **.**
from *align-field fl*
have *2* ^ (*align-td u*) *dvd n* **by** (*simp add*: *align-field-def*)
moreover
from *ptr-aligned-u* **have** *2* ^ *align-td t dvd unat a* **by** (*simp add*: *ptr-aligned-u-def*)

ultimately have *2* ^ *align-td u dvd unat* (*a* + *word-of-nat n*)
apply −
apply(*subst unat-word-ariths*)
apply(*rule dvd-mod*)
apply(*rule dvd-add*)
subgoal
using *align-u-t*
using *power-le-dvd* **by** *blast*
subgoal
apply(*subst unat-of-nat*)
apply(*subst Euclidean-Rings.mod-less*)
apply(*subst len-of-addr-card*)
apply (*rule n-bound*)
apply *assumption*
done
subgoal

apply(*subst len-of-addr-card*)
apply (*rule align-addr-card*)
apply (*rule wf-size-desc-u*)
apply (*rule sz-u*)
apply (*rule align-size-of-u*)
done
done

then show *?thesis*
by (*simp add*: *ptr-aligned-u-def*)
qed

lemma *c-guard-u-field-looup*:
assumes *wf-size-desc*: *wf-size-desc t*
assumes *wf-align*: *wf-align t*
assumes *align-field*: *align-field t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size-of*: *2* ^ *align-td t dvd size-td t*
assumes *align-size-of-u*: *2* ^ *align-td u dvd size-td u* — does not follow from a well-formedness condition on t
assumes *fl*: *field-lookup t path 0* = *Some* (*u*, *n*)
assumes *c-guard-u*: *c-guard-u t a*
shows *c-guard-u u* (*a* + *of-nat n*)
proof −
from *wf-size-desc* **have** *sz-t*: *0* < *size-td t*
using *wf-size-desc-gt*(*1*) **by** *blast*
from *wf-size-desc fl* **have** *sz-u*: *0* < *size-td u*
by (*simp add*: *field-lookup-wf-size-desc-gt*)
show *?thesis*
using *fl c-guard-u*
apply (*clarsimp simp add*: *c-guard-u-def*)
apply *safe*
subgoal by (*rule ptr-aligned-u-field-lookup* [*OF wf-size-desc wf-align align-field max-size align-size-of align-size-of-u fl*])
subgoal by (*meson c-null-guard-u-def field-lookup-offset-size′ intvl-le subsetD*)
done
qed

lemma *cvalid-u-field-lookup*:
assumes *wf-size-desc*: *wf-size-desc t*
assumes *wf-align*: *wf-align t*
assumes *align-field*: *align-field t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size-of*: *2* ^ *align-td t dvd size-td t*
assumes *align-size-of-u*: *2* ^ *align-td u dvd size-td u* — does not follow from a well-formedness condition on t
assumes *fl*: *field-lookup t path 0* = *Some* (*u*, *n*)
assumes *cvalid-u*: *cvalid-u t d a*

```
  shows cvalid-u u d (a + of-nat n)
proof −
  from wf-size-desc have sz-t: 0 < size-td t
    using wf-size-desc-gt(1) by blast
  from wf-size-desc fl have sz-u: 0 < size-td u
    by (simp add: field-lookup-wf-size-desc-gt)

  from fl have u-n-bound: size-td u + n ≤ size-td t
    by (simp add: field-lookup-offset-size')
  with max-size have n-bound: n < addr-card
    by simp
  from u-n-bound max-size have sz-u-addr-card: size-td u < addr-card
    by simp

 note c-guard = c-guard-u-field-looup [OF wf-size-desc wf-align align-field max-size
align-size-of align-size-of-u fl]

  from cvalid-u
  show ?thesis
    apply(clarsimp simp: cvalid-u-def valid-footprint-def Let-def  sz-t sz-u c-guard)
    subgoal for y
      apply(drule-tac x=n+y in spec)
      apply (erule impE)
      subgoal using u-n-bound by simp
      subgoal
        by (metis (mono-tags, lifting) ab-semigroup-add-class.add-ac(1)
            fl map-list-map-trans of-nat-add td-set-field-lookupD typ-slice-td-set)
      done
    done
qed

locale mem-type-u =
  fixes t::typ-uinfo
  assumes wf-desc: wf-desc t
  assumes wf-size-desc: wf-size-desc t
  assumes wf-align: wf-align t
  assumes align-field: align-field t
  assumes max-size: size-td t < addr-card
  assumes align-size: 2 ^ align-td t dvd size-td t
begin
lemmas ptr-aligned-u-field-lookup = ptr-aligned-u-field-lookup [OF wf-size-desc wf-align
align-field max-size align-size]
lemmas c-guard-u-field-looup = c-guard-u-field-looup [OF wf-size-desc wf-align align-field
max-size align-size]
lemmas cvalid-u-field-lookup = cvalid-u-field-lookup [OF wf-size-desc wf-align align-field
max-size align-size]

end
```

lemma *wf-size-desc-export-uinfo*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
wf-size-desc t ⟹ *wf-size-desc* (*export-uinfo t*) **and**
wf-size-desc-struct st ⟹ *wf-size-desc-struct* (*map-td-struct field-norm* (λ-. ()) *st*) **and**
wf-size-desc-list ts ⟹ *wf-size-desc-list* (*map-td-list field-norm* (λ-. ()) *ts*) **and**
wf-size-desc-tuple x ⟹ *wf-size-desc-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*)
proof (*induct t* **and** *st* **and** *ts* **and** *x*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **by** *auto*
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** (*auto simp add*: *export-uinfo-def*)
qed

context *mem-type*
begin

lemma *typ-uinfo-t-mem-type*[*simp*]: *mem-type-u* (*typ-uinfo-t*(*TYPE*('*a*)))
apply (*simp only*: *typ-uinfo-t-def*)
apply (*unfold-locales*)
apply (*simp add*: *typ-uinfo-t-def wf-desc-export-uinfo-pres*(*1*))
apply (*simp add*:*wf-size-desc-export-uinfo*)
apply (*metis export-uinfo-typdesc-simp local.wf-align typ-desc.exhaust wf-align-map-td*(*1*))
using *local.align-field local.align-field-uinfo local.typ-uinfo-t-def* **apply** *fastforce*
apply (*simp add*: *size-of-fold*)
using *align-size-of*
apply (*simp add*: *size-of-def align-of-def*)
done

sublocale *u*: *mem-type-u typ-uinfo-t*(*TYPE*('*a*))

by (*rule typ-uinfo-t-mem-type*)

end

lemma *field-names-u-composeI*:
assumes *wf-t*: *wf-desc t*
assumes *path1*: *path1* ∈ *set* (*field-names-u t u*)
assumes *path2*: *path2* ∈ *set* (*field-names-u u v*)
shows *path1* @ *path2* ∈ *set* (*field-names-u t v*)
proof −
from *path1 wf-t* **obtain** *n* **where** *fl1*: *field-lookup t path1 0* = *Some* (*u*, *n*)
using *field-names-u-filter-all-field-names-conv*(*1*) **by** *fastforce*
with *wf-t* **have** *wf-u*: *wf-desc u*
using *field-lookup-wf-desc-pres*(*1*) **by** *blast*
from *path2 wf-u* **obtain** *m* **where** *field-lookup u path2 0* = *Some* (*v*, *m*)
using *field-names-u-filter-all-field-names-conv*(*1*) **by** *fastforce*
hence *fl2*: *field-lookup u path2 n* = *Some* (*v*, *n* + *m*)
by (*simp add*: *field-lookup-offset*)
from *field-lookup-prefix-Some''*(*1*)[*rule-format*, *OF fl1 wf-t*, *of path2*] *fl2*
have *field-lookup t* (*path1* @ *path2*) *0* = *Some* (*v*, *n* + *m*) **by** *simp*
then
show *?thesis*
by (*simp add*: *field-lookup-all-field-names*(*1*) *field-names-u-filter-all-field-names-conv*(*1*)
wf-t)
qed

lemma *field-names-u-composeD*:
assumes *wf-t*: *wf-desc t*
assumes *append*: *path1* @ *path2* ∈ *set* (*field-names-u t v*)
shows ∃ *u*. *path1* ∈ *set* (*field-names-u t u*) ∧ *path2* ∈ *set* (*field-names-u u v*)
proof −

from *append wf-t* **obtain** *n* **where** *fl1*: *field-lookup t* (*path1* @ *path2*) *0* = *Some*
(*v*, *n*)
using *field-names-u-filter-all-field-names-conv*(*1*) **by** *fastforce*
from *fl1*
obtain *u m* **where**
path1: *field-lookup t path1 0* = *Some* (*u*, *m*) **and** *path2*: *field-lookup u path2 m*
= *Some* (*v*, *n*)
using *field-lookup-append-Some wf-t* **by** *blast*
thus *?thesis*
by (*smt* (*verit*, *ccfv-threshold*) *all-field-names-exists-field-names-u*(*1*)
field-lookup-all-field-names(*1*) *field-names-u-composeI field-names-u-filter-all-field-names-conv*(*1*)
local.fl1 mem-Collect-eq option.inject prod.inject set-filter wf-t)
qed

lemma *field-names-u-field-offset-untyped-append*:
assumes *wf-desc root*
assumes *path1*: *path1* ∈ *set* (*field-names-u root outer*)
assumes *path2*: *path2* ∈ *set* (*field-names-u outer inner*)
shows *field-offset-untyped root* (*path1* @ *path2*) = *field-offset-untyped root path1*
+ *field-offset-untyped outer path2*
by (*smt* (*verit*, *ccfv-threshold*) *assms*(1) *field-lookup-offset field-lookup-prefix-Some''*(1)
field-lookup-wf-desc-pres(1) *field-names-u-filter-all-field-names-conv*(1) *field-offset-untyped-def*
mem-Collect-eq option.sel path1 path2 prod.sel(2) *set-filter*)

lemma *root-ptr-valid-u-cases*:
assumes *root-ptr-valid-u t1 d a1*
assumes *root-ptr-valid-u t2 d a2*
shows (*t1* = *t2* ∧ *a1* = *a2*) ∨ ({*a1* ..+ *size-td t1*} ∩ {*a2* ..+ *size-td t2*} = {})
apply (*rule valid-root-footprints-cases*)
using *assms*
by (*auto simp add*: *root-ptr-valid-u-def*)

lemma *field-offset-untyped-empty*[*simp*]: *field-offset-untyped t* [] = *0*
by (*simp add*: *field-offset-untyped-def*)

lemma *field-names-u-refl*[*simp*]: *field-names-u t t* = [[]]
by (*cases t*) *auto*

lemma *field-names-u-size-td-bounds*:
assumes *wf-t*: *wf-desc t*
assumes *path*: *path* ∈ *set* (*field-names-u t u*)
shows *field-offset-untyped t path* + *size-td u* ≤ *size-td t*
proof −
from *wf-t path* **obtain** *n* **where** *fl*: *field-lookup t path 0* = *Some* (*u*, *n*)
using *field-names-u-filter-all-field-names-conv*(1) **by** *auto*

hence *n*: *field-offset-untyped t path* = *n*
by (*simp add*: *field-offset-untyped-def*)
from *field-lookup-offset-size'* [*OF fl*] *n*
show *?thesis* **by** *simp*
qed

lemma *field-lookup-path-cases*:
assumes *fl1*: *field-lookup t path 0* = *Some* (*u*, *n*)
shows (*t* = *u* ∧ *path* = []) ∨ ((*t* ≠ *u*) ∧ *path* ≠ [])
using *field-lookup-same-type-empty*(1) *local.fl1* **by** *blast*

lemma *field-lookup-cycle-cases*:
assumes *fl1*: *field-lookup t path1 0* = *Some* (*u*, *n1*)
assumes *fl2*: *field-lookup u path2 0* = *Some* (*v*, *n2*)
shows (*t* = *v* ∧ *u* = *v* ∧ *path1* = [] ∧ *path2* = [] ∧ *n1* = *0* ∧ *n2* = *0*) ∨

$(t \neq v)$
using *field-lookup-path-cases* [*OF fl1*] *field-lookup-path-cases* [*OF fl2*] *fl1 fl2*
by (*metis add.right-neutral diff-add-0 nat-less-le ordered-cancel-comm-monoid-diff-class.add-diff-inverse td-set-field-lookupD td-set-size-lte'*(*1*))

lemma *field-lookup-refl-iff*: *field-lookup t p n* = *Some* (*t*, *m*) ⟷ *p* = [] ∧ *n* = *m*
proof
assume *fl*: *field-lookup t p n* = *Some* (*t*, *m*)
note *CTypes.field-lookup-offset2*[*OF this*]
from *field-lookup-cycle-cases*[*OF this this*] *fl* **show** *p* = [] ∧ *n* = *m*
by *auto*
qed *auto*

lemma *valid-root-footprint-contained-sub-typ*:
assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *contained*: {*y* ..+ *size-td s*} ⊆ {*x* ..+ *size-td t*}
shows *s* ≤ *t*
proof −
from *valid-y* **have** *0* < *size-td s* **by** (*simp add*: *valid-footprint-def Let-def*)
hence {*y* ..+ *size-td s*} ≠ {}
by (*simp add*: *intvl-non-zero-non-empty*)
with *contained* **have** {*x* ..+ *size-td t*} ∩ {*y* ..+ *size-td s*} ≠ {} **by** *blast*
from *valid-root-footprint-overlap-sub-typ* [*OF valid-root-x valid-y this*]
show *?thesis***.**
qed

lemma *c-null-guard-u-no-overlow*: *c-null-guard-u t a* ⟹ *unat a* + *size-td t* ≤ *addr-card*
unfolding *c-null-guard-u-def addr-card-def*
using *zero-not-in-intvl-no-overflow*
by (*metis card-word*)

lemma *c-null-guard-u-size-td-limit*: *c-null-guard-u t a* ⟹ *size-td t* < *addr-card*
unfolding *c-null-guard-u-def addr-card-def*
by (*metis* (*no-types*, *lifting*) *Abs-fnat-hom-0 add-diff-cancel-right' add-leE cancel-comm-monoid-add-class.diff-cancel*
card-word first-in-intvl linorder-not-less nat-less-le unat-eq-of-nat zero-less-card-finite zero-not-in-intvl-no-overflow)

lemma *c-null-guard-u-intvl-nat-conv*:
assumes *c-null-guard*: *c-null-guard-u t a*
shows {*a* ..+ *size-td t*} = {*x*. (*unat a* ≤ *unat x* ∧ *unat x* < (*unat a* + *size-td t*))}
using *intvl-no-overflow-nat-conv* [*OF c-null-guard-u-no-overlow* [*OF c-null-guard*]]
by *simp*

lemma *valid-footprint-overlap-sub-typ*:
assumes *valid-x*: *valid-footprint d x t*

assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: $\{x\ ..+\ size\text{-}td\ t\} \cap \{y\ ..+\ size\text{-}td\ s\} \neq \{\}$
shows $s \leq t \vee t \leq s$
by (*meson field-of-sub order-less-imp-le overlap*
valid-footprint-neq-disjoint valid-x valid-y)

lemma *valid-footprint-overlap-sub-typ-cases* [*consumes 3*, *case-names eq less1 less2*]:
assumes *valid-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: $\{x\ ..+\ size\text{-}td\ t\} \cap \{y\ ..+\ size\text{-}td\ s\} \neq \{\}$
assumes *eq*: $s = t \Longrightarrow P$
assumes *less-s-t*: $s < t \Longrightarrow P$
assumes *less-t-s*: $t < s \Longrightarrow P$
shows *P*
using *valid-footprint-overlap-sub-typ* [*OF valid-x valid-y overlap*] *eq less-s-t less-t-s*
by *fastforce*

lemma *valid-footprint-contained-sub-typ*:
assumes *valid-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *contained*: $\{x\ ..+\ size\text{-}td\ t\} \subseteq \{y\ ..+\ size\text{-}td\ s\}$
shows $s \leq t \vee t \leq s$
by (*metis intvl-non-zero-non-empty le-iff-inf contained valid-footprint-def*
valid-footprint-overlap-sub-typ valid-x valid-y)

lemma *valid-footprint-contained-sub-typ-cases*:
assumes *valid-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *contained*: $\{x\ ..+\ size\text{-}td\ t\} \subseteq \{y\ ..+\ size\text{-}td\ s\}$
assumes *eq*: $s = t \Longrightarrow P$
assumes *less-s-t*: $s < t \Longrightarrow P$
assumes *less-t-s*: $t < s \Longrightarrow P$
shows *P*
using *valid-footprint-contained-sub-typ* [*OF valid-x valid-y contained*] *eq less-s-t less-t-s*
by *fastforce*

lemma *all-field-names-field-lookup'*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
wf-desc t $\Longrightarrow$ *f* $\in$ *set* (*all-field-names t*) $\Longrightarrow$ $\exists$ *u n*. *field-lookup t f 0* = *Some* (*u*, *n*) **and**
wf-desc-struct st $\Longrightarrow$ *f* $\in$ *set* (*all-field-names-struct st*) $\Longrightarrow$ $\exists$ *u n*. *field-lookup-struct st f 0* = *Some* (*u*, *n*) **and**
wf-desc-list ts $\Longrightarrow$ *f* $\in$ *set* (*all-field-names-list ts*) $\Longrightarrow$ $\exists$ *u n*. *field-lookup-list ts f*

0 = Some (u, n) **and**
wf-desc-tuple x ⟹ f ∈ set (all-field-names-tuple x) ⟹ ∃ u n. field-lookup-tuple x f 0 = Some (u, n)
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f* **and** *f* **and** *f* **and** *f*)
case (*TypDesc nat typ-struct list*)
then show *?case* **by** *auto*
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)
then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case* **apply** (*cases dt-tuple*)
apply (*clarsimp split*: *option.splits simp add*: *all-field-names-list-conv*)
by (*metis field-lookup-list-offset-None not-Some-eq-tuple*)
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **by** *auto*
qed

lemma *valid-foot-print-intvl-self*: *valid-footprint d a t ⟹ a ∈ {a ..+ size-td t}*
by (*simp add*: *valid-footprint-def Let-def intvl-self*)

lemma *field-lookup-Some-field-names-u*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
field-lookup t f n = Some (s, m) ⟹ f ∈ set (field-names-u t s)
field-lookup-struct st f n = Some (s, m) ⟹ f ∈ set (field-names-struct-u st s)
field-lookup-list ts f n = Some (s, m) ⟹ f ∈ set (field-names-list-u ts s)
field-lookup-tuple x f n = Some (s, m) ⟹ f ∈ set (field-names-tuple-u x s)
proof (*induct t* **and** *st* **and** *ts* **and** *x arbitrary*: *f n m* **and** *f n m* **and** *f n m* **and** *f n m*)
case (*TypDesc nat typ-struct list*)
then show *?case*
apply *clarsimp*
by (*metis TypDesc.prems field-lookup-same-type-empty*(*1*))
next
case (*TypScalar nat1 nat2 a*)
then show *?case* **by** *auto*
next
case (*TypAggregate list*)

then show *?case* **by** *auto*
next
case *Nil-typ-desc*
then show *?case* **by** *auto*
next
case (*Cons-typ-desc dt-tuple list*)
then show *?case*
by (*auto split*: *option.splits*)
next
case (*DTuple-typ-desc typ-desc list b*)
then show *?case* **apply** *clarsimp*
by (*metis imageI list.exhaust-sel option.distinct*(*1*))
qed

lemma *field-lookup-non-prefix-disj'*:
assumes *wf*: *mem-type-u t*
assumes *f*: *field-lookup t f 0* = *Some* (*tf*, *n*)
assumes *g*: *field-lookup t g 0* = *Some* (*tg*, *m*)
assumes *f-g*: *disj-fn f g*
shows *disjnt* {(*of-nat n*::*addr*) *..+ size-td tf*} {*of-nat m ..+ size-td tg*}
using *mem-type-u.max-size*[*OF wf*] *f-g*
using *field-lookup-offset-size'*[*OF f*]
using *field-lookup-offset-size'*[*OF g*]
using *field-lookup-non-prefix-disj*[*OF wf*[*THEN mem-type-u.wf-desc*] *f g*]
unfolding *intvl-eq-of-nat-Ico-add*
apply (*subst disjnt-of-nat*)
unfolding *len-of-addr-card*
by (*auto simp add*: *disjnt-def disj-fn-def less-eq-list-def*)

lemma *sub-typ-stack-byte-u*:
t ≤ *typ-uinfo-t* (*TYPE*(*stack-byte*)) ⟹ *t* = *typ-uinfo-t TYPE*(*stack-byte*)
by (*simp add*: *sub-typ-def typ-uinfo-t-def typ-info-stack-byte typ-tag-le-def*)

lemma *root-ptr-valid-not-subtype-disjoint-u*:
⟦ *root-ptr-valid d* (*p*::*'a*::*mem-type ptr*);
cvalid-u t d q;
¬ *t* ≤ *typ-uinfo-t TYPE*(*'a*) ⟧ ⟹
ptr-span p ∩ {*q ..+ size-td t*} = {}
by (*metis cvalid-u-def root-ptr-valid-valid-root-footprint size-of-fold*
typ-uinfo-size valid-root-footprint-overlap-sub-typ)

lemma *stack-allocs-disjoint-u*:
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 (*TYPE*(*'a*::*mem-type*)) *d*
assumes *no-stack*: *t* ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *cvalid-u t d q*
shows {*ptr-val p ..+ n* ∗ *size-of TYPE*(*'a*)} ∩ {*q ..+ size-td t*} = {}
proof −

from *stack-alloc* **obtain**
typ-uinfo-t (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) **and**
stack-bytes: ∀ *a* ∈ {*ptr-val p ..+ n * size-of TYPE*('*a*)}. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
by (*cases rule*: *stack-allocs-cases*) *auto*
from *no-stack* **have** *no-sub-typ*: ¬ *t* ≤ *typ-uinfo-t* (*TYPE*(*stack-byte*)) **by** (*metis sub-typ-stack-byte-u*)
{
fix *a*
assume *a*: *a* ∈ {*ptr-val p ..+ n * size-of TYPE*('*a*)}
have *a* ∉ {*q ..+ size-td t*}
proof −
from *stack-bytes* [*rule-format*, *OF a*] **have** *root-ptr-valid d* (*PTR* (*stack-byte*) *a*) **.**

from *root-ptr-valid-not-subtype-disjoint-u* [*OF this typed no-sub-typ*] **show** *?thesis*
by (*simp add*: *disjoint-iff first-in-intvl*)
qed
}
then show {*ptr-val p ..+ n * size-of TYPE*('*a*)} ∩ {*q ..+ size-td t*} = {}
by *auto*
qed

lemma *fold-ptr-retyp-disjoint-u*:
fixes *p*::'*a*::*mem-type ptr*
shows ⟦ *cvalid-u t d q*; {*ptr-val p..+ n * size-of TYPE*('*a*)} ∩ {*q ..+ size-td t*} = {} ⟧ ⟹
cvalid-u t (*fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [*0..<n*] *d*) *q*
apply(*clarsimp simp*: *cvalid-u-def*)
apply(*erule fold-ptr-retyp-valid-footprint-disjoint*)
apply(*fastforce simp*: *size-of-def*)
done

lemma *fold-ptr-force-type-disjoint-u*:
fixes *p*::'*a*::*mem-type ptr*
shows ⟦ *cvalid-u t d q*; {*ptr-val p..+ n * size-of TYPE*('*a*)} ∩ {*q ..+ size-td t*} = {} ⟧ ⟹
cvalid-u t (*fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0..<n*] *d*) *q*
apply(*clarsimp simp*: *cvalid-u-def*)
apply(*erule fold-ptr-force-type-valid-footprint-disjoint*)
apply(*fastforce simp*: *size-of-def*)
done

lemma *fold-ptr-retyp-disjoint2-u*:
fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n * size-of TYPE*('*a*)}
shows ⟦*cvalid-u t* (*fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [*0..<n*] *d*) *q*;
{*ptr-val p..+ n * size-of TYPE*('*a*)} ∩ {*q ..+ size-td t*}= {} ⟧

$\Longrightarrow$ *cvalid-u t d q*
apply(*clarsimp simp*: *cvalid-u-def*)
apply(*erule fold-ptr-retyp-valid-footprint-disjoint2* [*OF no-overflow*])
apply(*simp add*: *size-of-def*)
apply *fast*
done

lemma *fold-ptr-force-type-disjoint2-u*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* $\notin$ {*ptr-val p..+ n * size-of TYPE('a)*}
shows ⟦*cvalid-u t* (*fold* (*λi. ptr-force-type* (*p* $+_p$ *int i*)) [*0..<n*] *d*) *q*;
{*ptr-val p..+ n * size-of TYPE('a)*} ∩ {*q ..+ size-td t*}= {} ⟧
$\Longrightarrow$ *cvalid-u t d q*
apply(*clarsimp simp*: *cvalid-u-def*)
apply(*erule fold-ptr-force-type-valid-footprint-disjoint2* [*OF no-overflow*])
apply(*simp add*: *size-of-def*)
apply *fast*
done

lemma *fold-ptr-retyp-disjoint-iff-u*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* $\notin$ {*ptr-val p..+ n * size-of TYPE('a)*}
shows {*ptr-val p..+ n * size-of TYPE('a)*} ∩ {*q ..+ size-td t*} = {}
$\Longrightarrow$ *cvalid-u t* (*fold* (*λi. ptr-retyp* (*p* $+_p$ *int i*)) [*0..<n*] *d*) *q* = *cvalid-u t d q*
apply *standard*
apply (*erule* (*1*) *fold-ptr-retyp-disjoint2-u* [*OF no-overflow*])
apply (*erule* (*1*) *fold-ptr-retyp-disjoint-u*)
done

lemma *fold-ptr-force-type-disjoint-iff-u*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* $\notin$ {*ptr-val p..+ n * size-of TYPE('a)*}
shows {*ptr-val p..+ n * size-of TYPE('a)*} ∩ {*q ..+ size-td t*} = {}
$\Longrightarrow$ *cvalid-u t* (*fold* (*λi. ptr-force-type* (*p* $+_p$ *int i*)) [*0..<n*] *d*) *q* = *cvalid-u t d q*
apply *standard*
apply (*erule* (*1*) *fold-ptr-force-type-disjoint2-u* [*OF no-overflow*])
apply (*erule* (*1*) *fold-ptr-force-type-disjoint-u*)
done

lemma *stack-allocs-preserves-typing-u*:
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*(*'a*::*mem-type*)) *d*
assumes *no-stack*: *t* ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *cvalid-u t d q*
shows *cvalid-u t d' q*
proof −
from *stack-alloc* **obtain**
d': *d'* = *fold* (*λi. ptr-force-type* (*p* $+_p$ *int i*)) [*0..<n*] *d* **and**
no-overflow: *0* $\notin$ {*ptr-val p ..+ n * size-of TYPE('a)*}
by (*cases rule*: *stack-allocs-cases*) *auto*

from *stack-allocs-disjoint-u* [*OF stack-alloc no-stack typed*]
have {*ptr-val p..+n * size-of TYPE('a)*} ∩ {*q..+size-td t*} = {} .
from *fold-ptr-force-type-disjoint-iff-u* [*OF no-overflow this*, **where** *d=d*] *typed*
show *?thesis*
by (*simp add*: *d'*)
qed

lemma *stack-allocs-root-ptr-valid-u-new-cases*:
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 (*TYPE('a::mem-type)*) *d*
assumes *valid*: *root-ptr-valid-u t d' q*
shows (∃ *i<n*. *q* = *ptr-val* (*p* +$_p$ *int i*) ∧ *t* = *typ-uinfo-t* (*TYPE('a)*)) ∨ *root-ptr-valid-u t d q*
proof (*cases* {*ptr-val p ..+ n * size-of TYPE('a)*} ∩ {*q ..+ size-td t*} = {})
case *True*
with *stack-alloc valid* **show** *?thesis*
by (*smt* (*verit*) *disjoint-iff fold-ptr-force-type-other intvlI root-ptr-valid-u-def stack-allocs-cases valid-root-footprint-def*)
next
case *False*
with *stack-alloc array-to-index-span stack-alloc valid* **show** *?thesis*
by (*smt* (*verit*, *ccfv-SIG*) *disjoint-iff root-ptr-valid-root-ptr-valid-u-conv root-ptr-valid-u-cases size-of-tag stack-allocs-root-ptr-valid-same-type-cases*)
qed

lemma *cvalid-u-c-guard-u*:
cvalid-u t d a ⟹ *c-guard-u t a*
by (*simp add*: *cvalid-u-def*)

lemma *stack-allocs-preserves-root-ptr-valid-u*:
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 (*TYPE('a::mem-type)*) *d*
assumes *no-stack*: *t* ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *root-ptr-valid-u t d q*
shows *root-ptr-valid-u t d' q*
proof −
from *stack-alloc* **obtain**
d': *d'* = *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0..<n*] *d* **and**
no-overflow: *0* ∉ {*ptr-val p ..+ n * size-of TYPE('a)*}
by (*cases rule*: *stack-allocs-cases*) *auto*

from *typed* **have** *typed-q*: *cvalid-u t d q*
by (*simp add*: *root-ptr-valid-u-cvalid-u*)

from *stack-allocs-disjoint-u* [*OF stack-alloc no-stack this*]
have *disj*: {*ptr-val p..+n * size-of TYPE('a)*} ∩ {*q..+size-td t*} = {} .
from *stack-allocs-preserves-typing-u* [*OF stack-alloc no-stack typed-q*]
have *typed'*: *cvalid-u t d' q* .
hence *valid-fp*: *valid-footprint d' q t*
by (*simp add*: *cvalid-u-def*)

```
  show ?thesis
   apply (simp add: root-ptr-valid-u-def valid-root-footprint-valid-footprint-dom-conv
valid-fp
        cvalid-u-c-guard-u [OF typed'])
    apply (simp add: d')
    using disj fold-ptr-force-type-other
    by (smt (verit) d' disjoint-iff intvlI root-ptr-valid-u-def stack-alloc
        stack-allocs-cases typed valid-root-footprint-dom-typing)
qed

lemma stack-allocs-root-ptr-valid-u-other:
  assumes stack-alloc: (p, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
  assumes valid-d: root-ptr-valid-u t d q
  assumes non-stack: t ≠ typ-uinfo-t (TYPE(stack-byte))
  shows root-ptr-valid-u t d' q
proof (cases root-ptr-valid-u t d' q)
  case True
  then show ?thesis by simp
next
  case False
  from stack-alloc
  show ?thesis
    apply (rule stack-allocs-cases)
    using False valid-d non-stack
    using stack-alloc stack-allocs-preserves-root-ptr-valid-u by blast
qed

lemma stack-allocs-root-ptr-valid-u-same:
  assumes stack-alloc: (p, d') ∈ stack-allocs n S (TYPE('a::mem-type)) d
  assumes i: i < n
  assumes addr-eq: q = ptr-val (p +p int i)
  assumes match: t = typ-uinfo-t (TYPE('a))
  shows root-ptr-valid-u t d' q
proof (cases root-ptr-valid-u t d' q)
  case True
  then show ?thesis by simp
next
  case False
  from stack-alloc have root-ptr-valid d' (p +p int i)
    apply  (rule stack-allocs-cases)
    using i
    by auto

  with addr-eq match show ?thesis
    apply (clarsimp simp add: root-ptr-valid-def root-ptr-valid-u-def)
    apply (simp add: c-guard-def c-guard-u-def c-null-guard-def c-null-guard-u-def
        ptr-aligned-def ptr-aligned-u-def)
    apply (auto simp add: align-of-def align-td-uinfo[symmetric] size-of-def  )
```

done
qed

lemma *stack-allocs-root-ptr-valid-u-cases*:
assumes *stack-alloc*: $(p,\ d') \in$ *stack-allocs* $n\ \mathcal{S}$ $(TYPE('a::mem\text{-}type))\ d$
assumes *non-stack-byte*: $t \neq$ *typ-uinfo-t* $(TYPE(stack\text{-}byte))$
shows *root-ptr-valid-u* $t\ d'\ q \longleftrightarrow$
$(\exists\, i{<}n.\ q =$ *ptr-val* $(p +_p$ *int* $i) \wedge t =$ *typ-uinfo-t* $(TYPE('a))) \vee$
root-ptr-valid-u $t\ d\ q$

using *stack-alloc non-stack-byte*
stack-allocs-root-ptr-valid-u-new-cases *stack-allocs-root-ptr-valid-u-other*
stack-allocs-root-ptr-valid-u-same
by *metis*

lemma *stack-releases-root-ptr-valid-u1*:
fixes $p::'a::mem\text{-}type$ *ptr*
assumes *non-stack-p*: *typ-uinfo-t* $TYPE('a) \neq$ *typ-uinfo-t* $TYPE(stack\text{-}byte)$
assumes *non-stack-q*: $t \neq$ *typ-uinfo-t* $TYPE(stack\text{-}byte)$
assumes *root-q*: *root-ptr-valid-u* t $(stack\text{-}releases\ n\ p\ d)\ q$
shows $\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a::mem\text{-}type)\} \cap \{q\ ..+\ size\text{-}td\ t\} = \{\}$
$\wedge$ *root-ptr-valid-u* $t\ d\ q$
apply (*rule context-conjI*)
subgoal
using *assms*
by (*smt* (*verit*, *best*) *cvalid-u-def disjoint-iff in-ptr-span-itself ptr-val.ptr-val-def*
root-ptr-valid-u-cvalid-u size-of-tag stack-releases-valid-root-footprint sub-typ-stack-byte-u
valid-root-footprint-overlap-sub-typ)
subgoal
using *assms*
by (*smt* (*verit*, *best*) *disjoint-iff intvlI root-ptr-valid-u-def stack-releases-other*
valid-root-footprint-def)
done

lemma *stack-releases-root-ptr-valid-u2*:
fixes $p::'a::mem\text{-}type$ *ptr*
assumes *disj*: $\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a::mem\text{-}type)\} \cap \{q\ ..+\ size\text{-}td\ t\} = \{\}$
assumes *valid-q*: *root-ptr-valid-u* $t\ d\ q$
shows *root-ptr-valid-u* t $(stack\text{-}releases\ n\ p\ d)\ q$
using *assms*
by (*simp add*: *intvlI orthD2 root-ptr-valid-u-def stack-releases-other valid-root-footprint-def*)

lemma *stack-release-root-ptr-valid-u2*:
fixes $p::'a::mem\text{-}type$ *ptr*
assumes *disj*: *ptr-span* $p \cap \{q\ ..+\ size\text{-}td\ t\} = \{\}$
assumes *valid-q*: *root-ptr-valid-u* $t\ d\ q$
shows *root-ptr-valid-u* t $(stack\text{-}release\ p\ d)\ q$
using *assms*

by (*smt* (*verit*, *best*) *disjoint-iff intvlI root-ptr-valid-u-def stack-release-other valid-root-footprint-def*)

lemma *stack-releases-root-ptr-valid-u-cases*:
fixes *p*::'*a*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *t* ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid-u t* (*stack-releases n p d*) *q* ⟷
{*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*::*mem-type*)} ∩ {*q* ..+ *size-td t*} = {} ∧ *root-ptr-valid-u t d q*
using *assms stack-releases-root-ptr-valid-u1 stack-releases-root-ptr-valid-u2* **by** *blast*

lemma *valid-root-footprint-is-root*:
assumes *wf*: *wf-desc t*
assumes *wf-size*: *wf-size-desc t*
assumes *f*: *field-lookup t path 0* = *Some* (*s*, *n*)
assumes *footprint-t*: *valid-footprint d a t*
assumes *root-s*: *valid-root-footprint d* (*a* + *of-nat n*) *s*
shows (*t* = *s* ∧ *path* = [])
proof (*cases path*)
case *Nil*
with *f* **have** *t* = *s*
by *simp*
with *Nil* **show** *?thesis* **by** *simp*
next
case (*Cons fld path'*)
from *f* **have** *neq*: *t* ≠ *s*
using *field-lookup-same-type-empty*(*1*) *local.Cons* **by** *blast*
from *f* **have** *s-t*: *s* ≤ *t*
using *td-set-field-lookupD typ-tag-le-def* **by** *blast*
from *neq s-t* **have** ¬ *t* ≤ *s*
using *sub-tag-antisym* **by** *blast*
with *valid-root-footprint-overlap-sub-typ* [*OF root-s footprint-t*]
have {*a* + *word-of-nat n*..+*size-td s*} ∩ {*a*..+*size-td t*} = {}
by *blast*
moreover
from *f* **have** *td-set*: (*s*,*n*) ∈ *td-set t 0*
using *local.wf td-set-field-lookup* **by** *blast*
from *td-set-offset-size* [*OF td-set*] **have** {*a* + *word-of-nat n*..+*size-td s*} ⊆ {*a*..+*size-td t*}
by (*simp add*: *intvl-le*)
moreover
from *field-lookup-wf-size-desc-gt* [*OF f wf-size*] **have** *0* < *size-td s* **.**
hence {*a* + *word-of-nat n*..+*size-td s*} ≠ {}
by (*simp add*: *intvl-non-zero-non-empty*)
ultimately have *False*
by *blast*
then show *?thesis* **by** *simp*

qed

corollary *root-ptr-valid-u-is-root*:
assumes *wf*: *wf-desc t*
assumes *wf-size*: *wf-size-desc t*
assumes *f*: *field-lookup t path 0 = Some* (*s*, *n*)
assumes *cvalid-u-t*: *cvalid-u t d a*
assumes *root-s*: *root-ptr-valid-u s d* (*a* + *of-nat n*)
shows (*t* = *s* ∧ *path* = [])
proof −
from *cvalid-u-t* **have** *footprint-t*: *valid-footprint d a t* **by** (*simp add*: *cvalid-u-def*)
from *root-s* **have** *valid-root-footprint d* (*a* + *of-nat n*) *s* **by** (*simp add*: *root-ptr-valid-u-def*)
from *valid-root-footprint-is-root* [*OF wf wf-size f footprint-t this*] **show** *?thesis* **.**
qed

lemma *valid-footprint-field-lookup*:
assumes *wf*: *wf-desc t*
assumes *wf-size-desc*: *wf-size-desc t*
assumes *valid-t*: *valid-footprint d a t*
assumes *fl*: *field-lookup t path1 0 = Some* (*s*, *n*)
shows *valid-footprint d* (*a* + *word-of-nat n*) *s*
using *valid-t fl wf-size-desc*
apply (*clarsimp simp add*: *valid-footprint-def Let-def*, *intro conjI*)
subgoal using *field-lookup-wf-size-desc-gt* **by** *blast*
subgoal by (*smt* (*verit*, *ccfv-SIG*) *Abs-fnat-hom-add add.commute add-less-mono1 field-lookup-offset-size′*
group-cancel.add2 map-list-map-trans order-less-le-trans td-set-field-lookupD typ-slice-td-set)
done

lemma *in-set-mapD*:*x* ∈ *set* (*map f xs*) ⟹ ∃ *y* ∈ *set xs*. *x* = *f y*
by (*induct xs*) *auto*

lemma *findSomeI*: *x* ∈ *set xs* ⟹ *P x* ⟹ ∃ *x*. *find P xs* = *Some x*
by (*induct xs*) *auto*

lemma *find-in-set-inj-distinct*:
inj-on P (*set* (*map fst xs*)) ⟹ (*x*, *v*) ∈ *set xs* ⟹ *P x* ⟹ *distinct* (*map fst xs*) ⟹
find (λ(*x*, -). *P x*) *xs* = *Some* (*x*, *v*)
by (*induct xs*) (*auto simp add*: *injD rev-image-eqI*)

definition *inj-on-true P A* = (∀ *x y*. *x* ∈ *A* ⟶ *y* ∈ *A* ⟶ *P x* ⟶ *P y* ⟶ *x* = *y*)

lemma *inj-on P A* ⟹ *inj-on-true P A*
by (*auto simp add*: *inj-on-def inj-on-true-def*)

lemma *inj-on-trueD*: *inj-on-true P A* $\Longrightarrow$ $x \in A \Longrightarrow y \in A \Longrightarrow P\ x \Longrightarrow P\ y \Longrightarrow x = y$
by (*simp add*: *inj-on-true-def*)

lemma *inj-on-trueI*: $(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow P\ x \Longrightarrow P\ y \Longrightarrow x = y) \Longrightarrow$ *inj-on-true P A*
by (*simp add*: *inj-on-true-def*)

lemma *inj-on-true-mono*: $A \subseteq B \Longrightarrow$ *inj-on-true P B* $\Longrightarrow$ *inj-on-true P A*
by (*auto simp add*: *inj-on-true-def*)

lemma *find-in-set-inj-on-true-distinct*:
inj-on-true P (*set* (*map fst xs*)) $\Longrightarrow$ $(x, v) \in$ *set xs* $\Longrightarrow$ *P x* $\Longrightarrow$ *distinct* (*map fst xs*) $\Longrightarrow$
find ($\lambda(x, \text{-}).\ P\ x$) *xs* = *Some* (x, v)
apply (*induct xs*)
apply *simp*
apply *simp*
by (*smt* (*verit*, *best*) *case-prod-beta′ fst-conv image-iff inj-on-true-def insertCI*)

lemma *find-in-set-inj-on-true-distinct′*:
inj-on-true P (*set xs*) $\Longrightarrow$ $x \in$ *set xs* $\Longrightarrow$ *P x* $\Longrightarrow$ *distinct xs* $\Longrightarrow$
find ($\lambda x.\ P\ x$) *xs* = *Some x*
apply (*induct xs*)
apply *simp*
apply *simp*
by (*metis inj-on-true-def insertCI*)

lemma *typ-tag-le-size-le*: $t <$ (*u*::*typ-uinfo*) $\Longrightarrow$ *size t* < *size u*
using *td-set-size-lte*
by (*metis typ-tag-le-def typ-tag-lt-def*)

lemma *c-null-guard-intvl-nat-conv*:
fixes *p*::′*a*::*mem-type ptr*
assumes *c-null-guard*: *c-null-guard p*
shows *ptr-span p* = {*x*. (*unat* (*ptr-val p*) $\leq$ *unat x* $\wedge$ *unat x* < (*unat* (*ptr-val p*) + *size-of TYPE*(′*a*)))}
proof −
from *c-null-guard*
have *c-null-guard-u* (*typ-uinfo-t TYPE*(′*a*)) (*ptr-val p*)
by (*simp add*: *c-null-guard-c-null-guard-u-conv*)
from *c-null-guard-u-intvl-nat-conv* [*OF this*]
show *?thesis*
by (*simp add*: *size-of-def*)
qed

lemma *c-null-guard-ptr-add*:
fixes *p*::′*a*::*mem-type ptr*

assumes *bound*: *unat* (*ptr-val p*) + *Suc n* ∗ *size-of TYPE*('*a*) ≤ *addr-card*
assumes *not-null*: *ptr-val p* ≠ *0*
assumes *le*: *i* ≤ *n*
shows *c-null-guard* (*p* +$_p$ *int i*)
proof −

from *bound le* **have** *bound1*: *unat* (*ptr-val p*) + *i* ∗ *size-of TYPE*('*a*) ≤ *addr-card*
by (*meson add-le-mono dual-order.eq-iff dual-order.trans le-SucI mult-le-cancel2*)
from *bound le* **have** *bound2*: *unat* (*ptr-val p*) + *i* ∗ *size-of TYPE*('*a*) + *size-of TYPE*('*a*) ≤ *addr-card*
by (*smt* (*verit*, *ccfv-SIG*) *bound1 less-le-mult mult-strict-right-mono not-less-eq-eq of-nat-add of-nat-le-iff of-nat-mult*)

show *?thesis*
apply (*clarsimp simp add*: *ptr-add-def c-null-guard-def intvl-def*)
using *bound1 bound2 not-null*
by (*smt* (*verit*, *del-insts*) *Abs-fnat-hom-add Abs-fnat-hom-mult add-eq-0-iff-both-eq-0 add-mono-thms-linordered-field*(*4*) *eq-imp-le le-Suc-ex len-of-addr-card trans-less-add1 unat-of-nat-len unsigned-0 word-unat.Rep-inverse*′)
qed

lemma *ptr-add-disjoint-index*:
fixes *p*::′*a*::*mem-type ptr*
assumes *bound*: *unat* (*ptr-val p*) + *Suc n* ∗ *size-of TYPE*('*a*) ≤ *addr-card*
assumes *le*: *i* < *n*
assumes *n*: *c-null-guard* (*p* +$_p$ *int n*)
assumes *i*: *c-null-guard* (*p* +$_p$ *int i*)
shows *ptr-span* (*p* +$_p$ *int n*) ∩ *ptr-span* (*p* +$_p$ *int i*) = {}
proof −
from *bound le* **have** *bound-i*: *unat* (*ptr-val p*) + *Suc i* ∗ *size-of TYPE*('*a*) ≤ *addr-card*
by (*smt* (*verit*, *best*) *Suc-less-SucD add-le-cancel-left dual-order.strict-iff-not le-trans mult-less-cancel2 nat-le-linear*)

from *bound* **have** *bound1*: *unat* (*ptr-val p*) + *n* ∗ *size-of TYPE*('*a*) ≤ *addr-card*
by *simp*
from *bound* **have** *bound2*: *unat* (*ptr-val p*) + *n* ∗ *size-of TYPE*('*a*) + *size-of TYPE*('*a*) ≤ *addr-card*
by *simp*

from *bound-i* **have** *bound-i1*: *unat* (*ptr-val p*) + *i* ∗ *size-of TYPE*('*a*) ≤ *addr-card*
by *simp*
from *bound-i* **have** *bound-i2*: *unat* (*ptr-val p*) + *i* ∗ *size-of TYPE*('*a*) + *size-of TYPE*('*a*) ≤ *addr-card*
by *simp*

show *?thesis*
apply (*simp add*: *c-null-guard-intvl-nat-conv n i*)

```
    using le bound n i bound1 bound2 bound-i1 bound-i2
    by (auto simp add: ptr-add-def)
       (smt (verit) len-of-addr-card less-le-mult mult-strict-right-mono of-nat-add
of-nat-inverse of-nat-le-iff of-nat-less-iff of-nat-mult word-unat.Rep-inverse)
qed

lemma ptr-add-disjoint-last:
  fixes p::'a::mem-type ptr
  assumes bound:  unat (ptr-val p) + Suc n * size-of TYPE('a) ≤ addr-card
  assumes not-null: ptr-val p ≠ 0
  shows  {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span (p +p int n) = {}
proof −
  {
    fix i assume i-bound: i<n
    have ptr-span (p +p int i) ∩ ptr-span (p +p int n) = {}
    proof −
      from c-null-guard-ptr-add [OF bound not-null] i-bound obtain
        c-null-guard (p +p int n) and c-null-guard (p +p int i) by auto
      from ptr-add-disjoint-index [OF bound i-bound this]
      show ?thesis by blast
    qed
  }

  then show ?thesis
    by (auto simp add: array-index-span-conv)
qed

lemma h-val-fold:
  fixes p::'a::mem-type ptr
  assumes bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card
  assumes not-null: ptr-val p ≠ 0
  assumes i-bound: i < n
  shows h-val ((fold (λi. heap-update (p +p int i) v) [0..<n]) h) (p +p int i) = v
  using bound i-bound
proof (induct n arbitrary: h i)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems obtain i-bound: i < Suc n and bound: unat (ptr-val p) + Suc
n * size-of TYPE('a) ≤ addr-card by blast
  hence bound': unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card by simp
  show ?case
  proof (cases i = n)
    case True
    show ?thesis by (simp add: True)
  next
    case False
    with i-bound have i-bound': i < n by simp
```

```
    note hyp = Suc.hyps [OF bound' i-bound']
    from c-null-guard-ptr-add [OF bound not-null, of n]
    have null-guard-n: c-null-guard (p +_p int n) by simp
    from c-null-guard-ptr-add [OF bound not-null, of i] i-bound'
    have null-guard-i: c-null-guard (p +_p int i) by simp
    from ptr-add-disjoint-index [OF bound i-bound' null-guard-n null-guard-i]
    have disj: ptr-span (p +_p int n) ∩ ptr-span (p +_p int i) = {} .
    show ?thesis
      by (simp add: h-val-update-regions-disjoint [OF disj] hyp)
  qed
qed

lemma h-val-fold-disjoint:
  fixes p:: 'a::mem-type ptr
  fixes q:: 'b::mem-type ptr
  assumes disj: {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {}
  shows h-val (fold (λi. heap-update (p +_p int i) (v i)) [0..<n] h) q =
        h-val h q
  using disj
proof (induct n arbitrary: h)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems have disj: {ptr-val p..+Suc n * size-of TYPE('a)} ∩ ptr-span
q = {} .
  from disj have disj': {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {}
     by (smt (verit, best) first-in-intvl intvlD intvlI intvl-inter le-add2 mult-Suc
order-less-le-trans orthD2 sz-nzero)
  have ptr-span (p +_p int n) ⊆{ptr-val p..+Suc n * size-of TYPE('a)}
   by (metis (no-types, lifting) CTypesDefs.ptr-add-def intvl-le mult-Suc nat-le-linear
       of-int-of-nat-eq of-nat-mult ptr-val.simps)

  with disj have disj-n: ptr-span (p +_p int n) ∩ ptr-span q = {}
    using intvl-le by fastforce

  note hyp = Suc.hyps [OF disj']
  show ?case by (simp add: h-val-update-regions-disjoint [OF disj-n] hyp)
qed

lemma h-val-fold-zero-disjoint:
  fixes p:: 'a::mem-type ptr
  fixes q:: 'b::mem-type ptr
  assumes disj: {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {}
  shows h-val (fold (λi. heap-update (p +_p int i) c-type-class.zero) [0..<n] h) q =
        h-val h q
  using disj
  by (rule  h-val-fold-disjoint)
```

lemma *fold-heap-update-commute*:
fixes *p*::*'a*:: *mem-type ptr*
fixes *q*::*'b*:: *mem-type ptr*
assumes *disjoint*: {*ptr-val p ..+ n * size-of TYPE*(*'a*)} ∩ *ptr-span q* = {}
shows *fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) (*v i*)) [*0*..<*n*] (*heap-update q x h*) =
(*heap-update q x*) (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) (*v i*)) [*0*..<*n*] *h*)
using *disjoint*
proof (*induct n arbitrary*: *h*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **have** *disj*: {*ptr-val p..+Suc n * size-of TYPE*(*'a*)} ∩ *ptr-span q* = {} .
from *disj* **have** *disj'*: {*ptr-val p..+n * size-of TYPE*(*'a*)} ∩ *ptr-span q* = {}
by (*smt* (*verit*, *best*) *first-in-intvl intvlD intvlI intvl-inter le-add2 mult-Suc order-less-le-trans orthD2 sz-nzero*)
have *ptr-span* (*p* +$_p$ *int n*) ⊆{*ptr-val p..+Suc n * size-of TYPE*(*'a*)}
by (*metis* (*no-types*, *lifting*) *CTypesDefs.ptr-add-def intvl-le mult-Suc nat-le-linear of-int-of-nat-eq of-nat-mult ptr-val.simps*)

with *disj* **have** *disj-n*: *ptr-span* (*p* +$_p$ *int n*) ∩ *ptr-span q* = {}
using *intvl-le* **by** *fastforce*
note *hyp* = *Suc.hyps* [*OF disj'*]
note *commute* = *heap-update-commute* [*OF disj-n*, *simplified*]
show *?case* **apply** *simp*
by (*simp add*: *hyp commute*)
qed

lemma *fold-heap-update-padding-commute*:
fixes *p*::*'a*:: *mem-type ptr*
fixes *q*::*'b*:: *mem-type ptr*
assumes *disjoint*: {*ptr-val p ..+ n * size-of TYPE*(*'a*)} ∩ *ptr-span q* = {}
assumes *lbs*: ⋀*i*. *i* < *n* ⟹ *length* (*bs i*) = *size-of TYPE*(*'a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*(*'b*)
shows *fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*v i*) (*bs i*)) [*0*..<*n*] (*heap-update-padding q x bs' h*) =
(*heap-update-padding q x bs'*) (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*v i*) (*bs i*)) [*0*..<*n*] *h*)
using *disjoint lbs*
proof (*induct n arbitrary*: *h bs*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain** *disj*: {*ptr-val p..+Suc n * size-of TYPE*(*'a*)} ∩ *ptr-span q* = {} **and**
lbs-n: ⋀*i*. *i* < *n* ⟹ *length* (*bs i*) = *size-of TYPE*(*'a*) **and** *lbs-n'*: *length* (*bs n*) = *size-of TYPE*(*'a*)

by *auto*
from *disj* **have** *disj'*: {*ptr-val p..+n * size-of TYPE('a)*} ∩ *ptr-span q* = {}
by (*smt* (*verit*, *best*) *first-in-intvl intvlD intvlI intvl-inter le-add2 mult-Suc order-less-le-trans orthD2 sz-nzero*)
have *ptr-span* (*p* +$_p$ *int n*) ⊆{*ptr-val p..+Suc n * size-of TYPE('a)*}
by (*metis* (*no-types*, *lifting*) *CTypesDefs.ptr-add-def intvl-le mult-Suc nat-le-linear of-int-of-nat-eq of-nat-mult ptr-val.simps*)

with *disj* **have** *disj-n*: *ptr-span* (*p* +$_p$ *int n*) ∩ *ptr-span q* = {}
using *intvl-le* **by** *fastforce*
note *hyp* = *Suc.hyps* [*OF disj' lbs-n*]
note *commute* = *heap-update-padding-commute* [*OF disj-n lbs-n' lbs'*, *simplified*]
show *?case*
apply *simp*
apply (*simp add*: *hyp commute*)
done
qed

lemma *fold-heap-update-padding-heap-update-commute*:
fixes *p*::'*a*:: *mem-type ptr*
fixes *q*::'*b*:: *mem-type ptr*
assumes *disjoint*: {*ptr-val p ..+ n * size-of TYPE('a)*} ∩ *ptr-span q* = {}
assumes *lbs*: ⋀*i*. *i* < *n* ⟹ *length* (*bs i*) = *size-of TYPE('a)*
shows *fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*v i*) (*bs i*)) [*0..<n*] (*heap-update q x h*) =
(*heap-update q x*) (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*v i*) (*bs i*)) [*0..<n*] *h*)
using *disjoint lbs*
proof (*induct n arbitrary*: *h bs*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain** *disj*: {*ptr-val p..+Suc n * size-of TYPE('a)*} ∩ *ptr-span q* = {} **and**
lbs-n: ⋀*i*. *i* < *n* ⟹ *length* (*bs i*) = *size-of TYPE('a)* **and** *lbs-n'*: *length* (*bs n*) = *size-of TYPE('a)*
by *auto*
from *disj* **have** *disj'*: {*ptr-val p..+n * size-of TYPE('a)*} ∩ *ptr-span q* = {}
by (*smt* (*verit*, *best*) *first-in-intvl intvlD intvlI intvl-inter le-add2 mult-Suc order-less-le-trans orthD2 sz-nzero*)
have *ptr-span* (*p* +$_p$ *int n*) ⊆{*ptr-val p..+Suc n * size-of TYPE('a)*}
by (*metis* (*no-types*, *lifting*) *CTypesDefs.ptr-add-def intvl-le mult-Suc nat-le-linear of-int-of-nat-eq of-nat-mult ptr-val.simps*)

with *disj* **have** *disj-n*: *ptr-span* (*p* +$_p$ *int n*) ∩ *ptr-span q* = {}
using *intvl-le* **by** *fastforce*
note *hyp* = *Suc.hyps* [*OF disj' lbs-n*]
note *commute* = *heap-update-padding-heap-update-commute* [*OF disj-n lbs-n'* ,

simplified]
show *?case*
apply *simp*
apply (*simp add*: *hyp commute*)
done
qed

lemma *fold-heap-update-collapse*:
fixes *p*::*'a*::*xmem-type ptr*
assumes *bound*: *unat* (*ptr-val p*) + *n* ∗ *size-of TYPE*(*'a*) ≤ *addr-card*
assumes *not-null*: *ptr-val p* ≠ *0*
shows *fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) (*v i*)) [*0*..<*n*]
((*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) (*w i*)) [*0*..<*n*]) *h*) =
fold (λ*i*. *heap-update* (*p* +$_p$ *int i*) (*v i*)) [*0*..<*n*] *h*
using *bound*
proof (*induct n arbitrary*: *h*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain** *bound*: *unat* (*ptr-val p*) + *Suc n* ∗ *size-of TYPE*(*'a*) ≤ *addr-card* **by** *blast*
hence *bound'*: *unat* (*ptr-val p*) + *n* ∗ *size-of TYPE*(*'a*) ≤ *addr-card* **by** *simp*
from *ptr-add-disjoint-last* [*OF bound not-null*]
have *disj*: {*ptr-val p*..+*n* ∗ *size-of TYPE*(*'a*)} ∩ *ptr-span* (*p* +$_p$ *int n*) = {} **.**
show *?case*
apply *simp*
apply (*subst* (*2*) *Suc.hyps* [*OF bound'*, *symmetric*])
apply (*subst fold-heap-update-commute* [*OF disj*])
apply (*simp add*: *heap-update-collapse*)
done
qed

lemma *fold-heap-update-padding-collapse*:
fixes *p*::*'a*::*xmem-type ptr*
assumes *bound*: *unat* (*ptr-val p*) + *n* ∗ *size-of TYPE*(*'a*) ≤ *addr-card*
assumes *not-null*: *ptr-val p* ≠ *0*
assumes *lbs*: ⋀*i*. *i* < *n* ⟹ *length* (*bs i*) = *size-of TYPE*(*'a*)
assumes *lbs'*: ⋀*i*. *i* < *n* ⟹ *length* (*bs' i*) = *size-of TYPE*(*'a*)
shows *fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*v i*) (*bs i*)) [*0*..<*n*]
((*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*w i*) (*bs' i*)) [*0*..<*n*]) *h*) =
fold (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*v i*) (*bs i*)) [*0*..<*n*] *h*
using *bound lbs lbs'*
proof (*induct n arbitrary*: *h bs bs'*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain**

```
    bound: unat (ptr-val p) + Suc n * size-of TYPE('a) ≤ addr-card and
    lbs-n: ⋀i. i < n ⟹ length (bs i) = size-of TYPE('a) and
    lbs'-n: ⋀i. i < n ⟹ length (bs' i) = size-of TYPE('a) and
    lbs'-n': length (bs' n) = size-of TYPE('a) and
    lbs-n': length (bs n) = size-of TYPE('a)
    by auto
  hence bound': unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card by simp
  from ptr-add-disjoint-last [OF bound not-null]
  have disj: {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span (p +p int n) = {} .
  show ?case
    apply simp
    apply (subst (2)  Suc.hyps [OF bound' lbs-n lbs'-n, symmetric])
    apply (assumption)
    apply assumption
    apply (subst fold-heap-update-padding-commute [OF disj lbs-n lbs'-n'])
     apply (assumption)
    apply (simp add: heap-update-padding-collapse [OF lbs-n' lbs'-n'])
    done
qed

lemma fold-heap-update-padding-heap-update-collapse:
  fixes p::'a::xmem-type ptr
  assumes bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card
  assumes not-null: ptr-val p ≠ 0
  assumes lbs: ⋀i. i < n ⟹ length (bs i) = size-of TYPE('a)
  shows fold (λi. heap-update-padding (p +p int i) (v i) (bs i)) [0..<n]
         ((fold (λi. heap-update (p +p int i) (w i) ) [0..<n]) h) =
        fold (λi. heap-update-padding (p +p int i) (v i) (bs i)) [0..<n] h
  using bound lbs
proof (induct n arbitrary: h bs )
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems obtain
    bound: unat (ptr-val p) + Suc n * size-of TYPE('a) ≤ addr-card and
    lbs-n: ⋀i. i < n ⟹ length (bs i) = size-of TYPE('a) and
    lbs-n': length (bs n) = size-of TYPE('a)
    by auto
  hence bound': unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card by simp
  from ptr-add-disjoint-last [OF bound not-null]
  have disj: {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span (p +p int n) = {} .
  show ?case
    apply simp
    apply (subst (2)  Suc.hyps [OF bound' lbs-n , symmetric])
    apply (assumption)
    apply (subst fold-heap-update-padding-heap-update-commute [OF disj lbs-n ])
     apply (assumption)
    thm heap-update-padding-heap-update-collapse
```

apply (*simp add*: *heap-update-padding-heap-update-collapse* [*OF lbs-n'*])
done
qed

21.1 Open Types

named-theorems $\mathcal{T}$*-def* **and**
ptr-valid-definition **and** *fold-ptr-valid* **and** *ptr-valid* **and** *ptr-valid-u-recursion* **and**
addressable-ptr-valid-field **and** *plift-simps* **and** *stack-the-default-plift* **and**
plift-heap-update-padding-heap-update-conv **and**
plift-heap-update-list-stack-byte

lemma *list-ex-array-fields*[*simp*]:
list-ex P (*array-fields m*) $\longleftrightarrow$ ($\exists$ *n*<*m*. *P* [*replicate n CHR ''1''*])
by (*simp add*: *array-fields-def list-ex-iff Bex-def*)

lemma *list-all-field-lookup-array-fields*[*simp*]:
CARD('*n*) = *m* $\Longrightarrow$
list-all
(λ*f*. $\exists$ *a b*. *field-lookup* (*typ-uinfo-t TYPE*('*a*::*c-type*['*n*::*finite*])) *f 0* = *Some* (*a*, *b*))
(*array-fields m*)
by (*simp add*: *array-fields-def*)

lemma *list-ex-subst-def*:
c $\equiv$ *xs* $\Longrightarrow$ *list-ex P c* = *list-ex P xs*
by *simp*

lemma *ex-field-lookup-append*:
wf-desc t $\Longrightarrow$
field-lookup t p1 n1 = *Some* (*u*, *m1*) $\Longrightarrow$
field-lookup u p2 n2 = *Some* (*v*, *m2*) $\Longrightarrow$
$\exists$ *m*. *field-lookup t* (*p1* @ *p2*) *n3* = *Some* (*v*, *m*)
by (*metis field-lookup-offset-shift' field-lookup-prefix-Some'*)

named-theorems *addressable-field-exec*

locale *open-types* =
fixes $\mathcal{T}$:: (*typ-uinfo* $*$ *qualified-field-name list*) *list* — toplevel addressable fields of a type
assumes *mem-type-u*: $\bigwedge$*t fs*. (*t*, *fs*) $\in$ *set* $\mathcal{T}$ $\Longrightarrow$ *mem-type-u t*
assumes *wf-*$\mathcal{T}'$:
list-all (λ(*t*, *fs*).
[] $\notin$ *set fs* $\wedge$
list-all (λ*f*. *field-lookup t f 0* $\neq$ *None*) *fs* $\wedge$
distinct-prop disj-fn fs) $\mathcal{T}$
assumes *distinct-*$\mathcal{T}$: *distinct* (*map fst* $\mathcal{T}$)
begin

inductive *addressable-field* :: *typ-uinfo* ⇒ *qualified-field-name* ⇒ *typ-uinfo* ⇒ *bool*
where
  addressable-field-refl[*intro*!, *simp*]: ⋀*t*. *addressable-field t* [] *t*
| *addressable-field-step*: ⋀*t fs p p′ u v n*.
    map-of 𝒯 *t* = *Some fs* ⟹ *p* ∈ *set fs* ⟹ *field-lookup t p 0* = *Some* (*u*, *n*) ⟹
    addressable-field u p′ v ⟹ *addressable-field t* (*p* @ *p′*) *v*

inductive *ptr-valid-u* :: *typ-uinfo* ⇒ *heap-typ-desc* ⇒ *addr* ⇒ *bool* **for** *t d* **where**
  ⋀*r p a*. *root-ptr-valid-u r d a* ⟹ *addressable-field r p t* ⟹
    ptr-valid-u t d (*a* + *of-nat* (*field-offset-untyped r p*))

definition *ptr-valid* :: *heap-typ-desc* ⇒ *'a*::*mem-type ptr* ⇒ *bool* **where**
  [*ptr-valid-definition*]:
    ptr-valid d p ≡ *ptr-valid-u* (*typ-uinfo-t TYPE*(*'a*)) *d* (*ptr-val p*)

lemma *addressable-field-single*:
  map-of 𝒯 *t* = *Some fs* ⟹ *p* ∈ *set fs* ⟹ *field-lookup t p 0* = *Some* (*v*, *n*) ⟹
    addressable-field t p v
  using *addressable-field-step*[*of t fs p v n* []] **by** *auto*

lemma *addressable-field-append*:
  assumes ∗: *addressable-field t p1 u addressable-field u p2 v*
  shows *addressable-field t* (*p1* @ *p2*) *v*
proof (*use* ∗ **in** ‹*induction arbitrary*: *p2 v*›)
  case ∗: (*addressable-field-step t fs p p′ u v n w p1*)

  show *?case*
    unfolding *append-assoc*
    apply (*rule addressable-field-step*[*OF* ∗(*1*,*2*,*3*)])
    apply (*rule* ∗)+
    done
qed *simp*

lemma *addressable-field-rev-induct*[*consumes 1*, *case-names refl step*]:
  assumes ∗: *addressable-field t p u*
  assumes *1*: ⋀*t*. *P t* [] *t*
  assumes *2*: ⋀*t p′ u fs f v n*.
    addressable-field t p′ u ⟹ *map-of* 𝒯 *u* = *Some fs* ⟹ *f* ∈ *set fs* ⟹
      field-lookup u f 0 = *Some* (*v*, *n*) ⟹ *P t p′ u* ⟹ *P t* (*p′* @ *f*) *v*
  shows *P t p u*
proof −
  have *addressable-field t p1 u* ⟹ *P t p1 u* ⟹ *P t* (*p1* @ *p2*) *v*
    if ∗: *addressable-field u p2 v* **for** *t p1 u p2 v*
  proof (*use* ∗ **in** ‹*induction arbitrary*: *t p1*›)
    case ∗: (*addressable-field-step t fs f q u v n w*)
    show *?case*
      unfolding *append-assoc*[*symmetric*]
      by (*intro* ∗.*IH 2*[*OF* ∗.*prems*(*1*) ∗(*1*,*2*,*3*)] ∗)

```
      (intro addressable-field-append[OF *.prems(1)]
            addressable-field-single[OF *(1,2,3)])
  qed simp
  from this[OF * addressable-field-refl 1] show ?thesis by simp
qed

lemma addressable-field-rev-cases[consumes 1, case-names refl step]:
  assumes *: addressable-field t p u
  assumes 1: t = u ⟹ p = [] ⟹ P
  assumes 2: ⋀p' fs f v n.
    addressable-field t p' v ⟹ map-of 𝒯 v = Some fs ⟹ f ∈ set fs ⟹
      field-lookup v f 0 = Some (u, n) ⟹ p = p' @ f ⟹ P
  shows P
  using *
  by (induction t==t p==p u==u rule: addressable-field-rev-induct)
    (use 1 2 in auto)

lemma wf-𝒯: map-of 𝒯 t = Some fs ⟹ list-all (λf. field-lookup t f 0 ≠ None)
fs
  using wf-𝒯' by (auto simp: list-all-iff dest!: map-of-SomeD)

lemma 𝒯-not-nil: map-of 𝒯 t = Some fs ⟹ [] ∉ set fs
  using wf-𝒯' by (auto simp: list-all-iff dest!: map-of-SomeD)

lemma addressable-field-refl-iff[simp]:
  addressable-field t [] u ⟷ t = u
 by (subst addressable-field.simps) (auto simp add: 𝒯-not-nil cong: HOL.conj-cong)

lemma addressable-field-domain:
  addressable-field t p u ⟹ t ∈ fst ` set 𝒯 ∨ (t = u ∧ p = [])
  apply (subst (asm) addressable-field.simps)
  by (auto simp: image-iff split-paired-Ball dest!: map-of-SomeD) blast+

lemma addressable-fieldD-mem-type-u:
  addressable-field t p u ⟹ mem-type-u t ∨ (t = u ∧ p = [])
  by (auto dest!: addressable-field-domain mem-type-u)

lemma addressable-fieldD-field-lookup:
  assumes p: addressable-field t p u shows ∃n. field-lookup t p 0 = Some (u, n)
  by (use p in induction)
   (auto intro: ex-field-lookup-append mem-type-u mem-type-u.wf-desc dest: map-of-SomeD)

lemma addressable-field-same-iff[simp]:
  addressable-field t p t ⟷ p = []
  using addressable-fieldD-field-lookup[of t p t] field-lookup-same-type-empty(1)[of
t p 0 t]
  by auto

lemma addressable-fieldD-field-lookup':
```

addressable-field t p u $\Longrightarrow$ *field-lookup t p 0 = Some* (*u*, *field-offset-untyped t p*)
using *addressable-fieldD-field-lookup*[*of t p u*] **by** (*auto simp add*: *field-offset-untyped-def*)

lemma *addressable-fieldD-field-ti*:
assumes *p*: *addressable-field* (*typ-uinfo-t TYPE*('*a*::*c-type*)) *p t*
shows $\exists$ *u*. *field-ti TYPE*('*a*) *p* = *Some u* $\wedge$ *export-uinfo u* = *t*
using *field-lookup-uinfo-Some-rev*[*OF addressable-fieldD-field-lookup'*[*OF p*]]
by (*clarsimp simp*: *field-ti-def simp del*: *field-lookup-offset-untyped-eq*)

lemma *addressable-fieldD-field-names*:
addressable-field t p u $\Longrightarrow$ *p* $\in$ *set* (*field-names-u t u*)
by (*auto dest*: *addressable-fieldD-field-lookup intro*: *field-lookup-Some-field-names-u*(*1*))

lemma *mem-type-u-addressable-field*:
assumes $*$: *addressable-field t fs u*
shows *mem-type-u u* $\Longrightarrow$ *mem-type-u t*
by (*use* $*$ **in** *induction*) (*auto intro*: *mem-type-u dest*!: *map-of-SomeD*)

lemma *wf-desc-addressable-field*:
assumes $*$: *addressable-field t fs u*
shows *wf-desc t* $\Longrightarrow$ *wf-desc u*
using *addressable-fieldD-field-lookup'*[*OF* $*$] **by** (*auto intro*: *field-lookup-wf-desc-pres*)

lemma *wf-size-desc-addressable-field*:
assumes $*$: *addressable-field t fs u*
shows *wf-size-desc t* $\Longrightarrow$ *wf-size-desc u*
using *addressable-fieldD-field-lookup'*[*OF* $*$] **by** (*auto intro*: *field-lookup-wf-size-desc-pres*)

lemma *field-offset-append-of-addressable-field*:
assumes *p*: *addressable-field t p u* **and** *q*: *addressable-field u q v*
shows *field-offset-untyped t* (*p* @ *q*) = *field-offset-untyped t p* + *field-offset-untyped u q*
proof −
have *p* = [] $\wedge$ *q* = [] $\vee$ *mem-type-u t*
using *addressable-fieldD-mem-type-u*[*OF p*] *addressable-fieldD-mem-type-u*[*OF q*] **by** *auto*
then show *?thesis*
proof (*elim disjE conjE*)
assume *mem-type-u t* **then show** *?thesis*
by (*intro field-names-u-field-offset-untyped-append*[*OF mem-type-u.wf-desc addressable-fieldD-field-names*[*OF p*] *addressable-fieldD-field-names*[*OF q*]])
qed *simp*
qed

lemma *intvl-subset-of-addressable-field*:
assumes *p*: *addressable-field t p u*
shows {*a* + *of-nat* (*field-offset-untyped t p*) ..+ *size-td u*} $\subseteq$ {*a* ..+ *size-td t*}
using *addressable-fieldD-field-lookup*[*OF p*]

using *intvl-subset-of-field-lookup*[*of t p u field-offset-untyped t p a*]
by (*auto simp*: *field-offset-untyped-def*)

lemma *root-ptr-valid-u-ptr-valid-u*:
assumes *a*: *root-ptr-valid-u t d a* **shows** *ptr-valid-u t d a*
using *ptr-valid-u.intros*[*OF a addressable-field-refl*] **by** *simp*

lemma *root-ptr-valid-ptr-valid*: *root-ptr-valid d p* $\Longrightarrow$ *ptr-valid d p*
unfolding *root-ptr-valid-root-ptr-valid-u-conv ptr-valid-def*
by (*rule root-ptr-valid-u-ptr-valid-u*)

lemma *fold-ptr-valid'*[*fold-ptr-valid*]:
ptr-valid-u (*typ-uinfo-t TYPE*('*a*::*mem-type*)) *d x* = *ptr-valid d* (*PTR*('*a*) *x*)
by (*simp add*: *ptr-valid-def*)

lemma *ptr-valid-u-trans*:
assumes ∗: *ptr-valid-u t d a*
shows *addressable-field t p u* $\Longrightarrow$ *ptr-valid-u u d* (*a* + *of-nat* (*field-offset-untyped t p*))
proof (*use* ∗ **in** *induction*)
case (*1 r q a*)
have *eq*:
a + *word-of-nat* (*field-offset-untyped r q*) + *word-of-nat* (*field-offset-untyped t p*) =
a + *word-of-nat* (*field-offset-untyped r* (*q* @ *p*))
using *field-offset-append-of-addressable-field*[*OF 1*(*2*,*3*)] **by** *simp*
show *?case* **unfolding** *eq*
by (*intro ptr-valid-u.intros 1 addressable-field-append*[*OF 1*(*2*,*3*)])
qed

lemma *ptr-valid-u-step*:
assumes ∗: *map-of* $\mathcal{T}$ *t* = *Some F f* ∈ *set F field-lookup t f 0* = *Some* (*u*, *n*)
and *a*: *ptr-valid-u t d a*
shows *ptr-valid-u u d* (*a* + *of-nat* (*field-offset-untyped t f*))
by (*rule ptr-valid-u-trans*[*OF a*]) (*rule addressable-field-single*[*OF* ∗])

lemma *ptr-valid-u-recursion'*[*ptr-valid-u-recursion*]:
ptr-valid-u t d a ⟷
root-ptr-valid-u t d a ∨
list-ex (λ(*u*, *fs*). *list-ex* (λ*f*. ∃ *a' n*. *field-lookup u f 0* = *Some* (*t*, *n*) ∧
ptr-valid-u u d a' ∧ *a* = *a'* + *of-nat* (*field-offset-untyped u f*)) *fs*) $\mathcal{T}$
proof −
have
root-ptr-valid-u t d
(*a* + *word-of-nat* (*field-offset-untyped r p*)) ∨
(∃ *x*∈*set* $\mathcal{T}$. *case x of* (*u*, *fs*) ⇒ ∃ *f*∈*set fs*.
(∃ *n*. *field-lookup u f 0* = *Some* (*t*, *n*)) ∧
(∃ *a'*. *ptr-valid-u u d a'* ∧
a + *word-of-nat* (*field-offset-untyped r p*) =

```
        a' + word-of-nat (field-offset-untyped u f)))
    if p: addressable-field r p t and r: root-ptr-valid-u r d a
    for a r p
  proof (use p in ‹cases rule: addressable-field-rev-cases›)
    case refl with r show ?thesis by simp
  next
    case ∗: (step p' fs p'' u m)
    then have ∗∗: (u, fs) ∈ set T
      by (auto simp: map-of-SomeD)
    have mem-type-u u
      using mem-type-u[of u fs] ∗(2) by (auto dest: map-of-SomeD)
    with ∗ have mem-r: mem-type-u r
      by (auto dest: mem-type-u-addressable-field)

    have u-t: addressable-field u p'' t
      using ∗(2,3,4) by (rule addressable-field-single)

    show ?thesis
      apply (intro disjI2 bexI[of - (u, fs)] ∗∗)
      apply simp
      apply (intro conjI bexI[of - p''] ∗ exI[of - a + of-nat (field-offset-untyped r
p')])
      subgoal using ∗(4) by (rule exI)
      subgoal using r ∗(1) by (rule ptr-valid-u.intros)
     apply (simp add: field-offset-append-of-addressable-field[OF ∗(1) u-t] ‹p = p'
@ p''›)
      done
  qed
  moreover have
    ptr-valid-u t d
      (a' + word-of-nat (field-offset-untyped r p) + word-of-nat (field-offset-untyped
u f))
    if 1: (u, fs) ∈ set T and 2: f ∈ set fs field-lookup u f 0 = Some (t, n)
      and r: root-ptr-valid-u r d a' addressable-field r p u
    for u fs f n r p a'
  proof −
    have 1: map-of T u = Some fs
      using 1 by (simp add:  map-of-eq-Some-iff[OF distinct-T])
    show ?thesis
      using ptr-valid-u.intros[OF r] addressable-field-single[OF 1 2]
      by (rule ptr-valid-u-trans)
  qed
  ultimately show ?thesis
    by (auto simp: list-ex-iff del: disjCI
          elim!: ptr-valid-u.cases intro: root-ptr-valid-u-ptr-valid-u)
qed

lemma T-distinct: map-of T t = Some fs ⟹ distinct-prop disj-fn fs
  using wf-T' by (auto simp: list-all-iff dest!: map-of-SomeD)
```

lemma *addressable-field-unique*:
assumes *t*: *addressable-field r p t* **shows** *addressable-field r p t′* $\Longrightarrow$ *t* = *t′*
proof (*use t* **in** *induction*)
case *1*: (*addressable-field-step t fs p p′ u v n*)
from ‹*addressable-field t* (*p* @ *p′*) *t′*› **show** *?case*
proof *cases*
case *2*: (*addressable-field-step fs′ q q′ w m*)
with *1* **have** *map-of* $\mathcal{T}$ *t* = *Some fs p* ∈ *set fs q* ∈ *set fs* **by** *auto*
with *1* $\mathcal{T}$*-distinct*[*of t fs*] **have** *p* = *q* ∨ *disj-fn p q*
using *pairwise-set-of-distinct-prop*[*of disj-fn*, *OF disj-fn-comm*, *of fs*]
by (*auto simp*: *pairwise-def*)
with ‹*p* @ *p′* = *q* @ *q′*› **have** [*simp*]: *q* = *p* ∧ *q′* = *p′*
by (*auto simp*: *disj-fn-def less-eq-list-def prefix-def append-eq-append-conv2*)
with *1*(*3*) *2*(*4*) **have** *u* = *w*
by *simp*
with *1*(*5*) ‹*addressable-field w q′ t′*› **show** *?thesis*
by *simp*
qed (*use 1* **in** *simp*)
qed *simp*

lemma *addressable-fields-split*:
assumes *r*: *addressable-field r p t addressable-field r* (*p* @ *p′*) *u*
shows *addressable-field t p′ u*
proof (*use r* **in** *induction*)
case *1*: (*addressable-field-step t fs p0 p1 u v n*)
from *1*(*6*, *1*−*5*) **show** *?case*
proof *cases*
case *2*: (*addressable-field-step fs′ q q′ t′ m*)
with *1* **have** *map-of* $\mathcal{T}$ *t* = *Some fs p0* ∈ *set fs q* ∈ *set fs* **by** *auto*
with *1* $\mathcal{T}$*-distinct*[*of t fs*] **have** *p0* = *q* ∨ *disj-fn p0 q*
using *pairwise-set-of-distinct-prop*[*of disj-fn*, *OF disj-fn-comm*, *of fs*]
by (*auto simp*: *pairwise-def*)
with *2*(*1*) **have** *p0* = *q* ∧ *p1* @ *p′* = *q′*
by (*auto simp*: *disj-fn-def less-eq-list-def prefix-def append-eq-append-conv2*)
with *1 2* **show** *?thesis*
by *simp*
qed *simp*
qed *simp*

lemma *addressable-field-antisymm*:
addressable-field t p u $\Longrightarrow$ *addressable-field u q t* $\Longrightarrow$ *u* = *t*
using *addressable-field-append*[*of t p u q t*] **by** *simp*

lemma *addressable-field-sub-typ*: *addressable-field t path u* $\Longrightarrow$ *u* ≤ *t*
using *addressable-fieldD-field-lookup*[*of t path u*] *typ-tag-le-def*
by (*blast dest*: *td-set-field-lookupD*)

lemma *sub-typ-of-field-lookup*:

field-lookup (*typ-uinfo-t TYPE*('*a*)) *p 0* = *Some* (*typ-uinfo-t TYPE*('*b*), *n*) ⟹
TYPE('*b*::*mem-type*) ≤$_\tau$ *TYPE*('*a*::*mem-type*)
apply (*clarsimp simp add*: *sub-typ-def*)
apply (*rule field-of-sub*[*of of-nat n*])
using *field-lookup-offset-size*'[*of typ-uinfo-t TYPE*('*a*) *p*]
mem-type-class.u.max-size[**where** '*a*='*a*]
by (*auto simp add*: *field-of-def field-offset-def field-offset-untyped-def addr-card-len-of-conv*
intro!: *td-set-field-lookupD*[*of* - *p*] *unat-of-nat-eq*[*symmetric*])

lemma *addressable-fields-imp-sub-typ*:
addressable-field (*typ-uinfo-t TYPE*('*a*)) *p* (*typ-uinfo-t TYPE*('*b*)) ⟹
TYPE('*b*::*mem-type*) ≤$_\tau$ *TYPE*('*a*::*mem-type*)
by (*simp add*: *addressable-field-sub-typ sub-typ-def*)

lemma *ptr-valid-u-cases*:
assumes *t*: *mem-type-u t*
assumes *t-a*: *ptr-valid-u t d a*
assumes *u-b*: *ptr-valid-u u d b*
shows *disjnt* {*a*..+*size-td t*} {*b*..+*size-td u*} ∨
(*t* = *u* ∧ *a* = *b*) ∨
(∃ *p*. *addressable-field t p u* ∧ *t* ≠ *u* ∧ *b* = *a* + *word-of-nat* (*field-offset-untyped t p*)) ∨
(∃ *p*. *addressable-field u p t* ∧ *t* ≠ *u* ∧ *a* = *b* + *word-of-nat* (*field-offset-untyped u p*))
(**is** *?disj* ∨ *?addr*)
proof (*rule disjCI2*)
assume *a-b*: ¬ *?disj*
from *t-a* **obtain** *r f0 x* **where** *x0*: *root-ptr-valid-u r d x*
and *a*: *a* = *x* + *of-nat* (*field-offset-untyped r f0*)
and *r-t*: *addressable-field r f0 t*
by (*auto elim*!: *ptr-valid-u.cases*)
from *u-b* **obtain** *r*' *f1 x*' **where** *x1*: *root-ptr-valid-u r*' *d x*'
and *b*: *b* = *x*' + *of-nat* (*field-offset-untyped r*' *f1*)
and *r-u*: *addressable-field r*' *f1 u*
by (*auto elim*!: *ptr-valid-u.cases*)
from *a-b* **have** ¬ *disjnt* {*x* ..+ *size-td r*} {*x*' ..+ *size-td r*'}
using *intvl-subset-of-addressable-field*[*OF r-t*, *of x*]
using *intvl-subset-of-addressable-field*[*OF r-u*, *of x*']
by (*auto intro*: *disjnt-subset1 disjnt-subset2 simp*: *a b*)
with *root-ptr-valid-u-cases*[*OF x0 x1*] **have** [*simp*]: *r*' = *r x*' = *x*
by (*auto simp*: *disjnt-def*)
have *r*: *mem-type-u r*
using *r-t t* **by** (*rule mem-type-u-addressable-field*)
then have *wf-r*: *wf-desc r*
by (*rule mem-type-u.wf-desc*)
have ¬ *disjnt*
{(*word-of-nat* (*field-offset-untyped r f0*)::*addr*)..+*size-td t*}
{*word-of-nat* (*field-offset-untyped r f1*)..+*size-td u*}
using *a-b* **by** (*simp add*: *a b*)

```
  with field-lookup-non-prefix-disj'[OF r
      addressable-fieldD-field-lookup'[OF r-t]
      addressable-fieldD-field-lookup'[of r f1 u]] r-u
  obtain f' where f0 = f1 @ f' ∨ f1 = f0 @ f'
    by (auto simp: disj-fn-def less-eq-list-def prefix-def)
  then show ?addr
  proof (elim disjE)
    assume f0 = f1 @ f'
    with r-t r-u addressable-fields-split[of r f1 u f' t]
    show ?thesis
      by (cases u = t) (auto simp add: a b field-offset-append-of-addressable-field)
  next
    assume f1 = f0 @ f'
    with r-t r-u addressable-fields-split[of r f0 t f' u]
    show ?thesis
      by (cases u = t) (auto simp add: a b field-offset-append-of-addressable-field)
  qed
qed

lemma ptr-valid-cases':
  fixes p :: 'a::mem-type ptr
  fixes q :: 'b::mem-type ptr
  assumes p: ptr-valid d p
  assumes q: ptr-valid d q
  shows disjnt (ptr-span p) (ptr-span q) ∨
    (typ-uinfo-t TYPE('a) = typ-uinfo-t TYPE('b) ∧ ptr-val p = ptr-val q) ∨
    (∃f. addressable-field (typ-uinfo-t TYPE('a)) f (typ-uinfo-t TYPE('b)) ∧
      typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) ∧
      q = PTR('b) &(p → f)) ∨
    (∃f. addressable-field (typ-uinfo-t TYPE('b)) f (typ-uinfo-t TYPE('a)) ∧
      typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) ∧
      p = PTR('a) &(q → f))
  using ptr-valid-u-cases[OF - p[unfolded ptr-valid-def] q[unfolded ptr-valid-def]]
  by (simp add: size-of-def ptr-val-conv field-lvalue-def field-offset-def)

lemma ptr-valid-u-cases-weak:
  assumes mem-type-u t
  assumes ptr-valid-u t d a
  assumes ptr-valid-u u d b
  shows disjnt {a..+size-td t} {b..+size-td u} ∨
    (∃p. addressable-field t p u ∧ b = a + word-of-nat (field-offset-untyped t p)) ∨
    (∃p. addressable-field u p t ∧ a = b + word-of-nat (field-offset-untyped u p))
  using ptr-valid-u-cases[OF assms] by (cases t = u) auto

lemma ptr-valid-cases-weak:
  fixes p :: 'a::mem-type ptr
  fixes q :: 'b::mem-type ptr
  assumes p: ptr-valid d p
  assumes q: ptr-valid d q
```

shows *disjnt* (*ptr-span p*) (*ptr-span q*) ∨
(∃*f*. *addressable-field* (*typ-uinfo-t TYPE*('*a*)) *f* (*typ-uinfo-t TYPE*('*b*)) ∧
q = *PTR*('*b*) &(*p* → *f*)) ∨
(∃*f*. *addressable-field* (*typ-uinfo-t TYPE*('*b*)) *f* (*typ-uinfo-t TYPE*('*a*)) ∧
p = *PTR*('*a*) &(*q* → *f*))
using *ptr-valid-cases'*[*OF assms*]
by (*cases typ-uinfo-t TYPE*('*a*) = *typ-uinfo-t TYPE*('*b*))
(*auto simp*: *ptr-val-conv field-lvalue-def*)

lemma *ptr-valid-u-cvalid-u*:
assumes *t*: *mem-type-u t*
assumes *ptr-valid*: *ptr-valid-u t d a*
shows *cvalid-u t d a*
using *ptr-valid*
proof
fix *r fs a'* **assume** *r-t*: *addressable-field r fs t*
and *a-eq*: *a* = *a'* + *word-of-nat* (*field-offset-untyped r fs*)
and *a'*: *root-ptr-valid-u r d a'*
have *r*: *mem-type-u r*
using *mem-type-u-addressable-field*[*OF r-t t*] **.**
have *align-t*: *2* ^ *align-td t dvd size-td t*
using *t* **by** (*rule mem-type-u.align-size*)

show *cvalid-u t d a* **unfolding** *a-eq*
using *r align-t addressable-fieldD-field-lookup'*[*OF r-t*] *root-ptr-valid-u-cvalid-u*[*OF a'*]
by (*rule mem-type-u.cvalid-u-field-lookup*)
qed

lemma *ptr-valid-h-t-valid*: *ptr-valid d p* ⟹ *d* ⊨$_t$ (*p*::'*a*::*mem-type ptr*)
unfolding *ptr-valid-def cvalid-cvalid-u-conv*
by (*rule ptr-valid-u-cvalid-u*) *simp-all*

sublocale *ptr-valid?*: *wf-ptr-valid ptr-valid*
by *unfold-locales* (*rule ptr-valid-h-t-valid*)

lemma *h-val-heap-update'* [*plift-simps*]:
fixes *p*::'*a*::*mem-type ptr*
fixes *q*::'*a*::*mem-type ptr*
assumes *ptr-valid-p*: *ptr-valid d p*
assumes *ptr-valid-q*: *ptr-valid d q*
shows *h-val* (*heap-update p v h*) *q* = ((*h-val h*)(*p*:=*v*)) *q*
by (*metis c-type-class.lift-def lift-heap-update ptr-valid-h-t-valid ptr-valid-p ptr-valid-q*)

lemma *h-val-heap-update-padding'* [*plift-simps*]:
fixes *p*::'*a*::*mem-type ptr*
fixes *q*::'*a*::*mem-type ptr*
assumes *ptr-valid-p*: *ptr-valid d p*

assumes *ptr-valid-q*: *ptr-valid d q*
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *h-val* (*heap-update-padding p v bs h*) *q* = ((*h-val h*)(*p*:=*v*)) *q*
by (*smt* (*verit*, *best*) *fun-upd-apply-eq-cases h-val-heap-update-padding*
h-val-heap-update-padding-disjoint lbs local.ptr-valid-h-t-valid ptr-valid-p ptr-valid-q
ptr-valid-same-type-cases)

lemmas *ptr-valid-plift-Some-hval* [*plift-simps*]

theorem *disjoint-type-plift*:
fixes *p*::'*a*::*mem-type ptr*
assumes *unrelated1*: ¬ *TYPE*('*a*) ≤$_\tau$ *TYPE*('*b*::*mem-type*)
assumes *unrelated2*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *ptr-valid-p*: *ptr-valid* (*hrs-htd h*) *p*
shows *plift* ((*hrs-mem-update* (*heap-update p v*) *h*)) = (*plift h*::'*b ptr* ⇒ '*b option*)
proof −
from *unrelated1 unrelated2* **have** *disj*: *typ-uinfo-t TYPE*('*a*) ⊥$_t$ *typ-uinfo-t TYPE*('*b*)
using *tag-disj-def* **unfolding** *sub-typ-def* **by** *blast*
from *ptr-valid-p* **have** *hrs-htd h* ⊨$_t$ *p*
by (*simp add*: *ptr-valid-h-t-valid*)

from *clift-heap-update-same* [*OF this disj*]
have *clift* (*hrs-mem-update* (*heap-update p v*) *h*) = (*clift h*::'*b ptr* ⇒ '*b option*)
by *simp*
then show *?thesis*
apply −
apply (*rule ext*)
by (*metis hrs-htd-mem-update plift-None plift-clift-conv*)
qed

theorem *disjoint-type-update-h-val*:
fixes *p*::'*a*::*mem-type ptr*
fixes *q*::'*b*::*mem-type ptr*
assumes *unrelated1*: ¬ *TYPE*('*a*) ≤$_\tau$ *TYPE*('*b*)
assumes *unrelated2*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *typed-p*: *hrs-htd h* ⊨$_t$ *p*
assumes *typed-q*: *hrs-htd h* ⊨$_t$ *q*
shows *h-val* (*hrs-mem* (*hrs-mem-update* (*heap-update p v*) *h*)) *q* = *h-val* (*hrs-mem h*) *q*
proof −
from *unrelated1 unrelated2* **have** *disj*: *typ-uinfo-t TYPE*('*a*) ⊥$_t$ *typ-uinfo-t TYPE*('*b*)
using *tag-disj-def* **unfolding** *sub-typ-def* **by** *blast*
from *clift-heap-update-same* [*OF typed-p disj*]
have *clift* (*hrs-mem-update* (*heap-update p v*) *h*) *q* = *clift h q*
by *simp*
with *typed-p typed-q*
show *?thesis*
by (*metis clift-Some-eq-valid h-val-clift hrs-mem-def prod.collapse*)
qed

lemma *intvl-Suc-0*: $\{x\ ..+\ Suc\ 0\} = \{x\}$
by (*auto simp*: *intvl-def*)

lemma *ptr-span-subset-of-addressable-field*:
fixes *q*:: *'a*::*mem-type ptr*
shows *addressable-field t p* (*typ-uinfo-t TYPE*('*a*)) $\Longrightarrow$
ptr-val q = *x* + *word-of-nat* (*field-offset-untyped t p*) $\Longrightarrow$
ptr-span q $\subseteq$ $\{x..+size\text{-}td\ t\}$
using *intvl-subset-of-addressable-field*[*of t p typ-uinfo-t TYPE*('*a*) *x*]
by (*simp flip*: *size-of-tag*)

lemma *addressable-field-wf-descD*:
assumes *path*: *addressable-field t path u*
shows *t* = *u* $\vee$ (*wf-desc t* $\wedge$ *wf-desc u*)
using *addressable-fieldD-mem-type-u*[*OF path*] *wf-desc-addressable-field*[*OF path*]
by (*blast intro*: *mem-type-u.wf-desc*)

lemma *addressable-field-wf-size-descD*:
assumes *path*: *addressable-field t path u*
shows *t* = *u* $\vee$ (*wf-size-desc t* $\wedge$ *wf-size-desc u*)
using *addressable-fieldD-mem-type-u*[*OF path*] *wf-size-desc-addressable-field*[*OF path*]
by (*blast intro*: *mem-type-u.wf-size-desc*)

lemma *root-ptr-valid-u-ptr-valid-u-cases*:
fixes *p q* :: *addr*
assumes *root-valid-p*: *root-ptr-valid-u t d p*
assumes *valid-q*: *ptr-valid-u s d q*
shows $\{p\ ..+\ size\text{-}td\ t\} \cap \{q\ ..+\ size\text{-}td\ s\} = \{\} \vee$
($\exists$ *path*. *addressable-field t path s* $\wedge$ *q* = *p* + *of-nat* (*field-offset-untyped t path*))
proof (*cases* $\{p\ ..+\ size\text{-}td\ t\} \cap \{q\ ..+\ size\text{-}td\ s\} = \{\}$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *valid-q*
obtain *root path x* **where**
path: *addressable-field root path s* **and**
root-ptr-valid-u: *root-ptr-valid-u root d x* **and**
q: *q* = *x* + *word-of-nat* (*field-offset-untyped root path*)
by (*clarsimp simp add*: *ptr-valid-def elim*!: *ptr-valid-u.cases*)

then have *contained*: $\{q\ ..+\ size\text{-}td\ s\} \subseteq \{x..+size\text{-}td\ root\}$
using *intvl-subset-of-addressable-field*[*OF path*] **by** (*simp add*: *size-of-def*)

from *root-valid-p* **have** *root-ptr-valid-u-p*: *root-ptr-valid-u t d p*
by (*simp add*: *root-ptr-valid-root-ptr-valid-u-conv*)
from *root-ptr-valid-u-cases* [*OF root-ptr-valid-u-p root-ptr-valid-u*] *False contained*

obtain $t = root\ p = x$
by (*metis* (*no-types*, *lifting*) *disj-subset preorder-class.dual-order.refl*)
with *path q False* **show** *?thesis*
by *auto*
qed

lemma *root-ptr-valid-u-ptr-valid-cases*:
fixes p :: *addr* **and** q :: *'b*::*mem-type ptr*
assumes *root-valid-p*: *root-ptr-valid-u t d p*
assumes *valid-q*: *ptr-valid d q*
shows $\{p\ ..+\ size\text{-}td\ t\} \cap ptr\text{-}span\ q = \{\} \vee$
$(\exists\, path.\ addressable\text{-}field\ t\ path\ (typ\text{-}uinfo\text{-}t\ TYPE('b)) \wedge$
$ptr\text{-}val\ q = p + of\text{-}nat\ (field\text{-}offset\text{-}untyped\ t\ path))$
using *root-ptr-valid-u-ptr-valid-u-cases*[*OF root-valid-p valid-q*[*unfolded ptr-valid-def*]]
by (*simp add*: *size-of-def*)

lemma *root-ptr-valid-ptr-valid-cases*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *root-valid-p*: *root-ptr-valid d p*
assumes *valid-q*: *ptr-valid d q*
shows *ptr-span p* $\cap$ *ptr-span q* $= \{\} \vee$
$(\exists\, path.\ addressable\text{-}field\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ path\ (typ\text{-}uinfo\text{-}t\ TYPE('b))$
$\wedge$
$q = PTR\ ('b)\ \&(p \rightarrow path))$
using *root-ptr-valid-u-ptr-valid-cases*[*OF*
root-valid-p[*unfolded root-ptr-valid-root-ptr-valid-u-conv*] *valid-q*]
by (*simp add*: *size-of-def field-lvalue-def ptr-val-conv field-offset-def*)

lemma *stack-allocs-ptr-valid-cases1*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *alloc*: $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ d$
assumes *q-not-stack-byte*: $typ\text{-}uinfo\text{-}t\ TYPE('b) \neq typ\text{-}uinfo\text{-}t\ TYPE(stack\text{-}byte)$
assumes *valid-q*: *ptr-valid d' q*
shows $((\exists\, i\ path.\ i < n \wedge$
$addressable\text{-}field\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ path\ (typ\text{-}uinfo\text{-}t\ TYPE('b)) \wedge$
$q = PTR\ ('b)\ \&((p +_p int\ i) \rightarrow path)) \wedge \neg\ ptr\text{-}valid\ d\ q) \vee$
$(ptr\text{-}valid\ d\ q \wedge \{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\})$
proof −
from *valid-q*
obtain *root path x* **where**
path: *addressable-field root path* (*typ-uinfo-t TYPE*(*'b*)) **and**
root-ptr-valid-u: *root-ptr-valid-u root d' x* **and**
q: *ptr-val q* $= x +$ *word-of-nat* (*field-offset-untyped root path*)
by (*clarsimp simp add*: *ptr-valid-def elim*!: *ptr-valid-u.cases*)
from *path q-not-stack-byte* **have** *root-not-stack-byte*: $root \neq typ\text{-}uinfo\text{-}t\ TYPE(stack\text{-}byte)$
using *addressable-field-sub-typ sub-typ-stack-byte-u* **by** *blast*

```
 from stack-allocs-root-ptr-valid-u-cases [OF alloc root-not-stack-byte] root-ptr-valid-u
 consider i where i<n x = ptr-val (p +_p int i) root = typ-uinfo-t TYPE('a)
   | root-ptr-valid-u root d x by auto
 then show ?thesis
 proof cases
   case 1

   have ptr-span q ⊆ ptr-span (p +_p int i)
    using ptr-span-subset-of-addressable-field[OF path q] by (simp add: size-of-def
1)
   also have ... ⊆ {ptr-val p..+n * size-of TYPE('a)}
     using alloc 1(1) by (rule stack-allocs-contained)
   finally have ptr-val q ∈ {ptr-val p..+n * size-of TYPE('a)}
     using mem-type-self by auto
   then have root-q: root-ptr-valid d (PTR-COERCE('b → stack-byte) q)
    using stack-allocs-cases[OF alloc] by (auto simp add: q 1 simp flip: Ptr-ptr-coerce)
   have False if q': ptr-valid d q
     using root-ptr-valid-ptr-valid-cases[OF root-q q'] q-not-stack-byte
    by (auto simp add: intvl-Suc-0 dest!: addressable-field-sub-typ sub-typ-stack-byte-u)
   then have ¬ ptr-valid d q by auto
   with 1 path show ?thesis
     by (auto simp: ptr-val-conv q field-lvalue-def field-offset-def
           intro!: exI[of - i] exI[of - path])
 next
   case 2
   then have ptr-valid d q
     by (auto simp add: ptr-valid-def q intro!: ptr-valid-u.intros path)
   with stack-allocs-disjoint [OF alloc q-not-stack-byte ptr-valid-h-t-valid, of q]
   show ?thesis by simp
 qed
qed

lemma stack-allocs-ptr-valid-cases1'[consumes 3, case-names addressable-field dis-
joint]:
  fixes p:: 'a::mem-type ptr
  fixes q:: 'b::mem-type  ptr
  assumes alloc: (p, d') ∈ stack-allocs n S TYPE('a) d
  assumes q-not-stack-byte: typ-uinfo-t TYPE('b) ≠ typ-uinfo-t TYPE(stack-byte)
  assumes valid-q: ptr-valid d' q
  assumes addressable-field: ⋀i path. i < n ⟹
          addressable-field (typ-uinfo-t TYPE('a)) path (typ-uinfo-t TYPE('b)) ⟹
           q = PTR ('b) &((p +_p int i)→path) ⟹ ¬ ptr-valid d q ⟹  P
   assumes disjoint: ptr-valid d q ⟹ {ptr-val p ..+ n * size-of TYPE('a)} ∩
ptr-span q = {} ⟹ P
  shows P
   using stack-allocs-ptr-valid-cases1 [OF alloc q-not-stack-byte valid-q] address-
able-field disjoint by blast

lemma stack-allocs-ptr-valid-cases2:
```

fixes p:: $'a$::*mem-type ptr*
fixes q:: $'b$::*mem-type ptr*
assumes *alloc*: $(p,\ d') \in$ *stack-allocs* $n\ \mathcal{S}$ *TYPE*$('a)\ d$
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*$('b) \neq$ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *valid-q*: *ptr-valid d q*
shows *ptr-valid d' q*
proof −
from *valid-q*
obtain *root path x* **where**
path: *addressable-field root path* (*typ-uinfo-t TYPE*$('b)$) **and**
root-ptr-valid-u: *root-ptr-valid-u root d x* **and**
x: *ptr-val q* = *x* + *word-of-nat* (*field-offset-untyped root path*)
by (*clarsimp simp add*: *ptr-valid-def ptr-valid-u.simps*)

from *path q-not-stack-byte* **have** *root-not-stack-byte*: *root* ≠ *typ-uinfo-t TYPE*(*stack-byte*)
using *addressable-field-sub-typ sub-typ-stack-byte-u* **by** *blast*

from *stack-allocs-root-ptr-valid-u-other* [*OF alloc root-ptr-valid-u root-not-stack-byte*]
have *root-ptr-valid-u root d' x* **.**
with *path x*
show *ptr-valid d' q*
using *ptr-valid-def ptr-valid-u.simps* **by** *blast*
qed

lemma *stack-allocs-ptr-valid-cases3*:
fixes p:: $'a$::*mem-type ptr*
fixes q:: $'b$::*mem-type ptr*
assumes *alloc*: $(p,\ d') \in$ *stack-allocs* $n\ \mathcal{S}$ *TYPE*$('a)\ d$
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*$('b) \neq$ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *i*: *i* < *n*
assumes *path*: *addressable-field* (*typ-uinfo-t TYPE*$('a)$) *path* (*typ-uinfo-t TYPE*$('b)$)
assumes *q*: *q* = *PTR* $('b)$ &((*p* +$_p$ *int i*)→*path*)
shows *ptr-valid d' q*
apply (*simp add*: *q ptr-valid-def field-lvalue-def field-offset-def*)
apply (*intro ptr-valid-u.intros*[*OF* - *path*])
apply (*subst stack-allocs-root-ptr-valid-u-cases*[*OF alloc stack-allocs-no-stack-byte*[*OF alloc*]])
apply (*auto intro*: *i*)
done

theorem *stack-allocs-ptr-valid-cases*:
fixes p:: $'a$::*mem-type ptr*
fixes q:: $'b$::*mem-type ptr*
assumes *alloc*: $(p,\ d') \in$ *stack-allocs* $n\ \mathcal{S}$ *TYPE*$('a)\ d$
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*$('b) \neq$ *typ-uinfo-t TYPE*(*stack-byte*)
shows *ptr-valid d' q* ⟷
(((∃ *i path*. *i* < *n* ∧ *addressable-field* (*typ-uinfo-t TYPE*$('a)$) *path* (*typ-uinfo-t TYPE*$('b)$) ∧
q = *PTR* $('b)$ &((*p* +$_p$ *int i*)→*path*)) ∧ ¬ *ptr-valid d q*) ∨

```
    (ptr-valid d q ∧ {ptr-val p ..+ n * size-of TYPE('a)} ∩ ptr-span q = {}))
  using stack-allocs-ptr-valid-cases1 [OF alloc q-not-stack-byte]
    stack-allocs-ptr-valid-cases2 [OF alloc q-not-stack-byte]
    stack-allocs-ptr-valid-cases3 [OF alloc q-not-stack-byte]
  by blast

lemma stack-releases-ptr-valid-cases1:
  fixes p:: 'a::mem-type ptr
  fixes q:: 'b::mem-type  ptr
  assumes p-not-stack-byte: typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE(stack-byte)
  assumes q-not-stack-byte: typ-uinfo-t TYPE('b) ≠ typ-uinfo-t TYPE(stack-byte)
  assumes valid-q: ptr-valid (stack-releases n p d) q
  shows {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ∩ ptr-span q = {} ∧
        ptr-valid d q
proof −
  from valid-q
  obtain root path x where
    path: addressable-field root path (typ-uinfo-t TYPE('b)) and
    root-ptr-valid-u: root-ptr-valid-u root (stack-releases n p d) x and
    x: ptr-val q = x + word-of-nat (field-offset-untyped root path)
    by (clarsimp simp add: ptr-valid-def ptr-valid-u.simps)
  from path q-not-stack-byte have root-not-stack-byte: root ≠ typ-uinfo-t TYPE(stack-byte)
    using addressable-field-sub-typ sub-typ-stack-byte-u by blast

  from stack-releases-root-ptr-valid-u-cases [OF p-not-stack-byte root-not-stack-byte,
of n p d x ] root-ptr-valid-u
  obtain root-cases:
     {ptr-val p..+n * size-of TYPE('a)} ∩ {x..+size-td root} = {}
     root-ptr-valid-u root d x by auto
  moreover
  have ptr-span q ⊆ {x..+size-td root}
    using path x by (rule ptr-span-subset-of-addressable-field)
  moreover
  from root-cases(2) path x have ptr-valid d q
    using ptr-valid-def ptr-valid-u.simps by blast
  ultimately
  show ?thesis by blast
qed

lemma stack-releases-ptr-valid-cases2:
  fixes p:: 'a::mem-type ptr
  fixes q:: 'b::mem-type  ptr
  assumes p-not-stack-byte: typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE(stack-byte)
  assumes q-not-stack-byte: typ-uinfo-t TYPE('b) ≠ typ-uinfo-t TYPE(stack-byte)
  assumes root-p: ⋀i. i < n ⟹ root-ptr-valid d (p +p int i)
  assumes disj: {ptr-val p ..+ n * size-of TYPE('a::mem-type)} ∩ ptr-span q =
{}
  assumes valid-q: ptr-valid d q
  shows ptr-valid (stack-releases n p d) q
```

proof −
from *valid-q*
obtain *root path x* **where**
path: *addressable-field root path* (*typ-uinfo-t TYPE*('*b*)) **and**
root-ptr-valid-u: *root-ptr-valid-u root d x* **and**
x: *ptr-val q* = *x* + *word-of-nat* (*field-offset-untyped root path*)
by (*clarsimp simp add*: *ptr-valid-def ptr-valid-u.simps*)

from *path q-not-stack-byte* **have** *root-not-stack-byte*: *root* ≠ *typ-uinfo-t TYPE*(*stack-byte*)
using *addressable-field-sub-typ sub-typ-stack-byte-u* **by** *blast*

have *contained*: *ptr-span q* ⊆ {*x*..+*size-td root*}
using *path x* **by** (*rule ptr-span-subset-of-addressable-field*)

have {*ptr-val p*..+*n* ∗ *size-of TYPE*('*a*)} ∩ {*x*..+*size-td root*} = {}
proof (*rule ccontr*)
assume {*ptr-val p*..+*n* ∗ *size-of TYPE*('*a*)} ∩ {*x*..+*size-td root*} ≠ {}
then obtain *i* **where** *i*: *i* < *n* **and** *overlap*: *ptr-span* (*p* +$_p$ *int i*) ∩ {*x*..+*size-td root*} ≠ {}
by (*meson array-to-index-span disjoint-iff*)
from *root-p* [*OF i*] **have** *root-ptr-valid-u* (*typ-uinfo-t TYPE*('*a*)) *d* (*ptr-val* (*p* +$_p$ *int i*))
by (*simp add*: *root-ptr-valid-root-ptr-valid-u-conv*)
from *root-ptr-valid-u-cases* [*OF root-ptr-valid-u this*] *overlap*
obtain *root* = *typ-uinfo-t TYPE*('*a*) *x* = *ptr-val* (*p* +$_p$ *int i*)
by (*simp add*: *inf-commute size-of-tag*)
then have *ptr-span* (*p* +$_p$ *int i*) = {*x*..+*size-td root*}
by (*simp add*: *size-of-def*)
with *contained* **have** *ptr-span q* ⊆ *ptr-span* (*p* +$_p$ *int i*) **by** *simp*
with *disj i* **show** *False*
apply (*simp add*: *array-index-span-conv*)
by (*smt* (*verit*, *ccfv-SIG*) *UN-I disjoint-iff mem-type-self ord-class.lessThan-iff subsetD*)
qed

with *stack-releases-root-ptr-valid-u-cases* [*OF p-not-stack-byte root-not-stack-byte*] *root-ptr-valid-u*

have *root-ptr-valid-u root* (*stack-releases n p d*) *x* **by** *auto*
with *path x*
show *ptr-valid* (*stack-releases n p d*) *q*
using *ptr-valid-def ptr-valid-u.simps* **by** *blast*
qed

lemma *stack-releases-ptr-valid-cases3*:
fixes *p*:: '*a*::*mem-type ptr*
fixes *q*:: '*b*::*mem-type ptr*
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)

assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* ≠ *typ-uinfo-t TYPE(stack-byte)*
assumes *root-p*: ⋀*i*. *i* < *n* ⟹ *root-ptr-valid d* (*p* +$_p$ *int i*)
assumes *valid-q*: *ptr-valid d q*
assumes *invalid-q*: ¬ *ptr-valid* (*stack-releases n p d*) *q*
shows ∃ *path i*.
addressable-field (*typ-uinfo-t TYPE('a)*) *path* (*typ-uinfo-t TYPE('b)*) ∧ *i* < *n*
∧
q = *PTR('b)* &(*p* +$_p$ *int i*→*path*)
proof −
from *valid-q invalid-q stack-releases-ptr-valid-cases2* [*OF p-not-stack-byte q-not-stack-byte root-p - valid-q, of n*]
have {*ptr-val p..+n* ∗ *size-of TYPE('a)*} ∩ *ptr-span q* ≠ {} **by** *auto*
then obtain *i* **where** *i-bound*: *i* < *n* **and** *overlap*: *ptr-span* (*p* +$_p$ *int i*) ∩ *ptr-span q* ≠ {}
by (*auto simp add*: *array-index-span-conv*)

from *root-ptr-valid-ptr-valid-cases* [*OF root-p* [*OF i-bound*] *valid-q, simplified overlap, simplified*] *i-bound*
show *?thesis* **by** *blast*
qed

lemma *stack-releases-ptr-valid-cases*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE('a)* ≠ *typ-uinfo-t TYPE(stack-byte)*
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* ≠ *typ-uinfo-t TYPE(stack-byte)*
assumes *root-p*: ⋀*i*. *i* < *n* ⟹ *root-ptr-valid d* (*p* +$_p$ *int i*)
shows *ptr-valid* (*stack-releases n p d*) *q* ⟷
ptr-valid d q ∧ {*ptr-val p ..+ n* ∗ *size-of TYPE('a*::*mem-type)*} ∩ *ptr-span q* = {}
using *stack-releases-ptr-valid-cases1* [*OF p-not-stack-byte q-not-stack-byte*]
stack-releases-ptr-valid-cases2 [*OF p-not-stack-byte q-not-stack-byte root-p*] **by** *blast*

lemma *ptr-valid-same-type-neq-no-overlap-conv*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'a*::*mem-type ptr*
assumes *valid-p*: *ptr-valid d p*
assumes *valid-q*: *ptr-valid d q*
shows *p* ≠ *q* ⟷ *ptr-span p* ∩ *ptr-span q* = {}
using *cvalid-same-type-neq-disjoint assms ptr-valid-h-t-valid*
by *blast*

theorem *stack-allocs-the-default-plift* [*stack-the-default-plift*]:
fixes *p*:: *'a*::*xmem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE('a)* (*hrs-htd h*)
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* ≠ *typ-uinfo-t TYPE(stack-byte)*
shows *the-default c-type-class.zero*

```
        (plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..<n]) (hrs-htd-update (λ-. d') h)) q) =
        the-default c-type-class.zero (plift h q)
proof (cases ptr-valid d' q)
  case True
  from alloc q-not-stack-byte True
  show ?thesis
  proof (cases rule: stack-allocs-ptr-valid-cases1')
    case (addressable-field i path)
    from addressable-field obtain
      i: i < n and
       path: addressable-field (typ-uinfo-t TYPE('a)) path (typ-uinfo-t TYPE('b))
and
      q: q = PTR('b) &(p +p int i→path) and
      invalid: ¬ ptr-valid (hrs-htd h) q
      by blast
    from invalid have (plift h q) = None
      by (simp add: plift-None)
    moreover
    from stack-allocs-cases [OF alloc]
     obtain bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card and
not-null: ptr-val p ≠ 0
      by (metis Ptr-ptr-val c-guard-NULL-simp)

    from path obtain m where
      field-lookup (typ-uinfo-t TYPE('a)) path 0 = Some ((typ-uinfo-t TYPE('b)),
m)
      by (auto dest: addressable-fieldD-field-lookup)

   then obtain t where fl: field-lookup (typ-info-t TYPE('a)) path 0 = Some (t,
m) and
      match: export-uinfo t = typ-uinfo-t TYPE('b)
      using field-lookup-uinfo-Some-rev by blast
    from fl have field-ti: field-ti TYPE('a) path  = Some t
      using field-lookup-field-ti by blast
  from field-lookup-zero [OF fl match] have from-bytes-zero: from-bytes (access-ti0
t c-type-class.zero) = (c-type-class.zero::'b) .
  note h-val-field =  h-val-field-from-bytes' [OF field-ti match [simplified typ-uinfo-t-def]]
    from h-val-fold  [OF bound not-null i]
    have h-val-zero:
      h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..<n]  (hrs-mem
h)) (p +p int i) = c-type-class.zero .
    have path-zero:
      h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..<n] (hrs-mem
h)) (PTR('b) &(p +p int i→path)) = c-type-class.zero
      by (simp add: h-val-field h-val-zero from-bytes-zero)

    from True i path q
  have (plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
```

```
[0..<n]) (hrs-htd-update (λ-. d') h)) q) = Some c-type-class.zero
      apply −
      apply (subst ptr-valid-plift-Some-hval)
       apply (simp add: hrs-htd-update)
      apply (simp add: path-zero hrs-mem-update)
      done
    ultimately
    show ?thesis by simp
  next
    case disjoint
    then obtain valid-q: ptr-valid (hrs-htd h) q and
        disj: {ptr-val p..+n * size-of TYPE('a)} ∩ ptr-span q = {} by blast

    from valid-q have (plift h q) = Some (h-val (hrs-mem h) q)
      by (simp add: ptr-valid-plift-Some-hval valid-q)
    moreover
    have h-val-eq:
      h-val (fold (λi. heap-update (p +p int i) c-type-class.zero) [0..<n] (hrs-mem
h)) q =
        h-val (hrs-mem h) q by (simp add: h-val-fold-zero-disjoint [OF disj])
    from True have valid-q':
        ptr-valid (hrs-htd ((hrs-mem-update (fold (λi. heap-update (p +p int i)
c-type-class.zero) [0..<n])
                    (hrs-htd-update (λ-. d') h)))) q
      by (simp add: hrs-htd-update)
   have plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..<n])
           (hrs-htd-update (λ-. d') h)) q = Some (h-val (hrs-mem h) q)
    using True by (simp add: ptr-valid-plift-Some-hval [OF valid-q'] hrs-mem-update
h-val-eq)
    ultimately
    show ?thesis
      by simp
  qed
next
  case False
  from  False
  have invalid: ¬ ptr-valid (hrs-htd h) q
    using alloc q-not-stack-byte stack-allocs-ptr-valid-cases2 by blast
  then
  have plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..<n]) (hrs-htd-update (λ-. d') h)) q = None
    using plift-None
    by (metis (no-types, lifting) False hrs-htd-mem-update hrs-htd-update)
  moreover
  have (plift h q) = None
    using invalid by (simp add: plift-None)
  ultimately show ?thesis by simp
qed
```

theorem *stack-releases-the-default-plift* [*stack-the-default-plift*]:
fixes *p*:: *'a*::*xmem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *roots*: $\bigwedge i.\ i < n \Longrightarrow$ *root-ptr-valid* (*hrs-htd h*) (*p* +$_p$ *int i*)
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *the-default c-type-class.zero*
(*plift*
(*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*)
[*0*..<*n*]) *h*)
q) =
the-default c-type-class.zero (*plift* (*hrs-htd-update* (*stack-releases n p*) *h*) *q*)
proof (*cases ptr-valid* (*stack-releases n p* (*hrs-htd h*)) *q*)
case *True*
from *stack-releases-ptr-valid-cases1* [*OF p-not-stack-byte q-not-stack-byte True*]
obtain *disjoint*: {*ptr-val p*..+*n* ∗ *size-of TYPE*(*'a*)} ∩ *ptr-span q* = {} **and**
valid-q: *ptr-valid* (*hrs-htd h*) *q* **by** *blast*

from *True valid-q* **have** *plift* (*hrs-htd-update* (*stack-releases n p*) *h*) *q* = *Some* (*h-val* (*hrs-mem h*) *q*)
using *ptr-valid-plift-Some-hval*
by (*metis hrs-htd-update hrs-mem-htd-update*)

moreover
from *h-val-fold-zero-disjoint* [*OF disjoint*]
have *h-val* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*] (*hrs-mem h*)) *q* = *h-val* (*hrs-mem h*) *q* **.**
with *valid-q*
have (*plift* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*]) *h*) *q*) = *Some* (*h-val* (*hrs-mem h*) *q*)
by (*simp add*: *hrs-mem-update ptr-valid-plift-Some-hval*)
ultimately show *?thesis* **by** *simp*
next
case *False*
from *False*
have *eq1*: *plift* (*hrs-htd-update* (*stack-releases n p*) *h*) *q* = *None*
by (*simp add*: *plift-None hrs-htd-update*)
show *?thesis*
proof (*cases ptr-valid* (*hrs-htd h*) *q*)
case *True*
show *?thesis*
proof (*cases n*)
case *0*
then show *?thesis*
using *False True* **by** (*auto simp add*: *stack-releases-def*)
next
case (*Suc n'*)

from *roots* [*of 0*] *Suc*
have *not-null*: *ptr-val p* ≠ *0*
by (*metis CTypesDefs.ptr-add-def Ptr-ptr-val Zero-not-Suc add.right-neutral bot-nat-0.not-eq-extremum mem-type-self mult-zero-left of-int-0 root-ptr-valid-range-not-NULL roots semiring-1-class.of-nat-0*)

from *roots Suc*
have *bound*: *unat* (*ptr-val p*) + *n* ∗ *size-of TYPE*('*a*) ≤ *addr-card*
by (*metis array-to-index-span len-of-addr-card root-ptr-valid-range-not-NULL zero-not-in-intvl-no-overflow*)

from *stack-releases-ptr-valid-cases3* [*OF p-not-stack-byte q-not-stack-byte roots True False*]
obtain *path i* **where**
path: *addressable-field* (*typ-uinfo-t TYPE*('*a*)) *path* (*typ-uinfo-t TYPE*('*b*)) **and**
q: *q* = *PTR*('*b*) &(*p* +$_p$ *int i*→*path*) **and**
i-bound: *i* < *n*
by *blast*
from *path* **obtain** *m* **where**
field-lookup (*typ-uinfo-t TYPE*('*a*)) *path 0* = *Some* ((*typ-uinfo-t TYPE*('*b*)), *m*)
by (*auto dest*: *addressable-fieldD-field-lookup*)

then obtain *t* **where** *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *path 0* = *Some* (*t*, *m*) **and**
match: *export-uinfo t* = *typ-uinfo-t TYPE*('*b*)
using *field-lookup-uinfo-Some-rev* **by** *blast*
from *fl* **have** *field-ti*: *field-ti TYPE*('*a*) *path* = *Some t*
using *field-lookup-field-ti* **by** *blast*
from *field-lookup-zero* [*OF fl match*] **have** *from-bytes-zero*: *from-bytes* (*access-ti*$_0$ *t c-type-class.zero*) = (*c-type-class.zero*::'*b*) **.**
note *h-val-field* = *h-val-field-from-bytes*′ [*OF field-ti match* [*simplified typ-uinfo-t-def*]]
from *h-val-fold* [*OF bound not-null i-bound*]
have *h-val-zero*:
h-val (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*] (*hrs-mem h*)) (*p* +$_p$ *int i*) = *c-type-class.zero* **.**
have *path-zero*:
h-val (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*] (*hrs-mem h*)) (*PTR*('*b*) &(*p* +$_p$ *int i*→*path*)) = *c-type-class.zero*
by (*simp add*: *h-val-field h-val-zero from-bytes-zero*)
from *True i-bound path q*
have *plift* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*]) *h*) *q* = *Some c-type-class.zero*
apply −
apply (*subst ptr-valid-plift-Some-hval*)
apply (*simp add*: *hrs-htd-update*)
apply (*simp add*: *path-zero hrs-mem-update*)
done

```
    then show ?thesis by (simp add: eq1)
   qed
  next
   case False
   from False
   have plift (hrs-mem-update (fold (λi. heap-update (p +p int i) c-type-class.zero)
[0..<n]) h) q = None
    using plift-None
    by (metis False hrs-htd-mem-update )
   with eq1 show ?thesis by simp
  qed
qed

lemma h-val-heap-update-padding-heap-update-conv:
  fixes p::'a::mem-type ptr
  fixes q::'b::mem-type ptr
  assumes valid-p: ptr-valid d p
  assumes valid-q: ptr-valid d q
  assumes lbs: length bs = size-of TYPE('a)
  shows h-val ((heap-update-padding p v bs h)) q = h-val (heap-update p v h) q
  using ptr-valid-cases-weak[OF valid-p valid-q]
proof (elim disjE exE conjE)
  assume disjnt (ptr-span p) (ptr-span q) then show ?thesis
   by (simp add: h-val-heap-update-padding-disjoint h-val-update-regions-disjoint
lbs disjnt-def)
next
 fix f assume f: addressable-field (typ-uinfo-t TYPE('a)) f (typ-uinfo-t TYPE('b))
   and q: q = PTR('b) &(p→f)
  from addressable-fieldD-field-ti[OF f]
  obtain t where t: field-ti TYPE('a) f = Some t export-uinfo t = typ-uinfo-t
TYPE('b)
   by auto

  let ?s = from-bytes o access-ti0 t :: 'a ⇒ 'b
  interpret f: nested-field' t f ?s update-ti t o to-bytes-p
   using t by unfold-locales simp-all

  show ?thesis unfolding q using lbs
   by (subst (1 2) f.h-val-field) (simp add: h-val-heap-update-padding)
next
 fix f assume f: addressable-field (typ-uinfo-t TYPE('b)) f (typ-uinfo-t TYPE('a))
   and p: p = PTR('a) &(q→f)
  from addressable-fieldD-field-ti[OF f]
  obtain t where t: field-ti TYPE('b) f = Some t export-uinfo t = typ-uinfo-t
TYPE('a)
   by auto

  let ?s = from-bytes o access-ti0 t :: 'b ⇒ 'a
  interpret f: nested-field' t f ?s update-ti t o to-bytes-p
```

using *t* **by** *unfold-locales simp-all*

note *q* = *ptr-valid-h-t-valid*[*OF valid-q*]
show *?thesis* **unfolding** *p*
apply (*subst f.h-val-field-lvalue-update-padding*[*OF q q lbs*])
apply (*subst f.h-val-field-lvalue-update*[*OF q q*])
apply *simp*
done
qed

lemma *plift-heap-update-padding-heap-update-conv*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*
assumes *valid-p*: *ptr-valid* (*hrs-htd h*) *p*
assumes *lbs*: *length bs* = *size-of TYPE*(*'a*)
shows *plift* (*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q* = *plift* (*hrs-mem-update* (*heap-update p v*) *h*) *q*
using *h-val-heap-update-padding-heap-update-conv*
by (*metis hrs-htd-mem-update hrs-mem-update lbs valid-p wf-ptr-valid.plift-None wf-ptr-valid.ptr-valid-plift-Some-hval wf-ptr-valid-axioms*)

theorem *plift-heap-update-padding-heap-update-pointless-conv* [*plift-heap-update-padding-heap-update-conv*]:
fixes *p*::*'a*::*mem-type ptr*
assumes *valid-p*: *ptr-valid* (*hrs-htd h*) *p*
assumes *lbs*: *length bs* = *size-of TYPE*(*'a*)
shows *plift* (*hrs-mem-update* (*heap-update-padding p v bs*) *h*) = *plift* (*hrs-mem-update* (*heap-update p v*) *h*)
apply (*rule ext*)
using *plift-heap-update-padding-heap-update-conv assms*
by *blast*

lemma *ptr-valid-stack-byte-disjoint*:
fixes *q*::*'a*::{*mem-type*, *stack-type*} *ptr*
fixes *p*::*stack-byte ptr*
assumes *root-ptr-valid h p*
assumes *ptr-valid h q*
shows *ptr-span p* ∩ *ptr-span q* = {}
using *assms local.ptr-valid-h-t-valid no-stack-byte root-ptr-valid-not-subtype-disjoint sub-typ-stack-byte*
by *blast*

lemma *h-val-heap-update-list-stack-byte*:
fixes *q*::*'a*::{*mem-type*, *stack-type*} *ptr*
assumes *valid-p*: *root-ptr-valid d* (*p*::*stack-byte ptr*)
assumes *valid-q*: *ptr-valid d q*
assumes *lbs*: *length bs* = *size-of TYPE*(*stack-byte*)
shows *h-val* (*heap-update-list* (*ptr-val p*) *bs h*) *q* = *h-val h q*
using *ptr-valid-stack-byte-disjoint* [*OF valid-p valid-q*] *lbs*
by (*simp add*: *h-val-def heap-list-update-disjoint-same*)

lemma *h-val-heap-update-list-stack-byte'*:
fixes *q*::*'a*::{*mem-type*, *stack-type*} *ptr*
fixes *p*::*stack-byte ptr*
assumes *valid-p*: $\bigwedge i.\ i < length\ bs \Longrightarrow root\text{-}ptr\text{-}valid\ d\ (p +_p int\ i)$
assumes *valid-q*: *ptr-valid d q*
shows *h-val* (*heap-update-list* (*ptr-val p*) *bs h*) *q* = *h-val h q*
using *valid-p*
proof (*induct bs arbitrary*: *p h*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons b bs*)
from *Cons.prems Cons.prems* [*of 0*] **obtain**
valid-bs: $\bigwedge i.\ i < Suc\ (length\ bs) \Longrightarrow root\text{-}ptr\text{-}valid\ d\ (p +_p int\ i)$ **and**
valid-p: *root-ptr-valid d p* **by** *auto*
from *valid-bs* **have** *valid-bs'*: $\bigwedge i.\ i < length\ bs \Longrightarrow root\text{-}ptr\text{-}valid\ d\ ((p +_p 1) +_p int\ i)$
apply (*clarsimp simp add*: *ptr-add-def*)
by (*metis Ex-less-Suc group-cancel.add1 less-trans-Suc of-nat-Suc*)

have *length* [*b*] = *size-of TYPE*(*stack-byte*) **by** *auto*
note *b-eq* = *h-val-heap-update-list-stack-byte* [*OF valid-p valid-q this*]
have *val-eq*: *ptr-val* (*p* +$_p$ *1*) = *ptr-val p* + *1*
by (*auto simp add*: *ptr-add-def*)

note *hyp* = *Cons.hyps* [*OF valid-bs'*, *simplified val-eq*]
show *?case*
apply (*simp add*: *hyp*)
using *b-eq*
apply (*auto simp add*: *fun-upd-def*)
done
qed

lemma *plift-heap-update-list-stack-byte*:
fixes *q*::*'a*::{*mem-type*, *stack-type*} *ptr*
fixes *p*::*stack-byte ptr*
assumes *valid-p*: $\bigwedge i.\ i < length\ bs \Longrightarrow root\text{-}ptr\text{-}valid\ (hrs\text{-}htd\ h)\ (p +_p int\ i)$
shows *plift* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) *h*) *q* = *plift h q*
using *h-val-heap-update-list-stack-byte'*
by (*metis hrs-htd-mem-update hrs-mem-update valid-p wf-ptr-valid.plift-None*
wf-ptr-valid.ptr-valid-plift-Some-hval wf-ptr-valid-axioms)

lemma *plift-heap-update-list-stack-byte-pointless*[*plift-heap-update-list-stack-byte*]:
fixes *p*::*stack-byte ptr*
assumes *valid-p*: $\bigwedge i.\ i < length\ bs \Longrightarrow root\text{-}ptr\text{-}valid\ (hrs\text{-}htd\ h)\ (p +_p int\ i)$
shows *plift* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) *h*) =
(*plift h*::*'a*::{*mem-type*, *stack-type*} *ptr* ⇒ *'a option*)

using *assms plift-heap-update-list-stack-byte* **by** *blast*

lemma *plift-eq-plift*:
hrs-htd h = hrs-htd h' ⟹
(⋀p::'a ptr. ptr-valid (hrs-htd h') p ⟹
h-val (hrs-mem h) p = h-val (hrs-mem h') p) ⟹
(plift h::'a::mem-type ptr ⇒ 'a option) = plift h'
apply *(rule ext)*
apply *(cases h; cases h')*
apply *(simp add: plift-def lift-t-if h-t-valid-def root-ptr-valid-def hrs-htd-def*
hrs-mem-def)
done

definition *addressable-fields :: 'a::mem-type itself ⇒ (field-name list × 'a xtyp-info) list* **where**
addressable-fields TYPE('a) = (case map-of 𝒯 (typ-uinfo-t TYPE('a)) of
None ⇒ []
| Some fs ⇒ map (λf. (f, the (field-ti TYPE('a) f))) fs)

abbreviation *merge-addressable-fields :: 'a::mem-type ⇒ 'a ⇒ 'a* **where**
merge-addressable-fields ≡ merge-ti-list (map snd (addressable-fields TYPE('a)))

definition *read-dedicated-heap :: heap-raw-state ⇒ 'a::mem-type typ-heap-g* **where**
read-dedicated-heap h p = merge-addressable-fields ZERO('a) (the-default ZERO('a) (plift h p))

definition *write-dedicated-heap :: 'a::mem-type ptr ⇒ 'a ⇒ heap-raw-state ⇒ heap-raw-state* **where**
write-dedicated-heap p v = hrs-mem-update (heap-upd p (λold. merge-addressable-fields old v))

lemma *mem-addressable-fields*:
assumes *F: map-of 𝒯 (typ-uinfo-t TYPE('a::mem-type)) = Some F*
shows *f ∈ set F ⟹ field-ti TYPE('a) f = Some u ⟹ (f, u) ∈ set (addressable-fields TYPE('a))*
using *wf-𝒯[of typ-uinfo-t TYPE('a)]*
by *(force simp add: addressable-fields-def F image-iff list-all-iff field-ti-def)*

lemma *mem-snd-addressable-fields*:
assumes *F: map-of 𝒯 (typ-uinfo-t TYPE('a::mem-type)) = Some F f ∈ set F*
field-ti TYPE('a) f = Some u
shows *u ∈ snd ` set (addressable-fields TYPE('a))*
using *mem-addressable-fields[OF assms]* **by** *(force simp: image-iff)*

lemma *list-all-field-ti-addressable-fields*:
list-all (λ(f, u). field-ti TYPE('a) f = Some u) (addressable-fields TYPE('a::mem-type))
apply *(clarsimp simp add: addressable-fields-def split: option.splits)*
subgoal premises *prems*
using *prems[THEN wf-𝒯]*

unfolding *list.pred-map*
by (*rule list.pred-mono-strong*)
(*auto simp*: *list-all-iff field-ti-def typ-uinfo-t-def*
dest!: *field-lookup-export-uinfo-Some-rev field-lookup-field-ti'*)
done

lemma *distinct-disj-fn-addressable-fields*:
distinct-prop disj-fn (*map fst* (*addressable-fields TYPE*('*a*::*mem-type*)))
using *wf-*$\mathcal{T}'$
by (*auto simp*: *list-all-iff addressable-fields-def comp-def dest*!: *map-of-SomeD split*: *option.splits*)

lemma *list-all-field-ti-snd-addressable-fields*:
list-all (λ*u*. ∃*f*. *field-ti TYPE*('*a*) *f* = *Some u*) (*map snd* (*addressable-fields TYPE*('*a*::*mem-type*)))
unfolding *list.pred-map*
using *list-all-field-ti-addressable-fields*
by (*rule list.pred-mono-strong*) *auto*

sublocale *merge-addressable-fields*: *is-scene merge-addressable-fields* :: '*a*::*xmem-type* ⇒ '*a* ⇒ '*a*
using *list-all-field-ti-snd-addressable-fields* **by** (*rule is-scene-merge-ti-list*)

lemma *ptr-valid-u-cases-same-type*:
assumes *t*: *mem-type-u t* **and** *a*: *ptr-valid-u t d a* **and** *b*: *ptr-valid-u t d b*
shows *a* = *b* ∨ *disjnt* {*a*..+*size-td t*} {*b*..+*size-td t*}
using *ptr-valid-u-cases-weak*[*OF t a b*] **by** *auto*

lemma *read-dedicated-heap-heap-update-list-stack-byte-pointless*[*simp*]:
fixes *p*::*stack-byte ptr*
assumes *valid-p*: ⋀*i*. *i* < *length bs* ⟹ *root-ptr-valid* (*hrs-htd h*) (*p* +$_p$ *int i*)
shows *read-dedicated-heap* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) *h*)
=
(*read-dedicated-heap h*::'*a*::{*mem-type*, *stack-type*} *ptr* ⇒ '*a*)
unfolding *read-dedicated-heap-def*
by (*simp add*: *plift-heap-update-list-stack-byte-pointless*[*OF valid-p*])

lemma *read-dedicated-heap-write-dedicated-heap-ne*[*simp*]:
fixes *p* :: '*a*::*xmem-type ptr* **and** *q* :: '*b*::*xmem-type ptr*
assumes *ne*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
assumes *p*: *ptr-valid* (*hrs-htd h*) *p*
shows *read-dedicated-heap* (*write-dedicated-heap p x h*) *q* = *read-dedicated-heap h q*
proof *cases*
let *?A* = *typ-uinfo-t TYPE*('*a*) **and** *?B* = *typ-uinfo-t TYPE*('*b*)
assume *q*: *ptr-valid* (*hrs-htd h*) *q*
from *ptr-valid-cases-weak*[*OF p q*]
show *?thesis*
proof (*elim disjE exE conjE*)

```
    assume disjnt (ptr-span p) (ptr-span q) with p q show ?thesis
      by (simp add: read-dedicated-heap-def write-dedicated-heap-def disjnt-def
                hrs-mem-update-heap-upd plift-disjoint-region)
  next
   fix f' assume f'-n: addressable-field ?A f' ?B and q-eq: q = PTR('b) &(p→f')

    obtain F f1 f2 u where f'-eq: f' = f1 @ f2
      and F: f1 ∈ set F map-of 𝒯 ?A = Some F
      and f-u: field-ti TYPE('a) f1 = Some u
      and addressable-field (export-uinfo u) f2 ?B
       using f'-n ne by (cases rule: addressable-field.cases) (auto simp: field-ti-def
dest!: field-lookup-uinfo-Some-rev)

    obtain v n where v: export-uinfo v = ?B
      and f-fs: field-lookup (typ-info-t TYPE('a)) (f1 @ f2) 0 = Some (v, n)
      using addressable-fieldD-field-lookup[OF f'-n] unfolding f'-eq
      by (auto dest: field-lookup-uinfo-Some-rev)
    obtain m where u-v: field-lookup u f2 0 = Some (v, m)
      using f-fs f-u
      apply (subst (asm) field-lookup-append-eq)
      apply (auto simp add: f-u field-ti-def split: option.splits
          dest: CTypes.field-lookup-offset2)
      done
    have p': hrs-htd h ⊨t p
      using p by (rule ptr-valid-h-t-valid)

    have u-mem: u ∈ snd ` set (addressable-fields TYPE('a))
      using F(2,1) f-u(1) by (rule mem-snd-addressable-fields)
    have f1u-mem: (f1, u) ∈ set (addressable-fields TYPE('a))
      using F(2,1) f-u(1) by (rule mem-addressable-fields)
    have sz: size-of TYPE('b) = size-td v
      using v unfolding export-uinfo-size[symmetric] by (simp add: size-of-def)

    have length bs = size-of TYPE('b) ⟹
      access-ti v (merge-addressable-fields y x) bs = access-ti v y bs for y bs
      by (intro access-ti-merge-ti-list[OF list-all-field-ti-addressable-fields
                  distinct-disj-fn-addressable-fields f1u-mem u-v])
        (simp add: sz)
    then show ?thesis using q unfolding f'-eq q-eq
    apply (simp add: read-dedicated-heap-def write-dedicated-heap-def ptr-valid-plift-Some-hval
                  hrs-mem-update)
      apply (simp add: heap-upd-def
                ptr-valid-heap-update-subtype-field-access-ti-indep[OF p' p' f-fs v])
      done
  next
    fix f' n assume f'-n: addressable-field ?B f' ?A and p-eq: p = PTR('a) &(q
→ f')
    from f'-n ne obtain F f0 f1 t0 n where ∗: map-of 𝒯 ?B = Some F f0 ∈ set
F f' = f0 @ f1
```

```
      and f0: field-lookup ?B f0 0 = Some (t0, n) and f1: addressable-field t0 f1
?A
      by cases auto
    from addressable-fieldD-field-ti[OF f'-n]
    obtain tA where f'-u: field-ti TYPE('b) f' = Some tA and tA: export-uinfo
tA = ?A
      by auto
    have [simp]: size-td tA = size-of TYPE('a)
      unfolding export-uinfo-size[symmetric] tA by (simp add: size-of-def)

    from f0[unfolded typ-uinfo-t-def, THEN field-lookup-export-uinfo-Some-rev]
     obtain t0' where f0-t0': field-ti TYPE('b) f0 = Some t0' and t0: t0 =
export-uinfo t0'
      by (auto simp: field-ti-def typ-uinfo-t-def
               split: option.splits)

    obtain tA' n where t0'-f1: field-lookup t0' f1 0 = Some (tA', n)
      using addressable-fieldD-field-lookup'[OF f1, unfolded t0,
          THEN field-lookup-export-uinfo-Some-rev]
      by auto

    with field-ti-append-field-lookup[OF f0-t0' t0'-f1] f'-u
    have f0'-f1: field-lookup t0' f1 0 = Some (tA, n)
      by (simp add: *)

    note q' = ptr-valid.ptr-valid-h-t-valid[OF q]

    have f0t'-mem: (f0, t0') ∈ set (addressable-fields TYPE('b))
      using *(1,2) f0-t0' by (rule mem-addressable-fields)

    from q f'-n merge-ti-list-update-ti[OF
        list-all-field-ti-addressable-fields f0t'-mem distinct-disj-fn-addressable-fields]
        merge-ti-list-update-ti[OF
          list-all-field-ti-addressable-fields f0t'-mem distinct-disj-fn-addressable-fields
f0'-f1]
    show ?thesis unfolding p-eq
     by (simp add: read-dedicated-heap-def write-dedicated-heap-def hrs-mem-update
heap-upd-def
                   ptr-valid-plift-Some-hval h-val-field-update[OF f'-u tA q' q']
                   update-ti-t-def)
  qed
qed (simp add: read-dedicated-heap-def write-dedicated-heap-def plift-None)

lemma read-dedicated-heap-write-dedicated-heap[simp]:
  ptr-valid (hrs-htd h) (p::'a::xmem-type ptr) ⟹
    read-dedicated-heap (write-dedicated-heap p x h) =
    upd-fun p (λold. merge-addressable-fields old x) (read-dedicated-heap h)
 by (simp add: fun-eq-iff the-plift-hval-fun-upd-eqI read-dedicated-heap-def[abs-def]
upd-fun-def
```

write-dedicated-heap-def hrs-mem-update-heap-upd fun-upd-def)

lemma *hrs-mem-update-heap-update*:
fixes *p* :: *'a*::*xmem-type ptr*
shows *hrs-mem-update* (*heap-update p x*) *h* =
fold (λ(*f*, *u*). *hrs-mem-update* (*heap-upd-list* (*size-td u*) &(*p*→*f*) (*access-ti u x*)))
(*addressable-fields TYPE*(*'a*))
(*write-dedicated-heap p x h*)
proof −
show *?thesis*
apply (*subst heap-update-eq-fold-subfields*[*OF list-all-field-ti-addressable-fields*,
where *y*=*h-val* (*hrs-mem h*) *p*])
unfolding *hrs-mem-update-comp'*[*symmetric*] *write-dedicated-heap-def hrs-mem-update-heap-upd*
apply (*subst fold-functor*[*of hrs-mem-update*])
apply (*simp add*: *hrs-mem-update-def*)
apply (*simp add*: *hrs-mem-update-def fun-eq-iff*)
apply (*simp add*: *split-beta'*)
done
qed

lemma *hrs-mem-update-heap-update'*:
fixes *p* :: *'a*::*xmem-type ptr*
shows *hrs-mem-update* (*heap-update p x*) =
fold (λ(*f*, *u*). *hrs-mem-update* (*heap-upd-list* (*size-td u*) &(*p*→*f*) (*access-ti u x*)))
(*addressable-fields TYPE*(*'a*)) ∘
write-dedicated-heap p x
apply (*rule ext*)
unfolding *comp-apply*
apply (*rule hrs-mem-update-heap-update*)
done

lemma *hrs-htd-write-dedicated-heap*[*simp*]:
hrs-htd (*write-dedicated-heap p x h*) = *hrs-htd h*
by (*simp add*: *write-dedicated-heap-def*)

lemma *partial-pointer-lense-merge-addressable-fields*:
fixes *r* :: *'h* ⇒ *'a*::*xmem-type ptr* ⇒ *'a*
assumes *lense r w*
shows
partial-pointer-lense (λ*x y*. *merge-addressable-fields y x*)
(λ*h p x* . *merge-addressable-fields x* (*r h p*)) (λ*p x*. *w* (*upd-fun p* (λ*old*. *merge-addressable-fields old x*)))
proof −
interpret *lense r w* **by** *fact*
show *?thesis*
apply *unfold-locales*
apply (*simp-all add*: *upd-fun.upd-same upd-same*)

```
    subgoal by (simp add: comp-def)
    subgoal for p q x y h
      by (simp add: upd-fun-commute disj-ptr-span-ptr-neq disjnt-def Int-commute)
    done
qed

lemma pointer-lense-of-partials-cover:
  fixes rs :: ('h ⇒ 'a::xmem-type ptr ⇒ 'a ⇒ 'a) list
  assumes g-u: lense g u
  assumes ∗:
    length ms = length rs length ms = length ws
    list-all (λ(m, r, w). partial-pointer-lense m r w) (zip ms (zip rs ws))
    and ∗∗:
    ⋀a b c. distinct-prop (λm1 m2. m1 a (m2 b c) = m2 b (m1 a c)) ms
    ⋀p a q b h. distinct-prop (λw1 w2. p = q ∨ disjnt (ptr-span p) (ptr-span q) ⟶
        w1 p a (w2 q b h) = w2 q b (w1 p a h)) ((λp x. u (upd-fun p (λold.
merge-addressable-fields old x))) # ws)
  assumes ms-cover: ⋀a b. fold (λm. m a) ms b = merge-addressable-fields a b
  assumes R: ⋀h p. R h p = fold (λr. r h p) rs (g h p)
  assumes W: ⋀p f h. W p f h =
    fold (λw. w p (f (R h p))) ws (u (upd-fun p (λold. merge-addressable-fields old
(f (R h p))))) h)
  shows pointer-lense R W
  apply (rule pointer-lense-of-partials[where
    ms=(λa b. merge-addressable-fields b a) # ms and
    rs = (λh p x. merge-addressable-fields x (g h p)) # rs and
    ws = (λp x. u (upd-fun p (λold. merge-addressable-fields old x)))#ws])
  subgoal by (simp add: ∗(1))
  subgoal by (simp add: ∗(2))
  subgoal
    apply (rule list-all-zip-zip-cons)
    apply (rule partial-pointer-lense-merge-addressable-fields[OF g-u])
    apply (rule ∗)
    done
  subgoal for a b c
  proof −
    have scenes-ms: list-all is-scene ms
      using ∗ by (simp add: list-all-length partial-pointer-lense-def)
    have dis-ms: distinct-prop disjnt-scene ms
      using ∗∗(1) by (simp add: disjnt-scene-def[abs-def] distinct-prop-distrib-all)
    have comm-ms: pairwise comm-scene (set ms)
      apply (rule pairwise-set-of-distinct-prop)
      apply (simp add: comm-scene-comm)
      apply (use dis-ms in ‹auto simp: distinct-prop-iff-nth›)
      done
    have ms-y: list-all (λm. disjnt-scene m y ∨ m = y) ms if y: y ∈ set ms for y
      using in-set-conv-decomp[THEN iffD1, OF y] dis-ms
      by (auto simp: distinct-prop-append list-all-iff disjnt-scene-comm)
```

```
    show ?thesis
      apply (simp add: **)
      apply (simp flip: ms-cover)
      apply (intro ballI)
      apply (rule fold-disjnt-scene[OF scenes-ms comm-ms ms-y])
      apply auto
      done
  qed
  subgoal by (rule **)
  subgoal by (simp add: ms-cover)
  subgoal for h p
  proof −
    interpret fold: partial-pointer-lense
      λa. fold (λm. m a) ms λh p. fold (λr. r h p) rs λp x. fold (λw. w p x) ws
      apply (rule partial-pointer-lense-fold[OF *])
      subgoal using **(1) by auto
      subgoal using **(2) by auto
      done
    show R h p = fold (λr. r h p) ((λh p x. merge-addressable-fields x (g h p)) #
rs) x
      using fold.r-m by (simp add: R flip: ms-cover)
  qed
  subgoal by (simp add: W)
  done

lemma cover-read-dedicated-heap[simp]:
  merge-addressable-fields (read-dedicated-heap h p) = merge-addressable-fields ZERO('a::xmem-type)
  by (simp add: fun-eq-iff read-dedicated-heap-def)

lemma read-dedicated-heap-fun-upd-cover-zero-eq-upd-fun[simp]:
  ((read-dedicated-heap h)(p := merge-addressable-fields ZERO('a::xmem-type) x))
q =
    upd-fun p (λold. merge-addressable-fields old x) (read-dedicated-heap h) q
  by (simp add: upd-fun-def fun-upd-def fun-eq-iff)

theorem stack-allocs-read-dedicated-heap [stack-the-default-plift]:
  fixes p:: 'a::xmem-type ptr
  assumes alloc: (p, d') ∈ stack-allocs n S TYPE('a) (hrs-htd h)
  assumes q-not-stack-byte: typ-uinfo-t TYPE('b::xmem-type) ≠ typ-uinfo-t TYPE(stack-byte)
  shows read-dedicated-heap (hrs-mem-update (fold (λi. heap-update (p +p int i)
c-type-class.zero) [0..<n]) (hrs-htd-update (λ-. d') h)) =
    (read-dedicated-heap h :: 'b ptr ⇒ -)
  apply (rule ext)
  unfolding read-dedicated-heap-def stack-allocs-the-default-plift[OF assms] ..

theorem stack-releases-read-dedicated-heap [stack-the-default-plift]:
  fixes p:: 'a::xmem-type ptr
  assumes roots: ⋀i. i < n ⟹ root-ptr-valid (hrs-htd h) (p +p int i)
  assumes p-not-stack-byte: typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE(stack-byte)
```

assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* ≠ *typ-uinfo-t TYPE(stack-byte)*
shows *read-dedicated-heap* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*]) *h*) =
(*read-dedicated-heap* (*hrs-htd-update* (*stack-releases n p*) *h*) :: *'b*::*xmem-type ptr* ⇒ -)
unfolding *read-dedicated-heap-def stack-releases-the-default-plift*[*OF assms*, *of n*, *simplified*] **..**

lemma *ptr-valid-addressable-field-field-lvalue*[*addressable-field-exec*]:
fixes *p*::*'a*::*mem-type ptr*
assumes *ptr-valid h p*
assumes *addressable-field* (*typ-uinfo-t* (*TYPE('a)*)) *fs* (*typ-uinfo-t* (*TYPE('b*::*mem-type)*))
shows *ptr-valid h* (*PTR('b)* (&(*p*→*fs*)))
using *assms*
by (*metis field-lvalue-def field-offset-def local.fold-ptr-valid'*
local.open-types-axioms local.ptr-valid-u-trans open-types.ptr-valid-def)

lemma *addressable-field-exec*[*addressable-field-exec*]:
shows
addressable-field t ps v =
(*case ps of*
[] => *t* = *v*
| - ⇒ (∃ *fs p*. *map-of* $\mathcal{T}$ *t* = *Some fs* ∧
List.find (λ*p*. ∃ *u n*. *field-lookup t p 0* = *Some* (*u*, *n*) ∧ *p* @ *drop* (*length p*) *ps* = *ps* ∧
addressable-field u (*drop* (*length p*) *ps*) *v*) *fs* = *Some p*))
proof (*cases ps*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*Cons f ps'*)
show *?thesis*
proof (*cases map-of* $\mathcal{T}$ *t*)
case *None*
then show *?thesis*
apply (*simp add*: *Cons*)
by (*metis list.simps*(*2*) *local.addressable-field.cases option.simps*(*3*))
next
case (*Some fs*)
show *?thesis*
proof (*cases* ∃ *p*. *List.find* (λ*p*. ∃ *u n*. *field-lookup t p 0* = *Some* (*u*, *n*) ∧
p @ *drop* (*length p*) *ps* = *ps* ∧
addressable-field u (*drop* (*length p*) *ps*) *v*) *fs* = *Some p*)
case *True*
then obtain *p* **where**
find: *List.find* (λ*p*. ∃ *u n*. *field-lookup t p 0* = *Some* (*u*, *n*) ∧
p @ *drop* (*length p*) *ps* = *ps* ∧
addressable-field u (*drop* (*length p*) *ps*) *v*) *fs* = *Some p*
by *auto*

```
      then obtain u n where
        p: p ∈ set fs and
        ps: p @ drop (length p) ps = ps and
        u: field-lookup t p 0 = Some (u, n) and
        addr: addressable-field u (drop (length p) ps) v
        using findSomeD
        by (smt (verit, best))
      from addressable-field-step [OF Some p u addr]
      have addr': addressable-field t (p @ drop (length p) ps) v .
      from addressable-fieldD-field-lookup [OF addr] obtain m
        where v: field-lookup u (drop (length p) ps) 0 = Some (v, m)
        by blast
      from addressable-fieldD-field-lookup [OF addr'] obtain k where
         field-lookup t (p @ drop (length p) ps) 0 = Some (v, k)
        by blast
      have ps-eq: (p @ drop (length p) (f # ps')) = ps
        using ps
        by (simp add: Cons ps)
      show ?thesis
        using find addr'
        by (simp add: Cons Some ps-eq)
    next
      case False
      with Some show ?thesis
        apply (simp add: Cons)
        apply (force simp add: find-None-iff elim: addressable-field.cases)
        done
    qed
  qed
qed

local-setup ‹
let
  fun unfold-ss ctxt = ctxt addsimps
    Named-Theorems.get ctxt @{named-theorems T-def} @
    @{thms
      list-ex-iff
      fun-upd-other
      field-lookup-typ-uinfo-t-Some
      fold-typ-uinfo-t
      exists-nat-numeral}
in
   Cached-Theory-Simproc.declare-init-thy-simpset @{named-theorems T-def} un-
fold-ss
end
›

simproc-setup T (‹map-of T (typ-uinfo-t TYPE('a::mem-type))› | ‹map fst T›
| ‹set T›) = ‹
```

```
  let
    exception Abort;
    fun cache-simproc augment ctxt =
    let val umm-ctxt = Cached-Theory-Simproc.put-time-warp-simpset' @{named-theorems
𝒯-def} ctxt
    in Cached-Theory-Simproc.gen-simproc (umm-ctxt, Cached-Theory-Simproc.recert,
Cached-Theory-Simproc.add-cache) augment end
  in
    fn phi => fn ctxt => fn ct =>
      let
        val check =
          Match-Cterm.switch [
           (@{cterm-morph-match map-of 𝒯 (typ-uinfo-t ?T)} phi) #> (fn {T, ...}
=>
                if UMM-Proofs.is-ctype (Thm.term-of T) then () else raise Abort)
          , fn - => ()];
        val - = check ct
        val - = Utils.verbose-msg 3 ctxt (fn - =>
              𝒯 invoked: ^ Syntax.string-of-term ctxt (Thm.term-of ct))
      in
        cache-simproc (K single) ctxt ct
      end handle Match => NONE | Abort => NONE
  end
›
```

end

lemma *fold-field-lvalue*:
x + *word-of-nat* (*field-offset-untyped* (*typ-uinfo-t TYPE*('*a*::*c-type*)) *f*) = &(*PTR*('*a*) x→*f*)
by (*simp add*: *field-lvalue-def field-offset-def*)

lemma *field-lvalue-same-root-ptr-conv*:
&(*p*::'*a*:: *c-type ptr*→*f*) = &(*q*::'*a*:: *c-type ptr*→*f*) ⟷ *p* = *q*
by (*auto simp add*: *field-lvalue-same-conv*)

lemma *ptr-exhaust-eq*: **fixes** *p*::'*a*::*c-type ptr* **shows** *PTR*('*a*) (*ptr-val p*) = *p*
by (*cases p*) *simp*

lemma *fold-exists-ptr*: (∃ *x*. *P* (*PTR*('*a*::*c-type*) *x*)) ⟷ (∃ *q*. *P q*)
by (*metis ptr.exhaust*)

21.1.1 Syntax *PTR-VALID*('*a*)

context *open-types*
begin

We want to provide syntax that makes the type of *ptr-valid* visible on the term level for interpretations of the *open-types*. This is a bit tricky as Isabelle does not (yet) provide local syntax and local print / parse translations. We

implement it by a combination of theory level syntax and translations and a local declarations for the interpretations.

end
syntax
-ptr-valid :: type ⇒ logic (‹(‹*indent=1 notation=‹mixfix PTR-VALID›› PTR'-VALID/(1 '(-'))*›)

ML ‹
structure Ptr-Valid-Translation =
struct

val show-ptr-valid = Attrib.setup-config-bool @{binding show-ptr-valid} (K true)

val ptr-validN = @{const-name open-types.ptr-valid} |> Long-Name.base-name
val local-ptr-validN = Long-Name.qualify Long-Name.localN ptr-validN

type options = {ptr-valid:term, ptr-valid-name: string} option — NONE means we are within the (abstract) locale, SOME provides the relevant parameters from the global interpretations

fun ptr-valid-term NONE ctxt = Syntax.read-term ctxt local-ptr-validN
| ptr-valid-term (SOME {ptr-valid, ...}) - = ptr-valid

fun instantiate-ptr-valid opt ctxt typ =
let
val ***Const-*** ‹*open-types.ptr-valid - for T*› *= ptr-valid-term opt ctxt*
in
Const ‹*open-types.ptr-valid typ for T*›
end

fun const-name check ctxt s =
let
val Const (c, -) = Proof-Context.read-const {proper = true, strict = false} ctxt s
val res = check (Proof-Context.consts-of ctxt, c)
handle TYPE (msg, -, -) => error (msg ^ Position.here ***here****);*
in
res
end

fun const-syntax (SOME {ptr-valid-name, ...}) ctxt =
const-name (fn (-, c) => Lexicon.mark-const c) ctxt ptr-valid-name
| const-syntax NONE ctxt =
Lexicon.mark-const local-ptr-validN

fun gen-ptr-valid-tr' ptr-valid ctxt T ts = if Config.get ctxt show-ptr-valid then
let
val ptr-valid' as ***Const-*** ‹*open-types.ptr-valid typ for -*› *=*
Syntax.read-term ctxt ptr-validN

```
    val [-, Type ‹ptr typ›] = T |> binder-types
   val head = Syntax.const syntax-const ‹-ptr-valid› $ Syntax-Phases.term-of-typ
ctxt typ
  in
    Term.betapplys (head, ts)
  end
  handle Bind => raise Match
  else raise Match

fun add-ptr-valid-print-translation (opt:options) thy =
  let
    val ctxt = Proof-Context.init-global thy
    val const-syntax = const-syntax opt ctxt
    val ptr-valid = instantiate-ptr-valid opt
  in
    thy |> Sign.typed-print-translation [(const-syntax, gen-ptr-valid-tr' ptr-valid)]
  end
end›

parse-translation ‹
 let
   fun ptr-valid-tr ctxt [t] =
     let
       val Const- ‹open-types.ptr-valid - for T › =
             Syntax.read-term ctxt Ptr-Valid-Translation.ptr-validN
       val typ = Syntax-Phases.decode-typ t
     in
       Const ‹open-types.ptr-valid typ› $ T
     end
     handle Bind => error (PTR-VALID: no instance of ptr-valid in context )
 in
   [(syntax-const ‹-ptr-valid›, ptr-valid-tr)]
 end›

setup ‹Ptr-Valid-Translation.add-ptr-valid-print-translation NONE›
```

This theory level setup actually provides the local syntax when working within a locale like *open-types*. The *print-translation* is triggered on constant (abbreviation) *local.ptr-valid*

context *open-types*
begin
term *PTR-VALID*(*32 word*) — Works in both directions by theory level setup above.
declaration ‹*fn phi => fn context =>*

```
  let
    val ptr-valid = Morphism.term phi term ‹ptr-valid›
   val ptr-valid-name = Morphism.binding phi (Binding.make (Ptr-Valid-Translation.ptr-validN,
here ))
      |> Binding.name-of
```

val opt = SOME {ptr-valid = ptr-valid, ptr-valid-name = ptr-valid-name}
in
context |> Context.mapping (Ptr-Valid-Translation.add-ptr-valid-print-translation opt) I
end›

This declaration provides the setup for global interpretations. The print translation is triggered on constant (abbreviation) *<instance−name>.ptr-valid* introduced by the iterpretation. Keep in mind, that when printing a term on the internal *PTR-VALID('a)* first the notations / abbreviatons are applied. These introduce the syntactic constant *PTR-VALID('a)* on which the translation then triggers.

end

21.1.2 Syntax *IS-VALID('a)*

context *open-types*
begin

Building on top of *PTR-VALID* we provide syntax that also takes the *heap-typing* of lifted globals into accound: *IS-VALID(32 word) s p* is translated to *PTR-VALID('a) (heap-typing s) p* Note that *heap-typing* is a field of the lifted globals record that is defined later.

end

syntax
-is-valid :: type ⇒ logic ⇒ logic
(‹(‹*indent=1 notation=‹mixfix IS-VALID››IS'-VALID/(1'(-'))*) -› [*0, 1000*] *1000*)

parse-translation ‹
let
fun is-valid-tr ctxt [t, s] =
let
val ***Const-*** ‹*open-types.ptr-valid - for T* › =
Syntax.read-term ctxt Ptr-Valid-Translation.ptr-validN
val typ = Syntax-Phases.decode-typ t
in (***Const*** ‹*open-types.ptr-valid typ*› $ *T*) $ ((*Const (AC-Names.heap-typingN, dummyT*)) $ *s*) *end*
handle Bind => error (IS-VALID: no instance of ptr-valid in context)
in
[(***syntax-const*** ‹*-is-valid*›, *is-valid-tr*)]
end
›

print-translation ‹
let
val show-is-valid = Attrib.setup-config-bool @{binding show-is-valid} (K true)

```
  fun is-valid-tr' ctxt (typ::Const (marked-heap-typing-name, heap-typingT)$s::ts)
=
    if Config.get ctxt show-is-valid then
      let
       val heap-typing-name = try‹Lexicon.unmark-const marked-heap-typing-name
                                 catch - => raise Match›
      val - = if Long-Name.base-name heap-typing-name = AC-Names.heap-typingN
then () else raise Match
      in
        Term.betapplys (Syntax.const syntax-const‹-is-valid› $ typ $ s, ts)
      end
    else raise Match
in
  [(syntax-const‹-ptr-valid›, is-valid-tr')]
end
›

end


theory AbstractArrays
imports
  TypHeapLib
  WordSetup
begin


primrec
  array-addrs :: ('a::mem-type) ptr ⇒ nat ⇒ 'a ptr list
where
  array-addrs - 0 = []
| array-addrs p (Suc n) = p # (array-addrs (p +p 1) n)

declare array-addrs.simps(2) [simp del]


lemma hd-in-array-addrs [simp]:
  (x ∈ set (array-addrs x n)) = (n > 0)
  by (cases n, auto simp: array-addrs.simps(2))

lemma array-addrs-1 [simp]:
  array-addrs p (Suc 0) = [p]
  array-addrs p 1 = [p]
  by (auto simp: array-addrs.simps(2))


lemma array-addrs-ptr-aligned:
    ⟦ x ∈ set (array-addrs p n); ptr-aligned p ⟧ ⟹ ptr-aligned x
  apply (induct n arbitrary: x p)
```

```
  subgoal by (clarsimp)
  subgoal for n x p
    apply (clarsimp simp: array-addrs.simps(2))
    apply (erule disjE)
     apply clarsimp
    apply atomize
    apply (drule-tac x=x in spec)
    apply (drule-tac x=p +_p 1 in spec)
    apply (clarsimp simp: ptr-aligned-plus)
    done
  done


lemma set-array-addrs-unfold-last:
  shows set (array-addrs a (Suc n)) = set (array-addrs a n) ∪ {(a :: ('a::mem-type)
ptr) +_p int n}
    (is ?LHS a n = ?RHS a n)
proof (induct n arbitrary: a)
  fix a
  show ?LHS a 0 = ?RHS a 0
    by clarsimp
next
  fix a n
  assume induct: ⋀a. ?LHS a n = ?RHS a n
  show ?LHS a (Suc n) = ?RHS a (Suc n)
  apply (subst array-addrs.simps(2))
  apply (subst set-simps)
  apply (subst induct [where a=a +_p 1])
  apply (subst array-addrs.simps(2))
  apply (subst set-simps)
  apply (clarsimp simp: CTypesDefs.ptr-add-def field-simps insert-commute)
  done
qed


lemma set-array-addrs:
  set (array-addrs (p :: ('a::mem-type) ptr) n)
         = {x. ∃k. x = p +_p int k ∧ k < n }
  apply (induct n arbitrary: p)
  subgoal by (clarsimp simp: not-less)
  subgoal for n p
    apply (subst set-array-addrs-unfold-last)
    apply atomize
    apply (drule-tac x=p in spec)
    apply (erule ssubst)
    apply (rule set-eqI)
    apply (rule iffI)
     apply clarsimp
     apply (erule disjE)
```

apply *clarsimp*
apply *force*
apply *force*
apply *clarsimp*
apply (*rename-tac k*)
apply (*drule-tac x=k* **in** *spec*)
apply (*clarsimp simp*: *not-less*)
apply (*subgoal-tac k = n*)
apply *clarsimp*
apply *clarsimp*
done
done

end

theory *HeapLift*
imports
In-Out-Parameters
Split-Heap
AbstractArrays
begin

21.2 Refinement Lemmas

lemma *ucast-ucast-id*:
LENGTH('*a*) ≤ *LENGTH*('*b*) ⟹ *ucast* (*ucast* (*x*::'*a*::*len word*)::'*b*::*len word*) = *x*
by (*auto intro*: *ucast-up-ucast-id simp*: *is-up-def source-size-def target-size-def word-size*)

lemma *lense-ucast-signed*:
lense (*unsigned* :: '*a*::*len word* ⇒ '*a signed word*) (λ*v x*. *unsigned* (*v* (*unsigned x*)))
by (*rule lenseI-equiv*) (*simp-all add*: *ucast-ucast-id*)

lemma *pointer-lense-ucast-signed*:
fixes *r* :: '*h* ⇒ '*a*::*len8 word ptr* ⇒ '*a word*
assumes *pointer-lense r w*
shows *pointer-lense*
(λ*h p*. *UCAST*('*a* → '*a signed*) (*r h* (*PTR-COERCE*('*a signed word* → '*a word*) *p*)))
(λ*p m*. *w* (*PTR-COERCE*('*a signed word* → '*a word*) *p*)
(λ*w*. *UCAST*('*a signed* → '*a*) (*m* (*UCAST*('*a* → '*a signed*) *w*))))
proof −
interpret *pointer-lense r w* **by** *fact*
note *scast-ucast-norm*[*simp del*]
note *ucast-ucast-id*[*simp*]

```
  show ?thesis
    apply unfold-locales
    apply (simp add: read-write-same)
    apply (simp add: write-same)
    apply (simp add: comp-def)
    apply (simp add: write-other-commute typ-uinfo-t-signed-word-word-conv
                flip: size-of-tag typ-uinfo-size)
    done
qed

lemma (in xmem-type) length-to-bytes:
  length (to-bytes (v::'a) bs) = size-of TYPE('a)
  by (simp add: to-bytes-def lense.access-result-size)

lemma (in xmem-type) heap-update-padding-eq:
  length bs = size-of TYPE('a) ⟹
    heap-update-padding p v bs h = heap-update p v (heap-update-list (ptr-val p) bs
h)
  using u.max-size
  by (simp add: heap-update-padding-def heap-update-def size-of-def
      heap-list-heap-update-list-id heap-update-list-overwrite)

lemma (in xmem-type) heap-update-padding-eq':
  length bs = size-of TYPE('a) ⟹
    heap-update-padding p v bs = heap-update p v ∘ heap-update-list (ptr-val p) bs
  by (simp add: fun-eq-iff heap-update-padding-eq)

lemma split-disj-asm: P (x ∨ y) = (¬ (x ∧ ¬ P x ∨ ¬ x ∧ ¬ P y))
  by (smt (verit, best))

lemma comp-commute-of-fold:
  assumes x: x = fold f xs
  assumes xs: list-all (λx. f x o a = a o f x) xs
  shows x o a = a o x
  unfolding x using xs by (induction xs) (simp-all add: fun-eq-iff)

definition padding-closed-under-all-fields where
  padding-closed-under-all-fields t ⟷
    (∀ s f n bs bs'. field-lookup t f 0 = Some (s, n) ⟶
       eq-upto-padding t bs bs' ⟶ eq-upto-padding s (take (size-td s) (drop n bs))
(take (size-td s) (drop n bs')))

lemma padding-closed-under-all-fields-typ-uinfo-t:
  padding-closed-under-all-fields (typ-uinfo-t TYPE('a::xmem-type))
  unfolding padding-closed-under-all-fields-def
proof safe
  fix s f n bs bs' assume s-n: field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (s,
n)
    and bs-bs': eq-upto-padding (typ-uinfo-t TYPE('a)) bs bs'
```

```
  then have len: length bs = size-of TYPE('a) length bs' = size-of TYPE('a)
    by (auto simp: eq-upto-padding-def size-of-def)

  from s-n[THEN field-lookup-uinfo-Some-rev] obtain k where
    k: field-lookup (typ-info-t TYPE('a)) f 0 = Some (k, n) and k-s: export-uinfo
k = s
    by auto
  have [simp]: size-td k = size-td s by (simp flip: k-s)
  from xmem-type-field-lookup-eq-upto-padding-focus[OF k len bs-bs']
  show eq-upto-padding s (take (size-td s) (drop n bs)) (take (size-td s) (drop n
bs'))
    unfolding k-s by simp
qed

lemma (in open-types) plift-heap-update-list-eq-upto-padding:
  assumes t: mem-type-u t and t': padding-closed-under-all-fields t
  assumes a: ptr-valid-u t (hrs-htd h) a
  assumes bs-bs': eq-upto-padding t bs bs'
  shows plift (hrs-mem-update (heap-update-list a bs) h) =
    (plift (hrs-mem-update (heap-update-list a bs') h)::'a::xmem-type ptr ⇒ 'a op-
tion)
  apply (rule plift-eq-plift, simp-all add: h-val-def hrs-mem-update)
proof −
  from bs-bs' have [simp]: length bs = size-td t length bs' = size-td t
    by (simp-all add: eq-upto-padding-def)
  have [arith]: size-td t < addr-card
    using mem-type-u.max-size[OF t] by simp
  have a-bnd: size-of TYPE('a) ≤ addr-card
    using max-size[where 'a='a] by arith
  let ?A = typ-uinfo-t TYPE('a)

  fix p :: 'a ptr assume p: ptr-valid (hrs-htd h) p
  from ptr-valid-u-cases-weak[OF t a this[unfolded ptr-valid-def]]
 show from-bytes (heap-list (heap-update-list a bs (hrs-mem h)) (size-of TYPE('a))
(ptr-val p)) =
    (from-bytes (heap-list (heap-update-list a bs' (hrs-mem h)) (size-of TYPE('a))
(ptr-val p))::'a)
  proof (elim disjE exE conjE)
    assume disjnt {a..+size-td t} {ptr-val p..+size-td (typ-uinfo-t TYPE('a))}
    with bs-bs' show ?thesis
      unfolding heap-upd-list-def
     by (subst (1 2) heap-list-update-disjoint-same; simp add: size-of-def disjnt-def)
  next
    fix path assume path: addressable-field t path ?A and
      p-eq: ptr-val p = a + word-of-nat (field-offset-untyped t path)
    let ?n = field-offset-untyped t path
    have sz: size-of TYPE('a) + ?n ≤ size-td t
      using field-lookup-offset-size'[OF addressable-fieldD-field-lookup'[OF path]]
      by (simp add: size-of-def)
```

```
    let ?s = typ-uinfo-t TYPE('a)
    from addressable-fieldD-field-lookup'[OF path] t' bs-bs' have ∗:
      eq-upto-padding ?s (take (size-td ?s) (drop ?n bs)) (take (size-td ?s) (drop ?n
bs'))
      unfolding padding-closed-under-all-fields-def
      by (auto simp flip: typ-uinfo-size)

    show ?thesis unfolding p-eq
      using eq-upto-padding-from-bytes-eq[OF ∗] sz
      apply (subst (1 2) heap-list-update-list)
      apply (simp-all add: size-of-def)
      done
  next
    fix path assume path: addressable-field ?A path t
      and p-eq: a = ptr-val p + word-of-nat (field-offset-untyped ?A path)
    let ?n = field-offset-untyped ?A path
    have sz: size-td t + ?n ≤ size-of TYPE('a)
      using field-lookup-offset-size'[OF addressable-fieldD-field-lookup'[OF path]]
      by (simp add: size-of-def)
    from field-lookup-uinfo-Some-rev[OF addressable-fieldD-field-lookup'[OF path]]
obtain k
      where k: field-lookup (typ-info-t TYPE('a)) path 0 = Some (k, ?n)
        and eq-t: export-uinfo k = t by blast
    then have [simp]: size-td k = size-td t
      by (simp flip: eq-t)

    have ∗: eq-upto-padding (typ-uinfo-t TYPE('a))
     (super-update-bs bs (heap-list (hrs-mem h) (size-of TYPE('a)) (ptr-val p)) ?n)
       (super-update-bs bs' (heap-list (hrs-mem h) (size-of TYPE('a)) (ptr-val p))
?n)
      by (subst xmem-type-field-lookup-eq-upto-padding-super-update-bs[OF k(1)])
        (simp-all add: eq-t bs-bs')

    note 1 = c-guard-no-wrap'[OF ptr-valid-c-guard, OF p]
    show ?thesis unfolding p-eq using sz
      apply (subst (1 2) heap-update-list-super-update-bs-heap-list[OF 1])
      apply (simp-all add: heap-list-heap-update-list-id[OF a-bnd])
      apply (intro eq-upto-padding-from-bytes-eq[OF ∗])
      done
  qed
qed

lemma (in open-types) read-dedicated-heap-heap-update-list-eq-upto-padding[simp]:
  assumes t: mem-type-u t and t': padding-closed-under-all-fields t
  assumes a: ptr-valid-u t (hrs-htd h) a
  assumes bs-bs': eq-upto-padding t bs bs'
  shows read-dedicated-heap (hrs-mem-update (heap-update-list a bs) h) =
    (read-dedicated-heap (hrs-mem-update (heap-update-list a bs') h)::'a::xmem-type
ptr ⇒ 'a) ⟷ True
```

by (*simp add*: *plift-heap-update-list-eq-upto-padding*[*OF assms*] *read-dedicated-heap-def fun-eq-iff*)

definition *L2Tcorres st A C* = *corresXF st* (λ*r* -. *r*) (λ*r* -. *r*) (λ-. *True*) *A C*

lemma *L2Tcorres-id*:
L2Tcorres id C C
by (*metis L2Tcorres-def corresXF-id*)

lemma *L2Tcorres-fail*:
L2Tcorres st L2-fail X
apply (*clarsimp simp*: *L2Tcorres-def L2-defs*)
apply (*rule corresXF-fail*)
done

lemma *admissible-nondet-ord-L2Tcorres* [*corres-admissible*]:
ccpo.admissible Inf (≥) (λ*A*. *L2Tcorres st A C*)
unfolding *L2Tcorres-def*
apply (*rule admissible-nondet-ord-corresXF*)
done

lemma *L2Tcorres-top* [*corres-top*]: *L2Tcorres st* ⊤ *C*
by (*auto simp add*: *L2Tcorres-def corresXF-def*)

definition *abs-guard st A C* ≡ ∀*s*. *A* (*st s*) ⟶ *C s*
definition *abs-expr st P A C* ≡ ∀*s*. *P* (*st s*) ⟶ *C s* = *A* (*st s*)
definition *abs-modifies st P A C* ≡ ∀*s*. *P* (*st s*) ⟶ *st* (*C s*) = *A* (*st s*)

definition *struct-rewrite-guard A C* ≡ ∀*s*. *A s* ⟶ *C s*
definition *struct-rewrite-expr P A C* ≡ ∀*s*. *P s* ⟶ *C s* = *A s*
definition *struct-rewrite-modifies P A C* ≡ ∀*s*. *P s* ⟶ *C s* = *A s*

named-theorems *heap-abs*

named-theorems *heap-abs-fo*

named-theorems *derived-heap-defs* **and**
valid-array-defs **and**
heap-upd-cong **and**
valid-same-typ-descs

lemma *deepen-heap-upd-cong*: *f* = *f'* ⟹ *upd f s* = *upd f' s*
by *simp*

lemma *deepen-heap-map-cong*: *f* = *f'* ⟹ *upd f p s* = *upd f' p s*

by *simp*

lemma *abs-expr-fun-app2* [*heap-abs-fo*]:
⟦ *abs-expr st P f f′*;
abs-expr st Q g g′ ⟧ ⟹
abs-expr st (λ*s*. *P s* ∧ *Q s*) (λ*s a*. *f s a* (*g s a*)) (λ*s a*. *f′ s a* $ *g′ s a*)
by (*simp add*: *abs-expr-def*)

lemma *abs-expr-fun-app* [*heap-abs-fo*]:
⟦ *abs-expr st Y x x′*; *abs-expr st X f f′* ⟧ ⟹
abs-expr st (λ*s*. *X s* ∧ *Y s*) (λ*s*. *f s* (*x s*)) (λ*s*. *f′ s* $ *x′ s*)
apply (*clarsimp simp*: *abs-expr-def*)
done

lemma *abs-expr-Pair* [*heap-abs*]:
abs-expr st X f1 f1′ ⟹ *abs-expr st Y f2 f2′* ⟹
abs-expr st (λ*s*. *X s* ∧ *Y s*) (λ*s*. (*f1 s*, *f2 s*)) (λ*s*. (*f1′ s*, *f2′ s*))
apply (*clarsimp simp*: *abs-expr-def*)
done

lemma *abs-expr-constant* [*heap-abs*]:
abs-expr st (λ-. *True*) (λ*s*. *a*) (λ*s*. *a*)
apply (*clarsimp simp*: *abs-expr-def*)
done

lemma *abs-guard-expr* [*heap-abs*]:
abs-expr st P a′ a ⟹ *abs-guard st* (λ*s*. *P s* ∧ *a′ s*) *a*
by (*simp add*: *abs-expr-def abs-guard-def*)

lemma *abs-guard-constant* [*heap-abs*]:
abs-guard st (λ-. *P*) (λ-. *P*)
by (*clarsimp simp*: *abs-guard-def*)

lemma *abs-guard-conj* [*heap-abs*]:
⟦ *abs-guard st G G′*; *abs-guard st H H′* ⟧
⟹ *abs-guard st* (λ*s*. *G s* ∧ *H s*) (λ*s*. *G′ s* ∧ *H′ s*)
by (*clarsimp simp*: *abs-guard-def*)

lemma *L2Tcorres-modify* [*heap-abs*]:
⟦ *struct-rewrite-modifies P b c*; *abs-guard st P′ P*;
abs-modifies st Q a b ⟧ ⟹
L2Tcorres st (*L2-seq* (*L2-guard* (λ*s*. *P′ s* ∧ *Q s*)) (λ-. (*L2-modify a*)))
(*L2-modify c*)
apply (*auto intro*!: *refines-bind-guard-right refines-modify*
simp: *corresXF-refines-conv*

L2Tcorres-def L2-defs abs-modifies-def abs-guard-def struct-rewrite-modifies-def struct-rewrite-guard-def)
done

lemma *L2Tcorres-gets* [*heap-abs*]:
⟦ *struct-rewrite-expr P b c*; *abs-guard st P′ P*;
abs-expr st Q a b ⟧ ⟹
L2Tcorres st (*L2-seq* (*L2-guard* (λ*s*. *P′ s* ∧ *Q s*)) (λ-. *L2-gets a n*)) (*L2-gets c n*)
apply (*auto intro*!: *refines-bind-guard-right refines-gets*
simp: *corresXF-refines-conv L2Tcorres-def L2-defs abs-expr-def abs-guard-def struct-rewrite-expr-def struct-rewrite-guard-def*)
done

lemma *L2Tcorres-gets-const* [*heap-abs*]:
L2Tcorres st (*L2-gets* (λ-. *a*) *n*) (*L2-gets* (λ-. *a*) *n*)
apply (*simp add*: *corresXF-refines-conv refines-gets L2Tcorres-def L2-defs*)
done

lemma *L2Tcorres-guard* [*heap-abs*]:
⟦ *struct-rewrite-guard b c*; *abs-guard st a b* ⟧ ⟹
L2Tcorres st (*L2-guard a*) (*L2-guard c*)
apply (*simp add*: *corresXF-def L2Tcorres-def L2-defs abs-guard-def struct-rewrite-guard-def*)
done

lemma *L2Tcorres-while* [*heap-abs*]:
assumes *body-corres* [*simplified THIN-def*,*rule-format*]:
PROP THIN (⋀*x*. *L2Tcorres st* (*B′ x*) (*B x*))
and *cond-rewrite* [*simplified THIN-def*,*rule-format*]:
PROP THIN (⋀*r*. *struct-rewrite-expr* (*G r*) (*C′ r*) (*C r*))
and *guard-abs*[*simplified THIN-def*,*rule-format*]:
PROP THIN (⋀*r*. *abs-guard st* (*G′ r*) (*G r*))
and *guard-impl-cond*[*simplified THIN-def*,*rule-format*]:
PROP THIN (⋀*r*. *abs-expr st* (*H r*) (*C″ r*) (*C′ r*))
shows *L2Tcorres st* (*L2-guarded-while* (λ*i s*. *G′ i s* ∧ *H i s*) *C″ B′ i n*) (*L2-while C B i n*)
proof −

have *cond-match*: ⋀*r s*. *G′ r* (*st s*) ∧ *H r* (*st s*) ⟹ *C″ r* (*st s*) = *C r s*
using *cond-rewrite guard-abs guard-impl-cond*
by (*clarsimp simp*: *abs-expr-def abs-guard-def struct-rewrite-expr-def*)

have *corresXF st* (λ*r* -. *r*) (λ*r* -. *r*) (λ-. *True*)
(*do* { - ← *guard* (λ*s*. *G′ i s* ∧ *H i s*);
whileLoop C″
(λ*i*. *do* { *r* ← *B′ i*;
- ← *guard* (λ*s*. *G′ r s* ∧ *H r s*);
return r
}) *i*

```
        })
    (whileLoop C B i)
   apply (rule corresXF-guard-imp)
   apply (rule corresXF-guarded-while [where P=λ- -. True and P'=λ- -. True])
       apply (clarsimp cong: corresXF-cong)
       apply (rule corresXF-guard-imp)
        apply (rule body-corres [unfolded L2Tcorres-def])
       apply simp
      apply (clarsimp simp: cond-match)
     apply clarsimp
     apply (simp add: runs-to-partial-def-old split: xval-splits)
    apply simp
   apply simp
  apply simp
  done

 thus ?thesis
  by (clarsimp simp: L2Tcorres-def L2-defs gets-return top-fun-def)
qed


named-theorems abs-spec

definition abs-spec st P (A :: ('a × 'a) set) (C :: ('c × 'c) set)
        ≡ (∀ s t. P (st s) ⟶ (((s, t) ∈ C) ⟶ ((st s, st t) ∈ A)))
                    ∧ (∀ s. P (st s) ⟶ (∃ x. (st s, x) ∈ A) ⟶ (∃ x. (s, x) ∈
C))

lemma L2Tcorres-spec [heap-abs]:
 ⟦ abs-spec st P A C ⟧
   ⟹ L2Tcorres st (L2-seq (L2-guard P) (λ-. (L2-spec A))) (L2-spec C)
 unfolding corresXF-refines-conv L2Tcorres-def L2-defs
 apply (clarsimp simp add: abs-spec-def)
apply (intro refines-bind-guard-right refines-bind-bind-exn-wp refines-state-select)
 apply (force intro!: refines-select simp add: abs-spec-def split: xval-splits)
 apply blast
 done


definition abs-assume st P (A :: 'a ⇒ ('b × 'a) set) (C :: 'c ⇒ ('b × 'c) set)
 ≡ (∀ s t r. P (st s) ⟶ (((r, t) ∈ C s) ⟶ ((r, st t) ∈ A (st s))))


lemma refines-assume-result-and-state':
 refines (assume-result-and-state P) (assume-result-and-state Q) s t R
 if sim-set (λ(v, s) (w, t). R (Result v, s) (Result w, t)) (P s) (Q t)
 using that
 by (force simp: refines-def-old  sim-set-def rel-set-def  case-prod-unfold)
```

lemma *L2Tcorres-assume* [*heap-abs*]:
  ⟦ *abs-assume st P A C* ⟧
    ⟹ *L2Tcorres st* (*L2-seq* (*L2-guard P*) (λ-. (*L2-assume A*))) (*L2-assume C*)
  unfolding *corresXF-refines-conv L2Tcorres-def L2-defs*
  apply (*clarsimp simp add*: *abs-assume-def*) **thm** *refines-mono* [*OF - refines-assume-result-and-state*]
  apply (*intro refines-bind-guard-right refines-bind-bind-exn-wp refines-assume-result-and-state′*
)
  apply (*auto simp add*: *sim-set-def*)
  done

lemma *abs-spec-constant* [*heap-abs*]:
  abs-spec st (λ-. *True*) {(*a*, *b*). *C*} {(*a*, *b*). *C*}
  apply (*clarsimp simp*: *abs-spec-def*)
  done

lemma *L2Tcorres-condition* [*heap-abs*]:
  ⟦*PROP THIN* (*Trueprop* (*L2Tcorres st L L′*));
    PROP THIN (*Trueprop* (*L2Tcorres st R R′*));
    PROP THIN (*Trueprop* (*struct-rewrite-expr P C′ C*));
    PROP THIN (*Trueprop* (*abs-guard st P′ P*));
    PROP THIN (*Trueprop* (*abs-expr st Q C″ C′*))⟧ ⟹
   L2Tcorres st (*L2-seq* (*L2-guard* (λ*s*. *P′ s* ∧ *Q s*)) (λ-. *L2-condition C″ L R*))
(*L2-condition C L′ R′*)
  unfolding *THIN-def L2-defs L2Tcorres-def corresXF-refines-conv*
  apply *clarsimp*
  apply (*intro refines-bind-guard-right refines-condition*)
  apply (*auto simp add*: *abs-expr-def abs-guard-def struct-rewrite-expr-def struct-rewrite-guard-def*)
  done

lemma *L2Tcorres-seq* [*heap-abs*]:
  ⟦*PROP THIN* (*Trueprop* (*L2Tcorres st L′ L*)); *PROP THIN* (⋀*r*. *L2Tcorres st*
(*R′ r*) (*R r*)) ⟧
      ⟹ *L2Tcorres st* (*L2-seq L′ R′*) (*L2-seq L R*)
  unfolding *THIN-def L2-defs L2Tcorres-def corresXF-refines-conv*
  apply *clarsimp*
  apply (*intro refines-bind-bind-exn-wp*)
  subgoal for *t*
    apply (*erule-tac x*=*t* **in** *allE*)
    apply (*rule refines-weaken*)
     apply *assumption*
    apply (*auto split*: *xval-splits*)
    done
  done

lemma *L2Tcorres-guarded-simple* [*heap-abs*]:
  assumes *b-c*: *struct-rewrite-guard b c*
  assumes *a-b*: *abs-guard st a b*

assumes *f-g*: $\bigwedge s\ s'.\ c\ s \Longrightarrow s' = st\ s \Longrightarrow L2Tcorres\ st\ f\ g$
shows *L2Tcorres st* (*L2-guarded a f*) (*L2-guarded c g*)
unfolding *L2-guarded-def L2-defs L2Tcorres-def corresXF-refines-conv*
using *b-c a-b f-g*
by (*fastforce simp add*: *refines-def-old L2Tcorres-def corresXF-refines-conv reaches-bind succeeds-bind*
struct-rewrite-guard-def abs-guard-def abs-expr-def split: *xval-splits*)

lemma *L2Tcorres-catch* [*heap-abs*]:
$[\![PROP\ THIN\ (Trueprop\ (L2Tcorres\ st\ L\ L'));$
$PROP\ THIN\ (\bigwedge r.\ L2Tcorres\ st\ (R\ r)\ (R'\ r))$
$]\!] \Longrightarrow L2Tcorres\ st\ (L2\text{-}catch\ L\ R)\ (L2\text{-}catch\ L'\ R')$
unfolding *THIN-def*
apply (*clarsimp simp*: *L2Tcorres-def L2-defs*)
apply (*rule corresXF-guard-imp*)
apply (*erule corresXF-except* [**where** P'=$\lambda x\ y\ s.\ x = y$ **and** Q=λ-. *True*])
apply (*simp add*: *corresXF-refines-conv*)
apply (*simp add*: *runs-to-partial-def-old split*: *xval-splits*)
apply *simp*
apply *simp*
done

lemma *corresXF-return-same*:
corresXF st (λr -. *r*) (λr -. *r*) (λ-. *True*) (*return e*) (*return e*)
by (*clarsimp simp add*: *corresXF-def*)

lemma *corresXF-yield-same*:
corresXF st (λr -. *r*) (λr -. *r*) (λ-. *True*) (*yield e*) (*yield e*)
by (*auto simp add*: *corresXF-refines-conv intro*!: *refines-yield split*: *xval-splits*)

lemma *L2-try-catch*: *L2-try L* = *L2-catch L* ($\lambda e.$ *yield* (*to-xval e*))
unfolding *L2-defs*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*auto simp add*: *runs-to-def-old unnest-exn-def to-xval-def split*: *xval-splits sum.splits*)
done

lemma *L2Tcorres-try* [*heap-abs*]:
$[\![\ L2Tcorres\ st\ L\ L']\!] \Longrightarrow L2Tcorres\ st\ (L2\text{-}try\ L)\ (L2\text{-}try\ L')$
apply (*simp add*: *L2-try-catch*)
apply (*erule L2Tcorres-catch* [*simplified THIN-def*])
apply (*unfold L2Tcorres-def top-fun-def top-bool-def*)
apply (*rule corresXF-yield-same*)
done

lemma *L2Tcorres-unknown* [*heap-abs*]:
L2Tcorres st (*L2-unknown ns*) (*L2-unknown ns*)
apply (*clarsimp simp*: *L2-unknown-def*)

apply (*clarsimp simp*: *L2Tcorres-def*)
apply (*auto intro*!: *corresXF-select-select*)
done

lemma *L2Tcorres-throw* [*heap-abs*]:
L2Tcorres st (*L2-throw x n*) (*L2-throw x n*)
apply (*clarsimp simp*: *L2Tcorres-def L2-defs*)
apply (*rule corresXF-throw*)
apply *simp*
done

lemma *L2Tcorres-split* [*heap-abs*]:
⟦ ⋀*x y*. *L2Tcorres st* (*P x y*) (*P′ x y*) ⟧ ⟹
L2Tcorres st (*case a of* (*x*, *y*) ⇒ *P x y*) (*case a of* (*x*, *y*) ⇒ *P′ x y*)
apply (*clarsimp simp*: *split-def*)
done

lemma *L2Tcorres-seq-unused-result* [*heap-abs*]:
⟦*PROP THIN* (*Trueprop* (*L2Tcorres st L L′*)); *PROP THIN* (*Trueprop* (*L2Tcorres st R R′*)) ⟧
⟹ *L2Tcorres st* (*L2-seq L* (λ-. *R*)) (*L2-seq L′* (λ-. *R′*))
apply (*rule L2Tcorres-seq*, *auto*)
done

lemma *abs-expr-split* [*heap-abs*]:
⟦ ⋀*a b*. *abs-expr st* (*P a b*) (*A a b*) (*C a b*) ⟧
⟹ *abs-expr st* (*case r of* (*a*, *b*) ⇒ *P a b*)
(*case r of* (*a*, *b*) ⇒ *A a b*) (*case r of* (*a*, *b*) ⇒ *C a b*)
apply (*auto simp*: *split-def*)
done

lemma *abs-guard-split* [*heap-abs*]:
⟦ ⋀*a b*. *abs-guard st* (*A a b*) (*C a b*) ⟧
⟹ *abs-guard st* (*case r of* (*a*, *b*) ⇒ *A a b*) (*case r of* (*a*, *b*) ⇒ *C a b*)
apply (*auto simp*: *split-def*)
done

lemma *L2Tcorres-abstract-fail* [*heap-abs*]:
L2Tcorres st L2-fail L2-fail
apply (*clarsimp simp*: *L2Tcorres-def L2-defs*)
apply (*rule corresXF-fail*)
done

lemma *abs-expr-id* [*heap-abs*]:
abs-expr id (λ-. *True*) *A A*
apply (*clarsimp simp*: *abs-expr-def*)
done

lemma *abs-expr-lambda-null* [*heap-abs*]:

abs-expr st P A C $\Longrightarrow$ *abs-expr st P* ($\lambda s\ r.\ A\ s$) ($\lambda s\ r.\ C\ s$)
apply (*clarsimp simp*: *abs-expr-def*)
done

lemma *abs-modify-id* [*heap-abs*]:
abs-modifies id (λ-. *True*) *A A*
apply (*clarsimp simp*: *abs-modifies-def*)
done

lemma *corresXF-exec-concrete* [*intro?*]:
corresXF id ret-xf ex-xf P A C $\Longrightarrow$ *corresXF st ret-xf ex-xf P* (*exec-concrete st A*) *C*
by (*force simp add*: *corresXF-refines-conv refines-def-old reaches-exec-concrete succeeds-exec-concrete-iff split*: *xval-splits*)

lemma *L2Tcorres-exec-concrete* [*heap-abs*]:
L2Tcorres id A C $\Longrightarrow$ *L2Tcorres st* (*exec-concrete st* (*L2-call A emb ns*)) (*L2-call C emb ns*)
apply (*clarsimp simp*: *L2Tcorres-def L2-call-def map-exn-catch-conv*)
apply (*rule corresXF-exec-concrete*)
apply (*rule CorresXF.corresXF-except* [**where** $P' = \lambda x\ y\ s.\ x = y$])
apply *assumption*
subgoal
by (*auto simp add*: *corresXF-refines-conv*)
subgoal
by (*auto simp add*: *runs-to-partial-def-old split*: *xval-splits*)
subgoal by *simp*
done

lemma *L2Tcorres-exec-concrete-simpl* [*heap-abs*]:
L2Tcorres id A C $\Longrightarrow$ *L2Tcorres st* (*exec-concrete st* (*L2-call-L1 arg-xf gs ret-xf A*)) (*L2-call-L1 arg-xf gs ret-xf C*)
apply (*clarsimp simp*: *L2Tcorres-def L2-call-L1-def*)
apply (*rule corresXF-exec-concrete*)
apply (*clarsimp simp add*: *corresXF-refines-conv*)
apply (*rule refines-bind-bind-exn-wp*)
apply (*clarsimp split*: *xval-splits*)
apply (*rule refines-get-state*)
apply (*clarsimp split*: *xval-splits*)
apply (*rule refines-bind-bind-exn-wp*)
apply (*clarsimp split*: *xval-splits*)
apply (*rule refines-select*)
apply (*clarsimp split*: *xval-splits*)
subgoal for *x*
apply (*rule exI*[**where** *x*=*x*])
apply (*erule-tac x*=*x* **in** *allE*)
apply (*clarsimp*)
apply (*rule refines-run-bind*)
apply (*clarsimp split*: *exception-or-result-splits*)

apply (*erule refines-weaken*)
apply (*clarsimp split*: *xval-splits*)
apply (*rule refines-bind-bind-exn-wp*)
apply (*clarsimp split*: *xval-splits*)
apply (*rule refines-set-state*)
apply (*clarsimp split*: *xval-splits*)
done
done

lemma *corresXF-exec-abstract* [*intro?*]:
corresXF st ret-xf ex-xf P A C $\Longrightarrow$ *corresXF id ret-xf ex-xf P* (*exec-abstract st A*) *C*
by (*force simp*: *corresXF-refines-conv refines-def-old reaches-exec-abstract split*: *xval-splits*)

lemma *L2Tcorres-exec-abstract* [*heap-abs*]:
L2Tcorres st A C $\Longrightarrow$ *L2Tcorres id* (*exec-abstract st* (*L2-call A emb ns*)) (*L2-call C emb ns*)
apply (*clarsimp simp*: *L2-call-def map-exn-catch-conv L2Tcorres-def*)
apply (*rule corresXF-exec-abstract*)
apply (*rule CorresXF.corresXF-except* [**where** $P' = \lambda x\ y\ s.\ x = y$])
apply *assumption*
subgoal by (*auto simp add*: *corresXF-refines-conv*)
subgoal by (*auto simp add*: *runs-to-partial-def-old split*: *xval-splits*)
subgoal by *simp*
done

lemma *L2Tcorres-call* [*heap-abs*]:
L2Tcorres st A C $\Longrightarrow$ *L2Tcorres st* (*L2-call A emb ns*) (*L2-call C emb ns*)
unfolding *L2Tcorres-def L2-call-def map-exn-catch-conv*
apply (*rule CorresXF.corresXF-except* [**where** $P' = \lambda x\ y\ s.\ x = y$])
apply *assumption*
subgoal by (*auto simp add*: *corresXF-refines-conv*)
subgoal by (*auto simp add*: *runs-to-partial-def-old split*: *xval-splits*)
subgoal by *simp*
done

named-theorems
valid-implies-c-guard **and**
read-commutes **and**
write-commutes **and**
field-write-commutes **and**
write-valid-preservation **and**
lift-heap-update-padding-heap-update-conv

locale *valid-implies-cguard* =
  fixes *st*::'*s* ⇒ '*t*
  fixes *v*::'*t* ⇒ '*a*::*c-type ptr* ⇒ *bool*
  assumes *valid-implies-c-guard*[*valid-implies-c-guard*]: *v* (*st s*) *p* ⟹ *c-guard p*

locale *read-simulation* =
  fixes *st* ::'*s* ⇒ '*t*
  fixes *v* ::'*t* ⇒ '*a*::*c-type ptr* ⇒ *bool*
  fixes *r* :: '*t* ⇒ '*a ptr* ⇒ '*a*
  fixes *t-hrs*::'*s* ⇒ *heap-raw-state*
  assumes *read-commutes*[*read-commutes*]: *v* (*st s*) *p* ⟹
            r (*st s*) *p* = *h-val* (*hrs-mem* (*t-hrs s*)) *p*

locale *write-simulation* =
  heap-raw-state t-hrs t-hrs-upd
  for
    t-hrs :: ('*s* ⇒ *heap-raw-state*) **and**
    t-hrs-upd::(*heap-raw-state* ⇒ *heap-raw-state*) ⇒ '*s* ⇒ '*s* +
  fixes *st* ::'*s* ⇒ '*t*
  fixes *v* ::'*t* ⇒ '*a*::*mem-type ptr* ⇒ *bool*
  fixes *w* :: '*a ptr* ⇒ ('*a* ⇒ '*a*) ⇒ '*t* ⇒ '*t*

  assumes *write-padding-commutes*[*write-commutes*]: *v* (*st s*) *p* ⟹ *length bs* = *size-of TYPE*('*a*) ⟹
          st (*t-hrs-upd* (*hrs-mem-update* (*heap-update-padding p x bs*)) *s*) =
                  w p (λ-. *x*) (*st s*)

begin
lemma *write-commutes*[*write-commutes*]:
  assumes *valid*: *v* (*st s*) *p*
  shows *st* (*t-hrs-upd* (*hrs-mem-update* (*heap-update p x*)) *s*) =
                  w p (λ-. *x*) (*st s*)
proof −
  have *eq*: *hrs-mem-update* (*heap-update p x*) =
      hrs-mem-update (λ*h*. *heap-update-padding p x* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)) *h*)
    using *heap-update-heap-update-padding-conv*
    by *metis*

  show *?thesis*
    apply (*simp only*: *eq*)
   apply (*subst write-padding-commutes* [*symmetric*, **where** *bs*=*heap-list* (*hrs-mem* (*t-hrs s*)) (*size-of TYPE*('*a*)) (*ptr-val p*)])
     apply (*rule valid*)
    apply *clarsimp*
    by (*metis* (*no-types*, *lifting*) *heap.upd-cong*)
qed

lemma *lift-heap-update-padding-heap-update-conv*[*lift-heap-update-padding-heap-update-conv*]:
v (*st s*) *p* ⟹ *length bs* = *size-of TYPE*('*a*) ⟹
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update-padding p x bs*)) *s*) =
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update p x*)) *s*)
using *write-padding-commutes write-commutes* **by** *auto*

lemma *write-commutes-atomic*: ∀ *s p x*. *v* (*st s*) *p* ⟶
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update p x*)) *s*) =
w p (λ-. *x*) (*st s*)
using *write-commutes* **by** *blast*

end

locale *write-preserves-valid* =
fixes *v* ::'*t* ⇒ '*a*::*c-type ptr* ⇒ *bool*
fixes *w* :: '*a ptr* ⇒ ('*b* ⇒ '*b*) ⇒ '*t* ⇒ '*t*
assumes *valid-preserved*: *v* (*w p*' *f s*) *p* = *v s p*
begin
lemma *valid-preserved-pointless*[*simp*]: *v* (*w p*' *f s*) = *v s*
by (*rule ext*) (*rule valid-preserved*)
end

locale *valid-only-typ-desc-dependent* =
fixes *t-hrs* :: ('*s* ⇒ *heap-raw-state*)
fixes *st* ::'*s* ⇒ '*t*
fixes *v* ::'*t* ⇒ '*a*::*c-type ptr* ⇒ *bool*
assumes *valid-same-typ-desc* [*valid-same-typ-descs*]: *hrs-htd* (*t-hrs s*) = *hrs-htd* (*t-hrs t*) ⟹ *v* (*st s*) *p* = *v* (*st t*) *p*

locale *heap-typing-simulation* =
open-types $\mathcal{T}$ + *t-hrs*: *heap-raw-state t-hrs t-hrs-upd* + *heap-typing-state heap-typing heap-typing-upd*
for
$\mathcal{T}$ **and**
t-hrs :: ('*s* ⇒ *heap-raw-state*) **and**
t-hrs-upd :: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ ('*s* ⇒ '*s*) **and**
heap-typing :: '*t* ⇒ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ '*t* ⇒ '*t* +
fixes *st* ::'*s* ⇒ '*t*
assumes *heap-typing-commutes*[*simp*]: *heap-typing* (*st s*) = *hrs-htd* (*t-hrs s*)
assumes *lift-heap-update-list-stack-byte-independent*:
(⋀*i*. *i* < *length bs* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) ((*p*::*stack-byte ptr*) +$_p$ *int i*)) ⟹
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*)) *s*) = *st s*
assumes *st-eq-upto-padding*:
mem-type-u t ⟹ *padding-closed-under-all-fields t* ⟹
ptr-valid-u t (*hrs-htd* (*t-hrs s*)) *a* ⟹ *eq-upto-padding t bs bs*' ⟹

```
    st (t-hrs-upd (hrs-mem-update (heap-update-list a bs)) s) =
    st (t-hrs-upd (hrs-mem-update (heap-update-list a bs')) s)
begin

lemma heap-typing-upd-commutes: heap-typing (heap-typing-upd f (st s)) = hrs-htd
(t-hrs (t-hrs-upd (hrs-htd-update f) s))
  apply (simp add: hrs-htd-update)
  done

lemma write-simulation-alt:
  assumes v: ⋀s p. v (st s) p ⟹ ptr-valid (hrs-htd (t-hrs s)) p
  assumes ∗: ⋀s (p::'a::xmem-type ptr) x. v (st s) p ⟹
    st (t-hrs-upd (hrs-mem-update (heap-update p x)) s) = w p (λ-. x)  (st s)
  shows write-simulation t-hrs t-hrs-upd st v w
proof
  fix s p x and bs :: byte list assume p: v (st s) p and bs: length bs = size-of
TYPE('a)

  have [simp]: t-hrs-upd (hrs-mem-update (heap-update p x)) s =
    t-hrs-upd (hrs-mem-update (heap-update-list (ptr-val p)
      (to-bytes x (heap-list (hrs-mem (t-hrs s)) (size-of TYPE('a)) (ptr-val p))))) s
    by (rule t-hrs.heap.upd-cong) (simp add: heap-update-def)

  show st (t-hrs-upd (hrs-mem-update (heap-update-padding p x bs)) s) = w p (λ-.
x) (st s)
    apply (subst ∗[OF p, symmetric])
    apply (simp add: heap-update-padding-def[abs-def])
    apply (rule st-eq-upto-padding[where t=typ-uinfo-t TYPE('a)])
    apply (rule typ-uinfo-t-mem-type)
    apply (rule padding-closed-under-all-fields-typ-uinfo-t)
    apply (subst ptr-valid-def[symmetric])
    apply (simp add: v p)
    unfolding to-bytes-def typ-uinfo-t-def
    apply (rule field-lookup-access-ti-eq-upto-padding[where f=[] and 'b='a])
    apply (simp-all add: bs size-of-def)
    done
qed

end

locale typ-heap-simulation =
  heap-raw-state t-hrs t-hrs-update +
  read-simulation st v r t-hrs +
  write-simulation t-hrs t-hrs-update st v w  +
  write-preserves-valid v w +
  valid-implies-cguard st v +
  valid-only-typ-desc-dependent t-hrs st v +
  pointer-lense r w
  for
```

st:: *'s* ⇒ *'t* **and**
r:: *'t* ⇒ (*'a*::*xmem-type*) *ptr* ⇒ *'a* **and**
w:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'t* ⇒ *'t* **and**
v:: *'t* ⇒ (*'a*::*xmem-type*) *ptr* ⇒ *bool* **and**
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s*
begin

lemma *write-valid-preservation* [*write-valid-preservation*]:
shows *v* (*st* (*t-hrs-update* (*hrs-mem-update* (*heap-update q x*)) *s*)) *p* = *v* (*st s*) *p*
by (*metis hrs-htd-mem-update get-upd valid-same-typ-desc*)

lemma *write-padding-valid-preservation* [*write-valid-preservation*]:
shows *v* (*st* (*t-hrs-update* (*hrs-mem-update* (*heap-update-padding q x bs*)) *s*)) *p* = *v* (*st s*) *p*
by (*metis hrs-htd-mem-update get-upd valid-same-typ-desc*)

end

locale *stack-simulation* =
heap-typing-simulation $\mathcal{T}$ *t-hrs t-hrs-update heap-typing heap-typing-upd st* +
typ-heap-typing r w heap-typing heap-typing-upd $\mathcal{S}$
for
$\mathcal{T}$ **and**
st:: *'s* ⇒ *'t* **and**
r:: *'t* ⇒ (*'a*::*xmem-type*) *ptr* ⇒ *'a* **and**
w:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'t* ⇒ *'t* **and**
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s* **and**
heap-typing :: *'t* ⇒ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'t* ⇒ *'t* **and**
$\mathcal{S}$:: *addr set* +
assumes *sim-stack-alloc*:
⋀*p d vs bs s n*.
(*p*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*(*'a*) (*hrs-htd* (*t-hrs s*)) ⟹ *length vs* = *n* ⟹ *length bs* = *n* ∗ *size-of TYPE* (*'a*) ⟹
st (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*vs*!*i*) (*take* (*size-of TYPE*(*'a*)) (*drop* (*i* ∗ *size-of TYPE*(*'a*)) *bs*))) [*0*..<*n*]) ∘ *hrs-htd-update* (λ-. *d*)) *s*) =
(*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. (*vs* ! *i*))) [*0*..<*n*]) (*heap-typing-upd* (λ-. *d*) (*st s*))

assumes *sim-stack-release*: ⋀*p s n*. (⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*p* +$_p$ *int i*)) ⟹
length bs = *n* ∗ *size-of TYPE*(*'a*) ⟹
st (*t-hrs-update* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) ∘ *hrs-htd-update* (*stack-releases n p*)) *s*) =

((*heap-typing-upd* (*stack-releases n p*) (*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *c-type-class.zero*)) [*0*..<*n*] (*st s*))))

assumes *stack-byte-zero*: ⋀*p d i*. (*p*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*) (*hrs-htd* (*t-hrs s*)) ⟹ *i* < *n* ⟹ *r* (*st s*) (*p* +$_p$ *int i*) = *ZERO*('*a*)

lemma (**in** *typ-heap-simulation*) *L2Tcorres-IO-modify-paddingE* [*heap-abs*]:
assumes *abs-expr st P a c*
shows *L2Tcorres st* (*L2-seq* (*L2-guard* (λ*t*. *v t p* ∧ *P t*)) (λ-. (*L2-modify* (λ*s*. *w p* (λ-. *a s*) *s*))))
(*IO-modify-heap-paddingE* (*p*::'*a ptr*) *c*)
using *assms*
using *length-to-bytes write-padding-commutes* **unfolding** *liftE-IO-modify-heap-padding*
by (*auto simp add*: *abs-expr-def L2Tcorres-def corresXF-refines-conv L2-defs IO-modify-heap-padding-def refines-def-old reaches-bind succeeds-bind split*: *xval-splits*)

locale *typ-heap-typing-stack-simulation* =
typ-heap-simulation st r w v t-hrs t-hrs-update +
stack-simulation 𝒯 *st r w t-hrs t-hrs-update heap-typing heap-typing-upd* 𝒮
for
𝒯 **and**
st:: '*s* ⇒ '*t* **and**
r:: '*t* ⇒ ('*a*::*xmem-type*) *ptr* ⇒ '*a* **and**
w:: '*a ptr* ⇒ ('*a* ⇒ '*a*) ⇒ '*t* ⇒ '*t* **and**
v:: '*t* ⇒ ('*a*::*xmem-type*) *ptr* ⇒ *bool* **and**
t-hrs :: '*s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ '*s* ⇒ '*s* **and**
heap-typing :: '*t* ⇒ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ '*t* ⇒ '*t* **and**
𝒮:: *addr set*
begin

sublocale *monolithic*: *stack-heap-raw-state t-hrs t-hrs-update* 𝒮
by (*unfold-locales*)

definition *rel-split-heap* ≡ λs_c s_a. s_a = *st* s_c

lemma *rel-split-heap-stack-free-eq*:
rel-split-heap s_c s_a ⟹ *stack-free* (*hrs-htd* (*t-hrs* s_c)) = *stack-free* (*heap-typing* s_a)
by (*simp only*: *rel-split-heap-def heap-typing-commutes*)

definition *rel-stack-free-eq* **where**
rel-stack-free-eq s_c s_a ≡ *stack-free* (*hrs-htd* (*t-hrs* s_c)) = *stack-free* (*heap-typing* s_a)

lemma *rel-prod-rel-split-heap-conv*:
rel-prod (=) *rel-split-heap* = (λ(*v*, *t*) (*r*, *s*).
s = *st t* ∧ (*case v of Exn x* ⇒ *r* = *Exn x* | *Result x* ⇒ *r* = *Result x*))
by (*auto simp add*: *rel-split-heap-def rel-prod-conv fun-eq-iff split*: *prod.splits xval-splits*)

lemma *L2Tcorres-refines*:
L2Tcorres st f_a f_c ⟹ *refines* f_c f_a *s* (*st s*) (*rel-prod* (=) *rel-split-heap*)
by (*simp add*: *L2Tcorres-def corresXF-refines-conv rel-prod-rel-split-heap-conv*)

lemma *refines-L2Tcorres*:
assumes *f*: ⋀*s*. *refines* f_c f_a *s* (*st s*) (*rel-prod* (=) *rel-split-heap*)
shows *L2Tcorres st* f_a f_c
using *f*
by (*simp add*: *L2Tcorres-def corresXF-refines-conv rel-prod-rel-split-heap-conv*)

lemma *L2Tcorres-refines-conv*:
L2Tcorres st f_a f_c ⟷ (∀ *s*. *refines* f_c f_a *s* (*st s*) (*rel-prod* (=) *rel-split-heap*))
by (*auto simp add*: *L2Tcorres-refines refines-L2Tcorres*)

lemma *sim-guard-with-fresh-stack-ptr*:
fixes f_c:: *'a ptr* ⇒ (*'b*, *'c*, *'s*) *exn-monad*
assumes *init*: $init_a$ (*st s*) = $init_c$ *s*
assumes *f*: ⋀*s p*::*'a ptr*. *refines* (f_c *p*) (f_a *p*) *s* (*st s*) (*rel-prod* (=) *rel-split-heap*)
shows *refines*
(*monolithic.with-fresh-stack-ptr n* $init_c$ f_c)
(*guard-with-fresh-stack-ptr n* $init_a$ f_a) *s* (*st s*)
(*rel-prod* (=) *rel-split-heap*)
unfolding *monolithic.with-fresh-stack-ptr-def guard-with-fresh-stack-ptr-def*
stack-ptr-acquire-def stack-ptr-release-def assume-stack-alloc-def
apply (*rule refines-bind-bind-exn* [**where** *Q*= (*rel-prod* (=) *rel-split-heap*)])
subgoal
apply (*rule refines-assume-result-and-state'*)
using *sim-stack-alloc stack-byte-zero*
by (*fastforce simp add*: *sim-set-def rel-split-heap-def init split*: *xval-splits*)
apply *simp*
apply *simp*
apply *simp*
apply (*rule refines-rel-prod-guard-on-exit* [**where** *S'*=*rel-split-heap*])
apply (*subst* (*asm*) *rel-split-heap-def*)
apply *simp*
apply (*rule f*)
subgoal by (*auto simp add*: *rel-split-heap-def sim-stack-release*)
subgoal by (*simp add*: *Ex-list-of-length*)
done

lemma *sim-with-fresh-stack-ptr*:
fixes f_c:: *'a ptr* ⇒ (*'b*, *'c*, *'s*) *exn-monad*

assumes *init*: $init_a\ (st\ s) = init_c\ s$
assumes *f*: $\bigwedge s\ p{::}'a\ ptr.$ *refines* $(f_c\ p)\ (f_a\ p)\ s\ (st\ s)$ (*rel-prod* (=) *rel-split-heap*)
assumes *typing-unchanged*: $\bigwedge s\ p{::}'a\ ptr.\ (f_c\ p) \bullet s\ ?\{\!|\lambda r\ t.$ *typing.unchanged-typing-on* $\mathcal{S}\ s\ t|\!\}$
shows *refines*
(*monolithic.with-fresh-stack-ptr n* $init_c\ f_c$)
(*with-fresh-stack-ptr n* $init_a\ f_a$) *s* (*st s*)
(*rel-prod* (=) *rel-split-heap*)
apply (*simp add*: *monolithic.with-fresh-stack-ptr-def with-fresh-stack-ptr-def stack-ptr-acquire-def stack-ptr-release-def assume-stack-alloc-def*)
apply (*rule refines-bind-bind-exn* [**where** *Q*= $\lambda(r,t)\ (r',t').$
(*rel-prod* (=) *rel-split-heap*) $(r,t)\ (r',t') \wedge$
$(\exists p.\ r =$ *Result* $p \wedge (\forall i < n.$ *ptr-span* $(p +_p$ *int* $i) \subseteq \mathcal{S} \wedge$ *root-ptr-valid* (*heap-typing* t') $(p +_p$ *int* $i)))$], *simp-all*)

subgoal
apply (*rule refines-assume-result-and-state'*)
using *sim-stack-alloc stack-byte-zero stack-allocs-*$\mathcal{S}$
apply (*clarsimp simp add*: *sim-set-def init rel-split-heap-def*, *safe*)
apply *blast*+
by (*smt* (*verit*) *hrs-htd-update stack-allocs-cases*)

subgoal for *p t t'*
apply (*rule refines-runs-to-partial-rel-prod-on-exit* [**where** *S'*=*rel-split-heap* **and** *P*=*typing.unchanged-typing-on* $\mathcal{S}$ *t*])
apply (*subst* (*asm*) *rel-split-heap-def*)
apply *simp*
apply (*rule f*)
apply (*rule typing-unchanged*)
subgoal for $s_c\ s_a\ t_c$
apply *clarsimp*
apply (*clarsimp simp add*: *rel-split-heap-def*)
apply (*subst sim-stack-release*)
subgoal for *bs i*
using *typing.unchanged-typing-on-root-ptr-valid-preservation* [**where** *S*=$\mathcal{S}$ **and** *s*=*t* **and** *t*=s_c **and** *p*= $(p +_p$ *int* $i)$]
by *auto*
subgoal by *auto*
subgoal by *auto*
done
subgoal by (*simp add*: *Ex-list-of-length*)
done
done

lemma *sim-assume-with-fresh-stack-ptr*:
fixes $f_c{::}\ 'a\ ptr \Rightarrow ('b,\ 'c,\ 's)\ exn\text{-}monad$
assumes *init*: $init_a\ (st\ s) = init_c\ s$
assumes *f*: $\bigwedge s\ p{::}'a\ ptr.$ *refines* $(f_c\ p)\ (f_a\ p)\ s\ (st\ s)$ (*rel-prod* (=) *rel-split-heap*)
assumes *typing-unchanged*: $\bigwedge s\ p{::}'a\ ptr.\ (f_c\ p) \bullet s\ ?\{\!|\lambda r\ t.$ *typing.unchanged-typing-on*

$\mathcal{S}$ s t⦘
shows *refines*
(*monolithic.with-fresh-stack-ptr* n $init_c$ f_c)
(*assume-with-fresh-stack-ptr* n $init_a$ f_a) s (*st* s)
(*rel-prod* (=) *rel-split-heap*)
unfolding *monolithic.with-fresh-stack-ptr-def assume-with-fresh-stack-ptr-def stack-ptr-acquire-def stack-ptr-release-def assume-stack-alloc-def*
apply (*rule refines-bind-bind-exn* [**where** Q= $\lambda(r,t)$ (r',t').
(*rel-prod* (=) *rel-split-heap*) (r,t) (r',t') $\wedge$
$(\exists\, p.\ r$ = *Result* $p \wedge (\forall\, i < n.$ *ptr-span* $(p +_p$ *int* $i) \subseteq \mathcal{S} \wedge$ *root-ptr-valid* (*heap-typing* t') $(p +_p$ *int* $i)))$])

subgoal
apply (*rule refines-assume-result-and-state'*)
using *sim-stack-alloc stack-byte-zero stack-allocs-*$\mathcal{S}$
apply (*clarsimp simp add*: *sim-set-def init rel-split-heap-def hrs-htd-update stack-allocs-root-ptr-valid-same*)
apply *blast*
done
apply *simp*
apply *simp*
apply *simp*
subgoal for p q t t'
apply (*rule refines-runs-to-partial-rel-prod-assume-on-exit* [**where** S'=*rel-split-heap* **and** P=*typing.unchanged-typing-on* $\mathcal{S}$ t])
apply (*subst* (*asm*) *rel-split-heap-def*)
apply *simp*
apply (*rule f*)
apply (*rule typing-unchanged*)
subgoal for s_c s_a t_c
apply *clarsimp*
apply (*clarsimp simp add*: *rel-split-heap-def*)
apply (*subst sim-stack-release*)
subgoal for *bs i*
using *typing.unchanged-typing-on-root-ptr-valid-preservation* [**where** S=$\mathcal{S}$ **and** s=t **and** t=s_c **and** p= $(p +_p$ *int* $i)$]
by *auto*
subgoal by *auto*
subgoal by *auto*
done
subgoal by (*simp add*: *Ex-list-of-length*)
subgoal for s_c s_a
apply *clarsimp*
apply (*clarsimp simp add*: *rel-split-heap-def*)
subgoal for *i*
using *typing.unchanged-typing-on-root-ptr-valid-preservation* [**where** S=$\mathcal{S}$ **and** s=t **and** t=s_c **and** p= $(p +_p$ *int* $i)$]
by *auto*
done

done
done

lemma *L2Tcorres-guard-with-fresh-stack-ptr* [*heap-abs*]:
assumes *rew*: *struct-rewrite-expr P* $init_c'$ $init_c$
assumes *grd*: *abs-guard st P' P*
assumes *expr*: *abs-expr st Q* $init_a$ $init_c'$
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge$*p*::*'a ptr*. *L2Tcorres st* (f_a *p*) (f_c *p*))
shows *L2Tcorres st* (*L2-seq* (*L2-guard* (λ*s*. *P' s* $\wedge$ *Q s*))
(λ-. (*guard-with-fresh-stack-ptr n* $init_a$ (*L2-VARS* f_a *nm*))))
(*monolithic.with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*))
apply (*rule refines-L2Tcorres*)
apply (*simp add*: *L2-seq-def L2-guard-def L2-VARS-def*)
apply (*rule refines-bind-guard-right*)
apply *clarsimp*
apply (*rule sim-guard-with-fresh-stack-ptr*)
subgoal for *s*
using *rew grd expr*
by (*auto simp add*: *struct-rewrite-expr-def abs-guard-def abs-expr-def*)
subgoal for *s s' p*
apply (*rule L2Tcorres-refines*)
apply (*rule f*)
done
done

lemma *L2Tcorres-with-fresh-stack-ptr*:
assumes *typing-unchanged*: $\bigwedge$*s p*::*'a ptr*. (f_c *p*) $\cdot$ *s ?*{$\!|$$\lambda$*r t*. *typing.unchanged-typing-on* $\mathcal{S}$ *s t*$|\!$}
assumes *rew*: *struct-rewrite-expr P* $init_c'$ $init_c$
assumes *grd*: *abs-guard st P' P*
assumes *expr*: *abs-expr st Q* $init_a$ $init_c'$
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge$*p*::*'a ptr*. *L2Tcorres st* (f_a *p*) (f_c *p*))
shows *L2Tcorres st* (*L2-seq* (*L2-guard* (λ*s*. *P' s* $\wedge$ *Q s*))
(λ-. (*with-fresh-stack-ptr n* $init_a$ (*L2-VARS* f_a *nm*))))
(*monolithic.with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*))
apply (*rule refines-L2Tcorres*)
apply (*simp add*: *L2-seq-def L2-guard-def L2-VARS-def*)
apply (*rule refines-bind-guard-right*)
apply *clarsimp*
apply (*rule sim-with-fresh-stack-ptr*)
subgoal for *s*
using *rew grd expr*
by (*auto simp add*: *struct-rewrite-expr-def abs-guard-def abs-expr-def*)
subgoal for *s s' p*
apply (*rule L2Tcorres-refines*)

apply (*rule f*)
done
using *typing-unchanged* **by** *blast*

lemma *L2Tcorres-assume-with-fresh-stack-ptr*[*heap-abs*]:
assumes *typing-unchanged*: $\bigwedge s\ p{::}'a\ ptr.\ (f_c\ p) \bullet s\ ?\{\!|\lambda r\ t.\ typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ t|\!\}$
assumes *rew*: *struct-rewrite-expr P* $init_c{}'$ $init_c$
assumes *grd*: *abs-guard st P' P*
assumes *expr*: *abs-expr st Q* $init_a$ $init_c{}'$
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge p{::}'a\ ptr.$ *L2Tcorres st* (f_a *p*) (f_c *p*))
shows *L2Tcorres st* (*L2-seq* (*L2-guard* ($\lambda s.$ *P' s* $\wedge$ *Q s*))
(λ-. (*assume-with-fresh-stack-ptr n* $init_a$ (*L2-VARS* f_a *nm*))))
(*monolithic.with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*))
apply (*rule refines-L2Tcorres*)
apply (*simp add*: *L2-seq-def L2-guard-def L2-VARS-def*)
apply (*rule refines-bind-guard-right*)
apply *clarsimp*
apply (*rule sim-assume-with-fresh-stack-ptr*)
subgoal for *s*
using *rew grd expr*
by (*auto simp add*: *struct-rewrite-expr-def abs-guard-def abs-expr-def*)
subgoal for *s s' p*
apply (*rule L2Tcorres-refines*)
apply (*rule f*)
done
using *typing-unchanged* **by** *blast*

lemma *unchanged-typing-commutes*: *typing.unchanged-typing-on S s t* $\Longrightarrow$ *unchanged-typing-on S* (*st s*) (*st t*)
using *heap-typing-commutes* [*of s*] *heap-typing-commutes* [*of t*]
by (*auto simp add*: *unchanged-typing-on-def typing.unchanged-typing-on-def*)
end

named-theorems *read-stack-byte-ZERO-base*
and *read-stack-byte-ZERO-step*
and *read-stack-byte-ZERO-step-subst*

lemma (**in** *open-types*) *ptr-span-with-stack-byte-type-implies-ptr-invalid*:
fixes *p* :: (*'a* :: {*mem-type*, *stack-type*}) *ptr*
assumes $*$: $\forall\ a \in$ *ptr-span p. root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
shows $\neg$ *ptr-valid-u* (*typ-uinfo-t TYPE*(*'a*)) *d* (*ptr-val p*)
by (*metis assms disjoint-iff fold-ptr-valid' in-ptr-span-itself ptr-exhaust-eq*
ptr-valid-stack-byte-disjoint)

lemma (**in** *open-types*)
ptr-span-with-stack-byte-type-implies-ZERO[*read-stack-byte-ZERO-base*]:
fixes $p :: ('a :: \{mem\text{-}type, stack\text{-}type\})\ ptr$
assumes $\forall\, a \in ptr\text{-}span\ p.\ root\text{-}ptr\text{-}valid\ (hrs\text{-}htd\ d)\ (PTR\ (stack\text{-}byte)\ a)$
shows $the\text{-}default\ (ZERO('a))\ (plift\ d\ p) = ZERO('a)$
using *ptr-span-with-stack-byte-type-implies-ptr-invalid*[*OF assms*]
by (*simp add*: *fold-ptr-valid' plift-None*)

lemma *ptr-span-array-ptr-index-subset-ptr-span*:
fixes $p :: (('a :: \{array\text{-}outer\text{-}max\text{-}size\})['b :: array\text{-}max\text{-}count])\ ptr$
assumes $i < CARD('b)$
shows $ptr\text{-}span\ (array\text{-}ptr\text{-}index\ p\ c\ i) \subseteq ptr\text{-}span\ p$
using *assms*
apply (*simp add*: *array-ptr-index-def ptr-add-def*)
apply (*rule intvl-sub-offset*)
apply (*rule order-trans*[*of - i * size-of TYPE('a) + size-of TYPE('a)*])
apply (*simp add*: *unat-le-helper*)
apply (*simp add*: *add.commute less-le-mult-nat*)
done

lemma *read-stack-byte-ZERO-array-intro*[*read-stack-byte-ZERO-step*]:
fixes $q :: ('a :: \{array\text{-}outer\text{-}max\text{-}size\}['b :: array\text{-}max\text{-}count])\ ptr$
assumes *ptr-span-has-stack-byte-type*:
$\forall\, a{\in}ptr\text{-}span\ q.\ root\text{-}ptr\text{-}valid\ d\ (PTR(stack\text{-}byte)\ a)$
assumes *subtype-reads-ZERO*:
$\bigwedge p :: 'a\ ptr.\ \forall\, a{\in}ptr\text{-}span\ p.\ root\text{-}ptr\text{-}valid\ d\ (PTR(stack\text{-}byte)\ a) \Longrightarrow r\ s\ p = ZERO('a)$
shows $(ARRAY\ i.\ r\ s\ (array\text{-}ptr\text{-}index\ q\ c\ i)) = ZERO('a['b])$
apply (*rule array-ext*)
apply (*simp add*: *array-index-zero*)
apply (*rule subtype-reads-ZERO*)
using *ptr-span-has-stack-byte-type ptr-span-array-ptr-index-subset-ptr-span* **by** *blast*

lemma *read-stack-byte-ZERO-array-2dim-intro*[*read-stack-byte-ZERO-step*]:
fixes $q :: ('a :: \{array\text{-}inner\text{-}max\text{-}size\}['b :: array\text{-}max\text{-}count]['c :: array\text{-}max\text{-}count])\ ptr$
assumes *ptr-span-has-stack-byte-type*:
$\forall\, a{\in}ptr\text{-}span\ q.\ root\text{-}ptr\text{-}valid\ d\ (PTR(stack\text{-}byte)\ a)$
assumes *subtype-reads-ZERO*:
$\bigwedge p :: 'a\ ptr.\ \forall\, a{\in}ptr\text{-}span\ p.\ root\text{-}ptr\text{-}valid\ d\ (PTR(stack\text{-}byte)\ a) \Longrightarrow r\ s\ p = ZERO('a)$
shows $(ARRAY\ i\ j.\ r\ s\ (array\text{-}ptr\text{-}index\ (array\text{-}ptr\text{-}index\ q\ c\ i)\ c\ j)) = ZERO('a['b]['c])$
apply (*rule array-ext*)
apply (*simp add*: *array-index-zero*)
apply (*rule array-ext*)
apply (*simp add*: *array-index-zero*)
apply (*rule subtype-reads-ZERO*)
by (*metis* (*no-types*, *opaque-lifting*) *subset-iff ptr-span-has-stack-byte-type*

ptr-span-array-ptr-index-subset-ptr-span)

lemma *read-stack-byte-ZERO-field-intro*[*read-stack-byte-ZERO-step*]:
fixes $q :: 'a :: mem\text{-}type\ ptr$
assumes *ptr-span-has-stack-byte-type*:
$\forall a{\in}ptr\text{-}span\ q.\ root\text{-}ptr\text{-}valid\ d\ (PTR(stack\text{-}byte)\ a)$
assumes *subtype-reads-ZERO*:
$\bigwedge p :: 'b :: mem\text{-}type\ ptr.\ \forall a{\in}ptr\text{-}span\ p.\ root\text{-}ptr\text{-}valid\ d\ (PTR(stack\text{-}byte)\ a) \Longrightarrow r\ s\ p = ZERO('b)$
assumes *subtype-lookup*:
$field\text{-}lookup\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ f\ 0 = Some\ (typ\text{-}uinfo\text{-}t\ TYPE('b),\ n)$
shows $r\ s\ (PTR('b)\ (\&(q{\rightarrow}f))) = ZERO('b)$
proof −
have $ptr\text{-}span\ (PTR('b)\ (\&(q{\rightarrow}f))) \subseteq ptr\text{-}span\ q$
using *TypHeapSimple.field-tag-sub'*[*OF subtype-lookup*]
by (*simp*, *metis size-of-fold*)
thus *?thesis*
using *ptr-span-has-stack-byte-type subtype-lookup subtype-reads-ZERO* **by** *blast*
qed

context *open-types*
begin

lemma *ptr-span-with-stack-byte-type-implies-read-dedicated-heap-ZERO*[*simp*]:
$\forall a{\in}ptr\text{-}span\ p.\ root\text{-}ptr\text{-}valid\ (hrs\text{-}htd\ s)\ (PTR(stack\text{-}byte)\ a) \Longrightarrow read\text{-}dedicated\text{-}heap\ s\ p = ZERO('a{::}\{stack\text{-}type,\ xmem\text{-}type\})$
unfolding *read-dedicated-heap-def ptr-span-with-stack-byte-type-implies-ZERO*[*of p*] *merge-addressable-fields.idem* **..**

lemma *write-simulationI*:
fixes $R :: 's \Rightarrow 'a{::}xmem\text{-}type\ ptr \Rightarrow 'a$
assumes *fs*: $map\text{-}of\ \mathcal{T}\ (typ\text{-}uinfo\text{-}t\ TYPE('a)) = Some\ fs$
assumes *heap-typing-simulation* $\mathcal{T}$ *t-hrs t-hrs-update heap-typing heap-typing-update l*
and *l-w*: $list\text{-}all2\ (\lambda f\ w.\ \forall t\ u\ n\ h\ (p{::}'a\ ptr)\ x.\ field\text{-}ti\ TYPE('a)\ f = Some\ t \longrightarrow field\text{-}lookup\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ f\ 0 = Some\ (u,\ n) \longrightarrow ptr\text{-}valid\text{-}u\ u\ (hrs\text{-}htd\ (t\text{-}hrs\ h))\ \&(p{\rightarrow}f) \longrightarrow l\ (t\text{-}hrs\text{-}update\ (hrs\text{-}mem\text{-}update\ (heap\text{-}upd\text{-}list\ (size\text{-}td\ u)\ \&(p{\rightarrow}f)\ (access\text{-}ti\ t\ x)))\ h) = w\ p\ x\ (l\ h))\ fs\ ws$
and *l-u*: $\bigwedge(p{::}'a\ ptr)\ (x{::}'a)\ (s{::}'b).\ ptr\text{-}valid\ (hrs\text{-}htd\ (t\text{-}hrs\ s))\ p \Longrightarrow l\ (t\text{-}hrs\text{-}update\ (write\text{-}dedicated\text{-}heap\ p\ x)\ s) = u\ (upd\text{-}fun\ p\ (\lambda old.\ merge\text{-}addressable\text{-}fields\ old\ x))\ (l\ s)$
assumes *V*:
$\bigwedge h\ p.\ V\ (l\ h)\ p \longleftrightarrow ptr\text{-}valid\ (hrs\text{-}htd\ (t\text{-}hrs\ h))\ p$
assumes *W*:

```
    ⋀p f h. W p f h =
      fold (λw. w p (f (R h p))) ws (u (upd-fun p (λold. merge-addressable-fields
old (f (R h p))))) h)
  shows write-simulation t-hrs t-hrs-update l V W
proof −
  interpret hrs: heap-typing-simulation 𝒯 t-hrs t-hrs-update heap-typing heap-typing-update
l
    by fact

  have valid:
    list-all (λf. ∀ u n. field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n) ⟶
      ptr-valid-u u h &(p→f)) fs
    if ∗: ptr-valid-u (typ-uinfo-t TYPE('a)) h (ptr-val p) for h and p :: 'a ptr
    using ptr-valid-u-step[OF fs - - ∗]
    by (auto simp: list-all-iff field-lvalue-def field-offset-def)

  have fold': l (fold
        (λxa. t-hrs-update
              (hrs-mem-update
                (heap-upd-list (size-td (the (field-ti TYPE('a) xa))) &(p→xa)
                  (access-ti (the (field-ti TYPE('a) xa)) x))))
        fs s) =
      fold (λw. w p x) ws (l s)
    if p: ptr-valid-u (typ-uinfo-t TYPE('a)) (hrs-htd (t-hrs s)) (ptr-val p)
    for p x s
    using l-w wf-𝒯[OF fs] p[THEN valid]
  proof (induction arbitrary: s)
    case (Cons f fs w ws)
    from Cons.prems obtain u n where f-u :field-lookup (typ-uinfo-t TYPE('a))
f 0 = Some (u, n)
      and [simp]: list-all (λf. ∃ a b. field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some
(a, b)) fs
      by auto
    from f-u[THEN field-lookup-uinfo-Some-rev] obtain k where
    field-lookup (typ-info-t TYPE('a)) f 0 = Some (k, n) and u-eq: u = export-uinfo
k
      by auto
    then have [simp]: field-ti TYPE('a) f = Some k by (simp add: field-ti-def)
    have [simp]: size-td k = size-td u
      by (simp add: u-eq)
    have [simp]: ptr-valid-u u (hrs-htd (t-hrs s)) &(p→f)
      using Cons.prems(2) f-u by auto
    show ?case
      using Cons.prems Cons.hyps by (simp add: Cons.IH f-u)
  qed simp

  have fold:
    l ((fold (t-hrs-update ∘
          (λ(f, u). hrs-mem-update (heap-upd-list (size-td u) &(p→f) (access-ti u
```

```
x))))
        (addressable-fields TYPE('a)) ∘
      t-hrs-update (write-dedicated-heap p x)) s) =
    fold (λw. w p x) ws (u (upd-fun p (λold. merge-addressable-fields old x)) (l s))
    if p: ptr-valid-u (typ-uinfo-t TYPE('a)) (hrs-htd (t-hrs s)) (ptr-val p)
    for p x s
    by (subst addressable-fields-def)
      (simp add: fs fold-map fold' p ptr-valid-def l-u cong: fold-cong)

  show ?thesis
    apply (rule hrs.write-simulation-alt)
    apply (simp add: V)
    apply (subst hrs-mem-update-heap-update')
    apply (subst W)
    apply (subst (asm) V)
    apply (subst (asm) ptr-valid-def)
    apply (subst hrs.t-hrs.upd-comp[symmetric])
    apply (subst hrs.t-hrs.upd-comp-fold)
    apply (subst fold)
    apply simp-all
    done
qed

end

locale stack-simulation-heap-typing =
   typ-heap-simulation st r w λt p. open-types.ptr-valid 𝒯 (heap-typing t) p t-hrs
t-hrs-update +
  heap-typing-simulation 𝒯 t-hrs t-hrs-update heap-typing heap-typing-upd st +
  typ-heap-typing r w heap-typing heap-typing-upd 𝒮
  for
    st:: 's ⇒ 't and
    r:: 't ⇒ ('a::{xmem-type, stack-type}) ptr ⇒ 'a and
    w:: 'a ptr ⇒ ('a ⇒ 'a) ⇒ 't ⇒ 't and
    t-hrs :: 's ⇒ heap-raw-state and
    t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's and
    heap-typing :: 't ⇒ heap-typ-desc and
    heap-typing-upd :: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 't ⇒ 't and
    𝒮:: addr set and
    𝒯:: (typ-uinfo * qualified-field-name list) list +

assumes sim-stack-alloc-heap-typing:
  ⋀p d s n.
    (p, d) ∈ stack-allocs n 𝒮 TYPE('a) (hrs-htd (t-hrs s)) ⟹
    st (t-hrs-update (hrs-mem-update (fold (λi. heap-update (p +_p int i) c-type-class.zero)
[0..<n]) ∘ hrs-htd-update (λ-. d)) s) =
      (heap-typing-upd (λ-. d) (st s))

assumes sim-stack-release-heap-typing:
```

$\bigwedge(p::'a\ ptr)\ s\ n.\ (\bigwedge i.\ i < n \Longrightarrow root\text{-}ptr\text{-}valid\ (hrs\text{-}htd\ (t\text{-}hrs\ s))\ (p +_p int\ i)) \Longrightarrow$
st (*t-hrs-update* (*hrs-htd-update* (*stack-releases n p*)) *s*) =
heap-typing-upd (*stack-releases n p*)
(*st* (*t-hrs-update* (*hrs-mem-update* (*fold* ($\lambda i.$ *heap-update* ($p +_p$ *int i*) *c-type-class.zero*) [*0..<n*])) *s*))

assumes *sim-stack-stack-byte-zero*[*read-stack-byte-ZERO-step*]:
$\bigwedge p\ s.\ \forall\ a \in$ *ptr-span p. root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*PTR*(*stack-byte*) *a*) $\Longrightarrow$
r (*st s*) *p* = *ZERO*($'a$)

begin

lemma *fold-heap-update-simulation*:
assumes *valid*: $\bigwedge i.\ i < n \Longrightarrow$ *ptr-valid* (*heap-typing* (*st s*)) ($p +_p$ *int i*)
shows *st* (*t-hrs-update* (*hrs-mem-update* (*fold* ($\lambda i.$ *heap-update* ($p +_p$ *int i*) (*vs i*)) [*0..<n*])) *s*) =
fold ($\lambda i.$ *w* ($p +_p$ *int i*) (λ-. *vs i*)) [*0..<n*] (*st s*)
using *valid*
proof (*induct n arbitrary*: *vs s*)
case *0*
then show *?case*
by (*simp add*: *case-prod-unfold hrs-mem-update-def*)
next
case (*Suc n*)
from *Suc.prems* **obtain**
valid: $\bigwedge i.\ i < Suc\ n \Longrightarrow$ *ptr-valid* (*heap-typing* (*st s*)) ($p +_p$ *int i*) **by** *blast*

from *valid* **have** *valid'*: $\bigwedge i.\ i < n \Longrightarrow$ *ptr-valid* (*heap-typing* (*st s*)) ($p +_p$ *int i*)
by *auto*
note *hyp* = *Suc.hyps* [*OF valid'*]
show *?case*
apply (*simp add*: *hyp* [*symmetric*])
apply (*subst write-commutes* [*symmetric*])
using *valid*
apply *simp*
using *TypHeapSimple.hrs-mem-update-comp hrs-mem-update-def*
apply *simp*
done
qed

lemma *fold-heap-update-padding-simulation*:
assumes *valid*: $\bigwedge i.\ i < n \Longrightarrow$ *ptr-valid* (*heap-typing* (*st s*)) ($p +_p$ *int i*)
assumes *lbs*: *length bs* = *n* $*$ *size-of TYPE*($'a$)
shows *st* (*t-hrs-update* (*hrs-mem-update* (*fold* ($\lambda i.$ *heap-update-padding* ($p +_p$ *int i*) (*vs i*) (*take* (*size-of TYPE*($'a$)) (*drop* (*i* $*$ *size-of TYPE*($'a$)) *bs*))) [*0..<n*])) *s*) =
fold ($\lambda i.$ *w* ($p +_p$ *int i*) (λ-. *vs i*)) [*0..<n*] (*st s*)
using *valid lbs*
proof (*induct n arbitrary*: *bs vs s*)

```
  case 0
  then show ?case
    by (simp add: case-prod-unfold hrs-mem-update-def)
next
  case (Suc n)
  from Suc.prems obtain
    valid: ⋀i. i < Suc n ⟹ ptr-valid (heap-typing (st s)) (p +p int i) and
    lbs': length (take (n * (size-of TYPE('a))) bs) = n * size-of TYPE('a)
    by auto

  from valid have valid': ⋀i. i < n ⟹ ptr-valid (heap-typing (st s)) (p +p int i)
by auto
  note hyp = Suc.hyps [OF valid' lbs']
  have take-eq: ⋀i. i < n ⟹ take (size-of TYPE('a)) (drop (i * size-of TYPE('a))
(take (n * size-of TYPE('a)) bs)) =
        take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs)
    by (metis Groups.mult-ac(2) mult-less-cancel1 not-less not-less-eq
        order-less-imp-le take-all take-drop-take times-nat.simps(2))

  have fold-eq: ⋀h. fold
             (λi. heap-update-padding (p +p int i) (vs i)
                    (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) (take (n
* size-of TYPE('a)) bs))))
             [0..<n] h =
           fold
             (λi. heap-update-padding (p +p int i) (vs i)
                    (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs)))
               [0..<n] h

    apply (rule fold-cong)
      apply (rule refl)
     apply (rule refl)
    using take-eq
    apply simp
    done


  show ?case
    apply (simp add: hyp [symmetric])
     apply (subst write-padding-commutes [symmetric, where bs = take (size-of
TYPE('a)) (drop (n * size-of TYPE('a)) bs)])
    subgoal using valid
      by simp
    subgoal using Suc.prems by simp
    subgoal
    using TypHeapSimple.hrs-mem-update-comp hrs-mem-update-def
      by (simp add: fold-eq)
    done
qed
```

```
lemma sim-stack-alloc':
  assumes alloc: (p, d) ∈ stack-allocs n 𝒮 TYPE('a) (hrs-htd (t-hrs s))
  assumes len: length vs = n
  assumes lbs: length bs = n * size-of TYPE('a)
  shows st (t-hrs-update (hrs-mem-update (fold (λi. heap-update-padding (p +_p
int i) (vs!i) (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<n])
∘ hrs-htd-update (λ-. d)) s) =
          (fold (λi. w (p +_p int i) (λ-. (vs ! i))) [0..<n]) (heap-typing-upd (λ-. d)
(st s))
proof −
  {
    fix i
    assume i-bound: i < n
    have ptr-valid (heap-typing (st (t-hrs-update
              (hrs-mem-update (fold (λi. heap-update (p +_p int i) c-type-class.zero)
[0..<n]) ∘
                hrs-htd-update (λ-. d))
               s)))
         (p +_p int i)
    proof −
      from stack-allocs-cases [OF alloc] i-bound
      have ptr-valid: ptr-valid d (p +_p int i)
        using root-ptr-valid-ptr-valid by auto
      thus ?thesis
        using heap-typing-upd-commutes by (simp, metis)
    qed
  } note valids = this

  from stack-allocs-cases [OF alloc] obtain
    bound: unat (ptr-val p) + n * size-of TYPE('a) ≤ addr-card and
    not-null: ptr-val p ≠ 0
    by (metis Ptr-ptr-val c-guard-NULL-simp)

  show ?thesis
    apply (simp add: sim-stack-alloc-heap-typing [OF alloc, symmetric])
    apply (subst fold-heap-update-padding-simulation [OF valids lbs, symmetric])
      apply (simp)
     apply (simp add: len)
    apply (simp add: comp-def hrs-mem-update-comp')
   apply (subst fold-heap-update-padding-heap-update-collapse [OF bound not-null])
    using lbs
     apply (auto simp add: less-le-mult-nat nat-move-sub-le)
    done
qed

lemma sim-stack-release':
  fixes p :: 'a ptr
  assumes roots: ⋀i. i < n ⟹ root-ptr-valid (hrs-htd (t-hrs s)) (p +_p int i)
```

shows *st* (*t-hrs-update* (*hrs-htd-update* (*stack-releases n p*)) *s*) =
((*heap-typing-upd* (*stack-releases n p*) ((*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *c-type-class.zero*)) [*0*..<*n*]) (*st s*))))
proof −
from *roots root-ptr-valid-ptr-valid heap-typing-commutes*
have *valids*: ⋀*i* . *i* < *n* ⟹ *ptr-valid* (*heap-typing* (*st s*)) (*p* +$_p$ *int i*)
by *metis*
note *commutes* = *fold-heap-update-simulation* [*OF valids*, *symmetric*, *of n*, *simplified*]
show *?thesis*
apply (*simp add*: *commutes*)
apply (*simp add*: *sim-stack-release-heap-typing* [*OF roots*])
done
qed

lemma *sim-stack-release″*:
fixes *p* :: *′a ptr*
assumes *roots*: ⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*p* +$_p$ *int i*)
assumes *lbs*: *length bs* = *n* ∗ *size-of TYPE*(*′a*)
shows *st* (*t-hrs-update* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) *o hrs-htd-update* (*stack-releases n p*)) *s*) =
((*heap-typing-upd* (*stack-releases n p*) ((*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *c-type-class.zero*)) [*0*..<*n*]) (*st s*))))
proof −
{
fix *i*
assume *bound-i*: *i* < *length bs*
have *root-ptr-valid* (*hrs-htd* (*t-hrs* (*t-hrs-update* (*hrs-htd-update* (*stack-releases n p*)) *s*)))
((*PTR-COERCE*(*′a* → *stack-byte*) *p*) +$_p$ *int i*)
proof −
have *span*: *ptr-val* (((*PTR-COERCE*(*′a* → *stack-byte*) *p*) +$_p$ *int i*)) ∈ {*ptr-val p*..+*n* ∗ *size-of TYPE*(*′a*)}
using *lbs bound-i intvlI* **by** (*auto simp add*: *ptr-add-def*)
from *roots* **have** ∀ *i*<*n*. *c-guard* (*p* +$_p$ *int i*)
using *root-ptr-valid-c-guard* **by** *blast*
from *stack-releases-root-ptr-valid-footprint* [*OF span this*]
show *?thesis*
using *typing.get-upd* **by** *force*
qed
} note *sb* = *this*

show *?thesis*
apply (*simp add*: *lift-heap-update-list-stack-byte-independent* [*OF sb*, *simplified*])
apply (*simp add*: *sim-stack-release′* [*OF roots*])
done
qed

lemma *stack-byte-zero'*:
assumes $(p,\ d) \in$ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*hrs-htd* (*t-hrs s*))
assumes $i < n$
shows *r* (*st s*) ($p +_p$ *int i*) = *ZERO*('*a*)
by (*rule sim-stack-stack-byte-zero*)
(*meson assms stack-allocs-cases stack-allocs-contained subsetD*)

sublocale *stack-simulation*
using *sim-stack-alloc' sim-stack-release'' stack-byte-zero'*
by (*unfold-locales*) *auto*

sublocale *typ-heap-typing-stack-simulation* $\mathcal{T}$ *st r w* $\lambda t\ p.$ *open-types.ptr-valid* $\mathcal{T}$ (*heap-typing t*) *p t-hrs t-hrs-update heap-typing heap-typing-upd* $\mathcal{S}$
by (*unfold-locales*)

end

definition
valid-struct-field
:: *string list*
$\Rightarrow$ (('*p*::*xmem-type*) $\Rightarrow$ ('*f*::*xmem-type*))
$\Rightarrow$ (('*f* $\Rightarrow$ '*f*) $\Rightarrow$ ('*p* $\Rightarrow$ '*p*))
$\Rightarrow$ ('*s* $\Rightarrow$ *heap-raw-state*)
$\Rightarrow$ ((*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ '*s* $\Rightarrow$ '*s*)
$\Rightarrow$ *bool*
where
valid-struct-field field-name field-getter field-setter t-hrs t-hrs-update $\equiv$
(*lense field-getter field-setter*
$\wedge$ *field-ti TYPE*('*p*) *field-name* =
Some (*adjust-ti* (*typ-info-t TYPE*('*f*)) *field-getter* (*field-setter* $\circ$ (λx -. *x*)))
$\wedge$ ($\forall$ *p* :: '*p ptr*. *c-guard p* $\longrightarrow$ *c-guard* (*Ptr* &(*p*$\rightarrow$*field-name*) :: '*f ptr*))
$\wedge$ *lense t-hrs t-hrs-update*)

lemma *typ-heap-simulation-get-hvalD*:
$[\![$ *typ-heap-simulation st r w v*
t-hrs t-hrs-update; *v* (*st s*) *p* $]\!] \Longrightarrow$
h-val (*hrs-mem* (*t-hrs s*)) *p* = *r* (*st s*) *p*
by (*clarsimp simp*: *read-simulation.read-commutes* [*OF typ-heap-simulation.axioms*(*2*)])

lemma *valid-struct-fieldI* [*intro*]:
fixes *field-getter* :: ('*a*::*xmem-type*) ⇒ ('*f*::*xmem-type*)
shows ⟦
⋀*s f*. *f* (*field-getter s*) = (*field-getter s*) ⟹ *field-setter f s* = *s*;
⋀*s f f'*. *f* (*field-getter s*) = *f'* (*field-getter s*) ⟹ *field-setter f s* = *field-setter f' s*;
⋀*s f*. *field-getter* (*field-setter f s*) = *f* (*field-getter s*);
⋀*s f g*. *field-setter f* (*field-setter g s*) = *field-setter* (*f* ∘ *g*) *s*;
field-ti TYPE('*a*) *field-name* =
Some (*adjust-ti* (*typ-info-t TYPE*('*f*)) *field-getter* (*field-setter* ∘ (λ*x* -. *x*)));
⋀(*p*::'*a ptr*). *c-guard p* ⟹ *c-guard* (*Ptr* &(*p*→*field-name*) :: '*f ptr*);
⋀*s x*. *t-hrs* (*t-hrs-update x s*) = *x* (*t-hrs s*);
⋀*s f*. *f* (*t-hrs s*) = *t-hrs s* ⟹ *t-hrs-update f s* = *s*;
⋀*s f f'*. *f* (*t-hrs s*) = *f'* (*t-hrs s*) ⟹ *t-hrs-update f s* = *t-hrs-update f' s*;
⋀*s f g*. *t-hrs-update f* (*t-hrs-update g s*) = *t-hrs-update* (λ*x*. *f* (*g x*)) *s*
⟧ ⟹
valid-struct-field field-name field-getter field-setter t-hrs t-hrs-update
apply (*unfold valid-struct-field-def lense-def o-def*)
apply (*safe* | *assumption* | *rule ext*)+
done

lemma *typ-heap-simulation-t-hrs-updateD*:
⟦ *typ-heap-simulation st r w v*
t-hrs t-hrs-update; *v* (*st s*) *p* ⟧ ⟹
st (*t-hrs-update* (*hrs-mem-update* (*heap-update p v'*)) *s*) =
w p (λ*x*. *v'*) (*st s*)
by (*clarsimp simp*: *write-simulation.write-commutes* [*OF typ-heap-simulation.axioms*(*3*)])

lemma *heap-abs-expr-guard* [*heap-abs*]:
⟦ *typ-heap-simulation st getter setter vgetter t-hrs t-hrs-update*;
abs-expr st P x' x ⟧ ⟹
abs-guard st (λ*s*. *P s* ∧ *vgetter s* (*x' s*)) (λ*s*. (*c-guard* (*x s* :: ('*a*::*xmem-type*) *ptr*)))
apply (*clarsimp simp*: *abs-expr-def abs-guard-def*
simple-lift-def root-ptr-valid-def
valid-implies-cguard.valid-implies-c-guard [*OF typ-heap-simulation.axioms*(*5*)])
done

lemma *heap-abs-expr-h-val* [*heap-abs*]:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
abs-expr st P x' x ⟧ ⟹
abs-expr st
(λ*s*. *P s* ∧ *v s* (*x' s*))
(λ*s*. (*r s* (*x' s*)))
(λ*s*. (*h-val* (*hrs-mem* (*t-hrs s*))) (*x s*))
apply (*clarsimp simp*: *abs-expr-def simple-lift-def*)
apply (*metis typ-heap-simulation-get-hvalD*)
done

lemma *heap-abs-modifies-heap-update--unused*:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
abs-expr st Pb b′ b;
abs-expr st Pc c′ c ⟧ ⟹
abs-modifies st (λ*s*. *Pb s* ∧ *Pc s* ∧ *v s* (*b′ s*))
(λ*s*. *w* (*b′ s*) (λ*x*. (*c′ s*)) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*b s* :: (′*a*::*xmem-type*) *ptr*)
(*c s*))) *s*)
apply (*clarsimp simp*: *typ-simple-heap-simps abs-expr-def abs-modifies-def*)
apply (*metis typ-heap-simulation-t-hrs-updateD*)
done

definition *heap-lift--h-val* ≡ *h-val*

lemma *heap-abs-modifies-heap-update* [*heap-abs*]:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
abs-expr st Pb b′ b;
⋀*v*. *abs-expr st Pc* (*c′ v*) (*c v*) ⟧ ⟹
abs-modifies st (λ*s*. *Pb s* ∧ *Pc s* ∧ *v s* (*b′ s*))
(λ*s*. *w* (*b′ s*) (λ-. (*c′* (*r s* (*b′ s*)) *s*)) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update*
(*heap-update* (*b s* :: (′*a*::*xmem-type*) *ptr*)
(*c* (*heap-lift--h-val* (*hrs-mem* (*t-hrs s*)) (*b s*)) *s*))) *s*)
apply (*clarsimp simp*: *typ-simple-heap-simps abs-expr-def abs-modifies-def heap-lift--h-val-def*)
subgoal for *s*
apply (*rule subst*[**where** *t* = *h-val* (*hrs-mem* (*t-hrs s*)) (*b′* (*st s*))
and *s* = *r* (*st s*) (*b′* (*st s*))])
apply (*clarsimp simp*: *read-simulation.read-commutes* [*OF typ-heap-simulation.axioms*(*2*)])
apply (*simp add*: *write-simulation.write-commutes* [*OF typ-heap-simulation.axioms*(*3*)])
done
done

lemma *struct-rewrite-guard-expr* [*heap-abs*]:
struct-rewrite-expr P a′ a ⟹ *struct-rewrite-guard* (λ*s*. *P s* ∧ *a′ s*) *a*
by (*simp add*: *struct-rewrite-expr-def struct-rewrite-guard-def*)

lemma *struct-rewrite-guard-constant* [*heap-abs*]:
struct-rewrite-guard (λ-. *P*) (λ-. *P*)
by (*simp add*: *struct-rewrite-guard-def*)

lemma *struct-rewrite-guard-conj* [*heap-abs*]:

⟦ *struct-rewrite-guard b′ b*; *struct-rewrite-guard a′ a* ⟧ ⟹
struct-rewrite-guard (λ*s*. *a′ s* ∧ *b′ s*) (λ*s*. *a s* ∧ *b s*)
by (*clarsimp simp*: *struct-rewrite-guard-def*)

lemma *struct-rewrite-guard-split* [*heap-abs*]:
⟦ ⋀*a b*. *struct-rewrite-guard* (*A a b*) (*C a b*) ⟧
⟹ *struct-rewrite-guard* (*case r of* (*a*, *b*) ⇒ *A a b*) (*case r of* (*a*, *b*) ⇒ *C a b*)
apply (*auto simp*: *split-def*)
done

lemma *struct-rewrite-guard-c-guard-field* [*heap-abs*]:
⟦ *valid-struct-field field-name* (*field-getter* :: (*′a* :: *xmem-type*) ⇒ (*′f* :: *xmem-type*)) *field-setter t-hrs t-hrs-update*;
struct-rewrite-expr P p′ p;
struct-rewrite-guard Q (λ*s*. *c-guard* (*p′ s*)) ⟧ ⟹
struct-rewrite-guard (λ*s*. *P s* ∧ *Q s*)
(λ*s*. *c-guard* (*Ptr* (*field-lvalue* (*p s* :: *′a ptr*) *field-name*) :: *′f ptr*))
by (*simp add*: *valid-struct-field-def struct-rewrite-expr-def struct-rewrite-guard-def*)

lemma *align-of-array*: *align-of TYPE*((*′a* :: *array-outer-max-size*)[*′b* :: *array-max-count*]) = *align-of TYPE*(*′a*)
by (*simp add*: *align-of-def align-td-array*)

lemma *c-guard-array*:
⟦ *0* ≤ *k*; *nat k* < *CARD*(*′b*); *c-guard* (*p* :: ((*′a*::*array-outer-max-size*)[*′b*::*array-max-count*]) *ptr*) ⟧
⟹ *c-guard* (*ptr-coerce p* +*p k* :: *′a ptr*)
apply (*clarsimp simp*: *CTypesDefs.ptr-add-def c-guard-def c-null-guard-def*)
apply (*rule conjI*[*rotated*])
apply (*erule contrapos-nn*)
apply (*clarsimp simp*: *intvl-def*)
subgoal for *i*
apply (*rule exI*[**where** *x* = *nat k* ∗ *size-of TYPE*(*′a*) + *i*])
apply *clarsimp*
apply (*rule conjI*)
apply (*simp add*: *field-simps*)
apply (*rule less-le-trans*[**where** *y* = *Suc* (*nat k*) ∗ *size-of TYPE*(*′a*)])
apply *simp*
apply (*metis less-eq-Suc-le mult-le-mono2 mult.commute*)
done
apply (*subgoal-tac ptr-aligned* (*ptr-coerce p* :: *′a ptr*))
apply (*frule-tac p* = *ptr-coerce p* **and** *i* = *k* **in** *ptr-aligned-plus*)
apply (*clarsimp simp*: *CTypesDefs.ptr-add-def*)
apply (*clarsimp simp*: *ptr-aligned-def align-of-array*)
done

lemma *struct-rewrite-guard-c-guard-Array-field* [*heap-abs*]:

⟦ *valid-struct-field field-name* (*field-getter* :: ('*a* :: *xmem-type*) ⇒ ('*f*::*array-outer-max-size* ['*n*::*array-max-count*])) *field-setter t-hrs t-hrs-update*;
struct-rewrite-expr P p' p;
struct-rewrite-guard Q (λ*s*. *c-guard* (*p' s*)) ⟧ ⟹
struct-rewrite-guard (λ*s*. *P s* ∧ *Q s* ∧ *0* ≤ *k* ∧ *nat k* < *CARD*('*n*))
(λ*s*. *c-guard* (*ptr-coerce* (*Ptr* (*field-lvalue* (*p s* :: '*a ptr*) *field-name*) :: (('*f*['*n*]) *ptr*)) +$_p$ *k* :: '*f ptr*))
by (*simp del*: *ptr-coerce.simps add*: *valid-struct-field-def struct-rewrite-expr-def struct-rewrite-guard-def c-guard-array*)

lemma *struct-rewrite-expr-id*:
struct-rewrite-expr (λ-. *True*) *A A*
by (*simp add*: *struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-fun-app2* [*heap-abs-fo*]:
⟦ *struct-rewrite-expr P f f'*;
struct-rewrite-expr Q g g' ⟧ ⟹
struct-rewrite-expr (λ*s*. *P s* ∧ *Q s*) (λ*s a*. *f s a* (*g s a*)) (λ*s a*. *f' s a* $ *g' s a*)
by (*simp add*: *struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-fun-app* [*heap-abs-fo*]:
⟦ *struct-rewrite-expr Y x x'*; *struct-rewrite-expr X f f'* ⟧ ⟹
struct-rewrite-expr (λ*s*. *X s* ∧ *Y s*) (λ*s*. *f s* (*x s*)) (λ*s*. *f' s* $ *x' s*)
by (*clarsimp simp*: *struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-constant* [*heap-abs*]:
struct-rewrite-expr (λ-. *True*) (λ-. *a*) (λ-. *a*)
by (*clarsimp simp*: *struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-lambda-null* [*heap-abs*]:
struct-rewrite-expr P A C ⟹ *struct-rewrite-expr P* (λ*s* -. *A s*) (λ*s* -. *C s*)
by (*clarsimp simp*: *struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-split* [*heap-abs*]:
⟦ ⋀*a b*. *struct-rewrite-expr* (*P a b*) (*A a b*) (*C a b*) ⟧
⟹ *struct-rewrite-expr* (*case r of* (*a*, *b*) ⇒ *P a b*)
(*case r of* (*a*, *b*) ⇒ *A a b*) (*case r of* (*a*, *b*) ⇒ *C a b*)
apply (*auto simp*: *split-def*)
done

lemma *struct-rewrite-expr-basecase-h-val* [*heap-abs*]:
struct-rewrite-expr (λ-. *True*) (λ*s*. *h-val* (*h s*) (*p s*)) (λ*s*. *h-val* (*h s*) (*p s*))
by (*simp add*: *struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-indirect-h-val* [*heap-abs*]:
struct-rewrite-expr P a c ⟹
struct-rewrite-expr P (λ*s. h-val* (*h s*) (*a s*)) (λ*s. h-val* (*h s*) (*c s*))
by (*simp add*: *struct-rewrite-expr-def*)

lemma *struct-rewrite-expr-field* [*heap-abs*]:
⟦ *valid-struct-field field-name* (*field-getter* :: (*'a* :: *xmem-type*) ⇒ (*'f* :: *xmem-type*)) *field-setter t-hrs t-hrs-update*;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q a (λ*s. h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*)) ⟧
⟹ *struct-rewrite-expr* (λ*s. P s* ∧ *Q s*) (λ*s. field-getter* (*a s*))
(λ*s. h-val* (*hrs-mem* (*t-hrs s*)) (*Ptr* (*field-lvalue* (*p s*) *field-name*)))
apply (*clarsimp simp*: *valid-struct-field-def struct-rewrite-expr-def*)
apply (*subst h-val-field-from-bytes'*)
apply *assumption*
apply (*rule export-tag-adjust-ti*(*1*)[*rule-format*])
apply (*simp add*: *lense.fg-cons*)
apply *simp*
apply *simp*
done

lemma *abs-expr-field* [*heap-abs*]:
⟦ *valid-struct-field field-name* (*field-getter* :: (*'a* :: *xmem-type*) ⇒ (*'f* :: *xmem-type*)) *field-setter t-hrs t-hrs-update*;
abs-expr st P a c⟧
⟹ *abs-expr st P* (λ*s. field-getter* (*a s*)) (λ*s. field-getter* (*c s*))
by (*clarsimp simp add*: *valid-struct-field-def abs-expr-def*)

lemma *struct-rewrite-expr-Array-field* [*heap-abs*]:
⟦ *valid-struct-field field-name*
(*field-getter* :: (*'a* :: *xmem-type*) ⇒ *'f*::*array-outer-max-size* [*'n*::*array-max-count*])
field-setter t-hrs t-hrs-update;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q a (λ*s. h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*)) ⟧
⟹ *struct-rewrite-expr* (λ*s. P s* ∧ *Q s* ∧ *k* ≥ *0* ∧ *nat k* < *CARD*(*'n*))
(λ*s. index* (*field-getter* (*a s*)) (*nat k*))
(λ*s. h-val* (*hrs-mem* (*t-hrs s*))
(*ptr-coerce* (*Ptr* (*field-lvalue* (*p s*) *field-name*) :: (*'f*[*'n*]) *ptr*) +$_p$ *k*))
apply (*cases k*)
apply (*clarsimp simp*: *struct-rewrite-expr-def simp del*: *ptr-coerce.simps*)
subgoal for *n s*
apply (*subst struct-rewrite-expr-field*
[*unfolded struct-rewrite-expr-def*, *simplified*, *rule-format*, *symmetric*,
where *field-getter* = *field-getter* **and** *P* = *P* **and** *Q* = *Q* **and** *p* = *p* **and** *p'* = *p'*])
apply *assumption*
apply *simp*

apply *simp*
apply *simp*
apply (*rule subst*[**where** $s = p\ s$ **and** $t = p'\ s$])
apply *simp*
apply (*rule heap-access-Array-element*[*symmetric*])
apply *simp*
done
apply (*simp add*: *struct-rewrite-expr-def*)
done
declare *struct-rewrite-expr-Array-field* [*unfolded ptr-coerce.simps*, *heap-abs*]

lemma *struct-rewrite-modifies-id* [*heap-abs*]:
struct-rewrite-modifies (λ-. *True*) *A A*
by (*simp add*: *struct-rewrite-modifies-def*)

lemma *struct-rewrite-modifies-basecase* [*heap-abs*]:
⟦ *typ-heap-simulation st* (*getter* :: *'s* ⇒ *'a ptr* ⇒ (*'a*::*xmem-type*)) *setter vgetter t-hrs t-hrs-update*;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q v' v ⟧ ⟹
struct-rewrite-modifies (λ*s*. *P s* ∧ *Q s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*) (*v' s* :: *'a*))) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p s*) (*v s* :: *'a*))) *s*)
by (*simp add*: *struct-rewrite-expr-def struct-rewrite-modifies-def*)

lemma *heap-update-field-unpacked*:
⟦ *field-ti TYPE*(*'a*::*mem-type*) *f* = *Some* (*t* :: *'a xtyp-info*);
c-guard (*p* :: *'a*::*mem-type ptr*);
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*(*'b*::*mem-type*)) ⟧ ⟹
heap-update (*Ptr* &(*p*→*f*) :: *'b ptr*) *v hp* =
heap-update p (*update-ti t* (*to-bytes-p v*) (*h-val hp p*)) *hp*
oops

lemma *heap-update-Array-element-unpacked*:
n < *CARD*(*'b*::*array-max-count*) ⟹
heap-update (*ptr-coerce p'* +$_p$ *int n*) *w hp* =
heap-update (*p'*::(*'a*::*array-outer-max-size*[*'b*::*array-max-count*]) *ptr*)
(*Arrays.update* (*h-val hp p'*) *n w*) *hp*
oops

lemma *read-write-valid-hrs-mem*:
lense hrs-mem hrs-mem-update
by (*clarsimp simp*: *hrs-mem-def hrs-mem-update-def lense-def*)

definition *heap-lift--wrap-h-val* $\equiv (=)$

lemma *heap-lift-wrap-h-val* [*heap-abs*]:
heap-lift--wrap-h-val (*heap-lift--h-val s p*) (*h-val s p*)
by (*simp add*: *heap-lift--h-val-def heap-lift--wrap-h-val-def*)

lemma *heap-lift-wrap-h-val-skip* [*heap-abs*]:
heap-lift--wrap-h-val (*h-val s* (*Ptr* (*field-lvalue p f*))) (*h-val s* (*Ptr* (*field-lvalue p f*)))
by (*simp add*: *heap-lift--wrap-h-val-def*)

lemma *heap-lift-wrap-h-val-skip-array* [*heap-abs*]:
heap-lift--wrap-h-val (*h-val s* (*ptr-coerce p* $+_p$ *k*))
(*h-val s* (*ptr-coerce p* $+_p$ *k*))
by (*simp add*: *heap-lift--wrap-h-val-def*)

lemma *struct-rewrite-modifies-field--unused*:
⟦ *valid-struct-field field-name* (*field-getter* :: (*'a*::*xmem-type*) ⇒ (*'f*::*xmem-type*)) *field-setter t-hrs t-hrs-update*;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q f' f;
struct-rewrite-modifies R
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p''* *s*)
(*u s* (*field-setter* (λ-. *f' s*))))) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*)
(*field-setter* (λ-. *f' s*) (*h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*))))) *s*);
struct-rewrite-guard S (λ*s*. *c-guard* (*p' s*)) ⟧ ⟹
struct-rewrite-modifies (λ*s*. *P s* ∧ *Q s* ∧ *R s* ∧ *S s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p''* *s*)
(*u s* (*field-setter* (λ-. *f' s*))))) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*Ptr* (*field-lvalue* (*p s*) *field-name*))
(*f s*))) *s*)
apply (*clarsimp simp*: *struct-rewrite-expr-def struct-rewrite-guard-def struct-rewrite-modifies-def valid-struct-field-def*)
apply (*erule-tac x* = *s* **in** *allE*)+
apply (*erule impE*, *assumption*)+
apply (*erule-tac t* = *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p''* *s*)
(*u s* (*field-setter* (λ-. *f' s*))))) *s*
and *s* = *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*)
(*field-setter* (λ-. *f' s*) (*h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*))))) *s*
in *subst*)
apply (*rule lense.upd-cong*[**where** *get* = *t-hrs* **and** *upd* = *t-hrs-update*])
apply *assumption*

apply (*rule lense.upd-cong*[*OF read-write-valid-hrs-mem*])
apply (*subst heap-update-field-root-conv'''*)
apply *assumption*+
apply (*simp add*: *typ-uinfo-t-def lense.fg-cons*)
apply (*subst update-ti-update-ti-t*)
apply (*simp add*: *size-of-def*)
apply (*subst update-ti-s-adjust-ti-to-bytes-p*)
apply (*erule lense.fg-cons*)
apply *simp*
done

lemma *struct-rewrite-modifies-Array-field--unused*:
⟦ *valid-struct-field field-name* (*field-getter* :: ('*a*::*xmem-type*) ⇒ (('*f*::*array-outer-max-size*)['*n*::*array-max-coun*
field-setter t-hrs t-hrs-update;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q f' f;
struct-rewrite-modifies R
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p''* *s*)
(*u s* (*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s*)))))) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*)
(*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s*))
(*h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*))))) *s*);
struct-rewrite-guard S (λ*s*. *c-guard* (*p' s*)) ⟧ ⟹
struct-rewrite-modifies (λ*s*. *P s* ∧ *Q s* ∧ *R s* ∧ *S s* ∧ 0 ≤ *k* ∧ *nat k* < *CARD*('*n*))
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p''* *s*)
(*u s* (*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s*)))))) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update*
(*ptr-coerce* (*Ptr* (*field-lvalue* (*p s*) *field-name*) :: ('*f*['*n*]) *ptr*) +$_p$ *k*) (*f s*)))
s)
using *ptr-coerce.simps* [*simp del*]
apply (*clarsimp simp*: *struct-rewrite-expr-def struct-rewrite-guard-def struct-rewrite-modifies-def*
valid-struct-field-def)
subgoal for *s*
apply (*erule-tac x* = *s* **in** *allE*)+
apply (*erule impE*, *assumption*)+
apply (*erule-tac t* = *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p''* *s*)
(*u s*(*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s*)))))) *s*
and *s* = *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*)
(*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s*))
(*h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*))))) *s*
in *subst*)
apply (*rule lense.upd-cong*[**where** *get* = *t-hrs* **and** *upd* = *t-hrs-update*])
apply *assumption*
apply (*rule lense.upd-cong*[*OF read-write-valid-hrs-mem*])
apply (*cases k*, *clarsimp*)
apply (*subst heap-update-array-element*[*symmetric*])
apply *assumption*
apply *simp*
apply (*subst heap-update-field-root-conv'''*)

```
    apply assumption+
   apply (simp add: typ-uinfo-t-def lense.fg-cons)
  apply (subst h-val-field-from-bytes')
   apply assumption+
  apply (simp add: lense.fg-cons)
 apply clarsimp
 apply (subst update-ti-update-ti-t)
  apply (simp add: size-of-def)
 apply (subst update-ti-s-adjust-ti-to-bytes-p)
  apply (erule lense.fg-cons)
 apply clarsimp
 apply (subst lense.upd-cong[of field-getter field-setter])
  apply auto
 done
done
```

```
lemma struct-rewrite-modifies-field [heap-abs]:
  ⟦ valid-struct-field field-name (field-getter :: ('a::xmem-type) ⇒ ('f::xmem-type))
field-setter t-hrs t-hrs-update;
    struct-rewrite-expr P p' p;
    struct-rewrite-expr Q f' f;
    ⋀s. heap-lift--wrap-h-val (h-val-p' s) (h-val (hrs-mem (t-hrs s)) (p' s));
    struct-rewrite-modifies R
     (λs. t-hrs-update (hrs-mem-update (heap-update (p'' s)
            (u s (field-setter (f' s))))) s)
     (λs. t-hrs-update (hrs-mem-update (heap-update (p' s)
            (field-setter (f' s) (h-val-p' s)))) s);
    struct-rewrite-guard S (λs. c-guard (p' s)) ⟧ ⟹
  struct-rewrite-modifies (λs. P s ∧ Q s ∧ R s ∧ S s)
    (λs. t-hrs-update (hrs-mem-update (heap-update (p'' s)
           (u s (field-setter (f' s))))) s)
   (λs. t-hrs-update (hrs-mem-update (heap-update (Ptr (field-lvalue (p s) field-name))
           (f s (h-val (hrs-mem (t-hrs s)) (Ptr (field-lvalue (p s) field-name))))))
s)
 apply (clarsimp simp: struct-rewrite-expr-def struct-rewrite-guard-def struct-rewrite-modifies-def
valid-struct-field-def heap-lift--wrap-h-val-def)
 apply (erule-tac x = s in allE)+
 apply (erule impE, assumption)+
 apply (erule-tac t = t-hrs-update (hrs-mem-update (heap-update (p'' s)
                      (u s (field-setter (f' s))))) s
           and s = t-hrs-update (hrs-mem-update (heap-update (p' s)
                      (field-setter (f' s) (h-val (hrs-mem (t-hrs s)) (p' s))))) s
            in subst)
 apply (rule lense.upd-cong[where get = t-hrs and upd = t-hrs-update])
  apply assumption
 apply (rule lense.upd-cong[OF read-write-valid-hrs-mem])
 apply (subst heap-update-field-root-conv''')
```

apply *assumption*+
apply (*simp add*: *typ-uinfo-t-def lense.fg-cons*)
apply (*subst h-val-field-from-bytes'*)
apply *assumption*+
apply (*simp add*: *lense.fg-cons*)
apply (*subst update-ti-update-ti-t*)
apply (*simp add*: *size-of-def*)
apply (*subst update-ti-s-adjust-ti-to-bytes-p*)
apply (*erule lense.fg-cons*)
apply (*subst lense.upd-cong*[**where** *get* = *field-getter* **and** *upd* = *field-setter*])
apply *auto*
done

lemma *struct-rewrite-modifies-Array-field* [*heap-abs*]:
⟦ *valid-struct-field field-name* (*field-getter* :: ('*a*::*xmem-type*) ⇒ (('*f*::*array-outer-max-size*)['*n*::*array-max-coun*
field-setter t-hrs t-hrs-update;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q f' f;
⋀*s*. *heap-lift--wrap-h-val* (*h-val-p' s*) (*h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*));
struct-rewrite-modifies R
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p'' s*)
(*u s* (*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s* (*index a* (*nat
k)))))))) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*)
(*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s* (*index a* (*nat k*))))
(*h-val-p' s*)))) *s*);
struct-rewrite-guard S (λ*s*. *c-guard* (*p' s*)) ⟧ ⟹
struct-rewrite-modifies (λ*s*. *P s* ∧ *Q s* ∧ *R s* ∧ *S s* ∧ 0 ≤ *k* ∧ *nat k* < *CARD*('*n*))
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p'' s*)
(*u s* (*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s* (*index a* (*nat k*))))))))
s)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update*
(*ptr-coerce* (*Ptr* (*field-lvalue* (*p s*) *field-name*) :: ('*f*['*n*]) *ptr*) +$_p$ *k*)
(*f s* (*h-val* (*hrs-mem* (*t-hrs s*)) (*ptr-coerce* (*Ptr* (*field-lvalue* (*p s*)
field-name) :: ('*f*['*n*]) *ptr*) +$_p$ *k* :: '*f ptr*))))) *s*)
using *ptr-coerce.simps*[*simp del*]
apply (*clarsimp simp*: *struct-rewrite-expr-def struct-rewrite-guard-def struct-rewrite-modifies-def
valid-struct-field-def heap-lift--wrap-h-val-def*)
subgoal for *s*
apply (*erule-tac x* = *s* **in** *allE*)+
apply (*erule impE*, *assumption*)+
apply (*erule-tac t* = *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p'' s*)
(*u s*(*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s* (*index a*
(*nat k*)))))))) *s*
and *s* = *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p' s*)
(*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f' s* (*index a* (*nat
k))))
(*h-val* (*hrs-mem* (*t-hrs s*)) (*p' s*)))))) *s*

```
      in subst)
   apply (rule lense.upd-cong[where get = t-hrs and upd = t-hrs-update])
    apply assumption
   apply (rule lense.upd-cong[OF read-write-valid-hrs-mem])
   apply (cases k, clarsimp)
    apply (subst heap-update-array-element[symmetric])
     apply assumption
    apply simp
    apply (subst heap-update-field-root-conv''')
      apply assumption+
     apply (simp add: typ-uinfo-t-def lense.fg-cons)
    apply (subst h-val-field-from-bytes')
     apply assumption+
     apply (simp add: lense.fg-cons)
    apply (subst heap-access-Array-element[symmetric])
     apply simp
    apply (subst h-val-field-from-bytes')
     apply assumption+
     apply (simp add: lense.fg-cons)
    apply clarsimp
    apply (subst update-ti-update-ti-t)
     apply (simp add: size-of-def)
    apply (subst update-ti-s-adjust-ti-to-bytes-p)
     apply (erule lense.fg-cons)
    apply clarsimp
    apply (subst lense.upd-cong[of field-getter field-setter])
     apply auto
   done
 done
```

```
definition
  valid-globals-field ::
    ('s ⇒ 't)
    ⇒ ('s ⇒ 'a)
    ⇒ (('a ⇒ 'a) ⇒ 's ⇒ 's)
    ⇒ ('t ⇒ 'a)
    ⇒ (('a ⇒ 'a) ⇒ 't ⇒ 't)
    ⇒ bool
where
  valid-globals-field st old-getter old-setter new-getter new-setter ≡
    (∀ s. new-getter (st s) = old-getter s)
    ∧ (∀ s v. new-setter v (st s) = st (old-setter v s))

lemma abs-expr-globals-getter [heap-abs]:
  ⟦ valid-globals-field st old-getter old-setter new-getter new-setter ⟧
    ⟹ abs-expr st (λ-. True) new-getter old-getter
```

apply (*clarsimp simp*: *valid-globals-field-def abs-expr-def*)
done

lemma *abs-expr-globals-setter* [*heap-abs*]:
⟦ *valid-globals-field st old-getter old-setter new-getter new-setter*;
⋀*old*. *abs-expr st* (*P old*) (*v old*) (*v′ old*) ⟧
⟹ *abs-modifies st* (λ*s*. ∀ *old*. *P old s*) (λ*s*. *new-setter* (λ*old*. *v old s*) *s*) (λ*s*. *old-setter* (λ*old*. *v′ old s*) *s*)
apply (*clarsimp simp*: *valid-globals-field-def abs-expr-def abs-modifies-def*)
done

lemma *struct-rewrite-expr-globals-getter* [*heap-abs*]:
⟦ *valid-globals-field st old-getter old-setter new-getter new-setter* ⟧
⟹ *struct-rewrite-expr* (λ-. *True*) *old-getter old-getter*
apply (*clarsimp simp*: *struct-rewrite-expr-def*)
done

lemma *struct-rewrite-modifies-globals-setter* [*heap-abs*]:
⟦ *valid-globals-field st old-getter old-setter new-getter new-setter*;
⋀*old*. *struct-rewrite-expr* (*P old*) (*v′ old*) (*v old*) ⟧
⟹ *struct-rewrite-modifies* (λ*s*. ∀ *old*. *P old s*) (λ*s*. *old-setter* (λ*old*. *v′ old s*) *s*) (λ*s*. *old-setter* (λ*old*. *v old s*) *s*)
apply (*clarsimp simp*: *valid-globals-field-def struct-rewrite-expr-def struct-rewrite-modifies-def*)
done

lemma *abs-spec-may-not-modify-globals*[*heap-abs*]:
abs-spec st (λ-. *True*) {(*a*, *b*). *meq b a*} {(*a*, *b*). *meq b a*}
apply (*clarsimp simp*: *abs-spec-def meq-def*)
done

lemma *abs-spec-modify-global*[*heap-abs*]:
valid-globals-field st old-getter old-setter new-getter new-setter ⟹
abs-spec st (λ-. *True*) {(*a*, *b*). *C a b*} {(*a*, *b*). *C′ a b*} ⟹
abs-spec st (λ-. *True*) {(*a*, *b*). *mex* (λ*x*. *C* (*new-setter* (λ-. *x*) *a*) *b*)} {(*a*, *b*). *mex* (λ*x*. *C′* (*old-setter* (λ-. *x*) *a*) *b*)}
apply (*fastforce simp*: *abs-spec-def mex-def valid-globals-field-def*)
done

lemma *uint-scast*:
uint (*scast x* :: *′a word*) = *uint* (*x* :: *′a*::*len signed word*)
apply (*subst down-cast-same* [*symmetric*])
apply (*clarsimp simp*: *cast-simps*)
apply (*subst uint-up-ucast*)
apply (*clarsimp simp*: *cast-simps*)

apply *simp*
done

lemma *to-bytes-signed-word*:
to-bytes (*x* :: 'a::*len8 signed word*) *p* = *to-bytes* (*scast x* :: 'a *word*) *p*
by (*clarsimp simp*: *uint-scast to-bytes-def typ-info-word word-rsplit-def*)

lemma *from-bytes-signed-word*:
length p = *len-of TYPE*('a) *div 8* ⟹
(*from-bytes p* :: 'a::*len8 signed word*) = *ucast* (*from-bytes p* :: 'a *word*)
by (*clarsimp simp*: *from-bytes-def word-rcat-def scast-def cast-simps typ-info-word*)

lemma *hrs-mem-update-signed-word*:
hrs-mem-update (*heap-update* (*ptr-coerce p* :: 'a::*len8 word ptr*) (*scast val* :: 'a::*len8 word*))
= *hrs-mem-update* (*heap-update p* (*val* :: 'a::*len8 signed word*))
apply (*rule ext*)
apply (*clarsimp simp*: *hrs-mem-update-def split-def*)
apply (*clarsimp simp*: *heap-update-def to-bytes-signed-word*
size-of-def typ-info-word)
done

lemma *h-val-signed-word*:
(*h-val a p* :: 'a::*len8 signed word*) = *ucast* (*h-val a* (*ptr-coerce p* :: 'a *word ptr*))
apply (*clarsimp simp*: *h-val-def*)
apply (*subst from-bytes-signed-word*)
apply (*clarsimp simp*: *size-of-def typ-info-word*)
apply (*clarsimp simp*: *size-of-def typ-info-word*)
done

lemma *align-of-signed-word*:
align-of TYPE('a::*len8 signed word*) = *align-of TYPE*('a *word*)
by (*clarsimp simp*: *align-of-def typ-info-word*)

lemma *size-of-signed-word*:
size-of TYPE('a::*len8 signed word*) = *size-of TYPE*('a *word*)
by (*clarsimp simp*: *size-of-def typ-info-word*)

lemma *c-guard-ptr-coerce*:
⟦ *align-of TYPE*('a) = *align-of TYPE*('b);
size-of TYPE('a) = *size-of TYPE*('b) ⟧ ⟹
c-guard (*ptr-coerce p* :: ('b::*c-type*) *ptr*) = *c-guard* (*p* :: ('a::*c-type*) *ptr*)
apply (*clarsimp simp*: *c-guard-def ptr-aligned-def c-null-guard-def*)
done

lemma *word-rsplit-signed*:
(*word-rsplit* (*ucast v'* :: ('a::*len*) *signed word*) :: *8 word list*) = *word-rsplit* (*v'* :: 'a *word*)

apply (*clarsimp simp*: *word-rsplit-def*)
apply (*clarsimp simp*: *cast-simps*)
done

lemma *heap-update-signed-word*:
heap-update (*ptr-coerce p* :: *'a word ptr*) (*scast v*) = *heap-update* (*p* :: (*'a*::*len8*) *signed word ptr*) *v*
heap-update (*ptr-coerce p'* :: *'a signed word ptr*) (*ucast v'*) = *heap-update* (*p'* :: (*'a*::*len8*) *word ptr*) *v'*
apply (*auto simp*: *heap-update-def to-bytes-def typ-info-word word-rsplit-def cast-simps uint-scast*)
done

lemma *heap-update-padding-signed-word*:
heap-update-padding (*ptr-coerce p* :: *'a word ptr*) (*scast v*) *bs* = *heap-update-padding* (*p* :: (*'a*::*len8*) *signed word ptr*) *v bs*
heap-update-padding (*ptr-coerce p'* :: *'a signed word ptr*) (*ucast v'*) *bs* = *heap-update-padding* (*p'* :: (*'a*::*len8*) *word ptr*) *v' bs*
by (*auto simp*: *heap-update-padding-def to-bytes-def typ-info-word word-rsplit-def cast-simps uint-scast*)

lemma *typ-heap-simulation-c-guard*:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
v (*st s*) *p* ⟧ ⟹ *c-guard p*
by (*clarsimp simp*: *valid-implies-cguard.valid-implies-c-guard* [*OF typ-heap-simulation.axioms*(*5*)])

abbreviation (*input*)
scast-f :: ((*'a*::*len*) *signed word ptr* ⇒ *'a signed word*)
⇒ (*'a word ptr* ⇒ *'a word*)
where
scast-f f ≡ (λ*p*. *scast* (*f* (*ptr-coerce p*)))

abbreviation (*input*)
ucast-f :: ((*'a*::*len*) *word ptr* ⇒ *'a word*)
⇒ (*'a signed word ptr* ⇒ *'a signed word*)
where
ucast-f f ≡ (λ*p*. *ucast* (*f* (*ptr-coerce p*)))

abbreviation (*input*)
cast-f' :: (*'a ptr* ⇒ *'x*) ⇒ (*'b ptr* ⇒ *'x*)
where
cast-f' f ≡ (λ*p*. *f* (*ptr-coerce p*))

lemma *read-write-validE-weak*:
⟦ *lense r w*;
⟦ ⋀*f s*. *r* (*w f s*) = *f* (*r s*);
⋀*f s*. *f* (*r s*) = (*r s*) ⟹ *w f s* = *s* ⟧ ⟹ *R* ⟧
⟹ *R*
apply *atomize-elim*

apply (*unfold lense-def*)
apply *metis*
done

lemma *lense-transcode*:
⟦ *lense r w*; ⋀*v*. *f′* (*f v*) = *v*; ⋀*v*. *f* (*f′ v*) = *v* ⟧ ⟹
lense (λ*s*. *f′* (*r s*)) (λ*g s*. *w* (λ*old*. *f* (*g* (*f′ old*))) *s*)
apply (*unfold lense-def*)
apply (*simp add*:*comp-def*)
done

lemma *typ-heap-simulation-signed-word*:
notes *align-of-signed-word* [*simp*]
size-of-signed-word [*simp*]
heap-update-signed-word [*simp*]
shows
⟦ *typ-heap-simulation st*
(*r* :: *'s* ⇒ (*'a*::*len8*) *word ptr* ⇒ *'a word*) *w*
v t-hrs t-hrs-update ⟧
⟹ *typ-heap-simulation st*
(λ*s p*. *ucast* (*r s* (*ptr-coerce p*)) :: *'a signed word*)
(λ*p f*. (*w* (*ptr-coerce p*) ((λ*x*. *scast* (*f* (*ucast x*))))))
(λ*s p*. *v s* (*ptr-coerce p*))
t-hrs t-hrs-update
apply (*clarsimp simp*: *typ-heap-simulation-def*
map.compositionality o-def c-guard-ptr-coerce)
apply (*intro conjI impI*)
subgoal
apply (*clarsimp simp add*: *read-simulation-def*)
apply (*drule-tac x*=*s* **in** *spec*)+
apply (*drule-tac x*=*ptr-coerce p* **in** *spec*)+
by (*simp add*: *h-val-signed-word*)
subgoal
apply (*clarsimp simp add*: *write-simulation-def write-simulation-axioms-def*)
subgoal for *s p bs x*
apply (*erule-tac x*=*s* **in** *allE*)
apply (*erule-tac x*=(*PTR-COERCE*(*'a signed word* → *'a word*) *p*) **in** *allE*)
apply *clarsimp*
apply (*erule-tac x*=*bs* **in** *allE*)
apply *clarsimp*
apply (*erule-tac x*= *SCAST*(*'a signed* → *'a*) *x* **in** *allE*)
using *heap-update-padding-signed-word*
by (*metis* (*mono-tags*, *lifting*) *hrs-mem-update-cong*)
done

subgoal
by (*clarsimp simp add*: *write-preserves-valid-def*)

subgoal
apply (*clarsimp simp add*: *valid-implies-cguard-def*)
apply (*drule spec*, *drule spec*, *erule* (*1*) *impE*)+
apply (*subst* (*asm*) *c-guard-ptr-coerce*, *simp*, *simp*)
by *blast*
subgoal
apply (*simp* (*no-asm-use*) *add*: *valid-only-typ-desc-dependent-def*)
by *blast*
subgoal
apply (*simp* (*no-asm-use*) *add*: *pointer-lense-def*)
apply *clarsimp*
by (*metis comp-apply ucast-scast-id*)
done

lemma *c-guard-ptr-ptr-coerce*:
⟦ *c-guard* (*a* :: ('*a*::*c-type*) *ptr ptr*); *ptr-val a* = *ptr-val b* ⟧ ⟹
c-guard (*b* :: ('*b*::*c-type*) *ptr ptr*)
by (*clarsimp simp*: *c-guard-def ptr-aligned-def c-null-guard-def*)

abbreviation (*input*)
ptr-coerce-f :: ('*a ptr ptr* ⇒ '*a ptr*) ⇒ ('*b ptr ptr* ⇒ '*b ptr*)
where
ptr-coerce-f f ≡ (λ*p*. *ptr-coerce* (*f* (*ptr-coerce p*)))

abbreviation (*input*)
ptr-coerce-range-f :: ('*a ptr* ⇒ *bool*) ⇒ ('*b ptr* ⇒ *bool*)
where
ptr-coerce-range-f f ≡ (λ*p*. (*f* (*ptr-coerce p*)))

lemma *typ-heap-simulation-ptr-coerce*:
⟦ *typ-heap-simulation st*
(*r* :: '*s* ⇒ ('*a*::*c-type*) *ptr ptr* ⇒ '*a ptr*) *w*
v t-hrs t-hrs-update ⟧
⟹ *typ-heap-simulation st*
(λ*s p*. *ptr-coerce* (*r s* (*ptr-coerce p*)) :: ('*b*::*c-type*) *ptr*)
(λ*p f*. (*w* (*ptr-coerce p*) ((λ*x*. *ptr-coerce* (*f* (*ptr-coerce x*))))))
(λ*s p*. *v s* (*ptr-coerce p*))
t-hrs t-hrs-update
apply (*clarsimp simp*: *typ-heap-simulation-def fun-upd-def*)
apply (*intro conjI impI*)
subgoal
by (*clarsimp simp*: *read-simulation-def h-val-def typ-info-ptr from-bytes-def*)
subgoal
apply (*clarsimp simp add*: *write-simulation-def write-simulation-axioms-def*)
apply (*erule allE*, *erule allE*, *erule* (*1*) *impE*)+
apply (*erule-tac x*=*bs* **in** *allE*)
apply *clarsimp*
apply (*erule-tac x*=*ptr-coerce x* **in** *allE*)

apply (*clarsimp simp*: *heap-update-padding-def* [*abs-def*] *to-bytes-def typ-info-ptr*)
done
subgoal
apply (*clarsimp simp add*: *write-preserves-valid-def*)
done
subgoal
apply (*clarsimp simp add*: *valid-implies-cguard-def*)
apply (*drule spec*, *drule spec*, *erule* (*1*) *impE*)+
apply (*subst* (*asm*) *c-guard-ptr-coerce*, *simp*, *simp*)
by *blast*
subgoal
apply (*simp* (*no-asm-use*) *add*: *valid-only-typ-desc-dependent-def*)
by *blast*
subgoal
apply (*simp* (*no-asm-use*) *add*: *pointer-lense-def*)
apply *clarsimp*
by (*metis comp-apply ptr-coerce-id ptr-coerce-idem*)
done

lemmas *signed-typ-heap-simulations* =
typ-heap-simulation-signed-word
typ-heap-simulation-ptr-coerce [**where** *'a*=(*'x*::*len8*) *word* **and** *'b*=(*'x*::*len8*) *signed word*]
typ-heap-simulation-ptr-coerce [**where** *'a*=(*'x*::*len8*) *word ptr* **and** *'b*=(*'x*::*len8*) *signed word ptr*]
typ-heap-simulation-ptr-coerce [**where** *'a*=(*'x*::*len8*) *word ptr ptr* **and** *'b*=(*'x*::*len8*) *signed word ptr ptr*]
typ-heap-simulation-ptr-coerce [**where** *'a*=(*'x*::*len8*) *word ptr ptr ptr* **and** *'b*=(*'x*::*len8*) *signed word ptr ptr ptr*]

lemma *ptr-coerce-eq*:
(*ptr-coerce x* = *ptr-coerce y*) = (*x* = *y*)
by (*cases x*, *cases y*, *auto*)

lemma *signed-word-heap-opt* [*L2opt*]:
(*scast* (((λ*x*. *ucast* (*a* (*ptr-coerce x*))) (*p* := *v* :: *'a*::*len signed word*)) (*b* :: *'a signed word ptr*)))
= ((*a*(*ptr-coerce p* := (*scast v* :: *'a word*))) ((*ptr-coerce b*) :: *'a word ptr*))
by (*auto simp*: *fun-upd-def ptr-coerce-eq*)

lemma *signed-word-heap-ptr-coerce-opt* [*L2opt*]:
(*ptr-coerce* (((λ*x*. *ptr-coerce* (*a* (*ptr-coerce x*))) (*p* := *v* :: *'a ptr*)) (*b* :: *'a ptr ptr*)))
= ((*a*(*ptr-coerce p* := (*ptr-coerce v* :: *'b ptr*))) ((*ptr-coerce b*) :: *'b ptr ptr*))
by (*auto simp*: *fun-upd-def ptr-coerce-eq*)

declare *ptr-coerce-idem* [*L2opt*]
declare *scast-ucast-id* [*L2opt*]
declare *ucast-scast-id* [*L2opt*]

lemma *heap-abs-expr-c-guard-array* [*heap-abs*]:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
abs-expr st P x′ (λ*s*. *ptr-coerce* (*x s*) :: *′a ptr*) ⟧ ⟹
abs-guard st
(λ*s*. *P s* ∧ (∀ *a* ∈ *set* (*array-addrs* (*x′ s*) *CARD*(*′b*)). *v s a*))
(λ*s*. *c-guard* (*x s* :: (*′a*::*array-outer-max-size*, *′b*::*array-max-count*) *array ptr*))
apply (*clarsimp simp*: *abs-expr-def abs-guard-def simple-lift-def root-ptr-valid-def*)
apply (*subgoal-tac* ∀ *a*∈*set* (*array-addrs* (*x′* (*st s*)) *CARD*(*′b*)). *c-guard a*)
apply (*erule allE*, *erule* (*1*) *impE*)
apply (*rule c-guard-array-c-guard*)
apply (*subst* (*asm*) (*2*) *set-array-addrs*)
apply *force*
apply *clarsimp*
apply (*erule* (*1*) *my-BallE*)
apply (*drule* (*1*) *typ-heap-simulation-c-guard*)
apply *simp*
done

lemma *fold-over-st*:
⟦ *xs* = *ys*; *P s*;
⋀*s x*. *x* ∈ *set xs* ∧ *P s* ⟹ *P* (*g x s*) ∧ *f x* (*st s*) = *st* (*g x s*)
⟧ ⟹ *fold f xs* (*st s*) = *st* (*fold g ys s*)
apply (*erule subst*)
apply (*induct xs arbitrary*: *s*)
apply *simp*
apply *simp*
done

lemma *fold-lift-write*:
⟦ *xs* = *ys*; *lense r w*
⟧ ⟹ *fold* (λ*i*. *w* (*f i*)) *xs s* = *w* (*fold f ys*) *s*
apply (*erule subst*)
apply (*induct xs arbitrary*: *s*)
apply (*simp add*: *lense.upd-same*)
apply (*simp add*: *lense.upd-compose*)
done

lemma *fold-heap-update-list-nmem-same*:
⟦ *n* ∗ *size-of TYPE*(*′a* :: *mem-type*) < *addr-card*;
n ∗ *size-of TYPE*(*′a*) ≤ *k*; *k* < *addr-card*;
⋀*i h*. *length* (*pad i h*) = *size-of TYPE*(*′a*) ⟧ ⟹

```
    h (ptr-val (p :: 'a ptr) + of-nat k) =
    (fold (λi h. heap-update-list (ptr-val (p +_p int i))
            (to-bytes (val i h :: 'a) (pad i h)) h) [0..<n] h) (ptr-val p + of-nat k)
  apply (induct n arbitrary: k)
   apply simp
  apply (clarsimp simp: CTypesDefs.ptr-add-def simp del: mult-Suc)
  apply (subst heap-update-nmem-same)
   apply (subst len)
    apply simp
   apply (simp add: intvl-def)
   apply (intro allI impI)
   apply (subst (asm) of-nat-mult[symmetric] of-nat-add[symmetric])+
   apply (rename-tac j)
   apply (erule-tac Q = of-nat k = of-nat (n * size-of TYPE('a) + j) in contra-
pos-pn)
   apply (subst of-nat-inj)
     apply (subst len-of-addr-card)
     apply simp
    apply (subst len-of-addr-card)
    apply simp
   apply simp
  apply simp
  done

lemma heap-list-of-disjoint-fold-heap-update-list:
  ⟦ n * size-of TYPE('a :: mem-type) < addr-card;
    n * size-of TYPE('a) + k < addr-card;
    ⋀i h. length (pad i h) = size-of TYPE('a) ⟧ ⟹
   heap-list (fold (λi h. heap-update-list (ptr-val ((p :: 'a ptr) +_p int i))
                          (to-bytes (val i h :: 'a) (pad i h)) h) [0..<n] h)
          k (ptr-val (p +_p int n))
  = heap-list h k (ptr-val (p +_p int n))
  apply (rule nth-equalityI, force)
  subgoal for i
    apply (clarsimp simp: heap-list-nth)
    apply (rule subst[where t = ptr-val (p +_p int n) + of-nat i
       and s = ptr-val p + of-nat (n * size-of TYPE('a) + i)])
     apply (clarsimp simp: CTypesDefs.ptr-add-def)
    apply (rule fold-heap-update-list-nmem-same[symmetric])
      apply simp-all
    done
  done


lemma fold-heap-update-list:
  n * size-of TYPE('a :: mem-type) < 2^addr-bitsize ⟹
   fold (λi h. heap-update-list (ptr-val ((p :: 'a ptr) +_p int i))
               (to-bytes (val i :: 'a)
                 (heap-list h (size-of TYPE('a)) (ptr-val (p +_p int i)))) h)
```

```
      [0..<n] h =
   fold (λi. heap-update-list (ptr-val (p +p int i))
            (to-bytes (val i)
              (heap-list h (size-of TYPE('a)) (ptr-val (p +p int i)))))
      [0..<n] h
  apply (induct n)
   apply simp
  apply clarsimp
  apply (subst heap-list-of-disjoint-fold-heap-update-list)
    apply (simp add: len-of-addr-card[symmetric])+
  done


lemma access-ti-list-array-unpacked:
  ⟦ ∀ n. size-td-tuple (f n) = v3; length xs = v3 * n;
     ∀ m xs. length xs = v3 ∧ m < n ⟶
           access-ti-tuple (f m) (FCP g) xs = h m xs
   ⟧ ⟹
   access-ti-list (map f [0 ..< n]) (FCP g) xs
     = foldl (@) [] (map (λm. h m (take v3 (drop (v3 * m) xs))) [0 ..< n])
  apply (subgoal-tac ∀ ys. size-td-list (map f ys) = v3 * length ys)
   prefer 2
  subgoal
    apply (rule allI)
    subgoal for ys by (induct ys) auto
    done
  apply (induct n arbitrary: xs)
   apply simp
  apply (simp add: access-ti-append)
  apply (rule foldl-cong)
    apply simp
   apply (rule map-cong[OF refl])
   apply (subst take-drop)
   apply (subst take-take)
   apply (subst min-absorb1)
    apply clarsimp
    apply (metis Suc-leI mult-Suc-right nat-mult-le-cancel-disj)
   apply (subst take-drop[symmetric])
   apply (rule refl)
  apply simp
  done

lemma concat-nth-chunk:
  ⟦ ∀ x ∈ set xs. length (f x) = chunk; n < length xs ⟧
   ⟹ take chunk (drop (n * chunk) (concat (map f xs))) = f (xs ! n)
  apply (induct xs arbitrary: n, force)
  subgoal for x xs n
    apply (cases n)
     apply clarsimp
```

```
  apply clarsimp
  done
done


lemma array-update-split:
  ⟦ typ-heap-simulation st (r :: 's ⇒ ('a::array-outer-max-size) ptr ⇒ 'a) w
                  v t-hrs t-hrs-update;
    ∀ x ∈ set (array-addrs (ptr-coerce p) CARD('b::array-max-count)).
       v (st s) x
  ⟧ ⟹ st (t-hrs-update (hrs-mem-update (heap-update p (arr :: 'a['b])))) s) =
       fold (λi. w (ptr-coerce p +_p int i) (λx. index arr i))
            [0 ..< CARD('b)] (st s)
 apply (clarsimp simp: typ-heap-simulation-def heap-raw-state-def valid-implies-cguard-def
read-simulation-def write-preserves-valid-def)

  apply (drule write-simulation.write-commutes-atomic)


  apply (subst fold-over-st[OF refl,
         where P = λs. ∀ x∈set (array-addrs (ptr-coerce p) CARD('b)). v (st s) x
           and g = λi. t-hrs-update (hrs-mem-update (heap-update
                       (ptr-coerce p +_p int i) (index arr i)))])
   apply simp
  subgoal for sa x
   apply (subgoal-tac v (st sa) (ptr-coerce p +_p int x))
    apply clarsimp
   apply (clarsimp simp: set-array-addrs)
   apply metis
   done
  apply (rule arg-cong[where f = st])
  apply (subst hrs-mem-update-def)+


  apply (subst fold-lift-write[OF refl, where w = t-hrs-update])
   apply assumption
  apply (rule arg-cong[where f = λf. t-hrs-update f s])
  apply (rule ext)
  apply (subst fold-lift-write[OF refl,
       where r = fst and w = λf s. case s of (h, z) ⇒ (f h, z)])
   apply (simp (no-asm) add: lense-def)
  apply clarsimp


  apply (clarsimp simp: heap-update-def[abs-def])
  apply (subst coerce-heap-update-to-heap-updates[unfolded foldl-conv-fold,
         where chunk = size-of TYPE('a) and m = CARD('b)])
   apply (rule size-of-array[simplified mult.commute])
  apply simp
```

apply (*subst fold-heap-update-list*[*OF array-count-size*])
apply (*rule fold-cong*[*OF refl refl*])
subgoal for *a x*

apply (*clarsimp simp*: *CTypesDefs.ptr-add-def*)
apply (*rule arg-cong*[**where** *f* = *heap-update-list* (*ptr-val p* + *of-nat x* ∗ *of-nat* (*size-of TYPE*('*a*)))])

apply (*subst fcp-eta*[**where** *g* = *arr*, *symmetric*])
apply (*clarsimp simp*: *to-bytes-def typ-info-array array-tag-def array-tag-n-eq simp del*: *fcp-eta*)
apply (*subst access-ti-list-array-unpacked*)
apply *clarsimp*
apply (*rule refl*)
apply (*simp add*: *size-of-def*)
apply *clarsimp*
apply (*rule refl*)
apply (*clarsimp simp*: *foldl-conv-concat*)

apply (*subgoal-tac*
⋀*x*. *x* < *CARD*('*b*) ⟹
size-td (*typ-info-t TYPE*('*a*))
≤ *CARD*('*b*) ∗ *size-td* (*typ-info-t TYPE*('*a*)) − *size-td* (*typ-info-t TYPE*('*a*)) ∗ *x*)
prefer *2*
apply (*subst le-diff-conv2*)
apply *simp*
apply (*subst mult.commute*, *subst mult-Suc*[*symmetric*])
apply (*rule mult-le-mono1*)
apply *simp*

apply (*subst concat-nth-chunk*)
apply *clarsimp*
apply (*subst fd-cons-length*)
apply *simp*
apply (*simp add*: *size-of-def*)
apply (*simp add*: *size-of-def*)
apply *simp*
apply (*subst drop-heap-list-le*)
apply (*simp add*: *size-of-def*)
apply (*subst take-heap-list-le*)
apply (*simp add*: *size-of-def*)
apply (*clarsimp simp*: *size-of-def*)
apply (*subst mult.commute*, *rule refl*)
done
done

lemma *fold-update-id*:
⟦ *lense getter setter*;
∀ *i* ∈ *set xs*. ∀ *j* ∈ *set xs*. (*i* = *j*) = (*ind i* = *ind j*);
∀ *i* ∈ *set xs*. *val i* = *getter s* (*ind i*)
⟧ ⟹ *fold* (λ*i*. *setter* (λ*x*. *x*(*ind i* := *val i*))) *xs s* = *s*
apply (*induct xs*)
apply *simp*
apply (*rename-tac a xs*)
apply *clarsimp*
apply (*subgoal-tac setter* (λ*x*. *x*(*ind a* := *getter s* (*ind a*))) *s* = *s*)
apply *simp*
apply (*subst* (*asm*) *lense-def*)
apply *simp*
done

lemma *fold-update-id′*:
⟦ *pointer-lense r w*;
∀ *i* ∈ *set xs*. ∀ *j* ∈ *set xs*. (*i* = *j*) = (*ind i* = *ind j*);
∀ *i* ∈ *set xs*. *val i* = *r s* (*ind i*)
⟧ ⟹ *fold* (λ*i*. *w* (*ind i*) (λ-. *val i*)) *xs s* = *s*
apply (*induct xs*)
apply *simp*
apply (*rename-tac a xs*)
apply *clarsimp*
apply (*subgoal-tac w* (*ind a*) (λ*x*. *r s* (*ind a*)) *s* = *s*)
apply *simp*
apply (*subst* (*asm*) *pointer-lense-def*)
apply *simp*
done

lemma *array-count-index*:
⟦ *i* < *CARD*(*′b*::*array-max-count*); *j* < *CARD*(*′b*) ⟧
⟹ (*i* = *j*) =
((*of-nat* (*i* ∗ *size-of TYPE*(*′a*::*array-outer-max-size*)) :: *addr*)
= *of-nat* (*j* ∗ *size-of TYPE*(*′a*)))
apply (*rule subst*[**where** *t* = *i* = *j* **and** *s* = *i* ∗ *size-of TYPE*(*′a*) = *j* ∗ *size-of TYPE*(*′a*)])
apply *clarsimp*
apply (*subgoal-tac* ⋀ *i*. *i* < *CARD*(*′b*) ⟹ *i* ∗ *size-of TYPE*(*′a*) < 2 ^ *LENGTH*(*addr-bitsize*))
apply (*rule of-nat-inj*[*symmetric*]; *force*)
apply (*rule subst*[**where** *t* = *len-of TYPE*(*addr-bitsize*) **and** *s* = *addr-bitsize*], *force*)
apply (*rule less-trans*)
apply (*erule-tac b* = *CARD*(*′b*) **in** *mult-strict-right-mono*)
apply (*rule sz-nzero*)
apply (*rule array-count-size*)
done

lemma *le-outside-intvl*: $p < a \Longrightarrow 0 \notin \{a\ ..+b\} \Longrightarrow p \notin \{a\ ..+b\}$
apply (*clarsimp simp*: *intvl-def not-le not-less*)
by (*metis Word-Lemmas-Internal.word-wrap-of-natD add-increasing2 le0 le-iff-add less-le not-less*)

lemma *intvl-mult-split*:
$\{p\ ..+\ a * b\} = (\bigcup i{<}b.\ \{p + of\text{-}nat\ (i * a)\ ..+\ a\})$
proof *cases*
assume *a*: $0 < a$
note *of-nat-add*[*simp del*, *symmetric*, *simp*] *of-nat-mult*[*simp del*, *symmetric*, *simp*]
show *?thesis* **using** *a*
apply (*clarsimp simp*: *intvl-def*, *intro set-eqI iffI*; *clarsimp*)
subgoal for *i*
by (*intro bexI*[*of - i div a*] *exI*[*of - i mod a*])
(*simp-all add*: *dme pos-mod-bound less-mult-imp-div-less ac-simps*)
subgoal for *j k*
by (*intro exI*[*of - j * a + k*]) (*simp add*: *nat-index-bound*)
done
qed *simp*

lemma *intvl-mul-disjnt*:
fixes *n i* :: *'a*::*len word*
assumes *n*: *unat n* $< b$ **and** *i*: *unat i* $< b$ **and** *b*: $sz * b \leq 2\hat{}LENGTH('a)$
assumes *ni*: $n \neq i$
shows $\{n * word\text{-}of\text{-}nat\ sz..+sz\} \cap \{i * word\text{-}of\text{-}nat\ sz..+sz\} = \{\}$
proof −
{ fix *j l* **assume** *j*: $j < sz$ **and** *l*: $l < sz$
assume $n * word\text{-}of\text{-}nat\ sz + word\text{-}of\text{-}nat\ j = i * word\text{-}of\text{-}nat\ sz + word\text{-}of\text{-}nat\ l$
then have (*word-of-nat* (*unat n* $*$ *sz* $+$ *j*) :: *'a word*) $=$ *word-of-nat* (*unat i* $*$ *sz* $+$ *l*) **by** *simp*
moreover have *unat n* $*$ *sz* $+$ *j* $< 2\hat{}LENGTH('a)$
by (*intro less-le-trans*[*OF nat-index-bound b*] *n j*)
moreover have *unat i* $*$ *sz* $+$ *l* $< 2\hat{}LENGTH('a)$
by (*intro less-le-trans*[*OF nat-index-bound b*] *i l*)
ultimately have (*unat n* $*$ *sz* $+$ *j*) *div sz* $=$ (*unat i* $*$ *sz* $+$ *l*) *div sz*
by (*subst* (*asm*) *of-nat-inj*) *simp-all*
then have *unat n* $=$ *unat i*
using *j l* **by** *simp*
with *ni* **have** *False* **by** *simp* **}**
then show *?thesis*
by (*auto simp*: *intvl-def*)
qed

lemma *array-disj-helper*:

fixes $p :: ('a::mem\text{-}type['b])\ ptr$
assumes ni: $n < CARD('b)\ i < CARD('b)\ n \neq i$
assumes $valid$: $\forall x \in set\ (array\text{-}addrs\ (PTR\text{-}COERCE('a['b] \to 'a)\ p)\ CARD('b)).$
$c\text{-}guard\ x$
shows $\{ptr\text{-}val\ p + word\text{-}of\text{-}nat\ n * word\text{-}of\text{-}nat\ (size\text{-}of\ TYPE('a))..+size\text{-}of\ TYPE('a)\} \cap$
$\{ptr\text{-}val\ p + word\text{-}of\text{-}nat\ i * word\text{-}of\text{-}nat\ (size\text{-}of\ TYPE('a))..+size\text{-}of\ TYPE('a)\}$
$= \{\}$
proof −
have [$arith$]: $CARD('b) \leq size\text{-}of\ TYPE('a) * CARD('b)$
using $sz\text{-}nzero$[**where** $'a='a$, $arith$] **by** $simp$

have $0 \notin (\bigcup b{<}CARD('b).\ \{ptr\text{-}val\ p + of\text{-}nat\ (b * size\text{-}of\ TYPE('a))\ ..+\ size\text{-}of\ TYPE('a)\})$
using $valid$
apply ($clarsimp\ simp$: $set\text{-}array\text{-}addrs\ c\text{-}guard\text{-}def\ c\text{-}null\text{-}guard\text{-}def$)
subgoal premises $prems$ **for** b
using $prems(2-)\ prems(1)$[$rule\text{-}format$, $OF\ exI$, of
$Ptr\ (ptr\text{-}val\ p + word\text{-}of\text{-}nat\ b * word\text{-}of\text{-}nat\ (size\text{-}of\ TYPE('a)))\ b$]
by ($simp\ add$: $ptr\text{-}add\text{-}def$)
done
then have $0 \notin \{ptr\text{-}val\ p\ ..+\ size\text{-}of\ TYPE('a) * CARD('b)\}$
unfolding $intvl\text{-}mult\text{-}split$ **.**
from $zero\text{-}not\text{-}in\text{-}intvl\text{-}no\text{-}overflow$[$OF\ this$]
have $size\text{-}of\ TYPE('a) * CARD('b) \leq addr\text{-}card$
by ($simp\ add$: $addr\text{-}card$)
moreover note ni
ultimately show $?thesis$
by (smt ($verit$, $ccfv\text{-}SIG$) ‹$CARD('b) \leq size\text{-}of\ TYPE('a) * CARD('b)$› $addr\text{-}card\text{-}len\text{-}of\text{-}conv$
$intvl\text{-}disj\text{-}offset\ intvl\text{-}mul\text{-}disjnt\ order\text{-}less\text{-}le\text{-}trans\ unat\text{-}of\text{-}nat\text{-}len$)
qed

theorem $heap\text{-}abs\text{-}array\text{-}update$ [$heap\text{-}abs$]:
⟦ $typ\text{-}heap\text{-}simulation\ st\ (r :: 's \Rightarrow 'a\ ptr \Rightarrow 'a)\ w$
$v\ t\text{-}hrs\ t\text{-}hrs\text{-}update$;
$abs\text{-}expr\ st\ Pb\ b'\ b$;
$abs\text{-}expr\ st\ Pn\ n'\ n$;
$abs\text{-}expr\ st\ Pv\ y'\ y$
⟧ ⟹
$abs\text{-}modifies\ st\ (\lambda s.\ Pb\ s \wedge Pn\ s \wedge Pv\ s \wedge n'\ s < CARD('b) \wedge$
$(\forall ptr \in set\ (array\text{-}addrs\ (ptr\text{-}coerce\ (b'\ s))\ CARD('b)).\ (v\ s\ ptr)))$
$(\lambda s.\ w\ (ptr\text{-}coerce\ (b'\ s))\ +_p\ int\ (n'\ s))\ (\lambda v.\ y'\ s)\ \ s)$
$(\lambda s.\ t\text{-}hrs\text{-}update\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ (b\ s)\ (Arrays.update\ ((h\text{-}val\ (hrs\text{-}mem\ (t\text{-}hrs\ s))\ (b\ s)) :: ('a::array\text{-}outer\text{-}max\text{-}size)['b::array\text{-}max\text{-}count])\ (n\ s)\ (y\ s))))\ s)$

apply (*clarsimp simp*: *abs-modifies-def abs-expr-def*)
subgoal for *s*

apply (*subst array-update-split*
[**where** *st* = *st* **and** *t-hrs* = *t-hrs* **and** *t-hrs-update* = *t-hrs-update*])
apply *assumption*
apply *assumption*
apply (*clarsimp simp*: *typ-heap-simulation-def valid-implies-cguard-def read-simulation-def write-preserves-valid-def*)
apply (*drule write-simulation.write-commutes-atomic*)
apply (*subst fold-cong*[*OF refl refl*,
where $g = \lambda i.\ w\ \ (ptr\text{-}coerce\ (b'\ (st\ s))\ +_p\ int\ i)\ (\lambda x.$
if $i = n'\ (st\ s)$ *then* $y'\ (st\ s)$ *else* $r\ (st\ s)\ (ptr\text{-}coerce\ (b'\ (st\ s))\ +_p\ int\ i)$)])
subgoal for *x*
apply (*cases* $x = n'\ (st\ s)$)
apply *simp*
apply (*subst index-update2*)
apply *simp*
apply *simp*
apply (*rule arg-cong*[**where** $x = index\ (h\text{-}val\ (hrs\text{-}mem\ (t\text{-}hrs\ s))\ (b'\ (st\ s)))\ x$])
apply (*subst heap-access-Array-element*)
apply *simp*
apply (*clarsimp simp*: *set-array-addrs*)
apply *metis*
done

apply (*subst split-upt-on-n*[**where** $n = n\ s$])
apply *simp*
apply *clarsimp*
thm *fold-update-id*

apply (*subst fold-update-id'*[**where** $s = st\ s$])
apply *assumption*
apply (*clarsimp simp*: *CTypesDefs.ptr-add-def*)
apply (*subst of-nat-mult*[*symmetric*])+
apply (*rule array-count-index*)
apply (*erule less-trans*, *assumption*)+
apply *clarsimp*

apply (*subst fold-update-id'*)
apply *assumption*
apply (*clarsimp simp*: *CTypesDefs.ptr-add-def*)
apply (*subst of-nat-mult*[*symmetric*])+
apply (*erule array-count-index*)

apply *assumption*
apply *clarsimp*

apply (*subst pointer-lense.read-write-other*[**where** $r = r$ **and** $w = w$])
apply *assumption*
apply (*clarsimp simp*: *CTypesDefs.ptr-add-def disjnt-def*)
apply (*rule array-disj-helper*)
apply *simp*
apply *assumption*
apply *simp*
apply *blast*
apply (*rule refl*)
apply (*rule refl*)
done
done

lemma *heap-abs-array-access*[*heap-abs*]:
⟦ *typ-heap-simulation st* ($r :: 's \Rightarrow ('a::xmem\text{-}type)\ ptr \Rightarrow 'a$) w
v t-hrs t-hrs-update;
abs-expr st Pb b' b;
abs-expr st Pn n' n
⟧ ⟹
abs-expr st ($\lambda s.$ $Pb\ s \wedge Pn\ s \wedge n'\ s < CARD('b::finite) \wedge v\ s$ (*ptr-coerce* ($b'\ s$) $+_p$ *int* ($n'\ s$)))
($\lambda s.$ $r\ s$ (*ptr-coerce* ($b'\ s$) $+_p$ *int* ($n'\ s$)))
($\lambda s.$ *index* (*h-val* (*hrs-mem* (*t-hrs* s)) ($b\ s :: ('a['b])\ ptr$)) ($n\ s$))
apply (*clarsimp simp*: *typ-heap-simulation-def abs-expr-def valid-implies-cguard-def read-simulation-def write-simulation-def write-preserves-valid-def*)
apply (*subst heap-access-Array-element*)
apply *simp*
apply (*simp add*: *set-array-addrs*)
done

lemma *the-fun-upd-lemma1*:
($\lambda x.$ *the* ($f\ x$))($p := v$) = ($\lambda x.$ *the* ((f ($p :=$ *Some* v)) x))
by *auto*

lemma *the-fun-upd-lemma2*:
$\exists z.\ f\ p =$ *Some* $z \Longrightarrow$
($\lambda x.$ $\exists z.$ (f ($p :=$ *Some* v)) $x =$ *Some* z) = ($\lambda x.$ $\exists z.$ $f\ x =$ *Some* z)
by *auto*

lemma *the-fun-upd-lemma3*:
(($\lambda x.$ *the* ($f\ x$))($p := v$)) x = *the* ((f ($p :=$ *Some* v)) x)
by *simp*

lemma *the-fun-upd-lemma4*:
$\exists z.\ f\ p = Some\ z \Longrightarrow$
$(\exists z.\ (f\ (p := Some\ v))\ x = Some\ z) = (\exists z.\ f\ x = Some\ z)$
by *simp*

lemmas *the-fun-upd-lemmas* =
the-fun-upd-lemma1
the-fun-upd-lemma2
the-fun-upd-lemma3
the-fun-upd-lemma4

lemma *sword-update*:
$\bigwedge ptr :: ('a :: len)\ signed\ word\ ptr.$
$(\lambda(x :: 'a\ word\ ptr \Rightarrow 'a\ word)\ p :: 'a\ word\ ptr.$
if ptr-coerce $p = ptr$ *then scast* $(n :: 'a\ signed\ word)$ *else* x *(ptr-coerce* $p))$
$=$
$(\lambda(old :: 'a\ word\ ptr \Rightarrow 'a\ word)\ x :: 'a\ word\ ptr.$
if $x = ptr\text{-}coerce\ ptr$ *then scast* n *else* $old\ x)$
by *force*

Proof taken from $[\![?n < CARD(?'b);\ CARD(?'b) * size\text{-}of\ TYPE(?'a) < 2^{addr\text{-}bitsize}]\!] \Longrightarrow heap\text{-}update\ ?p\ (Arrays.update\ ?arr\ ?n\ ?v)\ ?hp = heap\text{-}update\ (array\text{-}ptr\text{-}index\ ?p\ False\ ?n)\ ?v\ (heap\text{-}update\ ?p\ ?arr\ ?hp)$

lemma (**in** *array-outer-max-size*) *array-index-unat-conv*:
assumes *x-bound*: $x < CARD('b :: array\text{-}max\text{-}count)$
assumes *k-bound*: $k < size\text{-}of\ TYPE('a)$
shows *unat* $(of\text{-}nat\ x * of\text{-}nat\ (size\text{-}of\ TYPE('a)) + (of\text{-}nat\ k :: addr))$
$= x * size\text{-}of\ TYPE('a) + k$
proof −
have *size*: $CARD('b) * size\text{-}of\ TYPE('a) < 2\ \hat{}\ addr\text{-}bitsize$
using *array-count-size* **by** *blast*
show *?thesis*
using *size x-bound k-bound*
apply (*cases size-of* $TYPE('a)$, *simp-all*)
apply (*cases* $CARD('b)$, *simp-all*)
apply (*subst unat-add-lem*[*THEN iffD1*])
apply (*simp add*: *unat-word-ariths unat-of-nat less-Suc-eq-le*)
apply (*subgoal-tac Suc* $x * size\text{-}of\ TYPE('a) < 2\ \hat{}\ addr\text{-}bitsize$, *simp-all*)
apply (*erule order-le-less-trans*[*rotated*], *simp add*: *add-mono*)
apply (*subst unat-mult-lem*[*THEN iffD1*])
apply (*simp add*: *unat-of-nat unat-add-lem*[*THEN iffD1*])
apply (*rule order-less-le-trans*, *erule order-le-less-trans*[*rotated*],
rule add-mono, *simp*+)
apply (*simp add*: *less-Suc-eq-le trans-le-add2*)
apply *simp*
apply (*simp add*: *unat-of-nat unat-add-lem*[*THEN iffD1*])

done
qed

lemma *ptr-span-array-index-disj*:
fixes *p*::('*a*::*array-outer-max-size*['*b*::*array-max-count*]) *ptr*
assumes *n-bound*: *n* < *CARD* ('*b*)
assumes *i-bound*: *i* < *n*
shows *disjnt* (*ptr-span* (*array-ptr-index p False n*)) (*ptr-span* (*array-ptr-index p False i*))
using *n-bound i-bound*
apply (*clarsimp simp add*: *array-ptr-index-def ptr-add-def intvl-def disjnt-def*)
apply (*intro set-eqI*)
apply *clarsimp*
apply (*drule word-unat.Rep-inject*[*THEN iffD2*])
apply (*clarsimp simp*: *array-index-unat-conv* [**where** '*b*='*b*] *nat-eq-add-iff1*)
by (*metis mult.commute nat-index-bound not-add-less1*)

lemma *ptr-span-array-ptr-index-disj*:
fixes *p*::('*a*::*array-outer-max-size*['*b*::*array-max-count*]) *ptr*
fixes *q*::('*a*['*b*::*array-max-count*]) *ptr*
assumes *n-bound*: *n* < *CARD* ('*b*)
assumes *m-bound*: *m* < *CARD* ('*b*)
assumes *disj*:*disjnt* (*ptr-span p*) (*ptr-span q*)
shows *disjnt* (*ptr-span* (*array-ptr-index p False n*)) (*ptr-span* (*array-ptr-index q False m*))
proof (*rule ccontr*)
assume ¬*disjnt* (*ptr-span* (*array-ptr-index p False n*)) (*ptr-span* (*array-ptr-index q False m*))

then obtain *k k'* **where**
k-bound: *k* < *size-of TYPE*('*a*) **and**
k'-bound: *k'* < *size-of TYPE*('*a*) **and**
eq: *ptr-val p* + *word-of-nat n* * *word-of-nat* (*size-of TYPE*('*a*)) + *word-of-nat k* =
ptr-val q + *word-of-nat m* * *word-of-nat* (*size-of TYPE*('*a*)) + *word-of-nat k'*
by (*auto simp add*: *intvl-def array-ptr-index-def ptr-add-def disjnt-def*)

define *i* **where** *i* = *unat* ((*word-of-nat n*::*addr*) * *word-of-nat* (*size-of TYPE*('*a*)) + *word-of-nat k*)
have *i-bound*: *i* < *CARD*('*b*) * *size-of TYPE*('*a*)
using *n-bound k-bound*
by(*simp add*: *i-def array-index-unat-conv* [*OF n-bound k-bound*] *mult.commute nat-index-bound*)

define *j* **where** *j* = *unat* ((*word-of-nat m*::*addr*) * *word-of-nat* (*size-of TYPE*('*a*)) + *word-of-nat k'*)
have *j-bound*: *j* < *CARD*('*b*) * *size-of TYPE*('*a*)
using *m-bound k'-bound*

by(*simp add*: *j-def array-index-unat-conv* [*OF m-bound k'-bound*] *mult.commute nat-index-bound*)

from *i-bound j-bound disj* **have** *ptr-val p + word-of-nat i ≠ ptr-val q + word-of-nat j*
by (*auto simp add*: *intvl-def disjnt-def*)
with *eq* **show** *False*
by (*simp add*: *i-def j-def add.commute add.left-commute*)
qed

named-theorems *select-conv* **and** *select-conv-independent* **and** *valid-conv* **and** *valid-array-conv* **and**
gen-update-commute **and** *gen-outer-update-commute* **and** *update-commute*

locale *pointer-array-lense = pointer-lense r w*
for *r*:: *'s ⇒ 'a*:: *array-outer-max-size ptr ⇒ 'a*
and *w*:: *'a ptr ⇒ ('a ⇒ 'a) ⇒ 's ⇒ 's*
begin

definition *heap-array* ::*'s ⇒ ('a['b*::*array-max-count]) ptr ⇒ 'a['b]* **where**
heap-array s p = FCP (*λi. r s* (*array-ptr-index p False i*))

lemmas [*read-stack-byte-ZERO-step-subst*] = *heap-array-def*

definition *heap-array-map* :: (*'a['b]*) *ptr ⇒* (*'a['b*::*array-max-count*] *⇒ 'a['b]*) *⇒ 's ⇒ 's* **where**
heap-array-map p f s ≡
fold (*λi. w* (*array-ptr-index p False i*) (*λ-.* (*f* (*heap-array s p*)).[*i*])) [*0..<CARD*('*b*)] *s*

lemma *element-heap-eq-heap-array-eq* [*select-conv*]: *r s = r s' ⟹ heap-array s = heap-array s'*
apply (*rule ext*)
apply (*simp add*: *heap-array-def*)
done

lemma *fold-write-independent-field*:
fixes *p*::(*'a['b*::*array-max-count*]) *ptr*
assumes *field-independent*: (*⋀p x s. f* (*w p* (*λ-. x*) *s*) = *f s*)
assumes *n-bound*: *n ≤ CARD*('*b*)
shows *f* (*fold* (*λi. w* (*array-ptr-index p False i*) (*λ-. g* (*heap-array s p*).[*i*])) [*0..<n*] *t*) = *f t*
using *n-bound*
proof (*induct n arbitrary*: *t*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case* **using** *field-independent*

by *simp*
qed

lemma *heap-array-map-independent-field* [*select-conv-independent*]:
assumes *field-independent*: $(\bigwedge p\ x\ s.\ f\ (w\ p\ (\lambda\text{-}.\ x)\ s) = f\ s)$
shows $f\ (heap\text{-}array\text{-}map\ p\ g\ s) = f\ s$
apply (*simp add*: *heap-array-map-def*)
apply (*rule fold-write-independent-field*)
apply (*rule field-independent*)
apply *simp*
done

lemma *fold-write-independent-field-upd-commute*:
fixes $p{::}('a['b{::}array\text{-}max\text{-}count])\ ptr$
assumes *write-commute*: $\bigwedge p\ x\ s.\ (f\text{-}upd\ (w\ p\ (\lambda\text{-}.\ x)\ s)) = w\ p\ (\lambda\text{-}.\ x)\ (f\text{-}upd\ s)$
assumes *n-bound*: $n \leq CARD('b)$
shows *f-upd* (*fold* $(\lambda i.\ w\ (array\text{-}ptr\text{-}index\ p\ False\ i)\ (\lambda\text{-}.\ g\ (heap\text{-}array\ s\ p).[i]))$ $[0..<n]\ t) =$
fold $(\lambda i.\ w\ \ (array\text{-}ptr\text{-}index\ p\ False\ i)\ (\lambda\text{-}.\ g\ (heap\text{-}array\ s\ p).[i]))\ [0..<n]$ (*f-upd t*)
using *n-bound*
proof (*induct n arbitrary*: *t*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case* **using** *write-commute*
by *simp*
qed

lemma *heap-array-map-independent-field-commute* [*gen-update-commute*]:
assumes *read-independent*: $\bigwedge s.\ r\ (f\text{-}upd\ s) = r\ s$
assumes *write-independent*: $\bigwedge p\ x\ s.\ (f\text{-}upd\ (w\ p\ (\lambda\text{-}.\ x)\ s)) = w\ p\ (\lambda\text{-}.\ x)\ (f\text{-}upd\ s)$
shows *f-upd* $(heap\text{-}array\text{-}map\ p\ g\ s) = (heap\text{-}array\text{-}map\ p\ g\ (f\text{-}upd\ s))$
apply (*simp add*: *heap-array-map-def element-heap-eq-heap-array-eq* [*OF read-independent*])
apply (*rule fold-write-independent-field-upd-commute*)
apply (*rule write-independent*)
apply *simp*
done

lemma *heap-array-index*[*simp*]: $i < CARD('b) \Longrightarrow$
$heap\text{-}array\ s\ (p{::}('a['b{::}array\text{-}max\text{-}count])\ ptr).[i] = r\ s\ (array\text{-}ptr\text{-}index\ p\ False\ i)$
by (*simp add*: *heap-array-def*)

lemma *read-write-same-fold-aux*:
fixes $p{::}('a['b{::}array\text{-}max\text{-}count])\ ptr$

assumes *n-bound*: $n \leq CARD\ ('b)$
assumes *i-bound*: $i < n$
shows $r\ (fold\ (\lambda i.\ w\ (array\text{-}ptr\text{-}index\ p\ False\ i)\ (\lambda\text{-}.\ f\ (heap\text{-}array\ s\ p).[i]))\ [0..<n]\ s)\ (array\text{-}ptr\text{-}index\ p\ False\ i) = f\ (heap\text{-}array\ s\ p).[i]$
using *n-bound i-bound*
proof (*induct n arbitrary*: *i s*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain** *i-bound*: $i < Suc\ n$ **and** *Suc-n-bound*: $Suc\ n \leq CARD\ ('b)$ **and**
n-bound: $n \leq CARD('b)$ **and** *n-bound'*: $n < CARD('b)$ **by** *auto*
have *size*: $CARD('b) * size\text{-}of\ TYPE('a\ ) < 2\ \widehat{}\ addr\text{-}bitsize$
using *Padding-Equivalence.array-count-size* **by** *blast*
show *?case*
proof (*cases i=n*)
case *True*
show *?thesis*
apply (*simp add*: *True*)
apply (*simp add*: *read-write-same*)
done
next
case *False*
from *False i-bound* **have** *i-bound'*: $i < n$ **by** *simp*
have *disj*: *disjnt* (*ptr-span* (*array-ptr-index p False n*)) (*ptr-span* (*array-ptr-index p False i*))
by (*rule ptr-span-array-index-disj* [*OF n-bound' i-bound'*])
show *?thesis*
apply (*simp add*: *False*)
apply (*simp add*: *read-write-other disj*)
apply (*rule Suc.hyps* [*OF n-bound i-bound'*])
done
qed
qed

lemma *array-read-write-same*: *heap-array* (*heap-array-map p f s*) $p = f$ (*heap-array s p*)
apply (*subst cart-eq*)
apply *clarsimp*
apply (*simp add*: *heap-array-map-def*)
apply (*rule read-write-same-fold-aux*)
by *simp-all*

lemma *read-write-other-fold-aux*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
fixes *p'*::('*a*['*b*::*array-max-count*]) *ptr*
assumes *n-bound*: $n \leq CARD\ ('b)$
assumes *i-bound*: $i < CARD\ ('b)$

```
  assumes disj:disjnt (ptr-span p) (ptr-span p')
   shows r (fold (λi. w (array-ptr-index p False i) (λ-. f (heap-array s p).[i]))
[0..<n] s) (array-ptr-index p' False i) =
        r s (array-ptr-index p' False i)
  using n-bound i-bound
proof (induct n arbitrary: i s)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems obtain  Suc-n-bound: Suc n ≤ CARD ('b) and
    n-bound: n ≤ CARD('b) and n-bound': n < CARD('b) and
    i-bound: i < CARD('b) by auto
  have size: CARD('b) * size-of TYPE('a ) < 2 ^ addr-bitsize
    using Padding-Equivalence.array-count-size by blast

  from ptr-span-array-ptr-index-disj [OF n-bound' i-bound disj]
  show ?case
    apply (simp add: read-write-other)
    apply (rule Suc.hyps [OF n-bound i-bound])
    done
qed

lemma array-read-write-other:
  fixes p::('a['b::array-max-count]) ptr
  fixes p'::('a['b::array-max-count]) ptr
  assumes disj:disjnt (ptr-span p) (ptr-span p')
  shows heap-array (heap-array-map p f s) p' = heap-array s p'
  apply (subst cart-eq )
  apply clarsimp
  apply (simp add: heap-array-map-def)
  apply (rule read-write-other-fold-aux [OF - - disj])
  by simp-all

lemma write-cong-fold-aux:
  fixes p::('a['b::array-max-count]) ptr
  assumes f-f': f (heap-array s p) = f' (heap-array s p)
  assumes n-bound: n ≤ CARD('b)
  shows fold (λi. w (array-ptr-index p False i) (λ-. f (heap-array s p).[i])) [0..<n]
s =
    fold (λi. w (array-ptr-index p False i) (λ-. f' (heap-array s p).[i])) [0..<n] s
  using n-bound
proof (induct n arbitrary:)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have suc-n-bound: Suc n ≤ CARD('b) using Suc.prems .
```

define s' **where** $s' = (fold\ (\lambda i.\ w\ (array\text{-}ptr\text{-}index\ p\ False\ i)\ (\lambda\text{-}.\ f'\ (heap\text{-}array\ s\ p).[i]))\ [0..<n]\ s)$

from *f-f' suc-n-bound*
have $(\lambda\text{-}.\ f\ (heap\text{-}array\ s\ p).[n]) = (\lambda\text{-}.\ f'\ (heap\text{-}array\ s\ p).[n])$
by *auto*
hence *eq*: $(\lambda\text{-}.\ f\ (heap\text{-}array\ s\ p).[n])\ (r\ s'\ (array\text{-}ptr\text{-}index\ p\ False\ n)) =$
$(\lambda\text{-}.\ f'\ (heap\text{-}array\ s\ p).[n])\ (r\ s'\ (array\text{-}ptr\text{-}index\ p\ False\ n))$ **by** *metis*
from *Suc* **show** *?case* **by** (*simp add*: *write-cong* [*OF eq*])
qed

lemma *array-write-cong*:
fixes p::$('a['b::array\text{-}max\text{-}count])\ ptr$
assumes *eq*: $f\ (heap\text{-}array\ s\ p) = f'\ (heap\text{-}array\ s\ p)$
shows $heap\text{-}array\text{-}map\ p\ f\ s = heap\text{-}array\text{-}map\ p\ f'\ s$
apply (*simp add*: *heap-array-map-def*)
apply (*rule write-cong-fold-aux*)
apply (*rule eq*)
apply *simp*
done

lemma *array-write-same-triv*[*simp*]:
fixes p::$('a['b::array\text{-}max\text{-}count])\ ptr$
shows $heap\text{-}array\text{-}map\ p\ (\lambda\text{-}.\ f\ (heap\text{-}array\ s\ p))\ s = heap\text{-}array\text{-}map\ p\ f\ s$
using *array-write-cong* **by** *fastforce*

lemma *array-write-fold-same-aux*:
fixes p::$('a['b::array\text{-}max\text{-}count])\ ptr$
assumes *f-id*: $f\ (heap\text{-}array\ s\ p) = heap\text{-}array\ s\ p$
assumes *n-bound*: $n \leq CARD('b)$
shows $fold\ (\lambda i.\ w\ (array\text{-}ptr\text{-}index\ p\ False\ i)\ (\lambda\text{-}.\ f\ (heap\text{-}array\ s\ p).[i]))\ [0..<n]\ s = s$
using *n-bound*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems f-id*
have *f-n-id*: $f\ (heap\text{-}array\ s\ p).[n] = r\ s\ (array\text{-}ptr\text{-}index\ p\ False\ n)$
by *simp*
from *Suc* **show** *?case*
by (*simp add*: *write-same f-n-id*)
qed

lemma *array-write-same*:
fixes p::$('a['b::array\text{-}max\text{-}count])\ ptr$
assumes *f-id*: $f\ (heap\text{-}array\ s\ p) = heap\text{-}array\ s\ p$

shows *heap-array-map p f s = s*
apply (*simp add*: *heap-array-map-def*)
apply (*rule array-write-fold-same-aux*)
apply (*rule f-id*)
by *simp*

lemma *array-write-triv* [*simp*]:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
shows *heap-array-map p* (λ-. *heap-array s p*) *s = s* **and** *heap-array-map p* (λ*x*. *x*) *s = s*
by (*simp-all add*: *array-write-same*)

lemma *write-fold-other-commute*:
fixes *p*::*nat* ⇒ '*a ptr*
and *q*:: '*a ptr*
assumes *disj*: ⋀*i*. *i* < *n* ⟹ *disjnt* (*ptr-span q*) (*ptr-span* (*p i*))
shows
w q f (*fold* (λ*i*. *w* (*p i*) (*g i*)) [*0*..<*n*] *s*) =
fold (λ*i*. *w* (*p i*) (*g i*)) [*0*..<*n*] (*w q f s*)
using *disj*
proof (*induct n arbitrary*: *s*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain**
q-n-disj: *disjnt* (*ptr-span q*) (*ptr-span* (*p n*)) **and**
disj': ⋀*i*. *i* < *n* ⟹ *disjnt* (*ptr-span q*) (*ptr-span* (*p i*)) **by** *auto*

note *n-commute* = *write-other-commute*[*OF q-n-disj*, *symmetric*]
show *?case*
apply *simp*
apply (*subst n-commute*)
apply (*subst Suc.hyps* [*OF disj'*])
apply (*simp-all*)
done
qed

lemma *write-fold-arr-commute*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
and *arr*:: '*a*['*b*::*array-max-count*]
assumes *n-bound*: *n* < *CARD*('*b*)
shows
w (*array-ptr-index p False n*) (λ-. *arr*.[*n*])
(*fold* (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *arr*.[*i*]))
[*0*..<*n*] *s*) =
fold

```
    (λi. w (array-ptr-index p False i) (λ-. arr.[i]))
    [0..<n] ( w (array-ptr-index p False n)  (λ-. arr.[n]) s)
  apply (rule write-fold-other-commute)
  using n-bound
  by (simp add: ptr-span-array-index-disj)



lemma array-fold-fuse-aux:
  fixes p::('a['b::array-max-count]) ptr
  fixes f::'a['b] ⇒ 'a['b]
  fixes g::'a['b] ⇒ 'a['b]
  assumes n-bound: n ≤ CARD('b)
  shows fold
     (λi. w (array-ptr-index p False i) (λ-. f (g (heap-array s p)).[i]))
     [0..<n]
     (fold
       (λi. w (array-ptr-index p False i) (λ-. g (heap-array s p).[i]))
       [0..<n] t) =
    fold
     (λi. w (array-ptr-index p False i) (λ-. f (g (heap-array s p)).[i]))
     [0..<n] t
  using n-bound
proof (induct n arbitrary: t)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems have n-bound: n ≤ CARD('b) and n-bound': n < CARD('b)
    by auto
    show ?case
      apply (simp add: )

      apply (subst (2) write-fold-arr-commute [OF n-bound'])
      apply (subst Suc.hyps [OF n-bound])
      apply (subst  write-fold-arr-commute [OF n-bound'])
      apply (subst write-comp)
      apply (subst (1) write-fold-arr-commute [OF n-bound'])
      by (meson comp-apply)
qed



lemma array-write-comp:
  fixes p::('a['b::array-max-count]) ptr
  shows heap-array-map p f (heap-array-map p g s) = heap-array-map p (f o g) s
proof −
  from  array-read-write-same [of p g s]
```

```
  have g-eq: heap-array (heap-array-map p g s) p = g (heap-array s p).
  show ?thesis
    apply (subst (1 3) heap-array-map-def)
    apply (simp add: g-eq)
    apply (subst (1) heap-array-map-def)
    apply (rule array-fold-fuse-aux)
    by simp
qed

lemma fold-fold-other-commute:
  fixes p::nat ⇒ 'a ptr
  and q:: nat ⇒ 'a ptr
assumes disj: ⋀i j. i < n ⟹ j < m ⟹ disjnt (ptr-span (p i)) (ptr-span (q j))
  shows
  fold (λi. w (p i) (f i)) [0..<n] (fold (λi. w (q i) (g i)) [0..<m] s) =
   fold (λi. w (q i) (g i)) [0..<m] (fold (λi. w (p i) (f i)) [0..<n] s)
  using disj
proof (induct n arbitrary: m s)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prems obtain
    disj': ⋀i j. i < n ⟹ j < m ⟹ disjnt (ptr-span (p i)) (ptr-span (q j)) and
    m-disj: ⋀j. j < m ⟹ disjnt (ptr-span (p n)) (ptr-span (q j))  by auto
  show ?case
    apply simp
    apply (subst  write-fold-other-commute [OF m-disj, symmetric])
    apply assumption
    apply (subst Suc.hyps [OF disj'])
      apply assumption
     apply assumption
    apply (rule refl)
    done
qed


lemma array-write-other-commute:
  fixes p::('a['b::array-max-count]) ptr
  fixes q::('a['b::array-max-count]) ptr
  assumes disj: disjnt (ptr-span p) (ptr-span q)
 shows heap-array-map q g (heap-array-map p f s) = heap-array-map p f (heap-array-map
q g s)
proof −
  from array-read-write-other [OF disj]
  have g-eq: g (heap-array (heap-array-map p f s) q) = g (heap-array s q) by simp

  from  array-read-write-other disj
  have f-eq: f (heap-array (heap-array-map q g s) p) = f (heap-array s p)
```

by (*metis Int-commute disjnt-def*)

show *?thesis*
apply (*subst* (*1 3*) *heap-array-map-def*)
apply (*simp add*: *g-eq f-eq*)
apply (*simp add*: *heap-array-map-def*)
apply (*rule fold-fold-other-commute*)
using *disj*
by (*metis Int-commute ptr-span-array-ptr-index-disj disjnt-def*)
qed

sublocale *array*: *pointer-lense heap-array heap-array-map*
apply (*unfold-locales*)
apply (*rule array-read-write-same*)
apply (*rule array-write-same*, *assumption*)
apply (*rule array-write-comp*)
apply (*rule array-write-other-commute*, *assumption*)
done

end

locale *pointer-two-dimensional-array-lense* = *pointer-array-lense r w*
for *r*:: *'s* ⇒ *'a*:: *array-inner-max-size ptr* ⇒ *'a*
and *w*:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'s* ⇒ *'s*
begin
sublocale *outer*: *pointer-array-lense heap-array heap-array-map*
by (*intro-locales*)

lemmas *outer.heap-array-map-independent-field-commute* [*gen-outer-update-commute*]

end

locale *valid-array-base* =
fixes *vgetter*:: (*'t* ⇒ *'a*::*array-outer-max-size ptr* ⇒ *bool*)
begin

definition *valid-array* ::*'t* ⇒ (*'a*[*'b*::*array-max-count*]) *ptr* ⇒ *bool* **where**
[*valid-array-defs*]: *valid-array s p* = (∀ *i* < *CARD*(*'b*). *vgetter s* (*array-ptr-index p False i*))

lemma *element-vgetter-eq-valid-array-eq* [*valid-array-conv*]: *vgetter s* = *vgetter s'*
⟹ *valid-array s* = *valid-array s'*
apply (*rule ext*)
apply (*simp add*: *valid-array-def*)
done

end

locale *valid-pointer-array-lense* = *pointer-array-lense r w* +

```
  valid-array-base vgetter +
  write-preserves-valid vgetter w
  for r:: 's ⇒ 'a:: array-outer-max-size ptr ⇒ 'a
  and w:: 'a ptr ⇒ ('a ⇒ 'a) ⇒ 's ⇒ 's
  and vgetter:: ('s ⇒ 'a ptr ⇒ bool)
begin

lemma setter-fold-keeps-vgetter:
  fixes p':: ('a['b::array-max-count]) ptr
  fixes p:: 'a ptr
  assumes n-bound: n ≤ CARD('b)
   shows  vgetter (fold (λi. w (array-ptr-index p' False i) (λ-. f (heap-array s
p').[i])) [0..<n] s) p = vgetter s p
  using n-bound
proof (induct n arbitrary: s)
  case 0
  then show ?case by (simp )
next
  case (Suc n)
  from Suc.prems obtain
    suc-n-bound: Suc n ≤ CARD('b) and
    n-bound: n ≤ CARD('b) by auto
  show ?case
    apply simp
    apply (rule Suc.hyps [OF n-bound])
    done
qed

lemma heap-array-map-keeps-vgetter:
  fixes p:: ('a['b::array-max-count]) ptr
  fixes p':: ('a['b]) ptr
  shows valid-array (heap-array-map p' f s) p = valid-array s p
 by (clarsimp simp add: valid-array-def heap-array-map-def setter-fold-keeps-vgetter)

sublocale array: write-preserves-valid valid-array heap-array-map
  apply (unfold-locales)
  apply (rule heap-array-map-keeps-vgetter)
  done

lemma heap-array-map-keeps-vgetter-element[simp]:
  fixes p':: ('a['b::array-max-count]) ptr
  shows vgetter (heap-array-map p' f s) = vgetter s
  apply (rule ext)
  by (simp add: heap-array-map-def setter-fold-keeps-vgetter)

lemma getter-keeps-valid-array[simp]:
  fixes p':: 'a ptr
   shows (valid-array::'s ⇒ ('a['b::array-max-count]) ptr ⇒ bool) (w p' f s) =
valid-array s
```

apply (*rule ext*)
by (*simp add*: *valid-array-def*)

lemma *getter-keeps-valid-array-pointwise*[]:
fixes *p'*:: *'a ptr*
fixes *p* :: (*'a*['*b*::*array-max-count*]) *ptr*
shows (*valid-array*::*'s* ⇒ (*'a*['*b*::*array-max-count*]) *ptr* ⇒ *bool*) (*w p' f s*) *p* = *valid-array s p*
by (*simp add*: *valid-array-def*)
end

locale *valid-pointer-two-dimensional-array-lense* = *pointer-two-dimensional-array-lense r w* +
valid-array-base vgetter +
write-preserves-valid vgetter w
for *r*:: *'s* ⇒ *'a*:: *array-inner-max-size ptr* ⇒ *'a*
and *w*:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'s* ⇒ *'s*
and *vgetter*:: (*'s* ⇒ *'a ptr* ⇒ *bool*)
begin
sublocale *inner*: *valid-pointer-array-lense r w vgetter*
by (*intro-locales*)

sublocale *outer*: *valid-pointer-array-lense heap-array heap-array-map valid-array*
by (*intro-locales*)

end

locale *array-typ-heap-simulation* =
lense t-hrs t-hrs-update +
read-simulation st v r t-hrs +
write-simulation t-hrs t-hrs-update st v w +
write-preserves-valid v w +
valid-implies-cguard st v +
valid-only-typ-desc-dependent t-hrs st v +
pointer-array-lense r w +
valid-array-base v
for
st:: *'s* ⇒ *'t* **and**
r:: *'t* ⇒ *'a*::*array-outer-max-size ptr* ⇒ *'a* **and**
w:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'t* ⇒ *'t* **and**
v:: *'t* ⇒ *'a ptr* ⇒ *bool* **and**
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s*
begin

sublocale *typ-heap-simulation*
by (*intro-locales*)

sublocale *valid-pointer-array-lense r w v*
by (*intro-locales*)

lemmas *typ-heap-simulation = typ-heap-simulation-axioms*

lemma *valid-array-implies-safe*: *valid-array* (*st s*) *p* ⟹ *c-guard p*
using *valid-implies-c-guard*
apply (*simp add*: *valid-array-def*)
by (*metis TypHeapSimple.c-guard-array-c-guard array-ptr-index-simps*(*1*))

lemma *heap-array-data-correct*:
assumes *valid-arr*: *valid-array* (*st s*) *p*
shows *heap-array* (*st s*) *p* = *h-val* (*hrs-mem* (*t-hrs s*)) *p*
apply (*subst cart-eq*)
apply *clarsimp*
apply (*subst heap-access-Array-element'*)
apply *simp*
using *read-commutes valid-arr*
by (*auto simp add*: *valid-array-def*)

lemma *write-fold-commutes*:
fixes *p*:: ('*a*['*b*::*array-max-count*]) *ptr*
assumes *n-bound*: *n* ≤ *CARD*('*b*)
assumes *valid*: *valid-array* (*st s*) *p*
shows *st* (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*array-ptr-index p False i*) (*x*.[*i*])) [*0*..<*n*])) *s*) =
fold (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *x*.[*i*])) [*0*..<*n*] (*st s*)
using *n-bound valid*
proof (*induct n arbitrary*: *s*)
case *0*
then show *?case*
by *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain** *suc-n-bound*: *Suc n* ≤ *CARD*('*b*) **and** *n-bound*: *n* ≤ *CARD*('*b*) **and** *n-bound'*: *n* < *CARD*('*b*)
and *valid*: *valid-array* (*st s*) *p* **by** *auto*

have *vgetter*: *v* (*st* (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*array-ptr-index p False i*) (*x*.[*i*])) [*0*..<*n*])) *s*))
(*array-ptr-index p False n*)
apply (*subst Suc.hyps* [*OF n-bound valid*])
apply (*subst setter-fold-keeps-vgetter* [*OF n-bound*])
using *valid n-bound'* **apply** (*simp add*: *valid-array-def*)
done
show *?case*
apply *simp*
apply (*subst Suc.hyps*[*OF n-bound valid*, *symmetric*])

```
    apply (subst write-commutes[symmetric])
     apply (rule vgetter)
    by (simp add: hrs-mem-update-def split-def comp-def)
qed

lemma heap-array-map-commutes:
  fixes p:: ('a['b::array-max-count]) ptr
  assumes valid: valid-array (st s) p
  shows st (t-hrs-update (hrs-mem-update (heap-update p x)) s) = heap-array-map
p (λ-. x) (st s)
proof −
  from valid-array-implies-safe [OF valid] have cgrd: c-guard p .
  show ?thesis
    apply (simp add: heap-array-map-def heap-update-array-pointless [OF cgrd])
    apply (rule write-fold-commutes)
    apply simp
    apply (rule valid)
    done
qed


lemma write-padding-fold-commutes:
  fixes p:: ('a['b::array-max-count]) ptr
  assumes n-bound: n ≤ CARD('b)
  assumes valid: valid-array (st s) p
  assumes lbs: length bs = CARD('b) * size-of TYPE('a)
  shows st (t-hrs-update
        (hrs-mem-update
          (fold
            (λi. heap-update-padding (array-ptr-index p False i) (x.[i])
                  (take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs)))
            [0..<n]))
        s) =
    fold (λi. w (array-ptr-index p False i) (λ-. x.[i])) [0..<n] (st s)
  using n-bound valid
proof (induct n arbitrary: s)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  from Suc.prems obtain  suc-n-bound: Suc n ≤ CARD('b) and n-bound: n ≤
CARD('b) and n-bound': n < CARD('b)
    and valid: valid-array (st s) p by auto

  from lbs suc-n-bound
  have lbs': length (take (size-of TYPE('a)) (drop (n * size-of TYPE('a)) bs)) =
size-of TYPE('a)
    by (auto simp add: less-le-mult-nat nat-move-sub-le)
```

```
  have vgetter: v (st (t-hrs-update
          (hrs-mem-update
            (λh. fold (λi. heap-update-padding (array-ptr-index p False i) (x.[i])
                          (take (size-of TYPE('a)) (drop (i * size-of TYPE('a))
bs)))
                  [0..<n] h))
          s))
    (array-ptr-index p False n)
    apply (subst Suc.hyps [OF n-bound valid] )

    apply (subst setter-fold-keeps-vgetter [OF n-bound])
    using valid n-bound' apply (simp add: valid-array-def)
    done

  show ?case apply simp
    apply (subst Suc.hyps[OF n-bound valid, symmetric])

     apply (subst write-padding-commutes[where bs = ‹take (size-of TYPE('a))
(drop (n * size-of TYPE('a)) bs)›, OF vgetter lbs', symmetric])
    apply simp
    apply (simp only: hrs-mem-update-comp)
    apply (simp only: comp-def)
    done
qed


lemma heap-array-padding-map-commutes:
  fixes p:: ('a['b::array-max-count]) ptr
  assumes valid: valid-array (st s) p
  assumes bound: length bs = size-of TYPE('a['b])
   shows st (t-hrs-update (hrs-mem-update (heap-update-padding p x bs)) s) =
heap-array-map p (λ-. x) (st s)
proof −
  from valid-array-implies-safe [OF valid] have cgrd: c-guard p .
  from bound have bound': length bs = CARD('b) * size-of TYPE('a)
    by auto
  show ?thesis
    apply (simp add: heap-array-map-def heap-update-padding-array-pointless [OF
cgrd bound'])
    apply (simp add: write-padding-fold-commutes [OF - valid bound'])
    done
qed


lemma array-valid-same-typ-desc:
  fixes p:: ('a['b::array-max-count]) ptr
  assumes typ-eq: hrs-htd (t-hrs s) = hrs-htd (t-hrs t)
```

shows *valid-array* (*st s*) *p* = *valid-array* (*st t*) *p*
apply (*simp add*: *valid-array-def*)
using *typ-eq valid-same-typ-desc* **by** *blast*

sublocale *array*: *typ-heap-simulation st heap-array heap-array-map valid-array t-hrs t-hrs-update*
apply (*unfold-locales*)
apply (*rule heap-array-data-correct*, *assumption*)
apply (*rule heap-array-padding-map-commutes*, *assumption*, *assumption*)
apply (*rule valid-array-implies-safe*, *assumption*)
apply (*rule array-valid-same-typ-desc*, *assumption*)
done

end

locale *two-dimensional-array-typ-heap-simulation* =
lense t-hrs t-hrs-update +
read-simulation st v r t-hrs +
write-simulation t-hrs t-hrs-update st v w +
write-preserves-valid v w +
valid-implies-cguard st v +
valid-only-typ-desc-dependent t-hrs st v +
pointer-two-dimensional-array-lense r w
for
st:: *'s* ⇒ *'t* **and**
r:: *'t* ⇒ *'a*::*array-inner-max-size ptr* ⇒ *'a* **and**
w:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'t* ⇒ *'t* **and**
v:: *'t* ⇒ *'a ptr* ⇒ *bool* **and**
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s*
begin

sublocale *inner*: *array-typ-heap-simulation st r w v t-hrs t-hrs-update*
by (*intro-locales*)

lemmas *inner-typ-heap-simulation* = *inner.typ-heap-simulation*

sublocale *outer*: *array-typ-heap-simulation st heap-array heap-array-map inner.valid-array t-hrs t-hrs-update*
by (*intro-locales*)

lemmas *outer-typ-heap-simulation* = *outer.typ-heap-simulation*

end

lemma *root-ptr-valid-field-lvalue*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*

assumes *root-p*: *root-ptr-valid d p*
assumes *root-q*: *root-ptr-valid d q*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*b*)) *f 0* = *Some* (*t*, *k*)
assumes *other*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
shows *ptr-val p* = &(*q*→*f*) ⟷ *False*
proof
assume *p*: *ptr-val p* = &(*q*→*f*)
from *field-tag-sub* [*OF fl*] **have** {&(*q*→*f*)..+*size-td t*} ⊆ *ptr-span q*.
moreover
from *root-ptr-valid-cases* [*OF root-p root-q*] *other* **have** *ptr-span p* ∩ *ptr-span q* = {} **by** *blast*
ultimately
show *False*
using *p*
by (*metis* (*mono-tags*, *lifting*) *disjoint-iff field-lookup-wf-size-desc-gt fl intvl-inter intvl-start-inter mem-type-self subset-iff wf-size-desc*)
qed *simp*

lemma *root-ptr-valid-field-lvalue'*:
fixes *q*:: '*b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d* (*PTR* ('*a*::*mem-type*)(&(*q*→*f*)))
assumes *root-q*: *root-ptr-valid d q*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*b*)) *f 0* = *Some* (*t*, *k*)
assumes *other*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
shows *False*
using *root-ptr-valid-field-lvalue* [*OF root-p root-q fl other*]
by *simp*

lemma *root-ptr-valid-array-ptr-index-dim1*:
fixes *q*:: ('*a*::*array-outer-max-size*['*c*::*array-max-count*]) *ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index q False i*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*a*['*c*])
assumes *i-bound*: *i* < *CARD*('*c*)
shows *False*
proof −
from *field-lookup-array* [*OF i-bound*]
obtain *t k* **where**
fl: *field-lookup* (*typ-info-t TYPE*('*a*['*c*])) [*replicate i* (*CHR* ''*1*'')] *0* = *Some* (*t*, *k*)
by *blast*
from *root-ptr-valid-field-lvalue* [*OF root-p root-q fl other*] *i-bound*
show *?thesis*
by (*simp add*: *array-ptr-index-field-lvalue-conv*)
qed

lemma *root-ptr-valid-field-lvalue-array-ptr-index-dim1*:

fixes *q*:: *'b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index* (*PTR*(*'a*[*'c*]) &(*q*→*f*)) *False i*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
assumes *i-bound*: *i* < *CARD*(*'c*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*t*, *k*)
assumes *t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*(*'a*::*array-outer-max-size*[*'c*::*array-max-count*]))
shows *False*
proof −
from *field-lookup-array* [*OF i-bound*]
obtain *s n* **where**
fl-i: *field-lookup* (*typ-info-t TYPE*(*'a*[*'c*])) [*replicate i* (*CHR ''1''*)] *k* = *Some* (*s*, *n*)
by *blast*

from *fl-i t* **obtain** *s' n'* **where**
fl-i': *field-lookup t* [*replicate i* (*CHR ''1''*)] *k* = *Some* (*s'*, *n'*)
by (*metis* (*no-types*, *lifting*) *Padding-Equivalence.field-lookup-export-uinfo-Some-rev*
field-lookup-export-uinfo-Some fold-typ-uinfo-t)

from *field-lookup-prefix-Some''*(*1*)[*rule-format*, **where** *f*=*f* **and** *g*=[*replicate i CHR ''1''*]
and *m*=*0*, *OF fl*]
have *fl-app*: *field-lookup* (*typ-info-t TYPE*(*'b*)) (*f* @ [*replicate i CHR ''1''*]) *0* = *Some* (*s'*, *n'*)
by (*simp add*: *fl-update fl-i'*)
have *conv*: &(*PTR*(*'a*[*'c*]) &(*q*→*f*)→[*replicate i CHR ''1''*]) = &(*q*→*f* @[*replicate i CHR ''1''*])
apply (*subst field-lvalue-append*)
apply *simp-all*
apply (*rule field-lookup-field-ti'*)
apply (*rule fl*)
apply (*simp add*: *typ-uinfo-t-def i-bound t*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-array* [*OF i-bound*])
done

from *root-ptr-valid-field-lvalue* [*OF root-p root-q fl-app other*] **show** *False*
by (*simp add*: *array-ptr-index-field-lvalue-conv i-bound conv*)
qed

lemma *root-ptr-valid-field-lvalue-array-ptr-index-dim1'*:
fixes *q*:: *'b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index* (*PTR*((*'a*::*array-outer-max-size*[*'c*::*array-max-count*])) &(*q*→*f*)) *False i*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
assumes *i-bound*: *i* < *CARD*(*'c*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t*

(*TYPE*('*a*['*c*]))) *acc upd*, *k*)
assumes *fg*: *fg-cons acc upd*
shows *False*
using *root-ptr-valid-field-lvalue-array-ptr-index-dim1* [*OF root-p root-q other i-bound fl*]
by (*simp add*: *fg*)

lemma *root-ptr-valid-array-ptr-index-dim2*:
fixes *q*:: (('*a*::*array-inner-max-size*['*c*::*array-max-count*])['*d*::*array-max-count*]) *ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index* (*array-ptr-index q False i*) *False j*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*a*['*c*]['*d*])
assumes *i-bound*: *i* < *CARD*('*d*)
assumes *j-bound*: *j* < *CARD*('*c*)
shows *False*
proof −
from *field-lookup-array* [*OF i-bound*, *of 0*, **where** *'a='a*['*c*]]
have *fl-i*: *field-lookup* (*typ-info-t TYPE*(('*a*['*c*])['*d*])) [*replicate i CHR ''1''*] *0* =
Some (*adjust-ti* (*typ-info-t TYPE*('*a*['*c*])) (λ*x*. *x*.[*i*]) (λ*x f*. *Arrays.update f i x*),
i ∗ *size-of TYPE*('*a*['*c*])) **by** *simp*
from *field-lookup-array* [*OF j-bound*, *of i* ∗ *size-of TYPE*('*a*['*c*]), **where** *'a='a*]
have *fl-j*: *field-lookup* (*typ-info-t TYPE*('*a*['*c*])) [*replicate j CHR ''1''*] (*i* ∗ *size-of TYPE*('*a*['*c*])) =
Some (*adjust-ti* (*typ-info-t TYPE*('*a*)) (λ*x*. *x*.[*j*]) (λ*x f*. *Arrays.update f j x*),
i ∗ *size-of TYPE*('*a*['*c*]) + *j* ∗ *size-of TYPE*('*a*)) **by** *simp*

have *append*: *&*(*PTR*('*a*['*c*]) *&*(*q*→[*replicate i CHR ''1''*])→[*replicate j CHR ''1''*]) =
&(*q*→[*replicate i CHR ''1''*, *replicate j CHR ''1''*])
apply (*subst field-lvalue-append*)
apply *simp-all*
apply (*rule field-lookup-field-ti'*)
apply (*rule fl-i*)
apply (*simp add*: *typ-uinfo-t-def i-bound*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-array* [*OF j-bound*])
done

have *sz-eq*: *size-of TYPE*('*a*['*c*]) = (*CARD*('*c*) ∗ *size-of TYPE*('*a*))
using *size-of-array* **by** *blast*
from *field-lookup-prefix-Some''*(*1*)[*rule-format*, **where** *f*=[*replicate i CHR ''1''*] **and** *g*=[*replicate j CHR ''1''*]
and *t* = (*typ-info-t TYPE*(('*a*['*c*])['*d*])) **and** *m=0*, *OF fl-i*]
obtain *t k*
where *fl*: *field-lookup* (*typ-info-t TYPE*('*a*['*c*]['*d*])) [*replicate i CHR ''1''*, *replicate j CHR ''1''*] *0* = *Some* (*t*, *k*)

apply (*simp add*: *i-bound*)
apply (*simp add*: *fl-update sz-eq*)
using *fl-j* [*simplified sz-eq*]
apply *simp*
done
show *?thesis*
using *root-ptr-valid-field-lvalue* [*OF root-p root-q fl other*] *i-bound j-bound*
apply (*simp add*: *array-ptr-index-field-lvalue-conv append*)
done
qed

lemma *root-ptr-valid-field-lvalue-array-ptr-index-dim2*:
fixes *q*:: *'b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index* (*array-ptr-index* (*PTR*(*'a*[*'c*][*'d*]) &(*q*→*f*)) *False i*) *False j*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
assumes *i-bound*: *i* < *CARD*(*'d*)
assumes *j-bound*: *j* < *CARD*(*'c*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*t*, *k*)
assumes *t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*((*'a*::*array-inner-max-size*[*'c*::*array-max-count*])[*'d*::*array-max*
shows *False*
proof −
from *field-lookup-array* [*OF i-bound*]
obtain *s n* **where**
fl-i: *field-lookup* (*typ-info-t TYPE*(*'a*[*'c*][*'d*])) [*replicate i* (*CHR ''1''*)] *k* = *Some* (*s*, *n*) **and**
s: *s* = *adjust-ti* (*typ-info-t TYPE*(*'a*[*'c*])) (λ*x*. *x*.[*i*]) (λ*x f*. *Arrays.update f i x*)
by *blast*

from *fl-i t s* **obtain** *s' n'* **where**
fl-i': *field-lookup t* [*replicate i* (*CHR ''1''*)] *k* = *Some* (*s'*, *n'*) **and**
s': *export-uinfo s'* = *typ-uinfo-t TYPE*(*'a*[*'c*])
by (*smt* (*verit*, *best*) *Padding-Equivalence.field-lookup-export-uinfo-Some-rev*
export-tag-adjust-ti(*1*) *fg-cons-array field-lookup-array fold-typ-uinfo-t i-bound wf-fd field-lookup-typ-uinfo-t-Some*)

from *field-lookup-prefix-Some''*(*1*)[*rule-format*, **where** *f*=*f* **and** *g*=[*replicate i CHR ''1''*]
and *m*=*0*, *OF fl*]
have *fl-app1*: *field-lookup* (*typ-info-t TYPE*(*'b*)) (*f* @ [*replicate i CHR ''1''*]) *0* = *Some* (*s'*, *n'*)
by (*simp add*: *fl-update fl-i'*)

from *field-lookup-array* [*OF j-bound*]
obtain *s'' n''* **where**
fl-j: *field-lookup* (*typ-info-t TYPE*(*'a*[*'c*])) [*replicate j* (*CHR ''1''*)] *n'* = *Some* (*s''*, *n''*)
by *blast*

from *fl-j s'* **obtain** *s''' n'''* **where**
fl-j': *field-lookup s' [replicate j (CHR ''1'')] n' = Some (s''', n''')*
by (*metis* (*no-types*, *lifting*) *CTypes.field-lookup-export-uinfo-Some-rev field-lookup-export-uinfo-Some fold-typ-uinfo-t*)

from *field-lookup-prefix-Some''*(*1*)[*rule-format*, **where** *f=f* @ [*replicate i CHR ''1''*] **and** *g=*[*replicate j CHR ''1''*]
and *m=0*, *OF fl-app1*]
have *fl-app2*: *field-lookup* (*typ-info-t TYPE*('*b*)) (*f* @ [*replicate i CHR ''1''*, *replicate j CHR ''1''*]) *0 = Some* (*s'''*, *n'''*)
by (*simp add*: *fl-update fl-j'*)

have *conv*: &(*PTR*('*a*['*c*]) &(*PTR*('*a*['*c*]['*d*]) &(*q*→*f*)→[*replicate i CHR ''1''*]→[*replicate j CHR ''1''*]) =
&(*q*→*f* @ [*replicate i CHR ''1''*, *replicate j CHR ''1''*])
apply (*subst field-lvalue-append*)
apply (*rule field-lookup-field-ti'*)
apply (*rule fl*)
apply (*simp add*: *typ-uinfo-t-def i-bound t*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-offset-shift'*)
apply (*rule fl-i*)

apply (*subst field-lvalue-append*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-offset-shift'*)
apply (*rule fl-app1*)
apply (*simp add*: *s'*)
apply (*rule field-lookup-field-ti'*)
apply (*rule field-lookup-offset-shift'*)
apply (*rule fl-j*)
apply *simp*
done
from *root-ptr-valid-field-lvalue* [*OF root-p root-q fl-app2 other*] **show** *False*
by (*simp add*: *array-ptr-index-field-lvalue-conv i-bound j-bound conv*)
qed

context *open-types*
begin

definition
typ-heap-simulation-of-field (*st*::'*s* ⇒ '*t*) *t-hrs t-hrs-update heap-typing-upd f' r' w'* ⟷
(∀ *d p f*. *heap-typing-upd d o w' p f = w' p f o heap-typing-upd d*) ∧
(∀ *t u n*.
field-ti TYPE('*a*::*mem-type*) *f' = Some t* ⟶
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f' 0 = Some* (*u*, *n*) ⟶
partial-pointer-lense (*merge-ti t*) *r' w'* ∧

(∀ (p::'a ptr) s. (∀ a∈ptr-span p. root-ptr-valid (hrs-htd (t-hrs s)) (PTR(stack-byte) a)) ⟶
r' (st s) p ZERO('a) = ZERO('a)) ∧
(∀ (p::'a ptr) x h. ptr-valid-u u (hrs-htd (t-hrs h)) &(p→f') ⟶
st (t-hrs-update (hrs-mem-update (heap-upd-list (size-td u) &(p→f') (access-ti t x))) h) =
w' p x (st h)))

end

definition
pointer-writer-disjnt ::
('a::c-type ptr ⇒ 't upd) ⇒ ('b::c-type ptr ⇒ 't upd) ⇒ bool
where
pointer-writer-disjnt f g ⟷
(∀ p q. disjnt (ptr-span p) (ptr-span q) ⟶ f p ∘ g q = g q o f p)

definition
pointer-writer-disjnt-eq ::
('a::c-type ptr ⇒ 'x ⇒ 't upd) ⇒ ('b::c-type ptr ⇒ 'y ⇒ 't upd) ⇒ bool
where
pointer-writer-disjnt-eq f g ⟷ (∀ p q x y.
disjnt (ptr-span p) (ptr-span q) ∨ ptr-val p = ptr-val q ⟶ f p x ∘ g q y = g q y o f p x)

named-theorems *pointer-writer-disjnt-intros*

lemma *pointer-writer-disjnt-eq*:
assumes *nm*: ∃ n1 n2.
field-lookup (typ-uinfo-t TYPE('a::mem-type)) f1 0 =
Some (typ-uinfo-t TYPE('b::mem-type), n1) ∧
field-lookup (typ-uinfo-t TYPE('a)) f2 0 = Some (typ-uinfo-t TYPE('c::mem-type), n2)
assumes *w1-w2*: ⋀x y. pointer-writer-disjnt (λp. w1 p x) (λq. w2 q y)
shows *disj-fn f1 f2* ⟶ *pointer-writer-disjnt-eq*
(λ(p::'a ptr). w1 (PTR('b) &(p→f1)))
(λ(p::'a ptr). w2 (PTR('c) &(p→f2))) (**is** *?disj* ⟶ *?goal*)
proof
assume *?disj*
from *nm* **obtain** *n1 n2* **where** *fl*:
field-lookup (typ-uinfo-t TYPE('a::mem-type)) f1 0 =
Some (typ-uinfo-t TYPE('b::mem-type), n1)
field-lookup (typ-uinfo-t TYPE('a)) f2 0 = Some (typ-uinfo-t TYPE('c::mem-type), n2)
by *auto*
from *fl*[*THEN field-tag-sub'*]
have *ptr-span* (PTR('b) &(p→f1)) ⊆ *ptr-span p* ∧ *ptr-span* (PTR('c) &(q→f2)) ⊆ *ptr-span q*
for *p q* :: 'a *ptr*

by (*auto intro*: *field-tag-sub'* *simp*: *size-of-def del*: *subsetI*)
then have *ne*[*simp*]: *disjnt* (*ptr-span p*) (*ptr-span q*) ⟹
disjnt (*ptr-span* (*PTR*('*b*) &(*p*→*f1*))) (*ptr-span* (*PTR*('*c*) &(*q*→*f2*))) **for** *p q* :: '*a ptr*
by (*blast intro*: *disjnt-subset1 disjnt-subset2*)
moreover
from *fl* **obtain** *u1 u2* **where** *fl-t*:
field-lookup (*typ-info-t TYPE*('*a*)) *f1 0* = *Some* (*u1*, *n1*)
field-lookup (*typ-info-t TYPE*('*a*)) *f2 0* = *Some* (*u2*, *n2*)
and *export*: *export-uinfo u1* = *typ-uinfo-t TYPE*('*b*) *export-uinfo u2* = *typ-uinfo-t TYPE*('*c*)
by (*auto dest*!: *field-lookup-export-uinfo-Some-rev simp*: *typ-uinfo-t-def*)
from *fa-fu-lookup-disj-interD*[*OF fl-t* ‹*disj-fn f1 f2*›]
have *disjnt* (*ptr-span* (*PTR*('*b*) &(*p*→*f1*))) (*ptr-span* (*PTR*('*c*) &(*p*→*f2*))) **for** *p* :: '*a ptr*
apply (*simp add*: *size-of-fold disjnt-def*)
apply (*simp add*: *size-of-def field-lvalue-def field-offset-def field-offset-untyped-def fl*
intvl-disj-offset
flip: *typ-uinfo-size export*)
done
ultimately show *?goal* **using** *w1-w2*
by (*auto simp*: *pointer-writer-disjnt-def pointer-writer-disjnt-eq-def*)
qed

lemma *pointer-writer-disjnt-sym*:
pointer-writer-disjnt f g ⟹ *pointer-writer-disjnt g f*
by (*auto simp add*: *pointer-writer-disjnt-def disjnt-sym*)

lemma *pointer-writer-disjnt-id-left*:
pointer-writer-disjnt (λ*p h*. *h*) *w*
by (*simp add*: *pointer-writer-disjnt-def fun-eq-iff*)

lemma *pointer-writer-disjnt-id-left'*:
pointer-writer-disjnt (λ*p*. *id*) *w*
by (*simp add*: *pointer-writer-disjnt-def fun-eq-iff*)

lemma *pointer-writer-disjnt-comp-left*:
pointer-writer-disjnt w1 w ⟹ *pointer-writer-disjnt w2 w* ⟹
pointer-writer-disjnt (λ*p h*. *w1 p* (*w2 p h*)) *w*
by (*simp add*: *pointer-writer-disjnt-def fun-eq-iff*)

lemma *pointer-writer-disjnt-comp-left'*:
pointer-writer-disjnt w1 w ⟹ *pointer-writer-disjnt w2 w* ⟹
pointer-writer-disjnt (λ*p*. *w1 p* ∘ *w2 p*) *w*
by (*simp add*: *pointer-writer-disjnt-def fun-eq-iff*)

lemma *pointer-writer-disjnt-fold-left*:
list-all (λ*w'*. *pointer-writer-disjnt* (*f w'*) *w*) *ws* ⟹

pointer-writer-disjnt (λ*p*. *fold* (λ*w*. *f w p*) *ws*) *w*
by (*induction ws*) (*auto intro*: *pointer-writer-disjnt-comp-left' pointer-writer-disjnt-id-left'*)

lemma *pointer-writer-disjntI*:
(⋀*p q h*. *w1 p* (*w2 q h*) = *w2 q* (*w1 p h*)) ⟹ *pointer-writer-disjnt w1 w2*
by (*auto simp*: *pointer-writer-disjnt-def*)

lemma *pointer-writer-disjntD*:
pointer-writer-disjnt w1 w2 ⟹ *disjnt* (*ptr-span p*) (*ptr-span q*) ⟹
w1 p (*w2 q h*) = *w2 q* (*w1 p h*)
by (*auto simp*: *pointer-writer-disjnt-def fun-eq-iff*)

lemma *pointer-writer-disjnt-ptr-map-left*:
fixes *w1* :: *'a*::*mem-type ptr* ⇒ *'s upd* **and** *w2* :: *'b*::*c-type ptr* ⇒ *'s upd*
assumes *w1-w2*: *pointer-writer-disjnt w1 w2*
assumes ⋀(*p*::*'c*::*c-type ptr*). *ptr-span* (*F p*) ⊆ *ptr-span p*
shows *pointer-writer-disjnt* (λ(*p*::*'c ptr*). *w1* (*F p*)) *w2*
using *assms* **by** (*auto simp*: *pointer-writer-disjnt-def disjnt-subset1*)

lemma *pointer-writer-disjnt-ptr-left*:
fixes *w1* :: *'a*::*mem-type ptr* ⇒ *'s upd* **and** *w2* :: *'b*::*mem-type ptr* ⇒ *'s upd*
assumes *w1-w2*: *pointer-writer-disjnt w1 w2*
assumes *n*:
∃ *n*. *field-lookup* (*typ-uinfo-t TYPE*(*'c*::*mem-type*)) *f 0* = *Some* (*typ-uinfo-t TYPE*(*'a*), *n*)
shows *pointer-writer-disjnt* (λ(*p*::*'c ptr*). *w1* (*PTR*(*'a*) &(*p*→*f*))) *w2*
proof (*rule pointer-writer-disjnt-ptr-map-left*[*OF w1-w2*])
show *ptr-span* (*PTR*(*'a*) &(*p*→*f*)) ⊆ *ptr-span p* **for** *p* :: *'c ptr*
using *field-tag-sub'*[**where** *'a*=*'c*, *of f typ-uinfo-t TYPE*(*'a*) - *p*] *n*
by (*auto simp*: *size-of-def*)
qed

lemma *pointer-writer-disjnt-ptr-corce*:
fixes *w1* :: *'a*::*mem-type ptr* ⇒ *'s upd* **and** *w2* :: *'b*::*mem-type ptr* ⇒ *'s upd*
assumes *w1-w2*: *pointer-writer-disjnt w1 w2*
and *size*: *size-of TYPE*(*'a*) ≤ *size-of TYPE*(*'c*)
shows *pointer-writer-disjnt* (λ(*p*::*'c*::*c-type ptr*). *w1* (*PTR-COERCE*(*'c* → *'a*) *p*)) *w2*
by (*rule pointer-writer-disjnt-ptr-map-left*[*OF w1-w2*])
(*simp add*: *intvl-start-le size*)

lemma *pointer-writer-disjnt-ptr-corce-signed*:
fixes *w1* :: *'a*::*len8 word ptr* ⇒ *'s upd* **and** *w2* :: *'b*::*mem-type ptr* ⇒ *'s upd*
assumes *w1-w2*: *pointer-writer-disjnt w1 w2*
shows *pointer-writer-disjnt* (λ(*p*::*'a signed word ptr*).
w1 (*PTR-COERCE*(*'a signed word* → *'a word*) *p*)) *w2*
using *w1-w2* **by** (*rule pointer-writer-disjnt-ptr-corce*) (*simp add*: *size-of-signed-word*)

lemma *pointer-writer-disjnt-ptr-corce-signed'*:

fixes *w1* :: *'a*::*len8 word ptr* ⇒ *'s upd* **and** *w2* :: *'b*::*mem-type ptr* ⇒ *'s upd*
shows *pointer-writer-disjnt w2 w1* ⟹
pointer-writer-disjnt w2 (λ(*p*::*'a signed word ptr*).
w1 (*PTR-COERCE*(*'a signed word* → *'a word*) *p*))
using *pointer-writer-disjnt-ptr-corce-signed*[*of w1 w2*] **by** (*simp add*: *pointer-writer-disjnt-sym*)

lemma (**in** *pointer-lense*) *pointer-writer-disjnt-replace-by-const*:
(⋀*x*. *pointer-writer-disjnt* (λ*p*. *w p* (λ-. *x*)) *w'*) ⟹ *pointer-writer-disjnt* (λ*p*. *w p f*) *w'*
by (*auto simp add*: *pointer-writer-disjnt-def fun-eq-iff*)
(*metis* (*no-types*, *lifting*) *read-write-same write-cong write-read*)

lemma (**in** *pointer-lense*) *read-commute-of-pointer-writer-disjnt*:
assumes *w'*: ⋀*f*. *pointer-writer-disjnt* (λ*p*. *w p f*) *w'*
assumes *p-q*: *disjnt* (*ptr-span p*) (*ptr-span q*)
shows *r* (*w' q h*) *p* = *r h p*
apply (*subst write-same*[*of* λ-. *r h p h p*, *OF refl*, *symmetric*])
apply (*subst pointer-writer-disjntD*[*OF w' p-q*, *symmetric*])
apply (*subst read-write-same*)
apply (*rule refl*)
done

lemma (**in** *pointer-lense*) *read-commute-of-pointer-writer-disjnt'*:
assumes *w'*: ⋀*f*. *pointer-writer-disjnt w'* (λ*p*. *w p f*)
assumes *p-q*: *disjnt* (*ptr-span p*) (*ptr-span q*)
shows *r* (*w' q h*) *p* = *r h p*
using *read-commute-of-pointer-writer-disjnt*[*OF w'*[*THEN pointer-writer-disjnt-sym*] *p-q*] .

lemma (**in** *pointer-lense*) *read-commute-of-commute*:
assumes *w'*: ⋀*p f*. *w p f o w'* = *w' o w p f*
shows *r* (*w' h*) *p* = *r h p*
using *read-write-same*[*of p* λ-. *r h p w' h*] *w'*
by (*subst write-same*[*of* λ-. *r h p h p*, *OF refl*, *symmetric*]) (*simp add*: *fun-eq-iff*)

lemma (**in** *pointer-lense*) *read-commute-of-commute'*:
assumes *w'*: ⋀*p f*. *w' o w p f* = *w p f o w'*
shows *r* (*w' h*) *p* = *r h p*
using *read-commute-of-commute*[*OF w'*[*symmetric*]] .

lemma (**in** *lense*) *get-commute-of-commute*:
assumes *w'*: ⋀*f*. *w' o upd f* = *upd f o w'*
shows *get* (*w' h*) = *get h*
using *get-upd*[*of* λ-. *get h w' h*] *w'*[*symmetric*]
by (*subst upd-same*[*of* λ-. *get h h*, *OF refl*, *symmetric*]) (*simp add*: *fun-eq-iff del*: *get-upd*)

lemma (**in** *pointer-lense*) *pointer-writer-disjnt-replace-dep-by-const*:
assumes ∗: ⋀*x*. *pointer-writer-disjnt* (λ*p*. *w p x*) *w'*

shows *pointer-writer-disjnt* $(\lambda p\ s.\ w\ p\ (f\ (r\ s\ p))\ s)\ w'$
by (*auto simp add*: *pointer-writer-disjnt-def fun-eq-iff*
read-commute-of-pointer-writer-disjnt[*OF* $*$]
pointer-writer-disjntD[*OF* $*$])

lemma (**in** *pointer-lense*) *pointer-writer-disjnt-upd*[*pointer-writer-disjnt-intros*]:
pointer-writer-disjnt $(\lambda p.\ w\ p\ x)\ (\lambda p.\ w\ p\ y)$
by (*simp add*: *pointer-writer-disjnt-def write-other-commute disjnt-def fun-eq-iff*)

lemma (**in** *partial-pointer-lense*) *read-commute-of-pointer-writer-disjnt*:
assumes w': $\bigwedge f.$ *pointer-writer-disjnt* $(\lambda p.\ w\ p\ f)\ w'$
assumes *p-q*: *disjnt* (*ptr-span p*) (*ptr-span q*)
shows $r\ (w'\ q\ h)\ p\ y = r\ h\ p\ y$
by (*metis m-r p-q pointer-writer-disjntD r-w w' w-r*)

lemma *pointer-lense-upd-fun-of-lense*:
fixes *get upd* **assumes** *lense get upd*
shows *pointer-lense get* $(\lambda p\ f.$ *upd* (*upd-fun p f*))
proof −
interpret *lense get upd* **by** *fact*
show *?thesis*
proof qed (*simp-all add*: *upd-same upd-fun.upd-same disj-ptr-span-ptr-neq*
upd-fun-commute Int-commute disjnt-def)
qed

locale *open-types-heap-typing-state* = *open-types* $\mathcal{T}$ +
heap-typing-state heap-typing heap-typing-upd
for
$\mathcal{T}$ **and**
heap-typing :: $'t \Rightarrow$ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'t \Rightarrow$ $'t$

locale *typ-heap-simulation-open-types* =
typ-heap-simulation st r w v t-hrs t-hrs-update +
open-types-heap-typing-state $\mathcal{T}$ *heap-typing heap-typing-upd*
for
$\mathcal{T}$ **and**
st:: $'s \Rightarrow 't$ **and**
r:: $'t \Rightarrow$ ($'a$::{*xmem-type*, *stack-type*}) *ptr* $\Rightarrow$ $'a$ **and**
w:: $'a$ *ptr* $\Rightarrow$ ($'a \Rightarrow 'a$) $\Rightarrow$ $'t \Rightarrow$ $'t$ **and**
v:: $'t \Rightarrow$ $'a$ *ptr* $\Rightarrow$ *bool* **and**
t-hrs :: $'s \Rightarrow$ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ $'s \Rightarrow$ $'s$ **and**
heap-typing :: $'t \Rightarrow$ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'t \Rightarrow$ $'t$ +
assumes *heap-typing-commutes*[*simp*]: *heap-typing* (*st s*) = *hrs-htd* (*t-hrs s*)
assumes *heap-typing-upd-write-commute*:
$\bigwedge p\ f\ h\ d.$ *heap-typing-upd d* $(w\ p\ f\ h) = w\ p\ f$ (*heap-typing-upd d h*)

assumes *ptr-valid-imp-v*: $\bigwedge p\ h.\ ptr\text{-}valid\ (heap\text{-}typing\ h)\ p \Longrightarrow v\ h\ p$
assumes *sim-stack-stack-byte-zero'*:
$\bigwedge p\ s.\ \forall a{\in}ptr\text{-}span\ p.\ root\text{-}ptr\text{-}valid\ (hrs\text{-}htd\ (t\text{-}hrs\ s))\ (PTR(stack\text{-}byte)\ a) \Longrightarrow$
$r\ (st\ s)\ p = ZERO('a)$
begin

lemma *heap-typing-write*[*simp*]: $heap\text{-}typing\ (w\ p\ f\ h) = heap\text{-}typing\ h$
apply (*subst upd-same*[*symmetric*, *of* λ-. *heap-typing h*, *OF refl*])
apply (*subst heap-typing-upd-write-commute*[*symmetric*])
apply (*rule get-upd*)
done

lemma *heap-typing-upd*[*simp*]: $r\ (heap\text{-}typing\text{-}upd\ d\ h) = r\ h$
apply (*rule ext*)
subgoal for *p*
apply (*subst write-same*[*symmetric*, *of* λ-. *r h p*, *OF refl*])
apply (*subst heap-typing-upd-write-commute*)
apply (*rule read-write-same*)
done
done

lemma *unchanged-typing-root-ptr-valid-preserved*:
unchanged-typing-on A t1 t2 $\Longrightarrow$ *ptr-span p* $\subseteq$ *A*
$\Longrightarrow$ *root-ptr-valid* (*heap-typing t1*) *p* = *root-ptr-valid* (*heap-typing t2*) *p*
by (*simp add*: *unchanged-typing-on-def equal-on-def intvlI*
root-ptr-valid-def size-of-tag subsetD valid-root-footprint-def)

lemma *unchanged-typing-ptr-valid-preserved*:
assumes *unch*: *unchanged-typing-on UNIV t1 t2*
shows *ptr-valid* (*heap-typing t1*) *p* = *ptr-valid* (*heap-typing t2*) *p*
proof −
from *unch* **have** *heap-typing t1* = *heap-typing t2*
by (*simp add*: *fun-eq-iff unchanged-typing-on-def equal-on-def*)
thus *?thesis*
by *simp*
qed

lemma *unchanged-typing-on-write* [*unchanged-typing-on-simps*]:
unchanged-typing-on A t (*w p f t*)
using *heap-typing-upd-write-commute*
by (*simp add*: *unchanged-typing-on-def*)

lemma *heap-typing-fold-upd-write*: $heap\text{-}typing\ (fold\ (\lambda i.\ w\ (x\ i)\ (g\ i))\ [0..{<}n]\ t)$
$= heap\text{-}typing\ t$
proof (*induct n*)
case *0*
then show *?case* **by** (*simp add*: *heap-typing-upd-write-commute*)
next
case (*Suc n*)

then show *?case* **by** (*simp add*: *heap-typing-upd-write-commute*)
qed

end

lemma *distinct-prop-conj*:
distinct-prop (λ*a b*. *R a b* ∧ *P a b*) *xs* ⟷ *distinct-prop R xs* ∧ *distinct-prop P xs*
by (*auto simp*: *distinct-prop-iff-nth*)

lemma *distinct-prop-zip-cons*:
list-all (λ(*c*, *d*). *P a b c d*) (*zip as bs*) ⟹
distinct-prop (λ(*a*, *b*) (*c*, *d*). *P a b c d*) (*zip as bs*) ⟹
distinct-prop (λ(*a*, *b*) (*c*, *d*). *P a b c d*) (*zip* (*a*#*as*) (*b*#*bs*))
by (*simp add*: *list-all-iff*)

lemma *distinct-prop-zip-nil*:
distinct-prop (λ(*a*, *b*) (*c*, *d*). *P a b c d*) (*zip* [] [])
by (*simp add*: *list-all-iff*)

lemma *list-all-zip-cons*:
P a b ⟹ *list-all* (λ(*a*, *b*). *P a b*) (*zip as bs*) ⟹
list-all (λ(*a*, *b*). *P a b*) (*zip* (*a*#*as*) (*b*#*bs*))
by *simp*

lemma *list-all-zip-nil*:
list-all (λ(*a*, *b*). *P a b*) (*zip* [] [])
by *simp*

named-theorems *disjnt-heap-fields-comp*

context *open-types*
begin

lemma *typ-heap-simulationI-part-addressable*:
fixes *R* :: *'t* ⇒ *'a*::{*xmem-type*, *stack-type*} *ptr* ⇒ *'a*
assumes *hrs*: *heap-typing-simulation* 𝒯 *hrs hrs-upd heap-typing heap-typing-upd l*
assumes *fs*: *map-of* 𝒯 (*typ-uinfo-t TYPE*(*'a*)) = *Some fs*
assumes *lense g u*
assumes *len*[*simp*]: *length rs* = *length fs length ws* = *length fs*
assumes ∗:
list-all (λ(*f*, *r*, *w*). *typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f r w*) (*zip fs* (*zip rs ws*))
and ∗∗: *distinct-prop* (λ(*f1*, *w1*) (*f2*, *w2*).
disj-fn f1 f2 ⟶ *pointer-writer-disjnt-eq w1 w2*)
(*zip fs ws*)
and *ws-u*: *list-all* (λ*w*. ∀ *p a f*. *w p a* ∘ *u f* = *u f* ∘ *w p a*) *ws*

and *l-u*: $\bigwedge(p::'a\ ptr)\ (x::'a)\ (s::'b)$.
PTR-VALID('a) (*hrs-htd* (*hrs s*)) *p* $\Longrightarrow$
l (*hrs-upd* (*write-dedicated-heap p x*) *s*) = *u* (*upd-fun p* (λ*old*. *merge-addressable-fields old x*)) (*l s*)
and *r-stack*:
$\bigwedge p\ s.\ \forall\, a{\in}$*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*hrs s*)) (*PTR*(*stack-byte*) *a*) $\Longrightarrow$
g (*l s*) *p* = *ZERO*(-)
and *heap-typing-u*: $\bigwedge x\ d\ h$. *heap-typing-upd d* (*u x h*) = *u x* (*heap-typing-upd d h*)
assumes *V*:
$\bigwedge h\ p$. *V h p* $\longleftrightarrow$ *PTR-VALID*('a) (*heap-typing h*) *p*
assumes *R*:
$\bigwedge h\ p$. *R h p* = *fold* (λr. *r h p*) *rs* (*g h p*)
assumes *W*:
$\bigwedge p\ f\ h$. *W p f h* =
fold (λw. *w p* (*f* (*R h p*))) *ws* (*u* (*upd-fun p* (λ*old*. *merge-addressable-fields old* (*f* (*R h p*)))) *h*)
shows *typ-heap-simulation-open-types* $\mathcal{T}$ *l R W V hrs hrs-upd heap-typing heap-typing-upd* $\wedge$
(*pointer-lense g* ($\lambda p\ f$. *u* (*upd-fun p f*))) $\wedge$
($\forall w$. ($\forall x$. *list-all* ($\lambda w'$. *pointer-writer-disjnt* (λp. *w' p x*) *w*) *ws*) $\longrightarrow$
($\forall x$. *pointer-writer-disjnt* (λp. *u* (*upd-fun p* (λ-. *x*))) *w*) $\longrightarrow$
($\forall f$. *pointer-writer-disjnt* (λp. *W p f*) *w*)) $\wedge$
($\forall w\ p$. ($\forall x$. *list-all* ($\lambda w'$. *w' p x o w* = *w o w' p x*) *ws*) $\longrightarrow$
($\forall x$. *u* (*upd-fun p* (λ-. *x*)) *o w* = *w o u* (*upd-fun p* (λ-. *x*))) $\longrightarrow$
($\forall f$. *W p f o w* = *w o W p f*))
(**is** *?t1* $\wedge$ *?t3* $\wedge$
($\forall w$. *?ws w* $\longrightarrow$ *?u w* $\longrightarrow$ ($\forall f$. *?t2 f w*)) $\wedge$
($\forall w\ p$. *?ws2 w p* $\longrightarrow$ *?u2 w p* $\longrightarrow$ ($\forall f$. *?t4 w p f*)))
proof (*intro conjI allI impI*)
interpret *hrs*: *heap-typing-simulation* $\mathcal{T}$ *hrs hrs-upd heap-typing heap-typing-upd l* **by** *fact*
interpret *lense g u* **by** *fact*

have [*simp*]: *length* (*addressable-fields TYPE*('a)) = *length fs*
by (*simp add*: *addressable-fields-def fs*)

have *list-all* (λf. $\exists\, u\ n$. *field-lookup* (*typ-uinfo-t TYPE*('a)) *f 0* = *Some* (*u*, *n*)) (*map fst* (*zip fs* (*zip rs ws*)))
using *wf-*$\mathcal{T}$[*OF fs*] **by** *auto*
then have *fs-exists*:
list-all (λx. $\exists\, u\ n$. *field-lookup* (*typ-uinfo-t TYPE*('a)) (*fst x*) *0* = *Some* (*u*, *n*)) (*zip fs* (*zip rs ws*))
by (*simp add*: *list.pred-map comp-def del*: *len*)

have *rs-ws*:
list-all (λ(*m*, *r*, *w*). *partial-pointer-lense m r w*) (*zip* (*map merge-ti* (*map snd* (*addressable-fields TYPE*('a)))) (*zip rs ws*))
apply (*simp add*: *zip-map1 list.pred-map addressable-fields-def fs split-beta'*

```
comp-def)
    using list-all-conj[OF * fs-exists]
    apply (rule list.pred-mono-strong)
   apply (auto simp: field-ti-def field-lookup-typ-uinfo-t-Some typ-heap-simulation-of-field-def
                dest!: field-lookup-uinfo-Some-rev)
    done

  have dist: distinct-prop disj-fn fs
    using 𝒯-distinct[OF fs] .

  interpret pointer-lense R W
    apply (rule pointer-lense-of-partials-cover[
     of g u map merge-ti (map snd (addressable-fields TYPE('a))) rs ws, OF ‹lense
g u› - - - - - - R])
    subgoal by simp
    subgoal by simp
    subgoal by fact
    subgoal for a b c
      apply (simp add: distinct-prop-map addressable-fields-def fs)
      apply (rule distinct-prop-mono[OF - dist])
      using wf-𝒯[OF fs]
      apply (intro disjnt-scene-merge-ti[THEN disjnt-sceneD,
        OF option.collapse[symmetric] option.collapse[symmetric]])
      apply (auto simp: list.pred-set field-ti-def disj-fn-def
                  dest!: field-lookup-uinfo-Some-rev)
      done
    subgoal
      using ws-u ** dist
      apply (clarsimp simp add: list-all-iff pointer-writer-disjnt-eq-def[abs-def])
      apply (clarsimp simp add: distinct-prop-iff-nth fun-eq-iff distinct-conv-nth
          disj-fn-def)
      apply metis
      done
    subgoal by (simp add: merge-ti-list-def fold-map comp-def)
    subgoal by (rule W)
    done

  have list-all (λ(f, w). ∀ t u n h p x.
        field-ti TYPE('a) f = Some t ⟶
        field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n) ⟶
        ptr-valid-u u (hrs-htd (hrs h)) &(p→f) ⟶
         l (hrs-upd (hrs-mem-update (heap-upd-list (size-td u) &(p→f) (access-ti t
x))) h) =
          w p x (l h))
     (map (λfrw. (fst frw, snd (snd frw))) (zip fs (zip rs ws)))
    unfolding list.pred-map
   using * by (rule list.pred-mono-strong) (auto simp: typ-heap-simulation-of-field-def)
  also have map (λfrw. (fst frw, snd (snd frw))) (zip fs (zip rs ws)) = zip fs ws
    by (simp add: list-eq-iff-nth-eq)
```

```
  finally have fs-ws: list-all2 (λf w. ∀ t u n h p x.
        field-ti TYPE('a) f = Some t ⟶
        field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n) ⟶
        ptr-valid-u u (hrs-htd (hrs h)) &(p→f) ⟶
         l (hrs-upd (hrs-mem-update (heap-upd-list (size-td u) &(p→f) (access-ti t
x))) h) =
          w p x (l h))
     fs ws
    by (subst (asm) list-all-zip-iff-list-all2; simp)

  have V-l: ⋀h p. V (l h) p ⟷ PTR-VALID('a) (hrs-htd (hrs h)) p
    by (simp add: V)

  interpret write-simulation hrs hrs-upd l V W
    apply (rule write-simulationI[OF fs hrs, of ws u V - R])
    apply (rule fs-ws)
    apply (rule l-u)
    apply assumption
    apply (rule V-l)
    apply (rule W)
    done

  show ?t3
    by (rule pointer-lense-upd-fun-of-lense) fact
  then interpret u: pointer-lense g λp f. u (upd-fun p f) .

  show ?t2 f w if ∗: ?ws w ?u w for w f
    by (rule pointer-writer-disjnt-replace-by-const)
      (use ∗ in ‹auto simp add: W[abs-def]
       intro!: pointer-writer-disjnt-comp-left[OF pointer-writer-disjnt-fold-left
            u.pointer-writer-disjnt-replace-by-const]›)

  show disj: ?t4 w p f if ∗: ?u2 w p ?ws2 w p for p f w
  proof (standard, unfold comp-def)
    fix h
    have w-ws: w (fold (λw. w p c) ws h) = fold (λw. w p c) ws (w h) for c h
      using ∗[rule-format, of c]
      by (induction ws arbitrary: h) (auto simp: fun-eq-iff)

    have w-u[simp]: w (u (upd-fun p (λ-. c)) h) = u (upd-fun p (λ-. c)) (w h) for
c h
      using ∗(1) by (auto simp: fun-eq-iff)

    have g-w[simp]: g (w h) p = g h p for h
      by (subst u.write-same[of λ-. g h p h p, OF refl, symmetric])
        (simp add: u.read-write-same)

    have w-u'[simp]: w (u (upd-fun p f) h) = u (upd-fun p f) (w h) for f h
      by (subst (1 2) u.write-cong[where f'=λ-. f (g h p)]) simp-all
```

```
  have w-W-const: w (W p (λx. c) h) = W p (λx. c) (w h) for c h
    by (simp add: W w-ws)

  have R-eq: R (w h) p = R h p
    apply (subst write-same[of λ-. R h p h p, OF refl, symmetric])
    apply (subst w-W-const)
    apply (subst read-write-same)
    apply (rule refl)
    done

  show W p f (w h) = w (W p f h)
    by (simp add: R-eq write-cong[of f - - λ-. f (R h p)] w-W-const)
qed

show ?t1
proof
  fix h p assume V-p: V (l h) p
  then show c-guard p by (simp add: ptr-valid.ptr-valid-c-guard V)

  have l-h: l h = W p (λx. h-val (hrs-mem (hrs h)) p) (l h)
    apply (subst write-commutes[OF V-p, symmetric])
    apply (subst hrs.t-hrs.heap.upd-same)
    apply (cases hrs h)
   apply (simp-all add: hrs-mem-update-def hrs-mem-def xmem-type-class.heap-update-id)
    done

  show R (l h) p = h-val (hrs-mem (hrs h)) p
    apply (subst l-h)
    apply (subst read-write-same)
    apply simp
    done
next
  fix p :: 'a ptr and s
  assume p: ∀ a∈ptr-span p. root-ptr-valid (hrs-htd (hrs s)) (PTR(stack-byte) a)
  moreover
  { fix f w and r :: 't ⇒ 'a ptr ⇒ 'a ⇒ 'a and u n
    assume typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f r w
      field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n)
    with p have r (l s) p ZERO('a) = ZERO('a)
      unfolding typ-heap-simulation-of-field-def
    by (auto simp: field-ti-def field-lookup-typ-uinfo-t-Some typ-heap-simulation-of-field-def
              dest!: field-lookup-uinfo-Some-rev) }
  note this[simp]
  from * fs-exists len have fold (λr. r (l s) p) rs ZERO('a) = ZERO('a)
    by (induction fs arbitrary: rs ws)
       (auto simp: length-Suc-conv simp del: len)
  ultimately show R (l s) p = ZERO('a)
    unfolding R by (simp add: r-stack)
```

```
  next
    { fix d p f
      have W p f o heap-typing-upd d = heap-typing-upd d o W p f
        apply (rule disj)
        subgoal using heap-typing-u by (simp add: fun-eq-iff)
        using ∗ len
        apply (induction fs arbitrary: ws rs)
       apply (auto simp add: length-Suc-conv typ-heap-simulation-of-field-def simp
del: len)
        done
      then show heap-typing-upd d (W p f h) = W p f (heap-typing-upd d h) for h
        by (simp add: fun-eq-iff) }
    note heap-typing-W = this

    show V (W p' f h) p = V h p for p' f h p
      unfolding V
      apply (subst hrs.upd-same[symmetric, of λ-. heap-typing h, OF refl])
      apply (subst heap-typing-W[symmetric])
      apply (subst hrs.get-upd)
      ..
  qed (simp-all add: V)
qed

lemma typ-heap-simulationI-no-addressable:
  fixes R :: 't ⇒ 'a::{xmem-type, stack-type} ptr ⇒ 'a
  assumes heap-typing-simulation T hrs hrs-upd heap-typing heap-typing-upd l
  assumes R-u: lense R u
  assumes fs: map-of T (typ-uinfo-t TYPE('a)) = None
    and l-u: ⋀(p::'a ptr) (x::'a) (s::'b).
      PTR-VALID('a) (hrs-htd (hrs s)) p ⟹
    l (hrs-upd (write-dedicated-heap p x) s) = u (upd-fun p (λold. merge-addressable-fields
old x)) (l s)
    and heap-typing-u: ⋀x d h. heap-typing-upd d (u x h) = u x (heap-typing-upd
d h)
    and r-stack:
      ⋀p s. ∀ a∈ptr-span p. root-ptr-valid (hrs-htd (hrs s)) (PTR(stack-byte) a) ⟹
        R (l s) p = ZERO(-)
  assumes V:
    ⋀h p. V h p ⟷ PTR-VALID('a) (heap-typing h) p
  assumes W:
    ⋀p f h. W p f h = u (λh'. h'(p := f (h' p))) h
 shows typ-heap-simulation-open-types T l R W V hrs hrs-upd heap-typing heap-typing-upd
proof −
  interpret hrs: heap-typing-simulation T hrs hrs-upd heap-typing heap-typing-upd
l by fact
  interpret lense R u by fact

  have [simp]: merge-addressable-fields = (λa b::'a. b)
    by (simp add: fun-eq-iff addressable-fields-def fs)
```

```
  interpret pointer-lense R W
    using pointer-lense-of-lense-fld[where 'd='a, OF R-u, of []]
    by (simp add: W[abs-def] upd-fun-def[abs-def])

  interpret write-simulation hrs hrs-upd l V W
    apply (rule hrs.write-simulation-alt)
    subgoal by (simp add: V)
    subgoal for h p x
      unfolding W V
      using l-u[of h p x]
      by (simp add: write-dedicated-heap-def heap-upd-const upd-fun-def[abs-def])
    done

  show ?thesis
  proof
    fix h p assume V-p: V (l h) p
    then show c-guard p by (simp add: ptr-valid.ptr-valid-c-guard V)

    have l-h: l h = W p (λx. h-val (hrs-mem (hrs h)) p) (l h)
      apply (subst write-commutes[OF V-p, symmetric])
      apply (subst hrs.t-hrs.heap.upd-same)
      apply (cases hrs h)
     apply (simp-all add: hrs-mem-update-def hrs-mem-def xmem-type-class.heap-update-id)
      done

    show R (l h) p = h-val (hrs-mem (hrs h)) p
      apply (subst l-h)
      apply (subst read-write-same)
      apply simp
      done
  next
    show V (W p' f h) p = V h p for p' f h p
      unfolding W V
      apply (subst hrs.upd-same[symmetric, of λ-. heap-typing h, OF refl])
      apply (subst heap-typing-u[symmetric])
      apply (subst hrs.get-upd)
      ..
  qed (simp-all add: V W heap-typing-u r-stack)
qed

lemma typ-heap-simulationI-all-addressable:
  fixes R :: 't ⇒ 'a::{xmem-type, stack-type} ptr ⇒ 'a
  assumes hrs: heap-typing-simulation 𝒯 hrs hrs-upd heap-typing heap-typing-upd
l
  assumes fs: map-of 𝒯 (typ-uinfo-t TYPE('a)) = Some fs
  assumes len[simp]: length rs = length fs length ws = length fs
  assumes ∗:
    list-all (λ(f, r, w). typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f
```

```
r w) (zip fs (zip rs ws))
    and **: distinct-prop (λ(f1, w1) (f2, w2).
        disj-fn f1 f2 ⟶ pointer-writer-disjnt-eq w1 w2)
      (zip fs ws)
  assumes all: ⋀a b. fold (λx. merge-ti (the (field-ti TYPE('a) x)) a) fs b = a
  assumes V:
    ⋀h p. V h p ⟷ PTR-VALID('a) (heap-typing h) p
  assumes R:
    ⋀h p x. R h p = fold (λr. r h p) rs x
  assumes W:
    ⋀p f h. W p f h = fold (λw. w p (f (R h p))) ws h
 shows typ-heap-simulation-open-types T l R W V hrs hrs-upd heap-typing heap-typing-upd
∧
    (∀w. (∀x. list-all (λw'. pointer-writer-disjnt (λp. w' p x) w) ws) ⟶
      (∀f. pointer-writer-disjnt (λp. W p f) w)) ∧
    (∀(w::'t ⇒ 't) p.
      (∀x. list-all (λw'. w' p x ∘ w = w ∘ w' p x) ws) ⟶
      (∀f. W p f ∘ w = w ∘ W p f))
    (is ?t1 ∧ (∀w. ?ws w ⟶ (∀f. ?t2 f w)) ∧ (∀w p. ?ws2 p w ⟶ (∀f. ?t3 p f
w)))
proof (intro conjI allI impI)
  interpret hrs: heap-typing-simulation T hrs hrs-upd heap-typing heap-typing-upd
l by fact

  have [simp]: length (addressable-fields TYPE('a)) = length fs
    by (simp add: addressable-fields-def fs)

  have list-all (λf. ∃u n. field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n))
    (map fst (zip fs (zip rs ws)))
    using wf-T[OF fs] by auto
  then have fs-exists:
    list-all (λx. ∃u n. field-lookup (typ-uinfo-t TYPE('a)) (fst x) 0 = Some (u, n))
      (zip fs (zip rs ws))
    by (simp add: list.pred-map comp-def del: len)

  have coer-eq[simp]: merge-addressable-fields = (λa b::'a. a)
   by (simp add: merge-ti-list-def addressable-fields-def fold-map fs fun-eq-iff comp-def
all)

  have dist: distinct-prop disj-fn fs
    using T-distinct[OF fs] .

  interpret pointer-lense R W
    apply (rule pointer-lense-of-partials[
      of map merge-ti (map snd (addressable-fields TYPE('a))) rs ws, OF - - - - -
- R])
    subgoal by simp
    subgoal by simp
    subgoal
```

apply (*simp add*: *zip-map1 list.pred-map addressable-fields-def fs split-beta′ comp-def*)
using *list-all-conj*[*OF* ∗ *fs-exists*]
apply (*rule list.pred-mono-strong*)
apply (*auto simp*: *field-ti-def field-lookup-typ-uinfo-t-Some typ-heap-simulation-of-field-def dest*!: *field-lookup-uinfo-Some-rev*)
done
subgoal for *a b c*
apply (*simp add*: *distinct-prop-map addressable-fields-def fs*)
apply (*rule distinct-prop-mono*[*OF - dist*])
using *wf-*$\mathcal{T}$[*OF fs*]
apply (*intro disjnt-scene-merge-ti*[*THEN disjnt-sceneD*, *OF option.collapse*[*symmetric*] *option.collapse*[*symmetric*]])
apply (*auto simp*: *list.pred-set field-ti-def disj-fn-def dest*!: *field-lookup-uinfo-Some-rev*)
done
subgoal
using ∗∗ *dist*
apply (*clarsimp simp add*: *distinct-prop-iff-nth fun-eq-iff pointer-writer-disjnt-eq-def*[*abs-def*] *distinct-conv-nth disj-fn-def*)
apply *metis*
done
subgoal by (*simp add*: *addressable-fields-def fs fold-map comp-def all*)
subgoal by (*rule W*)
done

have *list-all* (λ(*f*, *w*). ∀ *t u n h p x*.
field-ti TYPE('*a*) *f* = *Some t* ⟶
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*u*, *n*) ⟶
ptr-valid-u u (*hrs-htd* (*hrs h*)) &(*p*→*f*) ⟶
l (*hrs-upd* (*hrs-mem-update* (*heap-upd-list* (*size-td u*) &(*p*→*f*) (*access-ti t x*))) *h*) =
w p x (*l h*))
(*map* (λ*frw*. (*fst frw*, *snd* (*snd frw*))) (*zip fs* (*zip rs ws*)))
unfolding *list.pred-map*
using ∗ **by** (*rule list.pred-mono-strong*) (*auto simp*: *typ-heap-simulation-of-field-def*)
also have *map* (λ*frw*. (*fst frw*, *snd* (*snd frw*))) (*zip fs* (*zip rs ws*)) = *zip fs ws*
by (*simp add*: *list-eq-iff-nth-eq*)
finally have *fs-ws*: *list-all2* (λ*f w*. ∀ *t u n h p x*.
field-ti TYPE('*a*) *f* = *Some t* ⟶
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*u*, *n*) ⟶
ptr-valid-u u (*hrs-htd* (*hrs h*)) &(*p*→*f*) ⟶
l (*hrs-upd* (*hrs-mem-update* (*heap-upd-list* (*size-td u*) &(*p*→*f*) (*access-ti t x*))) *h*) =
w p x (*l h*))
fs ws
by (*subst* (*asm*) *list-all-zip-iff-list-all2*; *simp*)

have *V-l*: ⋀*h p*. *V* (*l h*) *p* ⟷ *PTR-VALID*('*a*) (*hrs-htd* (*hrs h*)) *p*

```
    by (simp add: V)

  interpret write-simulation hrs hrs-upd l V W
    apply (rule write-simulationI[OF fs hrs, of ws λx y. y V - R])
    apply (rule fs-ws)
    apply (simp add: write-dedicated-heap-def heap-upd-id hrs-mem-update-id3 flip:
id-def)
    apply (simp add: id-def)
    apply (rule V-l)
    apply (rule W)
    done

  show ?t2 f w if ∗: ?ws w for w f
    apply (rule pointer-writer-disjnt-replace-by-const)
    apply (simp add: W[abs-def])
    apply (intro pointer-writer-disjnt-fold-left ∗[rule-format])
    done

  show disj: ?t3 p f w if ∗: ?ws2 p w for p f w
  proof (standard, unfold comp-def)
    fix h
    have w-W-const: w (W p (λx. c) h) = W p (λx. c) (w h) for c h
      using ∗[rule-format, of c]
      apply (simp add: W)
      apply (induction ws arbitrary: h)
      apply (auto simp: fun-eq-iff)
      done
    have R-eq: R (w h) p = R h p
      apply (subst write-same[of λ-. R h p h p, OF refl, symmetric])
      apply (subst w-W-const)
      apply (subst read-write-same)
      apply (rule refl)
      done

    show W p f (w h) = w (W p f h)
      by (simp add: R-eq write-cong[of f - - λ-. f (R h p)] w-W-const)
  qed

  show ?t1
  proof
    fix h p assume V-p: V (l h) p
    then show c-guard p by (simp add: ptr-valid.ptr-valid-c-guard V)

    have l-h: l h = W p (λx. h-val (hrs-mem (hrs h)) p) (l h)
      apply (subst write-commutes[OF V-p, symmetric])
      apply (subst hrs.t-hrs.heap.upd-same)
      apply (cases hrs h)
      apply (simp-all add: hrs-mem-update-def hrs-mem-def xmem-type-class.heap-update-id)
      done
```

```
    show R (l h) p = h-val (hrs-mem (hrs h)) p
      apply (subst l-h)
      apply (subst read-write-same)
      apply simp
      done
  next
    fix p :: 'a ptr and s
    assume p: ∀ a∈ptr-span p. root-ptr-valid (hrs-htd (hrs s)) (PTR(stack-byte) a)
    moreover
    { fix f w and r :: 't ⇒ 'a ptr ⇒ 'a ⇒ 'a and u n
      assume typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f r w
        field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (u, n)
      with p have r (l s) p ZERO('a) = ZERO('a)
        unfolding typ-heap-simulation-of-field-def
      by (auto simp: field-ti-def field-lookup-typ-uinfo-t-Some typ-heap-simulation-of-field-def
              dest!: field-lookup-uinfo-Some-rev) }
    note this[simp]
    from ∗ fs-exists len have fold (λr. r (l s) p) rs ZERO('a) = ZERO('a)
      by (induction fs arbitrary: rs ws)
         (auto simp: length-Suc-conv simp del: len)
    ultimately show R (l s) p = ZERO('a)
      unfolding R[of - - ZERO('a)] by simp
  next
    { fix p f d
      have W p f o heap-typing-upd d = heap-typing-upd d o W p f
        apply (rule disj)
        using ∗ len
        apply (induction fs arbitrary: ws rs)
       apply (auto simp add: length-Suc-conv typ-heap-simulation-of-field-def simp
del: len)
        done
     then show heap-typing-upd d (W p f h) = W p f (heap-typing-upd d h) for h
        by (simp add: fun-eq-iff) }
    note heap-typing-W = this

    show V (W p' f h) p = V h p for p' f h p
      unfolding V
      apply (subst hrs.upd-same[symmetric, of λ-. heap-typing h, OF refl])
      apply (subst heap-typing-W[symmetric])
      apply (subst hrs.get-upd)
      ..
  qed (simp-all add: V)
qed

end

definition
  field-with-lense ::
```

qualified-field-name ⇒ (′*a*::*c-type* ⇒ ′*b*::*c-type*) ⇒ ((′*b* ⇒ ′*b*) ⇒ ′*a* ⇒ ′*a*) ⇒ *bool*
where
field-with-lense f g u ⟷
field-ti TYPE(′*a*) *f* = *Some* (*adjust-ti* (*typ-info-t TYPE*(′*b*)) *g* (*u o* (λ*x* -. *x*)))
∧
lense g u

lemma *update-desc-id*: *update-desc* (λ*x*. *x*) (λ*x* -. *x*) = *id*
by (*simp add*: *fun-eq-iff update-desc-def*)

lemma *field-with-lense-Nil*: *field-with-lense* [] (λ*x*. *x*) (λ*f x*. *f x*)
by (*simp add*: *field-with-lense-def lense-def comp-def adjust-ti-def update-desc-id map-td-id*(*1*))

lemma *field-with-lense-Cons*:
fixes *g* :: ′*a*::*mem-type* ⇒ ′*b*::*mem-type* **and** *gs* :: ′*b* ⇒ ′*c*::*mem-type*
assumes *fs*: *field-with-lense fs gs us*
assumes *f*: *field-ti TYPE*(′*a*) [*f*] = *Some* (*adjust-ti* (*typ-info-t TYPE*(′*b*)) *g* (*u o* (λ*x* -. *x*)))
assumes *g-u*: *fg-cons g* (*u o* (λ*x* -. *x*))
assumes *u*: ⋀*f x*. *u f x* = *u* (λ-. *f* (*g x*)) *x*
shows *field-with-lense* (*f* # *fs*) (λ*x*. *gs* (*g x*)) (λ*f*. *u* (*us f*))
unfolding *field-with-lense-def*
proof
from *fs* **obtain** *n* **where** *fs*:
field-lookup (*typ-info-t TYPE*(′*b*)) *fs 0* =
Some (*adjust-ti* (*typ-info-t TYPE*(′*c*)) *gs* (*us o* (λ*x* -. *x*)), *n*)
and *gs-us*: *lense gs us*
by (*auto simp add*: *field-with-lense-def field-ti-def split*: *option.splits*)

have *g-u*: *lense g u*
using *g-u*
apply (*rule lense-of-fg-cons*′)
apply (*rule u*)
done
then interpret *lense g u* **.**
show *lense* (λ*x*. *gs* (*g x*)) (λ*f*. *u* (*us f*))
using *g-u gs-us* **by** (*rule lense-compose*)

have (λ*x*. *gs* (*g x*)) = *gs* ∘ *g*
(λ*v x*. *u* (λ-. *us* (λ-. *v*) (*g x*)) *x*) = (λ*f*. *u* (*us f*)) ∘ (λ*x* -. *x*)
by (*simp-all add*: *fun-eq-iff cong*: *upd-cong*)
with *field-ti-append-field-lookup*[*OF f*, *of fs 0*]
show *field-ti TYPE*(′*a*) (*f* # *fs*) =
Some (*adjust-ti* (*typ-info-t TYPE*(′*c*)) (λ*x*. *gs* (*g x*)) ((λ*f*. *u* (*us f*)) ∘ (λ*x* -. *x*)))
unfolding *field-lookup-adjust-ti2*[*OF fs*]
by (*simp add*: *adjust-ti-adjust-ti*)

qed

lemma (**in** *typ-heap-simulation-open-types*) *typ-heap-simulation-of-field*:
assumes *f*: *field-with-lense f g u*
assumes *v*: $\bigwedge h\ p.$ *PTR-VALID*('*a*) (*hrs-htd* (*t-hrs h*)) *p* $\Longrightarrow$ *v* (*st h*) *p*
assumes *g-zero*: *g ZERO*('*x*::*mem-type*) = *ZERO*('*a*)
shows *typ-heap-simulation-of-field st t-hrs t-hrs-update heap-typing-upd f*
(λ*h p*. *u* (λ-. *r h* (*PTR*('*a*) &(*p*→*f*))))
(λ*p x*. *w* (*PTR*('*a*) &(*p*→*f*)) (λ-. *g x*))
proof −
have *g-u*: *lense g u* **using** *f* **by** (*simp add*: *field-with-lense-def*)
then interpret *lense g u* **.**

have [*simp*]: *fg-cons g* (*u* ∘ (λ*x* -. *x*))
by (*rule fg-cons*)
obtain *n* **where** [*simp*]:
field-lookup (*typ-uinfo-t TYPE*('*x*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*a*), *n*)
using *f*
by (*auto simp*: *field-ti-def field-lookup-typ-uinfo-t-Some field-with-lense-def*
split: *option.splits*)
have *sz*: *size-td* (*typ-info-t TYPE*('*a*)) = *size-of TYPE*('*a*)
by (*simp add*: *size-of-def*)

have *p-f-eq*: &(*p*→*f*) = *ptr-val* (*PTR*('*c*) &(*p*→*f*)) **for** *p* :: '*x ptr*
by *simp*

have *ptr-span-subset*: *ptr-span* (*PTR*('*a*) &(*p*→*f*)) ⊆ *ptr-span p* **for** *p* :: '*x ptr*
using *field-tag-sub*'[**where** '*a*='*x*, *of f - - p*] **by** (*simp add*: *size-of-def*)

have *u-zero*: *u* (λ-. *ZERO*('*a*)) *ZERO*('*x*) = *ZERO*('*x*)
by (*simp add*: *upd-same g-zero*)

show *?thesis* **using** *f*
apply (*clarsimp simp*: *typ-heap-simulation-of-field-def field-with-lense-def*, *intro conjI allI impI*)
subgoal by (*simp add*: *heap-typing-upd-write-commute fun-eq-iff*)
subgoal
apply (*rule partial-pointer-lenseI*[*OF g-u*])
apply (*rule pointer-lense-field-lvalue*)
apply *assumption*
apply (*simp add*: *size-of-def*)
done
subgoal for *p*
using *ptr-span-subset*[*of p*]
by (*subst sim-stack-stack-byte-zero*') (*auto simp*: *u-zero*)
subgoal
apply (*subst sz*)
apply (*subst p-f-eq*)
apply (*subst heap-update-eq-heap-upd-list*[*symmetric*])

```
    apply (rule write-commutes)
    apply (simp add: v ptr-valid-def)
    done
  done
qed

context pointer-array-lense
begin

lemma pointer-writer-disjnt-heap-array-map[pointer-writer-disjnt-intros]:
  assumes w': ⋀x. pointer-writer-disjnt (λp. w p x) w'
  shows pointer-writer-disjnt (λp. heap-array-map p x) w'
  apply (rule array.pointer-writer-disjnt-replace-by-const)
  apply (simp only: heap-array-map-def[abs-def])
  apply (rule pointer-writer-disjnt-fold-left)
  apply (clarsimp simp add: list-all-iff)
 apply (rule pointer-writer-disjnt-ptr-map-left[OF w' ptr-span-array-ptr-index-subset-ptr-span])
  apply simp
  done

lemma pointer-writer-disjnt-heap-array-map-right[pointer-writer-disjnt-intros]:
  (⋀x. pointer-writer-disjnt w' (λp. w p x)) ⟹ pointer-writer-disjnt w' (λp. heap-array-map
p x)
  by (metis pointer-writer-disjnt-heap-array-map pointer-writer-disjnt-sym)

lemma disjnt-comp-heap-array-map[disjnt-heap-fields-comp]:
  assumes w': ⋀q x. w q x ∘ w' = w' ∘ w q x
  shows heap-array-map p x ∘ w' = w' ∘ heap-array-map p x
proof −
  have heap-array-map p (λ-. c) ∘ w' = w' ∘ heap-array-map p (λ-. c) for c
    apply (rule comp-commute-of-fold)
    apply (subst heap-array-map-def[abs-def], rule refl)
    apply (simp add: w' list-all-iff)
    done
  then have ∗: w' (heap-array-map p (λ-. c) h) = heap-array-map p (λ-. c) (w' h)
for c h
    by (simp add: fun-eq-iff)

  have [simp]: heap-array (w' h) p = heap-array h p for h
    apply (subst array.write-same[symmetric, of λ-. heap-array h p, OF refl])
    apply (subst ∗)
    apply (subst array.read-write-same)
    apply (rule refl)
    done
  show ?thesis
    apply (rule ext)
    subgoal for h
      apply (clarsimp simp add: fun-eq-iff)
      apply (subst (1 2) array.write-cong[where f'=λ-. x (heap-array h p)])
```

```
    apply (simp-all flip: *)
    done
  done
qed

end

lemma (in open-types) valid-array-of-ptr-valid-arrayI:
  fixes r :: 't ⇒ 'a::{xmem-type, array-outer-max-size} ptr ⇒ 'a
  fixes p :: ('a['n::{finite, array-max-count}]) ptr
 assumes f: map-of 𝒯 (typ-uinfo-t TYPE('a['n])) = Some (array-fields CARD('n))
  assumes p: PTR-VALID('a['n]) h' p
  assumes v: ⋀p. PTR-VALID('a) h' p ⟹ v h p
  shows valid-array-base.valid-array v h p
  unfolding valid-array-base.valid-array-def
proof (intro allI impI v)
  fix n assume n: n < CARD('n)
  note fl-array = field-lookup-array[OF n, THEN field-lookup-typ-uinfo-t-Some]

  have [replicate n CHR ''1''] ∈ set (array-fields CARD('n))
    using n by (auto simp: array-fields-def)

  note * = ptr-valid-u-step[OF f this fl-array p[unfolded ptr-valid-def]]
  have field-offset-untyped (typ-uinfo-t TYPE('a['n])) [replicate n CHR ''1''] =
    n * size-of TYPE('a)
    by (simp add: field-offset-untyped-def fl-array)
  with * show PTR-VALID('a) h' (array-ptr-index p False n)
    by (simp add: n array-ptr-index-def ptr-add-def ptr-valid-def typ-uinfo-t-def)
qed

lemma (in open-types) valid-array-of-ptr-valid-array1[simp]:
  fixes r :: 't ⇒ 'a::{xmem-type, array-outer-max-size} ptr ⇒ 'a
  fixes p :: ('a['n::{finite, array-max-count}]) ptr
  assumes f:
    map-of 𝒯 (typ-uinfo-t TYPE('a['n])) = Some (array-fields CARD('n))
  assumes p: PTR-VALID('a['n]) (heap-typing h) p
  shows valid-array-base.valid-array (λh. PTR-VALID('a) (heap-typing h)) h p
  by (rule valid-array-of-ptr-valid-arrayI[of heap-typing h, OF f p])

lemma (in open-types) valid-array-of-ptr-valid-array2[simp]:
  fixes r :: 't ⇒ 'a::{xmem-type, array-inner-max-size} ptr ⇒ 'a
  fixes p :: ('a['n::{finite, array-max-count}]['m::{finite, array-max-count}]) ptr
  assumes f2:
    map-of 𝒯 (typ-uinfo-t TYPE('a['n]['m])) = Some (array-fields CARD('m))
  assumes f1:
    map-of 𝒯 (typ-uinfo-t TYPE('a['n])) = Some (array-fields CARD('n))
  assumes p: PTR-VALID('a['n]['m]) (heap-typing h) p
  shows valid-array-base.valid-array (valid-array-base.valid-array
    (λh. PTR-VALID('a) (heap-typing h))) h p
```

apply (*rule valid-array-of-ptr-valid-arrayI*[*of heap-typing h*, *OF f2 p*])
apply (*rule valid-array-of-ptr-valid-array1*[*OF f1*])
apply *assumption*
done

lemma (**in** *open-types*) *ptr-valid-unsigned*[*simp*]:
PTR-VALID('*a*::*len8 signed word*) *h p* ⟷
PTR-VALID('*a*::*len8 word*) *h* (*PTR-COERCE*('*a signed word* → '*a word*) *p*)
by (*simp add*: *typ-uinfo-t-signed-word-word-conv ptr-valid-def*)

lemma *ucast-zero-word*:
UCAST('*a*::*len8* → '*a signed*) *ZERO*('*a word*) = *ZERO*('*a signed word*)
using *len8-bytes*[**where** '*a*='*a*]
apply (*simp add*: *zero-def*)
apply (*subst from-bytes-signed-word*)
apply (*simp-all add*: *size-of-def typ-info-word ucast-ucast-id*)
done

definition *signed-heap*::
('*s* ⇒ '*a*::*len word ptr* ⇒ '*a word*) ⇒ ('*s* ⇒ '*a signed word ptr* ⇒ '*a signed word*)
where
signed-heap h s = *UCAST* ('*a*::*len* → '*a signed*) *o* (*h s*) *o PTR-COERCE*('*a signed word* → '*a word*)

lemma *signed-heap-apply*[*simp*]:
signed-heap h s p = *UCAST* ('*a*::*len* → '*a signed*) (*h s* (*PTR-COERCE*('*a signed word* → '*a word*) *p*))
by (*simp add*: *signed-heap-def*)

lemma *signed-typ-heap-simulation-of-typ-heap-simulation*:
fixes *r* :: - ⇒ - ⇒ '*a*::*len8 word*
assumes *r*:
typ-heap-simulation-open-types 𝒯 *st r w v t-hrs t-hrs-update heap-typing heap-typing-upd*
shows *typ-heap-simulation-open-types* 𝒯 *st*
(*signed-heap r*)
(λ*p m*. *w* (*PTR-COERCE*('*a signed word* → '*a word*) *p*) (λ*w*. *UCAST*('*a signed* → '*a*) (*m* (*UCAST*('*a* → '*a signed*) *w*))))
(λ*h p*. *v h* (*PTR-COERCE*('*a signed word* → '*a word*) *p*))
t-hrs t-hrs-update heap-typing heap-typing-upd
proof −
interpret *typ-heap-simulation-open-types* 𝒯 *st r w v t-hrs t-hrs-update heap-typing heap-typing-upd*
by *fact*
interpret *cast*: *pointer-lense*
signed-heap r
λ*p m*. *w* (*PTR-COERCE*('*a signed word* → '*a word*) *p*)
(λ*w*. *UCAST*('*a signed* → '*a*) (*m* (*UCAST*('*a* → '*a signed*) *w*)))
unfolding *signed-heap-def comp-def*
by (*rule pointer-lense-ucast-signed*) *unfold-locales*

have [*simp*]: *c-guard p* $\longleftrightarrow$ *c-guard* (*PTR-COERCE*('*a signed word* $\rightarrow$ '*a word*) *p*) **for** *p*
apply (*intro c-guard-ptr-coerce*[*symmetric*])
apply (*simp-all add*: *size-of-signed-word align-of-signed-word*)
done
note *scast-ucast-norm*[*simp del*]
note *ucast-ucast-id*[*simp*]
show *?thesis*
apply *unfold-locales*
apply (*simp-all add*: *ptr-valid-unsigned ptr-valid-imp-v read-commutes h-val-signed-word write-padding-commutes heap-typing-upd-write-commute*)
apply (*subst heap-update-padding-signed-word*(*1*)[*symmetric*])
apply (*subst write-padding-commutes*)
apply (*simp-all add*: *size-of-signed-word scast-ucast-down-same valid-implies-c-guard*)
apply (*rule valid-same-typ-desc*, *simp*)
apply (*simp add*: *sim-stack-stack-byte-zero'*)
apply (*simp add*: *ucast-zero-word*)
done
qed

lemma *array-typ-heap-simulation-of-typ-heap-simulation*:
fixes *r* :: - $\Rightarrow$ - $\Rightarrow$ '*a*::{*stack-type*, *xmem-type*, *array-outer-max-size*}
assumes *typ-heap-simulation-open-types* $\mathcal{T}$ *st r w v t-hrs t-hrs-update heap-typing heap-typing-upd*
assumes *f*: *map-of* $\mathcal{T}$ (*typ-uinfo-t TYPE*('*a*['*n*::{*finite*, *array-max-count*}])) =
Some (*array-fields CARD*('*n*))
shows
typ-heap-simulation-open-types $\mathcal{T}$ *st*
(*pointer-array-lense.heap-array r* :: - $\Rightarrow$ ('*a*['*n*]) *ptr* $\Rightarrow$ -)
(*pointer-array-lense.heap-array-map r w*)
(*valid-array-base.valid-array v*)
t-hrs t-hrs-update heap-typing heap-typing-upd
proof −
interpret *sim*: *typ-heap-simulation-open-types* $\mathcal{T}$ *st r w v t-hrs t-hrs-update* **by** *fact*
interpret *array-typ-heap-simulation st r w v t-hrs t-hrs-update* **..**
show *?thesis*
proof
fix *p* :: ('*a*['*n*]) *ptr* **and** *d f*
have *heap-array-map p f o heap-typing-upd d* = *heap-typing-upd d o heap-array-map p f*
by (*intro disjnt-comp-heap-array-map*)
(*simp add*: *fun-eq-iff sim.heap-typing-upd-write-commute*)
then show *heap-typing-upd d* (*heap-array-map p f h*) = *heap-array-map p f* (*heap-typing-upd d h*) **for** *h*
by (*simp add*: *fun-eq-iff*)
show *heap-array* (*st s*) *p* = *ZERO*(-)
if *p*: $\forall$ *a*$\in$*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*PTR*(*stack-byte*) *a*) **for** *s*

apply (*intro array-ext*)
apply (*simp add*: *array-index-zero*)
apply (*intro sim.sim-stack-stack-byte-zero'*)
subgoal for *i*
using *ptr-span-array-ptr-index-subset-ptr-span*[*of i p False*] *p*
apply *auto*
done
done
show *sim.ptr-valid* (*heap-typing h*) *p* ⟹ *valid-array h p* **for** *h*
by (*rule sim.valid-array-of-ptr-valid-arrayI*[*OF f*, *of heap-typing h*])
(*simp-all add*: *sim.ptr-valid-imp-v*)
qed *simp*
qed

lemma (**in** *typ-heap-simulation-open-types*) *stack-simulation-heap-typingI*:
assumes *hrs*: *heap-typing-simulation* 𝒯 *t-hrs t-hrs-update heap-typing heap-typing-upd st*
assumes *sim-stack-alloc-heap-typing*:
⋀*p d s n*.
(*p*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*) (*hrs-htd* (*t-hrs s*)) ⟹
st (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*]) ∘ *hrs-htd-update* (λ-. *d*)) *s*) =
(*heap-typing-upd* (λ-. *d*) (*st s*))
assumes *sim-stack-release-heap-typing*:
⋀(*p*::'*a ptr*) *s n*. (⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*p* +$_p$ *int i*)) ⟹
st (*t-hrs-update* (*hrs-htd-update* (*stack-releases n p*)) *s*) =
heap-typing-upd (*stack-releases n p*)
(*st* (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*])) *s*))
shows *stack-simulation-heap-typing st r w t-hrs t-hrs-update heap-typing heap-typing-upd* 𝒮 𝒯
proof −
interpret *hrs*: *heap-typing-simulation* 𝒯 *t-hrs t-hrs-update heap-typing heap-typing-upd st* **by** *fact*
interpret *write-simulation t-hrs t-hrs-update st* λ*t p*. *open-types.ptr-valid* 𝒯 (*heap-typing t*) *p w*
apply (*rule hrs.write-simulation-alt*)
apply *simp*
apply (*simp-all add*: *ptr-valid.ptr-valid-c-guard ptr-valid-imp-v write-commutes read-commutes*)
done
interpret *typ-heap-simulation*
st r w λ*t p*. *open-types.ptr-valid* 𝒯 (*heap-typing t*) *p t-hrs t-hrs-update*
by *unfold-locales*
(*simp-all add*: *ptr-valid.ptr-valid-c-guard ptr-valid-imp-v write-commutes read-commutes*)
show *?thesis*
proof

qed (*simp-all add*: *sim-stack-stack-byte-zero'* *sim-stack-alloc-heap-typing* *sim-stack-release-heap-typing*)
qed

end

theory *NatBitwise*
imports
More-Lib
begin

lemma *lsb-nat-def*:
‹*lsb n* = *lsb* (*int n*)›
by (*simp add*: *bit-simps*)

instantiation *nat* :: *msb*
begin

definition
msb x = *msb* (*int x*)

instance ..

end

lemma *not-msb-nat*:
‹¬ *msb n*› **for** *n* :: *nat*
by (*simp add*: *msb-nat-def msb-int-def*)

lemma *set-bit-nat-def*:
‹*set-bit x y z* = *nat* (*set-bit* (*int x*) *y z*)›
by (*rule bit-eqI*) (*simp add*: *bit-simps bin-sc-pos*)

lemma *nat-2p-eq-shiftl*:
(*2*::*nat*)^*x* = *1* << *x*
by *simp*

lemma *shiftl-nat-def*:
(*x*::*nat*) << *y* = *nat* (*int x* << *y*)
by (*simp add*: *nat-int-mul push-bit-eq-mult shiftl-def*)

lemma *nat-shiftl-less-cancel*:
n ≤ *m* ⟹ ((*x* :: *nat*) << *n* < *y* << *m*) = (*x* < *y* << (*m* − *n*))
apply (*simp add*: *nat-int-comparison*(*2*) *shiftl-nat-def shiftl-def*)
by (*metis int-shiftl-less-cancel shiftl-def*)

lemma *nat-shiftl-lt-2p-bits*:
(*x*::*nat*) < *1* << *n* ⟹ ∀ *i* ≥ *n*. ¬ *x* !! *i*
apply (*clarsimp simp*: *shiftl-nat-def zless-nat-eq-int-zless*
dest!: *le-Suc-ex*)
by (*metis bit-take-bit-iff not-add-less1 take-bit-nat-eq-self-iff*)

lemmas *nat-eq-test-bit* = *bit-eq-iff*
lemmas *nat-eq-test-bitI* = *bit-eq-iff*[*THEN iffD2*, *rule-format*]

end

Chapter 22

WA phase: Word Abstraction

theory *WordAbstract*
imports
L2Peephole
NatBitwise
begin

22.1 Basic Definitions

definition *WORD-MAX* $x \equiv ((2 \hat{} (\textit{len-of}\ x - 1)) - 1 :: \textit{int})$
definition *WORD-MIN* $x \equiv (- (2 \hat{} (\textit{len-of}\ x - 1)) :: \textit{int})$
definition *UWORD-MAX* $x \equiv ((2 \hat{} (\textit{len-of}\ x)) - 1 :: \textit{nat})$

lemma *WORD-values* [*simplified*]:
$\textit{WORD-MAX}\ (\textit{TYPE}(\ 8\ \textit{signed})) = (2 \hat{}\ 7 - 1)$
$\textit{WORD-MAX}\ (\textit{TYPE}(16\ \textit{signed})) = (2 \hat{}\ 15 - 1)$
$\textit{WORD-MAX}\ (\textit{TYPE}(32\ \textit{signed})) = (2 \hat{}\ 31 - 1)$
$\textit{WORD-MAX}\ (\textit{TYPE}(64\ \textit{signed})) = (2 \hat{}\ 63 - 1)$

$\textit{WORD-MIN}\ (\textit{TYPE}(\ 8\ \textit{signed})) = - (2 \hat{}\ 7)$
$\textit{WORD-MIN}\ (\textit{TYPE}(16\ \textit{signed})) = - (2 \hat{}\ 15)$
$\textit{WORD-MIN}\ (\textit{TYPE}(32\ \textit{signed})) = - (2 \hat{}\ 31)$
$\textit{WORD-MIN}\ (\textit{TYPE}(64\ \textit{signed})) = - (2 \hat{}\ 63)$

$\textit{UWORD-MAX}\ (\textit{TYPE}(\ 8)) = (2 \hat{}\ 8 - 1)$
$\textit{UWORD-MAX}\ (\textit{TYPE}(16)) = (2 \hat{}\ 16 - 1)$
$\textit{UWORD-MAX}\ (\textit{TYPE}(32)) = (2 \hat{}\ 32 - 1)$
$\textit{UWORD-MAX}\ (\textit{TYPE}(64)) = (2 \hat{}\ 64 - 1)$
by (*auto simp*: *WORD-MAX-def WORD-MIN-def UWORD-MAX-def*)

lemmas *WORD-values-add1* =
WORD-values [*THEN arg-cong* [**where** $f{=}\lambda x.\ x + 1$],
simplified semiring-norm, *simplified numeral-One*]

lemmas *WORD-values-minus1* =

WORD-values [*THEN arg-cong* [**where** *f*=λ*x*. *x* − *1*],
simplified semiring-norm, *simplified numeral-One nat-numeral*]

lemmas *WORD-values-fold* [*L1unfold*] =
WORD-values [*symmetric*]
WORD-values-add1 [*symmetric*]
WORD-values-minus1 [*symmetric*]

lemma *WORD-signed-to-unsigned* [*simp*]:
WORD-MAX TYPE('*a signed*) = *WORD-MAX TYPE*('*a*::*len*)
WORD-MIN TYPE('*a signed*) = *WORD-MIN TYPE*('*a*::*len*)
UWORD-MAX TYPE('*a signed*) = *UWORD-MAX TYPE*('*a*::*len*)
by (*auto simp*: *WORD-MAX-def WORD-MIN-def UWORD-MAX-def*)

lemma *INT-MIN-comparisons* [*simp*]:
⟦ *a* ≤ − (*2* ^ (*len-of TYPE*('*a*) − *1*)) ⟧ ⟹ *a* ≤ *WORD-MIN* (*TYPE*('*a*::*len*))
a < − (*2* ^ (*len-of TYPE*('*a*) − *1*)) ⟹ *a* < *WORD-MIN* (*TYPE*('*a*::*len*))
a ≥ − (*2* ^ (*len-of TYPE*('*a*) − *1*)) ⟹ *a* ≥ *WORD-MIN* (*TYPE*('*a*::*len*))
a > − (*2* ^ (*len-of TYPE*('*a*) − *1*)) ⟹ *a* ≥ *WORD-MIN* (*TYPE*('*a*::*len*))
by (*auto simp*: *WORD-MIN-def*)

lemma *INT-MAX-comparisons* [*simp*]:
a ≤ (*2* ^ (*len-of TYPE*('*a*) − *1*)) − *1* ⟹ *a* ≤ *WORD-MAX* (*TYPE*('*a*::*len*))
a < (*2* ^ (*len-of TYPE*('*a*) − *1*)) − *1* ⟹ *a* < *WORD-MAX* (*TYPE*('*a*::*len*))
a ≥ (*2* ^ (*len-of TYPE*('*a*) − *1*)) − *1* ⟹ *a* ≥ *WORD-MAX* (*TYPE*('*a*::*len*))
a > (*2* ^ (*len-of TYPE*('*a*) − *1*)) − *1* ⟹ *a* ≥ *WORD-MAX* (*TYPE*('*a*::*len*))
by (*auto simp*: *WORD-MAX-def*)

lemma *UINT-MAX-comparisons* [*simp*]:
x ≤ (*2* ^ (*len-of TYPE*('*a*))) − *1* ⟹ *x* ≤ *UWORD-MAX* (*TYPE*('*a*::*len*))
x < (*2* ^ (*len-of TYPE*('*a*))) − *1* ⟹ *x* ≤ *UWORD-MAX* (*TYPE*('*a*::*len*))
x ≥ (*2* ^ (*len-of TYPE*('*a*))) − *1* ⟹ *x* ≥ *UWORD-MAX* (*TYPE*('*a*::*len*))
x > (*2* ^ (*len-of TYPE*('*a*))) − *1* ⟹ *x* > *UWORD-MAX* (*TYPE*('*a*::*len*))
by (*auto simp*: *UWORD-MAX-def*)

lemma *is-up-SCAST-same-signed* [*simp*]: *is-up* (*SCAST* (('*a*::*len*) → '*a signed*))
unfolding *is-up*
by *simp*

lemma *sint-ucast-signed* [*simp*, *L2opt*]:*sint* (*UCAST* ('*a*::*len* → '*a signed*) *x*) = *sint x*
using *is-up-SCAST-same-signed scast-ucast-norm*(*2*) *sint-up-scast*
by (*metis scast-scast-id*(*1*))

22.2 Abstracting values and expressions

definition *introduce-typ-abs-fn f* ≡ *True*

declare *introduce-typ-abs-fn-def* [*simp*]

lemma *introduce-typ-abs-fn*:
introduce-typ-abs-fn f
by *simp*

definition
abstract-bool-binop :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('c \Rightarrow 'a)$
$\Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('c \Rightarrow 'c \Rightarrow bool) \Rightarrow bool$
where
abstract-bool-binop P f X X' $\equiv \forall\, a\ b.\ P\ (f\ a)\ (f\ b) \longrightarrow (X'\ a\ b = X\ (f\ a)\ (f\ b))$

definition
abstract-binop :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('c \Rightarrow 'a)$
$\Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('c \Rightarrow 'c \Rightarrow 'c) \Rightarrow bool$
where
abstract-binop P f X X' $\equiv \forall\, a\ b.\ P\ (f\ a)\ (f\ b) \longrightarrow (f\ (X'\ a\ b) = X\ (f\ a)\ (f\ b))$

definition *abstract-val P a f b* $\equiv P \longrightarrow (a = f\ b)$

definition *abs-var a f b* $\equiv$ *abstract-val True a f b*

lemmas *basic-abstract-defs* =
abstract-bool-binop-def
abstract-binop-def
abstract-val-def
abs-var-def

lemma *abstract-val-trivial*:
abstract-val True $(f\ b)\ f\ b$
by (*simp add*: *basic-abstract-defs*)

lemma *abstract-binop-is-abstract-val*:
abstract-binop P f X X' $= (\forall\, a\ b.$ *abstract-val* $(P\ (f\ a)\ (f\ b))\ (X\ (f\ a)\ (f\ b))\ f\ (X'\ a\ b))$
by (*auto simp add*: *basic-abstract-defs*)

lemma *abstract-expr-bool-binop*:
$[\![$ *abstract-bool-binop E f X X'*;
introduce-typ-abs-fn f;
abstract-val P a f a';
abstract-val Q b f b' $]\!] \Longrightarrow$
abstract-val $(P \wedge Q \wedge E\ a\ b)\ (X\ a\ b)$ *id* $(X'\ a'\ b')$

by (*clarsimp simp add*: *basic-abstract-defs*)

lemma *abstract-expr-binop*:
⟦ *abstract-binop E f X X'*;
abstract-val P a f a';
abstract-val Q b f b' ⟧ ⟹
abstract-val (*P* ∧ *Q* ∧ *E a b*) (*X a b*) *f* (*X' a' b'*)
by (*clarsimp simp add*: *basic-abstract-defs*)

lemma *unat-abstract-bool-binops*:
abstract-bool-binop (λ- -. *True*) (*unat* :: ('*a*::*len*) *word* ⇒ *nat*) (<) (<)
abstract-bool-binop (λ- -. *True*) (*unat* :: ('*a*::*len*) *word* ⇒ *nat*) (≤) (≤)
abstract-bool-binop (λ- -. *True*) (*unat* :: ('*a*::*len*) *word* ⇒ *nat*) (=) (=)
by (*auto simp*: *word-less-nat-alt word-le-nat-alt eq-iff basic-abstract-defs*)

lemmas *unat-mult-simple* = *iffD1* [*OF unat-mult-lem* [*unfolded word-bits-len-of*]]

lemma *le-to-less-plus-one*:
((*a*::*nat*) ≤ *b*) = (*a* < *b* + *1*)
by *arith*

lemma *unat-abstract-binops*:
abstract-binop (λ*a b*. *a* + *b* ≤ *UWORD-MAX TYPE*('*a*::*len*)) (*unat* :: '*a word* ⇒ *nat*) (+) (+)
abstract-binop (λ*a b*. *a* ∗ *b* ≤ *UWORD-MAX TYPE*('*a*)) (*unat* :: '*a word* ⇒ *nat*) (∗) (∗)
abstract-binop (λ*a b*. *a* ≥ *b*) (*unat* :: '*a word* ⇒ *nat*) (−) (−)
abstract-binop (λ*a b*. *True*) (*unat* :: '*a word* ⇒ *nat*) (*div*) (*div*)
abstract-binop (λ*a b*. *True*) (*unat* :: '*a word* ⇒ *nat*) (*mod*) (*mod*)
by (*auto simp*: *unat-plus-if' unat-div unat-mod UWORD-MAX-def le-to-less-plus-one WordAbstract.unat-mult-simple word-bits-def unat-sub word-le-nat-alt basic-abstract-defs*)

lemma *unat-of-int*:
⟦*i* ≥ *0*; *i* < *2* ^ *LENGTH*('*a*)⟧ ⟹ *unat* (*of-int i* :: '*a*::*len word*) = *nat i*
by (*metis nat-less-numeral-power-cancel-iff of-nat-nat unat-of-nat-len*)

lemma *unat-of-int-signed*:
⟦*i* ≥ *0*; *i* < *2* ^ *LENGTH*('*a*)⟧ ⟹ *unat* (*of-int i* :: '*a*::*len signed word*) = *nat i*
by (*simp add*: *unat-of-int*)

lemma *nat-sint*:
0 <=*s* (*x* :: '*a*::*len signed word*) ⟹ *nat* (*sint x*) = *unat x*
apply (*subst unat-of-int-signed*[**where** '*a*='*a*, *symmetric*])
apply (*simp add*: *word-sle-def*)
apply (*rule less-trans*[*OF sint-lt*])
apply *simp*
apply *simp*

done

lemma *int-unat-nonneg*:
0 <=s (x :: 'a::len signed word) ⟹ int (unat x) = sint x
by (*simp add*: *int-unat word-sle-msb-le sint-eq-uint*)

lemma *uint-sint-nonneg*:
0 <=s (x :: 'a::len signed word) ⟹ uint x = sint x
by (*simp add*: *int-unat word-sle-msb-le sint-eq-uint*)

lemma *unat-bitwise-abstract-binops*:
abstract-binop (λa b. True) (unat :: 'a::len word ⇒ nat) Bit-Operations.and Bit-Operations.and
abstract-binop (λa b. True) (unat :: 'a::len word ⇒ nat) Bit-Operations.or Bit-Operations.or
abstract-binop (λa b. True) (unat :: 'a::len word ⇒ nat) Bit-Operations.xor Bit-Operations.xor
apply (*simp add*: *unsigned-and-eq uint-nat unat-of-int basic-abstract-defs*)
apply (*simp add*: *unsigned-or-eq uint-nat unat-of-int basic-abstract-defs*)
apply (*simp add*: *unsigned-xor-eq uint-nat unat-of-int basic-abstract-defs*)
done

lemma *abstract-val-unsigned-bitNOT*:
abstract-val P x unat (x' :: 'a::len word) ⟹
abstract-val P (UWORD-MAX TYPE('a) − x) unat (NOT x')
apply (*clarsimp simp*: *UWORD-MAX-def NOT-eq basic-abstract-defs*)
by (*metis nat-le-Suc-less diff-Suc-eq-diff-pred mask-eq-sum-exp mask-eq-sum-exp-nat*

minus-diff-commute unat-lt2p unat-minus-one-word unat-sub-if-size unsigned-0)

lemma *snat-abstract-bool-binops*:
abstract-bool-binop (λ- -. True) (sint :: ('a::len) signed word ⇒ int) (<) (word-sless)
abstract-bool-binop (λ- -. True) (sint :: 'a signed word ⇒ int) (≤) (word-sle)
abstract-bool-binop (λ- -. True) (sint :: 'a signed word ⇒ int) (=) (=)
by (*auto simp*: *word-sless-def word-sle-def less-le basic-abstract-defs*)

lemma *snat-abstract-binops*:
abstract-binop (λa b. WORD-MIN TYPE('a::len) ≤ a + b ∧ a + b ≤ WORD-MAX TYPE('a)) (sint :: 'a signed word ⇒ int) (+) (+)
*abstract-binop (λa b. WORD-MIN TYPE('a) ≤ a * b ∧ a * b ≤ WORD-MAX TYPE('a)) (sint :: 'a signed word ⇒ int) (*) (*)*
abstract-binop (λa b. WORD-MIN TYPE('a) ≤ a − b ∧ a − b ≤ WORD-MAX TYPE('a)) (sint :: 'a signed word ⇒ int) (−) (−)
abstract-binop (λa b. WORD-MIN TYPE('a) ≤ a sdiv b ∧ a sdiv b ≤ WORD-MAX TYPE('a)) (sint :: 'a signed word ⇒ int) (sdiv) (sdiv)
abstract-binop (λa b. WORD-MIN TYPE('a) ≤ a smod b ∧ a smod b ≤ WORD-MAX TYPE('a)) (sint :: 'a signed word ⇒ int) (smod) (smod)
by (*auto simp*: *signed-arith-sint word-size WORD-MIN-def WORD-MAX-def basic-abstract-defs*)

lemma *sint-bitwise-abstract-binops*:
abstract-binop (λ*a b*. *True*) (*sint* :: *'a*::*len signed word* ⇒ *int*) *Bit-Operations.and Bit-Operations.and*
abstract-binop (λ*a b*. *True*) (*sint* :: *'a*::*len signed word* ⇒ *int*) *Bit-Operations.or Bit-Operations.or*
abstract-binop (λ*a b*. *True*) (*sint* :: *'a*::*len signed word* ⇒ *int*) *Bit-Operations.xor Bit-Operations.xor*
apply (*fastforce intro*: *int-eq-test-bitI*
simp: *nth-sint bin-nth-ops basic-abstract-defs*)+
done

lemma *abstract-val-signed-bitNOT*:
abstract-val P x sint (*x'* :: *'a*::*len signed word*) ⟹
abstract-val P (*NOT x*) *sint* (*NOT x'*)
by (*auto intro*: *int-eq-test-bitI*
simp add: *nth-sint bin-nth-ops word-nth-neq basic-abstract-defs min-less-iff-disj*)

lemma *abstract-val-signed-unary-minus*:
⟦ *abstract-val P r sint r'* ⟧ ⟹
abstract-val (*P* ∧ (− *r*) ≤ *WORD-MAX TYPE*(*'a*)) (− *r*) *sint* (− (*r'* :: (*'a* :: *len*) *signed word*))
apply (*clarsimp simp add*: *basic-abstract-defs*)
using *sint-range-size* [**where** *w*=*r'*]
apply −
apply (*subst signed-arith-sint*)
apply (*clarsimp simp*: *word-size WORD-MAX-def*)
apply *simp*
done

lemma *bang-big-nonneg*:
⟦ *0* <=*s* (*x*::*'a*::*len signed word*); *n* ≥ *size x* − *1* ⟧ ⟹ (*x* !! *n*) = *False*
apply (*cases n* = *size x* − *1*)
apply (*simp add*: *word-size msb-nth*[**where** *'a*=*'a signed*, *symmetric*, *simplified*] *word-sle-msb-le*)
apply (*simp add*: *test-bit-bl*)
apply *arith*
done

lemma *int-shiftr-nth*[*simp*]:
(*i* >> *n*) !! *m* = *i* !! (*n* + *m*) **for** *i* :: *int*
by (*simp add*: *shiftr-def bin-nth-shiftr*)

lemma *int-shiftl-nth*[*simp*]:
(*i* << *n*) !! *m* = (*n* ≤ *m* ∧ *i* !! (*m* − *n*)) **for** *i* :: *int*
by (*simp add*: *shiftl-def bin-nth-shiftl*)

lemma *sint-shiftr-nonneg*:
⟦ *0 <=s* (*x* :: *'a*::*len signed word*); *0* ≤ *n*; *n* < *LENGTH*(*'a*) ⟧ ⟹ *sint* (*x* >> *n*) = *sint x* >> *n*
apply (*rule int-eq-test-bitI*)
apply (*clarsimp simp*: *bang-big-nonneg*[*simplified word-size*] *nth-sint nth-shiftr field-simps*
simp del: *bit-signed-iff*)
done

lemma *abstract-val-unsigned-unary-minus*:
⟦ *abstract-val P r unat r'* ⟧ ⟹
abstract-val P (*if r = 0 then 0 else UWORD-MAX TYPE*(*'a*::*len*) + *1* − *r*) *unat* (− (*r'* :: *'a word*))
by (*clarsimp simp*: *unat-minus' word-size unat-eq-zero UWORD-MAX-def basic-abstract-defs*)

lemma *abstract-val-signed-shiftr-signed*:
⟦ *abstract-val Px x sint* (*x'* :: (*'a* :: *len*) *signed word*);
abstract-val Pn n sint (*n'* :: (*'b* :: *len*) *signed word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *x* ∧ *0* ≤ *n* ∧ *n* < *int LENGTH*(*'a*))
(*x* >> (*nat n*)) *sint* (*x'* >> (*unat n'*))
apply (*clarsimp simp only*: *abstract-val-def*)
apply (*subst nat-sint*, *simp add*: *word-sle-def*)
apply (*subst sint-shiftr-nonneg*)
apply (*simp add*: *word-sle-def*)
apply *simp*
apply (*subst SMT.nat-int-comparison*(*2*))
apply (*subst int-unat-nonneg*)
apply (*simp add*: *word-sle-def*)
apply *assumption*
apply (*rule refl*)
done

lemma *abstract-val-signed-shiftr-unsigned*:
⟦ *abstract-val Px x sint* (*x'* :: (*'a* :: *len*) *signed word*);
abstract-val Pn n unat (*n'* :: (*'b* :: *len*) *word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *x* ∧ *n* < *LENGTH*(*'a*))
(*x* >> *n*) *sint* (*x'* >> *unat n'*)
apply (*clarsimp simp*: *shiftr-int-def basic-abstract-defs*)
apply (*subst sint-shiftr-nonneg*)
apply (*simp add*: *word-sle-def*)
apply *simp*
apply *assumption*
apply (*clarsimp simp*: *shiftr-int-def*)
done

lemma *foo*: $\neg\ n - i < LENGTH('a::len) - Suc\ 0 \Longrightarrow$
$n < LENGTH('a) - Suc\ 0 = False$
apply *simp*
done

lemma *sint-shiftl-nonneg*:
$[\![\ 0 <= s\ (x :: 'a::len\ signed\ word);\ n < LENGTH('a);\ sint\ x << n < 2\hat{}(LENGTH('a) - 1)\ ]\!] \Longrightarrow$
$sint\ (x << n) = sint\ x << n$
apply (*rule int-eq-test-bitI*)
subgoal for *na*
apply (*simp add*: *nth-sint nth-shiftl word-sle-def int-shiftl-less-cancel int-2p-eq-shiftl*
bang-big-nonneg[*simplified word-size*]
del: *bit-signed-iff shiftl-1*)

apply (*intro impI iffI conjI*; (*solves simp*)*?*)
apply (*drule*(*1*) *int-shiftl-lt-2p-bits*[*rotated*])
apply (*clarsimp simp*: *nth-sint*)
apply (*drule-tac x=LENGTH('a) - 1 - n* **in** *spec*)
apply (*subgoal-tac LENGTH('a) - 1 - n < LENGTH('a) - 1*)
apply *simp*
apply *arith*
apply (*drule*(*1*) *int-shiftl-lt-2p-bits*[*rotated*])
apply (*clarsimp simp*: *nth-sint*)
apply (*drule-tac x=na - n* **in** *spec*)
apply *simp*
apply (*cases n = 0*)
apply (*simp add*: *word-sle-msb-le*[**where** *x=0*, *simplified word-sle-def*, *simplified*] *msb-nth*)
apply (*drule*(*1*) *int-shiftl-lt-2p-bits*[*rotated*])
apply (*clarsimp simp*: *nth-sint*)
apply (*drule-tac x=LENGTH('a) - 1 - n* **in** *spec*)
apply (*subgoal-tac LENGTH('a) - 1 - n < LENGTH('a) - 1*)
apply *simp*
apply *simp*
done
done

lemma *abstract-val-signed-shiftl-signed*:
$[\![$ *abstract-val Px x sint* $(x' :: ('a :: len)\ signed\ word);$
abstract-val Pn n sint $(n' :: ('b :: len)\ signed\ word)\ ]\!] \Longrightarrow$
abstract-val $(Px \wedge Pn \wedge 0 \leq x \wedge 0 \leq n \wedge n < int\ LENGTH('a) \wedge x << nat\ n < 2\hat{}(LENGTH('a) - 1))$
$(x << nat\ n)\ sint\ (x' << unat\ n')$
apply (*clarsimp simp add*: *basic-abstract-defs*)
by (*metis One-nat-def len-gt-0 nat-int nat-sint signed-0 sint-shiftl-nonneg word-sle-eq*

zless-nat-conj)

lemma *abstract-val-signed-shiftl-unsigned*:
⟦ *abstract-val Px x sint* (*x'* :: (*'a* :: *len*) *signed word*);
abstract-val Pn n unat (*n'* :: (*'b* :: *len*) *word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *x* ∧ *n* < *LENGTH*(*'a*) ∧ *x* << *n* < *2*^(*LENGTH*(*'a*) − *1*))
(*x* << *n*) *sint* (*x'* << *unat n'*)
by (*clarsimp simp*: *sint-shiftl-nonneg word-sle-def basic-abstract-defs*
nat-less-eq-zless[**where** *z*=*int LENGTH*(*'a*), *simplified*])

lemma *abstract-val-unsigned-shiftr-unsigned*:
⟦ *abstract-val Px x unat* (*x'* :: (*'a* :: *len*) *word*);
abstract-val Pn n unat (*n'* :: (*'a* :: *len*) *word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn*) (*x* >> *n*) *unat* (*x'* >> *unat n'*)
apply (*clarsimp simp add*: *basic-abstract-defs*)
apply (*simp add*: *shiftr-div-2n' shiftr-int-def*)
using *shiftr-eq-div* **by** *blast*

lemma *abstract-val-unsigned-shiftr-signed*:
⟦ *abstract-val Px x unat* (*x'* :: (*'a* :: *len*) *word*);
abstract-val Pn n sint (*n'* :: (*'b* :: *len*) *signed word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *n*) (*x* >> *nat n*) *unat* (*x'* >> *unat n'*)
apply (*clarsimp simp*: *shiftr-div-2n' shiftr-int-def basic-abstract-defs*)
by (*simp add*: *nat-sint shiftr-nat-def word-sle-eq*)

lemma *abstract-val-unsigned-shiftl-unsigned*:
⟦ *abstract-val Px x unat* (*x'* :: (*'a* :: *len*) *word*);
abstract-val Pn n unat (*n'* :: (*'b* :: *len*) *word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *n* < *LENGTH*(*'a*) ∧ *x* << *n* < *2*^*LENGTH*(*'a*))
(*x* << *n*) *unat* (*x'* << *unat n'*)
by (*clarsimp simp*: *shiftl-t2n Word-Lemmas-Internal.shiftl-nat-def unat-mult-simple field-simps basic-abstract-defs*)

lemma *abstract-val-unsigned-shiftl-signed*:
⟦ *abstract-val Px x unat* (*x'* :: (*'a* :: *len*) *word*);
abstract-val Pn n sint (*n'* :: (*'b* :: *len*) *signed word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *n* ∧ *n* < *int* (*LENGTH*(*'a*)) ∧ *x* << *nat n* < *2*^*LENGTH*(*'a*))
(*x* << *nat n*) *unat* (*x'* << *unat n'*)
apply (*clarsimp simp*: *shiftl-t2n Word-Lemmas-Internal.shiftl-nat-def unat-mult-simple field-simps basic-abstract-defs*)
apply (*simp add*: *sint-eq-uint word-msb-sint*)
by (*metis Word.of-nat-unat len-gt-0 nat-int unat-of-nat-len unat-power-lower word-arith-nat-mult*
zless-nat-conj)

lemma *signed-shiftl-c-guard-simp* :
⟦ *int bound* < *2*^*LENGTH*(*'a*); *a* ∗ *2*^*b* < *int bound*; *0* ≤ *a* ⟧ ⟹
unat (*of-int a* :: *'a*::*len word*) ∗ *2* ^ *b* < *bound*

apply (*subst unat-of-int*)
apply *assumption*
apply (*drule*(*1*) *less-trans*)
apply (*subgoal-tac a * 2^b < 2^LENGTH('a) * 2^b*)
apply *simp*
apply (*erule less-le-trans*)
apply *simp*
apply (*subgoal-tac nat* (*a * 2^b*) *< nat* (*int bound*))
apply (*simp add*: *nat-power-eq nat-mult-distrib*)
apply (*subst nat-mono-iff*)
apply (*rule le-less-trans*, *assumption*)
apply (*erule le-less-trans*[*rotated*])
apply (*simp add*: *mult-left-mono*[**where** *a=1*::*int*, *simplified*])
apply *simp*
done

lemmas *abstract-val-signed-ops* [*simplified simp-thms*] =
abstract-expr-bool-binop [*OF snat-abstract-bool-binops*(*1*)]
abstract-expr-bool-binop [*OF snat-abstract-bool-binops*(*2*)]
abstract-expr-bool-binop [*OF snat-abstract-bool-binops*(*3*)]
abstract-expr-binop [*OF snat-abstract-binops*(*1*)]
abstract-expr-binop [*OF snat-abstract-binops*(*2*)]
abstract-expr-binop [*OF snat-abstract-binops*(*3*)]
abstract-expr-binop [*OF snat-abstract-binops*(*4*)]
abstract-expr-binop [*OF snat-abstract-binops*(*5*)]
abstract-expr-binop [*OF sint-bitwise-abstract-binops*(*1*)]
abstract-expr-binop [*OF sint-bitwise-abstract-binops*(*2*)]
abstract-expr-binop [*OF sint-bitwise-abstract-binops*(*3*)]
abstract-val-signed-bitNOT
abstract-val-signed-unary-minus
abstract-val-signed-shiftr-signed
abstract-val-signed-shiftr-unsigned
abstract-val-signed-shiftl-signed
abstract-val-signed-shiftl-unsigned

lemmas *abstract-val-unsigned-ops* [*simplified simp-thms*] =
abstract-expr-bool-binop [*OF unat-abstract-bool-binops*(*1*)]
abstract-expr-bool-binop [*OF unat-abstract-bool-binops*(*2*)]
abstract-expr-bool-binop [*OF unat-abstract-bool-binops*(*3*)]
abstract-expr-binop [*OF unat-abstract-binops*(*1*)]
abstract-expr-binop [*OF unat-abstract-binops*(*2*)]
abstract-expr-binop [*OF unat-abstract-binops*(*3*)]
abstract-expr-binop [*OF unat-abstract-binops*(*4*)]
abstract-expr-binop [*OF unat-abstract-binops*(*5*)]
abstract-expr-binop [*OF unat-bitwise-abstract-binops*(*1*)]
abstract-expr-binop [*OF unat-bitwise-abstract-binops*(*2*)]
abstract-expr-binop [*OF unat-bitwise-abstract-binops*(*3*)]
abstract-val-unsigned-bitNOT
abstract-val-unsigned-unary-minus

abstract-val-unsigned-shiftr-signed
abstract-val-unsigned-shiftr-unsigned
abstract-val-unsigned-shiftl-signed
abstract-val-unsigned-shiftl-unsigned

lemma *mod-less*:
(*a* :: *nat*) < *c* ⟹ *a mod b* < *c*
by (*metis less-trans mod-less-eq-dividend order-leE*)

lemma *abstract-val-ucast*:
⟦ *introduce-typ-abs-fn* (*unat* :: ('*a*::*len*) *word* ⇒ *nat*);
abstract-val P v unat v' ⟧
⟹ *abstract-val* (*P* ∧ *v* ≤ *nat* (*WORD-MAX TYPE*('*a*)))
(*int v*) *sint* (*ucast* (*v'* :: '*a word*) :: '*a signed word*)
apply (*clarsimp simp*: *uint-nat* [*symmetric*] *basic-abstract-defs*)
apply (*subst sint-eq-uint*)
apply (*rule not-msb-from-less*)
apply (*clarsimp simp*: *word-less-nat-alt unat-ucast WORD-MAX-def le-to-less-plus-one*)
apply (*subst* (*asm*) *nat-diff-distrib*)
apply *simp*
apply *clarsimp*
apply *clarsimp*
apply (*metis of-nat-numeral nat-numeral nat-power-eq of-nat-0-le-iff*)
apply (*clarsimp simp*: *uint-up-ucast is-up*)
done

lemma *abstract-val-heap-sword-template*:
⟦ *introduce-typ-abs-fn* (*sint* :: ('*a*::*len*) *signed word* ⇒ *int*);
abstract-val P p' id p ⟧
⟹ *abstract-val P* (*sint* (*ucast* (*heap-get s p'* :: '*a word*) :: '*a signed word*))
sint (*ucast* (*heap-get s p*) :: '*a signed word*)
by (*simp add*: *basic-abstract-defs*)

lemma *abstract-val-scast*:
⟦ *introduce-typ-abs-fn* (*sint* :: ('*a*::*len*) *signed word* ⇒ *int*);
abstract-val P C' sint C ⟧
⟹ *abstract-val* (*P* ∧ *0* ≤ *C'*) (*nat C'*) *unat* (*scast* (*C* :: ('*a*::*len*) *signed*
word) :: ('*a*::*len*) *word*)
apply (*clarsimp simp*: *down-cast-same* [*symmetric*] *is-down unat-ucast basic-abstract-defs*)
apply (*subst sint-eq-uint*)
apply (*clarsimp simp*: *word-msb-sint*)
apply (*clarsimp simp*: *unat-def* [*symmetric*])
apply (*subst word-unat.norm-Rep* [*symmetric*])
apply *clarsimp*
done

lemma *abstract-val-scast-upcast*:
⟦ *len-of TYPE*('*a*::*len*) ≤ *len-of TYPE*('*b*::*len*);

abstract-val P C′ sint C ⟧
⟹ *abstract-val P* (*C′*) *sint* (*scast* (*C* :: *′a signed word*) :: *′b signed word*)
by (*clarsimp simp*: *down-cast-same* [*symmetric*] *sint-up-scast is-up basic-abstract-defs*)

lemma *abstract-val-scast-downcast*:
⟦ *len-of TYPE*(*′b*) < *len-of TYPE*(*′a*::*len*);
abstract-val P C′ sint C ⟧
⟹ *abstract-val P* (*sbintrunc* ((*len-of TYPE*(*′b*::*len*) − *1*)) *C′*) *sint* (*scast* (*C* :: *′a signed word*) :: *′b signed word*)
by (*metis Word.of-int-sint abstract-val-def len-signed word-sbin.inverse-norm*)

lemma *abstract-val-ucast-upcast*:
⟦ *len-of TYPE*(*′a*::*len*) ≤ *len-of TYPE*(*′b*::*len*);
abstract-val P C′ unat C ⟧
⟹ *abstract-val P* (*C′*) *unat* (*ucast* (*C* :: *′a word*) :: *′b word*)
by (*clarsimp simp*: *is-up unat-ucast-upcast basic-abstract-defs*)

lemma *abstract-val-ucast-downcast*:
⟦ *len-of TYPE*(*′b*::*len*) < *len-of TYPE*(*′a*::*len*);
abstract-val P C′ unat C ⟧
⟹ *abstract-val P* (*C′ mod* (*UWORD-MAX TYPE*(*′b*) + *1*)) *unat* (*ucast* (*C* :: *′a word*) :: *′b word*)
apply (*clarsimp simp*: *scast-def sint-uint UWORD-MAX-def basic-abstract-defs*)
unfolding *ucast-def unat-def*
apply (*subst int-word-uint*)
apply (*metis* (*mono-tags*) *uint-mod uint-power-lower unat-def unat-mod unat-power-lower*)
done

definition
valid-typ-abs-fn (*P* :: *′a* ⇒ *bool*) (*Q* :: *′a* ⇒ *bool*) (*A* :: *′c* ⇒ *′a*) (*C* :: *′a* ⇒ *′c*) ≡
(∀ *v*. *P v* ⟶ *A* (*C v*) = *v*) ∧ (∀ *v*. *Q* (*A v*) ⟶ *C* (*A v*) = *v*)

declare *valid-typ-abs-fn-def* [*simp*]

lemma *valid-typ-abs-fn-id*:
valid-typ-abs-fn (λ-. *True*) (λ-. *True*) *id id*
by *clarsimp*

lemma *valid-typ-abs-fn-unit*:
valid-typ-abs-fn (λ-. *True*) (λ-. *True*) *id* (*id* :: *unit* ⇒ *unit*)
by *clarsimp*

lemma *valid-typ-abs-fn-unat*:
valid-typ-abs-fn (λ*v*. *v* ≤ *UWORD-MAX TYPE*(*′a*::*len*)) (λ-. *True*) (*unat* :: *′a word* ⇒ *nat*) (*of-nat* :: *nat* ⇒ *′a word*)
supply *unsigned-of-nat* [*simp del*]
by (*clarsimp simp*: *unat-of-nat-eq UWORD-MAX-def le-to-less-plus-one*)

lemma *valid-typ-abs-fn-sint*:
valid-typ-abs-fn (λv. *WORD-MIN TYPE*('a::*len*) $\leq v \wedge v \leq$ *WORD-MAX TYPE*('a)) (λ-. *True*) (*sint* :: 'a *signed word* $\Rightarrow$ *int*) (*of-int* :: *int* $\Rightarrow$ 'a *signed word*)
by (*clarsimp simp*: *sint-of-int-eq WORD-MIN-def WORD-MAX-def*)

lemma *valid-typ-abs-fn-tuple*:
⟦ *valid-typ-abs-fn P-a Q-a abs-a conc-a*; *valid-typ-abs-fn P-b Q-b abs-b conc-b* ⟧ $\Longrightarrow$
valid-typ-abs-fn (λ(*a*, *b*). *P-a a* $\wedge$ *P-b b*) (λ(*a*, *b*). *Q-a a* $\wedge$ *Q-b b*) (*map-prod abs-a abs-b*) (*map-prod conc-a conc-b*)
by *clarsimp*

lemma *valid-typ-abs-fn-tuple-split*:
⟦ *valid-typ-abs-fn P-a Q-a abs-a conc-a*; *valid-typ-abs-fn P-b Q-b abs-b conc-b* ⟧ $\Longrightarrow$
valid-typ-abs-fn (λ(*a*, *b*). *P-a a* $\wedge$ *P-b b*) (λ(*a*, *b*). *Q-a a* $\wedge$ *Q-b b*) (λ(*a*, *b*). (*abs-a a*, *abs-b b*)) (*map-prod conc-a conc-b*)
by *clarsimp*

lemma *introduce-typ-abs-fn-tuple*:
⟦ *introduce-typ-abs-fn abs-a*; *introduce-typ-abs-fn abs-b* ⟧ $\Longrightarrow$
introduce-typ-abs-fn (*map-prod abs-a abs-b*)
by *clarsimp*

lemma *valid-typ-abs-fn-sum*:
⟦ *valid-typ-abs-fn P-a Q-a abs-a conc-a*; *valid-typ-abs-fn P-b Q-b abs-b conc-b* ⟧ $\Longrightarrow$
valid-typ-abs-fn (*case-sum P-a P-b*) (*case-sum Q-a Q-b*) (*map-sum abs-a abs-b*) (*map-sum conc-a conc-b*)
by (*auto simp add*: *map-sum-def split*: *sum.splits*)

lemma *introduce-typ-abs-fn-sum*:
⟦ *introduce-typ-abs-fn abs-a*; *introduce-typ-abs-fn abs-b* ⟧ $\Longrightarrow$
introduce-typ-abs-fn (*map-sum abs-a abs-b*)
by *clarsimp*

22.3 Refinement Lemmas

named-theorems *word-abs*

definition
corresTA P rx ex A C $\equiv$ *corresXF* ($\lambda s.\ s$) ($\lambda r\ s.\ rx\ r$) ($\lambda r\ s.\ ex\ r$) *P A C*

definition *rel-word-abs ex rx* $\equiv$ *rel-xval* ($\lambda c\ a.\ a = ex\ c$) ($\lambda c\ a.\ a = rx\ c$)

lemma *rel-word-abs-simps*[*simp*]:
rel-word-abs ex rx (*Result rc*) (*Exn la*) = *False*
rel-word-abs ex rx (*Exn lc*) (*Result ra*) = *False*

rel-word-abs ex rx (*Result rc*) (*Result ra*) = (*ra* = *rx rc*)
rel-word-abs ex rx (*Exn lc*) (*Exn la*) = (*la* = *ex lc*)
by (*auto simp add*: *rel-word-abs-def*)

lemma *corresTA-refines*:
corresTA P rx ex f_a f_c ⟹ *P s* ⟹ *refines* f_c f_a *s s* (*rel-prod* (*rel-word-abs ex rx*) (=))
unfolding *corresTA-def*
apply (*clarsimp simp add*: *corresXF-refines-conv rel-word-abs-def rel-xval.simps*)
apply (*clarsimp simp add*: *refines-def-old rel-xval.simps reaches-succeeds*)
by (*smt* (*verit*) *le-boolE le-boolI' linorder-not-le xval-split*)

lemma *refines-corresTA*:
assumes *sim*: ⋀*s*. *P s* ⟹ *refines* f_c f_a *s s* (*rel-prod* (*rel-word-abs ex rx*) (=))
shows *corresTA P rx ex* f_a f_c
unfolding *corresTA-def*
apply (*clarsimp simp add*: *corresXF-refines-conv rel-word-abs-def rel-xval.simps*)
using *sim*
apply (*fastforce simp add*: *refines-def-old rel-xval.simps reaches-succeeds rel-word-abs-def split*: *xval-splits*)
done

lemma *corresTA-refines-conv*:
corresTA P rx ex f_a f_c ⟷ (∀ *s*. *P s* ⟶ *refines* f_c f_a *s s* (*rel-prod* (*rel-word-abs ex rx*) (=)))
using *corresTA-refines refines-corresTA* **by** *metis*

lemma *admissible-nondet-ord-corresTA* [*corres-admissible*]:
ccpo.admissible Inf (≥) (λ*A*. *corresTA P rx ex A C*)
unfolding *corresTA-def*
apply (*rule admissible-nondet-ord-corresXF*)
done

lemma *corresTA-top* [*corres-top*]: *corresTA P rx st* ⊤ *C*
by (*auto simp add*: *corresTA-def corresXF-def*)

lemma *corresTA-assume-and-weaken-pre*:
assumes *A-C*: ⋀*s*. *P s* ⟹ *corresTA Q rt ex A C*
assumes *P-Q*: ⋀*s*. *P s* ⟹ *Q s*
shows *corresTA P rt ex A C*
unfolding *corresTA-def*
apply (*rule corresXF-assume-pre*)
apply (*rule corresXF-guard-imp*)
apply (*rule A-C* [*unfolded corresTA-def*])
apply *assumption*
apply (*rule P-Q*)
apply *assumption*
done

lemma *corresTA-L2-gets*:
⟦ ⋀*s*. *abstract-val* (*Q s*) (*C s*) *rx* (*C′ s*) ⟧ ⟹
corresTA Q rx ex (*L2-gets* (λ*s*. *C s*) *n*) (*L2-gets* (λ*s*. *C′ s*) *n*)
unfolding *L2-defs*
apply (*clarsimp simp add*: *corresTA-refines-conv*)
apply (*rule refines-gets*)
apply (*simp add*: *abstract-val-def*)
done

lemma *corresTA-L2-modify*:
⟦ ⋀*s*. *abstract-val* (*P s*) (*m s*) *id* (*m′ s*) ⟧ ⟹
corresTA P rx ex (*L2-modify* (λ*s*. *m s*)) (*L2-modify* (λ*s*. *m′ s*))
unfolding *L2-defs*
apply (*clarsimp simp add*: *corresTA-refines-conv*)
apply (*rule refines-modify*)
apply (*simp add*: *abstract-val-def*)
done

lemma (**in** *heap-state*) *corresTA-IO-modify-heap-paddingE*[*word-abs*]:
abstract-val P p id p′ ⟹ (⋀*s*. *abstract-val* (*Q s*) (*v s*) *id* (*v′ s*)) ⟹
corresTA (λ*s*. *P* ∧ *Q s*) *rx ex* (*IO-modify-heap-paddingE p v*) (*IO-modify-heap-paddingE p′ v′*)
apply (*clarsimp simp add*: *corresTA-refines-conv IO-modify-heap-padding-def*)
apply (*simp add*: *refines-liftE-right-iff refines-liftE-left-iff abstract-val-def*)
apply (*rule refines-assert-result-and-state*)
apply *auto*
done

lemma *refines-throw*: *R* (*Exn x*, *s*) (*Exn y*, *t*) ⟹ *refines* (*throw x*) (*throw y*) *s t R*
by (*auto simp add*: *refines-def-old Exn-def*)

lemma *corresTA-L2-throw*:
⟦ *abstract-val Q C ex C′* ⟧ ⟹
corresTA (λ-. *Q*) *rx ex* (*L2-throw C n*) (*L2-throw C′ n*)
unfolding *L2-defs*
apply (*clarsimp simp add*: *corresTA-refines-conv*)
apply (*simp add*: *abstract-val-def*)
done

lemma *corresTA-L2-skip*:
corresTA (λ-. *True*) *rx ex L2-skip L2-skip*
unfolding *L2-defs*
by (*auto simp add*: *corresTA-refines-conv refines-gets*)

lemma *corresTA-L2-fail*:

corresTA (λ-. *True*) *rx ex L2-fail L2-fail*
unfolding *L2-defs*
by (*auto simp add*: *corresTA-refines-conv*)

lemma *corresTA-L2-seq'*:
fixes L' :: ('*e*, '*c1*, '*s*) *exn-monad*
fixes R' :: '*c1* ⇒ ('*e*, '*c2*, '*s*) *exn-monad*
fixes *L* :: ('*ea*, '*a1*, '*s*) *exn-monad*
fixes *R* :: '*a1* ⇒ ('*ea*, '*a2*, '*s*) *exn-monad*
shows
⟦ *corresTA P rx1 ex L L'*;
⋀*r*. *corresTA* (*Q* (*rx1 r*)) *rx2 ex* (*R* (*rx1 r*)) (R' *r*) ⟧ ⟹
corresTA P rx2 ex
(*L2-seq L* (λ*r*. *L2-seq* (*L2-guard* (λ*s*. *Q r s*)) (λ-. *R r*)))
(*L2-seq L'* (λ*r*. R' *r*))
apply *atomize*
apply (*clarsimp simp*: *L2-seq-def L2-guard-def corresTA-def*)
apply (*erule corresXF-join* [**where** P'=λ*x y s*. *rx1 y* = *x*])
subgoal
by (*auto simp add*: *corresXF-def reaches-bind reaches-guard succeeds-bind split*: *xval-splits*)
subgoal
by (*auto simp add*: *runs-to-partial-def-old split*: *xval-splits*)
subgoal by *simp*
done

lemma *corresTA-L2-seq*:
⟦ *introduce-typ-abs-fn rx1*;
PROP THIN (*Trueprop* (*corresTA P* (*rx1* :: '*a* ⇒ '*b*) *ex L L'*));
PROP THIN (⋀*r r'*. *abs-var r rx1 r'* ⟹ *corresTA* (*Q r*) *rx2 ex* (*R r*) (R' *r'*)) ⟧ ⟹
corresTA P rx2 ex (*L2-seq L* (λ*r*. *L2-seq* (*L2-guard* (*Q r*)) (λ-. *R r*))) (*L2-seq L'* R')
unfolding *THIN-def*
by (*rule corresTA-L2-seq'*, (*simp add*: *basic-abstract-defs*)+)

lemma *corresTA-L2-seq-unused-result*:
⟦ *introduce-typ-abs-fn rx1*;
PROP THIN (*Trueprop* (*corresTA P* (*rx1* :: '*a* ⇒ '*b*) *ex L L'*));
PROP THIN (*Trueprop* (*corresTA Q rx2 ex R R'*))⟧ ⟹
corresTA P rx2 ex (*L2-seq L* (λ*r*. *L2-seq* (*L2-guard Q*) (λ-. *R*))) (*L2-seq L'* (λ-. R'))
unfolding *THIN-def*
by (*rule corresTA-L2-seq'*, *simp*+)

lemma *corresTA-L2-seq-unit*:
fixes L' :: ('*e*, *unit*, '*s*) *exn-monad*
fixes R' :: *unit* ⇒ ('*e*, '*r*, '*s*) *exn-monad*
fixes *L* :: ('*ea*, *unit*, '*s*) *exn-monad*

fixes $R :: ('ea, 'ra, 's)$ *exn-monad*
shows
⟦*PROP THIN* (*Trueprop* (*corresTA P id ex L L'*));
PROP THIN (*Trueprop* (*corresTA Q rx ex R* (*R'* ()))) ⟧ ⟹
corresTA P rx ex
(*L2-seq L* (λ*r*. *L2-seq* (*L2-guard Q*) (λ-. *R*)))
(*L2-seq L' R'*)
unfolding *THIN-def*
by (*rule corresTA-L2-seq'*, *simp*+)

lemma *corresTA-L2-catch'*:
fixes $L' :: ('e1, 'c, 's)$ *exn-monad*
fixes $R' :: 'e1 \Rightarrow ('e2, 'c, 's)$ *exn-monad*
fixes $L :: ('e1a, 'ca, 's)$ *exn-monad*
fixes $R :: 'e1a \Rightarrow ('e2a, 'ca, 's)$ *exn-monad*
shows
⟦*corresTA P rx ex1 L L'*;
⋀*r*. *corresTA* (*Q* (*ex1 r*)) *rx ex2* (*R* (*ex1 r*)) (*R' r*) ⟧ ⟹
corresTA P rx ex2 (*L2-catch L* (λ*r*. *L2-seq* (*L2-guard* (λ*s*. *Q r s*)) (λ-. *R r*)))
(*L2-catch L'* (λ*r*. *R' r*))
apply *atomize*
apply (*clarsimp simp*: *L2-seq-def L2-catch-def L2-guard-def corresTA-def*)
apply (*erule corresXF-except* [**where** *P'*=λ*x y s*. *ex1 y* = *x*])
subgoal
by (*auto simp add*: *corresXF-def reaches-bind reaches-guard succeeds-bind split*: *xval-splits*)
subgoal
by (*auto simp add*: *runs-to-partial-def-old split*: *xval-splits*)
subgoal by *simp*
done

lemma *corresTA-L2-catch*:
⟦ *introduce-typ-abs-fn ex1*;
PROP THIN (*Trueprop* (*corresTA P rx ex1 L L'*));
PROP THIN (⋀*r r'*. *abs-var r ex1 r'* ⟹ *corresTA* (*Q r*) *rx ex2* (*R r*) (*R' r'*)) ⟧ ⟹
corresTA P rx ex2 (*L2-catch L* (λ*r*. *L2-seq* (*L2-guard* (λ*s*. *Q r s*)) (λ-. *R r*)))
(*L2-catch L'* (λ*r*. *R' r*))
unfolding *THIN-def*
by (*rule corresTA-L2-catch'*, (*simp add*: *basic-abstract-defs*)+)

term *corresTA P rx ex f g*

lemma *corresTA-yield*:
abstract-val True v' (*map-xval ex rx*) *v* ⟹ *corresTA P rx ex* (*yield v'*) (*yield v*)
apply (*auto simp add*: *corresTA-refines-conv rel-word-abs-def abstract-val-def rel-xval.simps map-xval-def*

intro!: *refines-yield split*: *xval-splits*)
done

lemma *map-sum-apply*: *map-sum ex rx* = (λ*v*. *case v of Inl l* ⇒ *Inl* (*ex l*) | *Inr r* ⇒ *Inr* (*rx r*))
by (*simp add*: *fun-eq-iff split-sum-all*)

lemma *refines-try-rel-prod*:
assumes *refines f g s t* (*rel-prod* (*rel-xval* (*rel-sum L R*) *R*) *S*)
shows *refines* (*try f*) (*try g*) *s t* (*rel-prod* (*rel-xval L R*) *S*)
using *assms*
apply (*clarsimp simp add*: *refines-def-old reaches-try unnest-exn-def rel-xval.simps split*: *xval-splits sum.splits*)
subgoal for *r s′ r′*
apply (*erule-tac x*=*r′* **in** *allE*)
apply (*erule-tac x*=*s′* **in** *allE*)

apply (*cases r*)
subgoal for *e*
apply (*clarsimp simp add*: *default-option-def Exn-def*, *safe*)
apply (*metis Exception-eq-Exception Exn-def Exn-neq-Result exception-or-result-cases not-None-eq sum-all-ex*(*2*))
by (*smt* (*verit*, *ccfv-threshold*) *Exn-eq-Exception*(*2*) *Exn-neq-Result rel-sum.cases rel-sum-simps*(*2*) *theLeft.simps the-Exn-Exn*(*2*))

subgoal for *v*
apply (*clarsimp simp add*: *default-option-def Exn-def*, *safe*)
apply (*metis Exception-eq-Exception Exn-def Exn-neq-Result exception-or-result-cases not-None-eq sum-all-ex*(*2*))
apply (*smt* (*verit*, *ccfv-threshold*) *Exn-def Result-neq-Exn rel-sum.cases rel-sum-simps*(*3*) *unnest-exn-eq-simps*(*3*))
by (*metis Exn-def Result-eq-Result Result-neq-Exn*)
done
done

lemma *rel-map-xval-sum-rel-sum-conv*:
rel-xval (λ*c a*. *a* = *map-sum ex rx c*) (λ*c a*. *a* = *rx c*) =
rel-xval (*rel-sum* (λ*c a*. *a* = *ex c*) (λ*c a*. *a* = *rx c*)) (λ*c a*. *a* = *rx c*)
apply (*rule ext*)+
apply (*auto simp add*: *rel-xval.simps rel-sum.simps*)
done

lemma *corresTA-L2-try′*:
assumes *corres-L-L′*: *corresTA P rx* (*map-sum ex rx*) *L L′*
shows *corresTA P rx ex* (*L2-try L*) (*L2-try L′*)
unfolding *L2-defs*

apply (*clarsimp simp add*: *corresTA-refines-conv rel-word-abs-def*)
apply (*rule refines-try-rel-prod*)
using *corres-L-L'*
apply (*auto simp add*: *corresTA-refines-conv rel-word-abs-def rel-map-xval-sum-rel-sum-conv*)
done

lemma *corresTA-L2-while*:
assumes *init-corres*: *abstract-val Q i rx i'*
and *cond-corres*: *PROP THIN* (⋀*r r' s. abs-var r rx r'*
⟹ *abstract-val* (*G r s*) (*C r s*) *id* (*C' r' s*))
and *body-corres*: *PROP THIN* (⋀*r r'. abs-var r rx r'*
⟹ *corresTA* (*P r*) *rx ex* (*B r*) (*B' r'*))
shows *corresTA* (λ-. *Q*) *rx ex*
(*L2-guarded-while* (λ*r s. G r s*) (λ*r s. C r s*) (λ*r. L2-seq* (*L2-guard* (λ*s. P r s*)) (λ-. *B r*)) *i x*)
(*L2-while* (λ*r s. C' r s*) *B' i' x*)
proof −
note *cond-corres* = *cond-corres* [*unfolded THIN-def*, *rule-format*]
note *body-corres* = *body-corres* [*unfolded THIN-def*, *rule-format*]
note *body-corres'* =
corresXF-guarded-while-body [*OF body-corres* [*unfolded corresTA-def*]]

have *init-corres'*:
Q ⟹ *i* = *rx i'*
using *init-corres*
by (*simp add*: *basic-abstract-defs*)

note *basic-abstract-defs* [*simp*]
show *?thesis*
thm *corresXF-assume-pre*
apply (*clarsimp simp*: *L2-defs corresTA-def gets-return*)
apply (*rule corresXF-assume-pre*)
thm *corresXF-guarded-while*
apply (*rule corresXF-guarded-while* [**where** *P*=λ*r s. G* (*rx r*) *s*])
subgoal for *s s' x y*
apply (*cut-tac r'*=*x* **in** *body-corres*, *simp*)
apply (*fastforce simp add*: *corresTA-refines-conv corresXF-refines-conv refines-def-old reaches-bind succeeds-bind*
rel-word-abs-def rel-xval.simps split: *xval-splits*)
done
subgoal
apply (*insert cond-corres*)[*1*]
apply (*clarsimp*)
done
subgoal
by (*auto simp add*: *runs-to-partial-def-old split*: *xval-splits*)
subgoal using *init-corres*
by (*clarsimp*)
subgoal using *init-corres'*

by *clarsimp*
done
qed

lemma *corresTA-L2-guard*:
⟦ ⋀*s*. *abstract-val* (*Q s*) (*G s*) *id* (*G′ s*) ⟧
⟹ *corresTA* (λ-. *True*) *rx ex* (*L2-guard* (λ*s*. *G s* ∧ *Q s*)) (*L2-guard* (λ*s*. *G′ s*))
unfolding *L2-defs*
apply (*auto simp add*: *corresTA-refines-conv abstract-val-def intro*!: *refines-guard*)
done

lemma *corresTA-L2-guard′*:
⟦⋀*s*. *abstract-val* (*Q s*) (*G s*) *id* (*G′ s*);
⋀*s*. *R s* ⟹ *G s* ∧ *Q s*⟧
⟹ *corresTA* (λ-. *True*) *rx ex* (*L2-guard* (λ*s*. *R s*)) (*L2-guard* (λ*s*. *G′ s*))
unfolding *L2-defs*
apply (*auto simp add*: *corresTA-refines-conv abstract-val-def intro*!: *refines-guard*)
done

lemma *corresTA-L2-guarded-simple*:
assumes *G-G′*: ⋀*s*. *abstract-val* (*Q s*) (*G s*) *id* (*G′ s*)
assumes *f-f′*: ⋀*s*. *G′ s* ⟹ *Q s* ⟹ *G s* ⟹ *corresTA P rx ex f f′*
shows *corresTA* (λ-. *True*) *rx ex* (*L2-guarded* (λ*s*. *G s* ∧ *Q s* ∧ *P s*) *f*) (*L2-guarded G′ f′*)
unfolding *L2-defs L2-guarded-def*
apply (*clarsimp simp add*: *corresTA-refines-conv*)
apply (*rule refines-bind-guard-right*)
using *G-G′ f-f′*
by (*auto simp add*: *corresTA-refines-conv refines-def-old succeeds-bind reaches-bind abstract-val-def*)

lemma *corresTA-L2-spec*:
(⋀*s t*. *abstract-val* (*Q s*) (*P s t*) *id* (*P′ s t*)) ⟹
corresTA Q rx ex (*L2-spec* {(*s*, *t*). *P s t*}) (*L2-spec* {(*s*, *t*). *P′ s t*})
unfolding *L2-defs*
by (*auto simp add*: *corresTA-refines-conv abstract-val-def refines-def-old reaches-bind succeeds-bind*)

lemma *corresTA-L2-assume*:
(⋀*s r t*. *abstract-val* (*Q s*) (*P s*) (λ*X*. (λ(*x*, *y*). (*rx x*, *y*)) ‘ *X*) (*P′ s*)) ⟹
corresTA Q rx ex (*L2-assume P*) (*L2-assume P′*)
unfolding *L2-defs*
apply (*auto simp add*: *corresTA-refines-conv abstract-val-def refines-def-old reaches-bind succeeds-bind*
rel-word-abs-def rel-xval.simps)
done

lemma *corresTA-L2-condition*:
⟦*PROP THIN* (*Trueprop* (*corresTA P rx ex L L'*));
PROP THIN (*Trueprop* (*corresTA Q rx ex R R'*));
⋀*s. abstract-val* (*T s*) (*C s*) *id* (*C' s*) ⟧
⟹ *corresTA T rx ex*
(*L2-condition* (λ*s. C s*)
(*L2-seq* (*L2-guard P*) (λ-. *L*))
(*L2-seq* (*L2-guard Q*) (λ-. *R*))
) (*L2-condition* (λ*s. C' s*) *L' R'*)
unfolding *THIN-def L2-defs*
by (*auto simp add*: *corresTA-refines-conv abstract-val-def intro*!: *refines-condition refines-bind-guard-right*)

lemma *L2-call-L2-defs*: *L2-call x emb ns* = *L2-catch x* (λ*e. L2-throw* (*emb e*) *ns*)
unfolding *L2-defs L2-call-def*
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*auto simp add*: *runs-to-def-old map-exn-def split*: *xval-splits*)
done

lemma *corresTA-L2-call*:
⟦*corresTA P rx ex' A B*;
⋀*r r'. abs-var r ex' r'* ⟹ *abstract-val Q* (*emb r*) *ex* (*emb' r'*)
⟧ ⟹
corresTA (λ*s. P s* ∧ *Q*) *rx ex* (*L2-call A emb ns*) (*L2-call B emb' ns*)
unfolding *L2-call-def*
by (*force simp add*: *corresTA-refines-conv abstract-val-def abs-var-def refines-def-old reaches-map-value map-exn-def*
rel-word-abs-def rel-xval.simps)

lemma *corresTA-L2-call'*:
⟦ *corresTA P f1 ex' A B*;
valid-typ-abs-fn Q1 Q1' f1 f1';
valid-typ-abs-fn Q2 Q2' f2 f2';
⋀*r r'. abs-var r ex' r'* ⟹ *abstract-val Q* (*emb r*) *ex* (*emb' r'*)
⟧ ⟹
corresTA (λ*s. P s* ∧ *Q*) *f2 ex*
(*L2-seq* (*L2-call A emb ns*) (*ETA-TUPLED* (λ*ret.* (*L2-seq* (*L2-guard* (λ-. *Q1' ret*)) (λ-. *L2-gets* (λ-. *f2* (*f1' ret*)) *ns*)))))
(*L2-call B emb' ns*)
unfolding *L2-call-def L2-defs*
apply (*clarsimp simp add*: *ETA-TUPLED-def corresTA-refines-conv abstract-val-def abs-var-def*
refines-def-old reaches-map-value map-exn-def reaches-bind succeeds-bind reaches-succeeds
rel-word-abs-def rel-xval.simps)
subgoal for *s s' r'*

```
    apply (erule-tac x=s in allE)
    apply clarsimp
    apply (cases r')
    subgoal
      apply (simp add: default-option-def Exn-def [symmetric])
     by (metis (mono-tags, lifting) Exn-def Result-neq-Exn case-exception-or-result-Exn
case-xval-simps(1))
    subgoal
      apply simp
      by (metis (mono-tags, lifting) Exn-neq-Result case-exception-or-result-Result
case-xval-simps(2) the-Result-simp)
    done
  done

lemma corresTA-L2-unknown:
  corresTA (λ-. True) rx ex (L2-unknown x) (L2-unknown x)
  unfolding L2-defs
  by (auto simp add: corresTA-refines-conv intro!: refines-select)


lemma corresTA-L2-call-exec-concrete:
  ⟦ corresTA P rx ex' A B ;
    ⋀r r'. abs-var r ex' r' ⟹ abstract-val Q (emb r) ex (emb' r')⟧ ⟹
        corresTA (λs. ∀ s'. s = st s' ⟶ P s' ∧ Q) rx ex
              (exec-concrete st (L2-call A emb ns))
              (exec-concrete st (L2-call B emb' ns))
  unfolding L2-defs L2-call-def
  apply (clarsimp simp add: corresTA-refines-conv abstract-val-def abs-var-def)
 apply (clarsimp simp add: refines-def-old succeeds-exec-concrete-iff reaches-exec-concrete

     reaches-map-value rel-word-abs-def rel-xval.simps map-exn-def split: xval-splits
)
  subgoal for r t t' r'
    apply (erule-tac x=t in allE)
    apply clarsimp
    apply (cases r')
    subgoal
      apply (clarsimp simp add: default-option-def Exn-def [symmetric])
      by (metis Exn-eq-Exn Exn-neq-Result)
    subgoal
      apply simp
      by (metis Result-eq-Result Result-neq-Exn)
    done
  done


lemma corresTA-L2-call-exec-abstract:
  ⟦ corresTA P rx ex' A B;
    ⋀r r'. abs-var r ex' r' ⟹ abstract-val Q (emb r) ex (emb' r') ⟧ ⟹
```

```
      corresTA (λs. P (st s) ∧ Q) rx ex
            (exec-abstract st (L2-call A emb ns))
            (exec-abstract st (L2-call B emb' ns))
  unfolding L2-defs L2-call-def
  apply (clarsimp simp add: corresTA-refines-conv abstract-val-def abs-var-def)
 apply (clarsimp simp add: refines-def-old succeeds-exec-abstract-iff reaches-exec-abstract

     reaches-map-value rel-word-abs-def rel-xval.simps map-exn-def split: xval-splits
)
  subgoal for s r s' r'
    apply (erule-tac x=st s in allE)
    apply clarsimp
    apply (cases r')
    subgoal
      apply (clarsimp simp add: default-option-def Exn-def [symmetric])
      by (metis Exn-eq-Exn Exn-neq-Result)
    subgoal
      apply simp
      by (metis Result-eq-Result Result-neq-Exn)
    done
  done



lemma corresTA-L2-call-exec-concrete':
  ⟦ corresTA P f1 ex' A B;
     valid-typ-abs-fn Q1 Q1' f1 f1';
     valid-typ-abs-fn Q2 Q2' f2 f2';
     ⋀r r'. abs-var r ex' r' ⟹ abstract-val Q (emb r) ex (emb' r')
   ⟧ ⟹
    corresTA (λs. ∀ s'. s = st s' ⟶ P s' ∧ Q) f2 ex
       (L2-seq (exec-concrete st (L2-call A emb ns)) (λret. (L2-seq (L2-guard (λ-.
Q1' ret)) (λ-. L2-gets (λ-. f2 (f1' ret)) []))))
       (exec-concrete st (L2-call B emb' ns))
  unfolding L2-defs L2-call-def
  apply (clarsimp simp add: corresTA-refines-conv abstract-val-def abs-var-def)
 apply (clarsimp simp add: refines-def-old succeeds-exec-concrete-iff reaches-exec-concrete

     reaches-bind succeeds-bind
     reaches-map-value rel-word-abs-def rel-xval.simps map-exn-def split: xval-splits
)
  subgoal for r t t' r'
    apply (erule-tac x=t in allE)
    apply clarsimp
    apply (cases r')
    subgoal
      apply (clarsimp simp add: default-option-def Exn-def [symmetric])
    by (smt (verit, ccfv-threshold) Exn-def Exn-eq-Exn Exn-neq-Result case-exception-or-result-Exn)
```

```
    subgoal for v
      apply clarsimp
      apply (erule-tac x=Result v in allE)
      apply (erule-tac x=t' in allE)
     by (smt (verit, ccfv-threshold) Exn-neq-Result Result-eq-Result case-exception-or-result-Result)
     done
  done

lemma corresTA-L2-call-exec-abstract':
  ⟦ corresTA P f1 ex' A B;
     valid-typ-abs-fn Q1 Q1' f1 f1';
     valid-typ-abs-fn Q2 Q2' f2 f2';
    ⋀r r'. abs-var r ex' r' ⟹ abstract-val Q (emb r) ex (emb' r')
   ⟧ ⟹
   corresTA (λs. P (st s) ∧ Q) f2 ex
       (L2-seq (exec-abstract st (L2-call A emb ns)) (λret. (L2-seq (L2-guard (λ-.
Q1' ret)) (λ-. L2-gets (λ-. f2 (f1' ret)) [])))))
       (exec-abstract st (L2-call B emb' ns))
  unfolding L2-defs L2-call-def
  apply (clarsimp simp add: corresTA-refines-conv abstract-val-def abs-var-def)
 apply (clarsimp simp add: refines-def-old succeeds-exec-abstract-iff reaches-exec-abstract

      reaches-bind succeeds-bind
     reaches-map-value rel-word-abs-def rel-xval.simps map-exn-def split: xval-splits
)
  subgoal for s r s' r'
    apply (erule-tac x=st s in allE)
    apply clarsimp
    apply (cases r')
    subgoal
      apply (clarsimp simp add: default-option-def Exn-def [symmetric])
     by (smt (verit, del-insts) Exn-eq-Exn Result-neq-Exn case-exception-or-result-Exn)
    subgoal
      apply clarsimp
     by (smt (verit, ccfv-threshold) Exn-neq-Result Result-eq-Result case-exception-or-result-Result)
    done
  done
```

Avoid higher-order unification issues by explicit application with ($):

- in concrete program position enforces 'obvious' instantiation
- in abstract program position enforces introduction of two separate variables for a and b instead of a higher-order flex-flex pair.

```
lemma abstract-val-fun-app:
  ⟦ abstract-val Q b id b'; abstract-val P a id a' ⟧ ⟹
        abstract-val (P ∧ Q) (f $ (a $ b)) f (a' $ b')
```

by (*simp add*: *basic-abstract-defs*)

lemma *corresTA-precond-to-guard*:
corresTA (λ*s*. *P s*) *rx ex A A'* ⟹ *corresTA* (λ-. *True*) *rx ex* (*L2-seq* (*L2-guard* (λ*s*. *P s*)) (λ-. *A*)) *A'*
unfolding *L2-defs*
by (*auto simp add*: *corresTA-refines-conv intro*: *refines-bind-guard-right*)

lemma *corresTA-precond-to-asm*:
⟦ ⋀*s*. *P s* ⟹ *corresTA* (λ-. *True*) *rx ex A A'* ⟧ ⟹ *corresTA P rx ex A A'*
by (*clarsimp simp*: *corresXF-def corresTA-def*)

lemma *L2-guard-true*: *L2-seq* (*L2-guard* (λ-. *True*)) *A* = *A* ()
unfolding *L2-defs*
apply (*rule spec-monad-ext*)
apply (*auto simp add*: *run-bind run-guard*)
done

lemma *corresTA-simp-trivial-guard*:
corresTA P rx ex (*L2-seq* (*L2-guard* (λ-. *True*)) *A*) *C* ≡ *corresTA P rx ex* (*A* ()) *C*
by (*simp add*: *L2-guard-true*)

lemma *corresTA-extract-preconds-of-call-init*:
⟦ *corresTA* (λ*s*. *P*) *rx ex A A'* ⟧ ⟹ *corresTA* (λ*s*. *P* ∧ *True*) *rx ex A A'*
by *simp*

lemma *corresTA-extract-preconds-of-call-step*:
⟦ *corresTA* (λ*s*. (*abs-var a f a'* ∧ *R*) ∧ *C*) *rx ex A A'*; *abstract-val Y a f a'* ⟧
⟹ *corresTA* (λ*s*. *R* ∧ (*Y* ∧ *C*)) *rx ex A A'*
by (*clarsimp simp*: *corresXF-def corresTA-def basic-abstract-defs*)

lemma *corresTA-extract-preconds-of-call-final*:
⟦ *corresTA* (λ*s*. (*abs-var a f a'*) ∧ *C*) *rx ex A A'*; *abstract-val Y a f a'* ⟧
⟹ *corresTA* (λ*s*. (*Y* ∧ *C*)) *rx ex A A'*
by (*clarsimp simp*: *corresXF-def corresTA-def basic-abstract-defs*)

lemma *corresTA-extract-preconds-of-call-final'*:
⟦ *corresTA* (λ*s*. *True* ∧ *C*) *rx ex A A'* ⟧
⟹ *corresTA* (λ*s*. *C*) *rx ex A A'*
by (*clarsimp simp*: *corresXF-def corresTA-def*)

lemma *corresTA-extract-preconds-of-call-init-prems*:
⟦ *corresTA* (λ*s*. *P* ∧ *True*) *rx ex A A'*⟧ ⟹ *corresTA* (λ*s*. *P*) *rx ex A A'*
by *simp*

lemma *corresTA-extract-preconds-of-call-step-prems*:
⟦⋀*Y*. *abstract-val Y a f a'* ⟹ *corresTA* (λ*s*. *R* ∧ (*Y* ∧ *C*)) *rx ex A A'* ⟧

$\Longrightarrow$ *corresTA* ($\lambda s.$ (*abs-var a f a'* $\wedge$ *R*) $\wedge$ *C*) *rx ex A A'*
by (*clarsimp simp*: *corresXF-def corresTA-def basic-abstract-defs*)

lemma *corresTA-extract-preconds-of-call-final-prems*:
⟦$\bigwedge$*Y*. *abstract-val Y a f a'* $\Longrightarrow$ *corresTA* ($\lambda s.$ (*Y* $\wedge$ *C*)) *rx ex A A'*⟧
$\Longrightarrow$ *corresTA* ($\lambda s.$ (*abs-var a f a'*) $\wedge$ *C*) *rx ex A A'*
by (*auto simp*: *corresXF-def corresTA-def basic-abstract-defs split*: *prod.splits sum.splits*)

lemma *corresTA-extract-preconds-of-call-final'-prems*:
⟦*corresTA* ($\lambda s.$ *C*) *rx ex A A'* ⟧
$\Longrightarrow$ *corresTA* ($\lambda s.$ *True* $\wedge$ *C*) *rx ex A A'*
by (*clarsimp simp*: *corresXF-def corresTA-def*)

lemma *corresTA-case-prod*:
⟦ *introduce-typ-abs-fn rx1*;
introduce-typ-abs-fn rx2;
abstract-val (*Q x*) *x* (*map-prod rx1 rx2*) *x'*;
$\bigwedge$*a b a' b'*. ⟦ *abs-var a rx1 a'*; *abs-var b rx2 b'* ⟧
$\Longrightarrow$ *corresTA* (*P a b*) *rx ex* (*M a b*) (*M' a' b'*) ⟧ $\Longrightarrow$
corresTA ($\lambda s.$ *case x of* (*a*, *b*) $\Rightarrow$ *P a b s* $\wedge$ *Q* (*a*, *b*)) *rx ex* (*case x of* (*a*, *b*) $\Rightarrow$ *M a b*) (*case x' of* (*a*, *b*) $\Rightarrow$ *M' a b*)
apply (*clarsimp simp add*: *corresTA-def*)
apply (*rule corresXF-assume-pre*)
apply (*clarsimp simp*: *split-def map-prod-def basic-abstract-defs*)
done

lemma *abstract-val-case-prod*:
⟦ *abstract-val True r* (*map-prod f g*) *r'*;
$\bigwedge$*a b a' b'*. ⟦ *abs-var a f a'*; *abs-var b g b'* ⟧
$\Longrightarrow$ *abstract-val* (*P a b*) (*M a b*) *h* (*M' a' b'*) ⟧
$\Longrightarrow$ *abstract-val* (*P* (*fst r*) (*snd r*))
(*case r of* (*a*, *b*) $\Rightarrow$ *M a b*) *h*
(*case r' of* (*a*, *b*) $\Rightarrow$ *M' a b*)
apply (*cases r*, *cases r'*)
apply (*clarsimp simp*: *map-prod-def basic-abstract-defs*)
done

lemma *abstract-val-case-prod-fun-app*:
⟦ *abstract-val True r* (*map-prod f g*) *r'*;
$\bigwedge$*a b a' b'*. ⟦ *abs-var a f a'*; *abs-var b g b'* ⟧
$\Longrightarrow$ *abstract-val* (*P a b*) (*M a b s*) *h* (*M' a' b' s*) ⟧
$\Longrightarrow$ *abstract-val* (*P* (*fst r*) (*snd r*))
((*case r of* (*a*, *b*) $\Rightarrow$ *M a b*) *s*) *h*
((*case r' of* (*a*, *b*) $\Rightarrow$ *M' a b*) *s*)
apply (*cases r*, *cases r'*)
apply (*clarsimp simp*: *map-prod-def basic-abstract-defs*)
done

lemma *abstract-val-of-nat*:
abstract-val ($r \leq$ *UWORD-MAX TYPE*('*a*::*len*)) *r unat* (*of-nat r* :: '*a word*)
by (*clarsimp simp*: *unat-of-nat-eq UWORD-MAX-def le-to-less-plus-one basic-abstract-defs*)

lemma *abstract-val-of-int*:
abstract-val (*WORD-MIN TYPE*('*a*::*len*) $\leq r \wedge r \leq$ *WORD-MAX TYPE*('*a*)) *r*
sint (*of-int r* :: '*a signed word*)
by (*clarsimp simp*: *sint-of-int-eq WORD-MIN-def WORD-MAX-def basic-abstract-defs*)

lemma *abstract-val-tuple*:
⟦ *abstract-val P a absL a'*;
abstract-val Q b absR b' ⟧ $\Longrightarrow$
abstract-val ($P \wedge Q$) (*a*, *b*) (*map-prod absL absR*) (*a'*, *b'*)
by (*clarsimp simp add*: *basic-abstract-defs*)

lemma *abstract-val-Inl*:
⟦ *abstract-val P a absL a'*⟧ $\Longrightarrow$
abstract-val P (*Inl a*) (*map-sum absL absR*) (*Inl a'*)
by (*clarsimp simp add*: *basic-abstract-defs*)

lemma *abstract-val-Inr*:
⟦ *abstract-val P b absR b'*⟧ $\Longrightarrow$
abstract-val P (*Inr b*) (*map-sum absL absR*) (*Inr b'*)
by (*clarsimp simp add*: *basic-abstract-defs*)

lemma *abstract-val-func*:
⟦ *abstract-val P a id a'*; *abstract-val Q b id b'* ⟧
$\Longrightarrow$ *abstract-val* ($P \wedge Q$) (*f a b*) *id* (*f a' b'*)
by (*clarsimp simp add*: *basic-abstract-defs*)

lemma *abstract-val-conj*:
⟦ *abstract-val P a id a'*;
abstract-val Q b id b' ⟧ $\Longrightarrow$
abstract-val ($P \wedge (a \longrightarrow Q)$) ($a \wedge b$) *id* ($a' \wedge b'$)
apply (*clarsimp simp add*: *basic-abstract-defs*)
apply *blast*
done

lemma *abstract-val-disj*:
⟦ *abstract-val P a id a'*;
abstract-val Q b id b' ⟧ $\Longrightarrow$
abstract-val ($P \wedge (\neg\ a \longrightarrow Q)$) ($a \vee b$) *id* ($a' \vee b'$)
apply (*clarsimp simp add*: *basic-abstract-defs*)
apply *blast*
done

lemma *abstract-val-unwrap*:

⟦ *introduce-typ-abs-fn f*; *abstract-val P a f b* ⟧
⟹ *abstract-val P a id* (*f b*)
by (*simp add*: *basic-abstract-defs*)

lemma *abstract-val-uint*:
⟦ *introduce-typ-abs-fn unat*; *abstract-val P x unat x′* ⟧
⟹ *abstract-val P* (*int x*) *id* (*uint x′*)
by (*clarsimp simp add*: *basic-abstract-defs*)

lemma *abstract-val-lambda*:
⟦ ⋀*v*. *abstract-val* (*P v*) (*a v*) *id* (*a′ v*) ⟧ ⟹
abstract-val (∀ *v*. *P v*) (λ*v*. *a v*) *id* (λ*v*. *a′ v*)
by (*auto simp add*: *basic-abstract-defs*)

lemma *corresTA-call-L1*:
abstract-val True arg-xf id arg-xf′ ⟹
corresTA (λ-. *True*) *id id*
(*L2-call-L1 arg-xf gs ret-xf l1body*)
(*L2-call-L1 arg-xf′ gs ret-xf l1body*)
apply (*unfold corresTA-def abstract-val-def id-def*)
apply (*subst* (*asm*) *simp-thms*)
apply (*erule subst*)
apply (*rule corresXF-id*[*simplified id-def*])
done

context *stack-heap-state*
begin
lemma *corresTA-with-fresh-stack-ptr*[*word-abs*]:
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* (⋀*p*. *corresTA Q rx ex* (f_a *p*) (f_c *p*))
assumes *init*: ⋀*s*. *abstract-val* (*P s*) ($init_a$ *s*) *id* ($init_c$ *s*)
shows *corresTA P rx ex*
(*with-fresh-stack-ptr n* $init_a$ (*L2-VARS* (λ*p*. (*L2-seq* (*L2-guard Q*) (λ-. f_a *p*))) *nm*))
(*with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*))
apply (*rule refines-corresTA*)
apply (*simp add*: *L2-seq-def L2-guard-def rel-word-abs-def*)
apply (*rule refines-rel-prod-with-fresh-stack-ptr*)
subgoal for *s*
using *init* [*of s*]
by (*simp add*: *abstract-val-def*)
subgoal for *s s′ p*
apply (*rule refines-bind-guard-right*)
apply (*simp only*: *rel-word-abs-def* [*symmetric*])
apply (*rule corresTA-refines*)
apply (*rule f*)
apply *simp*

```
    done
  done
end

context typ-heap-typing
begin

lemma corresTA-guard-with-fresh-stack-ptr[word-abs]:
  assumes f[simplified THIN-def, rule-format]: PROP THIN (⋀p. corresTA Q rx
ex (f_a p) (f_c p))
  assumes init: ⋀s. abstract-val (P s) (init_a s) id (init_c s)
  shows corresTA P rx ex
          (guard-with-fresh-stack-ptr n init_a (L2-VARS (λp. (L2-seq (L2-guard Q)
(λ-. f_a p))) nm))
          (guard-with-fresh-stack-ptr n init_c (L2-VARS f_c nm))
  apply (rule refines-corresTA)
  apply (simp add: L2-seq-def L2-guard-def rel-word-abs-def)
  apply (rule refines-rel-xval-guard-with-fresh-stack-ptr )
  subgoal for s
    using init [of s]
    by (simp add: abstract-val-def)
  subgoal for s s' p
    apply (rule refines-bind-guard-right)
    apply (simp only: rel-word-abs-def [symmetric])
    apply (rule corresTA-refines)
     apply (rule f)
    apply simp
    done
  done

lemma corresTA-assume-with-fresh-stack-ptr[word-abs]:
  assumes f[simplified THIN-def, rule-format]: PROP THIN (⋀p. corresTA Q rx
ex (f_a p) (f_c p))
  assumes init: ⋀s. abstract-val (P s) (init_a s) id (init_c s)
  shows corresTA P rx ex
          (assume-with-fresh-stack-ptr n init_a (L2-VARS (λp. (L2-seq (L2-guard Q)
(λ-. f_a p))) nm))
          (assume-with-fresh-stack-ptr n init_c (L2-VARS f_c nm))
  apply (rule refines-corresTA)
  apply (simp add: L2-seq-def L2-guard-def rel-word-abs-def)
  apply (rule refines-rel-xval-assume-with-fresh-stack-ptr)
  subgoal for s
    using init [of s]
    by (simp add: abstract-val-def)
  subgoal for s s' p
    apply (rule refines-bind-guard-right)
    apply (simp only: rel-word-abs-def [symmetric])
    apply (rule corresTA-refines)
     apply (rule f)
```

apply *simp*
done
done

lemma *corresTA-with-fresh-stack-ptr*[*word-abs*]:
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge$*p*. *corresTA Q rx ex* (f_a *p*) (f_c *p*))
assumes *init*: $\bigwedge$*s*. *abstract-val* (*P s*) ($init_a$ *s*) *id* ($init_c$ *s*)
shows *corresTA P rx ex*
(*with-fresh-stack-ptr n* $init_a$ (*L2-VARS* (λ*p*. (*L2-seq* (*L2-guard Q*) (λ-. f_a *p*))) *nm*))
(*with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*))
apply (*rule refines-corresTA*)
apply (*simp add*: *L2-seq-def L2-guard-def rel-word-abs-def*)
apply (*rule refines-rel-xval-with-fresh-stack-ptr*)
subgoal for *s*
using *init* [*of s*]
by (*simp add*: *abstract-val-def*)
subgoal for *s s′ p*
apply (*rule refines-bind-guard-right*)
apply (*simp only*: *rel-word-abs-def* [*symmetric*])
apply (*rule corresTA-refines*)
apply (*rule f*)
apply *simp*
done
done
end

lemma *abstract-val-call-L1-args*:
abstract-val P x id x′ ⟹ *abstract-val P y id y′* ⟹
abstract-val P (*x and y*) *id* (*x′ and y′*)
by (*simp add*: *basic-abstract-defs*)

lemma *abstract-val-call-L1-arg*:
abs-var x id x′ ⟹ *abstract-val P* (λ*s*. *f s* = *x*) *id* (λ*s*. *f s* = *x′*)
by (*simp add*: *basic-abstract-defs*)

lemma *abstract-val-abs-var* [*consumes 1*]:
⟦ *abs-var a f a′* ⟧ ⟹ *abstract-val True a f a′*
by (*clarsimp simp*: *fun-upd-def basic-abstract-defs split*: *if-splits*)

lemma *abstract-val-abs-var-concretise* [*consumes 1*]:
⟦ *abs-var a A a′*; *introduce-typ-abs-fn A*; *valid-typ-abs-fn PA PC A* (*C* :: *′a* ⇒ *′c*) ⟧
⟹ *abstract-val* (*PC a*) (*C a*) *id a′*
by (*clarsimp simp*: *fun-upd-def basic-abstract-defs split*: *if-splits*)

lemma *abstract-val-abs-var-give-up* [*consumes 1*]:
⟦ *abs-var a id a′* ⟧ ⟹ *abstract-val True* (*A a*) *A a′*
by (*clarsimp simp*: *fun-upd-def basic-abstract-defs split*: *if-splits*)

lemma *abstract-val-abs-var-sint-unat* [*consumes 1*]:
⟦ *abs-var a sint a′* ⟧ ⟹ *abstract-val* (*0* ≤ *a*) (*nat a*) *id* (*unat a′*)
apply (*simp add*: *basic-abstract-defs*)
by (*metis nat-uint-eq signed-0 sint-eq-uint word-msb-0 word-sle-eq word-sle-msb-le*)

lemma *abstract-val-abs-var-uint-unat* [*consumes 1*]:
⟦ *abs-var a uint a′* ⟧ ⟹ *abstract-val True* (*nat a*) *id* (*unat a′*)
by (*simp add*: *basic-abstract-defs*)

lemma *abs-var-id*: (*abs-var a id a′*) = (*a′* = *a*)
by (*auto simp add*: *abs-var-def abstract-val-def*)

lemma *abstract-val-id*: *abstract-val P a id a*
by (*simp add*: *abstract-val-def*)
lemmas *abstract-val-id-unit-ptr* = *abstract-val-id* [**where** *a*= *a*::*unit ptr* **and** *P* = *True*] **for** *a*

lemma *abstract-val-id-True*: *abstract-val True a id a*
by (*rule abstract-val-id*)

lemmas *abs-var-id-unit-ptr* = *abs-var-id* [**where** *a*= *a*::*unit ptr*] **for** *a*

lemma *len-of-word-comparisons* [*L2opt*]:
len-of TYPE(*64*) ≤ *len-of TYPE*(*64*)
len-of TYPE(*32*) ≤ *len-of TYPE*(*64*)
len-of TYPE(*16*) ≤ *len-of TYPE*(*64*)
len-of TYPE(*8*) ≤ *len-of TYPE*(*64*)
len-of TYPE(*32*) ≤ *len-of TYPE*(*32*)
len-of TYPE(*16*) ≤ *len-of TYPE*(*32*)
len-of TYPE(*8*) ≤ *len-of TYPE*(*32*)
len-of TYPE(*16*) ≤ *len-of TYPE*(*16*)
len-of TYPE(*8*) ≤ *len-of TYPE*(*16*)
len-of TYPE(*8*) ≤ *len-of TYPE*(*8*)

len-of TYPE(*32*) < *len-of TYPE*(*64*)
len-of TYPE(*16*) < *len-of TYPE*(*64*)
len-of TYPE(*8*) < *len-of TYPE*(*64*)
len-of TYPE(*16*) < *len-of TYPE*(*32*)
len-of TYPE(*8*) < *len-of TYPE*(*32*)
len-of TYPE(*8*) < *len-of TYPE*(*16*)

len-of TYPE('a::len signed) = len-of TYPE('a)
(len-of TYPE('a) = len-of TYPE('a)) = True
by *auto*

lemma *sbintrunc-eq*: *0 ≤ i ⟹ i < 2^n ⟹ sbintrunc n i = i*
apply (*induction n arbitrary*: *i*)
apply *auto*
done

lemma *uint-ucast*:
uint (ucast x :: ('a :: len) word) = uint x mod 2 ^ LENGTH('a)
unfolding *uint-nat unat-ucast* **by** (*simp add*: *zmod-int*)

lemma *uint-scast'*:
uint (SCAST('a::len → 'b::len) c) = sint c mod 2^LENGTH('b)
by (*metis Word.of-int-sint word-uint.eq-norm*)

lemma *uint-ucast'*:
uint (UCAST('a::len → 'b::len) c) = uint c mod 2^LENGTH('b)
by (*metis Word.of-int-uint word-uint.eq-norm*)

lemma *sint-ucast'*:
LENGTH('a) < LENGTH('b) ⟹ sint (UCAST('a::len → 'b::len) c) = uint c
using *sint-ucast-eq-uint'* **by** *blast*

lemma *scast-ucast-eq-ucast* [*simp*, *L2opt*]:
LENGTH('a::len) < LENGTH('b::len) ⟹ LENGTH('b) ≤ LENGTH('c::len)
⟹
SCAST('b → 'c) (UCAST('a → 'b) x) = UCAST ('a → 'c) x
unfolding *word-uint-eq-iff uint-scast' uint-ucast sint-ucast'* **..**

lemma *scast-ucast-simps* [*simp*, *L2opt*]:
⟦ *len-of TYPE('b) ≤ len-of TYPE('a)*; *len-of TYPE('c) ≤ len-of TYPE('b)* ⟧
⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a
⟦ *len-of TYPE('c) ≤ len-of TYPE('a)*; *len-of TYPE('c) ≤ len-of TYPE('b)* ⟧
⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a
⟦ *len-of TYPE('a) ≤ len-of TYPE('b)*; *len-of TYPE('c) ≤ len-of TYPE('b)* ⟧
⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a
⟦ *len-of TYPE('a) ≤ len-of TYPE('b)* ⟧ ⟹
(scast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = scast a
⟦ *len-of TYPE('b) ≤ len-of TYPE('a)*; *len-of TYPE('c) ≤ len-of TYPE('b)* ⟧
⟹
(ucast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = scast a
⟦ *len-of TYPE('c) ≤ len-of TYPE('a)*; *len-of TYPE('c) ≤ len-of TYPE('b)* ⟧
⟹

$(ucast\ (scast\ (a :: 'a{::}len\ word) :: 'b{::}len\ word) :: 'c{::}len\ word) = scast\ a$
$[\![\ len\text{-}of\ TYPE('a) \leq len\text{-}of\ TYPE('b);\ len\text{-}of\ TYPE('c) \leq len\text{-}of\ TYPE('b)\ ]\!]$
$\Longrightarrow$
$(ucast\ (scast\ (a :: 'a{::}len\ word) :: 'b{::}len\ word) :: 'c{::}len\ word) = scast\ a$
$[\![\ len\text{-}of\ TYPE('c) \leq len\text{-}of\ TYPE('b)\ ]\!] \Longrightarrow$
$(ucast\ (ucast\ (a :: 'a{::}len\ word) :: 'b{::}len\ word) :: 'c{::}len\ word) = ucast\ a$
$[\![\ len\text{-}of\ TYPE('a) \leq len\text{-}of\ TYPE('b)\ ]\!] \Longrightarrow$
$(ucast\ (ucast\ (a :: 'a{::}len\ word) :: 'b{::}len\ word) :: 'c{::}len\ word) = ucast\ a$
$[\![\ len\text{-}of\ TYPE('a) \leq len\text{-}of\ TYPE('b)\ ]\!] \Longrightarrow$
$(scast\ (scast\ (a :: 'a{::}len\ word) :: 'b{::}len\ word) :: 'c{::}len\ word) = scast\ a$
by (*auto simp*: *is-up is-down*
scast-ucast-1 scast-ucast-3 scast-ucast-4
ucast-scast-1 ucast-scast-3 ucast-scast-4
scast-scast-a scast-scast-b
ucast-ucast-a ucast-ucast-b)

declare *len-signed* [*L2opt*]

lemmas [*L2opt*] = *zero-sle-ucast-up*

lemma *zero-sle-ucast-WORD-MAX* [*L2opt*]:
$(0 <=s ((ucast\ (b{::}('a{::}len)\ word))) :: ('a{::}len)\ signed\ word))$
$= (uint\ b \leq WORD\text{-}MAX\ (TYPE('a)))$
by (*clarsimp simp*: *WORD-MAX-def zero-sle-ucast*)

lemmas [*L2opt*] =
is-up is-down unat-ucast-upcast sint-ucast-eq-uint

lemmas [*L2opt*] =
ucast-down-add scast-down-add
ucast-down-minus scast-down-minus
ucast-down-mult scast-down-mult

lemma *eq-trivI*: $x = y \Longrightarrow x = y$
by *simp*

lemmas [*word-abs*] =
corresTA-L2-gets
corresTA-L2-modify
corresTA-L2-throw
corresTA-L2-skip
corresTA-L2-fail
corresTA-L2-seq
corresTA-L2-seq-unit
corresTA-L2-catch

corresTA-L2-try′

corresTA-L2-while
corresTA-L2-guard
corresTA-L2-guarded-simple
corresTA-L2-spec
corresTA-L2-assume
corresTA-L2-condition
corresTA-L2-unknown
corresTA-case-prod
corresTA-L2-call-exec-concrete′
corresTA-L2-call-exec-concrete
corresTA-L2-call-exec-abstract′
corresTA-L2-call-exec-abstract
corresTA-L2-call′
corresTA-L2-call
corresTA-call-L1

lemmas [*word-abs*] =
abstract-val-tuple
abstract-val-Inl
abstract-val-Inr
abstract-val-conj
abstract-val-disj
abstract-val-case-prod
abstract-val-trivial
abstract-val-of-int
abstract-val-of-nat
abstract-val-call-L1-args

lemmas *abs-var-rules* =
abstract-val-call-L1-arg
abstract-val-abs-var-sint-unat
abstract-val-abs-var-uint-unat
abstract-val-abs-var-give-up
abstract-val-abs-var-concretise
abstract-val-abs-var

lemmas *word-abs-base* [*word-abs*] =
valid-typ-abs-fn-id [**where** *′a=′a::c-type*]
valid-typ-abs-fn-id [**where** *′a=bool*]
valid-typ-abs-fn-id [**where** *′a=′gx c-exntype*]
valid-typ-abs-fn-tuple-split
valid-typ-abs-fn-tuple
valid-typ-abs-fn-sum
valid-typ-abs-fn-unit
valid-typ-abs-fn-sint
valid-typ-abs-fn-unat

len-of-word-comparisons

lemmas *word-abs-sword* =
abstract-val-signed-ops
abstract-val-scast
abstract-val-scast-upcast
abstract-val-scast-downcast
abstract-val-unwrap [**where** *f*=*sint*]
introduce-typ-abs-fn [**where** *f*=*sint* :: (*sword64* ⇒ *int*)]
introduce-typ-abs-fn [**where** *f*=*sint* :: (*sword32* ⇒ *int*)]
introduce-typ-abs-fn [**where** *f*=*sint* :: (*sword16* ⇒ *int*)]
introduce-typ-abs-fn [**where** *f*=*sint* :: (*sword8* ⇒ *int*)]

lemmas *word-abs-word* =
abstract-val-unsigned-ops
abstract-val-uint
abstract-val-ucast
abstract-val-ucast-upcast
abstract-val-ucast-downcast
abstract-val-unwrap [**where** *f*=*unat*]
introduce-typ-abs-fn [**where** *f*=*unat* :: (*word64* ⇒ *nat*)]
introduce-typ-abs-fn [**where** *f*=*unat* :: (*word32* ⇒ *nat*)]
introduce-typ-abs-fn [**where** *f*=*unat* :: (*word16* ⇒ *nat*)]
introduce-typ-abs-fn [**where** *f*=*unat* :: (*word8* ⇒ *nat*)]

lemmas *word-abs-default* =
introduce-typ-abs-fn [**where** *f*=*id* :: ('*a*::*c-type* ⇒ '*a*)]
introduce-typ-abs-fn [**where** *f*=*id* :: (*bool* ⇒ *bool*)]
introduce-typ-abs-fn [**where** *f*=*id* :: ('*gx c-exntype* ⇒ '*gx c-exntype*)]
introduce-typ-abs-fn [**where** *f*=*id* :: (*unit* ⇒ *unit*)]
introduce-typ-abs-fn-tuple
introduce-typ-abs-fn-sum

thm *word-abs*

lemma *int-bounds-to-nat-boundsF*: (*n*::*int*) < *numeral B* ⟹ *0* ≤ *n* ⟹ *nat n* < *numeral B*
by (*simp add*: *nat-less-iff*)

lemma *int-bounds-one-to-nat*: (*n*::*int*) < *1* ⟹ *0* ≤ *n* ⟹ *nat n* = *0*
by (*simp add*: *nat-less-iff*)

lemma *id-map-prod-unfold*: *id* = *map-prod id id*
by (*simp add*: *prod.map-id0*)

lemma *id-tuple-unfold*: $id = (\lambda(x,\ y).\ (id\ x,\ id\ y))$
by *simp*

end

theory *WordPolish*
imports *WordAbstract*
begin

definition [*simplified*]: $LONG\text{-}MAX \equiv (2 :: int) \hat{}\ 63 - 1$
definition [*simplified*]: $LONG\text{-}MIN \equiv - ((2 :: int) \hat{}\ 63)$
definition [*simplified*]: $ULONG\text{-}MAX \equiv (2 :: nat) \hat{}\ 64 - 1$

definition [*simplified*]: $INT\text{-}MAX \equiv (2 :: int) \hat{}\ 31 - 1$
definition [*simplified*]: $INT\text{-}MIN \equiv - ((2 :: int) \hat{}\ 31)$
definition [*simplified*]: $UINT\text{-}MAX \equiv (2 :: nat) \hat{}\ 32 - 1$

definition [*simplified*]: $SHORT\text{-}MAX \equiv (2 :: int) \hat{}\ 15 - 1$
definition [*simplified*]: $SHORT\text{-}MIN \equiv - ((2 :: int) \hat{}\ 15)$
definition [*simplified*]: $USHORT\text{-}MAX \equiv (2 :: nat) \hat{}\ 16 - 1$

definition [*simplified*]: $CHAR\text{-}MAX \equiv (2 :: int) \hat{}\ 7 - 1$
definition [*simplified*]: $CHAR\text{-}MIN \equiv - ((2 :: int) \hat{}\ 7)$
definition [*simplified*]: $UCHAR\text{-}MAX \equiv (2 :: nat) \hat{}\ 8 - 1$

lemma *WORD-MAX-simps*:
$WORD\text{-}MAX\ TYPE(64) = LONG\text{-}MAX$
$WORD\text{-}MAX\ TYPE(32) = INT\text{-}MAX$
$WORD\text{-}MAX\ TYPE(16) = SHORT\text{-}MAX$
$WORD\text{-}MAX\ TYPE(8) = CHAR\text{-}MAX$
by (*auto simp*: *LONG-MAX-def INT-MAX-def SHORT-MAX-def CHAR-MAX-def WORD-MAX-def*)

lemma *WORD-MIN-simps*:
$WORD\text{-}MIN\ TYPE(64) = LONG\text{-}MIN$
$WORD\text{-}MIN\ TYPE(32) = INT\text{-}MIN$
$WORD\text{-}MIN\ TYPE(16) = SHORT\text{-}MIN$
$WORD\text{-}MIN\ TYPE(8) = CHAR\text{-}MIN$
by (*auto simp*: *LONG-MIN-def INT-MIN-def SHORT-MIN-def CHAR-MIN-def WORD-MIN-def*)

lemma *UWORD-MAX-simps*:
$UWORD\text{-}MAX\ TYPE(64) = ULONG\text{-}MAX$
$UWORD\text{-}MAX\ TYPE(32) = UINT\text{-}MAX$

UWORD-MAX TYPE(*16*) = *USHORT-MAX*
UWORD-MAX TYPE(*8*) = *UCHAR-MAX*
by (*auto simp*: *ULONG-MAX-def UINT-MAX-def USHORT-MAX-def UCHAR-MAX-def UWORD-MAX-def*)

lemma *MIN-MAX-lemmas-schema*:
sint (*s*::'*a*::*len signed word*) ≤ *WORD-MAX TYPE*('*a*)
WORD-MIN TYPE('*a*) ≤ *sint* (*s*::'*a*::*len signed word*)
unat (*u*::'*a*::*len word*) ≤ *UWORD-MAX TYPE*('*a*)
¬ (*sint* (*s*::'*a*::*len signed word*) > *WORD-MAX TYPE*('*a*))
¬ (*WORD-MIN TYPE*('*a*) > *sint* (*s*::'*a*::*len signed word*))
¬ (*unat* (*u*::'*a*::*len word*) > *UWORD-MAX TYPE*('*a*))
WORD-MIN TYPE('*a*) ≤ *WORD-MAX TYPE*('*a*)
0 ≤ *WORD-MAX TYPE*('*a*)
WORD-MIN TYPE('*a*) ≤ *0*
unfolding *WORD-MAX-def WORD-MIN-def UWORD-MAX-def*
using *unat-lt2p*[**where** *x*=*u*] *less-eq-Suc-le*
sint-range-size [**where** *w*=*s*, *simplified word-size*, *simplified*]
by *auto*

lemmas *MIN-MAX-lemmas-pre* =
MIN-MAX-lemmas-schema[**where** *'a=64*]
MIN-MAX-lemmas-schema[**where** *'a=32*]
MIN-MAX-lemmas-schema[**where** *'a=16*]
MIN-MAX-lemmas-schema[**where** *'a=8*]

lemmas *INT-MIN-MAX-lemmas* [*simp*] =
MIN-MAX-lemmas-pre[*unfolded WORD-MAX-simps WORD-MIN-simps UWORD-MAX-simps*]

end

Chapter 23

TS phase: Type Strengthening (find suitable target monad)

theory *Refines-Spec*
imports
Option-MonadND
L2ExceptionRewrite
begin

lemma *gets-the-ocondition*:
Spec-Monad.gets-the (*ocondition C T F*) =
Spec-Monad.condition C (*Spec-Monad.gets-the T*) (*Spec-Monad.gets-the F*)
by (*simp add*: *spec-monad-ext-iff ocondition-def run-condition*)

lemma *gets-the-oreturn*:
Spec-Monad.gets-the (*oreturn x*) = *Spec-Monad.return x*
by (*simp add*: *spec-monad-ext-iff*)

lemma *gets-the-obind*:
Spec-Monad.gets-the (*obind f g*) =
Spec-Monad.bind (*Spec-Monad.gets-the f*) (λ*x*. *Spec-Monad.gets-the* (*g x*))
by (*simp add*: *spec-monad-ext-iff run-bind obind-def split*: *option.splits*)

setup ‹
let open Mutual-CCPO-Rec in
add-ccpo spec-monad-gfp (*fn ctxt => fn* (*T as* ***Type*** ‹*spec-monad E A S*›) =>
let
in
synth-gfp ctxt T

```
    end)
  |> Context.theory-map
end
›

lemma rel-map-the-Result[simp]: rel-map the-Result (Result v) b ⟷ v = b
  by (auto simp add: rel-map-def)

lemma holds-partial-post-state-Inf:
  assumes X: ⋀x. x ∈ X ⟹ holds-partial-post-state P x
  shows holds-partial-post-state P (⨅ X)
  apply (clarsimp simp add: Inf-post-state-def vimage-def)
  subgoal premises prems for p x
    using X[of Success x] prems by auto
  done

lemma holds-post-state-Inf:
  assumes X: ⋀x. x ∈ X ⟹ holds-post-state P x
  shows X ≠ {} ⟹ holds-post-state P (⨅ X)
  apply (clarsimp simp add: Inf-post-state-def vimage-def, safe)
  subgoal for x using X[of x] by (cases x) auto
  subgoal using assms by force
  done

lemma admissible-runs-to[corres-admissible]:
  ccpo.admissible Inf (λx y. y ≤ x) (λx. x • s ⦃ Q ⦄)
  apply (simp add: ccpo.admissible-def chain-def)
  apply transfer
  apply (auto intro: holds-post-state-Inf)
  done

lemma spec-monad-Inf-run: (run (⨅ A) s) = ⨅ ((λf. (run f s)) ‘ A)
  by (metis INF-apply Inf-spec-monad.rep-eq)

lemma outcomes-Inf-run-succeeds-conv:
  outcomes (⨅f∈A. run f t) =  outcomes (⨅f ∈ {f. f ∈ A ∧ succeeds f t}. run f
t)
  apply (clarsimp simp add: Inf-post-state-def succeeds-def top-post-state-def)
  apply fastforce
  done

lemma succeeds-outcomes-Inf-Inter-conv:
  g ∈ A ⟹ succeeds g t ⟹ outcomes (⨅f ∈ {f ∈ A. succeeds f t}. run f t) =
        (⋂f∈{f ∈ A. succeeds f t}. outcomes (run f t))
  apply (clarsimp simp add: Inf-post-state-def succeeds-def top-post-state-def, intro
impI conjI set-eqI iffI)
  subgoal by (auto simp add: vimage-def image-def split: post-state.splits)
  subgoal by (smt (verit, ccfv-SIG) INF-top-conv(2) Inf-post-state-def mem-Collect-eq
top-post-state-def)
```

subgoal by (*smt* (*verit*) *Success-vimage-image-cancel image-cong mem-Collect-eq outcomes-succeeds-run-conv succeeds-def top-post-state-def*)
subgoal by (*smt* (*verit*, *best*) *Success-vimage-image-cancel image-cong mem-Collect-eq outcomes-succeeds-run-conv succeeds-def top-post-state-def*)
done

lemma *admissible-refines-spec-fun-of-rel*:
assumes *prj*: *fun-of-rel R prj*
shows *ccpo.admissible Inf* (≥)
(λ(*A*::('*e*::*default*, '*a*, '*s*) *spec-monad*) . *refines C A s t* (*rel-prod R* (=)))
proof (*rule ccpo.admissibleI*)
fix *A*::('*e*, '*a*, '*s*) *spec-monad set*
assume *A-prop*: ∀ *g*∈*A*. *refines C g s t* (*rel-prod R* (=))
assume *chain*: *Complete-Partial-Order.chain* (≥) *A*
assume *non-empty*: *A* ≠ {}
have *Inf-lower*: ⋀*g*. *g* ∈ *A* ⟹ ⨅ *A* ≤ *g*
by (*simp add*: *Inf-lower*)

hence *run-Inf-lower*: ⋀*g s*. *g* ∈ *A* ⟹ *run* (⨅ *A*) *s* ≤ *run g s*
by (*simp add*: *le-funD less-eq-spec-monad.rep-eq*)

have (⋀*s*. (⋀*g*. *g* ∈ *A* ⟹ *run g s* = *Failure*) ⟹ *run* (⨅ *A*) *s* = *Failure*)
apply (*simp add*: *spec-monad-Inf-run*)
by (*simp add*: *non-empty*)

hence *succeeds*: ⋀*s*. *succeeds* (⨅ *A*) *s* ⟹ (∃ *g* ∈ *A*. *succeeds g s*)
by (*auto simp add*: *succeeds-def top-post-state-def*)

show *refines C* (⨅ *A*) *s t* (*rel-prod R* (=))
apply (*clarsimp simp add*: *refines-def-old reaches-def*)
apply (*intro conjI allI impI*)
proof −
assume *succeeds* (⨅ *A*) *t* **then**
show *succeeds C s*
using *succeeds A-prop non-empty*
by (*auto simp add*: *refines-def-old*)
next
fix *r s*′
assume *succeeds-Inf*: *succeeds* (⨅ *A*) *t*
then obtain *g* **where** *g*: *g* ∈ *A* **and** *succeeds-g*: *succeeds g t*
using *succeeds* **by** *blast*

assume *res*: (*r*, *s*′) ∈ *outcomes* (*run C s*)
show ∃ *r*′. (*r*′, *s*′) ∈ *outcomes* (*run* (⨅ *A*) *t*) ∧ *R r r*′
proof −
from *A-prop res prj*
have *prj-prop*: ⋀*g*. *g* ∈ *A* ⟹ *succeeds g t* ⟹ (*prj r*, *s*′) ∈ *outcomes* (*run g*

```
t) ∧ R r (prj r)
        apply (clarsimp simp add: refines-def-old )
        using fun-of-rel-def
        by (smt (verit) reaches-def)
      hence (prj r, s') ∈ outcomes (run (⨅ A) t)
        apply (simp add: spec-monad-Inf-run)
        apply (subst outcomes-Inf-run-succeeds-conv)
        apply (subst succeeds-outcomes-Inf-Inter-conv [OF g succeeds-g])
        apply auto
        done
      moreover from prj-prop g succeeds-g have R r (prj r) by blast
      ultimately show ?thesis by blast
    qed
  qed
qed

lemma admissible-refines-spec-funp:
  assumes funp R
  shows ccpo.admissible Inf (≥)
          (λ(A::('e::default, 'a, 's) spec-monad) . refines C A s t (rel-prod R (=)))
proof −
  from assms obtain prj where fun-of-rel R prj
    by (auto simp add: funp-def)
  then show ?thesis
    by (rule admissible-refines-spec-fun-of-rel)
qed

lemma admissible-refines-spec-res[corres-admissible]:
  shows ccpo.admissible Inf (≥)
          (λ(A::('a, 's) res-monad) . refines C A s t (rel-prod (rel-liftE ) (=)))
  by (rule admissible-refines-spec-funp) (intro funp-intros)

lemma admissible-refines-spec-exit-eq[corres-admissible]:
  shows ccpo.admissible Inf (≥)
          (λ(A::('e, 'a, 's) exn-monad) . refines C A s t (rel-prod (rel-xval (=) (=))
(=)))
  by (rule admissible-refines-spec-funp) (intro funp-intros)

lemma admissible-refines-spec-exit-the-Nonlocal[corres-admissible]:
  shows ccpo.admissible Inf (≥)
              (λ(A::('e, 'a, 's) exn-monad) . refines C A s t (rel-prod ((rel-xval
(rel-Nonlocal) (=))) (=)))
  by (rule admissible-refines-spec-funp) (intro funp-intros)


lemma gen-admissible-refines-gets-the[corres-admissible]:
  shows ccpo.admissible option.lub-fun option.le-fun (λA. refines C (gets-the A) s
t (rel-prod (rel-liftE) (=)))
  apply (clarsimp simp add: refines-def-old relcompp.simps rel-liftE-def rel-map-def
```

split: *option.splits*)
by (*smt* (*verit*) *ccpo.admissibleI chain-fun flat-lub-in-chain fun-lub-def mem-Collect-eq option.discI*)

theorem *refines-option-top* [*corres-top*]: *refines f* (*gets-the Map.empty*) *s t R*
by (*auto simp add*: *refines-def-old*)

23.1 Synthesize Rules Setup

Canonical format for the currently supported monads: *refines C* (*lift-to-spec A*) *s t* (*rel-prod rel-res* (=)) where *lift-spec*, *rel-res* is

- pure (Pure function): *return*, *rel-liftE*
- gets (Reader monad): *gets*,*rel-liftE*
- option (Option monad): *gets-the*,*rel-liftE*
- nondet: *id* (ommitted),*rel-liftE*
- exit:
  - *id* (ommitted),*rel-xval rel-Nonlocal* (=)
  - *id* (ommitted),*rel-xval* (=) (=) for those function that were lifted by the IO phase.

synthesize-rules *pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit*

add-synthesize-pattern *pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit* **where**
refines-pure-synth = ‹*Trueprop* (*refines ?f* ((*return ?f'*)::(*?'a*, *?'s*) *res-monad*) - - *?R*)› (*f'*)

add-synthesize-pattern *reader* **and** *option* **and** *nondet* **and** *exit* **where**
refines-reader-synth = ‹*Trueprop* (*refines ?f* (*gets ?f'*) - - *?R*)› (*f'*)

add-synthesize-pattern *option* **where**
refines-option-synth = ‹*Trueprop* (*refines ?f* (*gets-the ?f'*) - - *?R*)› (*f'*)

add-synthesize-pattern *nondet* **and** *exit* **where**
refines-nondet-synth = ‹*Trueprop* (*refines ?f ?f'* - - *?R*)› (*f'*) **and**
rel-liftE-nondet-synth = ‹*Trueprop* (*rel-liftE ?x ?x'*)› (*x'*) **and**
rel-liftE'-nondet-synth = ‹*Trueprop* (*rel-liftE' ?x ?x'*)› (*x'*) **and**
rel-sum-nondet-synth = ‹*Trueprop* (*rel-sum* - - *?x ?x'*)› (*x'*) **and**
rel-sum-nondet-synth = ‹*Trueprop* (*rel-xval* - - *?x ?x'*)› (*x'*) **and**
rel-compp-nondet-synth = ‹*Trueprop* ((- *OO* -) *?x ?x'*)› (*x'*) **and**
rel-map-synth = ‹*Trueprop* (*rel-map ?f ?x ?x'*)› (*x'*) **and**
rel-eq-nondet-synth = ‹*Trueprop* (*?x* = *?x'*)› (*x'*)

add-synthesize-pattern *exit* **where**
rel-Nondet-exit-synth = ‹*Trueprop* (*rel-Nonlocal ?x ?x'*)› (*x'*)

Recursive functions are defined using **fixed-point**. The option, nondet and exit monad are setup to handle these definitions. Hence the minimal monad for recursive functions is the option monad.

23.2 Pure Monad

lemma *refines-L2-call-embed-pure*:
assumes *f*: *refines f* (*return f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*return* (*L2-VARS f' ns*)::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-VARS-def L2-call-def*
using *f*
apply (*clarsimp simp add*: *refines-def-old reaches-map-value map-exn-def rel-liftE-def split*: *xval-splits*)
apply (*metis Exn-def default-option-def exception-or-result-cases not-Some-eq*)+
done

lemma *refines-L2-gets-pure*:
refines (*L2-gets* (λ-. *v*) *ns*) (*return* (*L2-VARS v ns*)::('*a*,'*s*) *res-monad*) *s s* (*rel-prod* (*rel-liftE*) (=))
unfolding *L2-VARS-def L2-defs gets-return*
by (*auto intro*: *refines-yield*)

lemma *refines-L2-seq-pure*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*return* (*g' v*)::('*a*,'*s*) *res-monad*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (*Trueprop* (*refines f* ((*return f'*)::('*b*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=)))))
shows *refines* (*L2-seq f g*) (*return* (*let v* = *f' in* (*g' v*))::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
using *g f* **unfolding** *L2-defs*
apply (*clarsimp simp add*: *refines-def-old reaches-bind succeeds-bind rel-liftE-def default-option-def split*: *exception-or-result-splits*)
apply (*metis default-option-def exception-or-result-cases option.exhaust-sel*)+
done

lemma *refines-L2-condition-pure*:
assumes *g*: *refines g* (*return g'*::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
assumes *f*: *refines f* (*return f'*::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-condition* (λ-. *c*) *f g*) (*return* (*if c then f' else g'*)::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
using *g f* **unfolding** *L2-defs*
by (*auto intro*: *refines-condition*)

lemma *refines-L2-try-L2-throw-pure*:

shows *refines* (*L2-try* (*L2-throw* (*Inr r*) *ns*)) (*return* (*L2-VARS r ns*)::('*a*,'*s*)*res-monad*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-VARS-def*
by (*auto simp add*: *refines-def-old reaches-try rel-liftE-def unnest-exn-def*)

lemma *refines-L2-try-L2-seq-pure*:
assumes *g*: ⋀*v t*. *refines* (*L2-try* (*g v*)) (*return* (*g*′ *v*)::('*a*,'*s*) *res-monad*) *t t* (*rel-prod rel-liftE* (=))
assumes *f*: *refines f* (*return f*′::('*b*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try* (*L2-seq f g*)) (*return* (*let v* = *f*′ *in* (*g*′ *v*))::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs try-def refines-map-value-left-iff return-let-bind*
apply (*rule refines-bind* [*OF f*])
subgoal by *auto*
subgoal by (*auto simp add*: *default-option-def Exn-def*[*symmetric*] *rel-liftE-def*)
subgoal by *auto*
subgoal using *g* [*simplified L2-defs try-def refines-map-value-left-iff*] **by** *auto*
done

lemma *refines-L2-catch-pure*:
assumes *f*: *refines f* (*return f*′::('*b*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-catch f g*) (*return f*′::('*b*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs catch-def refines-map-value-left-iff return-let-bind*
apply (*rule refines-bind-handle-left*′)
using *f* [*simplified refines-return-right-iff*]
by (*simp add*: *rel-liftE-def runs-to-weaken*)

lemma *refines-L2-try-L2-condition-pure*:
assumes *f*: *refines* (*L2-try f*) (*return f*′::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
assumes *g*: *refines* (*L2-try g*) (*return g*′::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try* (*L2-condition* (λ-. *c*) *f g*)) (*return* (*if c then f*′ *else g*′)::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
using *f g* **unfolding** *L2-defs*
apply (*auto simp add*: *refines-def-old reaches-try reaches-bind succeeds-bind rel-liftE-def unnest-exn-def*
default-option-def Exn-def [*symmetric*]
split: *xval-split exception-or-result-splits sum.splits*)
done

lemma *refines-L2-try-pure*:
assumes *f*: *refines f* (*return f*′::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) (*return f*′::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-guarded-def try-def* **using** *f*
apply (*clarsimp simp add*: *refines-def-old reaches-try reaches-map-value rel-liftE-def*

unnest-exn-def
default-option-def Exn-def [*symmetric*]
split: *xval-split exception-or-result-splits sum.splits*)
by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel sum-all-ex*(*2*))

print-synthesize-rules *pure*

lemmas *refines-monad-pure* =
refines-L2-call-embed-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]
refines-L2-gets-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]
refines-L2-seq-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510 split*: *g* **and** *g'*]
refines-L2-condition-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]
refines-L2-catch-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]
refines-L2-try-L2-throw-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*520*]
refines-L2-try-L2-seq-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*520 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*520*]
refines-L2-try-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]

print-synthesize-rules *pure*

23.3 Reader Monad (Gets)

lemma *refines-L2-call-embed-reader*:
assumes *f*: *refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*gets* (*L2-VARS f' ns*)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-call-def liftE-def L2-VARS-def* **using** *f*
apply (*clarsimp simp add*: *refines-def-old reaches-try reaches-map-value rel-liftE-def map-exn-def*
default-option-def Exn-def [*symmetric*]
split: *xval-split exception-or-result-splits sum.splits*)
by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)

lemma *refines-L2-gets-reader*:
refines (*L2-gets f ns*) (*gets* (*L2-VARS f ns*)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-VARS-def liftE-def*
by (*auto intro*: *refines-gets*)

lemma *refines-lift-pure-reader*:
assumes *f*: *refines f* (*return f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines f* (*gets* (λ-. *f'*)) *s s* (*rel-prod rel-liftE* (=))
using *f* **unfolding** *liftE-def*
by (*simp add*: *gets-return*)

lemma *refines-L2-seq-reader*:
assumes *g*: ⋀*v t*. *refines* (*g v*) (*gets* (*g' v*)) *t t* (*rel-prod rel-liftE* (=))
assumes *f*: *refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-seq f g*) (*gets* (λ*s*. *let v* = *f' s in* (*g' v s*))) *s s* (*rel-prod rel-liftE* (=))
using *g f* **unfolding** *L2-defs liftE-def*
apply (*clarsimp simp add*: *refines-def-old reaches-bind succeeds-bind rel-liftE-def*

default-option-def Exn-def [*symmetric*]
split: *xval-split exception-or-result-splits sum.splits*)
apply (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)+
done

lemma *refines-L2-condition-reader*:
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines g* (*gets g'*) *s s* (*rel-prod rel-liftE* (=))))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-condition c f g*) (*gets* (λ*s*. *if c s then f' s else g' s*)) *s s* (*rel-prod rel-liftE* (=))
using *g f* **unfolding** *L2-defs liftE-def*
apply (*auto simp add*: *refines-def-old reaches-bind succeeds-bind rel-liftE-def*
default-option-def Exn-def [*symmetric*]
split: *xval-split exception-or-result-splits sum.splits*)
done

lemma *refines-L2-try-L2-throw-reader*:
shows *refines* (*L2-try* (*L2-throw* (*Inr r*) *ns*)) (*gets* (*L2-VARS* (λ-. *r*) *ns*)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-VARS-def liftE-def*
apply (*auto simp add*: *refines-def-old reaches-try reaches-map-value rel-liftE-def map-exn-def*
default-option-def Exn-def [*symmetric*]
split: *xval-split exception-or-result-splits sum.splits*)
done

lemma *refines-L2-try-L2-seq-reader*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*L2-try* (*g v*)) (*gets* (*g' v*)) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*gets* (λ*s*. *let v* = *f' s in* (*g' v s*))) *s s* (*rel-prod rel-liftE* (=))

unfolding *L2-defs try-def refines-map-value-left-iff gets-let-bind*
apply (*rule refines-bind* [*OF f*])
subgoal by *auto*
subgoal by (*auto simp add*: *default-option-def Exn-def* [*symmetric*] *rel-liftE-def*)
subgoal by *auto*
subgoal using *g* [*unfolded L2-defs try-def refines-map-value-left-iff*] **by** *auto*
done

lemma *refines-L2-catch-reader*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-catch f g*) (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs catch-def refines-map-value-left-iff return-let-bind*
apply (*rule refines-bind-handle-left'*)
using *f*
by (*simp add*: *refines-iff-runs-to rel-liftE-def runs-to-gets-iff*)

lemma *refines-L2-try-L2-condition-reader*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try f*) (*gets f'*) *s s* (*rel-prod rel-liftE* (=))))
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try g*) (*gets g'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-condition* (*c*) *f g*)) (*gets* (λ*s*. *if c s then f' s else g' s*)) *s s* (*rel-prod rel-liftE* (=))
using *f g* **unfolding** *L2-defs liftE-def*
apply (*auto simp add*: *refines-def-old reaches-try*)
done

lemma *refines-L2-try-reader*:
assumes *f*: *refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-guarded-def try-def liftE-def* **using** *f*
by (*auto simp add*: *refines-def-old reaches-map-value rel-liftE-def unnest-exn-def split*: *xval-splits*)
(*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)+

lemmas *refines-monad-reader* =
refines-L2-call-embed-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:410*]
refines-L2-gets-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:410*]
refines-lift-pure-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:440*]
refines-L2-seq-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:410 split*: *g* **and** *g'*]
refines-L2-condition-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:410*]
refines-L2-catch-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:410*]

refines-L2-try-L2-throw-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:420*]
refines-L2-try-L2-seq-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:420 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:420*]
refines-L2-try-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority:410*]

23.4 Option (Reader) Monad

lemma *refines-lift-reader-option*:
assumes *f*: *refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines f* (*gets-the* (*ogets f'*)) *s s* (*rel-prod rel-liftE* (=))
using *f*
apply (*auto simp add*: *refines-def-old rel-liftE-def ogets-def*)
done

lemma *refines-L2-call-embed-option*:
assumes *f*: *refines f* (*gets-the f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*gets-the* (*L2-VARS f' ns*)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-VARS-def L2-call-def L2-defs* **using** *f*
apply (*clarsimp simp add*: *refines-def-old reaches-map-value map-exn-def rel-liftE-def split*: *xval-splits*)
apply (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)+
done

lemma *refines-L2-gets-option*:
refines (*L2-gets v ns*) (*gets-the* (*L2-VARS* (*ogets v*) *ns*)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-VARS-def L2-defs liftE-def*
apply (*auto simp add*: *refines-def-old rel-liftE-def ogets-def*)
done

lemma *refines-L2-seq-option*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*gets-the* (*g' v*)) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets-the f'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) (*gets-the* (*f'* |>> *g'*)) *s s* (*rel-prod rel-liftE* (=))
using *g f* **unfolding** *L2-defs obind-def* [*abs-def*]
by (*fastforce simp add*: *refines-def-old reaches-bind succeeds-bind rel-liftE-def split*: *xval-splits option.splits*)

lemma *refines-L2-condition-option*:
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines g* (*gets-the g'*) *s s* (*rel-prod rel-liftE* (=))))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets-the f'*)

s s (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-condition c f g*) (*gets-the* (*ocondition c f′ g′*)) *s s* (*rel-prod rel-liftE* (=))
using *g f* **unfolding** *L2-defs ocondition-def*
by (*auto simp add*: *refines-def-old rel-liftE-def split*: *xval-splits option.splits*)

lemma *refines-L2-try-L2-throw-option*:
shows *refines* (*L2-try* (*L2-throw* (*Inr r*) *ns*)) (*gets-the* (*L2-VARS* (*oreturn r*) *ns*)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-VARS-def oreturn-def K-def*
by (*auto simp add*: *refines-def-old reaches-try unnest-exn-def rel-liftE-def split*: *xval-splits*)

lemma *refines-L2-try-L2-seq-option*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*L2-try* (*g v*)) (*gets-the* (*g′ v*)) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*gets-the* (*f′* |>> *g′*)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs try-def refines-map-value-left-iff*
apply (*simp add*: *gets-the-obind*)
apply (*rule refines-bind* [*OF f*])
subgoal by *auto*
subgoal by (*auto simp add*: *default-option-def Exn-def* [*symmetric*] *rel-liftE-def*)
subgoal by *auto*
subgoal using *g* [*simplified L2-defs try-def refines-map-value-left-iff*] **by** *auto*
done

lemma *refines-L2-try-L2-condition-option*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try f*) (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))))
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try g*) (*gets-the g′*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-condition c f g*)) (*gets-the* (*ocondition c f′ g′*)) *s s* (*rel-prod rel-liftE* (=))
using *f g* **unfolding** *L2-defs ocondition-def*
apply (*auto simp add*: *refines-def-old reaches-try*)
done

lemma *refines-L2-try-option*:
assumes *f*: *refines f* (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs* **using** *f*
apply (*clarsimp simp add*: *refines-def-old reaches-try unnest-exn-def rel-liftE-def split*: *xval-splits*)
apply (*metis Exn-def default-option-def exception-or-result-cases not-None-eq*)+
done

lemma *le-whileLoop-succeeds-terminatesI*:
assumes $\bigwedge s.\ run\ (whileLoop\ C\ B\ r)\ s \neq \top \Longrightarrow run\ f\ s \leq run\ (whileLoop\ C\ B\ r)\ s$
shows $f \leq whileLoop\ C\ B\ r$
proof (*rule le-spec-monad-runI*)
fix *s*
show $run\ f\ s \leq run\ (whileLoop\ C\ B\ r)\ s$
proof (*cases run* $(whileLoop\ C\ B\ r)\ s = \top$)
case *False* **then show** *?thesis* **by** (*rule assms*)
qed *simp*
qed

lemma *gets-the-whileLoop*:
fixes $C :: 'a \Rightarrow 's \Rightarrow bool$
shows $whileLoop\ C\ (\lambda a.\ gets\text{-}the\ (B\ a))\ r =$
$((gets\text{-}the\ (owhile\ C\ B\ r))::('e::default, 'a, 's)\ spec\text{-}monad)$
proof (*rule antisym*)
show $(whileLoop\ C\ (\lambda a.\ Spec\text{-}Monad.gets\text{-}the\ (B\ a))\ r::('e, 'a, 's)\ spec\text{-}monad) \leq$
Spec-Monad.gets-the $(owhile\ C\ B\ r)$

proof −
{
fix *s v*
assume $(Some\ r, Some\ v) \in option\text{-}while'\ (\lambda a.\ C\ a\ s)\ (\lambda a.\ B\ a\ s)$
then have $run\ ((whileLoop\ C\ (\lambda a.\ Spec\text{-}Monad.gets\text{-}the\ (B\ a))\ r)::('e, 'a, 's)\ spec\text{-}monad)\ s \leq Success\ \{(Result\ v, s)\}$
proof (*induct Some r Some v arbitrary*: *r*)
assume $\neg\ C\ v\ s$
then
show $run\ (whileLoop\ C\ (\lambda a.\ Spec\text{-}Monad.gets\text{-}the\ (B\ a))\ v)\ s \leq Success\ \{(Result\ v, s)\}$
by (*subst whileLoop-unroll*) *simp*
next
fix $r\ r'$
assume $C\ r\ s$
and $B\ r\ s = Some\ r'$
and $(Some\ r', Some\ v) \in option\text{-}while'\ (\lambda a.\ C\ a\ s)\ (\lambda a.\ B\ a\ s)$
and $run\ ((whileLoop\ C\ (\lambda a.\ Spec\text{-}Monad.gets\text{-}the\ (B\ a))\ r')::('e, 'a, 's)\ spec\text{-}monad)\ s \leq Success\ \{(Result\ v, s)\}$
thus $run\ ((whileLoop\ C\ (\lambda a.\ Spec\text{-}Monad.gets\text{-}the\ (B\ a))\ r)::('e, 'a, 's)\ spec\text{-}monad)\ s \leq Success\ \{(Result\ v, s)\}$
by (*subst whileLoop-unroll*) (*simp add*: *run-bind*)
qed
} **note** $* = this$
show *?thesis*
apply (*rule le-spec-monad-runI*)
apply *simp*

```
      apply (clarsimp simp add: owhile-def option-while-def option-while'-THE
pure-post-state-def
          split: option.splits)
     by (rule *)
  qed
next
  show Spec-Monad.gets-the (owhile C B r) ≤ ((whileLoop C (λa. Spec-Monad.gets-the
(B a)) r)::('e, 'a, 's) spec-monad)
  proof (rule le-whileLoop-succeeds-terminatesI)
    fix s
    assume run (whileLoop C (λa. Spec-Monad.gets-the (B a)) r) s ≠ ⊤
    then show run (Spec-Monad.gets-the (owhile C B r)) s
              ≤ run ((whileLoop C (λa. Spec-Monad.gets-the (B a)) r)::('e, 'a, 's)
spec-monad) s
    proof (induction rule: whileLoop-ne-top-induct)
      case (step a s) then show ?case
        apply (subst owhile-unroll)
        apply (subst whileLoop-unroll)
         apply (auto simp add: gets-the-obind gets-the-ocondition gets-the-oreturn
run-condition
            run-bind runs-to-holds-def
            split: option.split)
        done
    qed
  qed
qed

lemma refines-L2-while-option:
  assumes f [unfolded THIN-def, rule-format]: PROP THIN (⋀v t. refines (b v)
(gets-the (b' v)) t t (rel-prod rel-liftE (=)))
  shows refines (L2-while c b i ns) (gets-the (L2-VARS (owhile c b' i) ns)) s s
(rel-prod rel-liftE (=))
  unfolding L2-while-def L2-VARS-def gets-the-whileLoop[symmetric]
  apply (rule refines-whileLoop)
  using f
  apply (simp-all add: rel-liftE-def)
  done

lemma refines-L2-fail-option:
  shows refines L2-fail (gets-the ofail) s s (rel-prod rel-liftE (=))
  unfolding L2-defs
  by (auto simp add: refines-def-old)


lemma refines-L2-guard-option:
  shows refines (L2-guard g) (gets-the (oguard g)) s s (rel-prod rel-liftE (=))
  unfolding L2-defs
  by (auto simp add: refines-def-old oguard-def)
```

lemma *refines-L2-guarded*:
assumes *f*: *g s* ⟹ *refines f* (*gets-the f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-guarded g f*) (*gets-the* (*oguard g* |>> (λ-. *f'*))) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-guarded-def* **using** *f*
by (*auto simp add*: *refines-def-old reaches-bind succeeds-bind obind-def oguard-def split*: *option.splits*)

lemmas *refines-monad-option* =
refines-lift-reader-option [*synthesize-rule option priority*:*350*]
refines-L2-call-embed-option [*synthesize-rule option priority*:*310*]
refines-L2-gets-option [*synthesize-rule option priority*:*310*]

refines-L2-seq-option [*synthesize-rule option priority*:*310 split*: *g* **and** *g'*]
refines-L2-condition-option [*synthesize-rule option priority*:*310*]
refines-L2-try-L2-throw-option [*synthesize-rule option priority*:*320*]
refines-L2-try-L2-seq-option [*synthesize-rule option priority*:*320 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-option [*synthesize-rule option priority*:*320*]

refines-L2-try-option [*synthesize-rule option priority*:*310*]
refines-L2-while-option [*synthesize-rule option priority*:*310 split*: *b* **and** *b'*]
refines-L2-fail-option [*synthesize-rule option priority*:*310*]
refines-L2-guard-option [*synthesize-rule option priority*:*310*]
refines-L2-guarded [*synthesize-rule option priority*:*310*]

23.5 Nondet Monad

Note that *L2-catch* is already replaced by *L2-try* during exception rewriting in phase L2.

lemma (**in** *heap-state*) *refines-IO-modify-heap-paddingE-nondet*:
refines (*IO-modify-heap-paddingE p v*) (*IO-modify-heap-padding p v*) *s s* (*rel-prod rel-liftE* (=))
apply (*clarsimp simp add*: *IO-modify-heap-padding-def*)
apply (*simp add*: *refines-liftE-left-iff*)
apply (*rule refines-assert-result-and-state*)
apply *auto*
done

lemma *refines-L2-call-embed-nondet*:
assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*L2-VARS f' ns*) *s s* (*rel-prod rel-liftE* (=))
using *f* **unfolding** *L2-call-def L2-VARS-def*
apply (*clarsimp simp add*: *refines-def-old reaches-map-value map-exn-def rel-liftE-def split*: *xval-splits*)
by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)

lemma *refines-L2-call-embed-exn*:

assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*liftE* (*L2-VARS f' ns*)) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
using *f* **unfolding** *L2-call-def L2-VARS-def*
apply (*clarsimp simp add*: *refines-def-old reaches-map-value map-exn-def reaches-liftE rel-liftE-def rel-xval.simps split*: *xval-splits*)
by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)

lemma *refines-liftE-exn*:
assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines f* (*liftE f'*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
using *f*
by (*fastforce simp add*: *refines-def-old reaches-map-value map-exn-def reaches-liftE rel-liftE-def rel-xval.simps split*: *xval-splits*)

lemma *refines-L2-seq-nondet-polymorphic*:
assumes *r* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (*Trueprop* (*refines f f' s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) (*bind f' g'*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-seq-def*
apply (*rule refines-bind* [*OF f*])
apply (*auto simp add*: *r rel-liftE-def*)
done

lemma *refines-L2-seq-nondet*:
fixes *f'*:: ('*a*, '*s*) *res-monad*
assumes *r* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (*Trueprop* (*refines f f' s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) ((*bind f' g'*)::('*b*, '*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
using *assms*
by (*rule refines-L2-seq-nondet-polymorphic*)

lemma *refines-L2-seq-exn*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod* (*rel-xval L* (=)) (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-xval L* (=)) (=))))
shows *refines* (*L2-seq f g*) (*bind f' g'*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
unfolding *L2-seq-def*
apply (*rule refines-bind* [*OF f*])
apply (*auto simp add*: *g rel-xval.simps default-option-def Exn-def*)
done

lemma *refines-try-bind-rel-liftE'*:
assumes *g* : $\bigwedge v\ s\ t.\ S\ s\ t \Longrightarrow$ *refines* (*try* (*g v*)) (*g' v*) *s t* (*rel-prod rel-liftE S*)
assumes *f*: *refines f f' s t* (*rel-prod rel-liftE S*)
shows *refines* (*try* (*bind f g*)) (*bind f' g'*) *s t* (*rel-prod rel-liftE S*)
unfolding *try-def refines-map-value-left-iff*
apply (*rule refines-bind* [*OF f*])
subgoal by *auto*
subgoal by (*auto simp add*: *default-option-def Exn-def* [*symmetric*] *rel-liftE-def*)
subgoal by *auto*
subgoal
using *g* [*simplified L2-defs try-def refines-map-value-left-iff*]
by *auto*
done

lemma *refines-try-bind-rel-liftE*:
assumes *g* : $\bigwedge v\ t.$ *refines* (*try* (*g v*)) (*g' v*) *t t* (*rel-prod rel-liftE* (=))
assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines* (*try* (*bind f g*)) (*bind f' g'*) *s s* (*rel-prod rel-liftE* (=))
apply (*rule refines-try-bind-rel-liftE'* [*OF - f*])
using *g* **by** *simp*

lemma *refines-L2-try-L2-seq-nondet*:
assumes *g* [*unfolded THIN-def L2-defs*, *rule-format*]: *PROP THIN* ($\bigwedge v\ t.$ *refines* (*L2-try* (*g v*)) (*g' v*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f f' s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*bind f' g'*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs* **using** *g f*
by (*rule refines-try-bind-rel-liftE*)

lemma *refines-L2-try-L2-condition-nondet*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try f*) *f' s s* (*rel-prod rel-liftE* (=))))
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try g*) *g' s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-condition c f g*)) (*condition c f' g'*) *s s* (*rel-prod rel-liftE* (=))
using *f g* **unfolding** *L2-defs*
apply (*auto simp add*: *refines-def-old succeeds-condition-iff reaches-condition-iff reaches-try*)
done

lemma *refines-L2-try-rel-LiftE-nondet*:
assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) *f' s s* (*rel-prod rel-liftE* (=))
using *f* **unfolding** *L2-defs*
apply (*clarsimp simp add*: *refines-def-old reaches-try rel-liftE-def unnest-exn-def split*: *xval-splits*)

by (*metis Exn-def default-option-def exception-or-result-cases option.exhaust-sel*)

lemma *refines-L2-catch-rel-LiftE-nondet*:
assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-catch f g*) *f' s s* (*rel-prod rel-liftE* (=))
using *f* **unfolding** *L2-defs*
apply (*auto simp add*: *refines-def-old reaches-catch succeeds-catch rel-liftE-def unnest-exn-def split*: *xval-splits*)
subgoal by *fastforce*
subgoal by (*metis Exn-def default-option-def exception-or-result-cases not-None-eq*)
done

lemma *refines-try-finally-rel-liftE*:
assumes *f*: *refines f f' s s* (*rel-prod* (*rel-xval rel-liftE'* (=)) (=))
shows *refines* (*try f*) (*finally f'*) *s s* (*rel-prod rel-liftE* (=))
using *f* **unfolding** *try-def finally-def*
apply (*rule refines-map-value*)
apply (*auto simp add*: *unnest-exn-def unite-def rel-liftE'-def split*: *xval-splits sum.splits*)
done

lemma *refines-L2-try-finally-nondet*:
assumes *f*: *refines f f' s s* (*rel-prod* (*rel-xval rel-liftE'* (=)) (=))
shows *refines* (*L2-try f*) (*finally f'*) *s s* (*rel-prod rel-liftE* (=))
using *f* **unfolding** *L2-defs*
by (*rule refines-try-finally-rel-liftE*)

lemma *refines-L2-try-exn*:
assumes *f*: *refines f f' s s* (*rel-prod* (*rel-xval* (*rel-sum L* (=)) (=)) (=))
shows *refines* (*L2-try f*) (*try f'*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
using *f* **unfolding** *L2-defs try-def*
apply (*rule refines-map-value*)
apply (*auto simp add*: *unnest-exn-def split*: *xval-splits sum.splits*)
done

lemma *refines-L2-condition-nondet*:
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines g g' s s* (*rel-prod V* (=))))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f f' s s* (*rel-prod V* (=))))
shows *refines* (*L2-condition c f g*) (*condition c f' g'*) *s s* (*rel-prod V* (=))
unfolding *L2-condition-def* **using** *g f*
apply (*auto intro*: *refines-condition*)
done

lemma *refines-L2-while-nondet*:
assumes *b* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*b v*) (*b' v*) *t t* (*rel-prod rel-liftE* (=)))

shows *refines* (*L2-while c b i ns*) (*L2-VARS* (*whileLoop c b′ i*) *ns*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-while-def L2-VARS-def*
apply (*rule refines-whileLoop′′*)
subgoal
by (*auto*)
subgoal
by (*auto intro*: *b*)
subgoal
by *auto*
subgoal
by (*auto simp add*: *rel-liftE-def rel-exception-or-result.simps*)
done

lemma *refines-L2-while-exn*:
assumes *b* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*b v*) (*b′ v*) *t t* (*rel-prod* (*rel-xval L* (=)) (=)))
shows *refines* (*L2-while c b i ns*) (*L2-VARS* (*whileLoop c b′ i*) *ns*) *s s* (*rel-prod* ((*rel-xval L* (=))) (=))
unfolding *L2-while-def L2-VARS-def*
apply (*rule refines-whileLoop-exn*)
apply (*auto simp add*: *b*)
done

lemma *refines-L2-unknown-nondet*:
shows *refines* (*L2-unknown ns*) (*L2-VARS* (*select UNIV*) *ns*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-VARS-def*
apply (*auto intro*: *refines-select*)
done

lemma *refines-L2-unknown-exn*:
shows *refines* (*L2-unknown ns*) (*L2-VARS* (*select UNIV*) *ns*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
unfolding *L2-defs L2-VARS-def*
apply (*auto intro*: *refines-select*)
done

lemma *refines-L2-modify-nondet*:
shows *refines* (*L2-modify f*) (*modify f*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs*
apply (*auto intro*: *refines-modify*)
done

lemma *refines-L2-gets-nondet*:
shows *refines* (*L2-gets f ns*) (*L2-VARS* (*gets f*) *ns*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs L2-VARS-def*

apply (*auto intro*: *refines-gets*)
done

lemma *refines-L2-throw-exn*:
L x x' $\Longrightarrow$
refines (*L2-throw x ns*) (*L2-VARS* (*throw x'*) *ns*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
unfolding *L2-throw-def L2-VARS-def*
apply (*auto simp add*: *refines-def-old rel-xval.simps Exn-def*)
done

lemma *refines-L2-spec-nondet*:
shows *refines* (*L2-spec r*) (*assert-result-and-state* ($\lambda s.$ $\{(v,\ t).\ (s,\ t) \in r\}$)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs*
apply (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)
done

lemma *refines-L2-assume-nondet*:
shows *refines* (*L2-assume r*) (*assume-result-and-state r*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs*
apply (*auto simp add*: *refines-def-old reaches-bind succeeds-bind*)
done

lemma *refines-L2-guard-nondet*:
shows *refines* (*L2-guard c*) (*guard c*) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-defs*
by (*auto intro*: *refines-guard*)

lemma *refines-L2-guarded-nondet*:
assumes *f*: *c s* $\Longrightarrow$ *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-guarded c f*) (*bind* (*guard c*) (λ-. *f'*)) *s s* (*rel-prod rel-liftE* (=))
unfolding *L2-guarded-def L2-seq-def L2-guard-def* **using** *f*
apply (*auto simp add*: *refines-def-old succeeds-bind reaches-bind*)
done

lemma *refines-L2-guarded-exn*:
assumes *f*: *c s* $\Longrightarrow$ *refines f f' s s* (*rel-prod* (*rel-xval L* (=)) (=))
shows *refines* (*L2-guarded c f*) (*bind* (*guard c*) (λ-. *f'*)) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
unfolding *L2-guarded-def L2-seq-def L2-guard-def* **using** *f*
apply (*auto simp add*: *refines-def-old succeeds-bind reaches-bind*)
done

lemma *refines-L2-fail-nondet*:
shows *refines L2-fail fail s s* (*rel-prod R* (=))

by *simp*

lemma *refines-exec-concrete-gen-nondet*:
assumes *f*: $\bigwedge s$. *refines f f' s s* (*rel-prod R* (=))
shows *refines* (*exec-concrete st f*) (*exec-concrete st f'*) *s s* (*rel-prod R* (=))
using *f*
by (*fastforce simp add*: *refines-def-old reaches-exec-concrete succeeds-exec-concrete-iff*)

lemmas *refines-exec-concrete-nondet* = *refines-exec-concrete-gen-nondet* [**where** *R*=*rel-liftE*]
lemmas *refines-exec-concrete-exit* = *refines-exec-concrete-gen-nondet* [**where** *R*=*rel-xval L* (=)] **for** *L*

lemma *refines-exec-abstract-gen-nondet*:
assumes *f*: $\bigwedge s$. *refines f f' s s* (*rel-prod R* (=))
shows *refines* (*exec-abstract st f*) (*exec-abstract st f'*) *s s* (*rel-prod R* (=))
using *f*
by (*fastforce simp add*: *refines-def-old reaches-exec-abstract*)

lemmas *refines-exec-abstract-nondet* = *refines-exec-abstract-gen-nondet* [**where** *R*=*rel-liftE*]
lemmas *refines-exec-abstract-exit* = *refines-exec-abstract-gen-nondet* [**where** *R*=*rel-xval L* (=)] **for** *L*

lemma *rel-map-ResultI*:
rel-map Result x (*Result x*)
by (*simp add*: *rel-map-def*)

lemma *rel-map-to-xval-InlI*:
rel-map to-xval (*Inl l*) (*Exn l*)
by (*simp add*: *rel-map-def*)

lemma *rel-map-to-xval-InrI*:
rel-map to-xval (*Inr r*) (*Result r*)
by (*simp add*: *rel-map-def*)

lemma *refines-rel-prod-eq-guard-on-exit*:
assumes *f*: *refines* f_c f_a *s s* (*rel-prod Q* (=))
shows *refines*
(*guard-on-exit* f_c *grd cleanup*)
(*guard-on-exit* f_a *grd cleanup*) *s s*
(*rel-prod Q* (=))
unfolding *guard-on-exit-bind-exception-or-result-conv*
apply (*rule refines-bind-exception-or-result-strong* [*OF f*])
apply *clarsimp*
apply (*rule refines-bind-no-throw* [**where** *Q*=*rel-prod* ⊤ (=)])

apply *simp*
apply *simp*
apply (*rule refines-guard*)
apply *simp*
apply *simp*
apply (*rule refines-bind-no-throw*)
apply *simp*
apply *simp*
apply (*clarsimp*)
apply (*rule refines-state-select-same*)
apply *clarsimp*
done

lemma *refines-rel-prod-eq-assume-on-exit*:
assumes *f*: *refines* f_c f_a *s s* (*rel-prod Q* (=))
shows *refines*
(*assume-on-exit* f_c *grd cleanup*)
(*assume-on-exit* f_a *grd cleanup*) *s s*
(*rel-prod Q* (=))
unfolding *assume-on-exit-bind-exception-or-result-conv*
apply (*rule refines-bind-exception-or-result-strong* [*OF f*])
apply *clarsimp*
apply (*rule refines-bind-no-throw* [**where** *Q*=*rel-prod* ⊤ (=)])
apply *simp*
apply *simp*
subgoal
by (*auto simp add*: *refines-def-old*)
apply (*rule refines-bind-no-throw*)
apply *simp*
apply *simp*
apply (*clarsimp*)
apply (*rule refines-state-select-same*)
apply *clarsimp*
done

context *stack-heap-state*
begin

thm *refines-rel-prod-with-fresh-stack-ptr*
lemma *refines-rel-prod-L2-try-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c$ *s* = $init_a$ *s*
assumes *f*: ⋀*s p*. *refines* (*L2-try* (f_c *p*)) (f_a *p*) *s s* (*rel-prod Q* (=))
shows
refines
(*L2-try* (*with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*)))
(*with-fresh-stack-ptr n* $init_a$ (*L2-VARS* f_a *nm*)) *s s*
(*rel-prod Q* (=))
apply (*simp add*: *L2-try-with-fresh-stack-ptr L2-VARS-def*)
apply (*rule refines-rel-prod-with-fresh-stack-ptr* [*unfolded L2-VARS-def*])

```
  apply (rule init-eq)
 apply (rule f)
 done
end

context typ-heap-typing
begin

lemma refines-rel-prod-guard-with-fresh-stack-ptr:
 assumes init-eq: init_c s = init_a s
 assumes f: ⋀s p. refines (f_c p) (f_a p) s s (rel-prod L (=))
 shows
   refines
     (guard-with-fresh-stack-ptr n init_c (L2-VARS f_c nm))
     (guard-with-fresh-stack-ptr n init_a (L2-VARS f_a nm)) s s
     (rel-prod L (=))
 apply (simp add: guard-with-fresh-stack-ptr-def L2-VARS-def assume-stack-alloc-def)
 apply (rule refines-bind')
 apply (subst refines-assume-result-and-state-iff)
 apply (subst init-eq)
 apply (intro sim-set-refl)
 apply clarsimp
 apply (rule refines-on-exit')
 apply (rule refines-weaken[OF f])
 apply simp
 apply (rule refines-bind'[OF refines-guard])
 apply simp-all
 apply (rule refines-assert-result-and-state)
 apply auto
 done

lemma refines-rel-prod-assume-with-fresh-stack-ptr:
 assumes init-eq: init_c s = init_a s
 assumes f: ⋀s p. refines (f_c p) (f_a p) s s (rel-prod L (=))
 shows
   refines
     (assume-with-fresh-stack-ptr n init_c (L2-VARS f_c nm))
     (assume-with-fresh-stack-ptr n init_a (L2-VARS f_a nm)) s s
     (rel-prod L (=))
 apply (simp add: assume-with-fresh-stack-ptr-def L2-VARS-def assume-stack-alloc-def)
 apply (rule refines-bind')
 apply (subst refines-assume-result-and-state-iff)
 apply (subst init-eq)
 apply (intro sim-set-refl)
 apply clarsimp
 apply (rule refines-on-exit')
 apply (rule refines-weaken[OF f])
 apply simp
 apply (rule refines-bind'[OF refines-assuming])
```

apply *simp-all*
apply (*rule refines-assert-result-and-state*)
apply *auto*
done

lemma *refines-rel-prod-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c\ s = init_a\ s$
assumes *f*: $\bigwedge s\ p.\ refines\ (f_c\ p)\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ L\ (=))$
shows
refines
($with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ (L2\text{-}VARS\ f_c\ nm)$)
($with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_a\ (L2\text{-}VARS\ f_a\ nm)$) *s s*
(*rel-prod L* (=))
apply (*simp add*: *with-fresh-stack-ptr-def L2-VARS-def assume-stack-alloc-def*)
apply (*rule refines-bind-no-throw*)
apply *simp*
apply *simp*
apply (*rule refines-assume-result-and-state-same'*)
apply (*simp only*: *init-eq*)
apply *clarsimp*
apply (*rule refines-rel-prod-eq-on-exit*)
apply (*rule f*)
done

lemma *refines-rel-prod-L2-try-guard-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c\ s = init_a\ s$
assumes *f*: $\bigwedge s\ p.\ refines\ (L2\text{-}try\ (f_c\ p))\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ L\ (=))$
shows
refines
($L2\text{-}try\ (guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ (L2\text{-}VARS\ f_c\ nm))$)
($guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_a\ (L2\text{-}VARS\ f_a\ nm)$) *s s*
(*rel-prod L* (=))
apply (*simp add*: *L2-try-guard-with-fresh-stack-ptr L2-VARS-def*)
apply (*rule refines-rel-prod-guard-with-fresh-stack-ptr* [*unfolded L2-VARS-def*])
apply (*rule init-eq*)
apply (*rule f*)
done

lemma *refines-rel-prod-L2-try-assume-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c\ s = init_a\ s$
assumes *f*: $\bigwedge s\ p.\ refines\ (L2\text{-}try\ (f_c\ p))\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ L\ (=))$
shows
refines
($L2\text{-}try\ (assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ (L2\text{-}VARS\ f_c\ nm))$)
($assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_a\ (L2\text{-}VARS\ f_a\ nm)$) *s s*
(*rel-prod L* (=))
apply (*simp add*: *L2-try-assume-with-fresh-stack-ptr L2-VARS-def*)
apply (*rule refines-rel-prod-assume-with-fresh-stack-ptr* [*unfolded L2-VARS-def*])

apply (*rule init-eq*)
apply (*rule f*)
done

lemma *refines-rel-prod-L2-try-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c\ s = init_a\ s$
assumes *f*: $\bigwedge s\ p.$ *refines* (*L2-try* ($f_c\ p$)) ($f_a\ p$) *s s* (*rel-prod L* (=))
shows
refines
(*L2-try* (*with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*)))
(*with-fresh-stack-ptr n* $init_a$ (*L2-VARS* f_a *nm*)) *s s*
(*rel-prod L* (=))
apply (*simp add*: *L2-try-with-fresh-stack-ptr L2-VARS-def*)
apply (*rule refines-rel-prod-with-fresh-stack-ptr* [*unfolded L2-VARS-def*])
apply (*rule init-eq*)
apply (*rule f*)
done

end

lemma *relcomppI-swapped*: *s b c* $\Longrightarrow$ *r a b* $\Longrightarrow$ (*r OO s*) *a c*
by (*rule relcomppI*)

lemmas *refines-monad-nondet* =
refines-L2-call-embed-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-call-embed-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-liftE-exn [*synthesize-rule nondet* **and** *exit priority*:*250*]
refines-L2-seq-nondet [*synthesize-rule nondet* **and** *exit priority*:*210 split*: *g* **and** *g'*]
refines-L2-seq-exn [*synthesize-rule nondet* **and** *exit priority*:*210 split*: *g* **and** *g'*]

refines-L2-catch-rel-LiftE-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-try-L2-seq-nondet [*synthesize-rule nondet* **and** *exit priority*:*230 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-nondet [*synthesize-rule nondet* **and** *exit priority*:*230*]
refines-L2-try-rel-LiftE-nondet [*synthesize-rule nondet* **and** *exit priority*:*220*]
refines-L2-try-finally-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-try-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-condition-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-while-nondet [*synthesize-rule nondet* **and** *exit priority*:*210 split*: *b* **and** *b'*]
refines-L2-while-exn [*synthesize-rule nondet* **and** *exit priority*:*210 split*: *b* **and** *b'*]

refines-L2-unknown-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-unknown-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-modify-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-gets-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-throw-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-spec-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-assume-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-guard-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-guarded-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-guarded-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-fail-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-exec-concrete-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-exec-concrete-exit [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-exec-abstract-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-exec-abstract-exit [*synthesize-rule nondet* **and** *exit priority*:*210*]

rel-liftE-trivial [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-liftE′-Inr [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-sum-Inl [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-sum-Inr [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-xval-Exn [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-xval-Result [*synthesize-rule nondet* **and** *exit priority*:*210*]

relcomppI-swapped [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-map-ResultI [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-map-to-xval-InlI [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-map-to-xval-InrI [*synthesize-rule nondet* **and** *exit priority*:*210*]

refl [*synthesize-rule nondet* **and** *exit priority*: *210*]

context *typ-heap-typing*
begin
lemmas *refines-monad-nondet* =
refines-rel-prod-L2-try-guard-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*232*]
refines-rel-prod-L2-try-assume-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*232*]
refines-rel-prod-L2-try-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit pri-*

ority:*232*]
refines-rel-prod-guard-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-prod-assume-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-prod-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-xval-guard-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-xval-assume-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-xval-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
end

context *stack-heap-state*
begin
lemmas *refines-nondet-monad* =
refines-rel-prod-L2-try-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*232*]
refines-rel-prod-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
end

context *heap-state*
begin
lemmas *refines-nondet-monad'* =
refines-IO-modify-heap-paddingE-nondet[*synthesize-rule nondet* **and** *exit priority*:*210*]
end
add-synthesize-pattern *nondet* **and** *exit* **where**
rel-throwE-nondet-synth = ‹*Trueprop* (*rel-throwE* - *?x ?x'*)› (*x'*) **and**
rel-throwE'-nondet-synth = ‹*Trueprop* (*rel-throwE'* - *?x ?x'*)› (*x'*)

23.5.1 Elimination of *L2-try* in the Error Monad

Eliminate Inner Exception

rules for elimination of *L2-try* when the inner exception layer is not needed, i.e., *rel-sum* (*rel-throwE L*) (=)

lemma *bind-bind-exception-or-result-conv*:
(*f* ≫= *g*) = (*bind-exception-or-result f* (λ*Exception e* ⇒ *yield* (*Exception e*) | *Result v* ⇒ *g v*))
apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*auto simp add*: *runs-to-def-old split*: *exception-or-result-splits*)
done

lemma *rel-throwE'-rel-throwE-conv*: *rel-throwE' L* = (*rel-map to-xval OO rel-throwE L*)
apply (*rule ext*)+
apply (*auto simp add*: *rel-throwE-def rel-throwE'-def rel-xval.simps relcompp.simps*

rel-map-def to-xval-def split: *sum.splits*)
done

lemma *rel-throwE* (*rel-throwE'* *L*) = (*rel-throwE* (*rel-map to-xval OO rel-throwE* *L*))
apply (*rule ext*)+
apply (*auto simp add*: *rel-throwE-def rel-throwE'-def rel-xval.simps relcompp.simps*

rel-map-def to-xval-def split: *sum.splits*)
done

lemma *refines-L2-try-L2-seq-exn*:
fixes *f*:: ((*'b* + *'c*), *'f*, *'a*) *exn-monad*
and *g*::*'f* ⇒ ((*'b* + *'c*), *'c*, *'a*) *exn-monad*
assumes *g* [*unfolded THIN-def L2-defs*, *rule-format*]:
PROP THIN (⋀*v t*. *refines* (*L2-try* (*g v*)) (*g'* *v*) *t t* (*rel-prod* ((*rel-xval L R*)) (=)))
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-xval* (*rel-throwE'* *L*) (=)) (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*bind f' g'*) *s s* (*rel-prod* (*rel-xval L R*) (=))
unfolding *L2-try-def L2-seq-def try-nested-bind-exception-or-result-conv*
apply (*simp add*: *bind-bind-exception-or-result-conv*)
apply (*rule refines-bind-exception-or-result*)
apply (*rule refines-weaken* [*OF f*])
apply (*auto split*: *xval-splits sum.splits simp*: *rel-xval.simps rel-prod.simps g*)
done

lemma *refines-L2-try-L2-condition-exit*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try f*) *f'* *s s* (*rel-prod R* (=))))
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try g*) *g'* *s s* (*rel-prod R* (=))))
shows *refines* (*L2-try* (*L2-condition c f g*)) (*condition c f' g'*) *s s* (*rel-prod R* (=))
using *f g* **unfolding** *L2-defs L2-try-def*
apply (*auto simp add*: *refines-def-old succeeds-condition-iff reaches-condition-iff reaches-try*)
done

lemma *refines-try-rel-xval-rel-throwE'*:
assumes *refines f f' s s* (*rel-prod* (*rel-xval* (*rel-throwE'* *A*) *B*) *S*)
shows *refines* (*try f*) *f' s s* (*rel-prod* (*rel-xval A B*) *S*)
unfolding *try-def refines-map-value-left-iff*
apply (*rule refines-weaken* [*OF assms*])
apply (*auto simp add*: *unnest-exn-def rel-xval.simps split*: *xval-splits sum.splits*)
done

lemma *refines-L2-try-rel-sum-rel-throwE-nondet*:
assumes *refines f f′ s s* (*rel-prod* (*rel-xval* (*rel-throwE′ A*) *B*) (=))
shows *refines* (*L2-try f*) *f′ s s* (*rel-prod* (*rel-xval A B*) (=))
using *assms* **unfolding** *L2-defs*
by (*rule refines-try-rel-xval-rel-throwE′*)

lemma *refines-try-rel-throwE*:
assumes *refines f f′ s s* (*rel-prod* (*rel-throwE* (*rel-throwE′ L*)) *S*)
shows *refines* (*try f*) *f′ s s* (*rel-prod* (*rel-throwE L*) *S*)
unfolding *try-def*
apply (*simp add*: *refines-map-value-left-iff*)
apply (*rule refines-weaken* [*OF assms*])
apply (*auto simp add*: *unnest-exn-def split*: *xval-splits sum.splits*)
done

lemma *refines-L2-try-rel-throwE-nondet*:
assumes *refines f f′ s s* (*rel-prod* (*rel-throwE* (*rel-throwE′ L*)) (=))
shows *refines* (*L2-try f*) *f′ s s* (*rel-prod* (*rel-throwE L*) (=))
using *assms* **unfolding** *L2-defs*
by (*rule refines-try-rel-throwE*)

lemmas *ts-L2-try-inner-exception* =
rel-throwE′-Inl[*synthesize-rule nondet* **and** *exit*]
rel-throwE-Exn[*synthesize-rule nondet* **and** *exit*]
refines-L2-try-L2-seq-exn [*synthesize-rule nondet* **and** *exit priority*: *218 split*: *g* **and** *g′*]
refines-L2-try-L2-condition-exit [*synthesize-rule nondet* **and** *exit priority*: *218*]
refines-L2-try-rel-throwE-nondet [*synthesize-rule nondet* **and** *exit priority*: *216*]
refines-L2-try-rel-sum-rel-throwE-nondet [*synthesize-rule nondet* **and** *exit priority*: *215*]
bundle *del-ts-L2-try-inner-exception* =
rel-throwE′-Inl[*synthesize-rule nondet* **and** *exit del*]
rel-throwE-Exn[*synthesize-rule nondet* **and** *exit del*]
refines-L2-try-L2-seq-exn [*synthesize-rule nondet* **and** *exit priority*: *218 split*: *g* **and** *g′ del*]
refines-L2-try-L2-condition-exit [*synthesize-rule nondet* **and** *exit priority*: *218 del*]
refines-L2-try-rel-throwE-nondet [*synthesize-rule nondet* **and** *exit priority*: *216 del*]
refines-L2-try-rel-sum-rel-throwE-nondet [*synthesize-rule nondet* **and** *exit priority*: *215 del*]

print-synthesize-rules *exit* ‹*Trueprop* (*refines* (*L2-seq* - -) - - - (*rel-prod rel-liftE* (=))) ›

Eliminate *L2-try* over exiting branches

Eliminate *L2-try* over a condition, when one branch always exits (*rel-throwE* (*rel-sum L R*))

lemma *rel-map-to-xval-rel-xval-rel-sum-conv*:

rel-map to-xval OO rel-xval L R = rel-sum L R OO rel-map to-xval
apply (*rule ext*)+
apply (*auto simp add*: *rel-sum.simps rel-xval.simps relcompp.simps*
rel-map-def to-xval-def split: *sum.splits*)
done

lemma *refines-L2-try-L2-seq-L2-condition-exit1*:
assumes *g* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines* (*L2-try* (*L2-seq g h*)) *gh′ s s* (*rel-prod LR* (=))))
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f′ s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO LR*)) (=))))
shows *refines* (*L2-try* (*L2-seq* (*L2-condition c f g*) *h*)) (*condition c f′ gh′*) *s s* (*rel-prod LR* (=))
using *g f*
apply (*simp add*: *L2-defs try-def refines-map-value-left-iff refines-condition-bind-left*
refines-condition-false)
apply (*intro impI refines-condition-true*)
apply *assumption*
subgoal premises *prems*
proof −
have *refines* (*f* >>= *h*) (*f′* ≫= *return*) *s s* (λ(*x*, *s*) (*y*, *t*). *LR* (*unnest-exn x*) *y* ∧ *s* = *t*)
apply (*intro refines-bind*[*OF prems*(*2*)])
apply (*auto simp*: *rel-throwE-def rel-map-def unnest-exn-def*
split: *xval-splits sum.splits*)
done
then show *?thesis* **by** *simp*
qed
done

lemma *refines-L2-try-L2-seq-L2-condition-exit2*:
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines* (*L2-try* (*L2-seq f h*)) *fh′ s s* (*rel-prod LR* (=))))
assumes *g* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines g g′ s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO LR*)) (=))))
shows *refines* (*L2-try* (*L2-seq* (*L2-condition c f g*) *h*)) (*condition c fh′ g′*) *s s* (*rel-prod LR* (=))
using *assms*
unfolding *L2-defs*
apply (*subst* (*1 2*) *condition-swap*)
apply (*erule* (*1*) *refines-L2-try-L2-seq-L2-condition-exit1*[*unfolded L2-defs*])
done

lemma *refines-L2-seq-L2-condition-rel-throwE1*:
assumes *g* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines* (*L2-seq g h*) *gh′ s s* (*rel-prod* (*rel-throwE*

```
LR)(=))))
  assumes f [unfolded THIN-def]:
    PROP THIN (Trueprop (refines f f' s s (rel-prod (rel-throwE LR) (=))))
  shows refines (L2-seq (L2-condition c f g) h) (condition c f' gh') s s (rel-prod
(rel-throwE LR) (=))
proof cases
  assume c s
  from this f
  show ?thesis
    unfolding L2-defs
     by (auto simp add: refines-def-old reaches-condition-iff succeeds-condition-iff
succeeds-bind reaches-bind rel-throwE-def
        default-option-def Exn-def
        split: xval-splits exception-or-result-splits)
         (metis default-option-def the-Exception-simp exception-or-result-cases op-
tion.exhaust-sel)+
next
  assume ¬c s
  then have *:
    run ((condition c f g) >>= h) s = run (g >>= h) s
    run (condition c f' gh') s = run gh' s
    by (auto simp add: run-condition run-bind)

  from g show ?thesis
    unfolding refines-def-old succeeds-def reaches-def
      L2-defs * .
qed



lemma refines-L2-seq-L2-condition-rel-throwE2:
  assumes f [unfolded THIN-def]:
    PROP THIN (Trueprop (refines (L2-seq f h) fh' s s (rel-prod (rel-throwE LR)
(=))))
  assumes g [unfolded THIN-def]:
    PROP THIN (Trueprop (refines g g' s s (rel-prod (rel-throwE LR) (=))))
  shows refines (L2-seq (L2-condition c f g) h) (condition c fh' g') s s (rel-prod
(rel-throwE LR) (=))
  using assms
  unfolding L2-defs
  by (subst (1 2) condition-swap)
    (erule (1) refines-L2-seq-L2-condition-rel-throwE1[unfolded L2-defs])

lemma refines-bind-rel-throwE-first:
  assumes f: refines f f' s s (rel-prod (rel-throwE LR) S)
  shows refines (f >>= g) f' s s (rel-prod (rel-throwE LR) S)
  using f unfolding L2-defs
  by (auto simp add: refines-def-old  succeeds-bind reaches-bind rel-throwE-def
      default-option-def Exn-def
```

split: *xval-splits exception-or-result-splits*)
(*metis default-option-def the-Exception-simp exception-or-result-cases option.exhaust-sel*)+

lemma *refines-L2-seq-rel-throwE-throwE*:
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-throwE LR*) (=))))
shows *refines* (*L2-seq f g*) *f' s s* (*rel-prod* (*rel-throwE LR*) (=))
unfolding *L2-defs* **using** *f*
by (*rule refines-bind-rel-throwE-first*)

lemma *refines-L2-seq-rel-throwE-throwE1*:
assumes *g*[*unfolded THIN-def*, *rule-format*] :
PROP THIN (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO rel-xval L R*)) (=)))
assumes *f* [*unfolded THIN-def*, *rule-format*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-xval* (*rel-throwE' L*) (=)) (=))))
shows *refines* (*L2-seq f g*) (*bind f' g'*) *s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO rel-xval L R*)) (=))
unfolding *L2-defs*
apply (*rule refines-bind-bind-exn* [*OF f*])
apply (*auto simp*: *g rel-throwE'-iff*)
done

lemma *refines-L2-seq-rel-throwE-liftE*:
assumes *g* [*unfolded THIN-def*, *rule-format*]:
PROP THIN (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod* (*rel-throwE LR*) (=)))
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) (*bind f' g'*) *s s* (*rel-prod* (*rel-throwE LR*) (=))
using *g f* **unfolding** *L2-defs*
apply (*clarsimp simp add*: *refines-def-old succeeds-bind reaches-bind reaches-try succeeds-condition-iff reaches-condition-iff*
rel-throwE-def rel-throwE'-def relcompp.simps rel-liftE-def rel-xval.simps
split: *xval-splits sum.splits*, *safe*)
apply (*metis the-Result-simp*)
apply (*smt* (*verit*) *case-exception-or-result-Result case-xval-simps*(*1*))
by (*metis* (*no-types*, *lifting*) *case-exception-or-result-Result case-xval-simps*(*2*))

lemma *refines-L2-throw-rel-throwE-Inl*:
assumes *L l l'*
shows *refines* (*L2-throw* (*Inl l*) *ns*) (*throw* (*L2-VARS l' ns*)) *s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO rel-xval L R*)) (=))
using *assms* **unfolding** *L2-defs L2-VARS-def*
by (*auto simp add*: *refines-def-old rel-throwE-def rel-xval.simps relcompp.simps rel-map-def to-xval-def Exn-def*
split: *sum.splits*)

lemma *refines-L2-throw-rel-throwE-Inr*:

assumes *R r r′*
shows *refines* (*L2-throw* (*Inr r*) *ns*) (*return* (*L2-VARS r′ ns*)) *s s* (*rel-prod* (*rel-throwE* ((*rel-map to-xval OO rel-xval L R*))) (=))
using *assms* **unfolding** *L2-defs L2-VARS-def*
by (*auto simp add*: *refines-def-old rel-throwE-def rel-xval.simps relcompp.simps rel-map-def to-xval-def Exn-def*
split: *sum.splits*)

lemma *refines-L2-throw-rel-throwE*:
assumes *R r* (*Result r′*)
shows *refines* (*L2-throw r ns*) (*L2-VARS* (*return r′*) *ns*) *s s* (*rel-prod* (*rel-throwE R*) (=))
using *assms* **unfolding** *L2-defs L2-VARS-def*
by (*auto simp add*: *refines-def-old rel-throwE-def rel-sum.simps relcompp.simps rel-map-def to-xval-def Exn-def*
split: *sum.splits*)

lemmas *ts-L2-try-condition-exit* =

refines-L2-try-L2-seq-L2-condition-exit1[*synthesize-rule nondet* **and** *exit priority*:*217*]
refines-L2-try-L2-seq-L2-condition-exit2[*synthesize-rule nondet* **and** *exit priority*:*217*]

refines-L2-seq-L2-condition-rel-throwE1[*synthesize-rule nondet* **and** *exit priority*:*216*]
— Note that 1. this together with *refines ?f ?f′ ?s ?s* (*rel-prod* (*rel-throwE* (*rel-throwE′ ?L*)) (=)) ⟹ *refines* (*L2-try ?f*) *?f′ ?s ?s* (*rel-prod* (*rel-throwE ?L*) (=)) does not replace 2. ⟦*THIN* (*refines* (*L2-try* (*L2-seq ?g ?h*)) *?gh′ ?s ?s* (*rel-prod ?LR* (=))); *THIN* (*refines ?f ?f′ ?s ?s* (*rel-prod* (*rel-throwE* (*rel-project to-xval OO ?LR*)) (=)))⟧ ⟹ *refines* (*L2-try* (*L2-seq* (*L2-condition ?c ?f ?g*) *?h*)) (*condition ?c*
?f′
?gh′) *?s ?s* (*rel-prod ?LR* (=)) and vice versa. 1 is more compositional and also works deeply nested but only handles the case *rel-throwE*, whereas 2 also handles *rel-liftE* and *rel-sum* by pushing down *L2-try* into the branch of the conditional.
refines-L2-seq-L2-condition-rel-throwE2[*synthesize-rule nondet* **and** *exit priority*:*216*]
refines-L2-seq-rel-throwE-throwE [*synthesize-rule nondet* **and** *exit priority*:*213*]
refines-L2-seq-rel-throwE-throwE1[*synthesize-rule nondet* **and** *exit priority*:*212 split*: *g* **and** *g′*]
refines-L2-seq-rel-throwE-liftE [*synthesize-rule nondet* **and** *exit priority*:*212 split*: *g* **and** *g′*]
refines-L2-throw-rel-throwE-Inl [*synthesize-rule nondet* **and** *exit priority*:*212*]
refines-L2-throw-rel-throwE-Inr [*synthesize-rule nondet* **and** *exit priority*:*212*]
refines-L2-throw-rel-throwE [*synthesize-rule nondet* **and** *exit priority*:*212*]
bundle *del-ts-L2-try-condition-exit* =
refines-L2-try-L2-seq-L2-condition-exit1[*synthesize-rule nondet* **and** *exit prior-*

ity:217 del]
refines-L2-try-L2-seq-L2-condition-exit2[*synthesize-rule nondet* **and** *exit priority:217 del*]
refines-L2-seq-L2-condition-rel-throwE1[*synthesize-rule nondet* **and** *exit priority:216 del*]
refines-L2-seq-L2-condition-rel-throwE2[*synthesize-rule nondet* **and** *exit priority:216 del*]
refines-L2-seq-rel-throwE-throwE [*synthesize-rule nondet* **and** *exit priority:213 del*]
refines-L2-seq-rel-throwE-throwE1[*synthesize-rule nondet* **and** *exit priority:212 split*: *g* **and** *g′ del*]
refines-L2-seq-rel-throwE-liftE [*synthesize-rule nondet* **and** *exit priority:212 split*: *g* **and** *g′ del*]
refines-L2-throw-rel-throwE-Inl [*synthesize-rule nondet* **and** *exit priority:212 del*]
refines-L2-throw-rel-throwE-Inr [*synthesize-rule nondet* **and** *exit priority:212 del*]
refines-L2-throw-rel-throwE [*synthesize-rule nondet* **and** *exit priority:212 del*]

23.6 Error Monad (exit)

lemma *refines-L2-call-embed-exit*:
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval rel-Nonlocal* (=)) (=)) (=))
assumes *emb*: ⋀*e′*. *L* (*emb* (*Nonlocal e′*)) (*emb′ e′*)
shows *refines* (*L2-call f emb ns*) (*L2-VARS* (*map-value* (*map-exn emb′*) *f′*) *ns*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
unfolding *L2-defs L2-VARS-def L2-call-def*
apply (*rule refines-map-value* [*OF f*])
using *emb*
by (*auto simp add*: *rel-xval.simps map-exn-def rel-Nonlocal-conv*)

lemma *refines-L2-call-embed-exit-in-out*:
assumes *emb*: ⋀*e′*. *L* (*emb e′*) (*emb′ e′*)
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval* (=) (=)) (=))
shows *refines* (*L2-call f emb ns*) (*L2-VARS* (*map-value* (*map-exn emb′*) *f′*) *ns*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
unfolding *L2-defs L2-VARS-def L2-call-def*
apply (*rule refines-map-value* [*OF f*])
using *emb*
by (*auto simp add*: *rel-xval.simps map-exn-def rel-Nonlocal-conv*)

lemma *refines-L2-catch-exit*:
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval* (=) *R*) (=))
assumes *h*: ⋀*s′ v*. *refines* (*h v*) (*h′ v*) *s′ s′* (*rel-prod* (*rel-xval L R*) (=))
shows *refines* (*L2-catch f h*) (*catch f′ h′*) *s s* (*rel-prod* (*rel-xval L R*) (=))
unfolding *L2-catch-def*
using *f h*
apply (*auto intro*!: *refines-catch*)
done

lemma *rel-xval-eq-refl*: *rel-xval* (=) (=) *x x*
by (*auto simp add*: *rel-xval-eq*)

lemmas *refines-monad-exit* =
refines-L2-call-embed-exit [*synthesize-rule exit priority*:*110*]
refines-L2-call-embed-exit-in-out [*synthesize-rule exit priority*:*109 split*: *emb* **and** *emb'*]
refines-L2-catch-exit [*synthesize-rule exit priority*:*110 split*: *h* **and** *h'*]
rel-Nonlocal-Nonlocal [*synthesize-rule exit priority*:*110*]
rel-sum-eq [*synthesize-rule exit priority*:*210*]
rel-xval-eq-refl [*synthesize-rule exit priority*:*210*]

print-synthesize-rules *exit*

end

theory *TypeStrengthen*
imports
Refines-Spec
begin

ML-file *monad-types.ML*

lemma *gets-the-ogets-return-conv* [*fun-ptr-simps*]: *gets-the* (*ogets* (λ-. *f*)) = *return f*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff ogets-def*)
done

lemma *gets-the-ogets-gets-conv* [*fun-ptr-simps*]: *gets-the* (*ogets f*) = *gets f*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff ogets-def*)
done

lemma *gets-the-ogets*: *gets-the* (*ogets f*) = *gets f*
apply (*simp add*: *gets-the-def assert-opt-def*[*abs-def*] *ogets-def gets-def*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *gets-the-obind*:
gets-the (*f* |>> *g*) = *gets-the f* >>= (λ*x*. *gets-the* (*g x*))
apply (*simp add*: *obind-def*)

apply (*rule spec-monad-eqI*)
apply (*clarsimp simp add*: *runs-to-iff*)
apply (*auto split*: *option.splits*)
done

lemma *gets-the-oguard*: *gets-the* (*oguard P*) = *guard P*
apply (*simp add*: *oguard-def*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *gets-the-ocondition*:
gets-the (*ocondition P f g*) = *condition P* (*gets-the f*) (*gets-the g*)
apply (*simp add*: *ocondition-def*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

A best-effort approach to determine the simplest possible 'monad' for the final definition is implemented. We first try to define a function into the most restrictive monad and if that fails successively try more expressive monads until we finally hit the most expressive monad. In the original autocorres version this phase was based on equations and not on on a simulation relation as all the other autocorres phases. With the switch to model recursive functions with a CCPO **fixed-point** instead of **function** with an explicit measure parameter this did no longer work, as equations are not *ccpo.admissible*. Fortunately simulation is admissible, so we changed this phase to *refines*, cf: `Refines_Spec.thy`. So the main purpose of this theory is to set up the available target monads by applying some meta information: `monad_types.ML`.

The correspondence equations have the format:

(*) p = *L2-call-lift* p'

where *L2-call-lift* depends on the (simpler) target monad and lifts the program p' from that simpler monad to the fully fledged monad we start with:

The program p is the definition we have from the last layer of autocorres (WA). The final definition will refer to p'.

For the code to work *L2-call-lift* has to be a distinct constant, as some matching is performed on that assumption. That is why some new definitions are introduced below.

Note that the final (most expressive) monad is characterised by the lifting function is *lift-exit-status* which merely removes the exception handling artefact from SIMPL by extracting the exception value *'a* from *'a c-exntype*. So it should be sufficiently expressive for any input C program.

When the proof for a certain monad fails it can either have a good reason (as the input function is just not expressible in that particular monad) or it

can fail because the implementation is missing some rules.

Note some peculiarities on the current state of affairs:

- When a guard remains (e.g. bounds for an integer) you end up at least in the option monad (to model the possible failure).
- As recursive functions are currently implemented with **fixed-point** they are at least in the option monad.

setup ‹
Monad-Types.new-monad-type
pure
Pure function
(∗ *unused ccpo-name for recursive definitions* ∗)
100
#resT
(*fn - => fn - => error monad-type pure*: *there is no previous monad to lift from*)
{*rules-name* = @{*synthesize-rules-name pure*},
relator = @{*term rel-liftE*::('*a*, '*b*) *xval* ⇒ '*b val* ⇒ *bool*},
relator-from-c-exntype = *NONE*, *lift* = @{*term return*},
dest-lift = (*fn* @{*term-pat return ?x*} => *SOME x* | - => *NONE*),
lift-prev = []}
|> *Context.theory-map*
›

setup ‹
Monad-Types.new-monad-type
gets (∗ *reader monad* ∗)
Read−only function
(∗ *unused ccpo-name for recursive definitions* ∗)
80
(*fn* {*stateT*, *resT*, *exT*} => *stateT* −−> *resT*)
(*fn - => fn stateT => let fun lift t = Abs* (-, *stateT*, *t*) *in Utils.lift-result-with-arity 0 lift end*)
{*rules-name* = @{*synthesize-rules-name reader*},
relator = @{*term rel-liftE*::('*a*, '*b*) *xval* ⇒ '*b val* ⇒ *bool*},
relator-from-c-exntype = *NONE*, *lift* = @{*term gets*},
dest-lift = (*fn* @{*term-pat gets ?x*} => *SOME x* | - => *NONE*),
lift-prev = @{*thms refines-lift-pure-reader*}}
|> *Context.theory-map*
›

lemma *monotone-L2-VARS* [*partial-function-mono*]:

monotone R X a $\Longrightarrow$ *monotone R X* (λf. *L2-VARS* (*a f*) *ns*)
by (*simp add*: *L2-VARS-def*)

lemma *monotone-ocondition* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (*fun-ord Q*) *X*
assumes *mono-Y*: *monotone R* (*fun-ord Q*) *Y*
shows *monotone R* (*fun-ord Q*)
(λf. (*ocondition C* (*X f*) (*Y f*)))
using *mono-X mono-Y* **unfolding** *ocondition-def monotone-def fun-ord-def*
by *auto*

declare *Complete-Partial-Order2.option.preorder* [*partial-function-mono*]

lemma *monotone-obind*[*partial-function-mono*]:
monotone R option.le-fun a $\Longrightarrow$ ($\bigwedge x$. *monotone R option.le-fun* (λf. *b f x*)) $\Longrightarrow$
monotone R option.le-fun (λf. *obind* (*a f*) (*b f*))
unfolding *monotone-def obind-def*
apply (*clarsimp simp add*: *flat-ord-def fun-ord-def split*: *option.splits*)
by (*metis option.sel option.simps*(*3*))

lemma *monotone-option-fun-const* [*partial-function-mono*]:
monotone R option.le-fun (λf. *c*)
by (*auto simp add*: *monotone-def flat-ord-def fun-ord-def*)

lemma *option-while-eq-Some*:
option-while C B I = *Some F* $\longleftrightarrow$ (*Some I*, *Some F*) $\in$ *option-while′ C B*
using *option-while′-THE* **by** (*force simp*: *option-while-def*)

lemma *option-while′-monotone*:
assumes *B*: $\bigwedge r$. *flat-ord None* (*B r*) (*B′ r*)
assumes *b*: (*a*, *b*) $\in$ *option-while′ C B b* $\neq$ *None* **shows** (*a*, *b*) $\in$ *option-while′ C B′*
using *b*
proof *induction*
case (*step r1 r2 s*) **then show** *?case*
by (*metis B flat-ord-def option.simps*(*2*) *option-while′.intros*(*3*))
qed (*auto intro*: *option-while′.intros*)
lemma *monotone-option-while*[*partial-function-mono*]:
assumes *B*: $\bigwedge a$. *monotone R* (*flat-ord None*) (λf. *B f a*)
shows *monotone R* (*flat-ord None*) (λf. *option-while C* (*B f*) *I*)
proof
fix *x y* **assume** *R x y*
show *option-ord* (*option-while C* (*B x*) *I*) (*option-while C* (*B y*) *I*)
unfolding *flat-ord-def*
proof (*intro disjCI2*)
assume *option-while C* (*B x*) *I* $\neq$ *None*

```
    then obtain F where option-while C (B x) I = Some F by auto
    then have x: (Some I, Some F) ∈ option-while' C (B x)
      by (auto simp: option-while-eq-Some)
    have (Some I, Some F) ∈ option-while' C (B y)
       using B ‹R x y› by (intro option-while'-monotone[OF - x]) (auto simp:
monotone-def)
    with x show option-while C (B x) I = option-while C (B y) I
      unfolding option-while-eq-Some[symmetric] by simp
  qed
qed

lemma monotone-owhile[partial-function-mono]:
  (⋀a. monotone R option.le-fun (λf. B f a)) ⟹
    monotone R option.le-fun (λf. owhile C (B f) I)
  unfolding owhile-def monotone-fun-ord-apply
  by (intro allI monotone-option-while) simp


setup ‹
let open Mutual-CCPO-Rec in
  add-ccpo option-state-monad (fn ctxt => fn T =>
    let
      val oT = range-type T
    in
      synth-fun ctxt (domain-type T) (synth-option ctxt oT)
    end)
  |> Context.theory-map
end
›

lemma refines-lift-pure-option:
  assumes f: refines f (return f') s s (rel-prod rel-liftE (=))
  shows refines f (gets-the (oreturn f')) s s (rel-prod rel-liftE (=))
  using f
  apply (auto simp add: refines-def-old)
  done



setup ‹
Monad-Types.new-monad-type
  option
  Option monad
  option-state-monad
  60
  (fn {stateT, resT, exT} =>
     stateT --> Term.map-atyps (fn T => if T = @{typ 'a} then resT else T)
@{typ 'a option})
  (fn ctxt => fn - => let fun lift t = infer-instantiate ‹t = t in term ‹ogets t››
```

```
ctxt
                    in Utils.lift-result-with-arity 1 lift end)
  {rules-name = @{synthesize-rules-name option},
   relator = @{term rel-liftE::('a, 'b) xval ⇒ 'b val ⇒ bool},
    relator-from-c-exntype = NONE, lift = @{term gets-the},
    dest-lift = (fn @{term-pat gets-the ?x} => SOME x | - => NONE),
   lift-prev = @{thms refines-lift-pure-option refines-lift-reader-option}}
|> Context.theory-map
›
```

```
setup ‹
Monad-Types.new-monad-type
  nondet
  Nondeterministic state monad
  spec-monad-gfp
  20
  (fn {stateT, resT, exT} =>
     Term.map-atyps (fn T => if T = @{typ 'a} then resT
                             else if T = @{typ 's} then stateT else T)
       @{typ ('a, 's) res-monad})
  (fn ctxt => fn - => let fun lift t = infer-instantiate ‹t = t in term ‹gets-the
t::('a, 's) res-monad›› ctxt in Utils.lift-result-with-arity 1 lift end)
  {rules-name = @{synthesize-rules-name nondet},
   relator = @{term rel-liftE::('a, 'b) xval ⇒ 'b val ⇒ bool},
   relator-from-c-exntype = NONE, lift = @{term ‹λx. x›},
   dest-lift = (fn - => NONE),
   lift-prev = []}
|> Context.theory-map
›

setup ‹
Monad-Types.new-monad-type
  exit
  Nondeterministic state monad with exit (default)
  spec-monad-gfp
  10
  (fn {stateT, resT, exT} =>
     Term.map-atyps (fn T => if T = @{typ 'a} then resT
                             else if T = @{typ 's} then stateT
                             else if T = @{typ 'e} then HP-TermsTypes.strip-c-exntype
exT
                             else T)
       @{typ ('e, 'a, 's) exn-monad})
  (fn ctxt => fn T => let fun lift t = infer-instantiate ‹'e=dummyT and t =
t in term ‹liftE t:: ('e, 'a, 's) exn-monad›› ctxt in Utils.lift-result-with-arity 0 lift
```

```
end)
  {rules-name = @{synthesize-rules-name exit},
    relator = @{term ‹rel-xval (=) (=)›},
     relator-from-c-exntype = SOME @{term ‹rel-xval rel-Nonlocal (=)›}, lift =
@{term ‹λx. x›},
    dest-lift = (fn - => NONE),
   lift-prev = []}
|> Context.theory-map
›


lemma id-comps:
  id o f = f
  ((λs. s) o f) = f
  by (simp-all add: comp-def)

lemma gets-bind-ign: gets f >>= (λx. m) = m
  apply (rule spec-monad-eqI)
  apply (auto simp add: runs-to-iff)
  done

end
```

Chapter 24

Polishing the Final Outcome

theory *Polish*
imports *HeapLift WordPolish TypeStrengthen*
begin

context *heap-typing-state*
begin

lemma *unchanged-typing-bind*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s* |} ($\bigwedge r\ s.\ g\ r \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s* |})
  shows (*bind f g*) $\cdot s$?{| $\lambda r.$ *unchanged-typing-on S s* |}
  apply (*runs-to-vcg*)
  using *unchanged-typing-on-trans* **by** *blast*

lemma *unchanged-typing-while*[*unchanged-typing*]:
    assumes *B*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow (B\ r) \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
    shows (*whileLoop C B i*) $\cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  apply (*rule runs-to-partial-whileLoop* [**where** $I = \lambda$-. *unchanged-typing-on S s*])
  subgoal by *simp*
  subgoal by *simp*
  subgoal by *simp*
  subgoal
    apply (*rule runs-to-partial-weaken* [*OF B*])
     apply (*simp*)
    by (*rule unchanged-typing-on-trans*)
  done

lemma *unchanged-typing-finally*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  shows *finally* $f \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  by (*runs-to-vcg*)

lemma *unchanged-typing-try*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}

shows *try f · s ?{| λr. unchanged-typing-on S s|}*
by (*runs-to-vcg*)

lemma *runs-to-partial-catch*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: *f · s ?{|λr. unchanged-typing-on S s|}*
assumes [*runs-to-vcg*]: *⋀r s. (g r) · s ?{|λr. unchanged-typing-on S s|}*
shows *(catch f g) · s ?{|λr. unchanged-typing-on S s|}*
by (*runs-to-vcg*) (*auto simp add*: *unchanged-typing-on-simps*)

lemma *runs-to-partial-bind-handle*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: *f · s ?{|λr. unchanged-typing-on S s|}*
assumes [*runs-to-vcg*]: *⋀r s. (g r) · s ?{|λr. unchanged-typing-on S s|}*
assumes [*runs-to-vcg*]: *⋀r s. (h r) · s ?{|λr. unchanged-typing-on S s|}*
shows *(bind-handle f g h) · s ?{|λr. unchanged-typing-on S s|}*
by (*runs-to-vcg*) (*auto simp add*: *unchanged-typing-on-simps*)

lemma *unchanged-typing-liftE*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: *f · s ?{| λr. unchanged-typing-on S s|}*
shows *liftE f · s ?{| λr. unchanged-typing-on S s|}*
by (*runs-to-vcg*)

end

context *open-types-heap-typing-state*
begin

lemma *unchanged-typing-ptr-valid-preserved*:
ptr-valid (heap-typing t1) p ⟹ unchanged-typing-on UNIV t1 t2 ⟹ ptr-valid (heap-typing t2) p
by (*simp add*: *unchanged-typing-on-UNIV-iff*)

lemma *reaches-unchanged-typing-ptr-valid-preserved*:
reaches f s r t ⟹ ptr-valid (heap-typing s) p ⟹ f · s ?{|λ- t. unchanged-typing-on UNIV s t|} ⟹
ptr-valid (heap-typing t) p
using *unchanged-typing-ptr-valid-preserved*
by (*simp add*: *runs-to-partial-def-old reaches-succeeds unchanged-typing-on-UNIV-iff*)
thm *unchanged-typing* [*no-vars*]
end

locale *typ-heap-simulation-open-types-stack* =
typ-heap-simulation-open-types 𝒯 st r w v t-hrs t-hrs-update heap-typing heap-typing-upd
for
𝒯 **and**
st:: *'s ⇒ 't* **and**
r:: *'t ⇒ ('a*::{*xmem-type*, *stack-type*}*) ptr ⇒ 'a* **and**
w:: *'a ptr ⇒ ('a ⇒ 'a) ⇒ 't ⇒ 't* **and**

```
    v:: 't ⇒ 'a ptr ⇒ bool and
    t-hrs :: 's ⇒ heap-raw-state and
    t-hrs-update:: (heap-raw-state ⇒ heap-raw-state) ⇒ 's ⇒ 's and
    heap-typing :: 't ⇒ heap-typ-desc and
    heap-typing-upd :: (heap-typ-desc ⇒ heap-typ-desc) ⇒ 't ⇒ 't +
  fixes S:: addr set
begin
  sublocale typ-heap-typing r w heap-typing heap-typing-upd S
    by intro-locales

lemma unchanged-typing-on-with-fresh-stack-ptr[unchanged-typing]:
  fixes f::'a ptr ⇒ ('e::default, 'b, 't) spec-monad
  assumes [runs-to-vcg]:⋀p t. (f p) · t ?{λ- t'. unchanged-typing-on A t t'}
  shows (with-fresh-stack-ptr n init f) · t ?{λ- t'. unchanged-typing-on A t t'}
 unfolding with-fresh-stack-ptr-def on-exit-def stack-ptr-acquire-def assume-stack-alloc-def
  apply runs-to-vcg
  subgoal for x d vs t'
     apply (subst (asm) typing-eq-left-unchanged-typing-on [of - (heap-typing-upd
(λ-. d) t)])
     apply (simp add: heap-typing-fold-upd-write)
   apply (simp add: unchanged-typing-on-def stack-ptr-release-def heap-typing-fold-upd-write
stack-allocs-releases-equal-on-typing)
    done
  done

lemma unchanged-typing-on-assume-with-fresh-stack-ptr[unchanged-typing]:
  fixes f::'a ptr ⇒ ('e::default, 'b, 't) spec-monad
  assumes [runs-to-vcg]:⋀p t. (f p) · t ?{λ- t'. unchanged-typing-on A t t'}
  shows (assume-with-fresh-stack-ptr n init f) · t ?{λ- t'. unchanged-typing-on A
t t'}
 unfolding assume-with-fresh-stack-ptr-def stack-ptr-acquire-def assume-stack-alloc-def
  apply runs-to-vcg
  subgoal for x d vs t'
     apply (subst (asm) typing-eq-left-unchanged-typing-on [of - (heap-typing-upd
(λ-. d) t)])
     apply (simp add: heap-typing-fold-upd-write)
   apply (simp add: unchanged-typing-on-def stack-ptr-release-def heap-typing-fold-upd-write
stack-allocs-releases-equal-on-typing)
    done
  done

lemma unchanged-typing-on-guard-with-fresh-stack-ptr[unchanged-typing]:
  fixes f::'a ptr ⇒ ('e::default, 'b, 't) spec-monad
  assumes [runs-to-vcg]:⋀p t. (f p) · t ?{λ- t'. unchanged-typing-on A t t'}
  shows (guard-with-fresh-stack-ptr n init f) · t ?{λ- t'. unchanged-typing-on A t
t'}
 unfolding guard-with-fresh-stack-ptr-def stack-ptr-acquire-def assume-stack-alloc-def
  apply runs-to-vcg
  subgoal for x d vs t'
```

apply (*subst* (*asm*) *typing-eq-left-unchanged-typing-on* [*of* - (*heap-typing-upd* (λ-. *d*) *t*)])
apply (*simp add*: *heap-typing-fold-upd-write*)
apply (*simp add*: *unchanged-typing-on-def stack-ptr-release-def heap-typing-fold-upd-write stack-allocs-releases-equal-on-typing*)
done
done

end

named-theorems *polish*
named-theorems *polish-cong* **and** *state-fold-congs*

declare *map-value-id*[*polish*]

lemmas [*polish*] = *mex-def meq-def*

declare *WORD-values-fold* [*polish*]

lemmas *WORD-bound-simps* [*polish*] =
WORD-MAX-simps
WORD-MIN-simps
UWORD-MAX-simps
WORD-signed-to-unsigned
INT-MIN-MAX-lemmas

declare *singleton-iff* [*polish*]
declare *the-Nonlocal.simps* [*polish*]
declare *map-exn-id* [*polish*]
declare *prod.case* [*polish*]

declare *mem-Collect-eq* [*polish*]

lemma *L2-VARS-polish* [*polish*]: *L2-VARS x ns* = *x*
by (*simp add*: *L2-VARS-def*)

lemmas *liftE-atomic* [*polish*] =
liftE-unknown
liftE-top
liftE-bot
liftE-fail
liftE-yield-Exception
liftE-return
liftE-throw-exception-or-result
liftE-get-state

liftE-set-state
liftE-select
liftE-assert
liftE-assume
liftE-gets
liftE-guard
liftE-assert-opt
liftE-gets-the
liftE-modify

lemma *map-value-map-exn-id*[*simp*, *polish*]:
map-value (*map-exn* (λ*e'*. *e'*)) *f* = *f*
by (*rule spec-monad-ext*) (*simp add*: *run-map-value*)

lemma *gets-to-return* [*polish*]:
gets (λ*x*. *a*) = *return a*
by (*rule spec-monad-ext*) *simp*

lemma *select-UNIV-unknown* [*polish*]: *select UNIV* = *unknown*
by (*clarsimp simp*: *unknown-def*)

lemma *unknown-unit* [*polish*, *simp*]: (*unknown* :: (*unit*,'*s*) *res-monad*) = *skip*
apply (*rule spec-monad-ext*)
apply (*auto*)
done

lemma *condition-to-when* [*polish*]:
condition (λ*s*. *C*) *A skip* = *when C A*
by (*simp add*: *when-def*)

lemma *condition-to-unless* [*polish*]:
condition (λ*s*. *C*) *skip A* = *unless C A*
apply (*simp add*: *when-def*)
apply (*rule condition-swap*)
done

lemma *bind-skip* [*simp*, *polish*]:
(*x* >>= (λ-. *skip*)) = *x*
by *simp*

lemma *skip-bind* [*simp*, *polish*]:
(*skip* >>= *P*) = (*P* ())
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)
done

lemma *catch-skip*[*simp*, *polish*]: *catch skip f* = *skip*
by (*rule spec-monad-ext*) (*simp add*: *run-catch*)

lemma *ogets-to-oreturn* [*polish*]: *ogets* (λ*s*. *P*) = *oreturn P*
apply (*clarsimp simp*: *ogets-def oreturn-def K-def*)
done

lemma *ocondition-ret-ret* [*polish*]:
ocondition P (*oreturn A*) (*oreturn B*) = *ogets* (λ*s*. *if P s then A else B*)
by (*auto simp*: *ocondition-def ogets-def*)

lemma *ocondition-gets-ret* [*polish*]:
ocondition P (*ogets A*) (*oreturn B*) = *ogets* (λ*s*. *if P s then A s else B*)
by (*auto simp*: *ocondition-def ogets-def*)

lemma *ocondition-ret-gets* [*polish*]:
ocondition P (*oreturn A*) (*ogets B*) = *ogets* (λ*s*. *if P s then A else B s*)
by (*auto simp*: *ocondition-def ogets-def*)

lemma *ocondition-gets-gets* [*polish*]:
ocondition P (*ogets A*) (*ogets B*) = *ogets* (λ*s*. *if P s then A s else B s*)
by (*auto simp*: *ocondition-def ogets-def*)

lemma *case-prod-trivial*[]: *NO-MATCH* (*g*::(*'e*::*default*, *'a*, *'s*) *spec-monad*) *f* ⟹ (λ(*x*,*y*). *f*) = (λ-. *f*)
— We avoid this rule during polish to keep brittle tuple structure (*case-prod*) and bound variable names in eg. (>>=). With the *NO-MATCH* setup the simplification might still trigger e.g. on conditions of while loops
by *auto*

lemma *bind-case-prod-trivial*[*polish*]: *bind f* (λ(*x*, *y*). *g*) = *bind f* (λ-. *g*)
by (*rule spec-monad-ext*) (*simp add*: *run-bind*)

lemma *condition-to-if* [*polish*]:
condition (λ*s*. *C*) (*return a*) (*return b*) = *return* (*if C then a else b*)
by (*rule spec-monad-ext*) (*simp add*: *run-condition*)

lemma *guard-merge-bind*:
guard P >>= (λ-. *guard Q* >>= *M*) = *guard* (*P and Q*) >>= *M*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *guard-merge-bind′*:
guard P >>= (λ-. *guard Q* >>= *M*) = *guard* (λ*s*. *P s* ∧ *Q s*) >>= *M*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *guard-merge*:

guard P >>= (λ-. guard Q) = guard (P and Q)
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *guard-merge′*:
guard P >>= (λ-. guard Q) = guard (λs. P s ∧ Q s)
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

lemma *pred-conj-commute*: (*P and Q*) = (*Q and P*)
by (*auto simp add*: *pred-conj-def*)

lemma *guard-True-skip*[*polish*, *simp*]: *guard* (λ-. *True*) = *skip*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-guard*)
done

lemma *guard-True-bind*: — subsumed by *skip >>= ?P = ?P* () and *guard* (λ-. *True*) = *skip*
(*guard* (λ-. *True*) >>= *M*) = *M* ()
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard*)
done

declare *assert-simps*(*1*) [*polish*]
declare *assume-simps*(*1*) [*polish*]

lemma *simple-bind-fail* [*simp*]:
(*guard P >>= (λ-. fail*)) = *fail*
(*modify f >>= (λ-. fail*)) = *fail*
(*return x >>= (λ-. fail*)) = *fail*
(*gets f >>= (λ-. fail*)) = *fail*
apply (*rule spec-monad-ext*, *simp add*: *run-bind run-guard*)+
done

declare *condition-fail-rhs* [*polish*]
declare *condition-fail-lhs* [*polish*]
declare *simple-bind-fail* [*polish*]
declare *condition-bind-fail* [*polish*]

lemma *whileLoop-fail*:
(*whileLoop C* (λ-. *fail*) *i*) = (*guard* (λs. ¬ *C i s*) >>= (λ-. *return i*))
apply (*subst whileLoop-unroll*)
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-condition run-bind run-guard*)
done

lemma *owhile-fail*:
(*owhile C* (λ-. *ofail*) *i*) = (*oguard* (λ*s*. ¬ *C i s*) |>> (λ-. *oreturn i*))
apply (*rule ext*)
apply (*subst owhile-unroll*)
apply (*clarsimp simp*: *obind-def oguard-def ocondition-def split*: *option.splits*)
done

declare *whileLoop-fail* [*polish*]
declare *owhile-fail* [*polish*]

lemma *oguard-True* [*simp*, *polish*]: *oguard* (λ-. *True*) = *oreturn* ()
by (*clarsimp simp*: *oreturn-def oguard-def K-def*)

lemma *oguard-False* [*simp*, *polish*]: *oguard* (λ-. *False*) = *ofail*
by (*clarsimp simp*: *ofail-def oguard-def K-def*)

declare *oreturn-bind* [*polish*]
declare *obind-return* [*polish*]
lemma *infinite-option-while'*: (*Some x*, *Some y*) ∉ *option-while'* (λ-. *True*) *B*
apply (*rule notI*)
apply (*induct Some x* :: *'a option Some y* :: *'a option*
arbitrary: *x y rule*: *option-while'.induct*)
apply *auto*
done

lemma *expand-guard-conj* [*polish*]:
guard (λ*s*. *A s* ∧ *B s*) = (*do* {*guard* (λ*s*. *A s*); *guard* (λ*s*. *B s*) })
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-guard run-bind*)
done

lemma *oguard-K-bind-cong* [*polish-cong*]: *g* = *g'* ⟹ (⋀*s*. *g' s* ⟹ *c s* = *c' s*)
⟹ (*oguard g* >>= (λ-. *c*)) = (*oguard g'* >>= (λ-. *c'*))
apply (*rule ext*)
apply (*auto simp add*: *oguard-def obind-def*)
done

lemma *oguard-obind-cong*: *g* = *g'* ⟹ (⋀*s*. *g' s* ⟹ *c s* = *c' s*) ⟹
do {- <− *oguard g* ; *c*} = *do* {- <− *oguard g'*; *c'*}
by (*auto simp add*: *oguard-def obind-def*)

lemma *oguard-True-K-bind* [*polish*]: (*oguard* (λ-. *True*) >>= (λ-. *c*)) = *c*
apply (*rule ext*)
apply (*auto simp add*: *oguard-def obind-def*)
done

lemma *oguard-False-bind* [*polish*]: (*oguard* (λ-. *False*) >>= *c*) = *ofail*
apply (*rule ext*)
apply (*auto simp add*: *oguard-def obind-def*)
done

lemma *guard-False-bind* [*polish*]: (*guard* (λ-. *False*) >>= *c*) = *fail*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-guard run-bind*)
done

lemma *expand-oguard-conj* [*polish*]:
oguard (λ*s*. *A s* ∧ *B s*) = (*obind* (*oguard* (λ*s*. *A s*)) (λ-. *oguard* (λ*s*. *B s*)))
by (*rule ext*) (*clarsimp simp*: *oguard-def obind-def split*: *option.splits*)

lemma *owhile-infinite-loop* [*simp*, *polish*]:
owhile (λ*r s*. *C*) *B x* = (*oguard* (λ*s*. ¬ *C*) |>> (λ-. *oreturn x*))
apply (*cases C*)
apply (*rule ext*)
apply (*clarsimp simp*: *owhile-def option-while-def obind-def split*: *option.splits*)
apply (*metis infinite-option-while′ None-not-eq option-while′-THE*)
apply (*subst owhile-unroll*)
apply (*clarsimp simp*: *obind-def oreturn-def K-def split*: *option.splits*)
done

declare *obind-return* [*polish*]
declare *bind-return* [*polish*]

lemma *fail-bind* [*simp*]:
fail >>= *f* = *fail*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)
done

declare *fail-bind* [*polish*]

declare *ofail-bind* [*polish*]
declare *obind-fail* [*polish*]

declare *singleton-iff* [*polish*]
declare *when-True* [*polish*]
declare *when-False* [*polish*]

lemma *let-triv* [*polish*]: (*let x* = *y in x*) = *y*
apply (*clarsimp simp*: *Let-def*)
done

lemma *ucast-scast-same* [*polish*, *L2opt*, *simp*]:
ucast ((*scast x* :: (′*a*::*len*) *word*)) = (*x* :: ′*a word*)

apply (*clarsimp simp*: *ucast-def scast-def*)
done

lemma *word-of-int-of-nat* [*polish*, *L2opt*, *simp*]:
word-of-int (*int x*) = *of-nat x*
by (*rule of-int-of-nat-eq*)

lemma *return-if-P-1-0-bind* [*polish*]:
(*return* (*if P then 1 else 0*) >>= (λ*x*. *Q x*))
= (*return P* >>= (λ*x*. *Q* (*if x then 1 else 0*)))
apply *simp*
done

lemma *gets-if-P-1-0-bind* [*polish*]:
(*gets* (λ*s*. *if P s then 1 else 0*) >>= (λ*x*. *Q x*))
= (*gets P* >>= (λ*x*. *Q* (*if x then 1 else 0*)))
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind*)
done

lemma *if-P-1-0-neq-0* [*polish*, *simp*]:
((*if P then 1 else* (*0*::('*a*::{*zero-neq-one*}))) ≠ *0*) = *P*
apply *simp*
done

lemma *if-P-1-0-eq-0* [*polish*, *simp*]:
((*if P then 1 else* (*0*::('*a*::{*zero-neq-one*}))) = *0*) = (¬ *P*)
apply *simp*
done

lemma *if-if-same-output* [*polish*]:
(*if a then if b then x else y else y*) = (*if a* ∧ *b then x else y*)
(*if a then x else if b then x else y*) = (*if a* ∨ *b then x else y*)
by *auto*

lemma *collect-guarded-conj*[*polish*]:
condition C1 (*do* { *guard G*; *gets* (λ*s*. *if C2 s then 1 else 0*) })
(*return 0*)
= *do* { *guard* (λ*s*. *C1 s* ⟶ *G s*);
gets (λ*s*. *if C1 s* ∧ *C2 s then 1 else 0*) }
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard run-condition*)
done

lemma *collect-guarded-disj*[*polish*]:
condition C1 (*return 1*)
(*do* {*guard G*; *gets* (λ*s*. *if C2 s then 1 else 0*) })
= *do* {*guard* (λ*s*. ¬ *C1 s* ⟶ *G s*);
gets (λ*s*. *if C1 s* ∨ *C2 s then 1 else 0*) }
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard run-condition*)
done

lemmas [*polish*] = *bind-assoc obind-assoc*

declare *not-not* [*polish*]

lemma *collect-then-cond-1-0*[*polish*]:
do {*cond* ← *gets* (λ*s*. *if P s then* (*1*::('*a*::{*zero-neq-one*})) *else 0*);
condition (λ-. *cond* ≠ *0*) *L R* }
= *condition P L R*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard run-condition*)
done

lemma *collect-then-cond-1-0-assoc*[*polish*]:
(*do* {*cond* ← *gets* (λ*s*. *if P s then* (*1*::('*a*::{*zero-neq-one*})) *else 0*);
condition (λ-. *cond* ≠ *0*) *L R*
>>= *f* })
= (*condition P L R* >>= *f*)
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-guard run-condition*)
done

lemma *bind-return-bind* [*polish*]:
(*A* >>= (λ*x*. (*return x* >>= (λ*y*. *B x y*)))) = (*A* >>= (λ*x*. *B x x*))
by *simp*

lemma *obind-oreturn-obind* [*polish*]:
(*A* |>> (λ*x*. (*oreturn x* |>> (λ*y*. *B x y*)))) = (*A* |>> (λ*x*. *B x x*))
by *simp*

declare *obind-assoc* [*polish*]

declare *if-distrib* [**where** *f*=*scast*, *polish*, *simp*]
declare *if-distrib* [**where** *f*=*ucast*, *polish*, *simp*]
declare *if-distrib* [**where** *f*=*unat*, *polish*, *simp*]

declare *if-distrib* [**where** *f*=*uint*, *polish*, *simp*]
declare *if-distrib* [**where** *f*=*sint*, *polish*, *simp*]

declare *cast-simps* [*polish*]

lemma *Suc-0-eq-1* [*polish*]: *Suc 0 = 1*
by *simp*

lemma *bind-return-case-prod* [*polish*, *simp*]:
(*do* {() ← *A0*; *return* () }) = *A0*
(*do* {(*a*) ← *A1*; *return* (*a*) }) = *A1*
(*do* {(*a*, *b*) ← *A2*; *return* (*a*, *b*) }) = *A2*
(*do* {(*a*, *b*, *c*) ← *A3*; *return* (*a*, *b*, *c*) }) = *A3*
(*do* {(*a*, *b*, *c*, *d*) ← *A4*; *return* (*a*, *b*, *c*, *d*) }) = *A4*
(*do* {(*a*, *b*, *c*, *d*, *e*) ← *A5*; *return* (*a*, *b*, *c*, *d*, *e*) }) = *A5*
(*do* {(*a*, *b*, *c*, *d*, *e*, *f*) ← *A6*; *return* (*a*, *b*, *c*, *d*, *e*, *f*) }) = *A6*
by (*auto simp*: *split-beta′ case-unit-Unity*[*abs-def*])

lemma *bind-return-case-prod′* [*polish*, *simp*]:
(*A1* ⤜ *return* ⤜ *g1*) = (*A1* ⤜ *g1*)
(*A2* ⤜ (λ(*a*, *b*). (*return* (*a*, *b*) ⤜ *g2*))) = (*A2* ⤜ *g2*)
(*A3* ⤜ (λ(*a*, *b*, *c*). (*return* (*a*, *b*, *c*) ⤜ *g3*))) = (*A3* ⤜ *g3*)
(*A4* ⤜ (λ(*a*, *b*, *c*, *d*). (*return* (*a*, *b*, *c*, *d*) ⤜ *g4*))) = (*A4* ⤜ *g4*)
(*A5* ⤜ (λ(*a*, *b*, *c*, *d*, *e*). (*return* (*a*, *b*, *c*, *d*, *e*) ⤜ *g5*))) = (*A5* ⤜ *g5*)
(*A6* ⤜ (λ(*a*, *b*, *c*, *d*, *e*, *f*). (*return* (*a*, *b*, *c*, *d*, *e*, *f*) ⤜ *g6*))) = (*A6* ⤜ *g6*)
by (*auto simp*: *split-beta′ case-unit-Unity*[*abs-def*])

lemma *obind-returnOk-prodE-case* [*polish*, *simp*]:
(*A1* |>> (λ*a*. *oreturn* (*a*))) = *A1*
(*A2* |>> (λ(*a*, *b*). *oreturn* (*a*, *b*))) = *A2*
(*A3* |>> (λ(*a*, *b*, *c*). *oreturn* (*a*, *b*, *c*))) = *A3*
(*A4* |>> (λ(*a*, *b*, *c*, *d*). *oreturn* (*a*, *b*, *c*, *d*))) = *A4*
(*A5* |>> (λ(*a*, *b*, *c*, *d*, *e*). *oreturn* (*a*, *b*, *c*, *d*, *e*))) = *A5*
(*A6* |>> (λ(*a*, *b*, *c*, *d*, *e*, *f*). *oreturn* (*a*, *b*, *c*, *d*, *e*, *f*))) = *A6*
by *auto*

lemma *obind-returnOk-prodE-case′* [*polish*, *simp*]:
(*A1* |>> (λ*a*. (*oreturn* (*a*) |>> *g1*))) = *A1* |>> *g1*
(*A2* |>> (λ(*a*, *b*). (*oreturn* (*a*, *b*) |>> *g2*))) = *A2* |>> *g2*
(*A3* |>> (λ(*a*, *b*, *c*). (*oreturn* (*a*, *b*, *c*) |>> *g3*))) = *A3* |>> *g3*
(*A4* |>> (λ(*a*, *b*, *c*, *d*). (*oreturn* (*a*, *b*, *c*, *d*) |>> *g4*))) = *A4* |>> *g4*
(*A5* |>> (λ(*a*, *b*, *c*, *d*, *e*). (*oreturn* (*a*, *b*, *c*, *d*, *e*) |>> *g5*))) = *A5* |>> *g5*
(*A6* |>> (λ(*a*, *b*, *c*, *d*, *e*, *f*). (*oreturn* (*a*, *b*, *c*, *d*, *e*, *f*) |>> *g6*))) = *A6* |>> *g6*
by *auto*

lemma *bind-fixup-1*:
(*bind A* (λ*x*. *case y of* (*a*, *b*) ⇒ *B a b x*)) =

(*case y of* (*a*, *b*) ⇒ *bind A* (λ*x*. *B a b x*))
by (*auto simp*: *split-def*)
lemma *bind-fixup-2*:
(*bind* (*case y of* (*a*, *b*) ⇒ *B a b*) *A*) =
(*case y of* (*a*, *b*) ⇒ *bind* (*B a b*) *A*)
by (*auto simp*: *split-def*)
lemma *finally-fixup*: (λ*x*. *finally* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *finally* (*f a b*))
by (*simp add*: *fun-eq-iff split*: *prod.splits*)
lemma *try-fixup*: (λ*x*. *try* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *try* (*f a b*))
by (*simp add*: *fun-eq-iff split*: *prod.splits*)
lemma *liftE-fixup*: (λ*x*. *liftE* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *liftE* (*f a b*))
by (*simp add*: *fun-eq-iff split*: *prod.splits*)

lemmas *spec-monad-split-fixup* [*polish*]= — should we even take more from *L2-seq*
?A (λ*x*. *case ?y of* (*a*, *b*) ⇒ *?B a b x*) = (*case ?y of* (*a*, *b*) ⇒ *L2-seq ?A* (*?B a b*))
L2-seq (*case ?y of* (*a*, *b*) ⇒ *?B a b*) *?A* = (*case ?y of* (*a*, *b*) ⇒ *L2-seq* (*?B a b*) *?A*)
(*case* (*?x*, *?y*) *of* (*a*, *b*) ⇒ *?P a b*) = *?P ?x ?y*
(λ(*a*, *b*). *?P a*) = (λ(*a*, *x*, *y*). *?P a*)
(λ*x*. *liftE* (*case x of* (*a*, *b*) ⇒ *?f a b*)) = (λ(*a*, *b*). *liftE* (*?f a b*))
(λ*x*. *finally* (*case x of* (*a*, *b*) ⇒ *?f a b*)) = (λ(*a*, *b*). *finally* (*?f a b*))
(λ*x*. *try* (*case x of* (*a*, *b*) ⇒ *?f a b*)) = (λ(*a*, *b*). *try* (*?f a b*))
L2-guard (λ*s*. *case ?y of* (*a*, *b*) ⇒ *?G a b s*) = (*case ?y of* (*a*, *b*) ⇒ *L2-guard* (*?G a b*))
L2-guard (*case ?y of* (*a*, *b*) ⇒ *?G a b*) = (*case ?y of* (*a*, *b*) ⇒ *L2-guard* (*?G a b*))
L2-gets (*case ?y of* (*a*, *b*) ⇒ *?G a b*) = (*case ?y of* (*a*, *b*) ⇒ *L2-gets* (*?G a b*))
L2-modify (*case ?y of* (*a*, *b*) ⇒ *?G a b*) = (*case ?y of* (*a*, *b*) ⇒ *L2-modify* (*?G a b*))
L2-gets (λ-. *case ?y of* (*a*, *b*) ⇒ *?G a b*) = (*case ?y of* (*a*, *b*) ⇒ *L2-gets* (λ-. *?G a b*))
(λ(*a*, *b*). *L2-gets* (*?P a b*) *?n*) = (λ(*a*, *x*, *y*). *L2-gets* (*?P a* (*x*, *y*)) *?n*)
(λ(*a*, *b*). *L2-guard* (*?P a b*)) = (λ(*a*, *x*, *y*). *L2-guard* (*?P a* (*x*, *y*)))
(λ(*a*, *b*). *L2-modify* (*?P a b*)) = (λ(*a*, *x*, *y*). *L2-modify* (*?P a* (*x*, *y*)))
(λ(*a*, *b*). *L2-spec* (*?P a b*)) = (λ(*a*, *x*, *y*). *L2-spec* (*?P a* (*x*, *y*)))
(λ(*a*, *b*). *L2-assume* (*?P a b*)) = (λ(*a*, *x*, *y*). *L2-assume* (*?P a* (*x*, *y*)))
(λ(*a*, *b*). *L2-throw* (*?P a b*) *?n*) = (λ(*a*, *x*, *y*). *L2-throw* (*?P a* (*x*, *y*)) *?n*)
(λ(*a*, *b*). *L2-seq* (*?L a b*) (*?R a b*)) = (λ(*a*, *x*, *y*). *L2-seq* (*?L a* (*x*, *y*)) (*?R a* (*x*, *y*)))
(λ(*a*, *b*). *L2-while* (*?C a b*) (*?B a b*) (*?I a b*) *?n*) = (λ(*a*, *x*, *y*). *L2-while* (*?C a* (*x*, *y*)) (*?B a* (*x*, *y*)) (*?I a* (*x*, *y*)) *?n*)
(λ(*a*, *b*). *L2-unknown ?n*) = (λ(*a*, *x*, *y*). *L2-unknown ?n*)
(λ(*a*, *b*). *L2-catch* (*?L a b*) (*?R a b*)) = (λ(*a*, *x*, *y*). *L2-catch* (*?L a* (*x*, *y*)) (*?R a* (*x*, *y*)))
(λ(*a*, *b*). *L2-condition* (*?C a b*) (*?L a b*) (*?R a b*)) = (λ(*a*, *x*, *y*). *L2-condition* (*?C a* (*x*, *y*)) (*?L a* (*x*, *y*)) (*?R a* (*x*, *y*)))
(λ(*a*, *b*). *L2-call* (*?M a b*)) = (λ(*a*, *x*, *y*). *L2-call* (*?M a* (*x*, *y*)))

$(\lambda(a,\ b).\ liftE\ (?M\ a\ b)) = (\lambda(a,\ x,\ y).\ liftE\ (?M\ a\ (x,\ y)))$
$(\lambda(a,\ b).\ finally\ (?M\ a\ b)) = (\lambda(a,\ x,\ y).\ finally\ (?M\ a\ (x,\ y)))$
$(\lambda(a,\ b).\ try\ (?M\ a\ b)) = (\lambda(a,\ x,\ y).\ try\ (?M\ a\ (x,\ y)))$?
bind-fixup-1
bind-fixup-2
finally-fixup
try-fixup
liftE-fixup

lemma *scast-1-simps* [*simp*, *L2opt*, *polish*]:
scast (*1* :: (*'a*::*len*) *bit1 word*) = *1*
scast (*1* :: (*'a*::*len*) *bit0 word*) = *1*
scast (*1* :: (*'a*::*len*) *bit1 signed word*) = *1*
scast (*1* :: (*'a*::*len*) *bit0 signed word*) = *1*
by *auto*

lemma *scast-1-simps-direct* [*simp*, *L2opt*, *polish*]:
scast (*1* :: *sword64*) = (*1* :: *word64*)
scast (*1* :: *sword64*) = (*1* :: *word32*)
scast (*1* :: *sword64*) = (*1* :: *word16*)
scast (*1* :: *sword64*) = (*1* :: *word8*)
scast (*1* :: *sword32*) = (*1* :: *word32*)
scast (*1* :: *sword32*) = (*1* :: *word16*)
scast (*1* :: *sword32*) = (*1* :: *word8*)
scast (*1* :: *sword16*) = (*1* :: *word16*)
scast (*1* :: *sword16*) = (*1* :: *word8*)
scast (*1* :: *sword8*) = (*1* :: *word8*)
by *auto*

declare *scast-0* [*L2opt*, *polish*]

declare *Word.sint-0* [*polish*]
declare *More-Word.sint-0* [*polish*]

lemma *sint-1-eq-1-x* [*polish*, *simp*]:
sint (*1* :: ((*'a*::*len*) *bit0*) *word*) = *1*
sint (*1* :: ((*'a*::*len*) *bit1*) *word*) = *1*
sint (*1* :: ((*'a*::*len*) *bit0*) *signed word*) = *1*
sint (*1* :: ((*'a*::*len*) *bit1*) *signed word*) = *1*
by *auto*

lemma *if-P-then-t-else-f-eq-t* [*L2opt*, *polish*]:
((*if P then t else f*) = *t*) = ($P \vee t = f$)
by *auto*

lemma *if-P-then-t-else-f-eq-f* [*L2opt*, *polish*]:
((*if P then t else f*) = *f*) = ($\neg\ P \vee t = f$)

by *auto*

lemma *sint-1-ne-sint-0*: *sint 1* $\neq$ *sint 0*
by *simp*

lemmas *if-P-then-t-else-f-eq-f-simps* [*L2opt*, *polish*] =
if-P-then-t-else-f-eq-f [**where** *t* = *sint 1* **and** *f* = *sint 0*, *simplified sint-1-ne-sint-0 simp-thms*]
if-P-then-t-else-f-eq-t [**where** *t* = *sint 1* **and** *f* = *sint 0*, *simplified sint-1-ne-sint-0 simp-thms*]
if-P-then-t-else-f-eq-f [**where** *t* = *1* :: *int* **and** *f* = *0* :: *int*, *simplified zero-neq-one-class.one-neq-zero simp-thms*]
if-P-then-t-else-f-eq-t [**where** *t* = *1* :: *int* **and** *f* = *0* :: *int*, *simplified zero-neq-one-class.one-neq-zero simp-thms*]

lemma *boring-bind-K-bind* [*simp*, *polish*]:
(*gets X* >>= (λ-. *M*)) = *M*
(*return Y* >>= (λ-. *M*)) = *M*
apply (*rule spec-monad-ext*, *simp add*: *run-bind*)+
done

lemma *pred-and-true-var*[*simp*]: ((λ-. *True*) *and P*) = *P*
by(*simp add*:*pred-conj-def*)
lemma *pred-and-true*[*simp*]: (*P and* (λ-. *True*)) = *P*
by(*simp add*:*pred-conj-def*)

declare *pred-and-true-var* [*L2opt*, *polish*]
declare *pred-and-true* [*L2opt*, *polish*]

lemmas [*polish*] = *rel-simps eq-numeral-extra*

declare *ptr-add-0-id*[*polish*]
declare *ptr-coerce.simps*[*polish*]

declare *uint-nat*[*symmetric*, *polish*]

lemma *finally-throw*: *finally* (*throw x*) = *return x*
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-finally*)
done

lemma *finally-liftE*: *finally* (*liftE m*) = *m*
apply (*auto simp add*: *runs-to-iff spec-monad-eq-iff intro*!: *runs-to-cong-pred-only*)
done

lemma *bind-post-state-map-post-state*:
bind-post-state (*map-post-state f g*) *h* = *bind-post-state g* (λ*x*. *h* (*f x*))
by (*simp flip*: *bind-post-state-pure-post-state2*)

lemma *map-post-state-case-exception-or-result-distrib*:
map-post-state f (*case-exception-or-result g h v*) =
(*case-exception-or-result* (λ*x*. *map-post-state f* (*g x*)) (λ*x*. *map-post-state f* (*h x*)) *v*)
by (*auto simp add*: *map-post-state-def split*: *exception-or-result-splits*)

lemma *map-post-state-case-exception-or-result-prod-distrib*:
map-post-state f
((*case v of* (*Exception e*, *t*) ⇒ *g e t*
| (*Result v*, *t*) ⇒ *h v t*)) =
((*case v of* (*Exception e*, *t*) ⇒ *map-post-state f* (*g e t*)
| (*Result v*, *t*) ⇒ *map-post-state f* (*h v t*)))
by (*auto simp add*: *map-post-state-def split*: *prod.splits exception-or-result-splits*)

lemma *finally-liftE-bind*: (*finally* ((*liftE L*) >>= (λ*r*. *R r*))) = (*L* >>= (λ*r*. (*finally* (*R r*))))
by (*auto simp add*: *spec-monad-eq-iff runs-to-iff intro*!: *runs-to-cong-pred-only*)

lemma *finally-guardE*: *finally* (*do* { *r* <− *guard X*; *Y* }) = *do* {*r* <− *guard X*; *finally Y* }
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-finally run-guard*)
done

lemma *finally-condition-distrib*: *finally* (*condition c X Y*) = *condition c* (*finally X*) (*finally Y*)
apply (*rule spec-monad-ext*)
apply (*simp add*: *run-bind run-finally run-liftE run-guard run-condition map-post-state-bind-post-state*)
done

lemma *unite-Exception-option*:
unite (*Exception* (*v*::'*a option*)) = *Result* (*the v*)
apply (*cases v*)
apply (*simp add*: *unite-def case-xval-def default-option-def option.the-def*)
by (*metis Exn-def option.sel unite-simps*(*1*))

lemma *finally-bindE-liftE-throw*: *finally* (*X* >>= (λ*v*. *L2-VARS* (*throw v*) *ns*)) = *finally X*
by (*auto simp add*: *spec-monad-eq-iff runs-to-iff L2-VARS-def intro*!: *runs-to-cong-cases*)

lemmas *finally-simps* =
finally-throw finally-liftE
finally-liftE-bind
finally-guardE
finally-condition-distrib
condition-fail-rhs condition-fail-lhs

bind-assoc

thm *finally-simps*
declare *finally-simps* [*polish*]

declare *finally-bindE-liftE-throw* [*simplified L2-VARS-def*, *polish*]

lemma *nested-bind*[*polish*, *simp*]:
do { *y* <− *f*;
return (*g y*)
} >>= *h* =
do { *y* <− *f*;
h (*g y*)
}
apply (*auto simp add*: *spec-monad-eq-iff runs-to-iff*)
done

lemma *select-UNIV-bind-const*[*simp*]: *select UNIV* >>= (λ-. *g*) = *g*
apply (*auto simp add*: *spec-monad-eq-iff runs-to-iff*)
done

lemma *do-bind-assoc*:
⋀*f fa fb*. *do* { *u* <− *f*::(*unit*, -) *res-monad*;
fa::(*unit*, -) *res-monad* } >>= *fb* = *do* {*u* <− *f*; *fa* >>=
fb::(*unit*, -) *res-monad* }
using *bind-assoc* **by** *blast*

24.1 Support to normalise guards and array index expressions

method *simp-guards* =
(*simp*
add:
guard-merge′ guard-merge-bind′ conj-commute
array-ptr-index-field-lvalue-conv
addressable-field-exec find-array-fields-Some
unat-less-helper
cong:
HOL.conj-cong)

simproc-setup *field-lvalue-unfold* (‹&(*p*::′*a*::*mem-type ptr*→*f*#*g*#*gs*)›) = ‹

```
let
  fun check @{term-pat ‹&(-→?fs)›} = is-some (try UMM-Proofs.dest-fields fs)
orelse raise Pattern.MATCH
    | check - = raise Pattern.MATCH
in
  fn - => fn ctxt => fn ct =>
  let
    val - = check (Thm.term-of ct)
    val {p, f, g, ...} = @{cterm-match ‹&(?p→?f#?g)›} ct
    val thm0 = Drule.infer-instantiate ctxt [((p, 0), p), ((f, 0), f), ((g, 0), g)]
          @{thm field-lvalue-cons-unfold}
    val thm1 = Utils.solve-sideconditions ctxt thm0 (
          ALLGOALS (asm-full-simp-tac ctxt))
   val - = Utils.verbose-msg 6 ctxt (fn - => field-lvalue-unfold:\n  ^ Thm.string-of-thm
ctxt thm1)
  in
    SOME thm1
  end
  handle THM - =>
           (Utils.verbose-msg 6 ctxt (fn - => field-lvalue-unfold proof failed:\n  ^
string-of-cterm ctxt ct);
          NONE)
       | Pattern.MATCH => NONE
end›
declare [[simproc del: field-lvalue-unfold]] — loops with ⟦field-ti TYPE(?'a) ?f =
Some ?t; export-uinfo ?t = typ-uinfo-t TYPE(?'b); field-ti TYPE(?'b) ?g = Some
?k⟧ ⟹ &(PTR(?'b) &(?p→?f)→?g) = &(?p→?f @ ?g)

attribute-setup array-bound-mksimps = ‹
  Scan.succeed (Thm.declaration-attribute
    (fn - => (Context.map-proof (UMM-Proofs.set-array-bound-mksimps))))›



lemmas whileLoop-congs-tupled =
whileLoop-cong
whileLoop-cong [split-tuple C and C' and B and B' arity: 2]
whileLoop-cong [split-tuple C and C' and B and B' arity: 3]
whileLoop-cong [split-tuple C and C' and B and B' arity: 4]
whileLoop-cong [split-tuple C and C' and B and B' arity: 5]
whileLoop-cong [split-tuple C and C' and B and B' arity: 6]
whileLoop-cong [split-tuple C and C' and B and B' arity: 7]
whileLoop-cong [split-tuple C and C' and B and B' arity: 8]
whileLoop-cong [split-tuple C and C' and B and B' arity: 9]
whileLoop-cong [split-tuple C and C' and B and B' arity: 10]
whileLoop-cong [split-tuple C and C' and B and B' arity: 11]
whileLoop-cong [split-tuple C and C' and B and B' arity: 12]
whileLoop-cong [split-tuple C and C' and B and B' arity: 13]
```

lemma *finally-condition-throw-conv*: *finally* (*condition c* (*throw x*) *g*) = *condition c* (*return x*) (*finally g*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-unless-throw-conv*: *finally* (*do* {*unless c* (*throw x*); *g*}) = *condition* (λ-. ¬ *c*) (*return x*) (*finally g*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-condition-throw-conv*: *finally* (*do* {*condition c* (*throw x*) *skip*; *h*}) = *condition c* (*return x*) (*finally h*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-return-conv*: *finally* (*return x*) = *return x*
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-guard-conv*: *finally* (*do* {*guard g*; *f*}) = *do* {*guard g*; *finally f*}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-modify-conv*: *finally* (*do* {*modify g*; *f*}) = *do* {*modify g*; *finally f*}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-gets-conv*: *finally* ((*gets g*) ⤜ *f*) = *do* {*r* ← *gets g*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-gets-the-conv*: *finally* (*gets-the g* ⤜ *f*) = *do* {*r* ← *gets-the g*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-unknown-conv*: *finally* (*unknown* ⤜ *f*) = *do* {*r* ← *unknown*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)

apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-get-state-conv*: *finally* (*get-state* $\gg\!=$ *f*) = *do* {*r* ← *get-state*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-set-state-conv*: *finally* (*set-state s* $\gg\!=$ *f*) = *do* {*r* ← *set-state s*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-select-conv*: *finally* (*select S* $\gg\!=$ *f*) = *do* {*r* ← *select S*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-assert-conv*: *finally* (*assert P* $\gg\!=$ *f*) = *do* {*r* ← *assert P*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-assume-conv*: *finally* (*assume P* $\gg\!=$ *f*) = *do* {*r* ← *assume P*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *finally-bind-assert-opt-conv*: *finally* (*assert-opt v* $\gg\!=$ *f*) = *do* {*r* ← *assert-opt v*; *finally* (*f r*)}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemmas *finally-convs* =
finally-liftE finally-liftE-bind
finally-condition-throw-conv
finally-unless-throw-conv
finally-bind-condition-throw-conv
finally-return-conv
finally-bind-guard-conv
finally-bind-modify-conv
finally-bind-gets-conv
finally-bind-gets-the-conv

finally-bind-unknown-conv
finally-bind-get-state-conv
finally-bind-set-state-conv
finally-bind-select-conv
finally-bind-assert-conv
finally-bind-assume-conv
finally-bind-assert-opt-conv
finally-throw

lemma *try-bind-liftE-conv*: (*try* (*liftE f* ⤜ *g*)) = (*liftE f* ⤜ (λ*x*. *try* (*g x*)))
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *try-bind-guard-conv*: *try* (*do* {*guard g*; *f*}) = *do* {*guard g*; *try f*}
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemma *try-unless-throw-Inr-conv*:
try (*do* {*unless c* (*throw* (*Inr x*)); *g*}) = *condition* (λ-. ¬ *c*) (*return x*) (*try g*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff unnest-exn-def*)
done

lemma *try-unless-throw-Inl-conv*:
try (*do* {*unless c* (*throw* (*Inl x*)); *g*}) = *condition* (λ-. ¬ *c*) (*throw x*) (*try g*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff unnest-exn-def*)
done

lemma *try-when-throw-Inr-conv*:
try (*do* {*when c* (*throw* (*Inr x*)); *g*}) = *condition* (λ-. *c*) (*return x*) (*try g*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff unnest-exn-def*)
done

lemma *try-when-throw-Inl-conv*:
try (*do* {*when c* (*throw* (*Inl x*)); *g*}) = *condition* (λ-. *c*) (*throw x*) (*try g*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff unnest-exn-def*)
done

lemma *try-bind-condition-throw-Inr-conv*: *try* (*do* {*condition c* (*throw* (*Inr x*)) *skip*; *h*}) = *condition c* (*return x*) (*try h*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff unnest-exn-def*)

done

lemma *try-bind-condition-throw-Inl-conv*: *try* (*do* {*condition c* (*throw* (*Inl x*)) *skip*; *h*}) = *condition c* (*throw x*) (*try h*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff unnest-exn-def*)
done

lemma *try-condition-conv*: *try* (*condition c f g*) = *condition c* (*try f*) (*try g*)
apply (*rule spec-monad-eqI*)
apply (*auto simp add*: *runs-to-iff*)
done

lemmas *try-convs* =
try-bind-liftE-conv
try-bind-guard-conv
try-unless-throw-Inr-conv
try-unless-throw-Inl-conv
try-when-throw-Inr-conv
try-when-throw-Inl-conv
try-bind-condition-throw-Inr-conv
try-bind-condition-throw-Inl-conv
try-condition-conv

lemma *runs-to-partial-case-prod* : (⋀*a b*. (*f a b*) ⋅ *s* ?⦃*Q*⦄) ⟹
(*case x of* (*a*, *b*) ⇒ *f a b*) ⋅ *s* ?⦃*Q*⦄
by (*cases x*) *simp*

24.2 Monad simplification with custom congruence rules

context *open-types*
begin

We supply a custom simplification method for spec monad expressions that gathers and propagates information from conditions and guards while descending into the term. The core motivation is to clean up repeated occurrences of *PTR-VALID*('*a*) (*heap-typing s*) *p*. Allthough this is a state dependent predicate it stays invariant as long as the typing does not change. So the goal of the simplification method is to prove preservation of such predicates while descending into the term. Supplying congruence rules to the simplifier is unfortunately not enough as we want to keep control over what kind of invariants are propagated. Unfortunately the simplifier has no concept of a *congprocs* that are triggered when descending into the term only the *simprocs* which are triggered by the usual bottom up simplifier strategy. To work around this limitation we implement *congprocs* by *simprocs*:

- We block the simplifier to descend into compound terms by adding

trivial congruence rules to the simplifier, like $f >>= g \equiv f >>= g$

- We add a simproc that then triggers on the $f >>= g$ and manually extends the context and invokes the simplifier on the subterms.

end

definition *ADD-FACT*:: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **where**
ADD-FACT P s = *P s*
definition *PRESERVED-FACTS*:: $('e::default, 'a, 's)$ *spec-monad* $\Rightarrow 's \Rightarrow ('e, 'a)$ *exception-or-result* $\Rightarrow 's \Rightarrow bool$ **where**
PRESERVED-FACTS f s r t = (*reaches f s r t*)
definition *PRESERVED-FACTS-WHILE* :: $('a \Rightarrow 's \Rightarrow bool) \Rightarrow ('a \Rightarrow ('e::default, 'a, 's)$ *spec-monad*$) \Rightarrow 'a \Rightarrow 's \Rightarrow 'a \Rightarrow 's \Rightarrow bool$ **where**
PRESERVED-FACTS-WHILE C B i s r t = *whileLoop-unroll-reachable C B i s r t*

definition *RUN-CASE-PROD* ::
$('a \Rightarrow 'b \Rightarrow ('c::default, 'd, 'e)$ *spec-monad*$) \Rightarrow 'a \times 'b \Rightarrow 'e \Rightarrow$
$('f \Rightarrow 'g \Rightarrow ('c, 'd, 'e)$ *spec-monad*$) \Rightarrow 'f \times 'g \Rightarrow bool$ **where**
RUN-CASE-PROD f x s f' x' $\longleftrightarrow$ *run* (*case-prod f x*) *s* = *run* (*case-prod f' x'*) *s*

lemma *ADD-FACT-D*: *ADD-FACT P s* $\Longrightarrow$ *P s*
by (*simp add*: *ADD-FACT-def*)

lemma *RUN-CASE-PROD-I*:$x \equiv x' \Longrightarrow$ *run* (*case-prod f x'*) *s* $\equiv$ *run* (*case-prod f' x'*) *s* $\Longrightarrow$ *RUN-CASE-PROD f x s f' x'*
by (*auto simp add*: *RUN-CASE-PROD-def*)

named-theorems
monad-simp-simp simplification rules to derive facts **and**
monad-simp-augment-rule conditional rules to augment facts **and**
monad-simp-split-tuple-cong congruence rules to control tuple expansion

lemma *PRESERVED-FACTS-runs-to-partial*: *PRESERVED-FACTS f s r t* $\Longrightarrow$ *f* $\cdot$ *s* $?\{\!|Q|\!\} \Longrightarrow$ *Q r t*
by (*auto simp add*: *runs-to-partial-def-old PRESERVED-FACTS-def reaches-succeeds*)

lemma *STOPI*: $x \equiv y \Longrightarrow x \equiv STOP\ y$ **by** (*simp add*: *STOP-def*)

lemma *expand-unused-case-prod*:
$(\lambda(x,y::('c * 'd)).\ f\ x) = (\lambda(x,\ y1,\ y2).\ f\ x)$
by *auto*

lemma *PRESERVED-FACTS-return-eq*: *PRESERVED-FACTS* (*return x*) *s* (*Result y*) *t* $\Longrightarrow$ *y* = *x*
by (*auto simp add*: *PRESERVED-FACTS-def*)

ML ‹

structure Monad-Cong-Simp =
struct
type rule = {*apply-thm*: *thm*, *stop-thm*: *thm*, *split-vars*: *string list*, *name*: *binding*}
type rules = *rule Net.net*

type cond-rule = {*thm*: *thm*, *name*: *binding*}
type cond-rules = *cond-rule Net.net*

fun rule-eq ({*apply-thm* = *thm1*, ...}:*rule*, {*apply-thm* = *thm2*, ...}:*rule*) = *Thm.eq-thm-prop* (*thm1*, *thm2*)
fun cond-rule-eq ({*thm* = *thm1*, ...}:*cond-rule*, {*thm* = *thm2*, ...}:*cond-rule*) = *Thm.eq-thm-prop* (*thm1*, *thm2*)

fun transfer-rule ctxt {*apply-thm*, *stop-thm*, *split-vars*, *name*}:*rule* =
{*apply-thm* = *Thm.transfer' ctxt apply-thm*,
stop-thm = *Thm.transfer' ctxt stop-thm*,
split-vars = *split-vars*, *name* = *name*}

fun merge-rules (*n1*, *n2*) =
if pointer-eq (*n1*, *n2*) *then n1*
else if Net.is-empty n2 then n1
else if Net.is-empty n1 then n2
else Net.merge rule-eq (*n1*, *n2*)

fun merge-cond-rules (*n1*, *n2*) =
if pointer-eq (*n1*, *n2*) *then n1*
else if Net.is-empty n2 then n1
else if Net.is-empty n1 then n2
else Net.merge cond-rule-eq (*n1*, *n2*)

type monad-cong-data = {*rules*: *rules*, *derive-rules*: *cond-rules*,
stop-congs: *thm list*, *simprocs*: *string list*}

structure Data = *Generic-Data* (
type T = *monad-cong-data*;
val empty = {*rules* = *Net.empty*, *derive-rules* = *Net.empty*, *stop-congs* = [], *simprocs* = []}
fun merge ({*rules* = *rules1*, *derive-rules* = *d1*, *stop-congs* = *congs1*, *simprocs* = *simprocs1*},
{*rules* = *rules2*, *derive-rules* = *d2*, *stop-congs* = *congs2*, *simprocs* = *simprocs2*}) =
{*rules* = *merge-rules* (*rules1*, *rules2*), *derive-rules* = *merge-cond-rules* (*d1*,

```
d2),
        stop-congs = Library.merge (Thm.eq-thm-prop) (congs1, congs2),
        simprocs = Ord-List.merge fast-string-ord (simprocs1, simprocs2)}
)

fun map-rules f = Data.map (fn {rules, derive-rules, stop-congs, simprocs}:monad-cong-data
=>
  {rules = f rules, derive-rules = derive-rules, stop-congs = stop-congs, simprocs
= simprocs})
fun map-derive-rules f = Data.map (fn {rules, derive-rules, stop-congs, simprocs}:monad-cong-data
=>
  {rules = rules, derive-rules = f derive-rules, stop-congs = stop-congs, simprocs
= simprocs})
fun map-stop-congs f = Data.map (fn {rules, derive-rules, stop-congs, simprocs}:monad-cong-data
=>
  {rules = rules, derive-rules = derive-rules, stop-congs = f stop-congs, simprocs
= simprocs})
fun map-simprocs f = Data.map (fn {rules, derive-rules, stop-congs, simprocs}:monad-cong-data
=>
  {rules = rules, derive-rules = derive-rules, stop-congs = stop-congs, simprocs =
f simprocs})


fun add-simproc p = map-simprocs (Ord-List.insert fast-string-ord p)

type fact = {pred: cterm, thms: thm list}

type proof-data = {states: cterm list, facts: fact list,
  start: (Timing.start * string) option,
  cache: thm Net.net Unsynchronized.ref }

structure Prf-Data = Proof-Data (
  type T = proof-data;
  val init = (K {states = [], facts = [], start = NONE, cache = Unsynchronized.ref
Net.empty});
)

fun map-states f = Prf-Data.map (fn {states, facts, start, cache}:proof-data =>
  {states = f states, facts = facts, start = start, cache = cache})
fun map-facts f = Prf-Data.map (fn {states, facts, start, cache}:proof-data =>
  {states = states, facts = f facts, start = start, cache = cache})
fun map-start f = Prf-Data.map (fn {states, facts, start, cache}:proof-data =>
  {states = states, facts = facts, start = f start, cache = cache})
fun map-cache f = Prf-Data.map (fn {states, facts, start, cache}:proof-data =>
  {states = states, facts = facts, start = start, cache = f cache})

val facts-of = #facts o Prf-Data.get
val states-of = #states o Prf-Data.get
val start-of = #start o Prf-Data.get
```

```
val cache-of = #cache o Prf-Data.get


val reset-cache = map-cache (fn - => Unsynchronized.ref Net.empty)

fun add-to-cache ctxt thm =
  let
    val cache-ref = cache-of ctxt
      val cache' = Net.insert-term (Thm.eq-thm-prop ) (Thm.prop-of thm, thm)
(!cache-ref)
    val - = cache-ref := cache'
    val - = Utils.verbose-msg 4 ctxt (fn - => cache add: ^ Thm.string-of-thm ctxt
thm)
  in
    ctxt
  end

fun lookup-cache ctxt ct =
  Net.match-term (!(cache-of ctxt)) (Thm.term-of ct)

fun start str = map-start (K (SOME (Timing.start (), str)))

fun stop str ctxt = ctxt |> map-start (Option.mapPartial (fn (start, str0) =>
  let
    val - = Utils.timing-msg' 3 ctxt (fn - => str0 ^ -> ^ str) start
  in
    NONE
  end))

fun monad-simp-active ctxt = not (null (states-of ctxt))

val lhs-of = Thm.dest-equals-lhs o Thm.cconcl-of
val rhs-of = Thm.dest-equals-rhs o Thm.cconcl-of

val lhs-of-rule = #apply-thm #> lhs-of


fun cong-lhs rule =
  let
    val lhs = lhs-of rule
    val {f, ...} = @{cterm-match (fo) run ?f ?s} lhs
  in
    f
  end

fun stop-cong rule =
  Thm.reflexive (cong-lhs rule)

fun mk-rule ctxt name split-vars thm =
```

```
  let
    val eq = Simplifier.mk-cong ctxt thm
    val apply-thm = eq |> Drule.zero-var-indexes
    val lhs = lhs-of apply-thm
    val stop-thm = stop-cong apply-thm
   val rule = {apply-thm = Thm.trim-context apply-thm, stop-thm = Thm.trim-context
stop-thm, split-vars = split-vars, name = name}
  in
    (Thm.term-of lhs, rule)
  end

fun mk-cond-rule name thm =
  let
    val prem = Thm.major-prem-of thm
    val rule = {thm = Thm.trim-context thm, name = name}
  in
    (prem, rule)
  end

fun compare-congs (thm1, thm2) =
  let
    val maxidx1 = Thm.maxidx-of thm1
    val lhs1 = lhs-of thm1
    val lhs2 = lhs-of thm2 |> Thm.incr-indexes-cterm (maxidx1 + 1)
    val result =
      if is-some (try Thm.match (lhs1, lhs2)) then
        SOME GREATER
      else if is-some (try Thm.match (lhs2, lhs1)) then
        SOME LESS
      else NONE
  in
   result
  end

fun equiv-congs (thm1, thm2) =
  compare-congs (thm1, thm2) = SOME GREATER andalso compare-congs (thm2,
thm1) = SOME GREATER

fun more-general thm1 thm2 =
  (case compare-congs (thm1, thm2) of
     SOME GREATER => SOME thm1
   | SOME LESS => SOME thm2
   | NONE => NONE)

fun more-specific thm1 thm2 =
  (case compare-congs (thm1, thm2) of
     SOME GREATER => SOME thm2
   | SOME LESS => SOME thm1
   | NONE => NONE)
```

```
fun insert-cong thm [] = [Thm.trim-context thm]
  | insert-cong thm (thms as (thm1::thms')) =
      (case compare-congs (thm1, thm) of
         SOME GREATER => thms
       | SOME LESS => insert-cong thm thms'
       | NONE => thm1 :: insert-cong thm thms')

fun rules-of ctxt = Data.get (Context.Proof ctxt)

fun build-congs stop-thms =
  [] |> fold (insert-cong) stop-thms

fun rebuild-stop-congs context =
  let
   val stop-thms = Data.get context |> #rules |> Net.entries |> map (Thm.transfer''
context o #stop-thm)
    val congs = build-congs stop-thms |> map Thm.trim-context
  in context |> (map-stop-congs (K congs)) end

val rebuild-stop-congs' = Context.proof-map rebuild-stop-congs

fun add-rule split-vars thm context =
  let
     val concl = Thm.concl-of thm
     val concl-vars = Term.add-vars concl [] |> map (#1 o #1)
     val unused = filter-out (member (op =) concl-vars) split-vars
     val - = null unused orelse error (split variables do not occur in conclusion: ^
@{make-string} unused)
     val ctxt = Context.proof-of context
     val name = Utils.guess-binding-of-thm ctxt thm
     val rule = mk-rule ctxt name split-vars thm
     val cong = #stop-thm (snd rule) |> Thm.transfer' ctxt
  in
    context
    |> map-rules (Net.insert-term-safe rule-eq rule)
    |> map-stop-congs (insert-cong cong)
  end

fun add-rule' split-vars thm =
  Context.proof-map (add-rule split-vars thm)

fun del-rule thm context =
  let
    val ctxt = Context.proof-of context
  in
    context
   |> map-rules (Net.delete-term-safe rule-eq (mk-rule ctxt Binding.empty [] thm))
    |> rebuild-stop-congs
```

```
  end

fun del-rule' thm = Context.proof-map (del-rule thm)

fun add-derive-rule thm0 context =
  let
    val ctxt = Context.proof-of context
    val thm = Simplifier.simplify ctxt thm0
    val name = Utils.guess-binding-of-thm ctxt thm
    val rule = mk-cond-rule name thm
  in
    context |> map-derive-rules (Net.insert-term-safe (* (K true) *)  cond-rule-eq
rule)
  end

fun del-derive-rule thm0 context =
  let
    val ctxt = Context.proof-of context
    val thm = Simplifier.simplify ctxt thm0
  in
    context
     |> map-derive-rules (Net.delete-term-safe cond-rule-eq (mk-cond-rule Bind-
ing.empty thm))
  end

fun add-global-stop-cong thm =
  map-stop-congs (fn congs => (build-congs (thm :: congs)))

fun del-global-stop-cong thm =
  map-stop-congs (fn congs => (build-congs (remove Thm.eq-thm-prop thm congs)))

fun pretty-entry label p = Pretty.block [(Pretty.str (suffix :  label)), p]

fun pretty-rule ctxt (rule:rule) =
 Pretty.item (
 [pretty-entry rule (More-Binding.here-pretty (#name rule)), Pretty.brk 1,
   Pretty.indent 2 (Thm.pretty-thm ctxt (#apply-thm rule) |> Pretty.cartouche),
Pretty.brk 1])

fun pretty-rules ctxt (rules: rule list) = Pretty.big-list (rules) (map (pretty-rule ctxt)
rules)

fun print-rules context =
  Data.get context |> #rules |> Net.entries
  |> pretty-rules (Context.proof-of context)
  |> Pretty.string-of |> writeln

fun pretty-cond-rule ctxt (rule:cond-rule) =
 Pretty.item (
```

```
 [pretty-entry cond-rule (More-Binding.here-pretty (#name rule)), Pretty.brk 1,
  Pretty.indent 2 (Thm.pretty-thm ctxt (#thm rule) |> Pretty.cartouche), Pretty.brk
1])

fun pretty-derive-rules ctxt (rules: cond-rule list) =
  Pretty.big-list (derive rules) (map (pretty-cond-rule ctxt) rules)

fun print-derive-rules context =
  Data.get context |> #derive-rules |> Net.entries
  |> pretty-derive-rules (Context.proof-of context)
  |> Pretty.string-of |> writeln

fun pretty-stop-cong ctxt thm =
  Pretty.item [Thm.pretty-thm ctxt thm]

fun pretty-stop-congs ctxt thms = Pretty.big-list (stop-congs) (map (pretty-stop-cong
ctxt) thms)

fun print-stop-congs context =
  Data.get context |> #stop-congs
  |> pretty-stop-congs (Context.proof-of context)
  |> Pretty.string-of |> writeln

fun map-theory-rules f thy =
  let
    val ctxt' = f (Proof-Context.init-global thy);
    val thy' = Proof-Context.theory-of ctxt';
  in
    Context.theory-map (Data.map (K (rules-of ctxt'))) thy'
  end

fun mk-fact thm =
  let
    val {P, ... } = @{cterm-match ‹Trueprop (ADD-FACT ?P ?s)›} (Thm.cconcl-of
thm)
  in
    {pred = P, thms = [@{thm ADD-FACT-D} OF [thm]]}:fact
  end

fun derive-facts ctxt thm =
  let
      val derive-rules = Net.match-term (#derive-rules (Data.get (Context.Proof
ctxt))) (Thm.concl-of thm)
    val derived = map-filter (try (fn rule => rule OF [thm])) (map (Thm.transfer'
ctxt o #thm) derive-rules)
  in
    map mk-fact derived
  end
```

```
fun get-more-specific [] = NONE
  | get-more-specific [rule] = SOME rule
  | get-more-specific (rule1::rules) =
    let
      fun get-more-specific' (lhs0, rule0) [] = SOME rule0
        | get-more-specific' (lhs0, rule0) (rule1::rules) =
        let
          val lhs1 = lhs-of-rule rule1
        in
          if is-some (try Thm.match (lhs0, lhs1)) then
            get-more-specific' (lhs1, rule1) rules
          else
            get-more-specific' (lhs0, rule0) rules
        end
    in
      get-more-specific' (lhs-of-rule rule1, rule1) rules
    end

fun matching-rule ctxt trm =
  let
    val rules = Data.get (Context.Proof ctxt) |> #rules
    val matches = Net.match-term rules trm
  in get-more-specific matches |> Option.map (transfer-rule ctxt) end

fun lookup-by-name tvars x =
  let
    val xs = Vars.dest tvars |> map (apfst fst)
  in
    AList.lookup (op =) xs (x, 0)
  end

fun lhs-concl-conv cv thm =
  let
    val n = Thm.nprems-of thm
  in
    Conv.fconv-rule
     (Conv.concl-conv n
        (Conv.arg1-conv cv)) thm
  end

(* Plain Conv.rewr-conv did not work in our setting. The matching of lhs and ct
fails.
   But Simplifier.simplify would work.
   So Conv.rewr-conv seems to do too much for our exact match but not as much
as simplifier.
   (Maybe some kind of weired eta-conversion stuff?)
   But as we have an exact match anyway there is no need to match again.
*)
fun rewr-exact-conv eq ct =
```

```
  let
    val eq2 =
      if Thm.lhs-of eq aconvc ct then eq
      else
        let val ceq = Thm.dest-fun2 (Thm.cprop-of eq)
        val eq1 = Thm.trivial (Thm.mk-binop ceq ct (Thm.rhs-of eq))
        in eq COMP eq1 end;
  in Thm.transitive eq2 (Thm.beta-conversion true (Thm.rhs-of eq2)) end;

fun inst-split-rule ctxt ({apply-thm, split-vars, ...}:rule) ct =
  let
    val lhs-rule = lhs-of apply-thm
    val env as (Tvars, tvars) = Thm.match (lhs-rule, ct)
  in
    if null split-vars then
      (Thm.instantiate env apply-thm, 0, [])
    else
      let
        val orig-var = hd split-vars
        val bdy = the (lookup-by-name tvars orig-var)
        val domT = Thm.typ-of-cterm bdy |> domain-type
        val arity =  domT |> HOLogic.flatten-tupleT |> length
        val names = Tuple-Tools.strip-case-prod' (Thm.term-of bdy)
        val split-ctxt = Proof-Context.init-global (Proof-Context.theory-of ctxt)
        val (no-split, splitted-thm) =
              if length names = 1 andalso arity > 1 then
                (true, apply-thm) (* corner case of trivial ‹λ(-::('a * ...)). f› *)
              else
                (false, Tuple-Tools.split-rule split-ctxt split-vars apply-thm arity)

        val tvars-eta = Vars.map (fn - => Tuple-Tools.eta-expand-tuple ctxt) tvars
        val eta-inst-rule = Thm.instantiate (Tvars, tvars-eta) apply-thm
        val lhs-eta-inst-rule = lhs-of eta-inst-rule
        val (inst-rule, names') =
          if arity <= 1  then (eta-inst-rule, names)
          else
            let
              val env1 = Thm.match (lhs-of splitted-thm, lhs-eta-inst-rule)
              val inst-thm1 = Thm.instantiate env1 splitted-thm
              val inst-thm2 =
                if Utils.cterm-eq (lhs-eta-inst-rule, ct) then inst-thm1
                else
                  let
                  val - = warning (eta expanding case prod from:\n ^ string-of-cterm
ctxt ct ^ to:\n ^
                                   string-of-cterm ctxt lhs-eta-inst-rule)
                  val eta-eq-prop = infer-instantiate ‹eta-lhs = lhs-eta-inst-rule
and orig-lhs = ct
                  in cterm ‹eta-lhs ≡ orig-lhs›› ctxt
```

```
                val eta-eq = Goal.prove-internal ctxt [] eta-eq-prop
                      (fn - => asm-full-simp-tac ((Simplifier.clear-simpset ctxt)
addsimps
                          @{thms case-prod-eta bind-case-prod-trivial case-prod-beta
                            case-prod-beta' prod.sel prod.collapse}) 1)
                val obj = Thm.term-of (Thm.lhs-of eta-eq)
                val pat = Thm.term-of (Utils.clhs-of (Thm.cconcl-of inst-thm1))
                val inst-thm' = lhs-concl-conv (rewr-exact-conv eta-eq) inst-thm1
                (* Preserve bound names *)
                val inst-thm'' = Thm.rename-boundvars pat obj inst-thm'
              in inst-thm'' end
            val bdy' = the (lookup-by-name tvars-eta orig-var)
             val names' = if no-split then names else Tuple-Tools.strip-case-prod'
(Thm.term-of bdy')
        in
          (inst-thm2, names')
        end

      val names'' = if arity = 1 andalso null names' then
                      [(r, domT)] (* make up name for eta-contracted corner case*)
                    else names'
    in
      (inst-rule, arity, names'')
    end
  end

fun strip-all-fix names ct ctxt =
  let
    val n = length names
    val (bounds, -) = Utils.strip-all-open [] (Thm.term-of ct) |> apfst rev
    val - = if n <> length bounds then raise CTERM (strip-all-fix: unequal length:
^ @{make-string} (names, bounds), [ct]) else ()
    val vars = names ~~ (map snd (take n bounds))
    val (fixes, ctxt') = Utils.fix-variant-cfrees vars ctxt
   val thm = Goal.prove-internal ctxt [] ct (fn - => ALLGOALS (Skip-Proof.cheat-tac
ctxt))
    val thm' = thm |> fold Thm.forall-elim fixes
    val ct' = Thm.cprop-of thm'
  in
   ((fixes, ct'), ctxt')
  end

fun fix-import-state-and-names ctxt names ct =
  let
    val states = Prf-Data.get ctxt |> #states
    val nstates = length states
    val ((fixes, ct'), ctxt') = strip-all-fix ((s ^ string-of-int nstates)::names) ct ctxt
  in ((fixes, ct'), ctxt') end
```

```
fun match-preserved-facts ctxt names ct =
 let
   val ((fixes, ct'), ctxt') = fix-import-state-and-names ctxt names ct
       handle CTERM - => raise Match
   val {f, s, r, t, lhs, rhs, ... } = @{cterm-match ‹PRESERVED-FACTS ?f ?s ?r
?t ⟹ ?lhs ≡ ?rhs›} ct'
   val {g', ...} = @{cterm-match ‹run ?g' ?s›} rhs
 in
   {f = f, s = s, t = t, r = r, results = tl fixes, lhs = lhs, g' = g', ctxt' = ctxt'}
 end

fun match-preserved-facts-while ctxt names ct =
  let
    val ((fixes, ct'), ctxt') = fix-import-state-and-names ctxt names ct
        handle CTERM - => raise Match
    val {C, B, i, s, r, t, C-lhs, C-rhs, B-lhs, B-rhs, ... } =
          @{cterm-match ‹PRESERVED-FACTS-WHILE ?C ?B ?i ?s ?r ?t ⟹
             ((?C-lhs ≡ ?C-rhs) &&& (?C-rhs ⟹ (?B-lhs ≡ ?B-rhs)))›} ct'
    val C' = fst (Utils.strip-comb-cterm C-rhs)
    val {B', ...} = @{cterm-match ‹run ?B' ?t›} B-rhs
  in
    {C = C, C' = C', B = B, B' = B', i = i, s = s, r = r, t =t, results = tl
fixes,
     C-lhs = C-lhs, B-lhs = B-lhs, ctxt' = ctxt' }
  end

fun rule-by-tactic-schematic ctxt thm tac =
  Utils.check-solve-sideconditions (K true) ctxt tac thm

fun rhs-conv conv = Conv.fconv-rule (Conv.arg-conv conv)
fun STOP-conv ctxt eq =
  let
  in
    Utils.timeit-msg 4 ctxt (fn - => STOP-conv: )
      (fn - => rhs-conv (Conv.try-conv (Conv.rewr-conv @{thm STOP-def})) eq)
  end


fun rewr-conv rules ct =
  let
    val thms = Net.match-term rules (Thm.term-of ct)
  in
    Conv.rewrs-conv thms ct
  end

fun mk-rewr-conv eqs =
  let
    fun prep eq =
      let val eq' = safe-mk-meta-eq eq
```

```
    in (Utils.rhs-of-eq (Thm.prop-of eq'), eq') end

  val rules = Net.empty |> fold (Net.insert-term (Thm.eq-thm) o prep) eqs
 in
  rewr-conv rules
 end


fun dest-abs-fresh (n, x) f =
 snd (Thm.dest-abs-fresh n f) handle CTERM - => Thm.apply f x

fun strip-case-prod-cterm ctxt ct =
 let
  val args = Tuple-Tools.strip-case-prod' (Thm.term-of ct)
  val (fixes, ctxt1) = Utils.fix-variant-cfrees args ctxt
  val orig-names = map fst args
  val names = map (fst o dest-Free o Thm.term-of) fixes
  val named-fixes = names ~~ fixes
  fun strip [x, y] ct =
        let
          val {f, ...} = @{cterm-match (fo) case-prod ?f} ct
        in dest-abs-fresh y (dest-abs-fresh x f) end
    | strip [x] ct = dest-abs-fresh x ct
    | strip [] ct = ct
    | strip (x::xs) ct =
        let
          val {f, ...} = @{cterm-match (fo) case-prod ?f} ct
          val ct' = dest-abs-fresh x f
        in strip xs ct' end
  val bdy = strip named-fixes ct
 in
  ((names ~~ fixes, orig-names, bdy), ctxt1)
 end


fun case-prod-bdy-conv ctxt ct =
 let
    val ((fixes, orig-names,  bdy), ctxt') = strip-case-prod-cterm ctxt ct
    val bdy-eq = Simplifier.rewrite ctxt' bdy
    val lhs = bdy-eq |> Thm.lhs-of
    val rhs = bdy-eq |> Thm.rhs-of
    val fixes' = map2 (fn (-, x) => fn n => (n, x)) fixes orig-names
    val f = Utils.lambdas-tupled fixes' lhs
    val g = Utils.lambdas-tupled fixes' rhs
    val eq-prop = infer-instantiate ‹f = f and g = g in cprop f ≡ g› ctxt
   val simps = if (Thm.term-of f) aconv (Thm.term-of g)  then [] else (Proof-Context.export
ctxt' ctxt [bdy-eq])
    val simp-ctxt = Simplifier.clear-simpset ctxt addsimps (@{thms fun-eq-iff} @
simps)
```

```
    val eq = Goal.prove-internal simp-ctxt [] eq-prop
          (fn - => simp-tac simp-ctxt 1)
  in
    eq
  end

val case-prod-cong = @{lemma x ≡ x' ⟹ case-prod f ≡ g ⟹ case-prod f x = g
x' for x x' f g by auto }
fun case-prod-apply-conv ctxt = Match-Cterm.switch [
  @{cterm-match case-prod ?f ?x} #> (fn {f, x, ct-, ...} =>
    let
      val x-eq = Simplifier.rewrite ctxt x
      val cp = infer-instantiate ‹f = f in cterm ‹case-prod f›› ctxt
      val cp-eq = case-prod-bdy-conv ctxt cp
    in case-prod-cong OF [x-eq, cp-eq] end)]


val pure-eqD = @{lemma P ≡ True ⟹ P by simp}
val pure-eqI = @{lemma P ⟹ P ≡ True by simp}

fun augment-rule-format thm =
 let
    val thm1 = the-default thm (try (fn thm => thm RSN (1, pure-eqI) — thm
[then pure-eqI]) thm)
    val thm2 = the-default thm1 (try (fn thm => thm OF [pure-eqD]) thm1)
 in
    thm2
 end

(* e.g. avoid substition with ‹n = global-variable s› *)
fun state-dependent-rhs ctxt thm =
  case states-of ctxt of
    [] => false
  | (s::-) =>
    let val s' = Thm.term-of s
    in exists-subterm (fn t => t = s') (Utils.rhs-of (Thm.concl-of thm)) end

fun gen-mksimps do-simp ctxt thm =
  let
   val augment-rules = Named-Theorems.get ctxt @{named-theorems monad-simp-augment-rule}
    fun augment thm =
      let
         val augs = map-filter (fn rule => try (fn rule => rule OF [thm]) rule)
augment-rules
          |> map (Simplifier.simplify ctxt) |> filter-out Utils.trivial-meta-eq-thm
        val - = Utils.verbose-msg 7 ctxt (fn - => augmenting ^ Thm.string-of-thm
ctxt thm ^ with ^ string-of-thms ctxt augs)
      in
        thm :: augs
```

```
    end
    val thm' = if do-simp then Simplifier.asm-full-simplify ctxt thm else thm
  in
     thm' |> Simplifier.mksimps ctxt |> filter-out ( Utils.trivial-meta-eq-thm orf
(state-dependent-rhs ctxt)) (* avoid loops *)
        |> maps augment
  end

val mksimps = gen-mksimps false


fun params-of prem = prem |> Thm.term-of |> Utils.strip-all-open [] |> fst

fun timeit-export str inner outer thms =
  Utils.timeit-msg 3 outer (fn - => export ^ str) (fn - => Proof-Context.export
inner outer thms)

fun preserved-facts-return-eqs ctxt preserved-thm =
   case try (fn thm => @{thm PRESERVED-FACTS-return-eq} OF [thm]) pre-
served-thm of
     NONE => []
  | SOME eq => gen-mksimps true ctxt eq

fun contains-frees frees thm =
  exists (member (op =) frees) (Term.add-frees (Thm.prop-of thm) [])

fun new-from-defs ctxt defs thms =
  let
    val lhss = map (Thm.term-of o lhs-of) defs
  in
    thms |> map (Local-Defs.fold ctxt defs)
    |> filter (exists-subterm (member (aconv) lhss) o Thm.prop-of)
  end

fun new-facts-from-defs ctxt defs ({pred, thms}:fact) =
  case new-from-defs ctxt defs thms of
    [] => NONE
  | thms' =>
   let val pred' = Drule.mk-term pred |> Local-Defs.fold ctxt defs |> Drule.dest-term

    in SOME ({pred=pred', thms=thms'}:fact) end

fun monad-state-conv (rule as {split-vars, ...}) ctxt ct =
  let
     val - = Utils.verbose-msg 4 ctxt (fn - => (rule:\n ^ Thm.string-of-thm ctxt
(#apply-thm rule)))

    val (inst-rule, arity, names) = Utils.timeit-msg 3 ctxt (fn - => inst-split-rule:
)
```

```
    (fn - =>inst-split-rule ctxt rule ct)
   val - = Utils.verbose-msg 7 ctxt (fn - => (inst-rule:\n  ^ Thm.string-of-thm
ctxt inst-rule))

  fun simp-rule rule =
    let
    in
    (case Thm.cprems-of rule of
      [] => rule
     | (prem::-) =>
      let val - = Utils.verbose-msg 7 ctxt (fn - => prem:  ^ string-of-cterm ctxt
prem)
      in prem
      |> Match-Cterm.switch [
        @{cterm-match (fo) Trueprop (RUN-CASE-PROD ?f ?x ?s ?f' ?x')} #>
(fn {f, x, s, f', x',...} =>
            let
              val ctxt = stop RUN-CASE-PROD ctxt
              val x-eq = Utils.timeit-msg 4 ctxt (fn - => eq (0) x: )
               (fn - => Simplifier.asm-full-rewrite ctxt x)
                    val - = Utils.verbose-msg 5 ctxt (fn - => (eq (0) x:\n  ^
Thm.string-of-thm ctxt x-eq))
              val start0 = Timing.start ();
              val x'-inst = rhs-of x-eq
              val f1 = infer-instantiate ‹f = f in cterm ‹case-prod f›› ctxt
              val ((fixes, orig-names, f-bdy), ctxt') = strip-case-prod-cterm ctxt f1
               val run-f = infer-instantiate ‹f = f-bdy and s = s in cterm‹run f
s› › ctxt'
              val - = Utils.timing-msg' 3 ctxt (fn - => case-prod (0)) start0

              val run-f-eq = Utils.timeit-msg 4 ctxt (fn - => eq (1) x: )
               (fn - => Simplifier.asm-full-rewrite (start eq (1) x ctxt') run-f |>
STOP-conv ctxt')
                    val - = Utils.verbose-msg 5 ctxt (fn - => (eq (1) f:\n  ^
Thm.string-of-thm ctxt run-f-eq))
              val start1 = Timing.start ();
              val {f = f1', ...} = @{cterm-match ‹run ?f ?s›} (rhs-of run-f-eq)
              val fixes' = map2 (fn (-, x) => fn n => (n, x)) fixes orig-names
              val f-tupled = Utils.lambdas-tupled fixes' f-bdy
              val f'-tupled = Utils.lambdas-tupled fixes' f1'
             val eq-prop = infer-instantiate ‹f = f-tupled and f' = f'-tupled and
x = x'-inst and s = s in
                  cprop ‹run (f x) s ≡ run (f' x) s›› ctxt
          val simps = if Utils.trivial-meta-eq-thm run-f-eq then [] else (timeit-export
RUN-CASE-PROD ctxt' ctxt [run-f-eq])
                    @ @{thms Spec-Monad.run-case-prod-distrib}
                val simp-ctxt = Simplifier.clear-simpset ctxt addsimps (@{thms
fun-eq-iff} @ simps)
              val eq = Goal.prove-internal simp-ctxt [] eq-prop
```

```
      (fn - => simp-tac simp-ctxt 1)
    val thm = @{thm RUN-CASE-PROD-I} OF [x-eq, eq]
    val rule1 = rule OF [thm]
    val - = Utils.timing-msg' 3 ctxt (fn - => case-prod (1)) start1
  in simp-rule rule1 end),
 @{cterm-match (fo) ?lhs ≡ run ?f ?s} #> (fn {lhs, f, s, ...} =>
  let
    val ctxt = stop lhs = run f s ctxt
    val fN = fst (dest-Var (Thm.term-of f))
    val eq = Utils.timeit-msg 4 ctxt (fn - => eq (1): )
        (fn -  => Simplifier.asm-full-rewrite (start eq (1) ctxt) lhs |>
STOP-conv ctxt)
   val - = Utils.verbose-msg 5 ctxt (fn - => (eq (1):\n  ^ Thm.string-of-thm
ctxt eq))
    val start0 = Timing.start ();
    val {f', ...} = @{cterm-match ‹run ?f' ?s›} (rhs-of eq)
    val rule1 = Drule.infer-instantiate ctxt [(fN, f')] rule
    val rule2 = rule1 OF [eq]
    val - = Utils.timing-msg' 3 ctxt (fn - => lhs = run f s (0)) start0
  in simp-rule rule2 end),
 @{cterm-match (fo) ?lhs ≡ ?f ?s} #> (fn {lhs, f, s, ct-,...} =>
  let
    val ctxt = stop lhs = f s ctxt
    val fN = fst (dest-Var (Thm.term-of f))
    val eq = Utils.timeit-msg 4 ctxt (fn - => eq (2): )
      (fn -  => Simplifier.asm-full-rewrite (start eq (2) ctxt) lhs)
   val - = Utils.verbose-msg 5 ctxt (fn - => (eq (2):\n  ^ Thm.string-of-thm
ctxt eq))
    val start0 = Timing.start ();
    val rhs-eq = rhs-of eq
    val f' = Thm.lambda-name (s, s) rhs-eq
    val rule1 = Drule.infer-instantiate ctxt [(fN, f')] rule
    val rule2 = rule1 OF [eq]
    val - = Utils.timing-msg' 3 ctxt (fn - => lhs = f s (0)) start0
  in simp-rule rule2 end),
 @{cterm-match (fo) ?lhs ≡ ?f} #> (fn {lhs, f,...} =>
  let
    val ctxt = stop lhs = f ctxt
    val fN = fst (dest-Var (Thm.term-of f))
    val eq = Utils.timeit-msg 4 ctxt (fn - => eq (3): )
      (fn -  => Simplifier.asm-full-rewrite (start eq (3) ctxt) lhs)
   val - = Utils.verbose-msg 5 ctxt (fn - => (eq (3):\n  ^ Thm.string-of-thm
ctxt eq))
    val start0 = Timing.start ();
    val rhs-eq = rhs-of eq
    val f' = rhs-eq
    val rule1 = Drule.infer-instantiate ctxt [(fN, f')] rule
    val rule2 = rule1 OF [eq]
    val - = Utils.timing-msg' 3 ctxt (fn - => lhs = f (0)) start0
```

```
    in simp-rule (rule2) end),
  @{cterm-match (fo) ADD-FACT ?P ?s ⟹ ?lhs ≡ run ?f ?s} #> (fn
{P, s, lhs, f, ...} =>
    let
      val ctxt = stop ADD-FACT ctxt
      val start0 = Timing.start ();
      val fN = fst (dest-Var (Thm.term-of f))
          val fact-prop = infer-instantiate ‹P = P and s = s in cprop
‹ADD-FACT P s›› ctxt
      val ([fact], ctxt1) = Assumption.add-assumes [fact-prop] ctxt
      val fact0 = Utils.apply-beta-conv P s
        |> Utils.Trueprop-cterm
      val fact1 = Goal.prove-internal ctxt1 [] fact0 (fn - =>
           Method.insert-tac ctxt1 [fact] 1 THEN
          asm-full-simp-tac (ctxt1 addsimps @{thms ADD-FACT-def}) 1)
      val fact-eqs = gen-mksimps true ctxt1 fact1
      val fact-ariths = Utils.iariths-of-eqs fact-eqs
     val ctxt2 = (ctxt1 |> map-facts (cons {pred = P, thms = fact-eqs}))
          addsimps fact-eqs
          |> Utils.add-ariths fact-ariths
          |> Simplifier.add-prems fact-eqs
      val - = Utils.timing-msg' 3 ctxt (fn - => ADD-FACT (0)) start0
    val - = Utils.verbose-msg 4 ctxt (fn - => (adding:\n  ^ string-of-thms
ctxt1 fact-eqs))
      val eq = Utils.timeit-msg 4 ctxt (fn - => eq (4): )
        (fn -  => Simplifier.asm-full-rewrite (start eq (4) ctxt2) lhs
          |> STOP-conv ctxt2 |> singleton (timeit-export (eq (4): ) ctxt2
ctxt))
  val - = Utils.verbose-msg 5 ctxt (fn - => (eq (4):\n  ^ Thm.string-of-thm
ctxt eq))
      val start1 = Timing.start ();
      val {f', ...} = @{cterm-match ‹run ?f' ?s›} (rhs-of eq)
      val rule1 = Utils.timeit-msg 4 ctxt (fn - => ADD-FACT rule1: )
        (fn - => Drule.infer-instantiate ctxt [(fN, f')] rule)
      val rule2 = Utils.timeit-msg 4 ctxt (fn - => ADD-FACT rule2: )
        (fn - =>  eq COMP rule1)
      val - = Utils.timing-msg' 3 ctxt (fn - => ADD-FACT (1)) start1
    in simp-rule rule2 end),
   match-preserved-facts ctxt (map fst names) #> (fn {f, s, t, r, results,
lhs, g'=g'0, ctxt'} =>
    let
      val ctxt' = stop preserved ctxt' |> reset-cache
      val facts = facts-of ctxt'
    val - = Utils.verbose-msg 5 ctxt (fn - => facts (1):\n  ^ string-of-thms
ctxt'  (maps #thms facts))
      val start0 = Timing.start ();
      val s-eqs = maps #thms facts
        val gN = fst (dest-Var (Thm.term-of (fst (Utils.strip-comb-cterm
g'0))))
```

```
                val preserved-prop = infer-instantiate ‹f = f and s = s and r = r
and t = t in
                    cprop PRESERVED-FACTS f s r t› ctxt'
                 val ([preserved], ctxt1) = Assumption.add-assumes [preserved-prop]
ctxt'
                val return-eqs = preserved-facts-return-eqs ctxt' preserved
                val derived-props = map #pred facts |> map (fn pred =>
                     (pred, Thm.apply pred t |> Utils.Trueprop-cterm))
                val derived-facts0 = derive-facts ctxt' preserved
                fun prover ctxt =
                  let
                     val ctxt = ctxt (*|> Config.put Simplifier.simp-trace true*)
                  in
                     Method.insert-tac ctxt [preserved] 1 THEN
                     (SOLVED-DETERM-verbose preserved ctxt
                       (asm-full-simp-tac (ctxt
                         addsimps @{thms PRESERVED-FACTS-def} @ s-eqs))) 1
                  end
               val derived-facts1 = derived-props |> map-filter (fn (pred, prop) =>
                    try (fn prop => Goal.prove-internal ctxt1 [] prop (fn - => prover
ctxt1)) prop |>
                        Option.map (fn thm => ({pred = pred, thms = mksimps ctxt1
thm})))
               val derived-facts3 = map-filter (new-facts-from-defs ctxt1 return-eqs)
(derived-facts0 @ derived-facts1)
                val derived-facts = derived-facts0 @ derived-facts1 @ derived-facts3
                val derived-eqs = maps #thms derived-facts
                val derived-ariths = Utils.iariths-of-eqs (maps #thms derived-facts)
                val ctxt2 = ctxt1
                  addsimps derived-eqs
                  |> Simplifier.add-prems derived-eqs
                  |> map-states (cons t)
                  |> map-facts (K derived-facts)
                  |> Utils.add-ariths derived-ariths
                val - = Utils.timing-msg' 3 ctxt (fn - => preserved (0)) start0
                   val - = Utils.verbose-msg 4 ctxt (fn - => derived-facts (1):\n  ^
string-of-thms ctxt1  (maps #thms derived-facts))
                val eq1 = Utils.timeit-msg 4 ctxt (fn - => eq (5): )
                  (fn -  =>  Simplifier.asm-full-rewrite (start eq (5) ctxt2) lhs
                      |> STOP-conv ctxt2 |> singleton (timeit-export eq (5):  ctxt2
ctxt'))
            val - = Utils.verbose-msg 5 ctxt (fn - => (eq (5):\n  ^ Thm.string-of-thm
ctxt' eq1))
                val start1 = Timing.start ();
                val eq = eq1 |> singleton (Proof-Context.export ctxt' ctxt)
                val {g', ...} = @{cterm-match ‹run ?g' ?s›} (rhs-of eq1)
                val - = if length names <> length results then
                     error (match-preserved-facts: unequal length ^ @{make-string}
(names, results)) else ()
```

```
    val results' = map fst names ~~ results
    val g' = Utils.timeit-msg 4 ctxt (fn - => preserved lambdas: )
      (fn - => Utils.lambdas (results') g')
    val rule1 =  Utils.timeit-msg 4 ctxt (fn - => preserved rule1: )
      (fn - => Drule.infer-instantiate ctxt [(gN, g')] rule)
    val rule2 = Utils.timeit-msg 4 ctxt (fn - => preserved rule2: )
      (fn - => (rule1 OF [eq])
        |> rule-by-tactic-schematic ctxt (Method.assm-tac ctxt 1))
    val - = Utils.timing-msg' 3 ctxt (fn - => preserved (1)) start1
  in
    simp-rule rule2
  end),
 match-preserved-facts-while ctxt (map fst names) #> (fn {C, C'=C'0,
B, B'=B'0, i, s, t, r, results,  C-lhs, B-lhs, ctxt'} =>
  let
    val ctxt' = stop while ctxt' |> reset-cache
    val start0 = Timing.start ();
    (* Generalize and merge code with previous case *)
    val facts = facts-of ctxt'
    val s-eqs = maps #thms facts
     val CN = fst (dest-Var (Thm.term-of (fst (Utils.strip-comb-cterm
C'0))))
     val BN = fst (dest-Var (Thm.term-of (fst (Utils.strip-comb-cterm
B'0))))
     val preserved-prop = infer-instantiate ‹C = C and B = B  and i
= i and s = s and r = r and t = t in
         cprop PRESERVED-FACTS-WHILE C B i s r t› ctxt'
    val ([preserved], ctxt1) = Assumption.add-assumes [preserved-prop]
ctxt'
    val derived-props = map #pred facts |> map (fn pred =>
         (pred, Thm.apply pred t |> Utils.Trueprop-cterm))
    fun prover ctxt =
         Method.insert-tac ctxt [preserved] 1 THEN
       asm-full-simp-tac (ctxt addsimps @{thms PRESERVED-FACTS-WHILE-def}
@ s-eqs) 1
    val derived-facts = derived-props |> map-filter (fn (pred, prop) =>
       try (fn prop => Goal.prove-internal ctxt1 [] prop (fn - => prover
ctxt1)) prop |>
         Option.map (fn thm => ({pred = pred, thms = mksimps ctxt1
thm})))
    val derived-eqs = maps #thms derived-facts
    val derived-ariths = Utils.iariths-of-eqs (maps #thms derived-facts)

    val ctxt2 = ctxt1
      addsimps derived-eqs
      |> Simplifier.add-prems derived-eqs
      |> map-states (cons t)
      |> map-facts (K derived-facts)
      |> Utils.add-ariths derived-ariths
```

```
val - = Utils.timing-msg' 3 ctxt (fn - => while (0)) start0
  val - = Utils.verbose-msg 4 ctxt (fn - => derived-facts (2):\n  ^
string-of-thms ctxt1  (maps #thms derived-facts))
val C-eq1 = Utils.timeit-msg 3 ctxt (fn - => eq (6): )
  (fn -  =>  Simplifier.asm-full-rewrite (start eq (6) ctxt2) C-lhs
      |> STOP-conv ctxt2|> singleton (timeit-export eq (6):  ctxt2
ctxt'))
val start1 = Timing.start ();
val C'1 = (rhs-of C-eq1)
val - = if length names <> length results then
 error (match-preserved-facts-while: unequal length ^@{make-string}
(names, results)) else ()
val results' = map fst names ~~ results
val C' = Utils.lambdas (results' @ [(s, t)]) C'1
val C'-pred = Utils.lambdas [(s, t)] C'1
val ([C'-thm], ctxt3) = Assumption.add-assumes [Utils.Trueprop-cterm
C'1] ctxt2
val C'-eqs = mksimps ctxt3 C'-thm
val C'-ariths = Utils.iariths-of-eqs C'-eqs
  val ctxt4 = (ctxt3 |> map-facts (cons {pred = C'-pred, thms =
C'-eqs}))
    addsimps C'-eqs
    |> Simplifier.add-prems C'-eqs
    |> Utils.add-ariths C'-ariths
val - = Utils.timing-msg' 3 ctxt (fn - => while (1)) start1
val - = Utils.verbose-msg 5 ctxt (fn - => (adding:\n  ^string-of-thms
ctxt1 C'-eqs))
val B-eq1 = Utils.timeit-msg 4 ctxt (fn - => eq (7): )
  (fn -  =>  Simplifier.asm-full-rewrite (start eq (7) ctxt4) B-lhs
      |> STOP-conv ctxt4 |> singleton (timeit-export eq (7):  ctxt4
ctxt'))
val start2 = Timing.start ();
val {B', ...} = @{cterm-match ‹run ?B' ?t›} (rhs-of B-eq1)
val B' = Utils.lambdas (results') B'
val rule1 = Drule.infer-instantiate ctxt [(CN, C'), (BN, B')] rule
val [C-eq, B-eq] = [C-eq1, B-eq1] |> Proof-Context.export ctxt' ctxt
val - = Utils.verbose-msg 5 ctxt (fn - => (eq (6):\n  ^Thm.string-of-thm
ctxt C-eq))
val - = Utils.verbose-msg 5 ctxt (fn - => (eq (7):\n  ^Thm.string-of-thm
ctxt B-eq))
val rule2 = Utils.solve-sideconditions ctxt rule1 (
    resolve-tac ctxt [Conjunction.conjunctionI] 1 THEN
    resolve-tac ctxt [C-eq] 1 THEN
    Method.assm-tac ctxt 1 THEN
    resolve-tac ctxt [B-eq] 1 THEN
    Method.assm-tac ctxt 1 THEN
    Method.assm-tac ctxt 1
    )
val - = Utils.timing-msg' 3 ctxt (fn - => while (2)) start2
```

```
            in simp-rule rule2
            end),
          fn ct => raise TERM (monad-state-conv: unexpected: , [Thm.term-of
ct])] end )
      end
  in
    simp-rule inst-rule
  end
  handle Option.Option => Thm.reflexive ct

fun monad-state-conv' (rules: rules) ctxt ct =
  let
    val matches0 = Net.match-term rules (Thm.term-of ct)
  in
    case get-more-specific matches0 of
      NONE => Thm.reflexive ct
    | SOME rule => monad-state-conv rule ctxt ct
  end

fun monad-run-simproc rule ctxt ct =
  let
    val eq = monad-state-conv rule ctxt ct
  in
    if Utils.is-reflexive ctxt eq then
      NONE
    else
      SOME (Utils.timeit-msg 3 ctxt (fn - => STOPI: ) (fn - => @{thm STOPI}
OF [eq]))
  end

end
›


attribute-setup monad-simp-global-stop-cong = ‹
 Scan.lift (Scan.option Args.add) >> (fn - => Thm.declaration-attribute (Monad-Cong-Simp.add-global-stop-c
 || Scan.lift Args.del >> (fn - => Thm.declaration-attribute (Monad-Cong-Simp.del-global-stop-cong))
› Global 'stop' congruence rule for monad simplification

attribute-setup monad-simp-cong = ‹
let
  val split = Args.$$$ split |-- Args.colon |-- Parse.and-list Parse.short-ident
  val opt-split = Scan.optional (Scan.lift (split)) []
  fun pos-here [] = (Position.none, [])
    | pos-here ts = (Token.pos-of (hd ts), ts)
in
  Scan.lift Args.add |-- (Scan.lift pos-here -- opt-split) >> (fn (pos, splits) =>
      Thm.declaration-attribute (Monad-Cong-Simp.add-rule splits))
 || Scan.lift Args.del >> (K (Thm.declaration-attribute (Monad-Cong-Simp.del-rule)))
```

```
  || opt-split >> (fn splits =>
      Thm.declaration-attribute (Monad-Cong-Simp.add-rule splits))
end› Monad congruence theorems

attribute-setup monad-simp-derive-rule = ‹
 Scan.lift (Scan.option Args.add) >> (fn - => Thm.declaration-attribute (Monad-Cong-Simp.add-derive-rule))
 || Scan.lift Args.del >> (fn - => Thm.declaration-attribute (Monad-Cong-Simp.del-derive-rule))
› Conditional rules to derive facts in monad simplification

simproc-setup monad-run-simproc (‹run f s›) = ‹fn - => fn ctxt => fn ct =>
   Monad-Cong-Simp.matching-rule ctxt (Thm.term-of ct) |> Option.mapPartial
(fn rule =>
   let
    val - = Utils.verbose-msg 7 ctxt (fn - => monad-run-simproc:\n ^string-of-cterm
ctxt ct)
   in
     Monad-Cong-Simp.monad-run-simproc rule ctxt ct
   end)›
declare [[simproc del: monad-run-simproc]]

lemma spec-monad-ext': (⋀s. run f s ≡ run f' s) ⟹ f ≡ f'
  by (smt (verit) spec-monad-ext)

ML ‹
structure Monad-Cong-Simp =
struct
  open Monad-Cong-Simp

fun dest-spec-monadT Type ‹spec-monad E A S› = {exnT = E, regT = A, stateT
= S}
  | dest-spec-monadT T = raise TYPE(dest-spec-monadT: , [T], [])

fun gen-monad-conv {stop} ctxt ct =
  let
    val cT = Thm.ctyp-of-cterm ct
    val {stateT, exnT, regT} = Thm.typ-of-cterm ct |> dest-spec-monadT
    val {rules, ...} = Data.get (Context.Proof ctxt)
    val ([s], ctxt1) = Utils.fix-variant-cfrees [(s, stateT)] ctxt
    val ctxt2 = map-states (K [s]) ctxt1
    val lhs = infer-instantiate ‹f = ct and s = ‹s›
         in cterm ‹run f s› for f::‹('e::default, 'a, 's) spec-monad›› ctxt
    val eq = monad-state-conv' rules ctxt2 lhs
    val eq' = singleton (Proof-Context.export ctxt2 ctxt) eq
    val eq' = @{thm spec-monad-ext'} OF [eq']
  in
    if stop then (@{thm STOPI} OF [eq']) |> Drule.zero-var-indexes else eq'
  end

end
```

›

We make a global setup here, as the locale one within *heap-typing-state* fails. The reason is that the pattern is not taken into account in simproc equality when inserted into the net. Hence multiple instances (e.g. for *globals* and *lifted-globals*) are considered a duplicate when adding them both to the net but the simplifier does not match the pattern for *globals* on a *lifted-globals* instance. In the upcoming Isabelle2024 this issue should be resolved.

```
simproc-setup unchanged-typing-spec-monad (‹c • s ?{|λr. heap-typing-state.unchanged-typing-on
htd 𝒮 s|}›  ) = ‹
  let
    val is-assume-stack-alloc = Match-Cterm.switch [
          @{cterm-match (fo) typ-heap-typing.assume-stack-alloc ?r ?w ?ht ?ht-upd
?S ?n ?X • ?s ?{|?Q|}} #>
            (fn - => true),
          (fn - => false)]

    val active = Config.declare-bool (active, Position.none) (K false);
    val prop-to-meta-eq = @{lemma ‹P ⟹ (P ≡ True)› for P by auto}
    fun try-prove ctxt prop =
      case Monad-Cong-Simp.lookup-cache ctxt prop of
        [] =>
        let
            val ctxt' = Context.proof-map (Named-Rules.add-thm @{named-rules
runs-to-vcg} @{thm runs-to-partial-case-prod}) ctxt
        in
       case try (Goal.prove-internal ctxt' [] prop) (fn - => Unchanged-Typing.unchanged-typing-tac
NONE ctxt' ) of
            NONE => (warning (unchanged-typing proof failed: ^ string-of-cterm
ctxt prop); NONE)
          | SOME thm => (Monad-Cong-Simp.add-to-cache ctxt thm; SOME thm)
        end
    | (thm::-) => (Utils.verbose-msg 4 ctxt (fn - => cache hit: ^ Thm.string-of-thm
ctxt thm); SOME thm)
  in
    fn phi => fn ctxt => fn ct =>
      if Config.get ctxt active orelse (is-assume-stack-alloc ct) then NONE else
      let
          val - = Utils.verbose-msg 4 ctxt (fn - => unchanged-typing-spec-monad
invoked on:\n ^ @{make-string} ct)
      in
      try-prove (Config.put active true ctxt) instantiate ‹P = ct in cterm ‹Trueprop
P› for P›
        |> Option.map (fn thm => prop-to-meta-eq OF [thm])
      end

  end
›
```

declare [[*simproc del*: *unchanged-typing-spec-monad*]]

setup ‹
let
val (*unchanged-typing-simproc-name*, -) =
Simplifier.check-simproc @{*context*} (*unchanged-typing-spec-monad*, ***here***)
in
Context.theory-map (*Monad-Cong-Simp.add-simproc unchanged-typing-simproc-name*)
end
›

simproc-setup *spec-monad-simproc* (‹*f*::('*a*, '*b*, '*c*) *spec-monad*›) = ‹*fn* - => *fn* *ctxt* =>
if Monad-Cong-Simp.monad-simp-active ctxt then K NONE else
Match-Cterm.switch [
@{*cterm-match STOP* -} #> (*fn* - => *NONE*),
fn ct =>
let
val - = *Utils.verbose-msg* 7 *ctxt* (*fn* - => (*spec-monad-simproc*:*n* ^ *string-of-cterm ctxt ct*))
val eq = *Monad-Cong-Simp.gen-monad-conv* {*stop*=*true*} *ctxt ct*
val - = *Utils.verbose-msg* 7 *ctxt* (*fn* - => (*spec-monad-simproc eq*:*n* ^ *Thm.string-of-thm ctxt eq*))
in
SOME eq
end]›
declare [[*simproc del*: *spec-monad-simproc*]]

ML ‹
structure Monad-Cong-Simp =
struct
open Monad-Cong-Simp

fun prepare-simpset max-depth ctxt =
let
val {*stop-congs*, *simprocs*, ...} = *Monad-Cong-Simp.Data.get* (*Context.Proof* *ctxt*)
val simprocs' = *map* (*snd o Simplifier.check-simproc ctxt o rpair Position.none*) *simprocs*
val visible = *Context-Position.is-visible ctxt*
val monad-cong-simps = *Named-Theorems.get ctxt* @{*named-theorems monad-simp-simp*}
val run-spec-monad = *Named-Theorems.get ctxt* @{*named-theorems run-spec-monad*}

val state-fold-congs = *Named-Theorems.get ctxt* @{*named-theorems state-fold-congs*}
val recursive-records-fold-congs = *Named-Theorems.get ctxt* @{*named-theorems recursive-records-fold-congs*}

```
    val ctxt' = (ctxt |> Context-Position.set-visible false
      |> Simplifier.add-cong @{thm STOP-cong}
      |> fold Simplifier.add-cong (state-fold-congs @ recursive-records-fold-congs)
      |> fold Simplifier.add-cong stop-congs)
    addsimprocs [@{simproc spec-monad-simproc}, @{simproc monad-run-simproc},
@{simproc Arrays.fold-update}] @ simprocs'
      delsimps @{thms return-bind bind-return Spec-Monad.run-case-prod-distrib}
@ run-spec-monad
      addsimps monad-cong-simps
      addsimps @{thms Arrays.fupdate-def [symmetric]}
      |> Context-Position.set-visible visible
      |> Config.map simp-depth-limit (K (max-depth + 20))
  in ctxt' end

fun monad-simp-tac ctxt = SUBGOAL (fn (goal, i) =>
  let
    val depth = strip-comb-depth-of-term goal
    val ctxt' = prepare-simpset depth ctxt
  in
    (asm-full-simp-tac ctxt' THEN-ALL-NEW
     (Utils.verbose-print-subgoal-tac 7 before monad-simp polish ctxt THEN'
      full-simp-tac (Simplifier.clear-simpset ctxt addsimps @{thms STOP-def pol-
ish}))) i
  end)

fun gen-monad-simplify simplify ctxt thm =
  let
    val depth = strip-comb-depth-of-term (Thm.prop-of thm)
    val ctxt' = prepare-simpset depth ctxt
  in
    thm |> simplify ctxt' |>
    simplify (Simplifier.clear-simpset ctxt addsimps @{thms STOP-def polish})
  end

val monad-simplify = gen-monad-simplify simplify
val monad-asm-full-simplify = gen-monad-simplify asm-full-simplify

fun monad-asm-full-rewrite ctxt ct =
  let
    val depth = strip-comb-depth-of-term (Thm.term-of ct)
    val ctxt' = prepare-simpset depth ctxt
  in
    ct |> (Simplifier.asm-full-rewrite ctxt' then-conv
   Simplifier.full-rewrite (Simplifier.clear-simpset ctxt addsimps @{thms STOP-def
polish}))
  end

fun gen-monad-simplify-import simplify ctxt thm =
```

```
  let
    val ctxt' = Variable.declare-thm thm ctxt
    val ((-, [thm']), ctxt'') = Variable.import true [Thm.transfer' ctxt thm] ctxt'
  in
    thm' |> simplify ctxt'' |> singleton (Variable.export ctxt'' ctxt')
    |> zero-var-indexes
  end

val monad-asm-full-simplify-import = gen-monad-simplify-import monad-asm-full-simplify
val monad-simplify-import = gen-monad-simplify-import monad-simplify

end
›

method-setup monad-simp = ‹Method.sections Simplifier.simp-modifiers >>
    (K (fn ctxt => METHOD (fn facts => HEADGOAL (
      Method.insert-tac ctxt facts THEN' (CHANGED-PROP oo Monad-Cong-Simp.monad-simp-tac)
ctxt))))›
  Simplification for Spec Monad with custom congruence rules



context open-types-heap-typing-state
begin
lemma PRESERVED-FACTS-unchanged-typing-ptr-valid-preserved[monad-simp-simp]:
  reaches f s r t ⟹ ptr-valid (heap-typing s) p ⟹ f • s ?{λ- t. unchanged-typing-on
UNIV s t} ⟹
 ptr-valid (heap-typing t) p
  by (rule reaches-unchanged-typing-ptr-valid-preserved)

lemma PRESERVED-FACTS-WHILE-unchanged-typing-ptr-valid-preserved[monad-simp-simp]:
  assumes reach: whileLoop-unroll-reachable C B i s r t
  assumes valid: ptr-valid (heap-typing s) p
  assumes B: ⋀r t. C r t ⟹ (B r) • t ?{λ-. unchanged-typing-on UNIV t}
  shows ptr-valid (heap-typing t) p
proof −
  have unchanged-typing-on UNIV s t
    using reach
  proof induction
    case (step b t X c u)
    with B[of b t] unchanged-typing-on-trans[of UNIV s t u]
    show ?case
      by (auto simp add: runs-to-partial-holds-partial-def )
  qed simp

  with valid show ?thesis
    by (simp add: unchanged-typing-on-UNIV-iff)
qed
```

end

lemma *PRESERVED-FACTS-run-bind-cong*[*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹ (⋀*t r*. *PRESERVED-FACTS f' s* (*Result r*) *t* ⟹ *run* (*g r*) *t* = *run* (*g' r*) *t*) ⟹
(*run* (*bind f g*) *s*) = (*run* (*bind f' g'*) *s*)
apply (*rule Spec-Monad.run-bind-cong*)
apply (*auto simp add*: *PRESERVED-FACTS-def runs-to-partial-def-old reaches-def*)
done

lemma *ADD-FACT-run-bind-guard-cong*[*monad-simp-cong*]:
P s = *P' s* ⟹ (*ADD-FACT P' s* ⟹ *run f s* = *run f' s*) ⟹
(*run* (*bind* (*guard P*) (λ-. *f*)) *s*) = (*run* (*bind* (*guard P'*) (λ-. *f'*)) *s*)
apply (*simp add*: *ADD-FACT-def*)
apply (*rule run-bind-cong*)
apply (*simp add*: *run-guard*)
apply (*auto simp add*: *run-guard if-split-asm runs-to-partial-def-old*)
done

lemma *PRESERVED-FACTS-WHILE-run-whileLoop-cong*[*monad-simp-cong split*: *B* **and** *B'* **and** *C* **and** *C'*]:
assumes *i*: *i* = *i'*
assumes *C-B*[*unfolded PRESERVED-FACTS-WHILE-def*]:
⋀*t r*. *PRESERVED-FACTS-WHILE C B i s r t* ⟹ (*C r t* ≡ *C' r t*) &&& (*C' r t* ⟹ (*run* (*B r*) *t* ≡ *run* (*B' r*) *t*))
shows *run* (*whileLoop C B i*) *s* = *run* (*whileLoop C' B' i'*) *s*
proof −
from *C-B*
have *C*: ⋀*r s'*. *whileLoop-unroll-reachable C B i s r s'* ⟹ *C r s'* = *C' r s'* **by** *blast*
from *C-B*
have *B*: ⋀*r s'*. *whileLoop-unroll-reachable C B i s r s'* ⟹ *C' r s'* ⟹ *run* (*B r*) *s'* = *run* (*B' r*) *s'*
by *blast*
with *C* **have** ⋀*r s'*. *whileLoop-unroll-reachable C B i s r s'* ⟹ *C r s'* ⟹ *run* (*B r*) *s'* = *run* (*B' r*) *s'*
by *simp*
from *run-whileLoop-unroll-reachable-cong* [*OF C this*]
show *?thesis* **by** (*simp add*: *i*)
qed

lemma *whileLoop-condition-only-cong*[*monad-simp-split-tuple-cong*]:
C = *C'* ⟹ *whileLoop C B I* = *whileLoop C' B I*
by *simp*

lemma *PRESERVED-FACTS-run-bind-exception-or-result-cong*[*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹ (⋀*t r*. *PRESERVED-FACTS f' s r t* ⟹ *run* (*g r*) *t* = *run* (*g' r*) *t*) ⟹

(*run* (*bind-exception-or-result f g*) *s*) = (*run* (*bind-exception-or-result f' g'*) *s*)
apply (*rule Spec-Monad.run-bind-exception-or-result-cong*)
apply (*auto simp add*: *PRESERVED-FACTS-def runs-to-partial-def-old reaches-def*)
done

lemma *PRESERVED-FACTS-run-on-exit'-cong*[*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹ (⋀*t r*. *PRESERVED-FACTS f' s r t* ⟹ *run g t* = *run g' t*) ⟹
(*run* (*on-exit' f g*) *s*) = (*run* (*on-exit' f' g'*) *s*)
unfolding *on-exit'-def*
apply (*rule PRESERVED-FACTS-run-bind-exception-or-result-cong*)
apply *assumption*
apply (*rule PRESERVED-FACTS-run-bind-cong*)
apply *assumption*
apply *simp*
done

lemma *PRESERVED-FACTS-run-on-exit-cong*[*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹
(⋀*t r*. *PRESERVED-FACTS f' s r t* ⟹ *run* ((*state-select g*::('*e*::*default*, *unit*, '*s*) *spec-monad*)) *t* = *run* (*state-select g'*) *t*) ⟹
(*run* (*on-exit f g*::('*e*::*default*, '*a*, '*s*) *spec-monad*) *s*) = (*run* (*on-exit f' g'*) *s*)
unfolding *on-exit-def*
apply (*rule PRESERVED-FACTS-run-on-exit'-cong*)
apply *assumption*
apply *simp*
done

lemma *ADD-FACT-run-condition-cong* [*monad-simp-cong*]:
assumes *c s* = *c' s*
assumes *ADD-FACT c' s* ⟹ *run f s* = *run f' s*
assumes *ADD-FACT* (λ*s*. ¬ *c' s*) *s* ⟹ *run g s* = *run g' s*
shows *run* (*condition c f g*) *s* = *run* (*condition c' f' g'*) *s*
using *assms*
by (*auto simp add*: *ADD-FACT-def run-condition*)

lemma *ADD-FACT-run-when* [*monad-simp-cong*]:
assumes *c* = *c'*
assumes *ADD-FACT* (λ-. *c'*) *s* ⟹ *run f s* = *run f' s*
shows *run* (*when c f*) *s* = *run* (*when c' f'*) *s*
using *assms*
by (*auto simp add*: *ADD-FACT-def when-def run-condition*)

lemma *PRESERVED-FACTS-run-catch-cong* [*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹ (⋀*t e*. *PRESERVED-FACTS f' s* (*Exn e*) *t* ⟹ *run* (*g e*) *t* = *run* (*g' e*) *t*) ⟹
(*run* (*catch f g*) *s*) = (*run* (*catch f' g'*) *s*)
apply (*simp add*: *run-catch PRESERVED-FACTS-def reaches-def*)
apply (*rule bind-post-state-cong*)

apply (*auto split*: *xval-splits*)
done

lemma *run-finally-cong* [*monad-simp-cong*]:
assumes *run f s* = *run f' s*
shows *run* (*finally f*) *s* = *run* (*finally f'*) *s*
using *assms*
by (*simp add*: *run-finally*)

lemma *run-liftE-cong* [*monad-simp-cong*]:
assumes *run f s* = *run f' s*
shows *run* (*liftE f*) *s* = *run* (*liftE f'*) *s*
using *assms*
by (*simp add*: *run-liftE*)

lemma *run-guard-cong* [*monad-simp-cong*]:
assumes *P s* = *P' s*
shows *run* (*guard P*) *s* = *run* (*guard P'*) *s*
using *assms*
by (*simp add*: *run-guard*)

lemma *run-try-cong* [*monad-simp-cong*]:
assumes *run f s* = *run f' s*
shows *run* (*try f*) *s* = *run* (*try f'*) *s*
using *assms*
by (*simp add*: *run-try*)

lemma *run-case-prod-cong*[*monad-simp-cong*]: *RUN-CASE-PROD f x s f' x'* ⟹

(*run* (*case-prod f x*) *s*) = (*run* (*case-prod f' x'*) *s*)
by (*cases x*) (*auto simp add*: *RUN-CASE-PROD-def*)

lemma *run-bind-when-throw-cong* [*monad-simp-cong*]:
c = *c'* ⟹ *e* = *e'* ⟹ (*ADD-FACT* (λ-. ¬ *c'*) *s* ⟹ *run f s* = *run f' s*) ⟹
run (*bind* (*when c* (*throw e*)) (λ-. *f*)) *s* = *run* (*bind* (*when c'* (*throw e'*)) (λ-. *f'*))
s
by (*auto simp add*: *ADD-FACT-def*)

ML ‹
Monad-Cong-Simp.print-rules (*Context.Proof* @{*context*})
›

lemma *with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:
typ-heap-typing.with-fresh-stack-ptr r w heap-typing heap-typing-upd 𝒮 *n init f* ≡
typ-heap-typing.with-fresh-stack-ptr r w heap-typing heap-typing-upd 𝒮 *n init f*
by (*simp*)

lemma *assume-with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:

typ-heap-typing.assume-with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f* ≡
typ-heap-typing.assume-with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f*
by (*simp*)

lemma *guard-with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:
typ-heap-typing.guard-with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f* ≡
typ-heap-typing.guard-with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f*
by (*simp*)

context *typ-heap-simulation-open-types-stack*
begin

lemma *PRESERVED-FACTS-assume-stack-alloc-ptr-valid-preserved*[*monad-simp-simp*]:
reaches (*assume-stack-alloc n init*) *s x t* ⟹ *ptr-valid* (*heap-typing s*) (*p*::*'b*::*mem-type ptr*) ⟹
typ-uinfo-t TYPE(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*) ⟹
ptr-valid (*heap-typing t*) *p*
apply (*clarsimp simp add*: *stack-ptr-acquire-def heap-typing-fold-upd-write reaches-def assume-stack-alloc-def*)
using *stack-allocs-ptr-valid-cases2* **by** *blast*

lemma *PRESERVED-FACTS-with-fresh-stack-ptr-cong*[*monad-simp-cong split*: *f* **and** *f'*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'d*, *'t*) *spec-monad*
assumes ⋀*t p*. *PRESERVED-FACTS* (*assume-stack-alloc n init*::(*'e*::*default*, *'a ptr*, *'t*) *spec-monad*) *s* (*Result p*) *t* ⟹
run (*f p*) *t* = *run* (*f' p*) *t*
shows *run* (*with-fresh-stack-ptr n init f*) *s* = *run* (*with-fresh-stack-ptr n init f'*) *s*
apply (*simp add*: *with-fresh-stack-ptr-def PRESERVED-FACTS-def ADD-FACT-def*)
apply (*rule run-bind-cong*)
subgoal by *simp*
subgoal
apply (*clarsimp simp add*: *runs-to-partial-def-old on-exit-def on-exit'-def*)
apply (*rule run-bind-exception-or-result-cong*)
subgoal
apply (*rule assms*)
apply (*simp add*: *PRESERVED-FACTS-def ADD-FACT-def*)
done
subgoal by *simp*
done
done

lemma *PRESERVED-FACTS-assume-with-fresh-stack-ptr-cong*[*monad-simp-cong*

split: *f* **and** *f'*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'d*, *'t*) *spec-monad*
assumes ⋀*t p*. *PRESERVED-FACTS* (*assume-stack-alloc n init*::(*'e*::*default*, *'a ptr*, *'t*) *spec-monad*) *s* (*Result p*) *t* ⟹
run (*f p*) *t* = *run* (*f' p*) *t*
shows *run* (*assume-with-fresh-stack-ptr n init f*) *s* = *run* (*assume-with-fresh-stack-ptr n init f'*) *s*
apply (*simp add*: *assume-with-fresh-stack-ptr-def PRESERVED-FACTS-def ADD-FACT-def*)
apply (*rule run-bind-cong*)
subgoal by *simp*
subgoal
apply (*clarsimp simp add*: *runs-to-partial-def-old on-exit-def on-exit'-def*)
apply (*rule run-bind-exception-or-result-cong*)
subgoal
apply (*rule assms*)
apply (*simp add*: *PRESERVED-FACTS-def ADD-FACT-def*)
done
subgoal by *simp*
done
done

lemma *PRESERVED-FACTS-guard-with-fresh-stack-ptr-cong*[*monad-simp-cong split*: *f* **and** *f'*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'d*, *'t*) *spec-monad*
assumes ⋀*t p*. *PRESERVED-FACTS* (*assume-stack-alloc n init*::(*'e*::*default*, *'a ptr*, *'t*) *spec-monad*) *s* (*Result p*) *t* ⟹
run (*f p*) *t* = *run* (*f' p*) *t*
shows *run* (*guard-with-fresh-stack-ptr n init f*) *s* = *run* (*guard-with-fresh-stack-ptr n init f'*) *s*
apply (*simp add*: *guard-with-fresh-stack-ptr-def PRESERVED-FACTS-def ADD-FACT-def*)
apply (*rule run-bind-cong*)
subgoal by *simp*
subgoal
apply (*clarsimp simp add*: *runs-to-partial-def-old on-exit-def on-exit'-def*)
apply (*rule run-bind-exception-or-result-cong*)
subgoal
apply (*rule assms*)
apply (*simp add*: *PRESERVED-FACTS-def ADD-FACT-def*)
done
subgoal by *simp*
done
done

lemma *ptr-valid-assume-stack-alloc* [*monad-simp-derive-rule*]:
PRESERVED-FACTS (*assume-stack-alloc n init*::(*'e*::*default*, *'a ptr*, *'t*) *spec-monad*) *s* (*Result p*) *t* ⟹
ADD-FACT (λ*t*. *PTR-VALID*(*'a*) (*heap-typing t*) *p*) *t*
apply (*simp add*: *PRESERVED-FACTS-def ADD-FACT-def*)
apply (*auto simp add*: *stack-ptr-acquire-def heap-typing-fold-upd-write*

reaches-def
assume-stack-alloc-def root-ptr-valid-ptr-valid elim!: *stack-allocs-cases*)
done

end

attribute-setup *augment-rule-format* = ‹
Scan.succeed (*Thm.rule-attribute* []
(*K* (*Monad-Cong-Simp.augment-rule-format*)))›

attribute-setup *augment-rule* = ‹
let
fun mk-mixed-attribute ctxt attribs-src =
let
val attribs = *map* (*Attrib.attribute ctxt*) *attribs-src*
val attrib = *Thm.mixed-attribute* (*fold* (*fn attrib* => *fn* (*context*, *thm*) =>
(*swap* (*Thm.apply-attribute attrib thm context*))) *attribs*)
in attrib end
in
Scan.succeed (*mk-mixed-attribute* @{*context*} @{*attributes* [*augment-rule-format*,
monad-simp-augment-rule]})
end›

attribute-setup *monad-simplified* = ‹
Scan.succeed (*Thm.rule-attribute* []
(*fn context* => *Monad-Cong-Simp.monad-simplify-import* (*Context.proof-of context*)))›

lemmas *word-augment-rules*[*augment-rule*] =
word-le-nat-alt [*THEN iffD1*]
word-less-nat-alt [*THEN iffD1*]
word-le-def [*THEN iffD1*]
word-less-def [*THEN iffD1*]
word-sle-def [*THEN iffD1*]
word-sless-alt [*THEN iffD1*]
lemmas *word-monad-simp*[*monad-simp-simp*] =
word-le-nat-alt [*THEN iffD2*]
word-less-nat-alt [*THEN iffD2*]
word-le-def [*THEN iffD2*]
word-less-def [*THEN iffD2*]
word-sle-def [*THEN iffD2*]
word-sless-alt [*THEN iffD2*]

thm *monad-simp-augment-rule*
thm *monad-simp-simp*

method *array-index-to-ptr-arith-simp* **uses** *simp cong* =
(*use TrueI*[*array-bound-mksimps*, *simproc add*: *field-lvalue-unfold*]
in ‹*monad-simp*
simp add:*field-lvalue-array-index′ simp*
simp del: *field-lvalue-append*
cong: *if-cong cong*›)

end

theory *Runs-To-VCG-StackPointer*
imports
HeapLift
Refines-Spec
begin

definition *distinct-span xs* ≡
distinct-prop (λ(*v1*, *s1*) (*v2*, *s2*). *disjnt* {*v1* ..+ *s1*} {*v2* ..+ *s2*}) *xs*

definition *distinct-spans xs* ≡
distinct-prop (λ(*v1*, *n1*, *s1*) (*v2*, *n2*, *s2*). *disjnt* {*v1* ..+ *n1* ∗ *s1*} {*v2* ..+ *n2* ∗ *s2*}) *xs*

lemma *distinct-span-spans-conv*:
distinct-span xs ⟷ *distinct-spans* (*map* (λ(*v*, *s*). (*v*, *1*, *s*)) *xs*)
unfolding *distinct-span-def distinct-spans-def*
by (*clarsimp simp*: *distinct-prop-map case-prod-beta intro*!: *distinct-prop-iff*)

definition *lvp-distinct xs* ≡
distinct-spans (*map* (λ(*d*, *v*, *n*, *s*). (*v*, *n*, *s*)) *xs*) ∧
distinct (*map* (λ(*d*, *v*, *n*, *s*). (*d*, *v*)) *xs*) ∧
sorted-wrt (λ(*d1*, -) (*d2*, -). *stack-free d1* ⊆ *stack-free d2*) *xs* ∧
(∀(*d*, *v*, *n*, *s*)∈*set xs*.
disjnt {*v* ..+ *n* ∗ *s*} (*stack-free d*) ∧ *s* > *0*)

lemma *lvp-distinct-pairwise-disjnt*:
⟦ *lvp-distinct xs*; *xs* ! *m1* = (*d1*, *v1*, *n1*, *s1*); *xs* ! *m2* = (*d2*, *v2*, *n2*, *s2*); *m1* < *m2*; *m2* < *length xs* ⟧ ⟹
disjnt {*v1* ..+ *n1* ∗ *s1*} {*v2* ..+ *n2* ∗ *s2*}
unfolding *lvp-distinct-def*
apply (*clarsimp*)+
apply (*simp add*: *distinct-spans-def distinct-prop-map case-prod-beta*)
apply (*drule*(*2*) *distinct-prop-nth*)
by *auto*

lemma *lvp-distinct-spansD*: *lvp-distinct xs* ⟹
distinct-spans (*map* (λ(*d*, *v*, *n*, *s*). (*v*, *n*, *s*)) *xs*)
by (*simp add*: *lvp-distinct-def*)

named-theorems *stack-ptr-simps*

context *stack-simulation-heap-typing* **begin**

lemmas [*stack-ptr-simps*] =
stack-ptr-acquire-def
stack-ptr-release-def
write-same-ZERO
stack-releases-stack-allocs-inverse
root-ptr-valid-ptr-valid

lemma *runs-to-guard-with-fresh-stack-ptr*[*runs-to-vcg*]:
assumes *f*[*runs-to-vcg*]: $\bigwedge$*d ptr vs.*
(*ptr*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*heap-typing s*) ⟹
vs ∈ *init s* ⟹
length vs = *n* ⟹
(∀ *i*∈{*0*..<*n*}. *r s* (*ptr* +$_p$ *int i*) = *ZERO*('*a*)) ⟹
(∀ *i*∈{*0*..<*n*}. *root-ptr-valid d* (*ptr* +$_p$ *int i*)) ⟹
(*f ptr*) ∙ (*stack-ptr-acquire n vs ptr d s*)
{|λ*r u*. (∀ *i*<*n*. *root-ptr-valid* (*heap-typing u*) (*ptr* +$_p$ *int i*)) ∧ *Q r* (*stack-ptr-release n ptr u*)|}
shows
(*guard-with-fresh-stack-ptr n init f*) ∙ *s* {| *Q* |}
apply (*unfold guard-with-fresh-stack-ptr-def assume-stack-alloc-def*)
apply *runs-to-vcg*
apply (*auto elim*: *stack-allocs-cases*)
done

lemma *runs-to-with-fresh-stack-ptr*[*runs-to-vcg*]:
assumes *f*[*runs-to-vcg*]:$\bigwedge$*d ptr vs.*
(*ptr*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*heap-typing s*) ⟹
vs ∈ *init s* ⟹
length vs = *n* ⟹
(∀ *i*∈{*0*..<*n*}. *r s* (*ptr* +$_p$ *int i*) = *ZERO*('*a*)) ⟹
(∀ *i*∈{*0*..<*n*}. *root-ptr-valid d* (*ptr* +$_p$ *int i*)) ⟹
(*f ptr*) ∙ (*stack-ptr-acquire n vs ptr d s*) {|λ*r u*. *Q r* (*stack-ptr-release n ptr u*)|}
shows
(*assume-with-fresh-stack-ptr n init f*) ∙ *s* {| *Q* |}
(*with-fresh-stack-ptr n init f*) ∙ *s* {| *Q* |}
subgoal
apply (*unfold assume-with-fresh-stack-ptr-def assume-stack-alloc-def*)
apply (*runs-to-vcg*)
apply (*auto elim*: *stack-allocs-cases simp add*: *singleton-iff*)
done
subgoal
apply (*unfold with-fresh-stack-ptr-def on-exit-def assume-stack-alloc-def*)
apply (*runs-to-vcg*)

apply (*auto elim*: *stack-allocs-cases simp add*: *singleton-iff*)
done
done

lemma *lvp-distinct-singleton-local-with-params*:
fixes *p*::'*a ptr*
assumes (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) *d*
assumes *root-ptr-valid d' p*
assumes *distinct-span params*
assumes ∀(*v*, *s*) ∈ *set params*. *disjnt* {*v* ..+ *s*} (*stack-free d*)
assumes ∀(*v*, *s*) ∈ *set params*. *s* > *0*
shows *lvp-distinct* ((*d'*, *ptr-val p*, *n*, *size-td* (*typ-uinfo-t TYPE*('*a*)))#(*map* (λ(*v*, *s*). (*d*, *v*, *1*, *s*)) *params*))
unfolding *lvp-distinct-def*
apply (*clarsimp simp*: *case-prod-beta distinct-spans-def assume-stack-alloc-def*)
apply (*intro conjI*, *clarsimp simp*: *distinct-prop-map*)
subgoal for *v t*
using *assms*(*4*) **apply** *clarsimp*
apply (*rule disjnt-subset1* [**where** *X*=*stack-free d*])
apply (*auto simp*: *disjnt-comm* [*simplified disjnt-def*] *disjnt-def*)[*1*]
apply (*simp add*: *size-of-fold*)
by (*rule stack-allocs-stack-subset-stack-free* [*OF assms*(*1*), *simplified*])
subgoal
apply (*clarsimp simp*: *distinct-prop-map*)
apply (*rule distinct-prop-iff* [*THEN iffD1*, *OF* - *assms*(*3*) [*simplified distinct-span-def*]])
by *clarsimp*
subgoal
apply (*clarsimp*)
by (*contradiction*, *rule stack-allocs-neq* [*OF assms*(*1*)], *erule sym*)
subgoal
apply (*rule distinct-iff-distinct-prop-ne* [*THEN iffD2*])
apply (*clarsimp simp*: *distinct-prop-map*)
apply (*rule distinct-prop-impl* [*OF* - *assms*(*3*) [*simplified distinct-span-def*]])
using *assms*(*5*) **apply** *clarsimp*
subgoal for *t v t'*
apply (*frule*(*1*) *bspec* [**where** *x*=(*v*, *t*)], *drule*(*1*) *bspec* [**where** *x*=(*v*, *t'*)])
by (*auto simp*: *intvl-start-inter disjnt-def*)
done
subgoal
apply (*rule ssubst* [*OF stack-free-stack-allocs* [*OF assms*(*1*)]])
by *clarsimp*
subgoal apply (*clarsimp simp*: *sorted-wrt-map*)
apply (*rule sorted-wrt-mono-rel* [**where** *P*=λ*x y*. *stack-free d* ⊆ *stack-free d*])
apply (*clarsimp*)
by (*simp*)
subgoal
apply (*simp add*: *size-of-fold disjnt-def*)
by (*rule fresh-ptrs-stack-free-disjunct* [*OF assms*(*1*), *simplified*])

subgoal
by (*simp add*: *size-of-fold*)
subgoal
using *assms*(*4*,*5*) **by** (*auto simp*: *disjnt-def dest*!: *bspec*)
done

lemma *runs-to-init-local-variables-and-parameters*:
assumes
$n > 0$
distinct-span params
$\forall (v,\ sz) \in$ *set params*. *disjnt* $\{v..+sz\}$ (*stack-free* (*heap-typing s*))
$\forall (v,\ sz) \in$ *set params*. $0 < sz$
$\bigwedge$*d ptr vs*.
(*ptr*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*heap-typing s*) ⟹
vs ∈ *init s* ⟹
length vs = *n* ⟹
$\forall i \in \{0..<n\}$. *r s* (*ptr* $+_p$ *int i*) = *ZERO*('*a*) ⟹
$\forall i \in \{0..<n\}$. *root-ptr-valid d* (*ptr* $+_p$ *int i*) ⟹
lvp-distinct ((*d*, *ptr-val ptr*, *n*, *size-td* (*typ-uinfo-t TYPE*('*a*)))#(*map* (λ(*v*, *sz*). (*heap-typing s*, *v*, *1*, *sz*)) *params*)) ⟹
f ptr ∙ (*stack-ptr-acquire n vs ptr d s*) {| λ*r u*. *Q r* (*stack-ptr-release n ptr u*) |}
shows *assume-with-fresh-stack-ptr n init f* ∙ *s* {| *Q* |}
apply (*rule runs-to-with-fresh-stack-ptr*)
apply (*rule assms*(*5*))
apply (*assumption*)+
apply (*rule lvp-distinct-singleton-local-with-params*)
apply (*simp add*: *assms*)
by (*drule bspec* [**where** *x*=*0*], *simp*, *rule assms*(*1*), *simp add*: *ptr-add-def*) (*rule assms*)+

lemma *lvp-distinct-add-stack-allocs*:
fixes *p*::'*a ptr*
assumes (*p*, *d*′) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*fst* (*hd xs*))
assumes $n > 0$
assumes *root-ptr-valid d*′ *p*
assumes *lvp-distinct xs*
shows *lvp-distinct* ((*d*′, *ptr-val p*, *n*, *size-td* (*typ-uinfo-t TYPE*('*a*)))#*xs*)
unfolding *lvp-distinct-def*
apply (*clarsimp simp*: *case-prod-beta distinct-spans-def*)
apply (*intro conjI*, *clarsimp*)
subgoal for *d v n t*
using *assms*(*4*) **apply** (*clarsimp simp*: *lvp-distinct-def*)
apply (*rule disjnt-subset1* [**where** *X*=*stack-free d*])
subgoal by (*auto simp*: *disjnt-comm* [*simplified disjnt-def*] *disjnt-def*)[*1*]
apply (*rule subset-trans*)
apply (*simp add*: *size-of-fold*)
apply (*rule stack-allocs-stack-subset-stack-free* [*OF assms*(*1*), *simplified*])
apply (*drule sorted-wrt-hd-min* [*rotated*], *clarsimp*)

by (*drule*(*1*) *bspec*, *auto*)
subgoal using *assms*(*4*) **by** (*auto simp*: *lvp-distinct-def distinct-spans-def*)
subgoal
using *assms*(*4*) **apply** (*clarsimp simp*: *lvp-distinct-def*)
apply (*drule*(*1*) *bspec*, *clarsimp*)
apply (*rule disjoint-no-subset*)
apply (*rule fresh-ptrs-stack-free-disjunct* [*OF assms*(*1*)], *simp add*: *assms*(*2*) *intvl-non-zero-non-empty*)
apply (*rule subset-trans*)
apply (*rule stack-allocs-stack-subset-stack-free* [*OF assms*(*1*)])
apply (*drule sorted-wrt-hd-min* [*rotated*], *clarsimp*)
by (*drule* (*1*) *bspec*, *auto*)
subgoal
using *assms*(*4*) **by** (*clarsimp simp*: *lvp-distinct-def*)
apply (*rule ballI*)
subgoal for *x*
apply (*cases x*; *simp*)
subgoal for *d v′ t′*
apply (*rule subset-trans*)
apply (*rule ssubst* [*OF stack-free-stack-allocs* [*OF assms*(*1*)]])
apply (*rule Diff-subset*)
using *assms*(*4*) **apply** (*clarsimp simp*: *lvp-distinct-def*)
apply (*drule sorted-wrt-hd-min* [*rotated*], *clarsimp*)
by (*drule*(*1*) *bspec*, *auto*)
done
subgoal using *assms*(*4*) **by** (*auto simp*: *lvp-distinct-def*)
subgoal
apply (*simp add*: *size-of-fold disjnt-def*)
by (*rule fresh-ptrs-stack-free-disjunct* [*OF assms*(*1*), *simplified*])
subgoal by (*simp add*: *size-of-fold*)
subgoal using *assms*(*4*) **by** (*auto simp*: *lvp-distinct-def case-prod-beta*)
done

lemma *runs-to-add-local-variable*:
assumes
lvp-distinct xs
heap-typing s = *fst* (*hd xs*)
n > *0*
⋀*d ptr vs*.
(*ptr*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*(*′a*) (*heap-typing s*) ⟹
vs ∈ *init s* ⟹
length vs = *n* ⟹
∀ *i*∈{*0*..<*n*}. *r s* (*ptr* +$_p$ *int i*) = *ZERO*(*′a*) ⟹
∀ *i*∈{*0*..<*n*}. *root-ptr-valid d* (*ptr* +$_p$ *int i*) ⟹
lvp-distinct ((*d*, *ptr-val ptr*, *n*, *size-td* (*typ-uinfo-t TYPE*(*′a*)))#*xs*) ⟹
(*f ptr*) ∙ (*stack-ptr-acquire n vs ptr d s*) {‖ λ*r u*. *Q r* (*stack-ptr-release n ptr u*) ‖}
shows (*assume-with-fresh-stack-ptr n init f*) ∙ *s* {‖ *Q* ‖}
apply (*rule runs-to-with-fresh-stack-ptr*)

apply (*rule assms*(*4*))
apply (*assumption*)+
apply (*rule lvp-distinct-add-stack-allocs*)
apply (*simp add*: *assms*)
apply (*rule assms*(*3*))
by (*drule bspec* [**where** *x=0*], *simp*, *rule assms*(*3*), *simp add*: *ptr-add-def*) (*rule assms*(*1*))

end

end

theory *AutoCorres*
imports
LocalVarExtract
AutoCorresSimpset
Polish
Runs-To-VCG-StackPointer

keywords
autocorres
init−autocorres
final−autocorres:: *thy-decl*
begin

no-syntax -*Lab*:: *'a bexp* ⇒ (*'a*,*'p*,*'f*) *com* ⇒ *bdy*
(‹-•/-› [*1000*,*71*] *81*) — avoid syntax conflict with $f \cdot s \{\!|\ Q\ |\!\}$

declare *word-neq-0-conv* [*simp del*]
declare *neq0-conv* [*simp del*]
declare *fun-upd-apply*[*simp del*]
declare *fun-upd-same* [*simp add*]
lemma *o-const-simp*[*simp*]: (λ*x*. *C*) ∘ *f* = (λ*x*. *C*)
by (*simp add*: *o-def*)

lemma *corresTA-trivial*: *corresTA* (λ-. *True*) (λ*x*. *x*) (λ*x*. *x*) *A A*
apply (*auto intro*: *corresXF-I simp add*: *corresTA-def*)
done

lemma *L2Tcorres-trivial-from-in-out-parameters*:
IOcorres P Q st rx ex A C ⟹ *L2Tcorres id A A*
by (*rule L2Tcorres-id*)

lemma *L2Tcorres-trivial-from-local-var-extract*:
L2corres st rx ex P A C $\Longrightarrow$ *L2Tcorres id A A*
by (*rule L2Tcorres-id*)

lemma *corresTA-trivial-from-heap-lift*:
L2Tcorres st A C $\Longrightarrow$ *corresTA* (λ-. *True*) (λ*x*. *x*) (λ*x*. *x*) *A A*
by (*rule corresTA-trivial*)

lemma *corresXF-from-L2-call*:
L2-call c-WA emb ns = *A* $\Longrightarrow$ *corresXF* (λ*s*. *s*) (λ*rv s*. *rv*) (λ*r t*. *emb r*) (λ-. *True*) *A c-WA*
unfolding *L2-call-def corresXF-refines-conv*
apply (*auto simp add*: *refines-def-old reaches-map-value map-exn-def split*: *xval-splits*)[*1*]
by (*smt* (*z3*) *Exn-neq-Result Result-eq-Result the-Exn-Exn*(*1*))

definition *ac-corres′ exn st check-termination AF* Γ *rx ex G* ≡
λ*A B*. ∀ *s*. (*G s* ∧ *succeeds A* (*st s*)) ⟶
(∀ *t*. Γ ⊢ ⟨*B*, *Normal s*⟩ ⇒ *t* ⟶
(*case t of*
Normal s′ ⇒ (*Result* (*rx s′*), *st s′*) ∈ *outcomes* (*run A* (*st s*))
| *Abrupt s′* ⇒ (*exn* (*ex s′*), *st s′*) ∈ *outcomes* (*run A* (*st s*))
| *Fault e* ⇒ *e* ∈ *AF*
| - ⇒ *False*))
∧ (*check-termination* ⟶ Γ ⊢ *B* ↓ *Normal s*)

lemma *ac-corres′-nd-monad*:
assumes *ac*: *ac-corres st check-termination AF* Γ *rx ex G B C*
assumes *refines*: ⋀*s*. *refines B A s s* (*rel-prod rel-liftE* (=))
shows *ac-corres′* (λ-. *Exception* (*default*::*unit*)) *st check-termination AF* Γ *rx ex G A C*
apply (*simp add*: *ac-corres′-def*)[*1*]
apply (*intro conjI allI impI*)
subgoal
using *assms*
apply (*auto simp add*: *ac-corres-def refines-def-old split*: *xstate.splits*) [*1*]
apply (*metis reaches-def rel-liftE-Result-Result-iff*)
by (*metis Exn-neq-Result rel-liftE-def*)
apply (*elim conjE*)
subgoal premises *prems* **for** *s*
proof −
from *refines* [*simplified refines-def-old*, *rule-format*, *OF prems* (*3*)] **have** *succeeds B* (*st s*) **by** *blast*
with *prems*(*2*) **have** *G s* ∧ *succeeds B* (*st s*)
by (*auto simp add*: *succeeds-def*)
from *ac* [*simplified ac-corres-def*, *rule-format*, *OF this*] *prems*(*1*)
show *?thesis*

by *blast*
qed
done

lemma *refines-spec-rel-Nonlocal-conv*:
shows *refines f g s t* (*rel-prod* (*rel-xval rel-Nonlocal* (=)) (=))
⟷ *refines f* (*map-value* (*map-exn Nonlocal*) *g*) *s t* (*rel-prod* (=) (=))
apply (*simp add*: *refines-def-old reaches-map-value rel-xval.simps map-exn-def split*: *xval-splits*)
apply (*intro iffI*)
apply (*metis Exn-eq-Exn Result-eq-Result Result-neq-Exn rel-Nonlocal-conv*)
apply (*simp add*: *rel-Nonlocal-def*)
apply *clarsimp*
subgoal for *r s*
apply (*erule-tac x*=*r* **in** *allE*)
apply (*erule-tac x*=*s* **in** *allE*)
by (*smt* (*verit*, *best*) *Exn-def c-exntype.case*(*5*) *default-option-def exception-or-result-cases not-Some-eq*)
done

lemmas *refines-eq-convs* = *refines-spec-rel-Nonlocal-conv sum.rel-eq rel-xval-eq Relation.eq-OO*

lemma *ac-corres'-exception-monad*:
assumes *ac*: *ac-corres st check-termination AF Γ rx ex G B C*
assumes *refines*: ⋀*s*. *refines B A s s* (*rel-prod* (=) (=))
shows *ac-corres' Exn st check-termination AF Γ rx ex G A C*
term *map-value* (*map-exn Nonlocal*) *A*
apply (*simp add*: *ac-corres'-def*, *intro allI impI conjI*)
subgoal
using *assms*
by (*auto simp add*: *refines-def-old reaches-map-value ac-corres-def map-exn-def rel-sum.simps rel-Nonlocal-def split*: *xstate.splits c-exntype.splits*) (*metis reaches-def*)+
apply (*elim conjE*)
subgoal premises *prems* **for** *s*
proof −
from *refines* [*simplified refines-def-old*, *rule-format*, *OF prems* (*3*)] **have** *succeeds B* (*st s*) **by** *blast*
with *prems*(*2*) **have** *G s* ∧ *succeeds B* (*st s*)
by (*auto simp add*: *succeeds-def*)
from *ac* [*simplified ac-corres-def*, *rule-format*, *OF this*] *prems*(*1*)
show *?thesis*
by *blast*
qed
done

lemma *ac-corres-chain*:
⟦ *L1corres check-termination Gamma c-L1 c*;

```
   L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
   L2Tcorres st-HL c-HL c-L2;
   corresTA P-WA rx-WA ex-WA c-WA c-HL;
   L2-call c-WA emb ns = A
 ⟧ ⟹
  ac-corres (st-HL o st-L2) check-termination {AssumeError, StackOverflow} Gamma
(rx-WA o rx-L2) (emb o ex-WA o ex-L2) (P-L2 and (P-WA o st-HL o st-L2)) A c

  apply (rule ccorresE-corresXF-merge)
      apply (unfold L1corres-alt-def)
      apply assumption
     apply (unfold L2corres-def L2Tcorres-def corresTA-def)
     apply (drule corresXF-from-L2-call)

      apply ((drule (1) corresXF-corresXF-merge)+, assumption)
     apply (clarsimp simp: L2-call-def L2-defs)

     apply simp
    apply clarsimp
   apply clarsimp
  done

lemma ac-corres-chain-sim-nd-monad:
⟦ L1corres check-termination Gamma c-L1 c;
   L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
   IOcorres P-IO Q-IO st-IO rx-IO ex-IO c-IO c-L2;
   L2Tcorres st-HL c-HL c-IO;
   corresTA P-WA rx-WA ex-WA c-WA c-HL;
   ⋀s. refines c-WA A s s (rel-prod rel-liftE (=))
 ⟧ ⟹
  ac-corres' (λ-. Exception (default::unit)) (st-HL o st-IO o st-L2) check-termination
{AssumeError, StackOverflow} Gamma
    (λs. (rx-WA o (λv. rx-IO v (st-L2 s)) o rx-L2) s)
    (λs. (ex-WA o (λe. ex-IO e (st-L2 s)) o ex-L2) s)
    (P-L2 and (P-IO o st-L2) and (P-WA o st-HL o st-IO o st-L2)) A c
  apply (rule ac-corres'-nd-monad)
  apply (rule ccorresE-corresXF-merge)
      apply (unfold L1corres-alt-def)
      apply assumption
     apply (unfold L2corres-def L2Tcorres-def corresTA-def IOcorres-def)
      apply (drule corresXF-post-to-corresXF)
      apply ((drule (1) corresXF-corresXF-merge)+, assumption)
     apply (clarsimp simp: L2-call-def L2-defs)
    apply simp
   apply clarsimp
  apply clarsimp
  apply assumption
  done
```

lemma *ac-corres-chain-sim-exception-monad*:
⟦ *L1corres check-termination Gamma c-L1 c*;
L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
IOcorres P-IO Q-IO st-IO rx-IO ex-IO c-IO c-L2;
L2Tcorres st-HL c-HL c-IO;
corresTA P-WA rx-WA ex-WA c-WA c-HL;
⋀*s. refines c-WA A s s* (*rel-prod* (=) (=))
⟧ ⟹
ac-corres′ Exn (*st-HL o st-IO o st-L2*) *check-termination* {*AssumeError*, *Stack-Overflow*} *Gamma*
(λ*s.* (*rx-WA o* (λ*v. rx-IO v* (*st-L2 s*)) *o rx-L2*) *s*)
(λ*s.* (*ex-WA o* (λ*e. ex-IO e* (*st-L2 s*)) *o ex-L2*) *s*)
(*P-L2 and* (*P-IO o st-L2*) *and* (*P-WA o st-HL o st-IO o st-L2*)) *A c*
apply (*rule ac-corres′-exception-monad*)
apply (*rule ccorresE-corresXF-merge*)
apply (*unfold L1corres-alt-def*)
apply *assumption*
apply (*unfold L2corres-def L2Tcorres-def corresTA-def IOcorres-def*)
apply (*drule corresXF-post-to-corresXF*)
apply ((*drule* (*1*) *corresXF-corresXF-merge*)+, *assumption*)
apply (*clarsimp simp*: *L2-call-def L2-defs*)
apply *simp*
apply *clarsimp*
apply *clarsimp*
apply *assumption*
done

lemmas *ac-corres-chain-sims* = *ac-corres-chain-sim-nd-monad ac-corres-chain-sim-exception-monad*

definition *FUNCTION-BODY-NOT-IN-INPUT-C-FILE* ≡ *fail*

lemma [*polish*]:
guard (λ*s. UNDEFINED-FUNCTION*) = *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*
(*FUNCTION-BODY-NOT-IN-INPUT-C-FILE* >>= *m*) = *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*
unknown >>= (λ*x. FUNCTION-BODY-NOT-IN-INPUT-C-FILE*) = *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*
liftE FUNCTION-BODY-NOT-IN-INPUT-C-FILE = *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*
by (*rule spec-monad-ext*,
simp add: *run-bind run-guard UNDEFINED-FUNCTION-def FUNCTION-BODY-NOT-IN-INPUT-C-FIL*

lemmas *ac-statistics-rewrites* =

L1-seq-def

L2-defs′

There might be unexpected simplification 'unfolding' of *id* due to eta-expansion: *id* might be expanded (e.g. by resolution to) *id*. Now the simp

rule *id ?x = ?x* triggers and rewrites it $\lambda x.\ x$. Folding this back to *id* might help in those cases.

named-theorems
l1-corres **and** *l2-corres* **and** *io-corres* **and** *hl-corres* **and** *wa-corres* **and** *ts-corres* **and** *ac-corres* **and**
known-function-corres **and** *known-function*

lazy-named-theorems
l1-def **and** *l2-def* **and** *io-def* **and** *hl-def* **and** *wa-def* **and** *ts-def* **and** *ac-def*
named-theorems
heap-update-syntax

lemma *fold-id*: $(\lambda x.\ x) = id$
by (*simp add*: *id-def*)

lemma *fold-id-unit*: $(\lambda\text{-}.\ ()) = id$
by (*simp add*: *id-def*)

ML-file *lib/set.ML*
ML-file *trace-antiquote.ML*

ML-file *function-info.ML*
ML-file *program-info.ML*
ML-file *autocorres-trace.ML*
ML-file *autocorres-options.ML*
ML-file *autocorres-data.ML*

ML-file *autocorres-util.ML*

ML-file *exception-rewrite.ML*
ML-file *simpl-conv.ML*

ML-file *prog.ML*
ML-file *pretty-bound-var-names.ML*
ML-file *l2-opt.ML*
ML-file *local-var-extract.ML*

context *globals-stack-heap-state*
begin
ML-file *in-out-parameters.ML*

declaration ‹*fn phi* =>

In-Out-Parameters.Data.add (In-Out-Parameters.operations phi) phi›
end

ML-file *record-utils.ML*
ML-file *heap-lift-base.ML*
ML-file *heap-lift.ML*

ML-file *word-abstract.ML*
ML-file *monad-convert.ML*

ML-file *type-strengthen.ML*

ML-file *autocorres.ML*

ML ‹
Outer-Syntax.command @{command-keyword init−autocorres}
Initialise Autocorres
(AutoCorres.init-autocorres-parser >>
(Toplevel.theory o (fn (opt, filename) => AutoCorres.do-init-autocorres opt filename true #> snd)))
›

ML ‹
Outer-Syntax.command @{command-keyword autocorres}
Abstract the output of the C parser into a monadic representation.
(AutoCorres.autocorres-parser >>
(Toplevel.theory o (fn (opt, filename) => AutoCorres.parallel-autocorres opt filename)))
›

ML ‹
Outer-Syntax.command @{command-keyword final−autocorres}
Finalise Autocorres
(AutoCorres.final-autocorres-parser >>
(Toplevel.theory o (fn filename => AutoCorres.final-autocorres-cmd filename)))
›

setup ‹
let
fun fresh-var maxidx (n, T) = Var ((- ^ n, maxidx + 1), T)
fun head-type t = t |> HOLogic.dest-Trueprop |> head-of |> fastype-of
fun get-maxidx maxidx ts =
if maxidx < 0
then fold (curry Int.max) (map maxidx-of-term ts) 0

else maxidx

— Note that the *mk-pattern* functions serve two purposes. When adding a rule into the term net we insert Var's for the positions that we want to synthesise. The concrete program '?C' serves as index into the net and is unmodified. Note that `Utils.open_beta_norm_eta` should actually be the identity (with respect to matching) when applied to the rules.
Before querying the term-net with a concrete subgoal we also use *mk-pattern*. Here `Utils.open_beta_norm_eta` is essential to make term-net-retrieval for matching (instead of 'unification') work, as it gets rid of beta-eta artefacts in the goal that are generated during recursive application of the rules.

```
  fun mk-corresTA-pattern maxidx (t as @{term-pat Trueprop (corresTA ?P ?rx
?ex ?A ?C)}) =
    let
      val mi = get-maxidx maxidx [P, rx, ex, A, C]
      val T = head-type t
      val [PT, rxT, exT, AT, -] = binder-types T
      val [P', rx', ex', A'] = map (fresh-var mi) [(P, PT), (rx, rxT), (ex, exT), (A,
AT)]
      val pat = HOLogic.mk-Trueprop (Const (@{const-name corresTA}, T) $
          P' $ rx' $ ex' $ A' $ Utils.open-beta-norm-eta C)

    in pat end

  fun mk-abstract-val-pattern maxidx (t as (@{term-pat Trueprop (abstract-val ?P
?A ?f ?C)})) =
    let
      val mi = get-maxidx maxidx [P, A, f, C]
      val T = head-type t
      val [PT, AT, fT, -] = binder-types T
      val [P', A', f'] = map (fresh-var mi) [(P, PT), (A, AT), (f, fT)]
      val pat = HOLogic.mk-Trueprop (Const (@{const-name abstract-val}, T) $
          P' $ A' $ f' $ Utils.open-beta-norm-eta C)
    in pat end

  fun mk-valid-typ-abs-fn-pattern maxidx (t as (@{term-pat Trueprop (valid-typ-abs-fn
?P ?Q ?A ?C)})) =
    let
      val mi = get-maxidx maxidx [P, Q, A, C]
      val T = head-type t
      val [PT, QT, AT, CT] = binder-types T
      val [P', Q', A', C'] = map (fresh-var mi) [(P, PT), (Q, QT), (A, AT), (C,
CT)]
      val pat = HOLogic.mk-Trueprop (Const (@{const-name valid-typ-abs-fn}, T)
$
          P' $ Q' $ A' $ C')
    in pat end

  fun mk-introduce-typ-abs-fn-pattern maxidx (t as (@{term-pat Trueprop (introduce-typ-abs-fn
```

```
?f)})) =
    let
      val mi = get-maxidx maxidx [f]
      val T = head-type t
      val [fT] = binder-types T
      val [f'] = map (fresh-var mi) [(f, fT)]
     val pat = HOLogic.mk-Trueprop (Const (@{const-name introduce-typ-abs-fn},
T) $ f')
    in pat end

  fun mk-id-pattern - t = t

  fun mk-abs-expr-pattern maxidx (t as (@{term-pat Trueprop (abs-expr ?st ?P ?A
?C)})) =
    let
      val mi = get-maxidx maxidx [st, P, A, C]
      val T = head-type t
      val [stT, PT, AT, -] = binder-types T
      val [st', P', A'] = map (fresh-var mi) [(st, stT), (P, PT), (A, AT)]
      val pat = HOLogic.mk-Trueprop (Const (@{const-name abs-expr}, T) $
            st' $ P' $ A' $ Utils.open-beta-norm-eta C)
    in pat end

  fun mk-abs-guard-pattern maxidx (t as (@{term-pat Trueprop (abs-guard ?st ?A
?C)})) =
    let
      val mi = get-maxidx maxidx[st, A, C]
      val T = head-type t
      val [stT, AT, -] = binder-types T
      val [st', A'] = map (fresh-var mi) [(st, stT), (A, AT)]
      val pat = HOLogic.mk-Trueprop (Const (@{const-name abs-guard}, T) $
            st' $ A' $ Utils.open-beta-norm-eta C)
    in pat end

  fun mk-L2Tcorres-pattern maxidx (t as (@{term-pat Trueprop (L2Tcorres ?st ?A
?C)})) =
    let
      val mi = get-maxidx maxidx[st, A, C]
      val T = head-type t
      val [stT, AT, -] = binder-types T
      val [st', A'] = map (fresh-var mi) [(st, stT), (A, AT)]
      val pat = HOLogic.mk-Trueprop (Const (@{const-name L2Tcorres}, T) $
            st' $ A' $ Utils.open-beta-norm-eta C)
    in pat end

  fun mk-abs-modifies-pattern maxidx (t as (@{term-pat Trueprop (abs-modifies ?st
?P ?A ?C)})) =
    let
      val mi = get-maxidx maxidx [st, P, A, C]
```

```
    val T = head-type t
    val [stT, PT, AT, -] = binder-types T
    val [st', P', A'] = map (fresh-var mi) [(st, stT), (P, PT), (A, AT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name abs-modifies}, T) $
        st' $ P' $ A' $ Utils.open-beta-norm-eta C)
  in pat end

 fun mk-struct-rewrite-guard-pattern maxidx (t as (@{term-pat Trueprop (struct-rewrite-guard
?A ?C)})) =
  let
    val mi = get-maxidx maxidx [A, C]
    val T = head-type t
    val [AT, -] = binder-types T
    val [A'] = map (fresh-var mi) [(A, AT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name struct-rewrite-guard},
T) $
        A' $ Utils.open-beta-norm-eta C)
  in pat end

 fun mk-struct-rewrite-expr-pattern maxidx (t as (@{term-pat Trueprop (struct-rewrite-expr
?P ?A ?C)})) =
  let
    val mi = get-maxidx maxidx[P, A, C]
    val T = head-type t
    val [PT, AT, -] = binder-types T
    val [P', A'] = map (fresh-var mi) [(P, PT), (A, AT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name struct-rewrite-expr},
T) $
        P' $ A' $ Utils.open-beta-norm-eta C)
  in pat end

 fun mk-struct-rewrite-modifies-pattern maxidx (t as (@{term-pat Trueprop (struct-rewrite-modifies
?P ?A ?C)})) =
  let
    val mi = get-maxidx maxidx [P, A, C]
    val T = head-type t
    val [PT, AT, -] = binder-types T
    val [P', A'] = map (fresh-var mi) [(P, PT), (A, AT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name struct-rewrite-modifies},
T) $
        P' $ A' $ Utils.open-beta-norm-eta C)
  in pat end

 fun mk-abs-spec-pattern maxidx (t as (@{term-pat Trueprop (abs-spec ?st ?P ?A
?C)})) =
  let
    val mi = get-maxidx maxidx[st, P, A, C]
    val T = head-type t
    val [stT, PT, AT, -] = binder-types T
```

```
    val [st', P', A'] = map (fresh-var mi) [(st, stT), (P, PT), (A, AT)]
    val pat = HOLogic.mk-Trueprop (Const (@{const-name abs-spec}, T) $
          st' $ P' $ A' $ Utils.open-beta-norm-eta C)
  in pat end

 fun mk-heap-lift--wrap-h-val-pattern maxidx (t as (@{term-pat Trueprop (heap-lift--wrap-h-val
?X ?Y)})) =
  let
    val mi = get-maxidx maxidx [X, Y]
    val T = head-type t
    val [XT, YT] = binder-types T
    val [X', Y'] = map (fresh-var mi) [(X, XT), (Y, YT)]
   val pat = HOLogic.mk-Trueprop (Const (@{const-name heap-lift--wrap-h-val},
T) $ X' $ Y' )
  in pat end

in
  Context.theory-map (fold WordAbstract.add-pattern [
    mk-corresTA-pattern,
    mk-abstract-val-pattern,
    mk-valid-typ-abs-fn-pattern,
    mk-introduce-typ-abs-fn-pattern,
    mk-id-pattern
  ]) #>
  Context.theory-map (fold HeapLift.add-pattern [
    mk-abs-expr-pattern,
    mk-abs-guard-pattern,
    mk-L2Tcorres-pattern,
    mk-abs-modifies-pattern,
    mk-struct-rewrite-guard-pattern,
    mk-struct-rewrite-expr-pattern,
    mk-struct-rewrite-modifies-pattern,
    mk-abs-spec-pattern,
    mk-heap-lift--wrap-h-val-pattern,
    mk-id-pattern
  ])
end

›


end
```

Part IV

Documentation

2015

Chapter 25

Quickstart

Japheth Lim, Rohan Jacob-Rao, David Greenaway and Norbert Schirmer

25.1 Introduction

AutoCorres is a tool that attempts to simplify the formal verification of C programs in the Isabelle/HOL theorem prover. It allows C code to be automatically abstracted to produce a higher-level functional specification.

AutoCorres relies on the C-Parser [7] developed by Michael Norrish at NICTA. This tool takes raw C code as input and produces a translation in SIMPL [8], an imperative language written by Norbert Schirmer on top of Isabelle. AutoCorres takes this SIMPL code to produce a "monadic" specification, which is intended to be simpler to reason about in Isabelle. The composition of these two tools (AutoCorres applied after the C-Parser) can then be used to reason about C programs.

This guide is written for users of Isabelle/HOL, with some knowledge of C, to get started proving properties of C programs. Using AutoCorres in conjunction with the verification condition generator (VCG) *runs-to-vcg*, one should be able to do this without an understanding of SIMPL nor all the details of the monadic representation produced by AutoCorres. We will see how this is possible in the next chapter.

25.2 A First Proof with AutoCorres

We will now show how to use these tools to prove correctness of some very simple C functions.

25.2.1 Two simple functions: `min` and `max`

Consider the following two functions, defined in a file `minmax.c`, which (we expect) return the minimum and maximum respectively of two unsigned integers.

```
unsigned min(unsigned a, unsigned b)
{
    if (a <= b) {
        return a;
    } else {
        return b;
    }
}

unsigned max(unsigned a, unsigned b)
{
    return UINT_MAX - (
                min(UINT_MAX - a, UINT_MAX - b));
}
```

It is easy to see that `min` is correct, but perhaps less obvious why `max` is correct. AutoCorres will hopefully allow us to prove these claims without too much effort.

25.2.2 Invoking the C-parser

As mentioned earlier, AutoCorres does not handle C code directly. The first step is to apply the C-Parser by using **install-C-file** to obtain a SIMPL translation. We do this using the `install-C-file` command in Isabelle, as shown.

install-C-file *sources/minmax.c*

For every function in the C source file, the C-Parser generates a corresponding Isabelle definition. These definitions are placed in an Isabelle "locale", whose name matches the input filename. For our file *minmax.c*, the C-Parser will place definitions in the locale *minmax*.[1]

For our purposes, we just have to remember to enter the appropriate locale before writing our proofs. This is done using the `context` keyword in Isabelle.

Let's look at the C-Parser's outputs for `min` and `max`, which are contained in the theorems `min_body_def` and `max_body_def`. These are simply definitions of the generated names *min_body* and *max_body*. We can also see here how our work is wrapped within the *minmax* context.

context *minmax-simpl* **begin**

[1] The C-parser uses locales to avoid having to make certain assumptions about the behaviour of the linker, such as the concrete addresses of symbols in your program.

thm *min-body-def*

min-body ≡
TRY
lvar-nondet-init (λ*upd*. *min.ret*′:=$_{\mathcal{L}}$ *upd*);;
IF ´*min.a* ≤ ´*min.b THEN*
creturn global-exn-var′-′*-update* (λ*upd*. *min.ret*′:=$_{\mathcal{L}}$ *upd*)
(λ*s*. *s* ·$_{\mathcal{L}}$ *min.a*)
ELSE
creturn global-exn-var′-′*-update* (λ*upd*. *min.ret*′:=$_{\mathcal{L}}$ *upd*)
(λ*s*. *s* ·$_{\mathcal{L}}$ *min.b*)
FI;;
Guard DontReach {} *SKIP*
CATCH ccatchreturn global-exn-var′-′
END

thm *max-body-def*

max-body ≡
TRY
lvar-nondet-init (λ*upd*. *max.ret*′:=$_{\mathcal{L}}$ *upd*);;
(´*max.ret*′*unsigned*′*1* :== *CALL*$_e$ *minmax.min*(− *1* − ´*max.a*,− *1* − ´*max.b*);;
creturn global-exn-var′-′*-update* (λ*upd*. *max.ret*′:=$_{\mathcal{L}}$ *upd*)
(λ*s*. − *1* − (*s* ·$_{\mathcal{L}}$ *max.ret*′*unsigned*′*1*)));;
Guard DontReach {} *SKIP*
CATCH ccatchreturn global-exn-var′-′
END

end

The definitions above show us the SIMPL generated for each of the functions; we can see that C-parser has translated `min` and `max` very literally and no detail of the C language has been omitted. For example:

- C `return` statements have been translated into exceptions which are caught at the outside of the function's body;
- *Guard* statements are used to ensure that behaviour deemed 'undefined' by the C standard does not occur. In the above functions, we see that a guard statement is emitted that ensures that program execution does not hit the end of the function, ensuring that we always return a value (as is required by all non-`void` functions).
- Function parameters are modelled as local variables, which are setup prior to a function being called. Return variables are also modelled as local variables, which are then read by the caller.

While a literal translation of C helps to improve confidence that the translation is sound, it does tend to make formal reasoning an arduous task.

25.2.3 Invoking AutoCorres

Now let's use AutoCorres to simplify our functions. This is done using the **autocorres** command, in a similar manner to the **install-C-file** command:

autocorres *minmax.c*

AutoCorres produces a definition in the `minmax` locale for each function body produced by the C parser. For example, our `min` function is defined as follows:

context *minmax-all-corres* **begin**
thm *min'-def*

$$min'\ ?a\ ?b \equiv if\ ?a \leq ?b\ then\ ?a\ else\ ?b$$

Each function's definition is named identically to its name in C, but with a prime mark (`'`) appended. For example, our functions `min` above was named *min'*, while the function `foo_Bar` would be named *foo-Bar'*.

AutoCorres does not require you to trust its translation is sound, but also emits a *correspondence* or *refinement* proof, as follows:

thm *min'-ac-corres*

$$\begin{aligned}&ac\text{-}corres'\ (\lambda\text{-}.\ Exception\ ())\ (lift\text{-}global\text{-}heap \circ (\lambda s.\ s) \circ globals)\ True\\ &\{AssumeError,\ StackOverflow\}\ \Gamma\ (\lambda s.\ s \cdot_{\mathcal{L}} min.ret')\ global\text{-}exn\text{-}var'\text{-}'\\ &((\lambda s.\ s \cdot_{\mathcal{L}} min.a = ?a')\ and\ (\lambda s.\ s \cdot_{\mathcal{L}} min.b = ?b')\ and\\ &\ (\lambda\text{-}.\ True) \circ globals\ and\\ &\ (\lambda x.\ abs\text{-}var\ ?a\ id\ ?a' \wedge abs\text{-}var\ ?b\ id\ ?b') \circ lift\text{-}global\text{-}heap \circ (\lambda s.\ s) \circ\\ &\ globals)\\ &(return\ (min'\ ?a\ ?b))\ (Call\ minmax.min)\end{aligned}$$

Informally, this theorem states that, assuming the abstract function *min'* can be proven to not fail for a partciular input, then for the associated input, the concrete C SIMPL program also will not fault, will always terminate, and will have a corresponding end state to the generated abstract program.

For more technical details, see [2] and [3] or chapter 26.

25.2.4 Verifying `min`

In the abstracted version of *min'*, we can see that AutoCorres has simplified away the local variable reads and writes in the C-parser translation of `min`, simplified away the exception throwing and handling code, and also simplified away the unreachable guard statement at the end of the function. In fact, *min'* has been simplified to the point that it exactly matches Isabelle's built-in function *min*:

thm *min-def*

min ?a ?b = (if ?a ≤ ?b then ?a else ?b)

So, verifying *min′* (and by extension, the C function `min`) should be easy:

lemma *min′-is-min*: *min′ a b = min a b*
unfolding *min-def min′-def*
by (*rule refl*)

25.2.5 Verifying `max`

Now we also wish to verify that *max′* implements the built-in function *max*. *min′* was nearly too simple to bother verifying, but *max′* is a bit more complicated. Let's look at AutoCorres' output for `max`:

thm *max′-def*

max′ ?a ?b ≡ − 1 − min′ (− 1 − ?a) (− 1 − ?b)

At this point, you might still doubt that *max′* is indeed correct, so perhaps a proof is in order. The basic idea is that subtracting from `UINT_MAX` flips the ordering of unsigned ints. We can then use *min′* on the flipped numbers to compute the maximum.

The next lemma proves that subtracting from `UINT_MAX` flips the ordering. To prove it, we convert all words to *int*'s, which does not change the meaning of the statement.

lemma *n1-minus-flips-ord*:
((a :: word32) ≤ b) = ((−1 − a) ≥ (−1 − b))
apply (*subst word-le-def*)+
apply (*subst word-n1-ge* [*simplified uint-minus-simple-alt*])+

Now that our statement uses *int*, we can apply Isabelle's built-in `arith` method.

apply *arith*
done

And now for the main proof:

lemma *max′-is-max*: *max′ a b = max a b*
unfolding *max′-def min′-def max-def*
using *n1-minus-flips-ord*
by *force*

end

In the next section, we will see how to use AutoCorres to simplify larger, more realistic C programs.

25.3 More Complex Functions with AutoCorres

In the previous section we saw how to use the C-Parser and AutoCorres to prove properties about some very simple C programs.

Real life C programs tend to be far more complex however; they read and write to local variables, have loops and use pointers to access the heap. In this section we will look at some simple programs which use loops and access the heap to show how AutoCorres can allow such constructs to be reasoned about.

25.3.1 A simple loop: `mult_by_add`

Our C function `mult_by_add` implements a multiply operation by successive additions:

```
unsigned mult_by_add(unsigned a, unsigned b)
{
    unsigned result = 0;
    while (a > 0) {
        result += b;
        a--;
    }
    return result;
}
```

We start by translating the program using the C parser and AutoCorres, and entering the generated locale `mult_by_add`.

install-C-file *sources/mult-by-add.c*
autocorres [*ts-rules = nondet*] *mult-by-add.c*

The C parser produces the SIMPL output as follows:

thm *mult-by-add-body-def*

```
mult-by-add-body ≡
TRY
  lvar-nondet-init (λupd. ret'_L := upd);;
  (´result :== SCAST(32 signed → 32) 0;;
   (WHILE SCAST(32 signed → 32) 0 < ´a DO
      ´result :== ´result + ´b;;
      ´a :== ´a − SCAST(32 signed → 32) 1
    OD;;
    creturn global-exn-var'-'-update (λupd. ret'_L := upd)
     (λs. s ·_L result)));;
  Guard DontReach {} SKIP
CATCH ccatchreturn global-exn-var'-'
END
```

Which is abstracted by AutoCorres to the following:

thm *mult-by-add'-def*

```
mult-by-add' ?a ?b ≡
do {
  (a, result) ← whileLoop (λ(a, result) s. 0 < a)
                          (λ(a, result). return (a − 1, result + ?b))
                          (?a, 0);
  return result
}
```

In this case AutoCorres has abstracted `mult_by_add` into a *monadic* form. Monads are a pattern frequently used in functional programming to represent imperative-style control-flow; an in-depth introduction to monads can be found elsewhere.

The monads used by AutoCorres in this example is a *non-deterministic state monad*; program state is implicitly passed between each statement, and results of computations may produce more than one (or possibly zero) results[2].

The HOL type is called specification monad: ('*e*, '*a*, '*s*) *spec-monad*, where

- '*e* type for exception / error results,
- '*r* type of the result and
- '*s* type of the state.

  When '*e* is instantiated with the unit type *unit*, there is an abbreviation ('*a*, '*s*) *res-monad*, for *result monad*, When it is instantiated with a proper type we have an abbreviation ('*e*, '*a*, '*s*) *exn-monad*, for *exception monad*.

  To uniformly model results and exceptions as values the type ('*e*, '*a*) *exception-or-result* is used. It is constructed as a sum type where we exclude a default value from the exception type '*e*. So when '*e* is instantiated with *unit* the type is isomorphic to the result type '*a*. Type *unit* is only inhabited by the unique unit value () which is also default value. This instance is abbreviated with '*a val*. For proper exceptions we instantiate '*e* with an option type '*x option*. This instance is abbreviated with ('*x*, '*a*) *xval*.

  Constructors and pattern matching:

- ('*e*, '*a*) *exception-or-result*:*Exception e*, *Result v*, and pattern matching *λx. case x of Exception e ⇒ f e | Result v ⇒ g v*

[2]Non-determinism becomes useful when modelling hardware, for example, where the exact results of the hardware cannot be determined ahead of time.

- *'a val*:*Result v*, with pattern matching *λRes v. g v*
- (*'e*, *'a*) *xval*: *Exn e*, *Result v*, and pattern matching *λx. case x of Exn e ⇒ f e | Result v ⇒ g v*

  This enconding allows us to give a uniform definition of the monadic (>>=) which works for the generic specification monad and thus also for its instances the result and the exception monad.

'Hoare Triples'

The bulk of *mult-by-add'* is wrapped inside the *whileLoop monad combinator*, which is really just a fancy way of describing the method used by AutoCorres to encode (potentially non-terminating) loops using monads.

If we want to describe the behaviour of this program, we can use Hoare logic as follows:

P s ⟹ mult-by-add' a b · s {‌{ Q }‌}

This predicate states that, assuming P holds on the initial program state, if we execute *mult-by-add' a b*, then Q will hold on the final state of the program.

There are a few details: while P has type *'s ⇒ bool* (i.e., takes a state and returns true if it satisifies the precondition), Q has an additional parameter *'r ⇒ 's ⇒ bool*. The additional parameter *'r* is the *return value* of the function; so, in our *mult-by-add'* example, it will be the result of the multiplication. Note that the precondition is not part of the 'hoare-triple' but is an ordinary Isabelle assumption on the initial state s. That way we can directly use Isabelle/Isar to decompose the precondition and can also refer to the initial state within the postcondition. The foundational constant for this hoare triple is *runs-to* which means total correctness: we have to proof that the program terminates and that there is no undefined behaviour (all guards must hold). There is also a weaker notion for partial correctness *runs-to-partial* with syntax *f · s ?{‌{ Q }‌}*.

For example one useful property to prove on our program would be:

*mult-by-add' a b · s {‌{ λRes r -. r = a * b }‌}*

That is, for all possible input states, our `mult_by_add'` function returns the product of a and b.

Our proof of *mult-by-add'* could then proceed as follows:

lemma *mult-by-add-correct*:
*mult-by-add' a b · s {‌{ λRes r -. r = a * b }‌}*

Unfold abstracted definition

apply (*unfold mult-by-add′-def*)

Annotate the loop with an invariant and a measure, by adding a specialised rule to the verification condition generator.

supply *runs-to-whileLoop-res* [**where**
I=λ(*a′*, *result*) *s*. *result* = (*a* − *a′*) ∗ *b* **and**
R=*measure′* (λ((*a′*, *result*), *s*). *a′*),
tags body post, — optional: tag resulting verification conditions
runs-to-vcg]

Run the "verification condition generator".

apply (*runs-to-vcg*)

Solve the program correctness goals.

subgoalsT *body*
apply (*simp add*: *field-simps*)
apply *unat-arith*
done
subgoalT *post*
apply (*auto simp*: *field-simps not-less*)
done
done

The main tool is the proof method *runs-to-vcg*, a verification condition generator for monadic programs. It uses weakest precondition style rules which are collected in named theorems collection *runs-to-vcg*. In that collection there is no rule for *whileLoop*. The verification generator will stop there unless you specify a rule to use. This was done in the proof above by instantiating the rule *runs-to-whileLoop-res* with an invariant and a measure and by adding it to the verification condition generator via attribute *runs-to-vcg*. We finally discharge the remaining subgoals left from *runs-to-vcg* with standard proof tools of Isabelle.

In the next section, we will look at how we can use AutoCorres to verify a C program that reads and writes to the heap.

25.3.2 `swap`

Here, we use AutoCorres to verify a C program that reads and writes to the heap. Our C function, `swap`, swaps two words in memory:

```
void swap(unsigned *a, unsigned *b)
{
    unsigned t = *a;
    *a = *b;
    *b = t;
}
```

Again, we translate the program using the C parser and AutoCorres.

install-C-file *sources/swap.c*
autocorres [*heap-abs-syntax*, *ts-rules* = *nondet*] *swap.c*

Most heap operations in C programs consist of accessing a pointer. AutoCorres abstracts the global C heap by creating one heap for each type. (In our simple `swap` example, it creates only a *Word-32.word32* heap.) This makes verification easier as long as the program does not access the same memory as two different types.

There are other operations that are relevant to program verification, such as changing the type that occupies a given region of memory. AutoCorres will not abstract any functions that use these operations, so verifying them will be more complicated (but still possible).

The C parser expresses `swap` like this:

thm *swap-body-def*

```
swap-body ≡
TRY
  Guard C-Guard {|c-guard ´a|} (´t :== h-val (hrs-mem ´t-hrs) ´a);;
  (Guard C-Guard {|c-guard ´a|}
    (Guard C-Guard {|c-guard ´b|}
      (´globals :==
         t-hrs-'-update
          (hrs-mem-update (heap-update ´a (h-val (hrs-mem ´t-hrs) ´b)))));;
   Guard C-Guard {|c-guard ´b|}
    (´globals :== t-hrs-'-update (hrs-mem-update (heap-update ´b ´t))))
CATCH ccatchreturn global-exn-var'-'
END
```

AutoCorres abstracts the function to this:

thm *swap'-def*

```
swap' ?a ?b ≡ do {
  guard (λs. IS-VALID(32 word) s ?a);
  guard (λs. IS-VALID(32 word) s ?b);
  t ← gets (λs. s[?a]);
  modify (λs. s[?a := s[?b]]);
  modify (λs. s[?b := t])
}
```

There are some things to note:

The function contains guards (assertions) that the pointers `a` and `b` are valid. We need to prove that these guards never fail when verifying `swap`. Conversely, when verifying any function that calls `swap`, we need to show that the arguments are valid pointers.

We saw a monadic program in the previous section, but here the monad is actually being used to carry the program heap.

Now we prove that `swap` is correct. We use x and y to "remember" the initial values so that we can talk about them in the post-condition. The heap access syntax $s[a]$ is used to select the split heap *heap-w32* from state s at pointer a. [3]

lemma $[\![\mathit{IS\text{-}VALID}(32\ \mathit{word})\ s\ a;\ s[a] = x;\ \mathit{IS\text{-}VALID}(32\ \mathit{word})\ s\ b;\ s[b] = y]\!] \Longrightarrow$
$\mathit{swap}'\ a\ b \cdot s$
$\{\!|\ \lambda\text{-}\ s.\ s[a] = y \wedge s[b] = x\ |\!\}$
apply (*unfold swap′-def*)
apply *runs-to-vcg*
apply *clarsimp*

The C parser and AutoCorres both model the C heap using functions, which takes a pointer to some object in memory. Heap updates are modelled using the functional update *fun-upd*:

$?f(?a := ?b) = (\lambda x.\ \mathit{if}\ x = ?a\ \mathit{then}\ ?b\ \mathit{else}\ ?f\ x)$

To reason about functional updates, we use the rule fun_upd_apply.

apply (*simp add*: *fun-upd-apply*)
done

Note that we have "only" proved that the function swaps its arguments. We have not proved that it does *not* change any other state. This is a typical *frame problem* with pointer reasoning. We can prove a more complete specification of `swap`:

lemma $[\![(\bigwedge x\ y\ s.\ P\ (s[a := x][b := y]) = P\ s);$
$\mathit{IS\text{-}VALID}(32\ \mathit{word})\ s\ a;\ s[a] = x;\ \mathit{IS\text{-}VALID}(32\ \mathit{word})\ s\ b;\ s[b] = y\ ;\ P\ s\ ]\!]$
$\Longrightarrow$
$(\mathit{swap}'\ a\ b) \cdot s$
$\{\!|\ \lambda\text{-}\ s.\ s[a] = y \wedge s[b] = x \wedge P\ s\ |\!\}$
apply (*unfold swap′-def*)
apply *runs-to-vcg*
apply (*clarsimp simp*: *fun-upd-apply*)+
done

In other words, if predicate P does not depend on the inputs to `swap`, it will continue to hold.

Separation logic provides a more structured approach to this problem.

[3] For more details on the split heap model in autocorres see section 26.21.

25.4 Command Options and Invocation

25.4.1 Session Structure

The supplied session structure has some peculiarities to accomodate the AFP. The AFP presents the documentation of the session that is named like the AFP-entry. This is `AutoCorres2`. This session also includes this documentation and other example application of AutoCorres. When you want to use AutoCorres for your own projects you do not have to import these examples. Thats why we also supply the leaner `AutoCorres2_Main` as entry point, which we recommend as parent session for your applications.

25.4.2 C-Parser

Options for **install-C-file**,
syntax `install_C_file <filename> [<option> = value, ...]`:

- `memsafe` (default false): add additional guards to ensure that well-typedness of pointers
- `c_types` (default true): import types to UMM model
- `c_defs` (default true): import function to SIMPL
- `roots`: (default all functions): List of C functions that are the root functions to import
- `prune_types` (default true): only import those types that are actually used in the program
- `machinety`: HOL type for ghost state modelling the machine
- `gostty`: HOL type for some additional ghost state
- `skip_asm` (default false): Skip inline assembler

Moreover there are some Isabelle options for more feedback / tracing:

- `c_parser_feedback_level` (default 0), higher number means more tracing
- `option c_parser_verbose` (default false), enable for verbose messages

Multiple C Files

Command **install-C-file** only has a single C file as argument. When your project consists of several `.c` and `.h` you can register those dependencies first, with command **include-C-file**. The main C file, which is the argument to **install-C-file**, can then include all these files. If there is no such C file in your project you can create one to accomodate **install-C-file**.

Target Architecture Selection `L4V_ARCH`

The Environment variable `L4V_ARCH` can be used to determine the target architecture which influences the sizes of C integral types and pointer types. Supported platforms are `ARM` (default), `ARM64`, `ARM_HYP`, `RISCV64` and `X64`.

For `ARM`, the sizes are:

- 128 bits: `int128`
- 64 bits: `long long`
- 32 bits: pointers, `long`, `int`
- 16 bits: `short`

For `X64`, `ARM64`, `ARM_HYP`:

- 128 bits: `int128`
- 64 bits: pointers, `long long`, `long`
- 32 bits: `int`
- 16 bits: `short`

For `ARM64` `char` is signed.

For example to build AutoCorres for ARM64:

```
L4V_ARCH=ARM64 isabelle build -d . AutoCorres_Main
```

25.4.3 AutoCorres

Options for **init−autocorres** / **autocorres**,
syntax `autocorres [option = value, ...] <filename>`:

- `phase`: perform autocorres up to specified phase only (L1, L2, HL, WA, IO, TS)
- `scope`: space separated list of functions to perform autocorres on. (Default all).

- `scope_depth`: depth of callees to also also include
- `single_threaded`: flag to disable parallel processing (e.g. to make more sense out of tracing messages)
- `no_heap_abs`: space separated list of functions that should be excluded from heap abstraction
- `in_out_parameters`: 'and' separated list of function specs e.g. `inc(y:in_out)`, cf. `../In_Out_Parameters_Ex.thy`
- `in_out_globals`: 'and' separated list of functions which modify global variables via pointers
- `skip_io_abs`: flag to disable IO abstraction
- `addressable_fields`: space separated list of paths to struct fields that should be addressable in the split heap, cf. `../open_struct.thy`
- `ignore_addressable_fields_error`: option to downgrade error to a warning
- `skip_heap_abs`: flag to disable heap abstraction (into split heap)
- `unsigned_word_abs`: space separated list of functions where unsigned words should be abstracted to *nat*.
- `unsigned_word_abs_known_functions`: assume unsigned word abstraction for function pointers
- `no_signed_word_abs`: space separated list of functions where abstraction of unsigned word to *int* should be disabled.
- `no_signed_word_abs_known_functions`: don't assume unsigned word abstraction for function pointers
- `skip_word_abs`: flag to disable word abstraction
- `ts_rules`: space separated list of rules to consider during TS phase (pure, gets, option, nondet, exit).
- `ts_force <rule-name>`: space separated list of function names to put in the specified monad (pure, gets, option, nondet, exit).
- `ts_force_known_functions`: assume function pointers live in the specified monad (pure, gets, option, nondet, exit)
- `heap_abs_syntax`: enable some additional syntax for heap accesses
- `do_polish` (default true): flag for polish phase

- `L1_opt` (RAW | PEEP (default)): level for L1 optimisation
- `L2_opt` (RAW | PEEP (default)): level for L2 optimisation
- `HL_opt` (RAW | PEEP (default)): level for HL optimisation
- `WA_opt` (RAW | PEEP (default)): level for WA optimisation
- `trace_opt`: flag to enable some tracing
- `gen_word_heaps`: flag to generate word heaps even if the program does not use them
- `keep_going`: continue despite some errors
- `lifted_globals_field_prefix`: custom prefix for split heap naming
- `lifted_globals_field_suffix`: custom suffix for split heap naming
- `function_name_prefix`: custom prefix for generated function names
- `function_name_suffix`: custom suffix for generated function names
- `no_c_termination`: flag to disable termination precondition for correspondence proofs
- `unfold_constructor_bind`: (Selectors (default) | Always | Never) to give some user level control to unfold certain "simple" binds. cf.: `../../tests/proof-tests/unfold_bind_options.thy`
- `base_locale`: custom base locale for all autocorres locales

Moreover there are some Isabelle options for more feedback / tracing:

- `option verbose` (default 0), higher value means more verbosity
- `verbose_timing` (default 0), higher value means more timing messages
- `timing_threshold` (default 3), threshold in milliseconds for timing messages

Chapter 26

Overview of AutoCorres

theory *AutoCorresInfrastructure*
imports *AutoCorres*
begin

This theory elaborates on some of the (internal) AutoCorres infrastructure and is also supposed to act as a testbench for this infrastructure, e.g. simplifier setup.

Following the Isabelle tradition user relevant changes to AutoCorres are described in the file `../NEWS`.

26.1 Building Blocks

'AutoCorres' has the following major building blocks and contributions.

- The 'C-parser' (by Michael Norrish) takes C-Input files and parses them to 'SIMPL' (by Norbert Schirmer) programs within Isabelle/HOL:
  - C-parser: chapter 28
  - SIMPL: https://www-wjp.cs.uni-saarland.de/leute/private_homepages/nschirmer/pub/schirmer_phd.pdf
- The unified memory model (UMM) (by Harvey Tuch) models C-types as HOL-types with conversions between the typed view and the raw-byte-level view. Moreover it provides a separation logic framework.
  - UMM: https://trustworthy.systems/publications/papers/Tuch%3Aphd.pdf
- AutoCorres (by David Greenaway): https://trustworthy.systems/publications/nicta_full_text/8758.pdf

Some historic remarks, that might give some insight why things are as they are. The motivation for AutoCorres was the C-verification tasks coming from the sel4 Projects https://sel4.systems, a verified microkernel, written in C.

The project started with high level HOL-specifications, refining them to monadic functions (within HOL and Haskell) and then refining them to C-code. The nondetermistic state-monad goes back to Cook. et al http://www.cse.unsw.edu.au/~kleing/papers/Cock_KS_08.pdf. To link the C-code with the monadic functions within Isabelle / HOL the C-parser was written. The C-parser produces SIMPL programs, and SIMPL is equipped with various semantics (big and small-step) and a Hoare-logic framework including a verification condition generator. The correspondence of the C-functions represented in SIMPL and the monadic functions was manually expressed within this framework.

AutoCorres was then built after (or in parallel) to this verification effort in order to speed up the process for future projects.

While the main concepts described in the publications above are still valid, a lot of things have evolved and were extended. Especially the implementation details have changed. In this document we also give some notes on these differences. Also see `../NEWS`.

26.2 C Parser

Some remarks on the C-Parser

- Lexical and syntactical analysis is implemented using ML-Lex / ML-Yacc coming from the MLton project http://mlton.org/. Originally mlton was used to generate the ML files from the grammers. Meanwhile this is all integrated into Isabelle/ML.

- The C parser first generates an ML-data structure representing the program. It then does some transformations on that data structure (e.g. storing results of nested function calls into temporary variables), before translating it to SIMPL. In roughly the following steps:

  1. Generating HOL-types for the C-types.
     - The C parser performs a complete program analysis to determine the types used.
     - Defines the types and performs the UMM proofs, in particular conversions and properties to and from raw byte lists.
  2. Defining the state-space type: global variables, local variables, heap variables. The C parser does a complete program analysis to determine how global and heap variables are used to decide

whether to model them as part of the globals-record or the heap. When no 'address-of' a variable is calculated it is stored as part of the global variable record, otherwise it is stored within the heap. Global variables are more abstract to deal with compared to heap variables.

3. Defining the SIMPL procedures
4. Proving 'modifies' specifications for the procedures, i.e. frame conditions on global variables.

26.3 AutoCorres

AutoCorres abstracts the SIMPL program to a monadic HOL function in various phases, producing correspondence (a.k.a. simulation, a.k.a. refinement) theorems along the way. Note that the translation (and the correspondence theorems) might only be partial in the following sense: A phase might introduce additional guards into the code. Simulation only holds for programs that do not fail on the abstract execution. Note that nontermination is currently also modelled as failure. So simulation only holds for terminating abstract programs:

- Correspondence relation: *ac-corres*

Each phase is processed in similar stages:

- Raw translation and refinement-proof
- Optimisation of the output program, sometimes distinguished between a more lightweight "peephole" optimisation and a more heavyweight "flow" optimisation
- Define a constant for the output of the translation for that phase.

  The phases are:

- From SIMPL (deep embedding of statements, shallow embedding of expressions) to L1 (shallow embedding of statements and expressions). The transformation preserves the state-space representation, and merely focuses on translating SIMPL statements to monadic functions:
  - Definition of L1: `../L1Defs.thy`
  - Correspondence relation: *L1corres*
  - Peephole optimisation: `../L1Peephole.thy`
  - Flow optimisation: `../ExceptionRewrite.thy`

  - Main ML file: `../simpl_conv.ML`

- Local variable abstraction, from L1 to L2. In L2, local variables are represented as lambda abstractions. So the underlying state-space type changes, getting rid of the local variable record. As local variables are now treated as lambda abstractions, and thus names become meaningless, autocorres keeps the original names around as part of (logically redundant) name annotations within the L2 constants, e.g. *L2-gets* (λ-. x) [*c-source-name-hint*]. Lists of names are used, as variables might pile up in tuples. These annotations are removed in the final polishing phase of autocorres, attempting to rename the bound variables accordingly on the fly.

  - Definition of L2: `../L2Defs.thy`
  - Correspondence relation: *L2corres*
  - Exception rewriting, introducing nested exceptions: `../L2ExceptionRewrite.thy`
  - Peephole optimisation: `../L2Peephole.thy`
  - Main ML files: `../local_var_extract.ML`, `../l2_opt.ML` From now on we stay in the language L2, and also the optimisation stages are reused.

- In/Out parameter abstraction (IO). Pointer parameters are replaced by passing value parameters. This step may eliminate pointers to local variables.

  - Theory: `../In_Out_Parameters.thy`
  - Correspondence relation: *IOcorres*
  - Documentation: `In_Out_Parameters_Ex.thy`

- Heap abstraction, referred to as heap-lifting (HL). The monolithic-byte-oriented heap is abstracted to a typed-split-heap model. For that the new type *lifted-globals* is introduced. Note, that the separation logic of Tuch also attempts to give a typed view on top of the byte-level-heap. In case of nested structure types it even is capable to express the various levels of typed-heap views from bytes to individual structure fields to (nested) structures. AutoCorres takes a simpler, more pragmatic approach here and only provides a single split-heap abstraction on the level of complete types. This means there is a heap for every compound type and for every primitive type but not for nested structures. However, autocorres allows to mix between the abstractions layers (split-heap vs. byte-heap) on the granularity of functions. So you can model low-level functions like memory-allocators on the

untyped byte-heap and then still use these functions within other functions in the split-heap abstraction. Meanwhile, genuine support for a split-heap approach with addressable nested structures was added. It is described in `open_struct.thy`.

  - Main theory: `../HeapLift.thy`
  - Correspondence relation: *L2Tcorres*
  - Main ML files: `../heap_lift_base.ML`, `../heap_lift.ML`

- Word abstraction (WA): (un)signed n-bit words are converted to *int* or *nat*. This conversion only affects local variable (lambda abstraction), so the underlying state-space type is maintained. When reading and storing to memory the values are converted accordingly.

  - Main theory: `../WordAbstract.thy`
  - Correspondence relation: *corresTA*
  - Main ML file: `../word_abstract.ML`

- Type strengthening (TS), sometimes also referred to as type lifting or type specialisation: Best effort approach to simplify the structure of the monad as far as possible:

  - Pure functions
  - Reader monad, read-only state-dependant functions.
  - Option (reader) monad, in particular for potential non-terminating functions (recursion / while). Nontermination is treated as failure in autocorres and failure of guards are represented as a result of *None*
  - Nondeterministic state monad (exceptional control flow confined within function boundaries). Failure (of guards) and nontermination are represented by *Failure* as outcome.
  - Nondeterministic state monad with exit (exceptional control flow crossing function boundaries). New monad types can be registered within Isabelle / ML.
  - Main theories: `../TypeStrengthen.thy`, `../Refines_Spec.thy`
  - Correspondence relation: *refines*. Originally this was actually based on equality. As we now implement (mutually) recursive functions by a CCPO **fixed-point** instead of introducing an explicit measure parameter we changed to simulation. Equality is not *ccpo.admissible*, whereas simulation is.
  - Main ML files: `../type_strengthen.ML`, `../monad_types.ML`, `../monad_convert.ML`

- Polish. Performed as a part of type strengthening. Here remaining special and limited L2 definitions are expanded to 'ordinary' monadic functions. Also annotations of original c variable names are removed and bound variables are named accordingly.

26.4 AutoCorres Flow

The original AutoCorres implementation of David Greenaway was refined in several ways. In particular from a high level perspective the concepts of incremental build and parallel processing where unified:

- Parallel processing is moved to the outermost invocation of autocorres. All tasks are calculated respecting the dependencies of the call graph and the various autocorres phases, cf. `AutoCorres.parallel_autocorres`.
- Every task then invokes a distinct call to `AutoCorres.do_autocorres` exactly specifying the scope (which function clique) and which phase via the options.
- The results of an invocation are maintained in generic data, such that they are available to subsequent dependent calls.

  The common parts of each phase are implemented in `AutoCorresUtil`. The last phase, Type strengthening (TS), historically played a special role and still does not make much use of `AutoCorresUtil`, but conceptually it follows the same idea:

  Functions are processed in a sequence of cliques, from the bottom of the call graph to the top. Here a clique is either a single function or a group of strongly connected recursive calls. After processing of a phase the clique might get splitted due to dead code elimination. This general idea is implemented in `AutoCorresUtil.convert_and_define_cliques`.

26.4.1 General Remarks

- The main autocorres files, putting everything together: `../AutoCorres.thy`, `../autocorres.ML`
- There are two main approaches in a single phase to come up with an abstract program (output of the phase) and the correspondence proof to the concrete program (input of the phase):
  - Implicit synthesis of the abstract program from the concrete program by applying "intro" rules, where the abstract program is initialised with a schematic variable. Prototypical examples are the heap abstraction, word abstraction and type strengthening.

  - First explicitly calculate the abstract program (fragment) within ML from the concrete program (fragment), and then perform the correspondence proof. An prototypical example is the local variable abstraction.

- The common aspects of the various stages within a single phase is (partially) generalised into library functions: `../autocorres_util.ML`

26.4.2 Links to more documentation / examples

- Pointers to local variables: `pointers_to_locals.thy`
- In-Out parameters: `In_Out_Parameters_Ex.thy`
- Addressable fields and open types: `open_struct.thy`
- Function pointers: `fnptr.thy`
- Unions: `union_ac.thy`

26.5 Overview on the Locales

The representation of local variables in SIMPL and L1 where changed from records to 'Statespaces' as implemented by **statespace** and finally to positional variables as in *locals* in `../c-parser/CLocals.thy`. This allows for an uniform representation of local variables as a function from 'nat to byte list', Typing is achieved by ML data organised in locales and bundles, cf. `../c-parser/CTranslationInfrastructure.thy`. These are also used to do the L1 and the L2 correspondence proofs. Starting from L2 on the local variables are represented as lambda bindings.

We describe the various locales (and bundles) later on with our running examples.

To support function pointers even more locales where introduced. See `fnptr.thy`.

26.6 Example Program

declare [[*c-parser-feedback-level=1*]]
install-C-file *autocorres-infrastructure-ex.c*
print-theorems
thm *upd-lift-simps*
thm *fl-ti-simps*

The variables consist of parameters (input and output) and the local variables. Note that this is merely a ML-declaration of the scope which can be accesses via a bundle, cf. `../c-parser/CTranslationInfrastructure.thy`.

context includes *add-variables*
begin
term *´n :== 42*
end

Adresses of global variables or function pointers are kept in the *global-addresses* locale.

print-locale *autocorres-infrastructure-ex-global-addresses*

This is everything necessary to define the body of a function. All bodies are defined in that locale.

context *autocorres-infrastructure-ex-global-addresses*
  opening *add-variables*
begin
term *add-body*
thm *add-body-def*
end

The implementation locale holds the defining equation of the function in SIMPL and is closed under callees.

context *call-add-impl*
begin
thm *add-impl*
thm *call-add-impl*
end

In case of a clique there is a clique locale and aliases for each function.

print-locale *even-odd-impl*
print-locale *even-impl*
print-locale *odd-impl*

The locale importing all implementations is named like the file with suffix *-simpl*.

print-locale *autocorres-infrastructure-ex-simpl*

26.6.1 Incremental Build

autocorres [*phase=L1*, *scope=add*] *autocorres-infrastructure-ex.c*
autocorres [*phase=L2*, *scope=add*] *autocorres-infrastructure-ex.c*
autocorres [*phase=HL*, *scope=add*] *autocorres-infrastructure-ex.c*
autocorres [*phase=WA*, *scope=add*] *autocorres-infrastructure-ex.c*
autocorres [*phase=TS*, *scope=add*] *autocorres-infrastructure-ex.c*

26.6.2 All the rest

autocorres *autocorres-infrastructure-ex.c*

context *unsigned-to-signed-impl*
begin
thm *unsigned-to-signed-body-def*
end
context *autocorres-infrastructure-ex-all-corres*
begin
thm *unsigned-to-signed′-def*

The SIMPL versions produced by c-parser

declare [[*clocals-short-names*]]
thm *max-body-def*
thm *add-body-def*
term *add-body*::((*globals*, *locals*, *32 signed word*) *CProof.state*, *unit ptr*, *strictc-errortype*) *com*
thm *call-add-body-def*
thm *seq-assign-body-def*
thm *call-seq-assign-body-def*
thm *inc-g-body-def*
thm *deref-body-def*
thm *deref-g-body-def*
thm *factorial-body-def*
thm *call-factorial-body-def*
thm *odd-body-def*
thm *even-body-def*
thm *dead-f-body-def*
thm *dead-h-body-def*
thm *dead-g-body-def*

L1 versions. Monadic with same state. Note that recursive procedures are defined with the **fixed-point** package and hence the *.simps* instead of *-def* makes more sense to look at. In this layer a first exception elimination optimisation takes places. The optimized definition has the extension "opt".

declare [[*show-types=false*]]
thm *l1-max′-def*
thm *l1-opt-max′-def*

term *l1-add′*::(*unit*, *unit*, (*globals*, *locals*, *32 signed word*) *CProof.state*) *exn-monad*
thm *l1-add′-def*
thm *l1-opt-add′-def*

thm *l1-call-add′-def*
thm *l1-opt-call-add′-def*

thm *l1-seq-assign′-def*
thm *l1-opt-seq-assign′-def*

thm *l1-call-seq-assign'-def*
thm *l1-opt-call-seq-assign'-def*

thm *l1-inc-g'-def*
thm *l1-opt-inc-g'-def*

thm *l1-deref'-def*
thm *l1-opt-deref'-def*

thm *l1-deref-g'-def*
thm *l1-opt-deref-g'-def*

thm *l1-factorial'.simps*
thm *l1-opt-factorial'-def*

thm *l1-call-factorial'-def*
thm *l1-opt-call-factorial'-def*

thm *l1-odd'.simps*
thm *l1-opt-odd'-def*

thm *l1-even'.simps*
thm *l1-opt-even'-def*

thm *l1-dead-f'.simps*
thm *l1-opt-dead-f'-def*

thm *l1-dead-h'.simps*
thm *l1-opt-dead-h'-def*

thm *l1-dead-g'.simps*
thm *l1-opt-dead-g'-def*

L2 version: lambda bindings for local variables

thm *l2-max'-def*
term *l2-add'*::*32 signed word* ⇒ *32 signed word*
⇒ (*32 signed word c-exntype*, *32 signed word*, *globals*) *exn-monad*
thm *l2-add'-def*
thm *l2-call-add'-def*
thm *l2-seq-assign'-def*
thm *l2-call-seq-assign'-def*
thm *l2-inc-g'-def*
thm *l2-deref'-def*
thm *l2-deref-g'-def*
thm *l2-factorial'.simps*
thm *l2-call-factorial'-def*
thm *l2-odd'.simps*
thm *l2-even'.simps*
thm *l2-dead-f'.simps*

thm *l2-dead-h'.simps*
thm *l2-dead-g'-def* — Note that *l2-dead-g'* is no longer part of the clique, because of dead code elimination.

Heap abstraction. Note that the change in the heap component of the global variables. The type changes to *lifted-globals* instead of *globals*.

print-record *globals* — contains byte level tagged heap: *t-hrs-'*
print-record *lifted-globals* — contains split heap with single component *heap-w32*, as we only have a pointers to unsigend.

thm *hl-max'-def*
term *hl-add'*:: *32 signed word* ⇒ *32 signed word*
⇒ (*32 signed word c-exntype*, *32 signed word*, *lifted-globals*) *exn-monad*
thm *hl-add'-def*
thm *hl-call-add'-def*
thm *hl-seq-assign'-def*
thm *hl-call-seq-assign'-def*
thm *hl-inc-g'-def*
thm *hl-deref'-def*
thm *hl-deref-g'-def*
thm *hl-factorial'.simps*
thm *hl-call-factorial'-def*
thm *hl-odd'.simps*
thm *hl-even'.simps*
thm *hl-dead-f'.simps*
thm *hl-dead-h'.simps*
thm *hl-dead-g'-def*

Word abstraction.

thm *wa-max'-def*
term *wa-add'*::*int* ⇒ *int* ⇒ (*32 signed word c-exntype*, *int*, *lifted-globals*) *exn-monad*
thm *wa-add'-def*
thm *wa-call-add'-def*
thm *wa-seq-assign'-def*
thm *wa-call-seq-assign'-def*
thm *wa-inc-g'-def*
thm *wa-deref'-def*
thm *wa-deref-g'-def*
thm *wa-factorial'.simps*
thm *wa-call-factorial'-def*
thm *wa-odd'.simps*
thm *wa-even'.simps*
thm *wa-dead-f'.simps*
thm *wa-dead-h'.simps*
thm *wa-dead-g'-def*

Final definition.

term *max'*::*int* ⇒ *int* ⇒ *int*
thm *max'-def*

term *add′*:: *int* ⇒ *int* ⇒ *lifted-globals* ⇒ *int option*
thm *add′-def*
thm *wa-call-add′-def*
thm *wa-seq-assign′-def*
thm *wa-call-seq-assign′-def*
thm *wa-inc-g′-def*
thm *wa-deref′-def*
thm *wa-deref-g′-def*
thm *wa-factorial′.simps*
thm *wa-call-factorial′-def*
thm *wa-odd′.simps*
thm *wa-even′.simps*
thm *wa-dead-f′.simps*
thm *wa-dead-h′.simps*
thm *wa-dead-g′-def*

Correspondence theorem.

thm *factorial′-ac-corres*
thm *call-factorial′-ac-corres*

end

26.7 Simplification strategy and dealing with tuples in L2-optimization phases

Consider a congruence rule from AutoCorres

lemma
assumes *c-eq*: ⋀*r s*. *c r s* = *c′ r s*
assumes *bdy-eq*: ⋀*r s*. *c′ r s* ⟹ *run* (*A r*) *s* = *run* (*A′ r*) *s*
shows *L2-while c A* = *L2-while c′ A′*
using *c-eq bdy-eq*
by (*rule L2-while-cong*)

Note that *r* could be a tuple and and we would like to "split" it. The condition *c* will already be formulated with *case-prod*, e.g. λ(*x*, *y*, *z*) *s*. *x* < *y*.

When the congruence rule above is applied, variable *?c′* has to be instantiated at some point. First the congruence rule is applied as is. Then we use *split-paired-all* to split up the *r*. For the example we will end up with the subgoal:

⋀*a b c s*. (*a* < *b*) = *?c′* (*a*, *b*, *c*) *s*

Mere Unification will fail here as it does not work "modulo" *case-prod*. To find the proper instantiation we developed `Tuple_Tools.tuple_inst_tac` which can be added to the simplifier as a looper. It will instantiate *?c′* with the respective *case-prod* instance introducing a new schematic variable: *?c′* = (λ(*a*, *b*, *c*). *?f a b c*). Now the unification will kick in and do the rest of the work.

However, allthough an instantiation is now found, it does not have the expected effect on `body_eq`. The problem is, that the premise of the implication is not simplified as it is added to the "prems" of the simplifier so it will be (*case* (*a*, *b*, *c*) *of* (*a*, *b*, *c*) ⇒ λ*s*. *a* < *b*) *s* ≡ *True* and not just *a* < *b* ≡ *True*". The former format is unfortunately ineffective in the simplification of the while body.

Fortunately the simpset can be instructed to apply a function to the premise before it is added. So adding `Tuple_Tools.mksimps` with `Simplifier.set_mksimps` helps with that respect. As the simplifier descends into the term it calls the function to do "beta-reduction" of tuple application before adding the premise.

lemma *L2-while* (λ(*x*,*y*,*z*) *s*. *0* < (*y*::*nat*)) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *0* < *y*)) (λ-. *X*)) *x ns*
=
L2-while (λ(*x*,*y*,*z*) *s*. *0* < *y*) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *True*)) (λ-. *X*)) *x ns*
apply (*tactic* ‹
simp-tac (@{*context*}
addloop (*tuple-inst-tac*, *Tuple-Tools.tuple-inst-tac*)
addsimps @{*thms split-paired-all*}
|> *Simplifier.add-cong* @{*thm L2-seq-guard-cong*}
|> *Simplifier.add-cong* @{*thm L2-while-cong*}
|> *Simplifier.set-mksimps* (*Tuple-Tools.mksimps*)
) *1*›)
done

Unfortunately there is still an issue. Linear arithmetic is applied as a simproc by the simplifier, e.g. *0* < *n* ⟹ *Suc 0* ≤ *n*

Unfortunately this does not work as expected in the setup above:

unbundle *clocals-string-embedding*

lemma *L2-while* (λ(*x*,*y*,*z*) *s*. *0* < (*y*::*nat*)) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *Suc 0* ≤ *y* ∧ *z* = *a*)) (λ-. *X*)) *x ns*
=
L2-while (λ(*x*,*y*,*z*) *s*. *0* < *y*) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *z* = *a*)) (λ-. *X*)) *x ns*
apply (*tactic* ‹
simp-tac (@{*context*}
addloop (*tuple-inst-tac*, *Tuple-Tools.tuple-inst-tac*)
addsimps @{*thms split-paired-all*}
|> *Simplifier.add-cong* @{*thm L2-seq-guard-cong*}
|> *Simplifier.add-cong* @{*thm L2-while-cong*}
|> *Simplifier.set-mksimps* (*Tuple-Tools.mksimps*)
) *1*›)
oops

After some investigation in the interplay between the simplifier and the linear arithmetic simproc it seems that the linear arithmetic somehow ig-

nores the transformed theorem that `Tuple_Tools.mksimps` yields. Without understanding all the details I guess the reason is the missmatch between the term in the hyps and the prop of the resulting theorem. As the simplifier adds descends into an implication $P \Longrightarrow Q$ it generates a theorem from P, by also adding P to the hyps, so we have $P\ [P]$ as a theorem. When `Tuple_Tools.mksimps` kicks in it only simplifies the prop not the hyps. So we end up with $a < b \equiv True\ [(\lambda(a,\ b,\ c)\ s.\ a < b)\ (a,\ b,\ c)\ s \equiv True]$ instead of $a < b \equiv True\ [a < b \equiv True]$. This seems to irritate the linear arithmetic. One solution could be to also simplify the hyps. We did not explore this path, as I would expect some issues when the hyps are eventually discharged. Nevertheless there could be a solution following that path.

We came up with a different solution. We get rid of the dependency of `Tuple_Tools.mksimps` by using $P\ =simp=>\ Q$ to trigger simplification of P.

Note the various options to trace the simplification process.

```
declare [[linarith-trace=false]]
declare [[simp-trace=false, simp-trace-depth-limit=100]]
declare [[simp-debug=false]]
declare [[show-hyps=false]]

lemma
  assumes c-eq: ⋀r s. c r s = c' r s
  assumes bdy-eq: ⋀r s. c' r s =simp=> run (A r) s = run (A' r) s
  shows L2-while c A = L2-while c' A'
  using c-eq bdy-eq by (rule L2-while-simp-cong)

lemma L2-while (λ(x,y,z) s. 0 < (y::nat)) (λ(x,y,z). L2-seq (L2-guard (λ-. Suc 0
≤ y ∧ z = a)) (λ-. X)) x ns
      =
       L2-while (λ(x,y,z) s. 0 < y) (λ(x,y,z). L2-seq (L2-guard (λ-. z = a)) (λ-.
X)) x ns
apply (tactic ‹
simp-tac ( @{context}
          addloop (tuple-inst-tac, Tuple-Tools.tuple-inst-tac)
          addsimps @{thms split-paired-all}
          |> Simplifier.add-cong @{thm L2-seq-guard-cong}
          |> Simplifier.add-cong @{thm L2-while-simp-cong}
) 1›)
  done
```

The more standard congruence rule also works fine.

```
lemma
  assumes c-eq: c = c'
  assumes bdy-eq: ⋀r s. c' r s =simp=> run (A r) s = run (A' r) s
  shows L2-while c A = L2-while c' A'
  using c-eq bdy-eq by (rule L2-while-simp-cong')
```

lemma *L2-while* (λ(*x*,*y*,*z*) *s. 0* < (*y*::*nat*)) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *Suc 0* ≤ *y* ∧ *z* = *a*)) (λ-. *X*)) *x ns*
= *L2-while* (λ(*x*,*y*,*z*) *s. 0* < *y*) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *z* = *a*)) (λ-. *X*)) *x ns*
apply (*tactic* ‹
simp-tac (@{*context*}
addloop (*tuple-inst-tac*, *Tuple-Tools.tuple-inst-tac*)
addsimps @{*thms split-paired-all*}
|> *Simplifier.add-cong* @{*thm L2-seq-guard-cong*}
|> *Simplifier.add-cong* @{*thm L2-while-simp-cong'*}
) *1*›)
done

To avoid repeated splitting (and diving into the subterms) of tuples with (⋀*x*. *PROP ?P x*) ≡ (⋀*a b*. *PROP ?P* (*a*, *b*)) consider the follwing simproc setup.

lemma *L2-while* (λ(*x*,*y*,*z*) *s. 0* < (*y*::*nat*)) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *Suc 0* ≤ *y* ∧ *z* = *a*)) (λ-. *X*)) *x ns*
= *L2-while* (λ(*x*,*y*,*z*) *s. 0* < *y*) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *z* = *a*)) (λ-. *X*)) *x ns*
apply (*tactic* ‹
simp-tac (@{*context*}
addloop (*tuple-inst-tac*, *Tuple-Tools.tuple-inst-tac*)
addsimprocs [*Tuple-Tools.split-tupled-all-simproc*, *Tuple-Tools.tuple-case-simproc*]
delsimps @{*thms Product-Type.prod.case Product-Type.case-prod-conv*}
|> *Simplifier.add-cong* @{*thm L2-seq-guard-cong*}
|> *Simplifier.add-cong* @{*thm L2-while-simp-cong'*}
) *1*›)
done

This is also the setup of `L2Opt.cleanup_ss`

lemma *L2-while* (λ(*x*,*y*,*z*) *s. 0* < (*y*::*nat*)) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *Suc 0* ≤ *y* ∧ *z* = *a*)) (λ-. *X*)) *x ns*
= *L2-while* (λ(*x*,*y*,*z*) *s. 0* < *y*) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *z* = *a*)) (λ-. *X*)) *x ns*
apply (*tactic* ‹
asm-full-simp-tac (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.HL FunctionInfo.PEEP*) *1*›)
done

lemma
assumes *XX-def*: *XX* = *L2-while* (λ(*x*, *y*, *z*) *s. 0* < *y*) (λ(*x1*, *x2*, *x3*). *L2-seq* (*L2-guard* (λ-. *Suc 0* ≤ *y1* ∧ *z1* = *a*)) (λ-. *X*)) *x ns*
shows *L2-while* (λ(*x*,*y*,*z*) *s. 0* < (*y*::*nat*)) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *Suc*

```
0 ≤ y1 ∧ z1 = a)) (λ-. X)) x ns
      =
       XX
  apply (tactic ‹
    asm-full-simp-tac (L2Opt.cleanup-ss @{context} [] FunctionInfo.HL Function-
Info.PEEP) 1›)
    apply (tactic ‹
   asm-full-simp-tac (Simplifier.clear-simpset @{context} addsimprocs [@{simproc
ETA-TUPLED}]) 1›)
  apply (subst XX-def)
  apply (rule refl)
  done

lemma L2-while (λ(x, y, z) s. y = 0)
     (λ(x, y, z).
        L2-seq (L2-gets (λs. y) [S ''ret'']) (λr. XXX21 r x))
     names =
    L2-while (λ(x, y, z) s. y = 0) (λ(x, y, z). L2-seq (L2-gets (λs. 0) [S ''ret'']))
(λr. XXX21 r x)) names
apply (tactic ‹
   asm-full-simp-tac (L2Opt.cleanup-ss @{context} [] FunctionInfo.HL Function-
Info.PEEP) 1›)
  done


lemma PROP SPLIT (⋀r. ((λ(x,y,z). y < z ∧ z=s) r) ⟹ P r)
 ≡ (⋀x y z. y < z ∧ z = s ⟹ P (x, y, s))
  apply (tactic ‹
asm-full-simp-tac (put-simpset HOL-basic-ss @{context}
 addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc] |> Sim-
plifier.add-cong @{thm SPLIT-cong}) 1
›)
  done

experiment
begin

schematic-goal foo:
⋀x1 x2 x3 s. (case (x1, x2, x3) of (x, y, z) ⇒ λs. 0 < y) s ⟹
 (case (x1, x2, x3) of (x, y, z) ⇒ L2-seq (L2-guard (λ-. Suc 0 ≤ y ∧ z = a)) X)
= ?A' (x1, x2, x3) s
  apply (tactic ‹
asm-full-simp-tac ( @{context}
          addloop (tuple-inst-tac, Tuple-Tools.tuple-inst-tac)
          addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc]
          delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}

) 1›)
  done
```

?A′ should be properly instantiated.

thm *foo*

end

```
declare [[simp-trace=true, simp-trace-depth-limit=100]]
lemma L2-while (λ(x,y,z) s. 0 < (y::nat)) (λ(x,y,z). L2-seq (L2-guard (λ-. Suc 0
≤ y ∧ z = a)) X) x ns
      =
       L2-while (λ(x,y,z) s. 0 < y) (λ(x,y,z). L2-seq (L2-guard (λ-. z = a)) X) x
ns
apply (tactic ‹
asm-full-simp-tac ( @{context}
          addloop (tuple-inst-tac, Tuple-Tools.tuple-inst-tac)
          addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc]
          delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}
          |> Simplifier.add-cong @{thm L2-seq-guard-cong}
          |> Simplifier.add-cong @{thm L2-while-cong-simp-split}
          |> Simplifier.add-cong @{thm SPLIT-cong}
) 1›)
  done
```

Finally the following setup implements a very controlled tuple esplitting on the While body.

```
lemma
  assumes c-eq: c = c′
  assumes bdy-eq: PROP SPLIT (⋀r s. c′ r s ⟹ run (A r) s = run (A′ r) s)
  shows L2-while c A = L2-while c′ A′
  using c-eq bdy-eq by (rule L2-while-cong-split)
```

With *SPLIT* we mark a position in the term that will trigger `Tuple_Tools.SPLIT_simproc`. By adding *SPLIT PROP ?P ≡ SPLIT PROP ?P* as congruence rule we prohibit the simplifier from diving into the loop body before we actually split the result variable. Note that the simplifier usually works bottom up, only congruence rules are applied top down.

```
lemma L2-while (λ(x,y,(z::nat)) s. 0 < (y::nat) ∧ y=x) (λ(x,y,z). L2-seq (L2-guard
(λ-. Suc 0 ≤ y ∧ z = a)) X) x ns
      =
  L2-while (λ(x, y, z) s. 0 < y ∧ y = x) (λ(x, x, z). L2-seq (L2-guard (λ-. z = a))
X) x ns
apply (tactic ‹
asm-full-simp-tac ( @{context}
          addloop (tuple-inst-tac, Tuple-Tools.tuple-inst-tac)
          addsimprocs [Tuple-Tools.SPLIT-simproc, Tuple-Tools.tuple-case-simproc]
          delsimps @{thms Product-Type.prod.case Product-Type.case-prod-conv}
          |> Simplifier.add-cong @{thm L2-seq-guard-cong}
          |> Simplifier.add-cong @{thm L2-while-cong-simp-split}
          |> Simplifier.add-cong @{thm SPLIT-cong}
```

) *1*›)
done

26.8 Simplification of conditions (guards, loops, conditionals)

The basic peephole optimization for conditions tries to simplify "trivial" guards / conditions by propagating the information of guards / conditions to subsequent guards / conditions. It is called "peephole" optimisation (in contrast to "flow"-sensitive), because it only propagates the state independent information of gaurds, i.e. constraints on constant expressions or local variables, not constraints on the state itself.

To facilitate this propagation with congruence rules and simprocs, we introduce two special constants *L2-seq-gets* and *L2-seq-guard* that are introduced as intermediate step by a conversion. With this marking we can distinguish the cases by congruence rules, without the marking both cases would be subject to a single congruence rule for *L2-seq*. (Note that the simplifier only considers the head term of a congruence rules.)

We add the congruence rules $?c = ?c' \Longrightarrow$ *L2-seq-gets* $?c\ ?n\ ?A \equiv$ *L2-seq-gets* $?c'\ ?n\ ?A$ and $[\![?P = ?P'; \bigwedge s.\ ?P'\ s \Longrightarrow run\ (?X\ ())\ s = run\ ?X'\ s]\!] \Longrightarrow$ *L2-seq-guard* $?P\ ?X =$ *L2-seq-guard* $?P'\ (\lambda\text{-}.\ ?X')$. The congruence rules for the guard is a standard congruence rules that adds the gaurd as a precondition for simplifying the second statement in the sequential composition. The congruence rule for $?c = ?c' \Longrightarrow$ *L2-seq-gets* $?c\ ?n\ ?A \equiv$ *L2-seq-gets* $?c'\ ?n\ ?A$ stops the simplifier from descending into the second term of the sequential composition. Instead the simproc `L2Opt.l2_marked_gets_bind_simproc` is triggered to analyse the situation. It does the following:

- If the returned value in the first statement is simple, only occurs once in second statement, or is a structure-constructor or update that is only appied to structure-selections then the value is directly propagated to the second statement and the sequential composition is removed.

- It peeks into the prems of the simplifier to see if there are already "interesting" facts collected from the congruence rules of guards / conditions. If not, it unfolds the marking and continues with the ordinary sequential composition by simplifying the second statement. "Interesting" means there is at least one premise that has impact on the new return value. E.g. if we have $x + y < (5::'a)$ in the prems, and we now we return $x + y$, than the simproc introduces a new variable r for return value, augments the context with the equation $r = x + y$ and derives the new premise $r < (5::'a)$ which is put to the premises of the simplifier as well as the simpset. Then the simplifier is called

recursively on the second statement of the sequential composition with the augmented context. When it is finished it uses the rule *?f ?c ≡ ?g ⟹ L2-seq-gets ?c ?n ?f ≡ STOP (L2-seq-gets ?c ?n (λ-. ?g))* to introduce the constant *STOP*. The congruence rule *STOP ?P ≡ STOP ?P* prohibits the simplifier to descend down into the just simplified second statement again.

Note that this setup works around the limitation of the simplifier that does not allow us to modify the context by something like a "cong-proc", similar to a "simproc". Keep in mind that a congruence rule is applied top-down as the simplifier works its way down into a term, whereas a simpproc (or any other simplification rule) is applied buttom up. So a simproc / simplification rule can build on the fact that the subterms are already normalised. The simplifier makes use of this by doing some bookkeeping with "term-skeletons" to avoid resimplification of subterms. This mechanism fails short in our setup of using a congruence rule to stop simplification of the second statement and calling the simproc instead. That is why we explicitly introduce *STOP*. It will be removed after simplification by an additional traversal of the term.

declare [[*simp-trace=false*, *ML-print-depth=1000*]]

Propagation of simple constant by unfolding.

lemma *L2-seq-gets c* [$\mathcal{S}$ *''r''*] (λ*r*. (*L2-guard* (λ-. *P r*))) =
L2-guard (λ-. *P c*)
by (*tactic* ‹*simp-tac* (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.L2 FunctionInfo.PEEP*) *1*›)

Propagation of term, as it only appears once in the second statement.

lemma *L2-seq-gets* (*c* + *d*) [$\mathcal{S}$ *''r''*] (λ*r*. (*L2-guard* (λ-. *P r*))) =
L2-guard (λ-. *P* (*c* + *d*))
by (*tactic* ‹*simp-tac* (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.L2 FunctionInfo.PEEP*) *1*›)

As nothing is in the prems yet, marking is just removed and second statement is thus simplified

lemma *L2-seq-gets* (*c* + *d*) [$\mathcal{S}$ *''r''*] (λ*r*. (*L2-guard* (λ-. *P r* ∧ (*P r* ⟶ *Q r*))))
=
STOP (*L2-seq-gets* (*c* + *d*) [$\mathcal{S}$ *''r''*] (λ*r*. *L2-guard* (λ-. *P r* ∧ *Q r*)))
by (*tactic* ‹*simp-tac* (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.L2 FunctionInfo.PEEP*) *1*›)

The first guard condition is propagated to the second guard, via the intermediate assignment *r* = *a* + *b*.

lemma *L2-seq-guard* (λ-. *a* + *b* < *5*)
(λ-. *L2-seq-gets* (*a* + *b*) [$\mathcal{S}$ *''r''*]

(λr. L2-guard (λ-. r < 5 ∧ P))) =
L2-seq-guard (λ-. a + b < 5)
(λ-. L2-guard (λ-. P))
by (*tactic* ‹*simp-tac* (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.L2 FunctionInfo.PEEP*) *1*›)

ML ‹
fun assert-rhs thm =
Tuple-Tools.assert-cterm' @{*context*} (*Thm.term-of* (*Thm.rhs-of thm*))

›
ML-val ‹
val ct = @{*cterm L2-seq-guard* (λ-. (*a*::*int*) + *b* < *5*)
(λ-. *L2-seq-gets* (*a* + *b*) [$\mathcal{S}$ ''*r*''] (λr. (*L2-guard* (λ-. *r* < *5* ∧ *P*))))}
val test = *Simplifier.asm-full-rewrite* (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.L2 FunctionInfo.PEEP*) *ct*
val - = assert-rhs test
@{*cterm STOP* (*L2-seq-guard* (λ-. (*a*::*int*) + *b* < *5*) (λ-. *L2-guard* (λ-. *P*)))}
›

lemma *L2-seq-guard* (λs. *V1* + *a* − (*r*::*int*) ≤ *2*)
(λ-. *L2-seq* (*L2-condition* (λs. *CC*) (*L2-seq-guard* (λs. *V1* + *a* − *r* ≤ *3*) (λ-. *X1*)) *X2*) (λ-. *X3*)) =
L2-seq-guard (λs. *V1* + *a* − *r* ≤ *2*)
(λ-. *L2-seq* (*L2-condition* (λs. *CC*) (*L2-seq-guard* (λs. *True*) (λ-. *X1*)) *X2*) (λ-. *X3*))
by (*tactic* ‹*asm-full-simp-tac* (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.L2 FunctionInfo.PEEP*) *1*›)

ML-val ‹
val ct = @{*cterm L2-seq-guard* (λs. (*V1* + *a* − (*r*::*int*) ≤ *2*))
(λ-. *L2-seq* (*L2-condition* (λs. *CC*) (*L2-seq-guard* (λs. *V1* + *a* − *r* ≤ *3*) (λ-. *X1*)) *X2*) (λ-. *X3*))}

val test = *Simplifier.asm-full-rewrite* (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.L2 FunctionInfo.PEEP*) *ct*
val - = assert-rhs test
@{*cterm STOP* (*L2-seq-guard* (λs. *V1* + *a* − *r* ≤ *2*)
(λ-. *L2-seq* (*L2-condition* (λs. *CC*) (*STOP* (*L2-seq-guard* (λs. *True*) (λ-. *X1*))) *X2*) (λ-. *X3*)))}
›

ML ‹
val ct = @{*cterm L2-seq-guard*
(λ-. ((*g k*)::*int*) + (*a*::*int*) − *r* ≤ *n*)
(λ-.

(*L2-seq-guard* (λ*s*. (*g k*) + *a* − *r* ≤ *n* + *1*) (λ-. *X1*)))}

val test = *Simplifier.asm-full-rewrite* (*Variable.set-body true* (*L2Opt.cleanup-ss* @{*context*} [] *FunctionInfo.L2 FunctionInfo.PEEP*)) *ct*
val - = *assert-rhs test* @{*cterm STOP* (*L2-seq-guard* (λ-. *g k* + *a* − *r* ≤ *n*) (λ-. *STOP* (*L2-seq-guard* (λ*s*. *True*) (λ-. *X1*))))}
›

declare [[*simp-trace*=*false*, *linarith-trace*=*false*]]

26.9 Tricks to enforce first-order unification

ML-val ‹
val fo = *Utils.reify-comb-conv* @{*context*} @{*pattern ?a* $ (*?g* $ *?x*)} @{*cterm* (*f z*) (*k y*) }
val orig = *Utils.unreify-comb-conv* @{*context*} @{*pattern ?a* $ (*?g* $ *?x*)} (*Thm.rhs-of fo*)
val t = *Utils.dest-reified-comb* @{*cterm a* $ *b*}
›

In heap lifting phase:

thm *heap-abs-fo*

In word abstraction phase:

thm *abstract-val-fun-app*

lemma *abstract-val-fun-app′*:
⟦*abstract-val Q x id x′*; *abstract-val P f id f′* ⟧ ⟹
abstract-val (*P* ∧ *Q*) (*g* (*f x*)) *g* (*f′ x′*)
by (*simp add*: *abstract-val-def*)

schematic-goal *abstract-val ?Q ?f g* (*f′ a′*)
apply (*rule abstract-val-fun-app′*)

oops

`Utils.fo_arg_resolve_tac` does the trick for us.

schematic-goal *abstract-val ?Q ?f g* (*f′ a′*)
apply (*tactic* ‹*Utils.fo-arg-resolve-tac* @{*context*} @{*thm abstract-val-fun-app*} @{*context*} *1*›)
oops

It turned out that `Utils.fo_arg_resolve_tac` and other tactics like simplification can become quite slow when operating under a lot of bound variables. We introduced a `WordAbstract.thin_abs_var_tac` to remove unused premises and bound variables. It is triggered by *THIN* in the rules, e.g. ⟦*introduce-typ-abs-fn ?rx1.0*; *THIN* (*corresTA ?P ?rx1.0 ?ex ?L ?L′*);

THIN (⋀*r r′*. *abs-var r ?rx1.0 r′* ⟹ *corresTA* (*?Q r*) *?rx2.0 ?ex* (*?R r*) (*?R′ r′*))⟧ ⟹ *corresTA ?P ?rx2.0 ?ex* (*L2-seq ?L* (λ*r*. *L2-seq* (*L2-guard* (*?Q r*)) (λ-. *?R r*))) (*L2-seq ?L′ ?R′*).

schematic-goal
⋀*n′ s r r′ ra r′a sa*.

abs-var r id r′ ⟹
abs-var ra id r′a ⟹ *abstract-val* (*?Q57 r ra sa*) (*?b60 r ra sa*) *id* (*6* ∗ *r′a*)

apply (*tactic* ‹*WordAbstract.thin-abs-var-tac* @{*context*} *1*›)
oops

schematic-goal
⋀*n′ s r r′ ra r′a sa*.
abs-var n id n′ ⟹
abs-var r id r′ ⟹
abs-var ra id r′a ⟹ *abstract-val* (*?Q57 r ra sa*) (*?b60 r ra sa*) *id* (*6* ∗ *r′a*)

apply (*tactic* ‹*WordAbstract.thin-abs-var-tac* @{*context*} *1*›)
oops

26.10 Exception Rewriting

This section highlights some of the ideas to rewrite and optimise exceptions.

In SIMPL and L1 the cause for an exception is stored in the state record in field *global-exn-var′-′*. The handler then inspects the state to decide whether it is responsible or not, and either handles the exception or re-raises it. In L2 this is refined in several steps.

- The cause of the exception is moved out of the state and represented as an error result in the L2 monad.
- The check for the cause in the handler of *L2-catch* is resolved to a static nesting of *L2-try* instead. The static nesting depth is directly reflected in the depth of the sum that reflects the error value *Inl* (*Inl* ...). For local exceptions this sequence of *Inl* ends with an *Inr*, global exception (crossing function boundaries) end in an *Inl*.
- The tuple arity of intermediate bindings in statements like *L2-seq*, *L2-while* is optimised to only propagate values that are actually used later on. As a side effect of this step, unnecessary nondeterministic initialisation steps for local variables (especially the technical return variable) are removed. The initialisation becomes unnecessary if the variable is strictly assigned to before used. This is always the case for

the return variable, as *return x* is translated to an assignment to the return variable followed by raising the *Return* exception.

Note that the field *global-exn-var'-'* has a special status, it is neither part of *globals* nor *locals*. In function calls it is treated like a global variable to ensure that the exception passes function boundaries. However, in the local variable abstraction of phase L2 it is treated similar to local variable and abstracted to lambda bindings.

declare [[*verbose=3*]]
declare [[*verbose-timing = 3*]]

26.10.1 Preliminary examples illustrating the usage of *rel-spec-monad*

schematic-goal *rel-spec-monad* (=) (=)
(*L2-catch*
(*L2-seq*
(*L2-catch*

(*L2-while* (λ*r s*. *True*)
(λ*r*. *L2-condition* (λ*s*. *3* < *a*) (*L2-throw* (*Return*, (*1*::*int*)) [$\mathcal{S}$ *''global-exn-var''*, $\mathcal{S}$ *''ret''*])
(*L2-seq*
(*L2-condition* (λ*s*. *1* < *a*)
((*L2-throw* (*Nonlocal* (*1*::*nat*), *r*) [$\mathcal{S}$ *''global-exn-var''*, $\mathcal{S}$ *''ret''*])::(*nat c-exntype* × *int*, *int*, *'a*) *exn-monad*)
(*L2-throw* (*Break*, *r*) [$\mathcal{S}$ *''global-exn-var''*, $\mathcal{S}$ *''ret''*]))
(λ*ra*. *L2-gets* (λ*s*. *r*) [$\mathcal{S}$ *''ret''*])))
r [$\mathcal{S}$ *''ret''*])

(λ(*a*, *b*).
L2-condition (λ*s*. *a* = *Break*) (*L2-gets* (λ-. *b*) [$\mathcal{S}$ *''ret''*])
(*L2-throw* (*a*, *b*) [$\mathcal{S}$ *''global-exn-var''*, $\mathcal{S}$ *''ret''*])))
(λ*r*. *L2-throw* (*Return*, *2*) [$\mathcal{S}$ *''global-exn-var''*, $\mathcal{S}$ *''ret''*]))
(λ(*a*, *b*).
L2-condition (λ*s*. *is-local a*) (*L2-gets* (λ*s*. *b*) [$\mathcal{S}$ *''ret''*]) (*L2-throw* (*the-Nonlocal a*) [$\mathcal{S}$ *''global-exn-var''*])))

?A

supply *c-exntype.case-distrib* [**where** *h=from-xval*, *simp*] *prod.case-distrib* [**where** *h=from-xval*, *simp*]
from-xval-simps [*simp*] *c-exntype.case-cong* [*cong*]
apply (*simp add*: *cond-return2 if-c-exntype-cases*)
apply (*rule rel-spec-monad-rel-xvalI*)
apply (*rule rel-spec-monad-rel-xval-try-catch* [*split-tuple f arity*: *2*])
apply (*rule rel-spec-monad-L2-seq-rel-xval-same-split*)
apply (*rule rel-spec-monad-rel-xval-try-catch* [*split-tuple f arity*: *2*])
apply (*rule rel-spec-monad-L2-while-rel-xval-same-split*)

```
  apply (rule rel-spec-monad-eq-rel-xval-L2-condition)
   apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
    apply (rule rel-xval.Exn)
    apply simp
    apply (rule rel-sum-Inl)
    apply simp
    apply (rule rel-sum-Inr)
    apply (rule refl)
   apply (simp add: SANITIZE-NAMES-def)
  apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
   apply (rule rel-spec-monad-eq-rel-xval-L2-condition)
    apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
     apply (rule rel-xval.Exn)
     apply simp
     apply (rule rel-sum-Inl)
     apply simp
     apply (rule rel-sum-Inl)
     apply (rule refl)
    apply (simp add: SANITIZE-NAMES-def)
   apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
    apply (rule rel-xval.Exn)
    apply simp
    apply (rule rel-sum-Inr)
    apply (rule refl)
   apply (simp add: SANITIZE-NAMES-def)
  apply (rule rel-spec-monad-rel-xval-L2-gets)
 apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
  apply (rule rel-xval.Exn)
  apply simp
  apply (rule rel-sum-Inr)
  apply (rule refl)
 apply (simp add: SANITIZE-NAMES-def)
 done


schematic-goal rel-spec-monad (=) (=)
(L2-seq (L2-unknown [S ''ret--int''])
 (λr. L2-catch
       (L2-seq
         (L2-catch
           (L2-seq
             (L2-while (λr s. True)
           (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S ''global-exn-var'',
S ''ret'']))
                     (L2-seq
                       (L2-condition (λs. 1 < a)
                         (L2-throw (Nonlocal 1, r) [S ''global-exn-var'', S ''ret''])
                         (L2-throw (Break, r) [S ''global-exn-var'', S ''ret'']))
                       (λra. L2-gets (λs. r) [S ''ret'']))))
```

```
        r [S ''ret'']) (λr. L2-gets (λs. ()) [S ''ret'']))
      (λ(a, b).
         L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S ''ret''])
          (L2-throw (a, b) [S ''global-exn-var'', S ''ret'']))) (λr. L2-throw (Return, 2) [S ''global-exn-var'', S ''ret'']))
    (λ(a, b).
       L2-condition (λs. is-local a) (L2-gets (λs. b) [S ''ret''])
        (L2-throw a [S ''global-exn-var''])))))

?A

 supply c-exntype.case-distrib [where h=from-xval, simp] prod.case-distrib [where h=from-xval, simp]
       from-xval-simps [simp] c-exntype.case-cong [cong]
 apply (simp add: cond-return2 if-c-exntype-cases)
 apply (rule rel-spec-monad-rel-xvalI)
 apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
  apply (rule rel-spec-monad-rel-xval-L2-unknown)
 apply (rule rel-spec-monad-rel-xval-try-catch [split-tuple f arity: 2])
 apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
  apply (rule rel-spec-monad-rel-xval-try-catch [split-tuple f arity: 2])
  apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
  apply (rule rel-spec-monad-L2-while-rel-xval-same-split)
   apply (rule rel-spec-monad-eq-rel-xval-L2-condition)
    apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
     apply (rule rel-xval.Exn)
     apply simp
     apply (rule rel-sum-Inl)
     apply simp
     apply (rule rel-sum-Inr)
     apply (rule refl)
    apply (simp add: SANITIZE-NAMES-def)
   apply (rule rel-spec-monad-L2-seq-rel-xval-same-split)
    apply (rule rel-spec-monad-eq-rel-xval-L2-condition)
    apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
     apply (rule rel-xval.Exn)
     apply simp
     apply (rule rel-sum-Inl)
     apply simp
     apply (rule rel-sum-Inl)
     apply (rule refl)
     apply (simp add: SANITIZE-NAMES-def)
   apply (rule rel-spec-monad-L2-throw-sanitize-names [where ns' = []])
     apply (rule rel-xval.Exn)
     apply simp
     apply (rule rel-sum-Inr)
     apply (rule refl)
    apply (simp add: SANITIZE-NAMES-def)
```

apply (*rule rel-spec-monad-rel-xval-L2-gets*)
apply (*rule rel-spec-monad-rel-xval-L2-gets*)
apply (*rule rel-spec-monad-L2-throw-sanitize-names* [**where** *ns'* = []])
apply (*rule rel-xval.Exn*)
apply *simp*
apply (*rule rel-sum-Inr*)
apply (*rule refl*)
apply (*simp add*: *SANITIZE-NAMES-def*)
done

26.10.2 From *L2-catch* and flat exceptions to *L2-try* and nested exceptions

schematic-goal *rel-spec-monad* (=) (=)
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--int*'])
(λ*r*. *L2-catch*
(*L2-seq*
(*L2-catch*
(*L2-seq*
(*L2-while* (λ*r s*. *True*)
(λ*r*. *L2-condition* (λ*s*. 3 < *a*) (*L2-throw* (*Return*, *1*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'])
(*L2-seq*
(*L2-condition* (λ*s*. *1* < *a*)
(*L2-throw* (*Nonlocal 1*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'])
(*L2-throw* (*Break*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*']))
(λ*ra*. *L2-gets* (λ*s*. *r*) [$\mathcal{S}$ ''*ret*'])))
r [$\mathcal{S}$ ''*ret*'])
(λ*r*. *L2-gets* (λ*s*. ()) [$\mathcal{S}$ ''*ret*']))
(λ(*a*, *b*).
L2-condition (λ*s*. *a* = *Break*) (*L2-gets* (λ-. ()) [$\mathcal{S}$ ''*ret*'])
(*L2-throw* (*a*, *b*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'])))
(λ*r*. *L2-throw* (*Return*, *2*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*']))
(λ(*a*, *b*).
L2-condition (λ*s*. *is-local a*) (*L2-gets* (λ*s*. *b*) [$\mathcal{S}$ ''*ret*'])
(*L2-throw a* [$\mathcal{S}$ ''*global-exn-var*'']))))

?A

apply (*simp add*: *cond-return2 if-c-exntype-cases cong*: *c-exntype.case-cong*)
apply (*rule rel-spec-monad-rel-xvalI*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)

apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
done

schematic-goal *rel-spec-monad* (=) (=)
(*L2-seq* (*L2-unknown* [*𝒮* ''*ret--int*''])
(*λr. L2-catch*
(*L2-seq*
(*L2-catch*
(*L2-seq*
(*L2-while* (*λr s. True*)
(*λr. L2-condition* (*λs. 3* < *a*) (*L2-throw* (*Return*, *1*) [*𝒮* ''*global-exn-var*'', *𝒮* ''*ret*''])
(*L2-seq*
(*L2-condition* (*λs. 1* < *a*)
(*L2-throw* (*Nonlocal 1*, *r*) [*𝒮* ''*global-exn-var*'', *𝒮* ''*ret*''])
(*L2-throw* (*Break*, *r*) [*𝒮* ''*global-exn-var*'', *𝒮* ''*ret*'']))
(*λra. L2-gets* (*λs. r*) [*𝒮* ''*ret*''])))
r [*𝒮* ''*ret*''])
(*λr. L2-gets* (*λs.* ()) [*𝒮* ''*ret*'']))
(*λ*(*a*, *b*).
L2-condition (*λs. a* = *Break*) (*L2-gets* (*λ-.* ()) [*𝒮* ''*ret*''])
(*L2-throw* (*a*, *b*) [*𝒮* ''*global-exn-var*'', *𝒮* ''*ret*''])))
(*λr. L2-throw* (*Return*, *2*) [*𝒮* ''*global-exn-var*'', *𝒮* ''*ret*'']))
(*λ*(*a*, *b*).
L2-condition (*λs. is-local a*) (*L2-gets* (*λs. b*) [*𝒮* ''*ret*''])
(*L2-throw a* [*𝒮* ''*global-exn-var*'']))))

?A

apply (*simp add*: *cond-return2 if-c-exntype-cases cong*: *c-exntype.case-cong*)
apply (*rule rel-spec-monad-rel-xvalI*)
apply (*rel-spec-monad-L2-rewrite*)
done

schematic-goal *rel-spec-monad* (=) (=)
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--int*'])
(λ*ret--int*.
L2-catch
(*L2-seq*
(*L2-condition* (λ*s*. *p* = *0x20* ∨ *2* <*s p*)
(*L2-condition* (λ*s*. *p* ≠ *0x1E* ∧ *p* ≠ *2*) (*L2-throw* (*Return*, *p*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'])
(*L2-gets* (λ-. *ret--int*) [$\mathcal{S}$ ''*ret*']))
(*L2-gets* (λ-. *ret--int*) [$\mathcal{S}$ ''*ret*']))
(λ*ret--int*.
L2-seq
(*L2-call* (*l2-add'* *undefined p p*) (λ*global-exn-var*. (*Nonlocal* (*the-Nonlocal global-exn-var*), *ret--int*))
[$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'])
(λ*p---int*. *L2-throw* (*Return*, *p*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'])))
(λ(*global-exn-var*, *ret--int*).
L2-condition (λ*s*. *is-local global-exn-var*) (*L2-gets* (λ-. *ret--int*) [$\mathcal{S}$ ''*ret*'])
(*L2-throw global-exn-var* [$\mathcal{S}$ ''*global-exn-var*'']))))
?A

apply (*simp add*: *cond-return2 if-c-exntype-cases cong*: *c-exntype.case-cong*)
apply (*rule rel-spec-monad-rel-xvalI*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)

apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
done

ML ‹
val thm = *L2-Exception-Rewrite.abstract-try-catch* @{*context*} @{*term* (*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--int*'']) (λ*r*. *L2-catch* (*L2-seq* (*L2-catch* (*L2-seq* (*L2-while* (λ*r s*. *True*)) (λ*r*. *L2-condition* (λ*s*. 3 < *a*) (*L2-throw* (*Return*, 1) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']) (*L2-seq* (*L2-condition* (λ*s*. 1 < *a*) (*L2-throw* (*Nonlocal* 1, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']) (*L2-throw* (*Break*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*''])) (λ*ra*. *L2-gets* (λ*s*. *r*) [$\mathcal{S}$ ''*ret*'']))) *r* [$\mathcal{S}$ ''*ret*'']) (λ*r*. *L2-gets* (λ*s*. ()) [$\mathcal{S}$ ''*ret*''])) (λ(*a*, *b*). *L2-condition* (λ*s*. *a* = *Break*) (*L2-gets* (λ-. ()) [$\mathcal{S}$ ''*ret*'']) (*L2-throw* (*a*, *b*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*''])) (λ*r*. *L2-throw* (*Return*, 2) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))) (λ(*a*, *b*). *L2-condition* (λ*s*. *is-local a*) (*L2-gets* (λ*s*. *b*) [$\mathcal{S}$ ''*ret*'']) (*L2-throw a* [$\mathcal{S}$ ''*global-exn-var*'']))))}
›

schematic-goal *rel-spec-monad* (=) (*rel-xval* (=) (=))
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--int*'']) (λ*r*. *L2-catch* (*L2-seq* (*L2-catch* (*L2-seq* (*L2-while* (λ*r s*. *True*)) (λ*r*. *L2-condition* (λ*s*. 3 < *a*) (*L2-throw* (*Return*, 1) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']) (*L2-condition* (λ*s*. 1 < *a*) (*L2-gets* (λ*s*. *r*) [$\mathcal{S}$ ''*ret*'']) (*L2-throw* (*Break*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*''])) *r* [$\mathcal{S}$ ''*ret*'']) (λ*r*. *L2-gets* (λ*s*. ()) [$\mathcal{S}$ ''*ret*''])) (λ(*a*, *b*). *L2-condition* (λ*s*. *a* = *Break*) (*L2-gets* (λ-. ()) [$\mathcal{S}$ ''*ret*'']) (*L2-throw* (*a*, *b*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*''])) (λ*r*. *L2-throw* (*Return*, 2) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))) (λ(*a*, *b*). *L2-condition* (λ*s*. *is-local a*) (*L2-gets* (λ*s*. *b*) [$\mathcal{S}$ ''*ret*'']) (*L2-throw*

a [$\mathcal{S}$ ''*global-exn-var*'']))))
?A
apply (*simp add*: *cond-return2 if-c-exntype-cases cong*: *c-exntype.case-cong*)
apply (*rel-spec-monad-L2-rewrite*)
done

schematic-goal
rel-spec-monad (=) (*rel-xval* (=) (=))
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--int*''])
(λ*r*. *L2-catch*
(*L2-seq*
(*L2-catch*
(*L2-seq*
(*L2-while* (λ(*a*, *b*) *a*. *True*)
(λ(*a*, *b*).
L2-seq
(*L2-condition* (λ*s*. *3* < *a*)
(*L2-throw* (*a*, *Return*, *1*) [$\mathcal{S}$ ''*a*'', $\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']) (*L2-seq*
(*L2-condition* (λ*s*. *1* < *a*) (*L2-gets* (λ*s*. *a* + *1*) [$\mathcal{S}$ ''*a*'']) (*L2-throw* (*a*, *Break*, *b*) [$\mathcal{S}$ ''*a*'', $\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))
(λ*r*. *L2-gets* (λ*s*. (*r*, *b*)) [$\mathcal{S}$ ''*a*'', $\mathcal{S}$ ''*ret*'']))) (λ(*a*, *b*). *L2-gets* (λ*s*. (*a*, *b*)) [$\mathcal{S}$ ''*a*'', $\mathcal{S}$ ''*ret*'']))
(*a*, *r*) [$\mathcal{S}$ ''*a*'', $\mathcal{S}$ ''*ret*'']) (λ(*a*, *b*). *L2-gets* (λ*s*. *a*) [$\mathcal{S}$ ''*a*'']))
(λ(*aa*, *a*, *b*).
L2-condition (λ*s*. *a* = *Break*) (*L2-gets* (λ*s*. *aa*) [$\mathcal{S}$ ''*a*'']) (*L2-throw* (*a*, *b*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))) (λ*r*. *L2-throw* (*Return*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))
(λ(*a*, *b*).
L2-condition (λ*s*. *is-local a*) (*L2-gets* (λ*s*. *b*) [$\mathcal{S}$ ''*ret*'']) (*L2-throw a* [$\mathcal{S}$ ''*global-exn-var*'']))))
?A
apply (*simp add*: *cond-return2 if-c-exntype-cases cong*: *c-exntype.case-cong*)
apply (*rel-spec-monad-L2-rewrite*)
done

declare [[*show-main-goal*=*true*]]

26.10.3 Flatten the error type of calls, aka get rid of constructor *Nonlocal*

schematic-goal *rel-spec-monad* (=) (*rel-xval rel-Nonlocal* (=))
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--int*''])
(λ*r*. *L2-catch*
(*L2-seq*
(*L2-catch*
(*L2-seq*

```
(L2-while (λr s. True)
 (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S ''global-exn-var'',
S ''ret'']))
          (L2-seq
            (L2-condition (λs. 1 < a)
              (L2-throw (Nonlocal 1, r) [S ''global-exn-var'', S ''ret''])
              (L2-throw (Break, r) [S ''global-exn-var'', S ''ret'']))
            (λra. L2-gets (λs. r) [S ''ret'']))))
    r [S ''ret''])
  (λr. L2-gets (λs. ()) [S ''ret'']))
 (λ(a, b).
    L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S ''ret''])
     (L2-throw (a, b) [S ''global-exn-var'', S ''ret'']))))
(λr. L2-throw (Return, 2) [S ''global-exn-var'', S ''ret'']))
(λ(a, b).
   L2-condition (λs. is-local a) (L2-gets (λs. b) [S ''ret''])
    (L2-throw a [S ''global-exn-var''])))))

?A
```

apply (*simp add*: *cond-return2 if-c-exntype-cases cong*: *c-exntype.case-cong*)
apply (*rel-spec-monad-L2-rewrite*)
done

Some statistics on the caches involved.

ML ‹
```
val statistics = map (apsnd  Fun-Cache.get-info) (Fun-Cache.all-handlers ())
```
›

26.11 Tuple optimization by analysing variable use and removing unused variables.

The cases for *L2-seq* are the points were the uses-analysis is invoked: `Rel_Spec_Monad_Synthesize_Ru`
The analysis follows the usage of the bound variable within the term.

schematic-goal
```
 rel-spec-monad (=) (rel-xval (=) (rel-project (λv. ())))
(L2-seq (L2-gets (λ-. x) [S ''x''])
     (λx. L2-seq
       (L2-gets (λ-. (x, y)) [S ''x'',S ''y''])
       (λ(x, y). L2-gets (λ-. y) [S ''y''])))
?X
```

apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)

apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
done

schematic-goal
rel-spec-monad (=) (*rel-xval* (=) (*rel-project* (λ*v*. *v*)))
(*L2-seq* (*L2-gets* (λ-. *x*) [$\mathcal{S}$ ''*x*''])
(λ*x*. *L2-seq*
(*L2-gets* (λ-. (*x*, *y*)) [$\mathcal{S}$ ''*x*'',$\mathcal{S}$ ''*y*''])
(λ(*x*, *y*). *L2-gets* (λ-. *y*) [$\mathcal{S}$ ''*y*'']))))
?X

apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
done

schematic-goal
rel-spec-monad (=) (*rel-xval* (=) (*rel-project* (λ*v*. *v*)))
(*L2-seq* (*L2-gets* (λ-. (*l*,*m*)) [$\mathcal{S}$ ''*l*'', $\mathcal{S}$ ''*m*''])
(λ(*x*, *y*). *L2-seq*
(*L2-gets* (λ-. (*x*, *k*)) [$\mathcal{S}$ ''*x*'', $\mathcal{S}$ ''*y*''])

(λ(*a*, *b*).
L2-seq
(*L2-while* (λ(*x*,*y*) -. *x* < *2*)
(λ(*x*,*y*). (*L2-gets* (λ-. (*y*, *k*)) [$\mathcal{S}$ ''*x*'', $\mathcal{S}$ ''*y*'']))
(*a*, *b*) [$\mathcal{S}$ ''*x*'', $\mathcal{S}$ ''*y*''])
(λ(*x*, *y*). *L2-gets* (λ-. *y*) [$\mathcal{S}$ ''*y*'']))))
?X
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)

apply (*rel-spec-monad-L2-step*)
done

declare [[*show-main-goal*=*true*]]
schematic-goal
rel-spec-monad (=) (*rel-xval*(=) (*rel-project* (λ*v*. *v*)))
(*L2-seq* (*L2-gets* (λ-. (*l*,*m*)) [$\mathcal{S}$ ''*l*'', $\mathcal{S}$ ''*m*''])
(λ(*x*, *y*). *L2-seq*
(*L2-gets* (λ-. (*x*, *k*)) [$\mathcal{S}$ ''*x*'', $\mathcal{S}$ ''*y*''])

(λ(*a*, *b*).
L2-seq
(*L2-while* (λ(*x*,*y*) -. *y* < *2*)
(λ(*x*,*y*). (*L2-gets* (λ-. (*y*, *k*)) [$\mathcal{S}$ ''*x*'', $\mathcal{S}$ ''*y*'']))
(*a*, *b*) [$\mathcal{S}$ ''*x*'', $\mathcal{S}$ ''*y*''])
(λ(*x*, *y*). *L2-gets* (λ-. *y*) [$\mathcal{S}$ ''*y*'']))))
?X
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
done

schematic-goal
rel-spec-monad (=) (*rel-xval* (=) (*rel-project* (λ*v*. *v*)))
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--unsigned*''])
(λ*x*. *L2-try*
(*L2-seq*
(*L2-while* (λ(*n---int*, *ret--unsigned*) *s*. *True*)
(λ(*x1*, *x2*).
L2-seq (*L2-guard* (λ*s*. *0* ≤ *2147483649* + *sint x1* ∧ *sint x1* ≤ *2147483646*))
(λ*xa*. *L2-seq* (*L2-gets* (λ-. *x1* + *1*) [$\mathcal{S}$ ''*n*''])
(λ*xb*. *L2-seq*
(*L2-condition* (λ*s*. *xb* <*s* *2*) (*L2-throw* (*Inr 1*) [$\mathcal{S}$ ''*ret*''])
(*L2-throw* (*Inr 2*) [$\mathcal{S}$ ''*ret*'']))

(λxc. L2-gets (λ-. (xb, xc)) [S ''n'', S ''ret''])))))
(n, x) [S ''n'', S ''ret''])
(λ(x1, x2). L2-fail))))
?X

apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
(L2-seq (L2-call (l2-plus' undefined 1 2) (λe. e) [])
(λx. L2-seq (L2-call (l2-plus' undefined 2 3) (λe. e) [])
(λxa. L2-gets (λs. if xa = 0 then 1 else 0) [S ''ret''])))
?XX
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
apply (rel-spec-monad-L2-step)
done

schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
(L2-seq (L2-unknown [S ''ret--int''])

(λx. *L2-seq* (*L2-gets* (λ-. *2*) [$\mathcal{S}$ ''x'']) (λxa. *L2-try* (*L2-seq* (*L2-try* (*L2-seq* (*L2-while* (λ(*a---int*, *ret--int*, *y---int*) *s*. *True*) (λ(*x1*, *x2*, *x3*). *L2-seq* (*L2-gets* (λ-. *4*) [$\mathcal{S}$ ''y'']) (λxb. *L2-seq* (*L2-condition* (λs. *3* <*s* *x1*) (*L2-throw* (*Inl* (*Inr* *1*)) [$\mathcal{S}$ ''ret'', $\mathcal{S}$ ''y'']) (*L2-seq* (*L2-condition* (λs. *1* <*s* *x1*) (*L2-seq* (*L2-guard* (λs. *0* $\leq$ *2147483649* + *sint* *x1* $\wedge$ *sint* *x1* $\leq$ *2147483646*)) (λxc. *L2-gets* (λs. *x1* + *1*) [$\mathcal{S}$ ''a''])) (*L2-try* (*L2-seq* (*L2-try* (*L2-while* (λ- *s*. *True*) (λxc. *L2-throw* (*Inr* *xc*) []) () [])) (λxc. *L2-throw* (*Inl* (*Inr* *xb*)) [])))) (λxc. *L2-gets* (λ-. (*xc*, *x2*)) [$\mathcal{S}$ ''a'', $\mathcal{S}$ ''ret'']))) (λ(*x1*, *x2*). *L2-gets* (λ-. (*x1*, *x2*, *xb*)) [$\mathcal{S}$ ''a'', $\mathcal{S}$ ''ret'', $\mathcal{S}$ ''y'']))) (*a*, *x*, *3*) [$\mathcal{S}$ ''a'', $\mathcal{S}$ ''ret'', $\mathcal{S}$ ''y'']) (λ(*x1*, *x2*, *x3*). *L2-gets* (λ-. *x3*) [$\mathcal{S}$ ''y''])) (λxb. *L2-seq* (*L2-guard* (λs. − *2147483648* $\leq$ *sint* *xa* + *sint* *xb* $\wedge$ *sint* *xa* + *sint* *xb* $\leq$ *2147483647*)) (λxc. *L2-throw* (*Inr* (*xa* + *xb*)) [$\mathcal{S}$ ''ret'']))))))
?X

apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)

apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
done

schematic-goal *rel-spec-monad* (=) (*rel-xval* (=) (*rel-project* (λ*v*. *v*)))
(*L2-condition* (λ*s*. *n* = *0*) (*L2-gets* (λ*s*. *0*) [$\mathcal{S}$ ''*ret*''])
(*L2-seq* (*L2-call* (*l2-even'* (*recguard-dec rec-measure'*) (*n* − *1*)) (λ*e*. *e*) [])
(λ*x*. *L2-gets* (λ*s*. *SCAST*(*32 signed* → *32*) (*if x* = *0 then 1 else 0*))
[$\mathcal{S}$ ''*ret*''])))
?X
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)

apply (*rel-spec-monad-L2-step*)
done

schematic-goal *rel-spec-monad* (=) (*rel-xval* (=) (*rel-project* (λ*v*. *v*)))
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--unsigned*'])
(λ*x*. *L2-condition* (λ*s*. *n* = *0*) (*L2-gets* (λ*s*. *0*) [$\mathcal{S}$ ''*ret*'])
(*L2-try*
(*L2-seq*
(*L2-call*
(*l2-fac-exit'* (*recguard-dec rec-measure'*) (*n* − *1*))
(λ*e*. *Inl* (*Nonlocal* (*the-Nonlocal e*))) [$\mathcal{S}$ ''*ret*'])
(λ*xa*. *L2-throw* (*Inr xa*) [$\mathcal{S}$ ''*ret*'])))))
?X
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
done

schematic-goal *rel-spec-monad* (=) (*rel-xval* (=) (*rel-project* (λ*v*. *v*)))
(*L2-try*
(*L2-call* (*l2-just-exit'* *undefined*)
(λ*e*. *Inl* (*Nonlocal* (*the-Nonlocal e*))) [])
)
?X
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
apply (*rel-spec-monad-L2-step*)
done

declare [[*show-main-goal*=*false*]]
declare [[*verbose*=*0*]]
declare [[*verbose-timing*=*0*]]

26.12 Locales, Local-Theories, Named-Targets, Morphisms and Declarations...

We explore some of the concepts of Locales, Local-Theories, Named-Targets, Declarations etc. Besided the corresponding chapters in the Isabelle/Isar Reference Manual as well as the Isabelle/Isar Implementation Manual, the following references give a solid theoretical background:

- Local Theories: http://isabelle.in.tum.de/~haftmann/pdf/local_theory_specifications_haftmann_wenzel.pdf
- Locales: http://www21.in.tum.de/~ballarin/publications/jar2013.pdf
- Morphisms and Declarations: https://www21.in.tum.de/~wenzelm/papers/context-methods.pdf

  Roughly speaking, locales provide the infrastructure to manage the dependency graph of locale declarations. When entering a locale this dependency graph is traversed and results in a blue-print to build a local context by expanding and evaluating the locale expressions in a canonical order. The result is presented to the user as a local theory. A local theory consists of an axiomatic part (think of fixes and assumes) and a derived part, consisting of definitions and theorems within the axiomatic part.

locale *Foo* =
  fixes *N*::*nat*
  fixes *M*::*nat*
  assumes *N-le-M*: $N < M$

Locale *Foo* is blue-print to build a local context. We can enter this context with **context**.

context *Foo*
begin

Within the context we can explore it. As an example the locale assumption becomes a theorem within the context

thm *N-le-M*

Within the context the axiomatic part (fixes and assumes) are often accessed implicitly. For example the fixes are presented as free Variables with the declared type fixed.

ML ‹@{*assert*} (@{*term N*} = *Free* (*N*, @{*typ nat*}))›

The locale assumptions are wrapped up in the locale predicate *Foo*, and internally become part of the hypothesis of a (local theory) theorem. Theorems derived within the local theory will all carry around this implicit hypothesis.

declare [[*show-hyps*]]
thm *N-le-M*
ML ‹@{*thm N-le-M*}›
thm *Foo-def*

There is also an exported version of the theorem that holds on the theory level. Here the implicit hypothesis becomes an explicit precondition.

thm *Foo.N-le-M*

A definition in a local theory has multiple effects. A generalised definition, where the locale fixes become explicit parameters is issued in the background theory. Moreover, an abbreviation that hides these parameters in introduced.

definition *G-def*: $G = M + N$

lemma $G = Foo.G\ N\ M$**..**

ML ‹@{*assert*} (@{*term G*} = @{*term Foo.G N M*})›

When working within ML we also deal with another view, the *auxiliary context* as referred to in the Isabelle documentation. On the Isar top-level we are usually only exposed to the *background theory*, e.g. *Foo.G* and the *target context*, e.g. *G*.

local-setup ‹ *fn lthy0 =>*
let

val ((*trm*, (*def-name*, *def-thm*)), *lthy1*) = *lthy0* |>
Specification.definition (*SOME* (*Binding.name F*, *NONE*, *NoSyn*))
[] [] (*Binding.empty-atts*, @{*prop F = G + M*})
— lthy1 is now the auxiliary context. Here *F* aka. *trm* is not an abbreviation but acutally a fixed variable in that auxiliary context.
val - = @{*print*} *trm*
val - = @{*assert*} (*trm = Free* (*F*, @{*typ nat*}))
— Theorems within this context will have an additional hypothesis reflecting the definition equation of the *F*, *F* ≡ *Foo.F N M*. This can be explored already in *def-thm*.

val - = *writeln* (*Thm.string-of-thm lthy1 def-thm*)

— One can switch between the different views / versions of a theorem by *exporting* it, either by means of an explicit export-morphism or by an export function. Out of the auxiliary context into the target context (view of the locale):
val [*target-def-thm*] = *Proof-Context.export lthy1 lthy0* [*def-thm*]
val - = *writeln* (*target-def-thm*: ^ *Thm.string-of-thm lthy1 target-def-thm*)
val phi-target = *Proof-Context.export-morphism lthy1 lthy0*
val target-def-thm′ = *Morphism.thm phi-target def-thm*
val - = *writeln* (*target-def-thm′*: ^ *Thm.string-of-thm lthy1 target-def-thm′*)
— Out on the theory level.

```
  val thy-lthy = Proof-Context.init-global (Proof-Context.theory-of lthy1)
  val [theory-def-thm] = Proof-Context.export lthy1 thy-lthy [def-thm]
  val - = writeln (theory-def-thm:  ^ Thm.string-of-thm lthy1 theory-def-thm)
  val phi-thy = Proof-Context.export-morphism lthy1 thy-lthy
  val theory-def-thm' = Morphism.thm phi-thy def-thm
  val - = writeln (theory-def-thm':  ^ Thm.string-of-thm lthy1 theory-def-thm')
 in
  lthy1
 end
›
```

Here we have the target view of the freshly defined F.

ML ‹@{*assert*} (@{*term F*} = @{*term Foo.F N M*})›

One notable logical difference of the view on a definition within a auxiliary context and the target context is polymorphism. Within the auxiliary context the definition is always monomorphic, maybe referring to implicitly fixed types 'variables', or better frees. Within the target context it might be polymorphic (depending on the definition), as it is just an abbreviation for the global constant applied to the locale parameters.

The fixed view of the auxiliary context can be quite handy when continuing work with the freshly defined function, do some proofs etc. in ML.

Working with nested contexts, like fixing and assuming new stuff opening and closing them and exporting results is supported within ML by `Local_Theory.begin_nested` and `Local_Theory.end_nested`. The term might be a little bit misleading as you don't actually open an close a target but open and close a context block within the same target.

The following example illustrates this first with toplevel commands and then with ML.

```
context
  fixes K
  assumes M-le-K: M < K
begin
lemma N-le-K: N < K
  by (rule less-trans [OF N-le-M M-le-K])
```

Here we have the local view on the theorem within the nested context.

```
thm N-le-K
end
```

Now we have the exported view of the theorem within the target context.

```
thm N-le-K
```

```
local-setup ‹fn lthy0 =>
let
  val (-, lthy) = lthy0 |> Local-Theory.begin-nested
  val ([K], lthy) = Variable.add-fixes [K] lthy
```

val - = writeln (is-fixed K: ^ @{make-string} (Variable.is-fixed lthy K))
val ([M-le-K], lthy-inner) = Assumption.add-assumes [(Thm.cterm-of lthy (Syntax.read-prop lthy M < K))] lthy
val thm = @{thm less-trans} OF [@{thm N-le-M}, M-le-K]
val ([(th-name,[thm])], lthy) = Local-Theory.notes [((Binding.name N-le-K',[]), [([thm], [])])] lthy-inner
val - = writeln (thm: ^ Thm.string-of-thm lthy thm)
val lthy = lthy |> Local-Theory.end-nested
val - = writeln (is-fixed K: ^ @{make-string} (Variable.is-fixed lthy K))
val [target-thm] = Proof-Context.export lthy-inner lthy [thm]
val - = writeln (target-thm: ^ Thm.string-of-thm lthy target-thm)
in
lthy
end

›

Here we have the exported view of the theorem within the target context.

thm *N-le-K'*

end

After a definition in a local theory the context is extended with a fixed variable for the lhs and an equation that relates it to the global definition. So everything what happens thereafter happens within that extended context.

ML-val ‹
val lthy = Named-Target.theory-init @{theory}
*val ((lhs,(name, thm)), lthy) = Local-Theory.define ((**binding**‹foo›, Mixfix.NoSyn), ((Binding.empty, []), @{term 42::nat})) lthy*
val prems = Assumption.all-prems-of lthy
›

By surrounding the definition with a `Local_Theory.begin_nested` and `Local_Theory.end_nested` you can step out of the extended context again. Don't forget to export the results as well.

ML-val ‹
val lthy = Named-Target.theory-init @{theory} |> Local-Theory.begin-nested |> snd
*val ((lhs,(name, thm)), lthy) = Local-Theory.define ((**binding**‹foo›, Mixfix.NoSyn), ((Binding.empty, []), @{term 42::nat})) lthy*
val lthy' = Local-Theory.end-nested lthy
val prems = Assumption.all-prems-of lthy'
val [thm'] = Proof-Context.export lthy lthy' [thm]
›

26.13 Morphisms and Declarations

As we have seen before, when working with locales and local theories, we deal with different views on the same results (like theorems). The locale

infrastructure takes care to provide the results in the 'expected' way by a careful naming policy for facts and (local) constants.

When maintaining results in custom (generic) context data, Isabelle provides mechanisms around the notion of a *declaration*, which comes in various flavours. In particular attached to (local) theorems / facts in the form of attributes (rule and declaration attributes) or as local theory declarations. In the case of attributes the morphism is implicit, as the theorem / facts they are attached to are already in the expected localized version. So the function can refer to these theorem / facts. In case of local theory declarations the morphism is explicitly passed as an argument to the function.

In ML, attributes are functions from `thm * Proof.context -> thm * Proof.context`, in practice presented as either a declaration attribute (modifying the context) `thm -> Proof.context -> Proof.context` or as a rule attribute (modifying the theorem) `Proof.context -> thm -> thm`. Local theory declarations are functions, taking a morphism and modifying the `local_theory`: `Morphism.morphism -> local_theory -> local_theory`.

All these declarations are part of the context-building-infrastructure of locales and local theories. In that sense a local theory declaration can be thought of as declaration attribute attached to a dummy fact. Whenever a context is entered and the fact is processed the declarations are reevaluated on the that context. As we have seen, at each point there are three contexts simultaneously, representing the background theory, the target context and the auxiliary context. So at each point there are also three views on the data, 'viewed' via the application of the explicit or implicit morphisms.

Note that when implementing context data and tools, morphisms come in two variants. Import and export morphisms. Exporting results or data out of a context into a more general context can often be treated explicit by `Proof_Context.export` or `Proof_Context.export_morphism` or variants of those functions. This is because going out of a context is unambigous and therefor a single morphism is enough.

Whereas when entering into a context (like entering a locale) the morphisms are internally generated and provided by the locale infrastructure. As a locale might be imported several times in different variants into the same local theory there is no unambigous morphism. Each import provides a different morphism.

When issuing a local theory declaration via `Local_Theory.declaration` there is also two flags to define, represented as a record `{syntax: bool, pervasive: bool}`.

- A pervasive declaration is also applied to the background-theory (besides the target and auxiliary context)
- A syntax declaration is activated already in the syntax phase of the locale roundup. This means, when your context data already has to be

ready to enabling parsing of new assumptions declaring a new locale you have to set the syntax flag.

Note that theorem attributes are treated like `{pervasive = false, syntax = false}`. That means that they are not evaluated on the background-theory version of the facts. This makes perfect sense for typical attributes like *simp*. The theorem is only added to the simpset within the locale. The background-theorem is not added to the simpset of the theory. Keep in mind that the background-theorem is typically an implication with the locale predicate as precondition. However, when you interprete the locale on the theory level the interpreted version of the theorem is added to the simpset of the theory, here the locale-predicate precondition is discharged.

In the following we have some examples illustrating the described behaviour.

ML ‹

```
structure MyData = Generic-Data (
  type T = int;
  val empty = 0;
  val merge = fst;
)
```
›

ML ‹
val n-theory = MyData.get (Context.Theory @{theory})
val - = @{assert} (n-theory = 0)
val n-aux = MyData.get (Context.Proof @{context})
val - = @{assert} (n-aux = 0)
›

setup ‹
Context.theory-map (MyData.put 1)
›
ML ‹
val n-theory = MyData.get (Context.Theory @{theory})
val - = @{assert} (n-theory = 1)
val n-aux = MyData.get (Context.Proof @{context})
val - = @{assert} (n-aux = 1)
›

context *Foo*
begin
ML ‹
val n-theory = MyData.get (Context.Theory @{theory})
val - = @{assert} (n-theory = 1)

```
val n-aux = MyData.get (Context.Proof @{context})
val - = @{assert} (n-aux = 1)
val n-target = MyData.get (Context.Proof (Local-Theory.target-of @{context}))
val - = @{assert} (n-target = 1)
›
```

Manipulating data by functions like `Context.proof_map` will only have very limited and especially temporary effect. It is applied to the auxiliary context only. When exiting the auxiliary context it will be away. Also it has no effect on the target context.

```
local-setup
‹fn lthy =>
 let
   val lthy = lthy |> Context.proof-map (MyData.put 2)
   val n-aux = MyData.get (Context.Proof lthy)
   val - = @{assert} (n-aux = 2)
   val n-target = MyData.get (Context.Proof (Local-Theory.target-of lthy))
   val - = @{assert} (n-target = 1)
   val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of lthy))
   val - = @{assert} (n-theory = 1)

 in
   lthy
 end
›
```

You might be surprised to see that the value is already reset here. The reason is that every toplevel command resets the context and is treated like a block. Think of a `Local_Theory.begin_nested` / `Local_Theory.end_nested`.

```
ML ‹
val n-aux = MyData.get (Context.Proof @{context})
val - = @{assert} (n-aux = 1)
›
```

A non pervasive declaration takes effect on the auxiliary and the target context but not the theory context.

```
local-setup ‹fn lthy =>
  let
     val lthy = lthy |> Local-Theory.declaration {pervasive=false, syntax=false,
pos=here} (fn phi =>  MyData.put 3)
   val n-aux = MyData.get (Context.Proof lthy)
   val - = @{assert} (n-aux = 3)
   val n-target = MyData.get (Context.Proof (Local-Theory.target-of lthy))
   val - = @{assert} (n-target = 3)
   val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of lthy))
   val - = @{assert} (n-theory = 1)
  in lthy end
›
```

As expected the effect is still there at this point.

ML ‹
val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of @{context}))
val - = @{assert} (n-theory = 1)
val n-aux = MyData.get (Context.Proof @{context})
val - = @{assert} (n-aux = 3)
val n-target = MyData.get (Context.Proof (Local-Theory.target-of @{context}))
val - = @{assert} (n-target = 3)
›
end

Note that the declaration is persistent in the sense that when reentering the locale it will be evaluated again. So when designing custom data and implementing declarations or attributes one has to be clear about the order of things and that they might be applied in many different situations. Using `MyData.map` instead of `MyData.put` might be handy to design robust data management.

context *Foo*
begin
ML ‹
val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of @{context}))
val - = @{assert} (n-theory = 1)
val n-aux = MyData.get (Context.Proof @{context})
val - = @{assert} (n-aux = 3)
val n-target = MyData.get (Context.Proof (Local-Theory.target-of @{context}))
val - = @{assert} (n-target = 3)
›

A pervasive declaration also effects the theory.

local-setup ‹*fn lthy =>*
let
*val lthy = lthy |> Local-Theory.declaration {pervasive=true, syntax=false, pos=**here**} (fn phi => MyData.put 4)*
val n-aux = MyData.get (Context.Proof lthy)
val - = @{assert} (n-aux = 4)
val n-target = MyData.get (Context.Proof (Local-Theory.target-of lthy))
val - = @{assert} (n-target = 4)
val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of lthy))
val - = @{assert} (n-theory = 4)
in lthy end
›

ML ‹
val n-theory = MyData.get (Context.Theory (Proof-Context.theory-of @{context}))
val - = @{assert} (n-theory = 4)
val n-aux = MyData.get (Context.Proof @{context})
val - = @{assert} (n-aux = 4)
val n-target = MyData.get (Context.Proof (Local-Theory.target-of @{context}))

val - = @{assert} (n-target = 4)
›

end

ML ‹
val n-theory = MyData.get (Context.Theory @{theory})
val - = @{assert} (n-theory = 4)
val n-aux = MyData.get (Context.Proof @{context})
val - = @{assert} (n-aux = 4)
›

26.13.1 Excurse on Proof Context vs. Local Theory.

A local theory is represented as a proof context. So a local theory is a semantic subtype of a proof context. Every local theory is a proof context but not every proof context is a local theory. In particular a local theory is a proof context that is ready to accept (local theory) definitions (`Local_Theory.define`) and notes (`Local_Theory.note`).

The Isabelle sources try to be consistent in the naming, i.e. 'ctxt' vs 'lthy' in parameters and variables and `Proof.context` vs. `local_theory` in signatures. But as `local_theory` is only a type synonym for `Proof.context`, there is no type check for this convention. The nesting level that you get with `Local_Theory.level` is an indicator. A Level of *0* is not a proper local theory.

With `Proof_Context.init_global` you get a proof context not a local theory.

ML-val ‹
val ctxt = Proof-Context.init-global @{theory}
val level = Local-Theory.level ctxt
val is-theory = Named-Target.is-theory ctxt
val locale = Named-Target.locale-of ctxt
val bottom-locale = Named-Target.bottom-locale-of ctxt
›

With `Named_Target.theory_init` you get a local theory.

ML-val ‹
val lthy = Named-Target.theory-init @{theory}
val level = Local-Theory.level lthy
val is-theory = Named-Target.is-theory lthy
val locale-of = Named-Target.locale-of lthy
val bottom-locale = Named-Target.bottom-locale-of lthy
›

With `Locale.init` you get a proof context not a local theory.

ML-val ‹
val ctxt = Locale.init (Locale.intern @{theory} Foo) @{theory}

```
val level = Local-Theory.level ctxt
val is-theory = Named-Target.is-theory ctxt
val locale-of = Named-Target.locale-of ctxt
val bottom-locale = Named-Target.bottom-locale-of ctxt
›
```

With `Named_Target.init` you get a local theory.

ML-val ‹
```
val lthy = Named-Target.init [] (Locale.intern @{theory} Foo) @{theory}
val level = Local-Theory.level lthy
val is-theory = Named-Target.is-theory lthy
val locale-of = Named-Target.locale-of lthy
val bottom-locale = Named-Target.bottom-locale-of lthy
›
```

26.13.2 Attributes vs. Local Theory Declarations

Attributes on local facts, are not applied on the underlying theory level foundation. Only when you interpret a locale on the theory level the attributes get activated. This corresponds to general `Local_Theory.declaration` where you have the flag *pervasive=false*.

To demonstrate this we define an artificial tracing attribute.

attribute-setup *trace-attr* =
```
‹
Attrib.thms >> (fn ths =>
  Thm.declaration-attribute
    (fn thm => fn context =>
      let
        val (kind, level, is-theory, locale, bottom-locale) = case context of
              Context.Proof ctxt =>
                let
                  val level = Local-Theory.level ctxt
                in
                 (if level = 0 then context else local-theory,
                  string-of-int level,
                  @{make-string} (Named-Target.is-theory ctxt),
                  @{make-string} (Named-Target.locale-of ctxt),
                  @{make-string} (Named-Target.bottom-locale-of ctxt))
                end
            | - => (theory, 0, true, NONE, NONE)

        val - = tracing (trace-attr ( ^ kind ^ , ^ level ^ , ^
          is-theory ^ , ^ locale ^ , ^ bottom-locale ^ ): ^
          Thm.string-of-thm (Context.proof-of context) thm)
      in context end))
›
```

First we try the attribute on a theory level theorem. When you put the cursor on or after the *simp* you see the tracing output of our attribute.

lemma *foo*[*trace-attr*]: $x = x$
by *simp*

Interestingly we see four tracing outputs. Here the outputs and my interpretation:

- *trace-attr* (*local-theory*, *1*, *true*, *NONE*, *NONE*): *?x* = *?x* This seems to be the immediate auxiliary context of the lemma.
- *trace-attr* (*theory*, *0*, *true*, *NONE*, *NONE*): *?x* = *?x* This is the theory level.
- *trace-attr* (*context*, *0*, *false*, *NONE*, *NONE*): *?x* = *?x* This seems to be a theory-context that might be generated with `Proof_Context.init_global`.
- *trace-attr* (*local-theory*, *1*, *true*, *NONE*, *NONE*): *?x* = *?x* This seems to be from reentering the original lemma context.

Now let us look at a nested context.

context *Foo*
begin
context
begin
lemma *silly*[*trace-attr*]: $M = M$ **by** *simp*
end
end

As you see there is no theory level trace:

- *trace-attr* (*local-theory*, *2*, *false*, *NONE*, *SOME Scratch.Foo*): $M = M$ [*Foo N M*] This is the original context of the lemma.
- *trace-attr* (*context*, *0*, *false*, *NONE*, *SOME Scratch.Foo*): $M = M$ [*Foo N M*] This seems to be the target context but not as a local theory.
- *trace-attr* (*local-theory*, *1*, *false*, *SOME Scratch.Foo*, *SOME Scratch.Foo*): $M = M$ [*Foo N M*] This seems to be the target context as a local theory.
- *trace-attr* (*local-theory*, *2*, *false*, *NONE*, *SOME Scratch.Foo*): $M = M$ [*Foo N M*] This seems to be from reentering the original lemma context.

Let us now try an interpretation of the locale on the theory level.

definition *NN*::*nat* **where** *NN* = *2*
definition *MM*::*nat* **where** *MM* = *3*

interpretation *Foo NN MM*
by (*unfold-locales*) (*simp add*: *NN-def MM-def*)

We can see the trace of the activation of the attribute: *trace-attr* (*theory*, *0*, *true*, *NONE*, *NONE*): *MM* = *MM*

26.14 ML Antiquotations

The notion of quote / antiquote is related to lisps quotation and quasi-quotation mechanisms or the notion of 'interpolation' that is used in the context of macros in other programming languages. In lisp, "everything is a list", in particular the program itself and you can dynamically evaluate a list. With quotation you can express that a list is meant to be plain data and should not be immediately evaluated. With quasi-quotation you can build data which inside uses a lisp-expression that evaluates / expands to some data that is inserted at that point.

This is a first intuition to understand ML antiquotations in Isabele/ML. An ML antiquotation is something embedded into the Isabelle/ML program text, that is not itself plain ML, but something that evaluates to a piece of ML text that is inserted in that position. A prominent example is the term antiquotation, which transforms a logical term (presented in the inner-syntax) of the logic, to the ML representation of that term. `term`

ML-val ‹
term ‹*x* + *y*›
›

The expansion takes place during compile time of the piece of Isabelle/ML. To be more precise it is expanded already during lexing of Isabelle/ML. This is important to keep in mind when writing your own antiquotations. The context you can refer to during the expansion of the antiquotation is the context you have during the lexing phase. This design decision facilitates robustness and predictability of the resulting ML code. Note that the expanded ML-code itself might of course be dynamically used later on in various contexts, but it is always the same code.

The main interface to write your own ML antiquotations is `ML_Antiquotation`. The more low-level (and more flexible) interface it is based on is `ML_Context.add_antiquotation`. The interface in `ML_Antiquotation` distinguishes the antiquotations into the variants *inline*, *value* and *special−form*.

The *inline* variant is the closest to the intuition of macro expansion. The inline-antiquotaton yields a piece of ML-text that is literally inserted at the position of the antiquotation. The term antiquotation is an example of this. `\<^term>\<open>x + y\<close>`. It is implemented with `ML_Antiquotation.inline_embedded`.

[1]

For the *value* variant the intuition is that the expanded antiquotation is immediatly evaluated in the compile time context and the resulting value is what is inserted at the position of the antiquotation. An example is the cterm antiquotation, e.g. `@{cterm "a = b"}`.

Following the static nature of antiquotations this abstract value is produced statically during compile time and is bound to some value as in *val x = ...compile−time−context...*, and then this value x is what appears in the expanded ML text. So to implement such an antiquotation means to provide two main ingredients: the code for the value binding (referred to as environment) and the code to reference that value (referred to as body). This is what the interface `ML_Antiquotation.declaration` offers. The argument `Proof.context -> string * string` is the central point. The pair of strings denotes the ML-text for the environment and the body respectively. See for example the theorem antiquotation: `@{thm refl}`. This can also be considered a *value* antiquotation, albeit being implemented by the more low-level interface.

Digging even deeper into `ML_Context.add_antiquotation` the `ML_Context.decl` also refers to a pair of ML-code, denoting an environment and the body, but here presented as `ML_Lex.token list` not as a string. This interface gives a lot of flexibility into the design of antiquotations. A notable example is the 'instantiate'-antiquotation. e.g:

ML-val ‹
fun foo t =
instantiate ‹*'a* = ***typ*** ‹*nat*› *and a* = ***term*** ‹*s*::*nat*› *and b = t in*
prop ‹*a + b = b + a*› *for a b*::‹*'a*::*plus*››
›

Note the nesting of antiquotations in that example. The 'environment-part' here consists of all the right hand sides of the instantiations as well as the proposition. The 'body-part' is an ML-expression that instantiates the proposition with the terms. The body part cannot be immediately evaluated during compile time as the value of parameter t of the function is not yet known. So the result is an ML-expression that references t among the right and sides of the instantiations that are evaluated at compile time.

The special-form antiquotation packs the text in a function thunk, e.g.

[1] Note that there is also a variant without the 'embedded'. suffix `ML_Antiquotation.inline`. What is the difference? According to a mail thread https://lists.cam.ac.uk/mailman/htdig/cl-isabelle-users/2021-October/msg00023.html the 'embedded' is a hint to Isabelle that antiquotation expects its argument as an 'embedded language' enclosed by a cartouche. This can nowadays be considered as the canonical way. Historically, Isabelle first distinguished between the outer-syntax (e.g. Isar or plain ML as Meta-Language) and the inner-syntax of the object logic like HOL or FOL. Meanwhile Isabelle embraces a lot of different languages that in some cases can even be nested, and the languages can be dynamically extended by declaring new commands or antiquotations.

`\<^try>\<open>I\<close>`.

26.15 Markup and Reports

Isabelle provides a rich set of markup to display information in Isabelle/-jEdit. Markup can either be directly attached to a string, or an existing source text can be decorated by sending *reports* to PIDE. An example for direct markup is the printing of terms, e.g.:

ML ‹
val str = Syntax.string-of-term @{context} @{term a + b}
›

You can make the markup visible by sending it to an output function, e.g.

ML ‹
tracing str
›

The markup is itself encoded in the string by the format `YXML`.

The markup can be expanded to the `XML.body` by `YXML.parse_body`:

ML-val ‹
YXML.parse-body str
›

The plain text (without markup) can be extracted like this:

ML-val ‹
XML.content-of o YXML.parse-body
›

It is also possible to produce pretty printed output without markup using `Pretty.pure_string_of` (see also `Pretty.string_of_ops` and `Pretty.pure_output_ops`).

To build up markup you can use `Markup`. The type for markup is a pair of element name and a list of properties. A property `Properties.T` is a key value pair which are both strings. First you build up your markup by using the various functions supplied in `Markup`, then you attach it to a string via `Markup.markup`.

ML-val ‹
val m = Markup.url http:\\www.isabelle.in.tum.de
val decorated = Markup.markup m attach url to this string
val xml = YXML.parse decorated
›

With `Position.report` you can add markup to a given position, e.g. here we add the warning-twigglies and an url:

ML-val ‹

*val pos = Position.range-position (Position.range (**here** , **here**))*
val w = Markup.warning
val u = Markup.url http:\\\\www.isabelle.in.tum.de
val - = Position.report pos w
val - = Position.report pos u
›

The standard message channels `error`, `warning` and `tracing` scan the string for position information and automatically markup the position.

ML-val ‹
*val this-spot = **here** (∗ wiggly line ∗)*
val msg = Look here: ^ Position.here this-spot
val - = warning msg
›

There are some fine points regarding the command region were the message is issued. By default markup is only shown in the actual command region.

ML ‹
*val this-spot-is-not-marked = **here** (∗ no wiggly line ∗)*
val msg1 = Look here: ^ Position.here this-spot-is-not-marked
›

We do not get the wiggly lines in the **ML** above, instead the **ML-val**

ML-val ‹
val - = warning msg1
›

We can put markup to a previous command if we first safe the command position of the first command, and temporary use this in the second command when issuing the message.

ML ‹
val command-thread-position = Position.thread-data ();
*val this-spot-is-marked = **here** (∗ wiggly line ∗)*
val msg2 = Look here: ^ Position.here this-spot-is-marked
›

ML-val ‹
val - = Position.setmp-thread-data command-thread-position warning msg2
›

To inspect the markup attached to the source you can use the panel Sidekick and select the language "isabelle-markup" When you hover over an entity in the upper half of the panel you see the attached markup in the lower half of the panel.

Reports are also used in autocorres `Feedback` to provide warning and error markup for C programs.

An illustrative example for custom markup and reports can be seen in the document antiquotation for rail diagrams: is `$ISABELLE_HOME/src/Pure/Tools/rail.ML`. It is used extensively in the Isabelle documentation, watch out for `\<^rail>\<open>\<close>` e.g. in `$ISABELLE_HOME/src/Doc/Datatypes/Datatypes.thy`

26.16 Term Synthesize via Intro Rules

A repeated task in the various AutoCorres phases is to synthesize a typically more abstract monadic function (output) from a given monadic function (input) together with a simulation theorem that connects both versions of the function. One strategy is to formulate introduction rules for the correspondence relation for the different language constructs and then recursively apply those rules guided by the constructs in the "input" function, and as a side effect of the rule application to synthesize the "output" function. Examples are, heap abstraction, word abstraction, exception rewriting (as described above) and type strengthening. Currently there is not (yet) an uniform implementation of this strategy, but the different instances are converging. Currently the most recent incarnation is the setup for type strengthening to perform the *refines* proofs:

- A term net is used for efficient lookup of potentially matching rules. It is indexed by the input function as well as the desired result relation.
- Splitting of tuples in rules is automatically performed to match the current term.
- Priorities of rules are specified to guarantee that the most specific rules are tried first
- A cache is used both for positive and negative proof attempts. As there might be multiple applicable rules when decomposing the input function, partial results might already be proven or have failed. Using a cache helps to prune repeated subproof attempts.

  The setup for this phase can be found in `../Refines_Spec.thy`. It employs **synthesize-rules**, as defined in `../lib/ml-helpers/Synthesize.thy`. This command can be seen as generalisation of **named-theorems**, with support for the kind of features just sketched.

  For a set of **synthesize-rules** you can generate patterns to support indexing of rules via subterms: **add-synthesize-pattern**.

  With **print-synthesize-rules** you can inspect the set:

print-synthesize-rules *pure*

You can also supply an term to select the matching rules:

print-synthesize-rules *pure ‹Trueprop (refines-spec (L2-try f) - (return ?f′) - (rel-prod rel-liftE (=)))›*

Rules are added via attribute *synthesize-rule*, where you can also specify the rule sets the priority and whether to split some variables. In order to preserve the theorem name for debugging purposes you should apply that attribute in a separate **lemmas** and not directly in the original **lemma**

The main tactic to drive application of those rules is `CT.cache_deepen_tac`. This tactic has type `context_tactic` and supports implementing a cache within the `Proof.context`. The user provides a cache and a single-step tactic that is recursively tried on the emerging subgoals.

The interface of the cache is captured in record `CT.ctxt_cache`:

- `#lookup: CT.ctxt_cache -> Proof.context -> cterm -> int -> context_tactic` The lookup function gets the goal presented as a `cterm` and returns a subgoal tactic. A cache miss is implemented by `K CT.no_tac`.
- `#insert: CT.ctxt_cache -> (Timing.timing * int * int) -> thm -> Proof.context -> Proof.context` The insert functions gets timing information tupled with the total number of alternatives and the current alternative as input together with the just proven subgoal. It can use that information to decide whether to really extend the cache or to ignore the result.
- `#propagate: CT.ctxt_cache -> Proof.context -> Proof.context -> Proof.context` To propagate the cache throughout backtracking the propagate function is supplied. It gets the current context and the old context as argument and can propagate the cache from the current context to the old one (to which it backtracks).

  The cache used for the type strengthening phase is implemented in `Monad_Convert.sim_nondet`. It is based on `Synthesize_Rules.gen_cond_cache`. It uses a term net to index the cached results and only adds rules for compound statements like *L2-seq*, not for atomic ones. The idea here is that proving simulation for an atomic statement is just a single rule application and is thus not worth to be cached.

  Failed proof attempts are represented by $FALSE \Longrightarrow PROP\ ?P$ instantiated with the failed subgoal. This is how they are presented to the cache and can be stored. When a cache lookup is performed and the cache results in such an instance of $FALSE \Longrightarrow PROP\ ?P$ which matches the current subgoal the proof attempt will be immediately aborted at that depth and potential alternative proof steps 'higher up' (smaller depth) may get explored.

Next we illustrate the approach with a small example.

lemma *refines-liftE-nondet*: *refines* (*liftE f*) *f s s* (*rel-prod rel-liftE* (=))
by (*auto simp add*: *refines-def-old reaches-liftE*)

lemma *refines-L2-unknown*:
shows *refines* (*L2-unknown ns*) (*L2-VARS* (*select UNIV*) *ns*) *s s* (*rel-prod rel-liftE* (=))
by (*rule Refines-Spec.refines-L2-unknown-nondet*)

lemmas *refines-spec-monad-rules* =
refines-L2-seq-nondet [*simplified THIN-def*]
refines-L2-gets-nondet
refines-L2-unknown

lemmas *refines-option-monad-rules* =
refines-L2-seq-option [*simplified THIN-def*]
refines-L2-gets-option

schematic-goal ‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-gets* (λ-. *n* + *m*) [])) (*?f'*::(*?'c* , *'a*) *res-monad*) *?s ?s ?R*›

apply (*rule refines-spec-monad-rules*)
prefer *2*
— Note that `CT.cache_deepen_tac` follows the convention of tactics to solve subgoals from the back. So the second subgoal is solved first.
apply (*rule refines-spec-monad-rules*)
apply (*rule refines-spec-monad-rules*)
done

First we define the single-step tactics, which just try some rules.

ML ‹
fun get-nondet-tacs goal = @{*thms refines-spec-monad-rules*}
|> *map* (*fn thm* => (*Utils.guess-binding-of-thm* @{*context*} *thm*, *thm*))
|> *CT.binding-resolve-tac*

fun get-option-tacs goal = @{*thms refines-option-monad-rules*}
|> *map* (*fn thm* => (*Utils.guess-binding-of-thm* @{*context*} *thm*, *thm*))
|> *CT.binding-resolve-tac*

fun get-all-tacs goal = *get-option-tacs goal* @ *get-nondet-tacs goal*
›

Now we define a simple cache which stores all emerging subgoals in a list and tries to resolve with those subgoals on a lookup.

ML ‹
structure List-Cache = *Proof-Data* (
type T = *thm list*
val init = *K* [])

```
val list-cache:CT.ctxt-cache = {
  lookup = fn ctxt => fn goal =>
    let
       val - = tracing (lookup: ^ Syntax.string-of-term ctxt (Thm.term-of goal))
    in
      CT.resolve-tac (List-Cache.get ctxt)
    end,
  insert = fn timing => fn thm => fn ctxt =>
    let
       val thm' = thm |> Thm.forall-elim-vars ((Thm.maxidx-of thm) + 1) |>
zero-var-indexes
      val - = tracing (insert: ^ Thm.string-of-thm ctxt thm')
    in
      ctxt |> List-Cache.map (cons thm')
    end,
  propagate = fn current => fn old => current
  }
›
```

We only try nondet-rules, so we end up in the nondet monad.

```
schematic-goal
‹refines ( L2-seq (L2-gets (λ-. n::nat) []) (λn. L2-gets (λ-. n + m) []))
  (?f'::(?'c, 'a) res-monad) ?s ?s ?R›
  apply (tactic ‹
    Context-Tactic.NO-CONTEXT-TACTIC @{context} (
    CT.cache-deepen-tac (K 1) list-cache get-nondet-tacs 1
   )›)
  done

schematic-goal
‹refines ( L2-seq (L2-gets (λ-. n::nat) []) (λn. L2-gets (λ-. n + m) [])) (?f'::(?'c,
'a) res-monad) ?s ?s ?R›
  thm refines-option-monad-rules
  apply (rule  refines-option-monad-rules)
   apply (rule  refines-option-monad-rules)
  apply (rule  refines-option-monad-rules)
  done
```

We only try the option-rules, so we end up in the option monad.

```
schematic-goal
‹refines ( L2-seq (L2-gets (λ-. n::nat) []) (λn. L2-gets (λ-. n + m) [])) (?f'::(?'c,
'a) res-monad) ?s ?s ?R›
  apply (tactic ‹
    Context-Tactic.NO-CONTEXT-TACTIC @{context} (
    CT.cache-deepen-tac (K 1) list-cache get-option-tacs 1
   )›)
  done
```

No we try both the option-rules and the nondet-rules. As the option

rules come first we end up in the option monad

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-gets* (λ-. *n* + *m*) [])) (*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
apply (*tactic* ‹
Context-Tactic.NO-CONTEXT-TACTIC @{*context*} (
CT.cache-deepen-tac (*K 1*) *list-cache get-all-tacs 1*
)›)
done

By including a *L2-unknown* in the example we have to end up in the nondet monad, as it cannot be handled in the option monad. Note the difference in the outcome of the fist and second proof attempt. In the first one we only provide the nondet-rules in the second one we also provide the option-rules. This results in a different "translation" of the first *L2-gets*.

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-unknown* [])) (*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
apply (*tactic* ‹
Context-Tactic.NO-CONTEXT-TACTIC @{*context*} (
CT.cache-deepen-tac (*K 1*) *list-cache get-all-tacs 1*
)›)
done

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-unknown* [])) (*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
apply (*tactic* ‹
Context-Tactic.NO-CONTEXT-TACTIC @{*context*} (
CT.cache-deepen-tac (*K 1*) *list-cache get-nondet-tacs 1*
)›)
done

end

26.17 Pointers to Local Variables

theory *pointers-to-locals* **imports** *AutoCorres*
begin

install-C-file *pointers-to-locals.c*
init−autocorres [*addressable-fields* = *pair.first buffer.buf 32 word*[*3*]]*pointers-to-locals.c*
autocorres [*no-heap-abs* = *inc-untyp*]*pointers-to-locals.c*

This story began with the desire to support "pointers to local variables". The idea to support "pointers to local variables" might be quite fearful as it covers lots of use cases. When trying to literally support pointers to local

variables (uniform with heap pointers) one has to answer questions about the memory layout of heap and stack, how to ensure that those regions are disjoint, what happens if we run out of stack space, how do we make sure that there is no dangling pointers to a stack-frame that is already popped from the stack etc.

So let us make a step back and ask ourselves for which C-idiomatic use cases are pointers to local variables actually used. C does only support call-by-value. So pointers are sometimes used to model call-by-reference semantics, for input as well as output parameters. Especially, the use case of an additional output parameter (besides the regular return value) is often implemented as a pointer parameter. An alternative might be to return a tuple encoded as a structure instead. But as there are no ad hoc tuples in C, this requires the boilerplate to define an auxiliary structure type. So typically a pointer parameter is used instead.

Here an example of a integer addition with overflow check. The return value is used for the status, the actual result is returned as a pointer parameter.

```
int add_check(int* result, int a, int b)
{
    *result = a + b;
    if(a > 0 && b > 0 && *result < 0)
        return -1;
    if(a < 0 && b < 0 && *result > 0)
        return -1;
    return 0;
}
```

A call to the function could be the following with a local variable for the result.

```
int res, status;
status = add_check(&res, 2, 3)
```

Here `res` is an output parameter. If C would support tuples we might have code like:

```
int add_check(int a, int b)
{
    int result = a + b;
    if(a > 0 && b > 0 && *result < 0)
        return (-1, resut);
    if(a < 0 && b < 0 && *result > 0)
        return (-1, result);
    return (0, result);
}
```

```
  int res, status;

  (status, res) = add_check(2, 3)
```

Another use case is to have in input / output parameter implemented by a pointer parameter.

```
void inc(int *a)
{
  (*a)++;
}

int x = 42;
inc(&x);
```

This could be also modelled as explicit input and output:

```
int inc(int a)
{
  return (a++);
}

int x = 42;
x = inc(x);
```

So one general idea is to model those kind of pointer-parameters as explicit input / output parameters. From a pattern like:

```
int f(int *p1, ..., int * p_n, int a1, ..., int a_m) {
  ... (*p1) ...
  ... (*p_i) ...
  return res;
}

r = f(q1,..., q_n, b1, ..., b_m)
```

We make

```
(int * int ... * int) f(int p1, ..., int p_n, int a1, ..., int
a_m) {
  ... p1 ...
  ... p_i ...
  return (res, p1, ... p_n);
}
```

```
(r, q1, ..., q_n) = f(q1,..., q_n, b1, ..., b_m)
```

In which cases is such a model faithful? Different behaviours might occur, when pointer-parameters are aliased. Moreover, the translation scheme only works out if the body of the function only uses the dereferenced pointer variables, i.e. `*p_i` to obtain the value or perform an update, and does not actually use the literal address value stored in the variable. E.g. it does not make things like `p1 = p2` or stores the pointer value in some global data structure. All examples we have seen so far are benign with this respect, as there was only one pointer-variable involved, and we only used it to dereference it.

What about this swap function:

```
void swap1(int* p, int* q) {
  int tmp = *p;
  *p = *q;
  *q = tmp;
}

int a = 1; int b =2;
swap1(&a, &b);
/* a = 2, b = 1 */


(int * int) swap2(int p, int q) {
  int tmp = p;
  p = q;
  q = tmp;
  return (p, q);
}

int a = 1; int b = 2;
(a, b) = swap2(a, b);
/* a = 2, b = 1 */
```

These two swap functions behave the same, even if the parameters would alias. But in general aliasing is problematic.

```
void inc_both1(int* p, int* q) {
  *p = *p + 1;
  *q = *q + 1;
}

int x = 2;
```

```
void inc_both1(&x, &x);

/* x = 4 */


(int * int) inc_both2(int p, int q) {
  p = p + 1;
  q = q + 1;
  return (p, q);
}

int x = 2;
(x, x) = inc_both2(&x, &x);

/* x = 3 */
```

Whats the relevant difference between the increment and the swap example? In the swap example, we never read from a pointer-parameter after any pointer-parameter was assigned to. So an update within the function might not alter the values we read via any pointer-parameter.

26.17.1 Design choices

C does not distinguish pointers to local variables from heap pointers they are all the same. E.g. the increment functions above could be applied to a heap location as well as a stack location. Moreover, in the context of systems programming it is common that low-level code explicitly deals with heap-layout and pointer values and "tricks" like storing some flags as bits in the pointer values themselves are often used. That is why we aim for an uniform model for stack and heap pointers and don't want to impose assumptions on the layout.

Whenever a function creates a pointer to a local variable, we model it as part of the heap. Core ideas:

- As part of ordinary heap-typing, we track free stack locations by a distinguished C type *stack-byte*. All addresses with that type are considered free stack location.
- Additionally to the typing information with *stack-byte* (that denotes free stack space) We have a fixed set of addresses $\mathcal{S}$ that describe all the stack space (free and allocated). Note that the set $\mathcal{S}$ does not depend on the state. The dynamic aspects are modelled within the heap typing.
- As prelude in a function that needs pointers to local variables we

allocate the variable in the heap, by non-deterministically retyping a region of free stack space that fits: We retype from *stack-byte* to the type we allocate.

- All accesses / updates to the variable within the function are modelled indirectly via the pointer
- As postlude of the function we again retype the stack space as free.

  Observations:

- Ordinary C functions, that do not have an AUX-UPDATE on the heap-type also do not mess around with the stack as they do not change heap-typing. So stack-space is preserved when calling a function.
- Prelude / postlude ensures that the stack typing is the same after a function call to a function that has pointers to local variables.

Stack Allocation

The central definition for stack allocation is *stack-allocs* describing the set of possible pointers and modified heap type descriptions:

$(p,\ d') \in$ *stack-allocs* $n\ \mathcal{S}$ *TYPE*$('a)\ d$

thm *stack-allocs-def*

After such an allocation we know that:

- Pointer p is a root pointer with the expected type in the new heap type description d'.
- The allocated addresses are within set $\mathcal{S}$.
- The new heap typing d' is obtained from the original heap typing d by retyping addresses of *stack-byte* to the type of the pointer p. The address of pointer p denotes the start the retyped region. The parameter n can be used to allocate a consecutive range of pointers in one step.

  Stack allocation might fail in the sense that the resulting set of *stack-allocs* can be empty. In Simpl we then fail in the terminal fault state reporting *StackOverflow*. In the autocorres monad we currently have decided to ignore possible stack overflows. We simply assume that stack allocation succeeds and does not return an empty set.

Stack Release

The counter part to the stack allocation above is *stack-releases n p d′*.

thm *stack-releases-def*

There is no non-determinism here. It is a plain function retyping back to *stack-byte*.

Stack discipline

We encapsulate the stacking discipline of allocation / release in some functions in the various layers.

Simpl In Simpl we have the command *With-Fresh-Stack-Ptr n init c* where

- *n* denotes the number of consecutive pointers we want to allocate, typically *1*. This number was introduced to allocate local arrays and obtain pointers to the elements. However, it turns out that allocating an array pointer $('a['b])$ *ptr* instead is sufficient. Hence the parameter is actually always *1*.
- *init s* returns a set of initial values from which we non-deterministically choose one. In case the addressed local variable is a function parameter this is a singleton set containing the value of the argument. In case of an ordinary local variable this is either the the universal set of all possible values (if uninitialized) or a singleton set of the initialisation value.
- *c* expects a fresh pointer *p* and then yields a Simpl command of the body.

  The idea is simple, we first allocate a pointer *p* with *stack-allocs*, then initialise the fresh heap location according to *init* then execute the body *c p* and finally release the stack pointer again.

thm *globals.With-Fresh-Stack-Ptr-def*
term *stack-heap-state.With-Fresh-Stack-Ptr*

A central role of the definition is the *DynCom* statements, which is the means of Simpl to bind certain execution points and provide state dependent commands. As the body of a *DynCom* always depends on the complete state (and not such a thing like the pointer *p*) we use the function *allocated-ptrs* calculate the fresh pointer from the typing information.

thm *allocated-ptrs-def*
thm *stack-allocs-allocated-ptrs*

Note that *spec-monad.Spec* might fail if the resulting set is empty, which is exposed to the failure *StackOverflow* of the command.

L1 / L2 In both L1 and L2 we use *globals.with-fresh-stack-ptr n init f*, where n as well as*init* are just the same as in Simpl and f is a monadic computation depending on the fresh pointer p.

term *globals.with-fresh-stack-ptr*
thm *globals.with-fresh-stack-ptr-def*
term *stack-heap-state.with-fresh-stack-ptr*

In contrast to Simpl we assume that stack allocation succeeds by using *assume-result-and-state*.

HL / Split Heap

In the split heap model there is no singleton heap anymore. Hence we introduce a family of functions depending on the type that is allocated.

term *heap-w32.with-fresh-stack-ptr*
thm *heap-w32.with-fresh-stack-ptr-def*
term *heap-w32.guard-with-fresh-stack-ptr*
thm *heap-w32.guard-with-fresh-stack-ptr-def*
term *heap-w32.assume-with-fresh-stack-ptr*
thm *heap-w32.assume-with-fresh-stack-ptr-def*
term *typ-heap-typing.with-fresh-stack-ptr*
term *typ-heap-typing.guard-with-fresh-stack-ptr*
term *typ-heap-typing.assume-with-fresh-stack-ptr*

While stack allocation remains the same upon stack release some more work has to be done in order to make the simulation work.

In the monolitic heap the stack release of a pointer p immediately results in *plift* $(h,$ *stack-releases* $n\ p\ d')\ p =$ *None*. So the simulation demands *the-default ZERO*$('a)$ $($*plift* $(h,$ *stack-releases* $n\ p\ d')\ p) =$ *zero*. [2].

So to simulate this in the split heap we explicitly zero out the memory. Moreover, for the simulation to work we have to argue that the stack release did not mess up with any of the other split heaps. This argument can be given if know that pointer p is still a valid root pointer before we do the release. As we know that p is a valid root pointer immediately after the allocation we have to establish that this is still true at stack release. Most C function do not alter the heap typing at all. So it can easily be shown that $f \cdot s$ *?*⦃ $\lambda r.$ *unchanged-typing-on* $\mathcal{S}\ s$ ⦄ holds and thus that p is still a valid root pointer after executing f. In case we cannot automatically proof the preservation of typing information, we enforce that p is still a valid root pointer by inserting a guard. So for code that changes the heap typing in more involved ways the argument is left for the user to prove.

thm *lifted-globals.unchanged-typing-on-def*
term *heap-typing-state.unchanged-typing-on*

Some syntax

[2] c.f. `open_struct.thy` for details on heap lifting simulation

term *PTR-VALID*(*32 word*)
term *IS-VALID*(*32 word*) *s*
term *IS-VALID*(*32 word*) *s p*

context *ts-definition-inc*
begin
thm *ts-def*
end

Pointer to an uninitialized local variable.

context *call-inc-local-uninitialized-impl*
begin
thm *call-inc-local-uninitialized-body-def*
end

context *ts-definition-call-inc-local-uninitialized*
begin
thm *ts-def*
end

Pointer to an initialized local variable.

context *ts-definition-call-inc-local-initialized*
begin
thm *ts-def*
end

Pointer to parameter.

context *ts-definition-call-inc-parameter*
begin
thm *ts-def*
end

When we cannot prove that the heap typing is unchanged we fall back to *heap-w32.guard-with-fresh-stack-ptr* instead of *heap-w32.assume-with-fresh-stack-ptr*

context *ts-definition-call-inc-untyp*
begin
thm *ts-def*
end

Nested pointers among the phases.

context *call-inc-nested-impl*
begin
thm *call-inc-nested-body-def*
end

context *l1-definition-call-inc-nested*
begin
thm *l1-def*

end

context *l2-definition-call-inc-nested*
begin
thm *l2-def*
end

context *hl-definition-call-inc-nested*
begin
thm *hl-def*
end

context *wa-definition-call-inc-nested*
begin
thm *wa-def*
end

context *ts-definition-call-inc-nested*
begin
thm *ts-def*
end

Global variable

context *ts-corres-call-inc-global*
begin
thm *ts-def*
end

Local array variable.

context *ts-definition-call-inc-array*
begin
thm *ts-def*
end

Local structure variable.

context *ts-definition-call-inc-first*
begin
thm *ts-def*
end

Local array in structure variable.

context *ts-definition-call-inc-buffer*
begin
thm *ts-def*
end

Mutual recursion and pointer to local variables.

context *l2-definition-odd-even*
begin

thm *unchanged-typing*
end
context *ts-corres-odd-even*
begin
thm *ts-def*
end

Proof rules for *runs-to-vcg*

context *pointers-to-locals-all-corres* **begin**

thm *stack-ptr-simps*

lemma (*call-inc-local-initialized'*) · *s* {‖ λ *r t*. *t* = *s* ∧ *r* = *Result 43* ‖}
apply (*unfold call-inc-local-initialized'-def inc'-def*)
apply *runs-to-vcg*
apply (*auto simp*: *stack-ptr-simps comp-def*)
done

lemma (*call-inc-local-uninitialized'*) · *s* {‖ λ *r t*. *t* = *s* ∧ *r* = *Result 42* ‖}
apply (*unfold call-inc-local-uninitialized'-def inc'-def*)
apply *runs-to-vcg*
apply (*auto simp*: *stack-ptr-simps comp-def*)
done

lemma (*call-inc-parameter' n*) · *s* {‖ λ *r t*. *t* = *s* ∧ *r* = *Result* (*n* + *1*) ‖}
apply (*unfold call-inc-parameter'-def inc'-def*)
apply *runs-to-vcg*
apply (*auto simp*: *stack-ptr-simps comp-def*)
done

end

26.17.2 Open Ends / TODOs

There are not yet any user level proof rules for *heap-w32.guard-with-fresh-stack-ptr* and *heap-w32.assume-with-fresh-stack-ptr* to include with *runs-to-vcg*. Note that there are a lot of theorems on the primitives *stack-allocs* and *stack-releases*.

thm *stack-allocs-cases*
thm *stack-allocs-ptr-valid-cases*
thm *stack-releases-ptr-valid-cases*
thm *stack-releases-ptr-valid-cases1*

Value parameters aka. In-Out parameters

Now that we have proper pointers to local variables there is room for further abstractions, motivated by the prominent use cases described in the begin-

ning. Those use cases suggest that in a lot of cases one can completely get rid of stack allocation / release by introducing value parameters: Instead of just passing a pointer value into the function, one passes in the actual (dereferenced) value and tuples the return value of the function with the final (dereferenced) value of that parameter.

It turns out that the core transformation here is not so much about pointers to local variables but more on transforming functions from passing in a pointer parameter to another function value parameters. This part of the transformation is independent of the question heap pointer vs. stack pointer. After this transformation is done for the body of a function which allocates a stack pointer, the actual address of the stack pointer becomes meaningless as it is only used in "dereferenced form". So we can remove the stack allocation / stack release and replace it by an ordinary local variable.

See `In_Out_Parameters_Ex.thy` for more information.

end

26.18 In-Out Parameters, Abstracting Pointers to Values

theory *In-Out-Parameters-Ex* **imports** *AutoCorres*
begin

consts *abs-step*:: *word32* ⇒ *word32* ⇒ *bool*

26.18.1 Overview

We introduce a new phase in AutoCorres to replace pointer parameters by *in/out parameters*: Instead of passing a pointer into a function we pass the initial value of the pointer (by dereferencing the pointer) into the function and return the value at the end of the function as additional output. E.g. a function `void inc(unsigned *n)` becomes `unsigned inc(unsigned n)`. The initial motivation for this phase was to get rid of explicit pointers to local variables and replace them by ordinary values instead. This conversion has two main aspects:

- Function signatures may change: pointer parameters become value parameters and the function may return the value of the pointer at the end as an additonal return value (tupled with the ordinary return value). Functions with side effects on the heap may become pure functions on values by this transformation.
- As a result of the first transformation, pointers to local variables may be eliminated and be replaced by bound variables. This means that *stack-heap-state.with-fresh-stack-ptr* disappears.

The new phase is called *IO* and is placed between L2 and HL. This means local variables are already represented as bound variables in L2 and we still operate on a monolithic heap.

26.18.2 Building Blocks

What ingredients do we need for the abstraction relation in the refinement proof?

- Heaps: As we eliminate pointers, the original function manipulates the heap more often then the resulting function. So we need a notion to relate the heap states and keep track of the relevant pointers. As the stack of local variable pointers is also modelled in the heap the stack free locations are also of interest. The heaps of the original and the resulting program may differ in these locations. The heap value of a stack free location should be irrelevant for the program.
- Function signatures: The function signature changes, pointer parameters become ordinary values, the return value is tupled with output parameters.

  In the following we refer to the original state (containing the heap) as s and the resulting / abstract state as t. As I first idea we want to relate states s and t such that the heaps are the same except for some pointers we want to eliminate and the stack free locations. It would be natural to describe this as a relation. However, it turns out that dealing with relations within the refinement proof are hard to make admissible in order to support recursive functions. Having a abstraction / lifting function (instead of a relation) from s to t makes admissibility straightforward.

  As a prerequisite we encourage the reader to consider the documentation about pointers to local variables: `pointers_to_locals.thy`

context *heap-state*
begin

frame

We want to express that certain portions of the heap (typing and values) are 'irrelevant', in particular regarding (free) stack space. The notion of 'irrelevant' is a bit vague, it means that the behaviour of the resulting program does not depend on those locations and also that it does not modify those locations. Moreover, we prefer an abstraction function rather than an relation between states to avoid admissibility issues for refinement. The

central property is $t = \mathit{frame}\ A\ t_0\ s$, for A being a set of addresses and t, t_0 and s being states. Think of A as the set of *allocated* addresses containing those pointers we want to *abstract* aka. eliminate.

thm *frame-def*

The standard use case we have in mind is $A \cap \mathit{stack\text{-}free}\ (\mathit{htd}\ s) = \{\}$, hence $A - \mathit{stack\text{-}free}\ (\mathit{htd}\ s) = A$, but nevertheless the intuition is:

- stack free typing from s is preserved, the framed sate has at least as many stack free addresses as the original one. So we can simulate any stack allocation.
- heap values for stack free and A are taken from reference state t_0, this captures that we do not depend on the original values in s for those addresses.
- typing for allocations in A is taken from reference state t_0, this captures that we do not depend on the original typing in s for those addresses.

  By taking the same reference state t_0 to frame two states s and s', we can express that the 'irrelevant' parts of the heap did not change in the respective framed states $\mathit{frame}\ A\ t_0\ s$ and $\mathit{frame}\ A\ t_0\ s'$.

rel-alloc, *rel-stack*, *rel-sum-stack*

The core invariant between the states that is maintained by the refinement is $\mathit{rel\text{-}alloc}\ \mathcal{S}\ M\ A\ t_0\ s\ t$:

- $\mathcal{S}$ is a static set of addresses containing the stack. Stack allocation and stack release are contained within $\mathcal{S}$.
- M is the set of addresses that might be modified by the original program.
- A is the set of addresses that are eliminated (abstracted) in the resulting program. Another good intuition about the set A is that the original function might read from and depend on those addresses but the resulting function does not. Moreover, the resulting function does not change any of heap locations in A. The resulting function is agnostic to those locations. It neither reads nor modifies them. For any heap valuation of A the abstract function simulates the original one.
- t_0 is the reference state explained above.
- s is the state of the original program
- t is the state of the resulting program.

thm *rel-alloc-def* [*of* $\mathcal{S}$ M A t_0]

The properties of *rel-alloc* are:

- $t = \mathit{frame}\ A\ t_0\ s$, the resulting heap in t is the same as the original in s, except for the addresses in A and the stack free space.
- $\mathit{stack\text{-}free}\ (\mathit{htd}\ s) \subseteq \mathcal{S}$: The stack free space is contained in $\mathcal{S}$.
- $\mathit{stack\text{-}free}\ (\mathit{htd}\ s) \cap A = \{\}$: We only want to eliminate properly allocated pointers, which are not contained in the stack free space.
- $\mathit{stack\text{-}free}\ (\mathit{htd}\ s) \cap M = \{\}$: We only mention properly allocated pointers to be modified. Modifications within the stack free space are irrelevant, as they are abstracted by the framing anyways.

For an original function f and the abstract aka. lifted function g the refinement property has the following shape: $\mathit{rel\text{-}alloc}\ \mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow \mathit{refines}\ f\ g\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ M1\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ L\ R))$

Note that the output relation of *refines* relates both the resulting states and return values to each other. The return values of the error monad are related by *rel-sum-stack*, taking a relation L for error values (*Inl*) and the R for normal termination (*Inr*).

thm *rel-sum-stack-def*
thm *rel-stack-def* [*of* $\mathcal{S}$ *M1* A s t_0 *rel-sum-stack* L R]

Consider the output states s' and t' and the output values v and w for the original and the abstracted function. Then we have:

- $\mathit{rel\text{-}xval\text{-}stack}\ L\ R\ (\mathit{hmem}\ s')\ v\ w$: The result values are related, (see below).
- $\mathit{rel\text{-}alloc}\ \mathcal{S}\ M1\ A\ t_0\ s'\ t'$: In particular $t' = \mathit{frame}\ A\ t_0\ s'$. Note that we use the same A and the same reference state t_0 as for the initial states. So this means that the abstract program g does not change the heap values and typing in A and does not depend on the values.
- $\mathit{equal\text{-}upto}\ (M1 \cup \mathit{stack\text{-}free}\ (\mathit{htd}\ s'))\ (\mathit{hmem}\ s')\ (\mathit{hmem}\ s)$: The original program only modifies pointers contained in *M1* or in the stack free portions of the heap (e.g. by nested function calls).
- $\mathit{equal\text{-}upto}\ M1\ (\mathit{htd}\ s')\ (\mathit{htd}\ s)$: Changes to the heap typing of the original program are confined within set *M1*.
- $\mathit{equal\text{-}on}\ \mathcal{S}\ (\mathit{htd}\ s')\ (\mathit{htd}\ s)$: The stack typing of the original program remains unchanged. Note that it might change temporarily by nested function calls but the stacking discipline restores the state. By construction we have $M1 \subseteq M$.

Stacking pointers: *rel-singleton-stack*, *rel-push* The relation on the output value captures the idea that portions of the heap are stacked / tupled into result values. On the boundaries of a function the relation on error values L is always the trivial λ-. (=). Errors are terminal as there is no exception handling in C. Within the boundaries of a function there can be more complex relations reflecting the nesting of break / continue / return and goto. For ordinary values the values of pointers are tupled by *rel-singleton-stack*, in case the original function returned no result (void), or nested by *rel-push* in case the original function returned some result.

thm *rel-singleton-stack-def*
thm *rel-push-def*

A typical relation is R = *rel-singleton-stack p* for some pointer *p*. This means that the abstract value can be obtained by looking into the heap at pointer *p*: *rel-singleton-stack p h* () *v* means that *v* = *h-val h p*. The heap of the original program is propagated down the into the relations. Similar *rel-push p* (λ-. (=)) *h x* (*v*, *x*) relates the result value *x* to the tuple (*v*, *x*) where *v* = *h-val h p*. Note that *rel-push* can be nested represent multiple output parameters. Depending on the role of the pointers there are additional assumptions about the pointers, in particular things like *ptr-span p* $\subseteq$ *A* or *ptr-span p* $\subseteq$ *M* and disjointness properties like *distinct-sets* [*ptr-span p*, *ptr-span q*].

IOcorres

There is an additional predicate *IOcorres* which represents the result of the refinement proof in the canonical autocorres ideom that is used within the other phases. This form is what is expected when constructing the final theorem combining all autocorres layers ⟦*L1corres ?check-termination ?Gamma ?c-L1.0 ?c*; *L2corres ?st-L2.0 ?rx-L2.0 ?ex-L2.0 ?P-L2.0 ?c-L2.0 ?c-L1.0*; *IOcorres ?P-IO ?Q-IO ?st-IO ?rx-IO ?ex-IO ?c-IO ?c-L2.0*; *L2Tcorres ?st-HL ?c-HL ?c-IO*; *corresTA ?P-WA ?rx-WA ?ex-WA ?c-WA ?c-HL*; ⋀*s. refines ?c-WA ?A s s* (*rel-prod rel-liftE* (=))⟧ ⟹ *ac-corres′* (λ-. *Exception default*) (*?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?check-termination* {*AssumeError*, *StackOverflow*} *?Gamma* (*λs.* (*?rx-WA* ∘ (*λv. ?rx-IO v* (*?st-L2.0 s*)) ∘ *?rx-L2.0*) *s*) (*λs.* (*?ex-WA* ∘ (*λe. ?ex-IO e* (*?st-L2.0 s*)) ∘ *?ex-L2.0*) *s*) (*?P-L2.0 and ?P-IO* ∘ *?st-L2.0 and ?P-WA* ∘ *?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?A ?c*

⟦*L1corres ?check-termination ?Gamma ?c-L1.0 ?c*; *L2corres ?st-L2.0 ?rx-L2.0 ?ex-L2.0 ?P-L2.0 ?c-L2.0 ?c-L1.0*; *IOcorres ?P-IO ?Q-IO ?st-IO ?rx-IO ?ex-IO ?c-IO ?c-L2.0*; *L2Tcorres ?st-HL ?c-HL ?c-IO*; *corresTA ?P-WA ?rx-WA ?ex-WA ?c-WA ?c-HL*; ⋀*s. refines ?c-WA ?A s s* (*rel-prod* (=) (=))⟧ ⟹ *ac-corres′ Exn* (*?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?check-termination* {*AssumeError*, *StackOverflow*} *?Gamma* (*λs.* (*?rx-WA* ∘ (*λv. ?rx-IO v* (*?st-L2.0*

s)) ∘ *?rx-L2.0*) *s*) (λ*s*. (*?ex-WA* ∘ (λ*e*. *?ex-IO e* (*?st-L2.0 s*)) ∘ *?ex-L2.0*) *s*) (*?P-L2.0 and ?P-IO* ∘ *?st-L2.0 and ?P-WA* ∘ *?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?A ?c*. The *refines* version and the *IOcorres* version can be converted into each other by ⟦*L1corres ?check-termination ?Gamma ?c-L1.0 ?c*; *L2corres ?st-L2.0 ?rx-L2.0 ?ex-L2.0 ?P-L2.0 ?c-L2.0 ?c-L1.0*; *IOcorres ?P-IO ?Q-IO ?st-IO ?rx-IO ?ex-IO ?c-IO ?c-L2.0*; *L2Tcorres ?st-HL ?c-HL ?c-IO*; *corresTA ?P-WA ?rx-WA ?ex-WA ?c-WA ?c-HL*; ⋀*s*. *refines ?c-WA ?A s s* (*rel-prod rel-liftE* (=))⟧ ⟹ *ac-corres′* (λ-. *Exception default*) (*?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?check-termination* {*AssumeError*, *StackOverflow*} *?Gamma* (λ*s*. (*?rx-WA* ∘ (λ*v*. *?rx-IO v* (*?st-L2.0 s*)) ∘ *?rx-L2.0*) *s*) (λ*s*. (*?ex-WA* ∘ (λ*e*. *?ex-IO e* (*?st-L2.0 s*)) ∘ *?ex-L2.0*) *s*) (*?P-L2.0 and ?P-IO* ∘ *?st-L2.0 and ?P-WA* ∘ *?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?A ?c*

⟦*L1corres ?check-termination ?Gamma ?c-L1.0 ?c*; *L2corres ?st-L2.0 ?rx-L2.0 ?ex-L2.0 ?P-L2.0 ?c-L2.0 ?c-L1.0*; *IOcorres ?P-IO ?Q-IO ?st-IO ?rx-IO ?ex-IO ?c-IO ?c-L2.0*; *L2Tcorres ?st-HL ?c-HL ?c-IO*; *corresTA ?P-WA ?rx-WA ?ex-WA ?c-WA ?c-HL*; ⋀*s*. *refines ?c-WA ?A s s* (*rel-prod* (=) (=))⟧ ⟹ *ac-corres′ Exn* (*?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?check-termination* {*AssumeError*, *StackOverflow*} *?Gamma* (λ*s*. (*?rx-WA* ∘ (λ*v*. *?rx-IO v* (*?st-L2.0 s*)) ∘ *?rx-L2.0*) *s*) (λ*s*. (*?ex-WA* ∘ (λ*e*. *?ex-IO e* (*?st-L2.0 s*)) ∘ *?ex-L2.0*) *s*) (*?P-L2.0 and ?P-IO* ∘ *?st-L2.0 and ?P-WA* ∘ *?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?A ?c*.

thm *IOcorres-def*
thm *IOcorres-refines-conv*
thm *ac-corres-chain-sims*
end

install-C-file *in-out-parameters.c*

26.18.3 Options

To control the in/out- parameter abstraction there are two options of autocorres:

- *skip-io-abs* a boolean which is *true* by default. This skips the complete phase for all functions.
- *in-out-parameters* which takes a list of in-out specifications for functions. As soon as there is at least one specification present the phase is enabled (taking precedence over *skip-io-abs*.

  So to properly enable the IO phase you have two main choices to specify what should be done either in **init−autocorres** or subsequent calls to **autocorres**. You can only disable option *skip-io-abs* in **init−autocorres** and then provide the individual options for *in-out-parameters* in subsequent **autocorres** invocations, or you can already define all *in-out-parameters* already in **init−autocorres**.

More details on the options are provided in the following examples.

26.18.4 Examples

init−autocorres [*in-out-parameters* =
compare(*cmp*:*out*) **and**
inc(*y*:*in-out*) **and**
inc-int(*y*:*in-out*) **and**
dec(*y*:*in-out*) **and**
swap(*x*:*in-out*, *y*:*in-out*) **and**
swap-pair(*x*:*in-out*, *y*:*in-out*) **and**
call-inc-ptr(*p*: *in-out*) **and**
inc-twice(*y*:*in-out*) **and**
inc2(*y*:*in-out*, *z*:*in-out*) **and**
heap-inc2-part(*z*:*in-out*) **and**
heap-inc2-part-swap(*y*:*in-out*) **and**
safe-add(*result*: *in-out*) **and**
inc-pair(*p*:*in-out*) **and**
get-arr-idx(*arr*:*in*) **and**
get-pair-arr-idx-second(*arr*:*in*) **and**
inc-pair(*p*:*in-out*) **and**
xyz(*x*:*in-out*) **and**
resab(*res*:*in-out*) **and**
abcmp(*cmpRst*:*in-out*) **and**
out(*out*:*out*) **and**
out-two(*out1*:*out*, *out2*:*out*) **and**
out2(*out*:*in-out*) **and**
out-seq(*out*:*out*) **and**
inc-loop (*x*:*in*, *y*:*in-out*) **and**
call-inc-int-other(*m*: *in-out*) **and**
inc-int-no-exit(*y*:*in-out*) **and**
call-inc-int-other-mixed(*m*: *in-out*) **and**
inc-int2(*y*:*in-out*, *z*:*in-out*) **and**
call-inc-int2(*n*:*in-out*) **and**
call-inc-int-pair(*p*:*in-out*) **and**
call-inc-int-array(*p*:*in-out*) **and**
call-compare-ptr(*m*:*out*) **and**
mixed-global-local-inc(*q*:*in-out*) **and**
g1(*out*:*in-out*) **and**
set-void(*p*:*data*) **and**
set-void2(*p*:*data*) **and**
set-byte(*p*:*data*) **and**
read-char(*p*:*in-out*, *len*: *in-out*) **and**
goto-read-char1(*p*:*in-out*) **and**
goto-read-char5(*elem*: *in*) **and** *smex7*() **and**
cmp-no-io() **where** *disjoint* [] **and**
call-cmp-no-io1() **where** *disjoint* [] **and**
deref-ptr(*p*:*in*) **and**

deref-ptr-glob(*p*:*in*) **and**
call-deref-ptr-glob-single() **and**
call-deref-ptr-glob-pair() **and**
uinc(*p*:*in-out*) **and**
udec(*p*:*in-out*) **and**
call-unop() **for** *p*(*in-out*); **and**
call-binop() **for** *p*(*keep*∗, *keep*∗); **and**
f-empty1(*p*:*in-out*) **and**
f-empty2(*p*:*in-out*) **and**
call-f-only-return(*p*:*in-out*) ,
method-in-out-fun-ptr-specs =
object.unop(*keep*∗) **and**
object.binop(*data*, *data*),

in-out-globals =
keep-inc-global-array
inc-global-array
keep-inc-global-array2
inc-global-array2
call-keep-inc-global-array
call-inc-global-array
mixed-global-local-inc shuffle global-array-update
read-char call-read-char call-read-char-loop
goto-read-char1 goto-read-char2 goto-read-char3 goto-read-char4 goto-read-char5
goto-read-char6
call-cmp-no-io call-cmp-no-io1
deref-ptr-glob call-deref-ptr-glob-pair call-deref-ptr-glob-single
,
ignore-addressable-fields-error,
addressable-fields =
pair.first
pair.second
array.elements
int-pair.int-first
int-pair.int-second

,
ts-force exit = *do-return*
]*in-out-parameters.c*

autocorres [*ts-force option* = *fnc1*] *in-out-parameters.c*

context includes *in-out-parameters-wa-impl-corres*
begin
thm *fun-ptr-intros*
term $\mathcal{P}$*-io-ptr-to-unsigned---unsigned*
end

context includes *in-out-parameters-ts-impl-corres*

begin
thm *fun-ptr-intros*
thm *known-function*
thm *known-function-corres*
end

context *in-out-parameters-hl-corres*
begin
thm *known-function*
thm *known-function-corres*
end

context *in-out-parameters-wa-corres*
begin
thm *known-function*
thm *known-function-corres*
end
context *in-out-parameters-ts-corres*
begin
thm *known-function*
thm *known-function-corres*
end
context *in-out-parameters-l2-corres*
begin
thm *known-function*
thm *known-function-corres*
end

context *in-out-parameters-io-corres*
begin
thm *known-function*
thm *known-function-corres*
end

In option *in-out-parameters* you provide a parameter specification for the functions you want to abstract. The parameter specs following the parameter name are

- *in*: pointer is only used as input to the function. The referenced value does not change during the function call.
- *out*: the pointer is only used to output a result from the function. The referenced value at the beginning is irrelevant, the abstracted function should return the referenced value at the end of the original function.
- *in-out* the pointer is used to input a value and t o return a result.

  When no specification is given for a function, the list is considered empty. This means that the signature of the function shall not be

changed by the abstraction. Pointer parameters in the original function stay pointer parameters in the resulting function. Note that this has a quite different effect from just skipping the phase, as autocorres is still attempting to do a proper refinement proof. More details are provided by the examples.

locale *keep-globals = in-out-parameters-global-addresses +*
  assumes *keep-global-array*: $\{$*ptr-val global-array ..+128*$\} \subseteq \mathcal{G}$
begin
lemma *global-array-contained1*:
  assumes *i-bound*: $i < 12$
  shows *ptr-span* (*global-array* $+_p$ (*int i*)) $\subseteq \mathcal{G}$
proof −
  have *ptr-span* (*global-array* $+_p$ (*int i*)) $\subseteq$ $\{$*ptr-val global-array ..+128*$\}$
    using *i-bound*
    apply (*clarsimp simp add*: *ptr-add-def intvl-def*)
    subgoal for *k*
    apply (*rule exI* [**where** *x*= $i * 4 + k$])
      by *force*
    done
  with *keep-global-array* **show** *?thesis* **by** *blast*
qed

lemma *gobal-array-contained2*[*polish*]:
  assumes *i-bound*: *numeral i* $<$ (*12*::*nat*)
  shows *ptr-span* (*global-array* $+_p$ (*numeral i*)) $\subseteq \mathcal{G}$
  using *global-array-contained1*
  by (*metis i-bound of-nat-numeral*)

end

autocorres [

  ts-force nondet = shuffle] *in-out-parameters.c*

context *in-out-parameters-all-corres*
begin

thm *io-corres*
thm *ts-def*
term *ptr-valid*

inc′

The function `inc` increments the value which is passed as a pointer `y`.

```
void inc (unsigned* y) {
  *y = *y + 1;
}
```

When we abstract this function to have an in-out parameter instead, we obtain a function like:

```
void inc (unsigned y) {
  y = y + 1;
  return y;
}
```

Note that this is now a pure function, that does no longer depend on the heap. This is also the final output of autocorres in *inc′ ?y ≡ ?y + 1*:

thm *inc′-def*

Let us zoom into the refinement step from L2 to IO.

thm *l2-inc′-def*
thm *io-inc′-def*

Note that the type of parameter y changes from a pointer type to a value type. Here is the generated refinement theorem.

thm *io-inc′-corres*
lemma *c-guard y* $\Longrightarrow$
distinct-sets [*ptr-span y*] $\Longrightarrow$
ptr-span y $\subseteq$ *A* $\Longrightarrow$
ptr-span y $\subseteq$ *M* $\Longrightarrow$
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines
(*l2-inc′ y*)
(*io-inc′* (*h-val* (*hrs-mem* (*t-hrs-′ s*)) *y*)) *s t*
(*globals.rel-stack* $\mathcal{S}$ (*ptr-span y*) *A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (*rel-singleton-stack y*)))
by (*rule io-inc′-corres*)

Parameter y was specified as *in-out*. So function *io-inc′* becomes the actual value of the pointer passed in: *h-val* (*hrs-mem* (*t-hrs-′ s*)) *y*. To discharge the *c-guard y* guards within the body of *l2-inc′* we need the assumption that the guard holds at the beginning. As we attempt to abstract pointer y we have its pointer span included in set A. Moreover, the content of pointer y is modified by *l2-inc′* hence it appears in M as well as explicit in the final relation *globals.rel-stack*. Pointer parameters are assumed to be distinct. As we have only one parameter the assumption trivially *distinct-sets* [*ptr-span y*] holds. Note that set A is only constrained by *ptr-span y* $\subseteq$ *A*. So we could instantiate it with the universal set *UNIV*. Recalling the meaning of A this means that the resulting function *io-inc′* is a pure function, in the sense that it does not depend on the heap anymore. The relation for the termination with `exit` is *rel-exit* (λ- - -. *False*), which expresses that this path is unreachable for the current function. The function will never terminate with `exit`.

call-inc-parameter′

Next we look at a function that calls our previously defined integer function on a pointer to its parameter.

```
unsigned call_inc_parameter(unsigned n) {
  inc(&n);
  return(n);
}
```

After autocorres the function is just a mere wrapper to the increment function

thm *call-inc-parameter′-def*

Here are the relevant definitions and the IO refinement theorem.

thm *l2-call-inc-parameter′-def*
thm *io-call-inc-parameter′-def*
thm *io-call-inc-parameter′-corres* [*no-vars*]

thm *io-call-inc-parameter′-corres*
lemma *distinct-sets* ([]::*addr set list*) $\Longrightarrow$
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines
(*l2-call-inc-parameter′ n*)
(*io-call-inc-parameter′ n*) *s t*
(*globals.rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
by (*rule io-call-inc-parameter′-corres*)

Note that here we do not have any assumption on the set A and the modified variables in the final state are empty. This is because we eliminate a pointer to a temporary stack variable that is out of scope after the function and in the beginning of the function it is part of the *stack-free* space.

const*keep-inc′*

Now let us consider a copy of the increment function were we do not convert to in out parameters but leave it as it is. Unsurprisingly, the resulting function still operates on pointers.

thm *keep-inc′-def*

Let us look into the IO corres phase.

thm *l2-keep-inc′-def*
thm *io-keep-inc′-def*
thm *io-keep-inc′-corres*

lemma *distinct-sets* [*ptr-span y*] $\Longrightarrow$
ptr-span y $\cap$ *A* = {} $\Longrightarrow$
ptr-span y $\cap$ *stack-free* (*hrs-htd* (*t-hrs-′ s*)) = {} $\Longrightarrow$ — FIXME: redundant assumption?, follows from the next ones

globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
ptr-span y $\subseteq$ *M* $\Longrightarrow$
refines
(*l2-keep-inc′ y*)
(*io-keep-inc′ y*) *s t*
(*globals.rel-stack* $\mathcal{S}$ (*ptr-span y*) *A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
by (*rule io-keep-inc′-corres*)

As pointer y is an allocated pointer that we do not abstract it should not belong to A and must not collide with *stack-free* space. As we might modify the pointer it is contained in M as well as the final modifies set.

Note that the bodies of *io-keep-inc′* and *l2-keep-inc′* are equivalent but the refinement statement is not a trivial statement and in particular is not the same as when skipping the IO corres phase. For example from the refinement statement we can derive that any heap location different from *ptr-span y* is unchanged.

Next let's look what happens when we call this function on a local parameter.

keep-call-inc-parameter′

thm *keep-call-inc-parameter′-def*
thm *io-keep-call-inc-parameter′-def*
thm *l2-keep-call-inc-parameter′-def*
thm *io-keep-call-inc-parameter′-corres* [*no-vars*]

lemma *distinct-sets* ([]::*addr set list*) $\Longrightarrow$
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines
(*l2-keep-call-inc-parameter′ n*)
(*io-keep-call-inc-parameter′ n*) *s t*
(*globals.rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
by (*rule io-keep-call-inc-parameter′-corres*)

The local variable pointer is still present in the final function. Still the refinement statement indicates that we have a 'pure' functions, as we do not have constraints on A and also do not modify anything. The notion of pure we use here, is that the function does not depend on proper heap pointers. The *stack-free* space that we temporarily use is excluded by our definitions of *globals.frame*, *globals.rel-alloc* and *globals.rel-stack*.

safe-add′

This function is an example of an arithmetic function that checks for overflows. The status of the check is the return value and the result of the operation is passed via pointer. We make an in-out parameter for the re-

sult. So the final function returns a tuple of the result value and the status (*rel-push* in the refinement statement).

thm *safe-add'-def*
thm *l2-safe-add'-def*
thm *io-safe-add'-def*
thm *io-safe-add'-corres*

Recursion

thm *l2-fac'.simps*
thm *io-fac'.simps*
thm *io-fac'-corres*

thm *l2-even'.simps l2-odd'.simps*
thm *io-even'.simps io-odd'.simps*
thm *io-even'-corres io-odd'-corres*

swap'

We abstract a function that swaps the values referenced by two input pointers by a function that takes the values and returns the swapped values (as a pair). Note that in the refinement statement we assume disjointness of the input pointers. The swap function would still be correct if the pointers are equal. To generalise the refinement proofs to 'disjoint or equal' inputs is left as future work.

thm *swap'-def*
thm *l2-swap'-def*
thm *io-swap'-def*
thm *io-swap'-corres*

Out parameters

The function *compare* has an mere 'out' parameter. Until phase L2 the function has three parameters, after phase IO the function only has two parameters.

thm *l2-compare'-def*
thm *io-compare'-def*
thm *compare'-def*

thm *l2-call-compare'-def*
thm *io-call-compare'-def*
thm *call-compare'-def*

Pointers to compound types (structs / arrays)

Here are some examples of function that work on (parts) of a compound type.

thm *get-arr-idx'-def*
thm *l2-get-arr-idx'-def*
thm *io-get-arr-idx'-def*

thm *inc-pair'-def*
thm *l2-inc-pair'-def*
thm *io-inc-pair'-def*

Misc

thm *wa-def*
thm *ts-def*
thm *io-corres*

26.18.5 Handling `exit` in function calls

When we call a function that was refined by in-out-parameters there are various cases we have to consider, e.g. do we pass in a heap pointer or a local pointer, do we eliminate the local pointer or keep it? In general when we pass in a pointer that is not supposed to be eliminated as an in-out parameter we first pass in the dereferenced value and after the function returns we take the value from the embellished result and assign it to the pointer. So the pointer assignment that would take place in the middle of the called function in the original program, is moved outside of the call in the refined program. After this the heaps of the original program and the refined program are in sync again at the pointer. To establish the heap refinement it turns out that we also have to bring the heaps in sync again even if the function terminates with `exit`. We catch the `exit`, update the heap and then re-throw the *exit*. So the called function also returns embellished exit values which hold the value of the relevant pointers. This extra work is a little annoying as the `exit` terminates the complete program and usually we do not catch an `exit`. So this step might introduce *L2-catch* to wrap up the procedure call. Note that this is currently the only remaining use case for *L2-catch*, the other occurrences of *L2-catch* were already transformed to *L2-try* as post processing in phase L2 when we move from 'flat' exception handling to 'nested' exception handling.

term *inc-int'*
thm *inc-int'-def*
term *call-inc-int-ptr'*
thm *call-inc-int-ptr'-def*

26.18.6 Global Heap Pointers

A core part of the IO phase is to keep track of pointers and in particular to do some bookkeeping of the parts of the heap that might get modified. This analysis and bookkeeping is focused around the pointers that are visible in

the signature of a function. This fails short as soon as pointers to global heap are used within the body of a function, which are not mentioned in the function signature. To handle some common use cases with global heap we have a mechanism to over-approximate the impact of a function on the heap by referring to a static set $\mathcal{G}$ of addresses that should not be abstracted by the IO phase. We can annotate a function to make use of this mechanism by mentioning them in **init−autocorres** with the option `in_out_globals`. For the refinement statement we assume that

- $\mathcal{G} \cap A = \{\}$: pointers in $\mathcal{G}$ are not supposed to be abstracted, and
- $\mathcal{G} \subseteq M$: the function might modify any global variable.

  Within the body of such a function (marked with option `in_out_globals`), whenever a pointer is used which does not occur in the signature as pointer parameter and is specified otherwise we assume that this pointer a global pointer and assert this formally by adding a guard *ptr-span* $x \subseteq \mathcal{G}$. If a function has multiple pointer parameters we also assert disjointness of pointers.

thm *io-keep-inc-global-array'-corres*
thm *io-keep-inc-global-array'-def*
thm *l2-keep-inc-global-array'-def*

thm *io-inc-global-array'-corres*
thm *io-inc-global-array'-def*
thm *l2-inc-global-array'-def*

thm *io-inc-global-array2'-corres*
thm *io-inc-global-array2'-def*
thm *l2-inc-global-array2'-def*
thm *ac-corres*

Note that a function can also have both in-out parameters and `in_out_globals`.

thm *io-read-char'-corres*
thm *io-read-char'-def*
thm *l2-read-char'-def*

thm *io-call-read-char'-corres*
thm *io-call-read-char'-def*
thm *l2-call-read-char'-def*

thm *io-call-read-char-loop'-corres*
thm *io-call-read-char-loop'-def*
thm *l2-call-read-char-loop'-def*

26.18.7 Disclaimer / Caution

As we have seen in the examples the refinement theorems can contain assumptions on the input pointers, in particular disjointness assumptions between different pointer parameters and also disjointness of pointer parameters to complete memory areas like *stack-free*. When a function is called these assumptions have to be discharged. When these pointer parameters are eliminated (replaced by in-out value parameters) the assumptions disappear. So when you work on the final output of autocorres you know nothing about pointers or disjointness.

In case a local pointer is eliminated these assumptions are be discharged locally as an intermediate step in the proof. So this use case is fine. However, if the function is called on a heap pointer these assumptions are propagated to the caller. So the assumptions move up the call stack and might end up as implicit hidden assumptions on your toplevel functions (API).

Rule of thumb: In case your toplevel API function has more then one pointer parameter don't specify any in-out parameters on that function to avoid implicit assumptions.

26.18.8 Implementation Aspects

The proof of the refinement theorem requires to keep track of various pointers and changes the signature of functions, including extending plain return values to tuples. We decided to make a rather explicit forward proof for this, to keep tight control of the various aspects. We make heavy use of ML antiquotations to match and build cterms. An obstacle here is that the state record *globals*. which holds the heap, is not yet defined but is only present as a locale. So our code is implemented within the locale *stack-heap-state*. We have introduced the concept of *interpretation data* (c.f. `../lib/ml-helpers/interpretation_data.ML`) to get hold of the interpretation we are interested in. The data is initialised in `../AutoCorres.thy` as declaration within *stack-heap-state*. In particular this allows us to match against and build heap lookup and heap update expressions.

The interface is via `In_Out_Parameters.operations`. This takes the morphism of the interpretation as an argument to derive the instance we are interested in. Note that we have crafted the main functions to benefit from the eager evaluation of ML. In particular `In_Out_Parameters.operations` takes the morphism *phi* as its only parameter and thus the instances of the main functions are only evaluated once.

end

26.18.9 Pointer-parameters as data

In a case where a pointer is not used to access the heap (e.g., *set-void*), the "modifies" part of the corres theorem should be empty. Otherwise func-

tions like *set-byte*, that pass "arbitrarily" computed pointers fail to get IO-lifted. We enforce this by declaring pointer parameters as "data" in the *in-out-parameters* option of autocorres.

Future work: Automate this analysis to infer which pointer-parameters are treated as pure data.

context *io-corres-set-void* **begin**

lemma *distinct-sets* ([]::*addr set list*)
— should be able to get rid of this assumption, too (for multiple parameters) $\Longrightarrow$
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines (*l2-set-void'* *p*) (*io-set-void'* *p*) *s t*
(*globals.rel-stack* $\mathcal{S}$ {} *A s* t_0
(*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
by (*rule io-set-void'-corres*)
end
context *io-corres-set-void2* **begin**
thm *io-set-void2'-corres*
end
context *io-corres-set-byte* **begin**
thm *io-set-byte'-corres*
end

26.18.10 Function pointers

The basics how function pointers are handled is decribed in `fnptr.thy`.

To support function pointers during the IO phase we need the "in-out specification" for the function pointer to build the corres proposition for the function pointer and to conduct the corres proof. The specification follows those for other functions with the peculiarity that the arguments of a function pointer are specified positionally (as we do not have names for the arguments).

The general format for a function pointer p is: `p(kind, kind, ...) [might_exit] [in_out_globals])`

- where kind is on of *data*, *in*, *out*, *in-out*, *keep* or *keep*∗. The default for a non-pointer type or a function pointer type is *data*. The default for a pointer parameter is *keep*∗. The distinction between *keep*∗ and *keep* allows us to fine tune the disjointness assumptions about the pointer parameters. Pointers specified with *in*, *out*, *in-out* and *keep* have to be disjoint. Pointers specified with *data* or *keep*∗ do not have to be disjoint. Note that the kind of a parameter can influence the type of the resulting function and thus in which program environment the abstracted function pointer is defined. Thus it can be the case that we need multiple environments depending on the target type of the abstraction.
- The default for *might-exit* is false

- The default for *in-out-globals* is false

  There are 3 use-cases of function pointers that are supported which are handled slightly differently:

1. Function pointer as explicit parameter of another function: The specification of the pointer parameters is given as part of *in-out-parameters* specification of the outer function: `outer_function(...) \<open>for\<close> p1(...) and p2(...) ... ;` where the specs for the pointer parameter are as described above.

2. Function pointer via global variable. You do not need an explicit specification here. It is inferred from the functions that might be referenced via the global variable. Note that all functions that might be target of the same function pointer must have the same in-out specification.

3. C-style object method calls. Here we have the section *method-in-out-fun-ptr-specs* in the autocorres parameters. You specify the function pointer by designating the structure field, `struct.field(...) ...` and giving a specification as described above. Note that currently all such method pointers with the same function signature must share the same in-out specification.

26.18.11 Future work / Open Ends

- support functions that might modify the heap typing (e.g. via *exec-concrete*) currently our *rel-alloc* / *rel-stack* setup enforces that heap typing does not change at all this is to restrictive. Before we had that heap typing does not change in $\mathcal{S}$ which was too weak to modify the A frame to handle invocations on heap pointers. I guess a good approximation would be to say that heap typing is unchanged on all relevant addresses: $\mathcal{S} \cup A \cup M$ Including M might be too restrictive, e.g. consider an alloc function that zeros some memory and adapts the heap-typing.

- Refined treatment of *in-out-globals*: allow the user to add a guard that is inserted in front of the abstract program. E.G. locale for procedure: *ptr-span* (*global-array* + *MAX-IDX*) $\subseteq \mathcal{G}$ + a Guard depending on a parameter *idx*, e.g. *idx* < *MAX-IDX*. Then we can automatically discharge local guards like *ptr-span* (*global-array* + *idx*) $\subseteq \mathcal{G}$. Not sure if this is useful though

- Cleanup unused *Generic-Data* that was replaced by interpretation data

- Remove struct-rewrite step from Heap Lifting. As we normalise pointers to root pointers in L2-Opt the pointers should already be in normal form and struct-rewrite does nothing.
- Add sanity checks to *in-out* specs, especially if parameter names actually occur in signature
- Remove redundant assumptions for keep parameters: *ptr-span* $y \cap$ *stack-free* (*hrs-htd* (*t-hrs-*$'$ s)) $= \{\} \Longrightarrow$ *globals.rel-alloc* $\mathcal{S}$ M A t_0 s $t \Longrightarrow$ *ptr-span* $y \subseteq M \Longrightarrow$ ...
  - The first one can be derived from the following ones
- Peephole: remove *ptr-disjoint* of singletons
- Peephole simplification, especially Guard True. Not really necessary, will disappear in hl anyways.
- Eliminate pointers to global variables: When a pointer to a global variable is only used as a *in-out parameter* it would be nice to abstract it to a record field in *lifted-globals* (or to something similar, i.e. lookup / update function in the heap with disjointnes and commutation of updates for other globals)
- Support more relaxed disjoint assumptions, especially *disjoint-or-eq*, e.g. swap also works if the input pointers are equal.
  - All 'keep' pointers are currently assumed disjoint and also are in the modifies set. We could be more relaxed here.
- HL phase has some explicit references to phase L2 instead of abstract *prev-phase* (probably related to function pointers)
- Would be nice if synthesize rules would also participate in thm, and add / del stuff like *named-rules*

end

26.19 Function Pointers

theory *fnptr*
imports *AutoCorres*
begin

install-C-file *fnptr.c*

ML ‹
val emp = Binaryset.empty fast-string-ord

val s1 = Binaryset.addList (emp, [even, odd])
val s1′ = Binaryset.listItems s1
val odd = Binaryset.addList (emp, [odd])
val even = Binaryset.addList (emp, [even])

val s3 = Binaryset.difference (odd, s1)
val s3′ = Binaryset.listItems s3

val s4 = Binaryset.difference (even, s1)
val s4′ = Binaryset.listItems s4

val s5 = Binaryset.difference (s1, odd)
val s5′ = Binaryset.listItems s5
›

declare [[*ML-print-depth=1000*]]
init−autocorres [*phase=L1*,
ts-force nondet = voidcaller,
ts-force-known-functions = option] *fnptr.c*

autocorres [*phase=L1*,*single-threaded*,
ts-force nondet = voidcaller,
ts-force-known-functions = option] *fnptr.c*

autocorres [*phase=L2*,*single-threaded*,
ts-force nondet = voidcaller,
ts-force-known-functions = option] *fnptr.c*

autocorres [*phase=HL*,
ts-force nondet = voidcaller,
ts-force-known-functions = option] *fnptr.c*

autocorres [*phase=TS*,
ts-force nondet = voidcaller,
ts-force-known-functions = option] *fnptr.c*

autocorres [
ts-force nondet = voidcaller,
ts-force-known-functions = option] *fnptr.c*

Dealing with function pointers in AutoCorres has the following main

challenges:

- Parameter passing
- Correspondence proofs
- Mutual recursion

term *from-bytes*

Parameter passing

In C a function pointer type does not have to specify the names of the function parameters. Hence when translating a call to a function pointer from C into SIMPL we need a uniform way to pass the function parameters. Recall that in SIMPL local variables are represented as part of the state, there is no lambda binding at that stage. For ordinary function calls we know the names of the parameters and can refer to them, for function pointers we do not know the names. Our approach is to generally switch to uniform names for the parameters, encoding the position and the type like: *in1-int*, *in2-unsigned*.

The mapping to the original names in the C file is implemented as syntactic sugar in the locale. As this mapping depends on the actual function we have to make sure to work in the correct context. The main interface to this mapping is defined in `../c-parser/HPInter.ML` e.g.:

- `HPInter.all_locvars`
- `HPInter.enter_scope`

  Note that the current implementation is a little bit fragile as it also depends on the syntax translations for lookup and update in the underlying **statespace**.

Correspondence proofs

When we come along a function pointer call in a correspondence proof we need a correspondence proof for the function pointer. Were to we get it from? In full generality we do not know much about were the function pointer might be pointing to. One drastic way out could be to add a guard at the call point, which guarantees that there is such a proof. This makes the proof trivial, but the code gets polluted with correspondence guards which also show up at the user level. So whenever one wants to proof a property about the resulting function one has to deal with this guard.

To keep the user level proofs clean, we decided to take another approach, by restricting the programs we can handle to some common use cases:

- Constant function pointer as parameter to a function. Here constant means that the parameter is not modified within the function body.
- Function pointer via constant global variable. Think of the global variable as a kind of dispatcher or class table, e.g. Array of structs representing "object descriptions" with function pointers as fields.
- C-style object method calls: Instead of a pure function pointer, a pointer to a structure which containes function pointers is passed as a parameter.

  In the first case we can add the correspondence assumption to the locale of the function. In the second case we can even resolve the correspondence proof as we can statically infer which functions might be called via the global variable.

Mutual recursion

The general approach to resolve a function pointer call is via a lookup in a program environment which maps the pointer to the definition of the function. In SIMPL the function is defined by a piece of syntax, and the program environment is an explicit context Γ which appears in the semantic rules for SIMPL. Moreover, for each function that might be called via a function pointer we introduce a pointer to that function. These pointers, and the assumption that all of them are distinct are put to the locale of global variables.

In each phase of Autocorres we introduce a similar program environment $\mathcal{P}$ that maps the pointer to a monadic HOL function definition. As from phase L2 on the parameters of functions become lambda abstractions we introduce a distinct program environment for each function type. These program environments are introduced as locale parameters. Then we successively add the implementation equations of the form $\mathcal{P}$ *some-function-pointer* = *some-function* as the functions are defined.

The correspondence theorems (or assumptions) for function pointer calls then relates the functions resolved via their respective environment, e.g. relate *the* (Γ *p*) with $\mathcal{P}$ *p*.

We also support mutual recursion via global function pointers (case 2 above), e.g. a function might call itself indirectly via a global function pointer variable. The global program analysis takes this into account to determine the strongly connected components aka. (recursive) cliques.

This adds another twist to the approach with program environments just described. The assumption $\mathcal{P}$ *some-function-pointer* = *some-recursive-function-via-*$\mathcal{P}$ can only be added after the function *some-recursive-function-via-*$\mathcal{P}$ is defined, but the definition already depends on the program environment. How do we cut the loop?

The core idea is to split the definition into two phases. First we define the **fixed-point** by explicitly extending the program environment at each call. Instead of calling $\mathcal{P}\ p$ directly we call *map-of-default* $\mathcal{P}$ [(*some-function-pointer*, *some-recursive-function-via-*$\mathcal{P}$)] p where *some-recursive-function-via-*$\mathcal{P}$ participates in the fixed point construction. Once the function is defined, we create a new locale, extending the program environment with the assumption $\mathcal{P}$ *some-function-pointer* = *some-recursive-function-via-*$\mathcal{P}$ and simplifying

map-of-default $\mathcal{P}$ [(*some-function-pointer*, *some-recursive-function-via-*$\mathcal{P}$)] $p = \mathcal{P}\ p$

in the function body. After this extra step we again uniformly represent recursive and non-recursive function pointer calls with $\mathcal{P}\ p$.

26.19.1 Global locales

The global locales fix the program environments (mapping function pointers to definitions) for a phase. These locales are created in the initialisation phase of **autocorres** before the individual functions are processed.

The fundamental locale is storing the addresses of global variables, including function pointers and some basic properties about them.

context *fnptr-global-addresses*
begin

For each function pointer a constant is declared e.g. *fnptr.odd-disp*, *fnptr.add*. Note that these constants have to be qualified by the program name. In case there would be a conflict with Isabelle internal names, e.g. a function ending with -- the name is suffixed with `Hoare.proc_deco`, cf. `NameGeneration.fun_ptr_name`.

ML ‹*Hoare.proc-deco*›

Moreover the global variables and their defining equations, i.e. initialisation expressions are collected in *odd-even-dispatcher* ≡ *fupdate* (*Suc 0*) (λ-. *fnptr.even-disp*) (*fupdate 0* (λ-. *fnptr.odd-disp*) (*ARRAY* -. *NULL*))

gi ≡ *0*

dispatcher-u ≡ *fupdate* (*Suc 0*) (*unop-u-C-update* (λ-. *fnptr.inc-u*)) (*fupdate* (*Suc 0*) (*binop-u-C-update* (λ-. *fnptr.add-gu*)) (*fupdate 0* (*unop-u-C-update* (λ-. *fnptr.inc-u*)) (*fupdate 0* (*binop-u-C-update* (λ-. *fnptr.add-u*)) (*ARRAY* -. *object-u-C NULL NULL*))))

dispatcher ≡ *fupdate 4* (*object-C.unop-C-update* (λ-. *NULL*)) (*fupdate 4* (*binop-C-update* (λ-. *NULL*)) (*fupdate 3* (*object-C.unop-C-update* (λ-. *fnptr.inc*)) (*fupdate 3* (*binop-C-update* (λ-. *fnptr.minus*)) (*fupdate 2* (*object-C.unop-C-update* (λ-. *fnptr.dec*)) (*fupdate 2* (*binop-C-update* (λ-. *fnptr.mul*)) (*fupdate* (*Suc 0*) (*object-C.unop-C-update* (λ-. *fnptr.dec*)) (*fupdate* (*Suc 0*) (*binop-C-update* (λ-. *fnptr.minus*)) (*fupdate 0* (*object-C.unop-C-update* (λ-. *fnptr.inc*)) (*fupdate 0* (*binop-C-update* (λ-. *fnptr.add*)) (*ARRAY* -. *object-C NULL NULL*)))))))))

add-u-p ≡ *fnptr.add-u*

add-p ≡ *fnptr.add*.

thm *global-const-defs*
thm *global-const-array-selectors* — more efficient access to array components
thm *global-const-non-array-selectors*
thm *global-const-selectors*

To express distinctness of function pointers we make use of the infrastructure in `$ISABELLE_HOME/src/HOL/Statespace/DistinctTreeProver.thy`, which is also the foundation for **statespace**. A tree data structure is used to efficiently derive distinctness and subset properties.

thm *fun-ptr-distinct*
thm *fun-ptr-subtree*

For every function pointer the locale assumes that they are not NULL, i.e. *c-fnptr-guard*. Based on the distinctness of the function pointers we derive some simplification rules for *map-of-default*.

thm *fun-ptr-simps*
end

Note that the program environment $\mathcal{P}$ for l1 programs is only declared as constant (without a definition). This environment is 'populated' in derived locales as the definitions of function become available.

A program environment for each distinct function pointer type is introduced for each further phase. e.g. $\mathcal{P}$*-l2-unsigned---unsigned*, $\mathcal{P}$*-l2-int---int*

context *fnptr-global-addresses*
begin

For example if the function pointer would index into the dispatcher array *dispatcher-u* and select a *binop-u-C*, it would result into subgoals for each of the possible pointers, namely and *fnptr.add-u fnptr.add-gu*.

term *dispatcher-u*

In this bundle we define some introduction rules to support the correspondence proofs. In phase L1 and L2 the correspondence proof is quite explicitly done in ML and the rules for function pointer calls are explicitly instantiated in ML. Here in layer HL the proof is done by applying intro rules and synthesising the abstract version of the program. The following rules relate the abstract and concrete program environments that are used for function pointer calls.

context includes *fnptr-hl-impl-corres*
begin
thm *fun-ptr-intros*
end

Note for example, that in the current goal state the concrete program would be of the shape $\mathcal{P}$*-l2----int p* whereas the abstract program would be a

plain schematic variable. This variable is then instantiated with $\mathcal{P}$*-hl----int p*. Once the rule is applied the identity marker *DYN-CALL* triggers the side-condition splitter / solver: `AutoCorresUtil.dyn_call_split_simp_sidecondition_tac`, which splits the generic correspondence of some generic function pointer to its possible values.

For example if the function pointer would index into the dispatcher array *dispatcher-u* and select a *binop-u-C*, it would result into subgoals for each of the possible pointers, namely *fnptr.add-u* and *fnptr.add-gu*.

context includes *fnptr-wa-impl-corres*
begin

Like in phase HL we also have some custom intro rules to support the instantiation of function pointer calls.

thm *fun-ptr-intros*
thm *global-const-defs*
thm *fun-ptr-simps*
end

The TS phase adds another dimension to the program environments, the monad type. E.g. $\mathcal{P}$*-pure-unsigned---unsigned*, $\mathcal{P}$*-gets-unsigned---unsigned*, $\mathcal{P}$*-option-unsigned---unsigned*, $\mathcal{P}$*-nondet-unsigned---unsigned*, $\mathcal{P}$*-exit-unsigned---unsigned*.

context includes *fnptr-ts-impl-corres*
begin

Again we have custom intro rules to support instantiation of the abstract program environment.

thm *fun-ptr-intros*
thm *global-const-defs*
thm *fun-ptr-simps*
end
end

26.19.2 Function / Recursive-clique specific locales

L1

context *l1-definition-add* — holds the definition of the function
begin
thm *l1-def* — all relevant l1 definitions
thm *ac-def* — definitions of all layers

Note that there are two variants of the definition of *l1-add′*. First the original definition and then an optimised version, already removing some exception handling. Here are the names of the theorems:

thm *l1-add′-def*
thm *l1-opt-add′-def*
end

context *l1-impl-add*
begin

As *l1-add′* might also be called indirectly via a function pointer, we populate the program environment with the definition

thm *l1-add′-impl* — Mapping of the function-pointer to the definition in the corresponding environment.

The equation is also added to *gets-the* (*ogets* (λ-. *?f*)) = *return ?f*
gets-the (*ogets ?f*) = *gets ?f*
c-fnptr-guard fnptr.f
c-fnptr-guard fnptr.add
c-fnptr-guard fnptr.dec
c-fnptr-guard fnptr.inc
c-fnptr-guard fnptr.mul
c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.odd-disp* = *?f1.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.even-disp* = *?f2.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.intcallable2* = *?d fnptr.intcallable2*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.callable1* = *?d fnptr.callable1*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-gu* = *?d fnptr.add-gu*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.minus* = *?d fnptr.minus*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-u* = *?d fnptr.add-u*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc* = *?d fnptr.inc*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.mul* = *?d fnptr.mul*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add* = *?d fnptr.add*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.f*

$=$ *?d fnptr.f*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.dec*
$=$ *?d fnptr.dec*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc-u*
$=$ *?d fnptr.inc-u*
$\mathcal{P}$ *fnptr.add* $=$ *l1-add′*.

thm *fun-ptr-simps*
thm *l1-def*
end

context *l1-definition-call-binop*
begin

l1-call-binop′ makes a function pointer call via the *dispatcher* array selecting field *binop-C*. So it might call *l1-add′*, *l1-minus′* or *l1-mul′*. The definitions of these functions are also imported into the locale (cf. *l1-add′* $\equiv$ *L1-catch* (*L1-seq* (*L1-init* (λ*upd*. *add.ret′*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-modify* *add.k*:=$_{\mathcal{L}}$ (λ-. *gi*)) (*L1-seq* (*L1-guard* (λ*s*. $-$ *2147483648* $\leq$ *sint* (*s* $\cdot_{\mathcal{L}}$ *add.n*) $+$ *sint* (*s* $\cdot_{\mathcal{L}}$ *add.m*) $\wedge$ *sint* (*s* $\cdot_{\mathcal{L}}$ *add.n*) $+$ *sint* (*s* $\cdot_{\mathcal{L}}$ *add.m*) $\leq$ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*add.ret′* :=$_{\mathcal{L}}$ λ-. (*s* $\cdot_{\mathcal{L}}$ *add.n*) $+$ (*s* $\cdot_{\mathcal{L}}$ *add.m*)⟩)) (*L1-seq* (*L1-modify* (*global-exn-var′-′-update* (λ-. *Return*))) *L1-throw*))))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var′-′ s*)) *L1-skip L1-throw*)

l1-add′ $\equiv$ *L1-seq* (*L1-init* (λ*upd*. *add.ret′*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-modify* *add.k*:=$_{\mathcal{L}}$ (λ-. *gi*)) (*L1-seq* (*L1-guard* (λ*s*. $-$ *2147483648* $\leq$ *sint* (*s* $\cdot_{\mathcal{L}}$ *add.n*) $+$ *sint* (*s* $\cdot_{\mathcal{L}}$ *add.m*) $\wedge$ *sint* (*s* $\cdot_{\mathcal{L}}$ *add.n*) $+$ *sint* (*s* $\cdot_{\mathcal{L}}$ *add.m*) $\leq$ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*add.ret′* :=$_{\mathcal{L}}$ λ-. (*s* $\cdot_{\mathcal{L}}$ *add.n*) $+$ (*s* $\cdot_{\mathcal{L}}$ *add.m*)⟩)) (*L1-modify* (*global-exn-var′-′-update* (λ-. *Return*))))))

l1-mul′ $\equiv$ *L1-catch* (*L1-seq* (*L1-init* (λ*upd*. *mul.ret′*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-guard* (λ*s*. $-$ *2147483648* $\leq$ *sint* (*s* $\cdot_{\mathcal{L}}$ *mul.n*) $*$ *sint* (*s* $\cdot_{\mathcal{L}}$ *mul.m*) $\wedge$ *sint* (*s* $\cdot_{\mathcal{L}}$ *mul.n*) $*$ *sint* (*s* $\cdot_{\mathcal{L}}$ *mul.m*) $\leq$ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*mul.ret′* :=$_{\mathcal{L}}$ λ-. (*s* $\cdot_{\mathcal{L}}$ *mul.n*) $*$ (*s* $\cdot_{\mathcal{L}}$ *mul.m*)⟩)) (*L1-seq* (*L1-modify* (*global-exn-var′-′-update* (λ-. *Return*))) *L1-throw*)))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var′-′ s*)) *L1-skip L1-throw*)

l1-mul′ $\equiv$ *L1-seq* (*L1-init* (λ*upd*. *mul.ret′*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-guard* (λ*s*. $-$ *2147483648* $\leq$ *sint* (*s* $\cdot_{\mathcal{L}}$ *mul.n*) $*$ *sint* (*s* $\cdot_{\mathcal{L}}$ *mul.m*) $\wedge$ *sint* (*s* $\cdot_{\mathcal{L}}$ *mul.n*) $*$ *sint* (*s* $\cdot_{\mathcal{L}}$ *mul.m*) $\leq$ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*mul.ret′* :=$_{\mathcal{L}}$ λ-. (*s* $\cdot_{\mathcal{L}}$ *mul.n*) $*$ (*s* $\cdot_{\mathcal{L}}$ *mul.m*)⟩)) (*L1-modify* (*global-exn-var′-′-update* (λ-. *Return*)))))

l1-minus′ $\equiv$ *L1-catch* (*L1-seq* (*L1-init* (λ*upd*. *ret′*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-guard* (λ*s*. $-$ *2147483648* $\leq$ *sint* (*s* $\cdot_{\mathcal{L}}$ *n*) $-$ *sint* (*s* $\cdot_{\mathcal{L}}$ *m*) $\wedge$ *sint* (*s* $\cdot_{\mathcal{L}}$ *n*) $-$ *sint* (*s* $\cdot_{\mathcal{L}}$ *m*) $\leq$ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*ret′* :=$_{\mathcal{L}}$ λ-. (*s* $\cdot_{\mathcal{L}}$ *n*) $-$ (*s* $\cdot_{\mathcal{L}}$ *m*)⟩)) (*L1-seq* (*L1-modify* (*global-exn-var′-′-update* (λ-. *Return*))) *L1-throw*)))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var′-′ s*)) *L1-skip L1-throw*)

l1-minus′ $\equiv$ *L1-seq* (*L1-init* (λ*upd*. *ret′*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-guard* (λ*s*.

$-$ *2147483648* $\leq$ *sint* (*s* $\cdot_{\mathcal{L}}$ *n*) $-$ *sint* (*s* $\cdot_{\mathcal{L}}$ *m*) $\wedge$ *sint* (*s* $\cdot_{\mathcal{L}}$ *n*) $-$ *sint* (*s* $\cdot_{\mathcal{L}}$ *m*) $\leq$ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*ret*$'$:=$_{\mathcal{L}}$ λ-. (*s* $\cdot_{\mathcal{L}}$ *n*) $-$ (*s* $\cdot_{\mathcal{L}}$ *m*)⟩)) (*L1-modify* (*global-exn-var*$'$-$'$*-update* (λ-. *Return*)))))

l1-call-binop$'$ $\equiv$ *L1-catch* (*L1-seq* (*L1-init* (λ*upd*. *call-binop.ret*$'$:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-modify call-binop.r*:=$_{\mathcal{L}}$ (λ-. *0*)) (*L1-seq* (*L1-guard* (λ*s*. *UCAST*(*8* $\rightarrow$ *32 signed*) (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*) <*s* *5*)) (*L1-seq* (*L1-guard* (λ*s*. *0* $\leq$*s* *UCAST*(*8* $\rightarrow$ *32 signed*) (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*))) (*L1-seq* (*L1-condition* (λ*s*. *PTR-COERCE*(*unit* $\rightarrow$ *unit*) (*binop-C* (*dispatcher.*[*unat* (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*)])) $\neq$ *NULL*) (*L1-guarded* (λ*s*. *0* $\leq$*s* *UCAST*(*8* $\rightarrow$ *32 signed*) (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*) $\wedge$ *UCAST*(*8* $\rightarrow$ *32 signed*) (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*) <*s* *5* $\wedge$ *c-fnptr-guard* (*binop-C* (*dispatcher.*[*unat* (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*)]))) (*do* {
p $\leftarrow$ *gets* (λ*s*. *binop-C* (*dispatcher.*[*unat* (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*)]));
L1-call (λ*s*. *locals-update* (*cupdate* (*Suc 0*) (λ-. *s* $\cdot_{\mathcal{L}}$ *call-binop.m*)) (*locals-update* (*cupdate 0* (λ-. *s* $\cdot_{\mathcal{L}}$ *call-binop.n*)) *s*)) (*P* *p*) (λ*s t*. *s*(|*globals* := *globals t*|)) (λ*s t*. *s*(|*global-exn-var*$'$-$'$:= *Nonlocal* (*the-Nonlocal* (*global-exn-var*$'$-$'$ *t*))|)) (λ*uu*-. *call-binop.r*:=$_{\mathcal{L}}$ (λ-. *clookup 2* (*locals uu*-)))
})) *L1-skip*) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*call-binop.ret*$'$:=$_{\mathcal{L}}$ λ-. *s* $\cdot_{\mathcal{L}}$ *call-binop.r*⟩)) (*L1-seq* (*L1-modify* (*global-exn-var*$'$-$'$*-update* (λ-. *Return*))) *L1-throw*)))))))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var*$'$-$'$ *s*)) *L1-skip* *L1-throw*)

l1-call-binop$'$ $\equiv$ *L1-seq* (*L1-init* (λ*upd*. *call-binop.ret*$'$:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-modify call-binop.r*:=$_{\mathcal{L}}$ (λ-. *0*)) (*L1-seq* (*L1-guard* (λ*s*. *UCAST*(*8* $\rightarrow$ *32 signed*) (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*) <*s* *5*)) (*L1-seq* (*L1-guard* (λ*s*. *0* $\leq$*s* *UCAST*(*8* $\rightarrow$ *32 signed*) (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*))) (*L1-catch* (*L1-seq* (*L1-condition* (λ*s*. *PTR-COERCE*(*unit* $\rightarrow$ *unit*) (*binop-C* (*dispatcher.*[*unat* (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*)])) $\neq$ *NULL*) (*L1-guarded* (λ*s*. *0* $\leq$*s* *UCAST*(*8* $\rightarrow$ *32 signed*) (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*) $\wedge$ *UCAST*(*8* $\rightarrow$ *32 signed*) (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*) <*s* *5* $\wedge$ *c-fnptr-guard* (*binop-C* (*dispatcher.*[*unat* (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*)]))) (*do* {
p $\leftarrow$ *gets* (λ*s*. *binop-C* (*dispatcher.*[*unat* (*s* $\cdot_{\mathcal{L}}$ *call-binop.i*)]));
L1-call (λ*s*. *locals-update* (*cupdate* (*Suc 0*) (λ-. *s* $\cdot_{\mathcal{L}}$ *call-binop.m*)) (*locals-update* (*cupdate 0* (λ-. *s* $\cdot_{\mathcal{L}}$ *call-binop.n*)) *s*)) (*P* *p*) (λ*s t*. *s*(|*globals* := *globals t*|)) (λ*s t*. *s*(|*global-exn-var*$'$-$'$:= *Nonlocal* (*the-Nonlocal* (*global-exn-var*$'$-$'$ *t*))|)) (λ*uu*-. *call-binop.r*:=$_{\mathcal{L}}$ (λ-. *clookup 2* (*locals uu*-)))
})) *L1-skip*) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*call-binop.ret*$'$:=$_{\mathcal{L}}$ λ-. *s* $\cdot_{\mathcal{L}}$ *call-binop.r*⟩)) (*L1-seq* (*L1-modify* (*global-exn-var*$'$-$'$*-update* (λ-. *Return*))) *L1-throw*))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var*$'$-$'$ *s*)) *L1-skip* *L1-throw*))))))), as well as the corresponding entries in the program environment (cf. *gets-the* (*ogets* (λ-. *?f*)) = *return ?f*

gets-the (*ogets ?f*) = *gets ?f*

c-fnptr-guard fnptr.f

c-fnptr-guard fnptr.add

c-fnptr-guard fnptr.dec

c-fnptr-guard fnptr.inc

c-fnptr-guard fnptr.mul

c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.odd-disp* = *?f1.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.even-disp* = *?f2.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.intcallable2* = *?d fnptr.intcallable2*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.callable1* = *?d fnptr.callable1*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-gu* = *?d fnptr.add-gu*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.minus* = *?d fnptr.minus*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-u* = *?d fnptr.add-u*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc* = *?d fnptr.inc*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.mul* = *?d fnptr.mul*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add* = *?d fnptr.add*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.f* = *?d fnptr.f*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.dec* = *?d fnptr.dec*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc-u* = *?d fnptr.inc-u*
$\mathcal{P}$ *fnptr.add* = *l1-add'*
$\mathcal{P}$ *fnptr.mul* = *l1-mul'*
$\mathcal{P}$ *fnptr.minus* = *l1-minus'*). Also note that the function pointer call is wrapped in a *L1-guarded*. In particular the guard ensures that the index into the array is in range. It is essential to have this information available in the correspondence proof to split the function pointer call to its potential targets.

thm *l1-call-binop'-def*
thm *global-const-defs*
thm *l1-def*

thm *fun-ptr-simps*
end

context *l1-corres-call-binop*
begin

The correspondence proofs are performed within the corres locales. The results are added to *L1corres True* Γ *l1-add′* (*Call fnptr.add*)
L1corres True Γ *l1-mul′* (*Call fnptr.mul*)
L1corres True Γ *l1-minus′* (*Call fnptr.minus*)
L1corres True Γ *l1-call-binop′* (*Call fnptr.call-binop*). Note that the correspondence proofs of the potential callees are present. **autocorres** resolves call dependencies and first performs the correspondence proofs of the potential callees.

thm *l1-call-binop′-corres*
thm *l1-corres*
thm *l1-def*
end

context *l1-definition-parameter-call*
begin

This function performs a function pointer call on a function parameter. Recall that we only support the case were the function pointer parameter value stays the same within the function. The phase L1 is special, as we postpone the correspondence proof of the function pointer call to the L2 layer. The reason is that in the L2 layer we replace the parameters with lambda abstraction and thus the fact that the value does not change becomes trivial. In L1 we would have to propagate this information trough every state update, from the invocation of the function to the function pointer call. This proof is anyway performed in the L2 phase. Postponing the correspondence proof is achieved by just assuming the correspondence in *L1-guarded*.

thm *l1-parameter-call′-def*
thm *l1-def*
end

context *l1-corres-parameter-call*
begin
thm *l1-parameter-call′-corres*
thm *l1-corres*
end

L2

context *l2-definition-add*
begin
thm *fun-ptr-simps*

end

context *l2-impl-add*
begin
thm *fun-ptr-simps*
end

context *l2-definition-call-binop*
begin
thm *fun-ptr-simps*
thm *l2-call-binop'-def*
thm *l2-def*
end

context *l2-corres-call-binop*
begin
thm *l2-call-binop'-corres*
thm *l2-corres*
thm *l2-def*
end

context *l2-definition-parameter-call*
begin

Note that in contrast to L1 we do not have any correspondence assumption in *L2-guarded*. The function pointer parameter is just inserted in the function pointer call and thus is immediately the same value.

thm *l2-parameter-call'-def*
thm *l2-def*
end

context *l2-corres-parameter-call*
begin

In contrast to the function pointer call via a global variable, where we statically resolve which functions are potentially called, we add an explicit assumption on the parameter to the correspondence theorem. As in phase L2 we also have to proof the postponed L1 correspondence we actually have both assumptions here.

thm *l2-parameter-call'-corres*
thm *l2-corres*
thm *l2-def*
end

When defining (mutual) recursion indirectly via function pointers, there is a subtlety in the definition of the functions. In order to have an admissible *L2corres* property the program environment for the current clique is temporarily explicitly exended via *map-of-default* in the function bod-

ies. After definition of the function and extending the hypothetical program environment with the new definitions this construction is hidden again.

Compare the variation of in *l2-corres-odd-disp-even-disp* vs. *l2-impl-odd-disp-even-disp* and the extended program environment in *gets-the* (*ogets* (λ-. *?f*)) = *return ?f*

gets-the (*ogets ?f*) = *gets ?f*.

Also recall the general flow of how to arrive at a new level with definition and correspondence proof.

- First the induction step of the correspondence proof is performed, assuming that the recursive calls are in the correspondence relation.
- If that proof is successful the function body or bodies are used to do the actual definition(s) of the function(s).
- After the definition is done the actual correspondence proof is performed using the induction step as major ingredient, replacing the body / bodies with the new definition(s).
- The program environments are extended with the assumptions that the pointers are mapped to the new definitions and the right hand sides of the definitions are simplified.

context *l2-corres-odd-disp-even-disp*
begin
thm *fun-ptr-simps*
thm *l2-corres*
thm *l2-def*
thm *l2-odd-disp′-def* — Foundational definition of **fixed-point**. FIXME: should we conceale this?
thm *l2-odd-disp′.simps*
end

context *l2-impl-odd-disp-even-disp*
begin
thm *fun-ptr-simps*
thm *l2-corres*
thm *l2-def*
thm *l2-odd-disp′.simps*
thm *l2-impl-odd-disp′-def* — canonical variant. FIXME: should we rename this (simps?)?
end

HL

context *hl-definition-call-binop*
begin
thm *fun-ptr-simps*

thm *hl-call-binop′-def*
thm *hl-def*
end

context *hl-corres-call-binop*
begin
thm *hl-call-binop′-corres*
thm *hl-corres*
thm *hl-def*
end

context *hl-definition-parameter-call*
begin
thm *hl-parameter-call′-def*
thm *hl-def*
end

context *hl-corres-parameter-call*
begin
thm *hl-parameter-call′-corres*
thm *hl-corres*
thm *hl-def*
end

WA

context *wa-definition-call-binop*
begin
thm *fun-ptr-simps*
thm *wa-call-binop′-def*
thm *wa-def*
end

context *wa-corres-call-binop*
begin
thm *wa-call-binop′-corres*
thm *wa-corres*
thm *wa-def*
end

context *wa-definition-parameter-call*
begin
thm *wa-parameter-call′-def*
thm *wa-def*
end

context *wa-corres-parameter-call*
begin
thm *wa-parameter-call′-corres*

thm *wa-corres*
thm *wa-def*
end

TS

context *ts-impl-add*
begin

add′ is defined within the option monad. Note that (lifted version) also end up in the more expressive program environments (cf. *gets-the* (*ogets* (λ-. *?f*)) = *return ?f*

gets-the (*ogets ?f*) = *gets ?f*
c-fnptr-guard fnptr.f
c-fnptr-guard fnptr.add
c-fnptr-guard fnptr.dec
c-fnptr-guard fnptr.inc
c-fnptr-guard fnptr.mul
c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.odd-disp* = *?f1.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.even-disp* = *?f2.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.intcallable2* = *?d fnptr.intcallable2*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.callable1* = *?d fnptr.callable1*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-gu* = *?d fnptr.add-gu*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.minus* = *?d fnptr.minus*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-u* = *?d fnptr.add-u*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc* = *?d fnptr.inc*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.mul* = *?d fnptr.mul*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add*

$= ?d\ fnptr.add$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.f$
$= ?d\ fnptr.f$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.dec$
$= ?d\ fnptr.dec$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.inc\text{-}u$
$= ?d\ fnptr.inc\text{-}u$
$\quad \mathcal{P}\text{-}option\text{-}int'int\text{---}int\ fnptr.add = add'$
$\quad \mathcal{P}\text{-}nondet\text{-}int'int\text{---}int\ fnptr.add = (\lambda x1\ x2.\ gets\text{-}the\ (add'\ x1\ x2))$
$\quad \mathcal{P}\text{-}exit\text{-}int'int\text{---}int\ fnptr.add = (\lambda x1\ x2.\ liftE\ (gets\text{-}the\ (add'\ x1\ x2)))).$

So function pointer calls to this function can be performed in any of those monads.

thm *add′-def*
thm *fun-ptr-simps*
thm *ts-def*
end

context *ts-definition-call-binop*
begin
thm *fun-ptr-simps*
thm *call-binop′-def*
thm *ts-def*
end

context *ts-corres-call-binop*
begin
thm *call-binop′-corres*
thm *ts-corres*
thm *ts-def*
end

context *ts-definition-fac*
begin
thm *fun-ptr-simps*
thm *fac′.simps*
thm *ts-def*
end

context *ts-corres-fac*
begin
thm *fac′-corres*
thm *ts-corres*
thm *ts-def*
end

Recall that in case of a function pointer call via a parameter we assume correspondence of the parameter. So in which monad should we end up? Which monad should we assume for the parameter? Currently we assume the

same monad as for the function. We consecutively try the correspondence from the most restrictive to the most expressive monad and we end up in the monad with the first successful proof. If you want a different (more expressive) monad you can configure this via the autocorres option *ts-force*.

context *ts-definition-parameter-call*
begin
thm *parameter-call′-def*
thm *wa-def*
end

context *ts-corres-parameter-call*
begin
thm *parameter-call′-corres*
thm *wa-corres*
thm *wa-def*
end

context *ts-impl-call-binop*
begin
thm *ts-def*
end

Final AutoCorres

Once all phases are passed and the correspondence proofs between the consecutive layers are ready they are combined to the final correspondence layer, from SIMPL upto TS.

context *corres-parameter-call*
begin
thm *parameter-call′-ac-corres*
end

context *corres-call-binop*
begin
thm *call-binop′-ac-corres*
end

context *corres-odd-disp-even-disp*
begin
thm *odd-disp′-ac-corres*
thm *even-disp′-ac-corres*
end

When all functions of a C file are translated the following summary locales are introduced.

context *fnptr-all-impl* — All implementation locales of phase TS
begin
thm *fun-ptr-simps*

thm *even-disp'.simps*
thm *impl-even-disp'-def*
thm *ac-def*
end

context *fnptr-all-corres* — All corres locales of all phases
begin
thm *l1-corres*
thm *l2-corres*
thm *hl-corres*
thm *wa-corres*
thm *ts-corres*
thm *ac-corres*

thm *l1-def*
thm *l2-def*
thm *l2-even-disp'-def*
thm *l2-even-disp'.simps*
thm *hl-def*
thm *wa-def*
thm *ts-def*
thm *ac-def*

thm *global-const-defs*
thm *fun-ptr-simps*
thm *fun-ptr-intros*
thm *fun-ptr-distinct*
thm *fun-ptr-subtree*
end

Method calls of C-style objects

The goal is to also support function pointer invocations via C-style object methods: Instead of a plain function pointer a function gets a pointer to a structure (object) that contains function pointers as structure fields. This adds an important twist to the refinement proofs of autocorres. When we invoke a method via `p->method(...)` we actually calculate the function pointer from the heap object referenced by pointer `p`. As with the cases described before at that point we have to argue that the referenced function fulfills the 'corres' predicate. However, now we have the indirection via the heap. So somehow we have to keep track of the function pointers stored in the heap. As other pointer objects might also be stored and manipulated in the heap this typically requires user-level reasoning about the heap layout and disjointness of certain pointers. As autocorres is mainly used as a preprocessing tool for arbitrary C programs we want to avoid making too restrictive assumptions here. Instead we postpone the argument to the user. The core idea is that we want to argue that the actually referenced

function is a 'known function' for which we know or can assume that the 'corres' predicate holds. We introduce a predicate *known-function* into our locale hierarchy, which takes a function pointer p and tells us if this is a known function for which we can assume 'corres'. On every invocation of a object method we introduce a guard *known-function* (*method p*) into the code. Autocorres can then utilise this guard together with the assumption that the correspondence holds for all known functions. It is then left as an exercise for the user to discharge the guards. This boils down to an invariant argument that every time you assign something to *method p* it has to be a *known-function*. To support this, everytime a function is defined which might be such a function pointer, we utilise the implementation locales to store the information that this is a *known-function*.

Here are the locales about the *known-function* which state correspondence. These are imported as parents in all individual corres locales.

context *fnptr-l1-corres*
begin
thm *known-function-corres*
end

context *fnptr-l2-corres*
begin
thm *known-function-corres*
end

context *fnptr-hl-corres*
begin
thm *known-function-corres*
end

The locales for phases WA and TS are special as they might change the signature of the resulting function depending on the called functions. We make assumptions about called methods to derive the definition of the calling function. E.g. for phase WA we have to know whether a called method will do signed or unsigned word abstractions. For phase TS we have to know in which monad the called methods are defined in order to select the right monad for the caller. We can use the autocorres options `ts_force_known_functions`, `unsigned_word_abs_known_functions` and `no_signed_word_abs_known_functions` analogous to the corresponding options for individual functions.

In our example program we have chosen the option monad as a result monad. Note that autocorres does some sanity checks to ensure that e.g. the chosen monad is expressible enough, but ultimately it is the responsibility of the user to ensure that the monad and the word abstraction options match all potential instances of the methods.

context *fnptr-wa-corres*

begin
thm *known-function-corres*
end

context *fnptr-ts-corres*
begin
thm *known-function-corres*
end

context *ts-definition-call-object-binop-return*
begin
thm *ts-def*
end

context *ts-definition-call-object-binop-assign*
begin
thm *ts-def*
end

context *ts-definition-call-object-binop-emb*
begin
thm *ts-def*
end

Here you can see all known functions that autocorres has detected. Note that for this analysis autocorres assumes that it has total knowledge of the complete program. It collects all functions for which addresses are calculated in the program and considers them as *known-function*.

context *fnptr-all-impl*
begin
thm *known-function*
thm *fun-ptr-simps*
thm *fun-ptr-distinct*
end

context *fnptr-all-corres*
begin

thm *global-const-defs*
thm *fun-ptr-simps*
thm *fun-ptr-intros*
thm *fun-ptr-distinct*
thm *fun-ptr-subtree*
thm *known-function*
thm *known-function-corres*
end

end

26.20 Unions

theory *union-ac*
imports
AutoCorres

begin

26.20.1 Union support in C-Parser and Autocorres

Fully fledged C unions can have a lot of semantic subtleties in particular with respect to undefined behavior when it comes to different padding in the variants, effective types and accessing the union via different variants, e.g. writing a pointer through one variant and subsequently reading or writing the same union via another variant. See for example https://robbertkrebbers.nl/research/thesis.pdf in subsections 2.5.5 ff. for a discussion of some fine points.

Our current solution is very pragmatic by only supporting a subset of unions that does not exhibit the more "weired" behaviours:

- Single variant unions
- Multi-variant unions where each variant is packed (no implicit padding) and has the same size.

  For these limited unions the semantics is deterministic and basically boils down to casting / coercing different C-types. Note that every *c-type* is equipped with conversions *from-bytes* and *to-bytes* which allows us to take a value, convert it to a byte list and then make another value (potentially with a different type) from that byte list again. So the basic machinery is already there.

  The C parser analyses which variants of a union are actually actively used in the code you want to verify. Only those variants (internally referred to as active variants) are taken into account. If there is only a single variant we are fine and a single record type is constructed in HOL as the C-type representing that type. It is the same construction as for a structure type of that variant.

  In case there are multiple active variants the C parser checks that all those variants have the same size and are *packed* which means that there is no padding. The original C code might have to be rewritten with explicit padding fields to fulfill this requirement.

  We create a record type for each variant of the union, like a 'struct' for each variant. One variant is considered as the *canonical* variant. Currently this always is the first variant of the union. The idea of the canonical variant is that every access or update to a union is always

done via this canonical variant and then 'casted' to the variant you need. This casting is implemented via *PTR-COERCE*($'a \to 'b$) for pointers and *COERCE*($'a \to 'b$) and *coerce-map* for values.

Example:

```
typedef struct {
  unsigned fst;
  unsigned snd;
 } unsigned_pair;

typedef struct {
  signed fst;
  signed snd;
 } signed_pair;



typedef union {
  unsigned_pair u;
  signed_pair s;
} open_union;
```

The union *open-union* has two variants. The first one u is considered to be the canonical one. So when you want to update a variant s in the heap via a pointer to *open-union* we first read the canonical variant u from the heap then coerce the value to a variant s, then the update on the value is performed then the resulting value is coerced back to variant u and finally written back to the heap.

A good mental model for this implementation is that accesses to unions are normalised towards the canonical variants of the root pointers. This also carries over to the split heap model you obtain when you perform the HL phase of autocorres. There is only one relevant heap for the union, which is the heap for the canonical variant. This means that you can also define *addressable-fields* for the canonical variant. Currently you cannot define *addressable-fields* for the other variants.

include-C-file *union.h* **for** *union.c*
install-C-file *union.c*

autocorres [*addressable-fields* = *open-union.u*]*union.c*

context *union-all-corres*
begin
thm *ts-def*
thm *l2-def*

Names

The union *open-union* has two variants *u* and *s*. The first one, *u*, is considered to be the canonical one. The corresponding c-type becomes "THE" type of union and is named *open-union-C*. The other variant *s* gets the qualified name *open-union-C′s-C*.

term *open-union-C*
term *u-C*
thm *u-C-def*
term *u-C-update*
thm *u-C-update-def*

term *open-union-C′s-C*
term *s-C*
thm *s-C-def*
term *s-C-update*
thm *s-C-update-def*

term *heap-open-union-C*
thm *heap-open-union-C-def*

Anonymous Structures and Union

For anonymous structures and types we create names by considering the surrounding typedef and structure / union. For example see the typedef of *my-union*. The canonical variant for the union gets *my-union-C*. The variant *f* is an anonymous structure. The type of the anonymous structure is type *my-union-C′f-C′struct*, whereas the type of the variant *f* is *my-union-C′f-C*. Analogously there is also a suffix *union* in case of an anoymous union. Note the naming convention: For nested names note that sufffix *union* or *struct* is for the nested union or structure itself, in contrast to the name without suffix which denotes the variant of the enclosing union.

typ *my-union-C*
term *my-union-C*

typ *my-union-C′f-C′struct*
term *f1-C*
thm *f1-C-def*

typ *my-union-C′f-C*

term *f-C*
thm *f-C-def*

typ *my-union-C'g-C'union*
term *x1-C*
thm *x1-C-def*

typ *my-union-C'g-C*
term *g-C*
thm *g-C-def*

Lemmas to coerce values / heap accesses

thm *coerce-id*
thm *coerce-cancel-packed*
thm *coerce-map-id*
thm *coerce-coerce-map-cancel-packed*

thm *h-val-coerce-ptr-coerce-packed*
thm *h-val-field-ptr-coerce-from-bytes-packed*
thm *heap-update-field-ptr-coerce-from-bytes-packed*
thm *heap-state.L2-modify-heap-update-ptr-coerce-packed-conv*
thm *heap-state.L2-modify-heap-update-ptr-coerce-packed-field-root-conv*
thm *L2-modify-heap-update-field-root-conv*

end

end

26.21 Pointers into Structures in Split Heap

theory *open-struct*
imports *AutoCorres*
begin

26.21.1 Overview

Heap lifting in AutoCorres transforms the monolithic byte-level heap *heap-mem* with explicit term level heap type description *heap-typ-desc* and typing constraints, like *c-guard*, *h-t-valid* and $d \models_t p$ to a more abstract split heap model with implicit HOL-typing. The major advantage of the split heap model is that, updates to one particular heap do not affect all other distinct heaps, removing the obligation to explicitly take care about aliasing. Aliasing between different heaps is ruled out by the model already.

The question that arises is on which granularity do we split the byte-level heap? The original AutoCorres splitted at the granularity of types. Each

distinct c-type is mapped into a separate heap. In that model you can still have intra-type aliasing (i.e. pointers of the same type can alias), but no inter-type aliasing (i.e. pointers of different type are distinct, and cannot alias). As an example a pointer to a list structure refers to a different (split) heap as a pointer to a tree structure. So when a function modifies lists, no pointer to a tree is affected.

Of course not every C-program can be represented in this split-heap model. In those cases the abstraction fails. In particular think of a low-level memory allocator, that allocates some raw-bytes and delivers a void-pointer, that is then retyped to point to the actual structure. To be able to combine those byte-level functions with high-level typed functions (in the split heap) one can switch between both layers of abstraction by *exec-concrete* and *exec-abstract*. The good thing is that this gives us a very expressive tool to switch between byte-level and typed view. We can do the specification and proofs of different parts of the program on the adequate level and still combine the results. But switching between the layers can be tedious and thus should be limited to the low-level functions that really inherently need byte-level reasoning.

Unfortunately the type granularity of the split heaps also rules out programs dealing with pointers into a structure. For example a structure containing a *int* field *count* cannot pass a *int* pointer to that field *count* to a another function, that might update it. The complete structure has a separate (split) heap that is distinct from the heap containing *int*s. This would mean that we have to resort to the byte-level heap to reason about those functions.

We extended the split heap model to support this common use-case. The core idea is that in addition to split heaps on the granularity of types you can also specify split heaps on the granularity of structure fields. Conceptually you 'open' the structure into its components (aka. fields) and each field gets a separate heap. Additionally you can specify which fields should be *addressable* and thus shared. In the example you can specify that *count* is addressable, so it is represented within the same heap as all other *int*. This means that an *int* pointer can now point to a plain *int* or to a *count* field within a structure. You explicitly allow this limited inter-type aliasing on *int* pointers. All other fields that are not explicitly marked as *addressable* still have their own heap and thus cannot alias with another pointer of the same field type. Note that mainly for performance reasons we actually avoid to have a separate heap for each single field but try to minimise the number of heaps by clustering the fields that are treated the same. Some more details on this construction are below.

26.21.2 Example Program and some Intuition

We illustrate the approach with the following example specifying some structures, with addressable and non-addressable fields. Note that for an addressable field that is of an array type, each element of the array is considered to be addressable.

install-C-file *open-struct.c*

declare [[*show-types=false*, *show-sorts=false*,
verbose = 0, *verbose-timing = 0*, *ML-print-depth = 1000*, *goals-limit = 20*]]

init−autocorres [*addressable-fields* =
inner.fld1
inner.fld2
outer.inner
outer-array.inner-array
array.elements
other.fy
two-dimensional.matrix
data-struct1.d1.y1
data-struct1.d1.y2
data-struct1.d1.y3
data-struct1.d1.y4
data-struct1.d2
data-struct2.d
data-array.array] *open-struct.c*

autocorres *open-struct.c*

Which split heaps are generated? Each split heap is a component of the record *lifted-globals*.

print-record *lifted-globals*

Structure *closed* does not have any addressable field. So there is a separate heap for that component *closed-C*.

Structure *inner* has three *unsigned* fields *fld1-C*, *fld2-C* and *fld3-C*, The first two fields are addressable whereas the third one is not. This means that the first two fields are put into the same common heap for unsigned integers *heap-w32* whereas the third field is stored in a dedicated heap for all remaining fields *dedicated-heap-inner-C*.

We use the following nomenclature to distinguish the various kinds of split heaps and types:

- closed type or structure: There is no addressable field within the type / structure, e.g. *closed-C*.
- open type or structure: There is at least one addressable field within the type / structure, e.g. *inner-C*.

- atomic heap: Heap for a closed type, or a common heap that might be nested in an open structure. e.g. *heap-closed-C*.
- dedicated heap: Internal heap for all non-addressable fields of an open type, e.g. *dedicated-heap-inner-C*. These dedicated heaps are used for the construction of a derived heap.
- derived heap: Heap for an open type / structure. The name 'derived' indicates that it is not a fundamental heap but a composition of several common heaps and a dedicated heap, e.g. *outer-C.inner-C*
- fundamental heap: those heaps that are directly present in the new global state *lifted-globals*. These are atomic heaps, and dedicated heaps.
- virtual heap: heaps that represent the user level view. They are directly visible in the program after heap-lifting. This excludes dedicated heaps, which non-the-less may be used in the model as representation of that type.

  Dedicated heaps play a special role, as they are subsumed within derived heaps. In a sense they are not part of the 'API' for split heaps: After heap lifting the resulting program only directly mentions atomic, common or derived heaps. Nevertheless dedicated heaps are important for the construction of derived heaps and might show up after unfolding the definitions for derived heaps.

  Also dedicated heaps may be used by the user as the actual heap for a specific type. Only the dedicated heaps and the atomic heaps commute, the derived heaps do not necessary commute with each other as they might have common components that alias each other.

26.21.3 Background

The starting point for the split heap abstraction is still the unified memory model (UMM) by Harvey Tuch (https://trustworthy.systems/publications/papers/Tuch%3Aphd.pdf). The UMM provides the general concept of *c-type* to convert between a byte-list representations of C values to abstract HOL-types. It also provides an explicit notion of typing and what a well-typed byte level heap is (with respect to a heap type description). With these notions in place one can describe the set of valid pointers for a C program. The model is quite elaborate and respects the potential nesting of structures. A well-typed pointer to a structure in the heap gives rise to a bunch of other valid sub-pointers, pointing to sub-fields, which again could point to a nested structure. In the UMM, array types are modelled as a special case of a structure, where each index corresponds to an artificial field, where the field

name is a unary encoding of the index. Hence, for most of the explanation in this file it is sufficient to elaborate on structures. The main limitation of the UMM is that it does not handle unions.

For the original split heap version of AutoCorres https://trustworthy.systems/publications/nicta_full_text/8758.pdf, David Greenaway put a simplified layer on top of the UMM focusing only on the root-pointers of a structure, ignoring any nested pointers to a field. This provides a split heap abstraction on the granularity of types.

The new split heap model takes an intermediate view. For closed structures it coincides with the original split heap model. Only root-pointers to a closed structure are valid. In addition for open structures also the nested pointers to addressable fields are welcome and valid. This flexibility allows the user to carefully open up the structures only as far as needed. Note the trade-off between expressibility and "proof-burden" of open structures. Aliasing is by construction ruled out for distinct split heaps, but has to be dealt with explicitly on the user level for common heaps.

To allow this flexibility some existing notions where slightly refined and some notions where introduced.

Footprints and valid pointers

The central typing judgement of UMM is *h-t-valid*, with infix syntax $d,g \models_t p$, meaning pointer p is valid with respect to heap type description d and a guard g. The default guard is *c-guard*, ensuring that the pointer is aligned, is not NULL and that the referenced value fits into the address space, meaning that the addresses of the individual bytes do not overflow the address space. This instantiation with *c-guard* is abbreviated with $d \models_t p$.

thm *c-guard-def*
thm *TypHeap.h-t-valid-def*
thm *h-t-valid-valid-footprint*

The judgement $d,g \models_t p$ also ensures that the footprint is valid: *valid-footprint* d (*ptr-val* p) (*typ-uinfo-t* $TYPE('a)$). Note that p is a typed pointer, carrying the (phantom) type of the value it points to p. This type annotation is no longer present in *valid-footprint* which only talks about the address of the pointer *ptr-val* and relates it to a type tag via *typ-uinfo-t*.

thm *valid-footprint-def*

Without going into details of the definition of *valid-footprint* we get guarantees for all nested pointers of the structure which also have *valid-footprint*. Moreover, what we do not know is whether the pointer p is itself a nested pointer of some enclosing structure.

To get the additional knowledge that term p is a root pointer, meaning it is not enclosed in a bigger structure, we have the notion *valid-root-footprint*. This notion was refined from the original version of AutoCorres *valid-simple-footprint*

to get the important property that every *valid-root-footprint* is indeed also a *valid-footprint*.

thm *valid-root-footprint-valid-footprint*

Also for all types that fit into the address space we can go from *valid-root-footprint* to *valid-simple-footprint*. The restriction on the address space size follows from the implicit property of the legacy *valid-simple-footprint*: The first byte and all the rest have a different tag, so there must not be a complete wrap around in the address space.

thm *valid-simple-footprint-size-td*
thm *valid-root-footprint-valid-simple-footprint*

The corresponding judgment to $d \models_t p$ for root pointers is *root-ptr-valid d p*. It implies *c-guard p* as well as *valid-root-footprint d* (*ptr-val p*) (*typ-uinfo-t* $TYPE('a)$).

thm *root-ptr-valid-def*

From byte heap to typed heaps

In UMM the core wrapper functions to provide a typed view on the raw byte level heap are:

- *h-val* to lookup typed values by typed pointers and
- *heap-update* to update the heap on the location designated by a typed pointer with a typed value.

  Provided a byte level heap and a heap type description there is a function to lift the heap into a typed heap:

- *lift-t* takes a pointer guard and a pair of byte level heap and heap type description and provides a partial function from typed pointers to typed values. The most prominent instance is abbreviation *clift* = *clift*.

  The resulting function is partial as it only lifts those pointers that actually point to valid addresses according to *h-t-valid*. This fact and the connection to *h-val* is reflected in *lift-t ?g* (*?h*, *?d*) = (λp. *if ?d*,*?g* $\models_t$ *p then Some* (*h-val ?h p*) *else None*).

thm *lift-t-if*

To lift root pointers there is

- *simple-lift*. It yields only defined values for root pointers, according to *root-ptr-valid*.

thm *simple-lift-def*

For the derived heaps we introduce the family of functions *plift*, which are defined in *wf-ptr-valid* and instantiated for each collection of open types. Like *clift* the definition is based on *lift-t*, but instead of the plain *c-guard* a custom validity-guard for the type is supplied $PTR\text{-}VALID('a)$. The relevant definitions are collected as named theorems:

thm *plift-defs*

Note that we used to introduced a separate distinct instance of *plift* and $PTR\text{-}VALID('a)$ for each type. Now we have encapsulated this construction in *open-types*. From a specification of which fields of a type should be addressable we first derive an variant without phantom typing of all the concepts especially *open-types.ptr-valid-u* and then define the phantom typed version based on that. This separation works around the limitation of the Isabelle type system that are also reflected in locales. It does not allow explicit quantification on type variables. So we can only have fixed type "variables" in the locale assumptions. However, the lemmas derived within a locale can still be polymorphic. So we express the needed assumptions in the locale *open-types* in the "untyped" world and derive the "typed" polymorphic versions as lemmas within that locale.

thm *open-struct.ptr-valid-u.simps*
thm *open-types.ptr-valid-u.simps*
thm *open-struct.ptr-valid-def*
thm *open-types.ptr-valid-def*

Let us look into the definitions for the open structure *inner-C* which itself is part of the open structure *outer-C*.

lemma *plift h = lift-t (ptr-valid (hrs-htd h)) h*
by (*simp add*: *open-struct.plift-def*)

thm *inner-C.ptr-valid.unfold*
lemma
fixes *p*::*inner-C ptr*
shows *ptr-valid d p =*
(*root-ptr-valid d p* ∨
(∃ *q*::*outer-C ptr. ptr-valid d q* ∧ *ptr-val p* = &(*q*→[''*inner-C*'']) ∨
(∃ *i*<5. ∃ *q*::(*inner-C*[5]) *ptr. ptr-valid d q* ∧ *ptr-val p* = &(*q*→[*replicate i CHR* ''*1*'']))))
by (*rule inner-C.ptr-valid.unfold*)

lemma
fixes *p*::*outer-C ptr*
shows *ptr-valid d p = root-ptr-valid d p*
by (*simp add*: *ptr-valid*)

A pointer to an *inner-C* is valid according to *PTR-VALID*($'a$) *d p*, if it is either

- a root pointer to an *inner-C*, or
- there is a valid pointer to an enclosing pointer *q* to*outer-C*, where *p* is obtained by dereferencing field *''inner-C''*, (Note that valid *outer-C* are already root pointers). or
- there is a valid pointer to an array *q*, where *p* is obtained by indexing into the array. Indexing into arrays is modelled by unary encoded field names *replicate i CHR ''1''*. (Note that valid array pointers themselve might have various possible extensions to a root pointer of an enclosing structure)

  These pointer-valid predicates are derived from the specification of addressable fields in the corresponding **autocorres** command. They enumerate the possibilities to arrive at a valid pointer starting from a valid root-pointer. The specification is converted to the parameter $\mathcal{T}$ of *open-types*. In our example program this is:

thm $\mathcal{T}$*-def*

The $\mathcal{T}$ is an association list mapping a type tag *typ-uinfo-t* of a *mem-type* to the list of addressable fields within that structure. From this specification we derive a lot of useful notions within the locale. Notably:

thm *ptr-valid* — These go up the chain to the root pointer.

From typed heaps to split heaps

Note that the functions in the previous subsubsection still are defined on the level of the monolithic UMM memory model. They are central building blocks to finally arrive at the split heap model. Essentially there is a split heap component in the lifted globals corresponding to *plift* for all the fundamental types.

However, instead of partial functions as provided by *wf-ptr-valid.plift* the split heap separates definedness from the actual value, by providing a pair of a total function from pointer to value and a validity predicate which indicates definedness. This design choice was introduced in the original AutoCorres.

As a prerequisite to lift the Functions into the split heap model, the new global state record *lifted-globals* is defined. For each fundamental heap this record contains a field for the split heap e.g. *heap-w32*.

These are the lookup functions provided by the record package, there are of course also the corresponding update functions, e.g.: *heap-w32-update*

Note that the update function behaves like a "map", which maps an update function on the selected heap to an update function on *lifted-globals*.

The typing information (heap type description) is preserved from the monolithic heap in component *heap-typing*.

print-record *lifted-globals*

For the derived heaps we provide a similar abstraction level by defining a derived heap lookup and a derived heap update function. These functions are a composition of the underlying functions for the fields of the structure.

thm *derived-heap-defs*

For example for *outer-C* we have.

- Lookup: *heap-outer-C*
- Pointwise update: *heap-outer-C-map*
- Validity is generically based on the heap typing for all heaps: $\lambda s.$ *IS-VALID*$('a)$ s

  Note the difference in the signature of the derived update *heap-outer-C-map* and the fundamental *heap-w32-update*. Besides the difference in naming between 'map' and 'update', the derived update is defined pointwise instead of heap-wide. The reason is that the pointwise update is what we actually need for autocorres and it is natural to break down a pointwise update defined via a function on f to the update functions on the fields of *outer-C*.

thm *heap-outer-C-def*
thm *heap-outer-C-map-def*

The connection between *globals* (containing the byte level heap) and *lifted-globals* is defined via the function *lift-global-heap*. The fundamental heaps are lifted via the corresponding *wf-ptr-valid.plift* functions. Validity of pointers is maintained by literally preserving the heap typing from the monolithic heap.

thm *lift-global-heap-def*
thm *lift-t-def*
thm *plift-defs*

26.21.4 User Level

Due to the abstraction layer of derived heap types, the structure of the programs after heap lifting is analogous for closed and opened structures.

context *ts-definition-set-c1*
begin

thm *ts-def*

lemma *set-c1'* *p* *c* ≡ *do* {*guard* (λ*s*. *IS-VALID*(*closed-C*) *s* *p*);
modify (*heap-closed-C-update* (λ*h*. *h*(*p* := *c1-C-update* (λ-. *c*) (*h* *p*))))
}
by (*rule* *ts-def*)

end

context *ts-definition-outer-fld1-upd*
begin
thm *ts-def*
end

context *ts-definition-set-element*
begin
thm *ts-def*
end

context *ts-definition-set-matrix-element*
begin
thm *ts-def*
end

Of course, when it comes to specifications and proving properties about the programs, the user has to take care about aliasing of addressable fields. AutoCorres generates a bunch of theorems and sets up the simpsets with some obvious rules for derived heaps, which hopefully helps in doing so. However, as a last resort one might still have to unfold the definitions and work with the parts. Here is some examples of some theorems or collections of theorems.

thm *lifted-globals-ext-simps*
thm *ptr-valid*

thm *heap-inner-C.write-comp*
thm *heap-inner-C.write-id*
thm *heap-inner-C.write-other-commute*
thm *heap-inner-C.write-same*
thm *update-commute*
thm *read-write-same*
thm *read-write-other*
thm *write-cong*
thm *update-compose*
thm *valid-implies-c-guard*
thm *read-commutes*
thm *write-commutes*

thm *fg-cons-simps*

thm *typ-info-simps*
thm *td-names-simps*
thm *typ-name-simps*
thm *upd-lift-simps*
thm *upd-other-simps*
thm *size-align-simps*
thm *fl-Some-simps*
thm *fl-ti-simps*
thm *typ-tag-defs*
thm *size-simps*
thm *typ-name-itself*

addressable-fields,*merge-addressable-fields*,*read-dedicated-heap*,*write-dedicated-heap*

We distinguish three aspects when thinking and reasoning about split heaps: the monolithic heap stored in *globals* the split heap components in *lifted-globals* and the function *lift-global-heap* which transforms between both types. Autocorres does a refinement proof that a program operating on *lifted-globals* refines a program on *globals* where the heaps are related by function *lift-global-heap*.

For open structures, the fields which are addressable have two representations in *lifted-globals*:

1. the relevant value in the common heap, i.e. *heap-w32* for the *inner.fld1* pointers
2. an unused value in the field of the dedicate heap, i.e. the *fld1-C* in the *dedicated-heap-inner-C*. Having the dedicated heap allows us to keep all non-addressable fields of a structure in a single split heap component of *lifted-globals*. The values of the addressable fields are irrelevant for the virtual heap. When reading from a virtual heap we start with the value in the dedicated heap and overwrite the addressable fields by considering the common heaps.

thm *heap-inner-C-def*

When writing to a virtual heap we have a series of updates on *lifted-globals*. We start by updating the non-addressable fields in the dedicated heap and then updating the common heaps with the new values for the addressable fields:

thm *heap-inner-C-map-def*

To express "update the non-addressable fields of the dedicated heap" in an uniform way we make use of the concept of *scenes* from the world of *lenses*. In contrast to lenses, scenes are formalised by properties of a single function $'a \Rightarrow 'a \Rightarrow 'a$ which is referred to as *merge* or 'overrider' operation. The intuition of *merge* x y is that we take certain portions of a compound value x and merge (or override) them into another compound value y. Scenes

are formalised in locale *is-scene*. The notion *merge* emphasises the the symmetric nature of the merge operation. Like a lense, a scene 'focuses' on a portion of a compound value $'a$. The merge operation takes that portion from the first argument and the complement of that portion from the second argument. Think for example of a pair and the 'portion' being the first component. Then *merge x y* = (*fst x*, *snd y*). The notion *overrider* puts more emphasis on the merge as an update function, in particular this view becomes obvious if we think of a partial application *merge x*: this is an update function which overrides the 'portion' within an value *y* to that portion of *x*. You can also think of fixing the 'portion' to those values in *x*. Analogously λx. *merge x y* fixes the complement of the 'portion' to those values in *y*. Scenes are closely related to lenses in the sense that it is straightforward to derive a scene from a lense *lense ?r ?w* $\Longrightarrow$ *is-scene* (λa. *?w* (λ-. *?r a*)).

thm *is-scene-of-lense*

For HOL the decisive advantage of scenes compared to lenses is that they only have one type parameter, which fixes the type of the compound value. But still you can formally 'talk' about different 'portions' of that value in an uniform way. For example you can have a list or a set of scenes to describe their composition. This property can be used to uniformly describe "all addressable fields" of a value or "all non-addressable fields" of a value.

The relevant merging function for us is *merge-addressable-fields* which is an abbreviation $\lambda\mathcal{T}$. *merge-ti-list* (*map snd* (*open-types.addressable-fields* $\mathcal{T}$ *TYPE*($'a$))).

term ‹*open-types.merge-addressable-fields*›

The term*merge-addressable-fields old new* has to effects:

- The addressable fields are taken from *old*
- The non-addressable fields are taken from *new*.

To express the same semantics that we have just described for *lifted-globals* on the UMM heap in *globals* we make use of the functions *read-dedicated-heap* and *write-dedicated-heap*. These functions internally again make use of *merge-addressable-fields*. We zero out all parts of the compound value we are not actually interested in. This gives us canonical values which we can easily relate to the *lifted-globals*. For example for *read-dedicated-heap* we zero out all addressable fields and also all values which are not even *PTR-VALID*($'a$) (according to *plift*).

lemma (**in** *open-types*)
read-dedicated-heap h p =
merge-addressable-fields ZERO($'a$::*mem-type*) (*the-default ZERO*($'a$) (*plift h p*))

by (*simp add*: *read-dedicated-heap-def*)

thm *open-types.read-dedicated-heap-def*
thm *open-types.write-dedicated-heap-def*

The lifting function *lift-global-heap* directly uses *read-dedicated-heap* to construct a dedicated heap from the UMM heap:

thm *lift-global-heap-def*

To have a canonical representation we overwrite the fields in the dedicated heap with zeros, using the generic constants *addressable-fields* and *merge-addressable-fields* defined in *open-types*.

read-dedicated-heap,*write-dedicated-heap* combine these with *h-val*, to get the canonical heap access.

Why all the variants? *h-val* / *read-dedicated-heap*

thm *lifted-globals-ext-simps*
thm *lift-global-heap-def*
thm *open-types.read-dedicated-heap-def*

When taking a close look in the definition of *lift-global-heap* one sees that the abstraction for a split heap component from the monolithic heap is:

read-dedicated-heap (*t-hrs-*′ *g*)

So this might seem a bit surprising at first sight. What is all the indirection useful for:

1. the *read-dedicated-heap* adds an option layer to the plain *h-val* and *plift*
2. with *t-hrs-*′ we get the heap representation (including bytes and types)

   When we know that a pointer is valid there is a tight connection between *read-dedicated-heap* and *h-val*

thm *open-struct.ptr-valid.ptr-valid-plift-Some-hval*
thm *open-struct.read-dedicated-heap-def*

This connection directly carries over to lifting.

thm *read-commutes*

The difference between *read-dedicated-heap* and *h-val* becomes apparent when looking at partial heaps and invalid pointers. While *read-dedicated-heap* always yields *ZERO*(′*a*) the plain *h-val* will still construct some value out of the bytes in the heap. So in a sense the indirection to *read-dedicated-heap* makes heaps equal if the agree on the valid pointers only. So we encode "heap equality only on valid pointers" in an ordinary equality on the complete lifted heap. This choice ensure compositionality for the polymorphic

ZERO('*a*), in the sense that the *ZERO*('*a*) of a structure means that all subcomponents are *ZERO*('*a*).

Moreover *read-dedicated-heap* only reads the fields which are not addressable, and overwrites the addressable fields with *ZERO*('*a*). This gives us a canonical view on the partial heap.

thm *zero-simps*
thm *open-struct.ptr-valid.plift-None*

I miss the typing in the split heap!

A peculiar property of the original split heap model of autocorres was that you loose parts of the type information that were directly available in the byte-level heap. There was no heap-type description in *lifted-globals* but only the more abstract *is-valid-<type>* fields for each type. For closed types this isn't so much of an issue as essentially all relevant type information is captured in the fact that all pointers to that type have a distinct split heap, which is separate from all other split heaps. But for shared types this is no longer true. Thus we maintain the typing information also in the split heap in the compponent *heap-typing*. The relation to the typing information of the original byte-level heap is encapsulated in *lift-global-heap*. For low-level operations that are embedded via *exec-concrete* you can directly connect the typing of the monolithic and the split heap. In particular you can derive *c-guard p* from that information. Or you can derive that if you have two valid pointers of the same type, where you only know that the address is different, you can still conclude that the complete pointer span of the pointers do not overlap.

But keep in mind that typing is only a "discipline". As in the split heap you can manipulate the values on the heap or split heap independent of the typing information. When well-typedness or *PTR-VALID*('*a*) is available you might be able to derive properties like distinctness of pointer spans. The distinctness of pointers spans is the actual reason that certain heap updates commute, e.g. *disjnt* (*ptr-span ?p*) (*ptr-span ?q*) $\Longrightarrow$ *heap-inner-C-map ?q ?g* (*heap-inner-C-map ?p ?f ?s*) = *heap-inner-C-map ?p ?f* (*heap-inner-C-map ?q ?g ?s*)

thm *heap-inner-C.write-other-commute*

26.21.5 Simulation Proof

The simulation of a concrete program *C* operating on the byte-level heap by an abstract program *A* operating on the split heaps is captured in predicate*L2Tcorres lift-global-heap A C*. The proof for an instance of *C* follows the syntactic structure of *C* by applying introduction rules and synthesizing the abstract program *A*.

Intuitively the core properties on which the simulation proof builds are:

- Abstract programs only operate on abstract heap values not on byte-lists. Byte-level concepts like padding fields are irrelevant for the abstract program. The abstract heaps and programs don't distinguish between two byte-level heaps if they agree on all values of non-padding fields of properly allocated and typed portions of the heap.
- Lookup and update via a pointer into a structure can be simulated by an lookup and update via the root pointer of the structure.
- Each non-addressable field of a structure is mapped into exactly one split heap. For a 'closed' structure without any addressable fields this is the heap for the type.
- For an 'open' structure an addressable field is mapped into a shared heap or multiple shared heaps in case the field is again an open structure.
- The pointer spans of two valid root pointers of different types do not overlap.
- The pointer spans of two valid root pointers to the same type might overlap only if the pointer value is the same.

  The actual proof of a simulation is divided into three main steps.
- First some general theorems relating byte-level and split heaps are derived. Most prominently *typ-heap-simulation* and related predicates.
- These theorems are used to instantiate the syntax driven introduction rules collected in named theorems ⟦*abs-expr ?st ?X ?f1.0 ?f1'*; *abs-expr ?st ?Y ?f2.0 ?f2'*⟧ ⟹ *abs-expr ?st* (λ*s*. *?X s* ∧ *?Y s*) (λ*s*. (*?f1.0 s*, *?f2.0 s*)) (λ*s*. (*?f1' s*, *?f2' s*))

  abs-expr ?st (λ-. *True*) (λ*s*. *?a*) (λ*s*. *?a*)

  abs-expr ?st ?P ?a' ?a ⟹ *abs-guard ?st* (λ*s*. *?P s* ∧ *?a' s*) *?a*

  abs-guard ?st (λ-. *?P*) (λ-. *?P*)

  ⟦*abs-guard ?st ?G ?G'*; *abs-guard ?st ?H ?H'*⟧ ⟹ *abs-guard ?st* (λ*s*. *?G s* ∧ *?H s*) (λ*s*. *?G' s* ∧ *?H' s*)

  ⟦*struct-rewrite-modifies ?P ?b ?c*; *abs-guard ?st ?P' ?P*; *abs-modifies ?st ?Q ?a ?b*⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *L2-modify ?a*)) (*L2-modify ?c*)

  ⟦*struct-rewrite-expr ?P ?b ?c*; *abs-guard ?st ?P' ?P*; *abs-expr ?st ?Q ?a ?b*⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *L2-gets ?a ?n*)) (*L2-gets ?c ?n*)

  L2Tcorres ?st (*L2-gets* (λ-. *?a*) *?n*) (*L2-gets* (λ-. *?a*) *?n*)

$[\![$*struct-rewrite-guard ?b ?c*; *abs-guard ?st ?a ?b*$]\!] \Longrightarrow$ *L2Tcorres ?st* (*L2-guard ?a*) (*L2-guard ?c*)

$[\![$*THIN* ($\bigwedge$*x. L2Tcorres ?st* (*?B′ x*) (*?B x*)); *THIN* ($\bigwedge$*r. struct-rewrite-expr* (*?G r*) (*?C′ r*) (*?C r*)); *THIN* ($\bigwedge$*r. abs-guard ?st* (*?G′ r*) (*?G r*)); *THIN* ($\bigwedge$*r. abs-expr ?st* (*?H r*) (*?C″ r*) (*?C′ r*))$]\!] \Longrightarrow$ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λ*s. ?G′ ?i s* $\wedge$ *?H ?i s*)) (λ*-. L2-while ?C″* (λ*i. L2-seq* (*?B′ i*) (λ*r. L2-seq* (*L2-guard* (λ*s. ?G′ r s* $\wedge$ *?H r s*)) (λ*-. L2-gets* (λ*-. r*) *?n*))) *?i ?n*)) (*L2-while ?C ?B ?i ?n*)

abs-spec ?st ?P ?A ?C $\Longrightarrow$ *L2Tcorres ?st* (*L2-seq* (*L2-guard ?P*) (λ*-. L2-spec ?A*)) (*L2-spec ?C*)

abs-assume ?st ?P ?A ?C $\Longrightarrow$ *L2Tcorres ?st* (*L2-seq* (*L2-guard ?P*) (λ*-. L2-assume ?A*)) (*L2-assume ?C*)

abs-spec ?st (λ*-. True*) {(*a*, *b*). *?C*} {(*a*, *b*). *?C*}

$[\![$*THIN* (*L2Tcorres ?st ?L ?L′*); *THIN* (*L2Tcorres ?st ?R ?R′*); *THIN* (*struct-rewrite-expr ?P ?C′ ?C*); *THIN* (*abs-guard ?st ?P′ ?P*); *THIN* (*abs-expr ?st ?Q ?C″ ?C′*)$]\!] \Longrightarrow$ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* $\wedge$ *?Q s*)) (λ*-. L2-condition ?C″ ?L ?R*)) (*L2-condition ?C ?L′ ?R′*)

$[\![$*THIN* (*L2Tcorres ?st ?L′ ?L*); *THIN* ($\bigwedge$*r. L2Tcorres ?st* (*?R′ r*) (*?R r*))$]\!] \Longrightarrow$ *L2Tcorres ?st* (*L2-seq ?L′ ?R′*) (*L2-seq ?L ?R*)

$[\![$*struct-rewrite-guard ?b ?c*; *abs-guard ?st ?a ?b*; $\bigwedge$*s s′. [*$\![$*?c s*; *s′ = ?st s*$]\!] \Longrightarrow$ *L2Tcorres ?st ?f ?g*$]\!] \Longrightarrow$ *L2Tcorres ?st* (*L2-guarded ?a ?f*) (*L2-guarded ?c ?g*)

$[\![$*THIN* (*L2Tcorres ?st ?L ?L′*); *THIN* ($\bigwedge$*r. L2Tcorres ?st* (*?R r*) (*?R′ r*))$]\!] \Longrightarrow$ *L2Tcorres ?st* (*L2-catch ?L ?R*) (*L2-catch ?L′ ?R′*)

L2Tcorres ?st ?L ?L′ $\Longrightarrow$ *L2Tcorres ?st* (*L2-try ?L*) (*L2-try ?L′*)

L2Tcorres ?st (*L2-unknown ?ns*) (*L2-unknown ?ns*)

L2Tcorres ?st (*L2-throw ?x ?n*) (*L2-throw ?x ?n*)

($\bigwedge$*x y. L2Tcorres ?st* (*?P x y*) (*?P′ x y*)) $\Longrightarrow$ *L2Tcorres ?st* (*case ?a of* (*x*, *y*) $\Rightarrow$ *?P x y*) (*case ?a of* (*x*, *y*) $\Rightarrow$ *?P′ x y*)

$[\![$*THIN* (*L2Tcorres ?st ?L ?L′*); *THIN* (*L2Tcorres ?st ?R ?R′*)$]\!] \Longrightarrow$ *L2Tcorres ?st* (*L2-seq ?L* (λ*-. ?R*)) (*L2-seq ?L′* (λ*-. ?R′*))

($\bigwedge$*a b. abs-expr ?st* (*?P a b*) (*?A a b*) (*?C a b*)) $\Longrightarrow$ *abs-expr ?st* (*case ?r of* (*a*, *b*) $\Rightarrow$ *?P a b*) (*case ?r of* (*a*, *b*) $\Rightarrow$ *?A a b*) (*case ?r of* (*a*, *b*) $\Rightarrow$ *?C a b*)

($\bigwedge$*a b. abs-guard ?st* (*?A a b*) (*?C a b*)) $\Longrightarrow$ *abs-guard ?st* (*case ?r of* (*a*, *b*) $\Rightarrow$ *?A a b*) (*case ?r of* (*a*, *b*) $\Rightarrow$ *?C a b*)

L2Tcorres ?st L2-fail L2-fail

abs-expr id (λ-. *True*) *?A ?A*

abs-expr ?st ?P ?A ?C ⟹ *abs-expr ?st ?P* (λ*s r*. *?A s*) (λ*s r*. *?C s*)

abs-modifies id (λ-. *True*) *?A ?A*

L2Tcorres id ?A ?C ⟹ *L2Tcorres ?st* (*exec-concrete ?st* (*L2-call ?A ?emb ?ns*)) (*L2-call ?C ?emb ?ns*)

L2Tcorres id ?A ?C ⟹ *L2Tcorres ?st* (*exec-concrete ?st* (*L2-call-L1 ?arg-xf ?gs ?ret-xf ?A*)) (*L2-call-L1 ?arg-xf ?gs ?ret-xf ?C*)

L2Tcorres ?st ?A ?C ⟹ *L2Tcorres id* (*exec-abstract ?st* (*L2-call ?A ?emb ?ns*)) (*L2-call ?C ?emb ?ns*)

L2Tcorres ?st ?A ?C ⟹ *L2Tcorres ?st* (*L2-call ?A ?emb ?ns*) (*L2-call ?C ?emb ?ns*)

⟦*typ-heap-simulation ?st ?getter ?setter ?vgetter ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?x' ?x*⟧ ⟹ *abs-guard ?st* (λ*s*. *?P s* ∧ *?vgetter s* (*?x' s*)) (λ*s*. *c-guard* (*?x s*))

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?x' ?x*⟧ ⟹ *abs-expr ?st* (λ*s*. *?P s* ∧ *?v s* (*?x' s*)) (λ*s*. *?r s* (*?x' s*)) (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*?x s*))

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?Pb ?b' ?b*; ⋀*v*. *abs-expr ?st ?Pc* (*?c' v*) (*?c v*)⟧ ⟹ *abs-modifies ?st* (λ*s*. *?Pb s* ∧ *?Pc s* ∧ *?v s* (*?b' s*)) (λ*s*. *?w* (*?b' s*) (λ-. *?c'* (*?r s* (*?b' s*)) *s*) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?b s*) (*?c* (*heap-lift--h-val* (*hrs-mem* (*?t-hrs s*)) (*?b s*)) *s*))) *s*)

struct-rewrite-expr ?P ?a' ?a ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?a' s*) *?a*

struct-rewrite-guard (λ-. *?P*) (λ-. *?P*)

⟦*struct-rewrite-guard ?b' ?b*; *struct-rewrite-guard ?a' ?a*⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?a' s* ∧ *?b' s*) (λ*s*. *?a s* ∧ *?b s*)

(⋀*a b*. *struct-rewrite-guard* (*?A a b*) (*?C a b*)) ⟹ *struct-rewrite-guard* (*case ?r of* (*a*, *b*) ⇒ *?A a b*) (*case ?r of* (*a*, *b*) ⇒ *?C a b*)

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-guard ?Q* (λ*s*. *c-guard* (*?p' s*))⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?Q s*) (λ*s*. *c-guard* (*PTR*(*?'f*) &(*?p s*→*?field-name*)))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-guard ?Q* (λ*s*. *c-guard* (*?p' s*))⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?Q s* ∧ *0* ≤ *?k* ∧ *nat ?k* < *CARD*(*?'n*)) (λ*s*. *c-guard* (*PTR-COERCE*(*?'f*[*?'n*] → *?'f*) (*PTR*(*?'f*[*?'n*]) &(*?p s*→*?field-name*)) +$_p$ *?k*))

struct-rewrite-expr (λ-. *True*) (λ-. *?a*) (λ-. *?a*)

struct-rewrite-expr ?P ?A ?C ⟹ struct-rewrite-expr ?P (λs -. ?A s) (λs -. ?C s)

(⋀a b. struct-rewrite-expr (?P a b) (?A a b) (?C a b)) ⟹ struct-rewrite-expr (case ?r of (a, b) ⇒ ?P a b) (case ?r of (a, b) ⇒ ?A a b) (case ?r of (a, b) ⇒ ?C a b)

struct-rewrite-expr (λ-. True) (λs. h-val (?h s) (?p s)) (λs. h-val (?h s) (?p s))

struct-rewrite-expr ?P ?a ?c ⟹ struct-rewrite-expr ?P (λs. h-val (?h s) (?a s)) (λs. h-val (?h s) (?c s))

⟦valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update; struct-rewrite-expr ?P ?p′ ?p; struct-rewrite-expr ?Q ?a (λs. h-val (hrs-mem (?t-hrs s)) (?p′ s))⟧ ⟹ struct-rewrite-expr (λs. ?P s ∧ ?Q s) (λs. ?field-getter (?a s)) (λs. h-val (hrs-mem (?t-hrs s)) (PTR(?′f) &(?p s→?field-name)))

⟦valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update; abs-expr ?st ?P ?a ?c⟧ ⟹ abs-expr ?st ?P (λs. ?field-getter (?a s)) (λs. ?field-getter (?c s))

⟦valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update; struct-rewrite-expr ?P ?p′ ?p; struct-rewrite-expr ?Q ?a (λs. h-val (hrs-mem (?t-hrs s)) (?p′ s))⟧ ⟹ struct-rewrite-expr (λs. ?P s ∧ ?Q s ∧ 0 ≤ ?k ∧ nat ?k < CARD(?′n)) (λs. ?field-getter (?a s).[nat ?k]) (λs. h-val (hrs-mem (?t-hrs s)) (PTR-COERCE(?′f[?′n] → ?′f) (PTR(?′f[?′n]) &(?p s→?field-name)) +p ?k))

⟦valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update; struct-rewrite-expr ?P ?p′ ?p; struct-rewrite-expr ?Q ?a (λs. h-val (hrs-mem (?t-hrs s)) (?p′ s))⟧ ⟹ struct-rewrite-expr (λs. ?P s ∧ ?Q s ∧ 0 ≤ ?k ∧ nat ?k < CARD(?′n)) (λs. ?field-getter (?a s).[nat ?k]) (λs. h-val (hrs-mem (?t-hrs s)) (PTR(?′f) &(?p s→?field-name) +p ?k))

struct-rewrite-modifies (λ-. True) ?A ?A

⟦typ-heap-simulation ?st ?getter ?setter ?vgetter ?t-hrs ?t-hrs-update; struct-rewrite-expr ?P ?p′ ?p; struct-rewrite-expr ?Q ?v′ ?v⟧ ⟹ struct-rewrite-modifies (λs. ?P s ∧ ?Q s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (?p′ s) (?v′ s))) s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (?p s) (?v s))) s)

heap-lift--wrap-h-val (heap-lift--h-val ?s ?p) (h-val ?s ?p)

heap-lift--wrap-h-val (h-val ?s (PTR(?′a) &(?p→?f))) (h-val ?s (PTR(?′a) &(?p→?f)))

heap-lift--wrap-h-val (h-val ?s (PTR-COERCE(?′b → ?′a) ?p +p ?k)) (h-val ?s (PTR-COERCE(?′b → ?′a) ?p +p ?k))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-expr ?Q ?f' ?f*; ⋀*s. heap-lift--wrap-h-val* (*?h-val-p' s*) (*h-val* (*hrs-mem* (*?t-hrs s*)) (*?p' s*)); *struct-rewrite-modifies ?R* (λ*s. ?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p'' s*) (*?u s* (*?field-setter* (*?f' s*))))) *s*) (λ*s. ?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p' s*) (*?field-setter* (*?f' s*) (*?h-val-p' s*)))) *s*); *struct-rewrite-guard ?S* (λ*s. c-guard* (*?p' s*))⟧ ⟹ *struct-rewrite-modifies* (λ*s. ?P s* ∧ *?Q s* ∧ *?R s* ∧ *?S s*) (λ*s. ?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p'' s*) (*?u s* (*?field-setter* (*?f' s*))))) *s*) (λ*s. ?t-hrs-update* (*hrs-mem-update* (*heap-update* (*PTR*(*?'f*) &(*?p s*→*?field-name*)) (*?f s* (*h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR*(*?'f*) &(*?p s*→*?field-name*)))))) *s*)

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-expr ?Q ?f' ?f*; ⋀*s. heap-lift--wrap-h-val* (*?h-val-p' s*) (*h-val* (*hrs-mem* (*?t-hrs s*)) (*?p' s*)); *struct-rewrite-modifies ?R* (λ*s. ?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p'' s*) (*?u s* (*?field-setter* (λ*a. Arrays.update a* (*nat ?k*) (*?f' s* (*a.*[*nat ?k*])))))))) *s*) (λ*s. ?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p' s*) (*?field-setter* (λ*a. Arrays.update a* (*nat ?k*) (*?f' s* (*a.*[*nat ?k*]))) (*?h-val-p' s*)))) *s*); *struct-rewrite-guard ?S* (λ*s. c-guard* (*?p' s*))⟧ ⟹ *struct-rewrite-modifies* (λ*s. ?P s* ∧ *?Q s* ∧ *?R s* ∧ *?S s* ∧ *0* ≤ *?k* ∧ *nat ?k* < *CARD*(*?'n*)) (λ*s. ?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p'' s*) (*?u s* (*?field-setter* (λ*a. Arrays.update a* (*nat ?k*) (*?f' s* (*a.*[*nat ?k*])))))))) *s*) (λ*s. ?t-hrs-update* (*hrs-mem-update* (*heap-update* (*PTR-COERCE*(*?'f*[*?'n*] → *?'f*) (*PTR*(*?'f*[*?'n*]) &(*?p s*→*?field-name*)) +$_p$ *?k*) (*?f s* (*h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR-COERCE*(*?'f*[*?'n*] → *?'f*) (*PTR*(*?'f*[*?'n*]) &(*?p s*→*?field-name*)) +$_p$ *?k*))))) *s*)

valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter ⟹ *abs-expr ?st* (λ*-. True*) *?new-getter ?old-getter*

⟦*valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter*; ⋀*old. abs-expr ?st* (*?P old*) (*?v old*) (*?v' old*)⟧ ⟹ *abs-modifies ?st* (λ*s.* ∀ *old. ?P old s*) (λ*s. ?new-setter* (λ*old. ?v old s*) *s*) (λ*s. ?old-setter* (λ*old. ?v' old s*) *s*)

valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter ⟹ *struct-rewrite-expr* (λ*-. True*) *?old-getter ?old-getter*

⟦*valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter*; ⋀*old. struct-rewrite-expr* (*?P old*) (*?v' old*) (*?v old*)⟧ ⟹ *struct-rewrite-modifies* (λ*s.* ∀ *old. ?P old s*) (λ*s. ?old-setter* (λ*old. ?v' old s*) *s*) (λ*s. ?old-setter* (λ*old. ?v old s*) *s*)

abs-spec ?st (λ*-. True*) {(*a*, *b*). *b may-not-modify-globals a*} {(*a*, *b*). *b may-not-modify-globals a*}

⟦*valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter*; *abs-spec ?st* (λ*-. True*) {(*a*, *b*). *?C a b*} {(*a*, *b*). *?C' a b*}⟧ ⟹ *abs-spec*

?st (λ-. *True*) {(*a*, *b*). *mex* (λ*x*. *?C* (*?new-setter* (λ-. *x*) *a*) *b*)} {(*a*, *b*). *mex* (λ*x*. *?C′* (*?old-setter* (λ-. *x*) *a*) *b*)}

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?x′* (λ*s*. *PTR-COERCE*(*?′a*[*?′b*] → *?′a*) (*?x s*))⟧ ⟹ *abs-guard ?st* (λ*s*. *?P s* ∧ (∀ *a*∈*set* (*array-addrs* (*?x′ s*) *CARD*(*?′b*)). *?v s a*)) (λ*s*. *c-guard* (*?x s*))

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?Pb ?b′ ?b*; *abs-expr ?st ?Pn ?n′ ?n*; *abs-expr ?st ?Pv ?y′ ?y*⟧ ⟹ *abs-modifies ?st* (λ*s*. *?Pb s* ∧ *?Pn s* ∧ *?Pv s* ∧ *?n′ s* < *CARD*(*?′b*) ∧ (∀ *ptr*∈*set* (*array-addrs* (*PTR-COERCE*(*?′a*[*?′b*] → *?′a*) (*?b′ s*)) *CARD*(*?′b*)). *?v s ptr*)) (λ*s*. *?w* (*PTR-COERCE*(*?′a*[*?′b*] → *?′a*) (*?b′ s*) +$_p$ *int* (*?n′ s*)) (λ*v*. *?y′ s*) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?b s*) (*Arrays.update* (*h-val* (*hrs-mem* (*?t-hrs s*)) (*?b s*)) (*?n s*) (*?y s*)))) *s*)

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?Pb ?b′ ?b*; *abs-expr ?st ?Pn ?n′ ?n*⟧ ⟹ *abs-expr ?st* (λ*s*. *?Pb s* ∧ *?Pn s* ∧ *?n′ s* < *CARD*(*?′b*) ∧ *?v s* (*PTR-COERCE*(*?′a*[*?′b*] → *?′a*) (*?b′ s*) +$_p$ *int* (*?n′ s*))) (λ*s*. *?r s* (*PTR-COERCE*(*?′a*[*?′b*] → *?′a*) (*?b′ s*) +$_p$ *int* (*?n′ s*))) (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*?b s*).[*?n s*])

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*32 word*) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-w32-update* (λ*h*. *h*(*?p* := *?a s*)) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*32 word*) *t* (*PTR-COERCE*(*32 signed word* → *32 word*) *?p*) ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-w32-update* (λ*h*. *h*(*PTR-COERCE*(*32 signed word* → *32 word*) *?p* := *UCAST*(*32 signed* → *32*) (*?a s*))) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*8 word*) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-w8-update* (λ*h*. *h*(*?p* := *?a s*)) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*8 word*) *t* (*PTR-COERCE*(*8 signed word* → *8 word*) *?p*) ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-w8-update* (λ*h*. *h*(*PTR-COERCE*(*8 signed word* → *8 word*) *?p* := *UCAST*(*8 signed* → *8*) (*?a s*))) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*32 word ptr*) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-w32′ptr-update* (λ*h*. *h*(*?p* := *?a s*)) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unit ptr) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-unit'ptr-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(closed-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-closed-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unpacked-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-unpacked-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(arr-struct-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-arr-struct-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(unpacked-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. unpacked-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (valid-array-base.valid-array (λh. IS-VALID(unpacked-C) h)) t ?p ∧ ?P t)) (λ-. L2-modify (λs. unpacked-C.outer.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(two-dimensional-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-two-dimensional-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-struct1-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-struct1-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-struct2-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-struct2-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(inner-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-inner-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(inner-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. inner-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(outer-array-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-outer-array-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(data-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. data-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-array-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-array-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(outer-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-outer-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(other-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-other-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(unpacked-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. unpacked-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(array-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-array-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-w32.guard-with-fresh-stack-ptr ?n ?init$_a$

(*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))

⟦$\bigwedge s\ p.\ ?f_c\ p \cdot s$ *?{* $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* *}*; *struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* ($\bigwedge p$. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P' s* ∧ *?Q s*)) (λ-. *heap-w32.assume-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))

⟦*struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* ($\bigwedge p$. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P' s* ∧ *?Q s*)) (λ-. *heap-w8.guard-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))

⟦$\bigwedge s\ p.\ ?f_c\ p \cdot s$ *?{* $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* *}*; *struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* ($\bigwedge p$. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P' s* ∧ *?Q s*)) (λ-. *heap-w8.assume-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))

⟦*struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* ($\bigwedge p$. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P' s* ∧ *?Q s*)) (λ-. *heap-w32'ptr.guard-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))

⟦$\bigwedge s\ p.\ ?f_c\ p \cdot s$ *?{* $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* *}*; *struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* ($\bigwedge p$. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P' s* ∧ *?Q s*)) (λ-. *heap-w32'ptr.assume-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))

⟦*struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* ($\bigwedge p$. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P' s* ∧ *?Q s*)) (λ-. *heap-unit'ptr.guard-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}unit'ptr.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}C.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}closed\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}closed\text{-}C.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}unpacked\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap$

$(?f_a\ p)\ (?f_c\ p))[\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-unpacked-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))*

$[\![$*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))*$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-arr-struct-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))*

$[\![\bigwedge$*s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on 𝒮 s }*$]\!]$; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))*$[\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-arr-struct-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))*

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unpacked-C[3][2]) t ?p ∧ ?P t)) (λ-. L2-modify (λs. unpacked-C.outer.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

$[\![$*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))*$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-unpacked-C′array-3′array-2.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))*

$[\![\bigwedge$*s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on 𝒮 s }*$]\!]$; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))*$[\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-unpacked-C′array-3′array-2.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))*

$[\![$*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))*$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-two-dimensional-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))*

$[\![\bigwedge$*s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on 𝒮 s }*$]\!]$; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$; abs-guard lift-global-heap ?P′ ?P;*

abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-two-dimensional-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-data-struct1-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦$\bigwedge$s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-data-struct1-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-data-struct2-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦$\bigwedge$s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-data-struct2-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-inner-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦$\bigwedge$s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-inner-C.assume-with-fresh-stack-ptr ?n

$?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(inner\text{-}C[5])\ t\ ?p \wedge\ ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ inner\text{-}C.heap\text{-}array\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c;\ abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P;\ abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c';\ THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge\ ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}inner\text{-}C'array\text{-}5.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\};\ struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c;\ abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P;\ abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c';\ THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge\ ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}inner\text{-}C'array\text{-}5.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c;\ abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P;\ abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c';\ THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge\ ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}outer\text{-}array\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\};\ struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c;\ abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P;\ abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c';\ THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge\ ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}outer\text{-}array\text{-}C.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(data\text{-}C[10])\ t\ ?p \wedge\ ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ data\text{-}C.heap\text{-}array\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c;\ abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P;\ abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c';\ THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge\ ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}C'array\text{-}10.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}C'array\text{-}10.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}array\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}array\text{-}C.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}outer\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}outer\text{-}C.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}other\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap$

$(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-other-C.assume-with-fresh-stack-ptr ?n ?init*$_a$ (*L2-VARS ?f*$_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n ?init*$_c$ (*L2-VARS ?f*$_c$ *?nm*))

abs-expr lift-global-heap ?P ?a ?c $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda t.$ *IS-VALID*(*unpacked-C*[*2*]) *t ?p* $\wedge$ *?P t*)) (λ-. *L2-modify* ($\lambda s.$ *unpacked-C.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

$[\![$*struct-rewrite-expr ?P ?init*$_c'$ *?init*$_c$; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init*$_a$ *?init*$_c'$; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f*$_a$ *p*) (*?f*$_c$ *p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-unpacked-C′array-2.guard-with-fresh-stack-ptr ?n ?init*$_a$ (*L2-VARS ?f*$_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n ?init*$_c$ (*L2-VARS ?f*$_c$ *?nm*))

$[\![\bigwedge s\ p.$ *?f*$_c$ *p* $\cdot$ *s ?*$\{\!|$ $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* $|\!\}$; *struct-rewrite-expr ?P ?init*$_c'$ *?init*$_c$; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init*$_a$ *?init*$_c'$; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f*$_a$ *p*) (*?f*$_c$ *p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-unpacked-C′array-2.assume-with-fresh-stack-ptr ?n ?init*$_a$ (*L2-VARS ?f*$_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n ?init*$_c$ (*L2-VARS ?f*$_c$ *?nm*))

$[\![$*struct-rewrite-expr ?P ?init*$_c'$ *?init*$_c$; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init*$_a$ *?init*$_c'$; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f*$_a$ *p*) (*?f*$_c$ *p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-array-C.guard-with-fresh-stack-ptr ?n ?init*$_a$ (*L2-VARS ?f*$_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n ?init*$_c$ (*L2-VARS ?f*$_c$ *?nm*))

$[\![\bigwedge s\ p.$ *?f*$_c$ *p* $\cdot$ *s ?*$\{\!|$ $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* $|\!\}$; *struct-rewrite-expr ?P ?init*$_c'$ *?init*$_c$; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init*$_a$ *?init*$_c'$; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f*$_a$ *p*) (*?f*$_c$ *p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-array-C.assume-with-fresh-stack-ptr ?n ?init*$_a$ (*L2-VARS ?f*$_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n ?init*$_c$ (*L2-VARS ?f*$_c$ *?nm*)).

- The derived introduction rules are recursively applied to the program.

  The first step is the one that was significantly extended to support addressable fields in open structures. The second and third step remained almost unchanged.

thm *heap-abs*

When thoroughly inspecting the rules $[\![abs\text{-}expr\ ?st\ ?X\ ?f1.0\ ?f1'; abs\text{-}expr\ ?st\ ?Y\ ?f2.0\ ?f2']\!] \Longrightarrow abs\text{-}expr\ ?st\ (\lambda s.\ ?X\ s \wedge ?Y\ s)\ (\lambda s.\ (?f1.0\ s,\ ?f2.0\ s))\ (\lambda s.\ (?f1'\ s,\ ?f2'\ s))$

$abs\text{-}expr\ ?st\ (\lambda\text{-}.\ True)\ (\lambda s.\ ?a)\ (\lambda s.\ ?a)$

$abs\text{-}expr\ ?st\ ?P\ ?a'\ ?a \Longrightarrow abs\text{-}guard\ ?st\ (\lambda s.\ ?P\ s \wedge ?a'\ s)\ ?a$

$abs\text{-}guard\ ?st\ (\lambda\text{-}.\ ?P)\ (\lambda\text{-}.\ ?P)$

$[\![abs\text{-}guard\ ?st\ ?G\ ?G'; abs\text{-}guard\ ?st\ ?H\ ?H']\!] \Longrightarrow abs\text{-}guard\ ?st\ (\lambda s.\ ?G\ s \wedge ?H\ s)\ (\lambda s.\ ?G'\ s \wedge ?H'\ s)$

$[\![struct\text{-}rewrite\text{-}modifies\ ?P\ ?b\ ?c; abs\text{-}guard\ ?st\ ?P'\ ?P; abs\text{-}modifies\ ?st\ ?Q\ ?a\ ?b]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ L2\text{-}modify\ ?a))\ (L2\text{-}modify\ ?c)$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?b\ ?c; abs\text{-}guard\ ?st\ ?P'\ ?P; abs\text{-}expr\ ?st\ ?Q\ ?a\ ?b]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ L2\text{-}gets\ ?a\ ?n))\ (L2\text{-}gets\ ?c\ ?n)$

$L2Tcorres\ ?st\ (L2\text{-}gets\ (\lambda\text{-}.\ ?a)\ ?n)\ (L2\text{-}gets\ (\lambda\text{-}.\ ?a)\ ?n)$

$[\![struct\text{-}rewrite\text{-}guard\ ?b\ ?c; abs\text{-}guard\ ?st\ ?a\ ?b]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}guard\ ?a)\ (L2\text{-}guard\ ?c)$

$[\![THIN\ (\bigwedge x.\ L2Tcorres\ ?st\ (?B'\ x)\ (?B\ x));\ THIN\ (\bigwedge r.\ struct\text{-}rewrite\text{-}expr\ (?G\ r)\ (?C'\ r)\ (?C\ r));\ THIN\ (\bigwedge r.\ abs\text{-}guard\ ?st\ (?G'\ r)\ (?G\ r));\ THIN\ (\bigwedge r.\ abs\text{-}expr\ ?st\ (?H\ r)\ (?C''\ r)\ (?C'\ r))]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?G'\ ?i\ s \wedge ?H\ ?i\ s))\ (\lambda\text{-}.\ L2\text{-}while\ ?C''\ (\lambda i.\ L2\text{-}seq\ (?B'\ i)\ (\lambda r.\ L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?G'\ r\ s \wedge ?H\ r\ s))\ (\lambda\text{-}.\ L2\text{-}gets\ (\lambda\text{-}.\ r)\ ?n)))\ ?i\ ?n))\ (L2\text{-}while\ ?C\ ?B\ ?i\ ?n)$

$abs\text{-}spec\ ?st\ ?P\ ?A\ ?C \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}seq\ (L2\text{-}guard\ ?P)\ (\lambda\text{-}.\ L2\text{-}spec\ ?A))\ (L2\text{-}spec\ ?C)$

$abs\text{-}assume\ ?st\ ?P\ ?A\ ?C \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}seq\ (L2\text{-}guard\ ?P)\ (\lambda\text{-}.\ L2\text{-}assume\ ?A))\ (L2\text{-}assume\ ?C)$

$abs\text{-}spec\ ?st\ (\lambda\text{-}.\ True)\ \{(a,\ b).\ ?C\}\ \{(a,\ b).\ ?C\}$

$[\![THIN\ (L2Tcorres\ ?st\ ?L\ ?L');\ THIN\ (L2Tcorres\ ?st\ ?R\ ?R');\ THIN\ (struct\text{-}rewrite\text{-}expr\ ?P\ ?C'\ ?C);\ THIN\ (abs\text{-}guard\ ?st\ ?P'\ ?P);\ THIN\ (abs\text{-}expr\ ?st\ ?Q\ ?C''\ ?C')]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ L2\text{-}condition\ ?C''\ ?L\ ?R))\ (L2\text{-}condition\ ?C\ ?L'\ ?R')$

$[\![THIN\ (L2Tcorres\ ?st\ ?L'\ ?L);\ THIN\ (\bigwedge r.\ L2Tcorres\ ?st\ (?R'\ r)\ (?R\ r))]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}seq\ ?L'\ ?R')\ (L2\text{-}seq\ ?L\ ?R)$

$[\![struct\text{-}rewrite\text{-}guard\ ?b\ ?c; abs\text{-}guard\ ?st\ ?a\ ?b; \bigwedge s\ s'.\ [\![?c\ s; s' = ?st\ s]\!] \Longrightarrow L2Tcorres\ ?st\ ?f\ ?g]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}guarded\ ?a\ ?f)\ (L2\text{-}guarded\ ?c\ ?g)$

$[\![THIN\ (L2Tcorres\ ?st\ ?L\ ?L');\ THIN\ (\bigwedge r.\ L2Tcorres\ ?st\ (?R\ r)\ (?R'\ r))]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}catch\ ?L\ ?R)\ (L2\text{-}catch\ ?L'\ ?R')$

$L2Tcorres\ ?st\ ?L\ ?L' \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}try\ ?L)\ (L2\text{-}try\ ?L')$

$L2Tcorres\ ?st\ (L2\text{-}unknown\ ?ns)\ (L2\text{-}unknown\ ?ns)$

$L2Tcorres\ ?st\ (L2\text{-}throw\ ?x\ ?n)\ (L2\text{-}throw\ ?x\ ?n)$

$(\bigwedge x\ y.\ L2Tcorres\ ?st\ (?P\ x\ y)\ (?P'\ x\ y)) \Longrightarrow L2Tcorres\ ?st\ (case\ ?a\ of\ (x,\ y) \Rightarrow ?P\ x\ y)\ (case\ ?a\ of\ (x,\ y) \Rightarrow ?P'\ x\ y)$

⟦*THIN* (*L2Tcorres ?st ?L ?L'*); *THIN* (*L2Tcorres ?st ?R ?R'*)⟧ ⟹ *L2Tcorres ?st* (*L2-seq ?L* (λ-. *?R*)) (*L2-seq ?L'* (λ-. *?R'*))

(⋀*a b*. *abs-expr ?st* (*?P a b*) (*?A a b*) (*?C a b*)) ⟹ *abs-expr ?st* (*case ?r of* (*a*, *b*) ⇒ *?P a b*) (*case ?r of* (*a*, *b*) ⇒ *?A a b*) (*case ?r of* (*a*, *b*) ⇒ *?C a b*)

(⋀*a b*. *abs-guard ?st* (*?A a b*) (*?C a b*)) ⟹ *abs-guard ?st* (*case ?r of* (*a*, *b*) ⇒ *?A a b*) (*case ?r of* (*a*, *b*) ⇒ *?C a b*)

L2Tcorres ?st L2-fail L2-fail

abs-expr id (λ-. *True*) *?A ?A*

abs-expr ?st ?P ?A ?C ⟹ *abs-expr ?st ?P* (λ*s r*. *?A s*) (λ*s r*. *?C s*)

abs-modifies id (λ-. *True*) *?A ?A*

L2Tcorres id ?A ?C ⟹ *L2Tcorres ?st* (*exec-concrete ?st* (*L2-call ?A ?emb ?ns*)) (*L2-call ?C ?emb ?ns*)

L2Tcorres id ?A ?C ⟹ *L2Tcorres ?st* (*exec-concrete ?st* (*L2-call-L1 ?arg-xf ?gs ?ret-xf ?A*)) (*L2-call-L1 ?arg-xf ?gs ?ret-xf ?C*)

L2Tcorres ?st ?A ?C ⟹ *L2Tcorres id* (*exec-abstract ?st* (*L2-call ?A ?emb ?ns*)) (*L2-call ?C ?emb ?ns*)

L2Tcorres ?st ?A ?C ⟹ *L2Tcorres ?st* (*L2-call ?A ?emb ?ns*) (*L2-call ?C ?emb ?ns*)

⟦*typ-heap-simulation ?st ?getter ?setter ?vgetter ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?x' ?x*⟧ ⟹ *abs-guard ?st* (λ*s*. *?P s* ∧ *?vgetter s* (*?x' s*)) (λ*s*. *c-guard* (*?x s*))

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?x' ?x*⟧ ⟹ *abs-expr ?st* (λ*s*. *?P s* ∧ *?v s* (*?x' s*)) (λ*s*. *?r s* (*?x' s*)) (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*?x s*))

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?Pb ?b' ?b*; ⋀*v*. *abs-expr ?st ?Pc* (*?c' v*) (*?c v*)⟧ ⟹ *abs-modifies ?st* (λ*s*. *?Pb s* ∧ *?Pc s* ∧ *?v s* (*?b' s*)) (λ*s*. *?w* (*?b' s*) (λ-. *?c'* (*?r s* (*?b' s*)) *s*) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?b s*) (*?c* (*heap-lift--h-val* (*hrs-mem* (*?t-hrs s*)) (*?b s*)) *s*))) *s*)

struct-rewrite-expr ?P ?a' ?a ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?a' s*) *?a*

struct-rewrite-guard (λ-. *?P*) (λ-. *?P*)

⟦*struct-rewrite-guard ?b' ?b*; *struct-rewrite-guard ?a' ?a*⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?a' s* ∧ *?b' s*) (λ*s*. *?a s* ∧ *?b s*)

(⋀*a b*. *struct-rewrite-guard* (*?A a b*) (*?C a b*)) ⟹ *struct-rewrite-guard* (*case ?r of* (*a*, *b*) ⇒ *?A a b*) (*case ?r of* (*a*, *b*) ⇒ *?C a b*)

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-guard ?Q* (λ*s*. *c-guard* (*?p' s*))⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?Q s*) (λ*s*. *c-guard* (*PTR*(*?'f*) &(*?p s*→*?field-name*)))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-guard ?Q* (λ*s*. *c-guard* (*?p' s*))⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?Q s* ∧ *0* ≤ *?k* ∧ *nat ?k* <

$CARD(?'n))$ $(\lambda s.$ *c-guard* $(PTR\text{-}COERCE(?'f[?'n] \to ?'f)$ $(PTR(?'f[?'n])$ $\&(?p\ s \to ?field\text{-}name))$ $+_p\ ?k))$

struct-rewrite-expr $(\lambda\text{-}.\ True)$ $(\lambda\text{-}.\ ?a)$ $(\lambda\text{-}.\ ?a)$

struct-rewrite-expr $?P\ ?A\ ?C \Longrightarrow$ *struct-rewrite-expr* $?P$ $(\lambda s\ \text{-}.\ ?A\ s)$ $(\lambda s\ \text{-}.\ ?C\ s)$

$(\bigwedge a\ b.$ *struct-rewrite-expr* $(?P\ a\ b)$ $(?A\ a\ b)$ $(?C\ a\ b)) \Longrightarrow$ *struct-rewrite-expr* $(case\ ?r\ of\ (a,\ b) \Rightarrow ?P\ a\ b)$ $(case\ ?r\ of\ (a,\ b) \Rightarrow ?A\ a\ b)$ $(case\ ?r\ of\ (a,\ b) \Rightarrow ?C\ a\ b)$

struct-rewrite-expr $(\lambda\text{-}.\ True)$ $(\lambda s.$ *h-val* $(?h\ s)$ $(?p\ s))$ $(\lambda s.$ *h-val* $(?h\ s)$ $(?p\ s))$

struct-rewrite-expr $?P\ ?a\ ?c \Longrightarrow$ *struct-rewrite-expr* $?P$ $(\lambda s.$ *h-val* $(?h\ s)$ $(?a\ s))$ $(\lambda s.$ *h-val* $(?h\ s)$ $(?c\ s))$

$[\![$*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-expr ?Q ?a* $(\lambda s.$ *h-val* $(hrs\text{-}mem$ $(?t\text{-}hrs\ s))$ $(?p'\ s))]\!] \Longrightarrow$ *struct-rewrite-expr* $(\lambda s.\ ?P\ s \wedge ?Q\ s)$ $(\lambda s.$ *?field-getter* $(?a\ s))$ $(\lambda s.$ *h-val* $(hrs\text{-}mem$ $(?t\text{-}hrs\ s))$ $(PTR(?'f)$ $\&(?p\ s \to ?field\text{-}name)))$

$[\![$*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?a ?c*$]\!] \Longrightarrow$ *abs-expr ?st ?P* $(\lambda s.$ *?field-getter* $(?a\ s))$ $(\lambda s.$ *?field-getter* $(?c\ s))$

$[\![$*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-expr ?Q ?a* $(\lambda s.$ *h-val* $(hrs\text{-}mem$ $(?t\text{-}hrs\ s))$ $(?p'\ s))]\!] \Longrightarrow$ *struct-rewrite-expr* $(\lambda s.\ ?P\ s \wedge ?Q\ s \wedge 0 \leq ?k \wedge nat\ ?k < CARD(?'n))$ $(\lambda s.$ *?field-getter* $(?a\ s).[nat\ ?k])$ $(\lambda s.$ *h-val* $(hrs\text{-}mem$ $(?t\text{-}hrs\ s))$ $(PTR\text{-}COERCE(?'f[?'n] \to ?'f)$ $(PTR(?'f[?'n])$ $\&(?p\ s \to ?field\text{-}name))$ $+_p\ ?k))$

$[\![$*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-expr ?Q ?a* $(\lambda s.$ *h-val* $(hrs\text{-}mem$ $(?t\text{-}hrs\ s))$ $(?p'\ s))]\!] \Longrightarrow$ *struct-rewrite-expr* $(\lambda s.\ ?P\ s \wedge ?Q\ s \wedge 0 \leq ?k \wedge nat\ ?k < CARD(?'n))$ $(\lambda s.$ *?field-getter* $(?a\ s).[nat\ ?k])$ $(\lambda s.$ *h-val* $(hrs\text{-}mem$ $(?t\text{-}hrs\ s))$ $(PTR(?'f)$ $\&(?p\ s \to ?field\text{-}name)$ $+_p\ ?k))$

struct-rewrite-modifies $(\lambda\text{-}.\ True)$ $?A\ ?A$

$[\![$*typ-heap-simulation ?st ?getter ?setter ?vgetter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-expr ?Q ?v' ?v*$]\!] \Longrightarrow$ *struct-rewrite-modifies* $(\lambda s.\ ?P\ s \wedge ?Q\ s)$ $(\lambda s.$ *?t-hrs-update* $(hrs\text{-}mem\text{-}update$ $(heap\text{-}update$ $(?p'\ s)$ $(?v'\ s)))\ s)$ $(\lambda s.$ *?t-hrs-update* $(hrs\text{-}mem\text{-}update$ $(heap\text{-}update$ $(?p\ s)$ $(?v\ s)))$ $s)$

heap-lift--wrap-h-val $(heap\text{-}lift\text{--}h\text{-}val\ ?s\ ?p)$ $(h\text{-}val\ ?s\ ?p)$

heap-lift--wrap-h-val $(h\text{-}val\ ?s\ (PTR(?'a)$ $\&(?p \to ?f)))$ $(h\text{-}val\ ?s\ (PTR(?'a)$ $\&(?p \to ?f)))$

heap-lift--wrap-h-val $(h\text{-}val\ ?s\ (PTR\text{-}COERCE(?'b \to ?'a)$ $?p\ +_p\ ?k))$ $(h\text{-}val\ ?s\ (PTR\text{-}COERCE(?'b \to ?'a)$ $?p\ +_p\ ?k))$

$[\![$*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p' ?p*; *struct-rewrite-expr ?Q ?f' ?f*; $\bigwedge s.$ *heap-lift--wrap-h-val* $(?h\text{-}val\text{-}p'\ s)$ $(h\text{-}val\ (hrs\text{-}mem\ (?t\text{-}hrs\ s))\ (?p'\ s))$; *struct-rewrite-modifies ?R*

(λs. ?t-hrs-update (hrs-mem-update (heap-update (?p'' s) (?u s (?field-setter (?f' s))))) s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (?p' s) (?field-setter (?f' s) (?h-val-p' s)))) s); struct-rewrite-guard ?S (λs. c-guard (?p' s))⟧ ⟹ struct-rewrite-modifies (λs. ?P s ∧ ?Q s ∧ ?R s ∧ ?S s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (?p'' s) (?u s (?field-setter (?f' s))))) s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (PTR(?'f) &(?p s→?field-name)) (?f s (h-val (hrs-mem (?t-hrs s)) (PTR(?'f) &(?p s→?field-name)))))) s)

⟦valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update; struct-rewrite-expr ?P ?p' ?p; struct-rewrite-expr ?Q ?f' ?f; ⋀s. heap-lift--wrap-h-val (?h-val-p' s) (h-val (hrs-mem (?t-hrs s)) (?p' s)); struct-rewrite-modifies ?R (λs. ?t-hrs-update (hrs-mem-update (heap-update (?p'' s) (?u s (?field-setter (λa. Arrays.update a (nat ?k) (?f' s (a.[nat ?k])))))))) s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (?p' s) (?field-setter (λa. Arrays.update a (nat ?k) (?f' s (a.[nat ?k])))) (?h-val-p' s)))) s); struct-rewrite-guard ?S (λs. c-guard (?p' s))⟧ ⟹ struct-rewrite-modifies (λs. ?P s ∧ ?Q s ∧ ?R s ∧ ?S s ∧ 0 ≤ ?k ∧ nat ?k < CARD(?'n)) (λs. ?t-hrs-update (hrs-mem-update (heap-update (?p'' s) (?u s (?field-setter (λa. Arrays.update a (nat ?k) (?f' s (a.[nat ?k])))))))) s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (PTR-COERCE(?'f[?'n] → ?'f) (PTR(?'f[?'n]) &(?p s→?field-name)) +$_p$?k) (?f s (h-val (hrs-mem (?t-hrs s)) (PTR-COERCE(?'f[?'n] → ?'f) (PTR(?'f[?'n]) &(?p s→?field-name)) +$_p$?k))))) s)

valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter ⟹ abs-expr ?st (λ-. True) ?new-getter ?old-getter

⟦valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter; ⋀old. abs-expr ?st (?P old) (?v old) (?v' old)⟧ ⟹ abs-modifies ?st (λs. ∀ old. ?P old s) (λs. ?new-setter (λold. ?v old s) s) (λs. ?old-setter (λold. ?v' old s) s) s)

valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter ⟹ struct-rewrite-expr (λ-. True) ?old-getter ?old-getter

⟦valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter; ⋀old. struct-rewrite-expr (?P old) (?v' old) (?v old)⟧ ⟹ struct-rewrite-modifies (λs. ∀ old. ?P old s) (λs. ?old-setter (λold. ?v' old s) s) (λs. ?old-setter (λold. ?v old s) s)

abs-spec ?st (λ-. True) {(a, b). b may-not-modify-globals a} {(a, b). b may-not-modify-globals a}

⟦valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter; abs-spec ?st (λ-. True) {(a, b). ?C a b} {(a, b). ?C' a b}⟧ ⟹ abs-spec ?st (λ-. True) {(a, b). mex (λx. ?C (?new-setter (λ-. x) a) b)} {(a, b). mex (λx. ?C' (?old-setter (λ-. x) a) b)}

⟦typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update; abs-expr ?st ?P ?x' (λs. PTR-COERCE(?'a[?'b] → ?'a) (?x s))⟧ ⟹ abs-guard ?st (λs. ?P s ∧ (∀ a∈set (array-addrs (?x' s) CARD(?'b)). ?v s a)) (λs. c-guard (?x s))

⟦typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update; abs-expr ?st ?Pb ?b' ?b; abs-expr ?st ?Pn ?n' ?n; abs-expr ?st ?Pv ?y' ?y⟧ ⟹ abs-modifies ?st

(λs. ?Pb s ∧ ?Pn s ∧ ?Pv s ∧ ?n′ s < CARD(?′b) ∧ (∀ ptr∈set (array-addrs (PTR-COERCE(?′a[?′b] → ?′a) (?b′ s)) CARD(?′b)). ?v s ptr)) (λs. ?w (PTR-COERCE(?′a[?′b] → ?′a) (?b′ s) +$_p$ int (?n′ s)) (λv. ?y′ s) s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (?b s) (Arrays.update (h-val (hrs-mem (?t-hrs s)) (?b s)) (?n s) (?y s)))) s)

⟦typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update; abs-expr ?st ?Pb ?b′ ?b; abs-expr ?st ?Pn ?n′ ?n⟧ ⟹ abs-expr ?st (λs. ?Pb s ∧ ?Pn s ∧ ?n′ s < CARD(?′b) ∧ ?v s (PTR-COERCE(?′a[?′b] → ?′a) (?b′ s) +$_p$ int (?n′ s))) (λs. ?r s (PTR-COERCE(?′a[?′b] → ?′a) (?b′ s) +$_p$ int (?n′ s))) (λs. h-val (hrs-mem (?t-hrs s)) (?b s).[?n s])

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(32 word) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-w32-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(32 word) t (PTR-COERCE(32 signed word → 32 word) ?p) ∧ ?P t)) (λ-. L2-modify (λs. heap-w32-update (λh. h(PTR-COERCE(32 signed word → 32 word) ?p := UCAST(32 signed → 32) (?a s))) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(8 word) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-w8-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(8 word) t (PTR-COERCE(8 signed word → 8 word) ?p) ∧ ?P t)) (λ-. L2-modify (λs. heap-w8-update (λh. h(PTR-COERCE(8 signed word → 8 word) ?p := UCAST(8 signed → 8) (?a s))) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(32 word ptr) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-w32′ptr-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unit ptr) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-unit′ptr-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(closed-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-closed-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unpacked-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-unpacked-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq

(*L2-guard* (λ*t*. *IS-VALID*(*arr-struct-C*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-arr-struct-C-update* (λ*h*. *h*(*?p* := *?a s*)) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *valid-array-base.valid-array* (λ*h*. *IS-VALID*(*unpacked-C*) *h*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *unpacked-C.heap-array-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *valid-array-base.valid-array* (*valid-array-base.valid-array* (λ*h*. *IS-VALID*(*unpacked-C*) *h*)) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *unpacked-C.outer.heap-array-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*two-dimensional-C*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-two-dimensional-C-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*data-struct1-C*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-data-struct1-C-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*data-struct2-C*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-data-struct2-C-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*inner-C*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-inner-C-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *valid-array-base.valid-array* (λ*h*. *IS-VALID*(*inner-C*) *h*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *inner-C.heap-array-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*outer-array-C*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-outer-array-C-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *valid-array-base.valid-array* (λ*h*. *IS-VALID*(*data-C*) *h*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *data-C.heap-array-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*data-array-C*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-data-array-C-map* *?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE* *?p* *?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*outer-C*) *t* *?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-outer-C-map*

?p (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)
abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*other-C*) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-other-C-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)
abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *valid-array-base.valid-array* (λ*h*. *IS-VALID*(*unpacked-C*) *h*) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *unpacked-C.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)
abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*array-C*) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *heap-array-C-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)
⟦*struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-w32.guard-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))
⟦⋀*s p*. $?f_c$ *p* ⋆ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-w32.assume-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))
⟦*struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-w8.guard-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))
⟦⋀*s p*. $?f_c$ *p* ⋆ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-w8.assume-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))
⟦*struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-w32'ptr.guard-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))
⟦⋀*s p*. $?f_c$ *p* ⋆ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P* $?init_c'$ $?init_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* ($?f_a$ *p*) ($?f_c$ *p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-w32'ptr.assume-with-fresh-stack-ptr ?n* $?init_a$ (*L2-VARS* $?f_a$ *?nm*))) (*globals.with-fresh-stack-ptr ?n* $?init_c$ (*L2-VARS* $?f_c$ *?nm*))

$[\![$*struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ*-. heap-unit'ptr.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

$[\![\bigwedge$*s p. ?f$_c$ p* $\cdot$ *s ?{* λ*r. globals.typing.unchanged-typing-on* $\mathcal{S}$ *s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ*-. heap-unit'ptr.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

$[\![$*struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ*-. heap-data-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

$[\![\bigwedge$*s p. ?f$_c$ p* $\cdot$ *s ?{* λ*r. globals.typing.unchanged-typing-on* $\mathcal{S}$ *s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ*-. heap-data-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

$[\![$*struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ*-. heap-closed-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

$[\![\bigwedge$*s p. ?f$_c$ p* $\cdot$ *s ?{* λ*r. globals.typing.unchanged-typing-on* $\mathcal{S}$ *s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ*-. heap-closed-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

$[\![$*struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ*-. heap-unpacked-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

$[\![\bigwedge$*s p. ?f$_c$ p* $\cdot$ *s ?{* λ*r. globals.typing.unchanged-typing-on* $\mathcal{S}$ *s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ*-. heap-unpacked-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-arr-struct-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ∙ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-arr-struct-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*unpacked-C*[*3*][*2*]) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *unpacked-C.outer.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-unpacked-C'array-3'array-2.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ∙ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-unpacked-C'array-3'array-2.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-two-dimensional-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ∙ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-two-dimensional-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-data-struct1-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge s\ p.\ ?f_c\ p \cdot s$?{ $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s$ }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-data-struct1-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-data-struct2-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge s\ p.\ ?f_c\ p \cdot s$?{ $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s$ }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-data-struct2-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-inner-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge s\ p.\ ?f_c\ p \cdot s$?{ $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s$ }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-inner-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

abs-expr lift-global-heap ?P ?a ?c $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda t.$ *IS-VALID*(*inner-C*[5]) *t ?p* $\wedge$ *?P t*)) (λ-. *L2-modify* ($\lambda s.$ *inner-C.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-inner-C′array-5.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge s\ p.\ ?f_c\ p \cdot s$?{ $\lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s$ }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* ($\bigwedge p.$ *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* ($\lambda s.$ *?P′ s* $\wedge$ *?Q s*)) (λ-. *heap-inner-C′array-5.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-outer-array-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ⋆ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-outer-array-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*data-C*[*10*]) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *data-C.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-data-C'array-10.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ⋆ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-data-C'array-10.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-data-array-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ⋆ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-data-array-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-outer-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ⋆ *s* ?{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr*

?P ?init$_c$' ?init$_c$; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-outer-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-other-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge$*s p. ?f$_c$ p* $\cdot$ *s ?*{ λ*r. globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-other-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

abs-expr lift-global-heap ?P ?a ?c $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t. IS-VALID*(*unpacked-C*[*2*]) *t ?p* $\wedge$ *?P t*)) (λ-. *L2-modify* (λ*s. unpacked-C.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-unpacked-C'array-2.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge$*s p. ?f$_c$ p* $\cdot$ *s ?*{ λ*r. globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-unpacked-C'array-2.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-array-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge$*s p. ?f$_c$ p* $\cdot$ *s ?*{ λ*r. globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-array-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*)) and keeping in mind that they are used by recursively applying them as introduction rules

to a concrete program C and a schematic variable for the abstract program A one can see the following strategy:

- First some normalisation of expressions involving pointers is performed on the concrete program C. In particular an update to a dereferenced pointer $\&(p{\rightarrow}f)$ by a value v is transformed to an update on p, where first the value of p is fetched from memory and then field f of this value is updated by v and the resulting compound value is put back to memory.

- Only after the normalisation on C, the actual translation to an abstract A on the split heap is performed.

  The normalisation step is guided by predicates *struct-rewrite-guard*, *struct-rewrite-expr* and *struct-rewrite-modifies*. After that normalisation the atomic building block for the simulation is provided by an instance of *typ-heap-simulation*, which tells how a lookup and update of a pointer of a given type in the byte heap is simulated in the split heap.

  The proofs of the necessary instances of *typ-heap-simulation* are provided once and forall in the initialisation phase of **autocorres** or **init−autocorres**. They are provided for atomic and shared heaps as well as derived heaps.

Proving *typ-heap-simulation*

As stated before this step was extended quite excessively and quite some infrastructure of both theorems as well as ML code were built to support it.

Typed vs. Untyped Dialects/ Records vs. Byte-Lists First some general remarks on the general concepts and setup. The UMM was designed in the spirit to take advantage of HOL type and class inference to support reasoning about C-types and pointers. As a consequence some of the definitions and theorems are rather subtle and somehow live an the edge of what can be typed and expressed in the HOL type system. Sometimes the essence or the limitations of a theorem only become 'visible' when also looking at the types of expressions, especially the pointer types. Unfortunately writing about these concepts is rather ambiguous with respect to the notion type. 'Type' might refer to the original C-structure in the C program, this type is related to an abstract HOL-type, which is a record. This record is an instance of various type-classes: *c-type*, *mem-type*, *xmem-type*. For types of *c-type* we define overloaded *typ-info-t*, which associates a HOL-term describing the type (referred to as type-description or type-information). Moreover,

pointers represented in HOL carry the HOL-type they point to as phantom type.

An illustration for the subtle typing is the two terms $\&(p{\rightarrow}f) \neq \&(q{\rightarrow}g)$ vs. $p \neq q \Longrightarrow \&(p{\rightarrow}f) \neq \&(q{\rightarrow}g)$. Whereas in first term the pointers p and q have a different type, in the second term they both have the same type as the inequation in the precondition also makes the types equal. This might not always be the intended meaning. When the type of the pointer is irrelevant one might resort to plain addresses instead of pointers *ptr-val p* $=$ *ptr-val q* $\Longrightarrow \&(p{\rightarrow}f) = \&(q{\rightarrow}g)$. This general theme occurs in various places. There often is a typed variant of a concept and also an untyped variant, based on addresses or byte lists, like p vs. *ptr-val p*. Both are closely related. The typed variant somehow reflects the 'API' of the concept and is more abstract, whereas during a proof one actually resort to the untyped variant to gain flexibility. These two views on a concepts leads to a duplication of lemmas. Moreover, a lemma might not immediately be available in the form one expects, but can be 'easily' derived from some related lemmas.

The central UMM HOL-datatype to reflect C-types into HOL terms is ($'a$, $'b$) *typ-desc* and is defined mutually recursive with ($'a$, $'b$) *typ-struct*. It is basically a tree describing the nested structure of an aggregate type. Field names, alignment and size information is encoded, and lookup and update of sub-fields can be derived from this information.

The most prominent instances of this type are the typed variant $'a$ *xtyp-info* vs. the untyped variant *typ-uinfo*. The former can be transformed to the latter by *export-uinfo*. As with pointer types the type variable in $'a$ *xtyp-info* denotes the abstract HOL-C-type that is described by the type information. So to derive a field-lookup or field-update on HOL-types from the type information one needs to resort to the typed variant. Exporting the type information via *export-uinfo* maintains the shape of the tree, but removes all the HOL-C-type dependent components. The resulting *typ-uinfo* has two main use cases:

- It can be seen as a type-tag identifying a C-type. It can be used to relate two C-types on the HOL-term level, e.g. ask if they are equal or if one is contained in the other.
- It can be used to normalise a byte-list representing a value of that type. Normalisation means that all padding bytes in the byte-list are set to zero.

  For each C-type the C-Parser also generates the type information which can be accessed via the overloaded function: *typ-info-t TYPE*($'a$).

  The main use cases for *typ-info-t TYPE*($'a$) are

- Provide lenses for fields (access / update) on the abstract value (record),

via *access-ti*, *update-ti* and *field-lookup* The main use cases for *typ-uinfo-t* $TYPE('a)$ are

- comparison of type descriptors: equality and order $s \leq_\tau t$
- normalisation of byte-lists *norm-tu*, in particular setting all padding bytes to zero.

thm *sub-typ-def*
thm *typ-tag-le-def*

So *typ-info-t* is more related to the abstract view on a value (as a HOL record), whereas *typ-uinfo-t* is more related to the byte-list encoding of the value. This duality somehow also reflects the dual nature of types in C. On the one hand C is a statically typed language and on the other hand it allows to break the abstraction and switch to a low-level byte oriented view.

context
fixes *p*::*'a*::*c-type ptr*
fixes *f*::*qualified-field-name*
fixes *t*::*'a xtyp-info*
fixes *n*:: *nat*
begin

A central function on both typed and untyped typ-information is the function *field-lookup* which retrieves the type-information for a field of a type. A common application of this function is to state some property on a dereferenced pointer: $PTR('b)$ $\&(p{\rightarrow}f)$. Note that p is an pointer to an $'a$: p. A typical precondition is to retrieve the type-information for field f from the type information of $'a$: *field-lookup* (*typ-info-t* $TYPE('a)$) $f\ 0 =$ *Some* (t, n)

Note that the retrieved type information t is still tagged with $'a$. The number n is the offset of the selected field. Typically we want to relate t to the type of the selected field $'b$ (which only happens to be the same $'a$ in case the field is the empty list). The relation can be established via *export-uinfo*, namely *export-uinfo* $t =$ *export-uinfo* (*typ-info-t* $TYPE('b)$). Note that directly equating t to *typ-info-t* $TYPE('b)$ is not even a well-typed expression in HOL, as $'a$ is not equal to $'b$.

The right hand side abbreviates to constant *typ-uinfo-t* $TYPE('b)$.

end

A concrete example might give some further insight to the relation of *typ-info-t* and *typ-uinfo-t* and how the type-information is constructed. Let us consider the field $''inner\text{-}C''$ of *outer-C*, which selects a field of *inner-C*. We can retrieve the information for the field:

lemma *field-lookup* (*typ-info-t* $TYPE(outer\text{-}C)$) $[''inner\text{-}C'']$ $0 =$

Some (*adjust-ti* (*typ-info-t TYPE*(*inner-C*)) *outer-C.inner-C* (*inner-C-update* ∘ (λ*x* -. *x*)), *0*)
by (*simp*)

Note that the retrieved type information is constructed from the nested type information for *inner-C*, by adjusting it. Adjusting means that we say how we can lookup and update the sub-field of the record *outer-C*. This adjustment only affects the typed-view of the type information. Exporting the type information collapses to the adjusted inner type:

lemma *export-uinfo* (*adjust-ti* (*typ-info-t TYPE*(*inner-C*)) *outer-C.inner-C* (*inner-C-update* ∘ (λ*x* -. *x*)))
= *export-uinfo* (*typ-info-t TYPE*(*inner-C*))
by *simp*

In the realm of 'lenses", *adjust-ti* can be viewed as a form of composition of lenses. A lense for an inner type is transformed to a lense on the outer type. The lense assotiated to type information is captured in *access-ti*, for the 'lense-lookup' aka. 'lense-get' part, and *update-ti* for the 'lense-update' aka 'lense-put'.

Function *field-lookup* is the essential building block to relate field names e.g. as part of dereferencing a pointer to their abstracts operations on the associated HOL-type. One can convert between the typed and untyped version:

thm *field-lookup-export-uinfo-Some-rev*
thm *field-lookup-export-uinfo-Some*

Here are some closely related, mostly derived concepts around *field-lookup*:

- *field-lookup*, *field-ti*, *field-offset*, *field-of*
- *td-set*, *sub-typ* ($s \leq_\tau t$
- Family of field name functions: *all-field-names*, *TypHeap.field-names*, *field-names-u*, *field-names-no-padding*, *all-field-names-no-padding*

  Often lemmas might not be available for all variants, but via some simple indirections, via definitions or conversion lemmas. In an Isar-proof, **sledgehammer** can often help to find the connections.

thm *td-set-field-lookupD*
thm *td-set-field-lookup*
thm *all-field-names-union-field-names-export-uinfo-conv*
thm *set-field-names-all-field-names-conv*
thm *field-names-u-field-names-export-uinfo-conv*(*1*)
thm *set-field-names-no-padding-all-field-names-no-padding-conv*
thm *all-field-names-no-padding-typ-uinfo-t-conv*

Doing the Proof The fundamental goal is to derive interpretations *typ-heap-simulation* for every relevant type. It states that the virtual heap for a type as obtained from *lifted-globals* simulates the original UMM heap in *globals*. Both states are connected by the abstraction function *lift-global-heap* which is the abstract function *st* in *typ-heap-simulation*. To facilitate the construction of the proofs we introduce intermediate helper locale *typ-heap-simulation-open-types* and make use of the infrastructure of lenses and scenes that we mentioned before.

To optimize the construction (in particular to minimise the number of heaps we have to put into into *lifted-globals*) we distinguish three main cases:

- A structure is closed and has no addressable fields: ⟦*open-types ?𝒯*; *heap-typing-simulation ?𝒯 ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd ?l*; *lense ?R ?u*; *map-of ?𝒯* (*typ-uinfo-t TYPE*(*?'a*)) = *None*; ⋀*p x s*. *open-types.ptr-valid ?𝒯* (*hrs-htd* (*?hrs s*)) *p* ⟹ *?l* (*?hrs-upd* (*open-types.write-dedicated-heap ?𝒯 p x*) *s*) = *?u* (*upd-fun p* (λ*old*. *merge-ti-list* (*map snd* (*open-types.addressable-fields ?𝒯 TYPE*(*?'a*))) *old x*)) (*?l s*); ⋀*x d h*. *?heap-typing-upd d* (*?u x h*) = *?u x* (*?heap-typing-upd d h*); ⋀*p s*. ∀ *a*∈*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*?hrs s*)) (*PTR*(*stack-byte*) *a*) ⟹ *?R* (*?l s*) *p* = *ZERO*(*?'a*); ⋀*h p*. *?V h p* = *open-types.ptr-valid ?𝒯* (*?heap-typing h*) *p*; ⋀*p f h*. *?W p f h* = *?u* (λ*h'*. *h'*(*p* := *f* (*h' p*))) *h*⟧ ⟹ *typ-heap-simulation-open-types ?𝒯 ?l ?R ?W ?V ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd* In this case there is only one relevant heap in *lifted-globals*. No overlay of a dedicated heap with some common heaps is necessary.

- A structure is completely open, meaning that all fields are addressable: ⟦*open-types ?𝒯*; *heap-typing-simulation ?𝒯 ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd ?l*; *map-of ?𝒯* (*typ-uinfo-t TYPE*(*?'a*)) = *Some ?fs*; *length ?rs* = *length ?fs*; *length ?ws* = *length ?fs*; *list-all* (λ(*f*, *r*, *w*). *open-types.typ-heap-simulation-of-field ?𝒯 ?l ?hrs ?hrs-upd ?heap-typing-upd f r w*) (*zip ?fs* (*zip ?rs ?ws*)); *distinct-prop* (λ(*f1*, *w1*) (*f2*, *w2*). *disj-fn f1 f2* ⟶ *pointer-writer-disjnt-eq w1 w2*) (*zip ?fs ?ws*); ⋀*a b*. *fold* (λ*x*. *merge-ti* (*the* (*field-ti TYPE*(*?'a*) *x*)) *a*) *?fs b* = *a*; ⋀*h p*. *?V h p* = *open-types.ptr-valid ?𝒯* (*?heap-typing h*) *p*; ⋀*h p x*. *?R h p* = *fold* (λ*r*. *r h p*) *?rs x*; ⋀*p f h*. *?W p f h* = *fold* (λ*w*. *w p* (*f* (*?R h p*))) *?ws h*⟧ ⟹ *typ-heap-simulation-open-types ?𝒯 ?l ?R ?W ?V ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd* ∧ (∀ *w*. (∀ *x*. *list-all* (λ*w'*. *pointer-writer-disjnt* (λ*p*. *w' p x*) *w*) *?ws*) ⟶ (∀ *f*. *pointer-writer-disjnt* (λ*p*. *?W p f*) *w*)) ∧ (∀ *w p*. (∀ *x*. *list-all* (λ*w'*. *w' p x* ∘ *w* = *w* ∘ *w' p x*) *?ws*) ⟶ (∀ *f*. *?W p f* ∘ *w* = *w* ∘ *?W p f*)). In this case we do not need a dedicated heap. The structure is described by a overlay of common heaps.

- A structure is partially open, some fields are addressable and some not:

⟦*open-types ?𝒯*; *heap-typing-simulation ?𝒯 ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd ?l*; *map-of ?𝒯* (*typ-uinfo-t TYPE*(*?'a*)) = *Some ?fs*; *lense ?g ?u*; *length ?rs* = *length ?fs*; *length ?ws* = *length ?fs*; *list-all* (λ(*f*, *r*, *w*). *open-types.typ-heap-simulation-of-field ?𝒯 ?l ?hrs ?hrs-upd ?heap-typing-upd f r w*) (*zip ?fs* (*zip ?rs ?ws*)); *distinct-prop* (λ(*f1*, *w1*) (*f2*, *w2*). *disj-fn f1 f2* ⟶ *pointer-writer-disjnt-eq w1 w2*) (*zip ?fs ?ws*); *list-all* (λ*w*. ∀ *p a f*. *w p a* ∘ *?u f* = *?u f* ∘ *w p a*) *?ws*; ⋀*p x s*. *open-types.ptr-valid ?𝒯* (*hrs-htd* (*?hrs s*)) *p* ⟹ *?l* (*?hrs-upd* (*open-types.write-dedicated-heap ?𝒯 p x*) *s*) = *?u* (*upd-fun p* (λ*old*. *merge-ti-list* (*map snd* (*open-types.addressable-fields ?𝒯 TYPE*(*?'a*))) *old x*)) (*?l s*); ⋀*p s*. ∀ *a*∈*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*?hrs s*)) (*PTR*(*stack-byte*) *a*) ⟹ *?g* (*?l s*) *p* = *ZERO*(*?'a*); ⋀*x d h*. *?heap-typing-upd d* (*?u x h*) = *?u x* (*?heap-typing-upd d h*); ⋀*h p*. *?V h p* = *open-types.ptr-valid ?𝒯* (*?heap-typing h*) *p*; ⋀*h p*. *?R h p* = *fold* (λ*r*. *r h p*) *?rs* (*?g h p*); ⋀*p f h*. *?W p f h* = *fold* (λ*w*. *w p* (*f* (*?R h p*))) *?ws* (*?u* (*upd-fun p* (λ*old*. *merge-ti-list* (*map snd* (*open-types.addressable-fields ?𝒯 TYPE*(*?'a*))) *old* (*f* (*?R h p*)))) *h*)⟧ ⟹ *typ-heap-simulation-open-types ?𝒯 ?l ?R ?W ?V ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd* ∧ *pointer-lense ?g* (λ*p f*. *?u* (*upd-fun p f*)) ∧ (∀ *w*. (∀ *x*. *list-all* (λ*w'*. *pointer-writer-disjnt* (λ*p*. *w' p x*) *w*) *?ws*) ⟶ (∀ *x*. *pointer-writer-disjnt* (λ*p*. *?u* (*upd-fun p* (λ-. *x*))) *w*) ⟶ (∀ *f*. *pointer-writer-disjnt* (λ*p*. *?W p f*) *w*)) ∧ (∀ *w p*. (∀ *x*. *list-all* (λ*w'*. *w' p x* ∘ *w* = *w* ∘ *w' p x*) *?ws*) ⟶ (∀ *x*. *?u* (*upd-fun p* (λ-. *x*)) ∘ *w* = *w* ∘ *?u* (*upd-fun p* (λ-. *x*))) ⟶ (∀ *f*. *?W p f* ∘ *w* = *w* ∘ *?W p f*)). In that case we need an overlay of a dedicated heap and some common heaps.

On an intuitive abstract level the lemmas and proof argue about composing field-level lookup and updates within some heap(s) to lookup and updates of the complete compound structure. We want to argue that lookup and update in the UMM heap is simulated by the overlayed updates of a dedicated heap and some common heaps in the split heap. To argue about different fields of a structure we build on the idea of scenes and also extend it to reads. Note that the general problem is that a structure is represented as a HOL record and each field has a distinct type. Because of the limitations of the HOL type system we cannot directly combine e.g. functions like readers for different fields like *'a* ⇒ *'b1* and *'a* ⇒ *'b2* in a HOL list. Here the scene idea to express everything via a merge or update on *'a* is a helpful 'trick'. E.g. 'reading' the value of a field can in a sense be as well expressed as a function *'a* ⇒ *'a* ⇒ *'a* that reads the field of the first argument and puts it into any structure you give it as second argument:

thm *lift-global-heap-def*
thm *open-types.typ-heap-simulationI-all-addressable*
thm *open-types.typ-heap-simulationI-part-addressable*

thm *open-types.typ-heap-simulationI-no-addressable*
thm *pointer-lense-def*
thm *partial-pointer-lense-def*
thm *typ-heap-simulation-of-field-def*

Other important building blocks are:

- *pointer-lense* which describes a virtual heap.
- *partial-pointer-lense* which describes the effect of field lookup / updates on a virtual heap by combining a scene identifying the field and the pointer lense for the virtual heap.
- *typ-heap-simulation-of-field ?st ?t-hrs ?t-hrs-update ?heap-typing-upd ?f′ ?r′ ?w′* = ((∀ *d p f*. *?heap-typing-upd d* ∘ *?w′ p f* = *?w′ p f* ∘ *?heap-typing-upd d*) ∧ (∀ *t u n*. *field-ti TYPE*(*?′a*) *?f′* = *Some t* ⟶ *field-lookup* (*typ-uinfo-t TYPE*(*?′a*)) *?f′ 0* = *Some* (*u*, *n*) ⟶ *partial-pointer-lense* (*merge-ti t*) *?r′ ?w′* ∧ (∀ *p s*. (∀ *a*∈*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*?t-hrs s*)) (*PTR*(*stack-byte*) *a*)) ⟶ *?r′* (*?st s*) *p ZERO*(*?′a*) = *ZERO*(*?′a*)) ∧ (∀ *p x h*. *ptr-valid-u u* (*hrs-htd* (*?t-hrs h*)) &(*p*→*?f′*) ⟶ *?st* (*?t-hrs-update* (*hrs-mem-update* (*heap-upd-list* (*size-td u*) &(*p*→*?f′*) (*access-ti t x*))) *h*) = *?w′ p x* (*?st h*)))) which is used to break down *typ-heap-simulation* to the level of fields in the structure.

The automation is implemented in `HeapLiftBase.gen_new_heap`.

The following paragraphs describe some abstract arguments and lemma collections. In a previous version the proof *typ-heap-simulation* was more directly based on those arguments implemented in ML. Meanwhile we were able to replace most parts of the ML code by lemmas described before based on scenes and pointer lenses. Nevertheless the arguments are still valid and the lemma collections might be useful in other places.

Fundamental heaps We start with the fundamental heaps. The simulation property for heap-updates is captured in *write-simulation*. It is a commutation property. Provided we have a valid pointer in the split heap (obtained from lifting from the byte-level heap), we can either first perform the heap update in the byte-level heap and then lift the state via *lift-global-heap*, or first lift the byte-level heap into the split heap and apply the corresponding update there. Both ways yield the same final state. So we have to prove equality of two states of *lifted-globals*. The first question is which parts of the state did change? Intuitively only the affected split heap did change, all other heaps stayed the same. But all split heaps are derived from the same byte level heap. So the proof decomposes into two main arguments, one is for the affected heap where we have to show that

both updates, the one on the byte-level heap and and the one on the affected split heap, are the same. The other argument is that all other split heaps remain unchanged. Actually the first case is the more straight forward one and could be handled with theorems like *PTR-VALID*(*?'a*) (*hrs-htd ?h*) *?p* $\Longrightarrow$ *plift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) = (*plift ?h*)(*?p* $\mapsto$ *?v*).

thm *plift-heap-update*

The second case, to prove what is not changed is more involved. As a split heap might not only contain root pointers but also valid pointers that are nested in other types we have to distinguish several cases. We developed a general theory in *open-types* and the essential theorems for the commutation proof are collected in ⟦*PTR-VALID*(*?'a*) *?d ?p*; *PTR-VALID*(*?'a*) *?d ?q*⟧ $\Longrightarrow$ *h-val* (*heap-update ?p ?v ?h*) *?q* = ((*h-val ?h*)(*?p* := *?v*)) *?q*

⟦*PTR-VALID*(*?'a*) *?d ?p*; *PTR-VALID*(*?'a*) *?d ?q*; *length ?bs* = *size-of TYPE*(*?'a*)⟧ $\Longrightarrow$ *h-val* (*heap-update-padding ?p ?v ?bs ?h*) *?q* = ((*h-val ?h*)(*?p* := *?v*)) *?q*

PTR-VALID(*?'a*) (*hrs-htd ?h*) *?p* $\Longrightarrow$ *plift ?h ?p* = *Some* (*h-val* (*hrs-mem ?h*) *?p*). When we update the heap at pointer *p* and lookup the value of pointer *q* we can distinguish various cases by analysing the type relation (e.g. if one type is nested in the other). Note that by introducing guards into the code and the design of *lift-global-heap* we only have to care about the case were both pointers are valid according to *PTR-VALID*('*a*). To be more specific. That the pointer that is updated is *PTR-VALID*('*a*) is ensured by the corresponding guard on the update. That the pointer we read from is actually *PTR-VALID*('*a*) is more subtle. Here we rely on the construction of *lift-global-heap* were a split heap is constructed by λp. *the-default ZERO*('*a*) (*plift* (*t-hrs-'* *g*) *p*). In case the pointer is invalid we always obtain *ZERO*('*a*) and hence only the valid pointers may be affected by an heap update. This reduction to valid pointers and *h-val* is encapsulated in theorems *root-ptr-valid* (*hrs-htd ?h*) *?p* $\Longrightarrow$ *the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) *?q*) = ((λp. *the-default ZERO*(*?'a*) (*simple-lift ?h p*))(*?p* := *?v*)) *?q*

⟦*root-ptr-valid* (*hrs-htd ?h*) *?p*; *length ?bs* = *size-of TYPE*(*?'a*)⟧ $\Longrightarrow$ *the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update-padding ?p ?v ?bs*) *?h*) *?q*) = ((λp. *the-default ZERO*(*?'a*) (*simple-lift ?h p*))(*?p* := *?v*)) *?q*

root-ptr-valid (*hrs-htd ?h*) *?p* $\wedge$ *?f ?x* = *?v* $\Longrightarrow$ *?f* (*the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update ?p ?x*) *?h*) *?q*)) = ((λp. *?f* (*the-default ZERO*(*?'a*) (*simple-lift ?h p*)))(*?p* := *?v*)) *?q*

⟦*root-ptr-valid* (*hrs-htd ?h*) *?p* $\wedge$ *?f ?x* = *?v*; *length ?bs* = *size-of TYPE*(*?'a*)⟧ $\Longrightarrow$ *?f* (*the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*) *?h*) *?q*)) = ((λp. *?f* (*the-default ZERO*(*?'a*) (*simple-lift ?h p*)))(*?p* := *?v*)) *?q*

(⟦*root-ptr-valid* (*hrs-htd ?h*) *?q*; *hrs-htd ?h* $\models_t$ *?q*⟧ $\Longrightarrow$ *h-val* (*heap-update*

?p ?v (hrs-mem ?h)) ?q = h-val (hrs-mem ?h) ?q) ⟹ the-default ZERO(?'a) (simple-lift (hrs-mem-update (heap-update ?p ?v) ?h) ?q) = the-default ZERO(?'a) (simple-lift ?h ?q)

⟦length ?bs = size-of TYPE(?'b); ⟦root-ptr-valid (hrs-htd ?h) ?q; hrs-htd ?h ⊨t ?q⟧ ⟹ h-val (heap-update-padding ?p ?v ?bs (hrs-mem ?h)) ?q = h-val (hrs-mem ?h) ?q⟧ ⟹ the-default ZERO(?'a) (simple-lift (hrs-mem-update (heap-update-padding ?p ?v ?bs) ?h) ?q) = the-default ZERO(?'a) (simple-lift ?h ?q)

(⟦root-ptr-valid (hrs-htd ?h) ?q; hrs-htd ?h ⊨t ?q⟧ ⟹ ?f (h-val (heap-update ?p ?v (hrs-mem ?h)) ?q) = ?f (h-val (hrs-mem ?h) ?q)) ⟹ ?f (the-default ZERO(?'a) (simple-lift (hrs-mem-update (heap-update ?p ?v) ?h) ?q)) = ?f (the-default ZERO(?'a) (simple-lift ?h ?q))

⟦length ?bs = size-of TYPE(?'b); ⟦root-ptr-valid (hrs-htd ?h) ?q; hrs-htd ?h ⊨t ?q⟧ ⟹ ?f (h-val (heap-update-padding ?p ?v ?bs (hrs-mem ?h)) ?q) = ?f (h-val (hrs-mem ?h) ?q)⟧ ⟹ ?f (the-default ZERO(?'a) (simple-lift (hrs-mem-update (heap-update-padding ?p ?v ?bs) ?h) ?q)) = ?f (the-default ZERO(?'a) (simple-lift ?h ?q))

PTR-VALID(?'a) (hrs-htd ?h) ?p ⟹ the-default ZERO(?'a) (plift (hrs-mem-update (heap-update ?p ?v) ?h) ?q) = ((λp. the-default ZERO(?'a) (plift ?h p))(?p := ?v)) ?q

⟦PTR-VALID(?'a) (hrs-htd ?h) ?p; length ?bs = size-of TYPE(?'a)⟧ ⟹ the-default ZERO(?'a) (plift (hrs-mem-update (heap-update-padding ?p ?v ?bs) ?h) ?q) = ((λp. the-default ZERO(?'a) (plift ?h p))(?p := ?v)) ?q

PTR-VALID(?'a) (hrs-htd ?h) ?p ∧ ?f ?x = ?v ⟹ ?f (the-default ZERO(?'a) (plift (hrs-mem-update (heap-update ?p ?x) ?h) ?q)) = ((λp. ?f (the-default ZERO(?'a) (plift ?h p)))(?p := ?v)) ?q

⟦PTR-VALID(?'a) (hrs-htd ?h) ?p ∧ ?f ?x = ?v; length ?bs = size-of TYPE(?'a)⟧ ⟹ ?f (the-default ZERO(?'a) (plift (hrs-mem-update (heap-update-padding ?p ?x ?bs) ?h) ?q)) = ((λp. ?f (the-default ZERO(?'a) (plift ?h p)))(?p := ?v)) ?q

(⟦PTR-VALID(?'a) (hrs-htd ?h) ?q; hrs-htd ?h ⊨t ?q⟧ ⟹ h-val (heap-update ?p ?v (hrs-mem ?h)) ?q = h-val (hrs-mem ?h) ?q) ⟹ the-default ZERO(?'a) (plift (hrs-mem-update (heap-update ?p ?v) ?h) ?q) = the-default ZERO(?'a) (plift ?h ?q)

⟦length ?bs = size-of TYPE(?'b); ⟦PTR-VALID(?'a) (hrs-htd ?h) ?q; hrs-htd ?h ⊨t ?q⟧ ⟹ h-val (heap-update-padding ?p ?v ?bs (hrs-mem ?h)) ?q = h-val (hrs-mem ?h) ?q⟧ ⟹ the-default ZERO(?'a) (plift (hrs-mem-update (heap-update-padding ?p ?v ?bs) ?h) ?q) = the-default ZERO(?'a) (plift ?h ?q)

(⟦PTR-VALID(?'a) (hrs-htd ?h) ?q; hrs-htd ?h ⊨t ?q⟧ ⟹ ?f (h-val (heap-update ?p ?v (hrs-mem ?h)) ?q) = ?f (h-val (hrs-mem ?h) ?q)) ⟹ ?f (the-default ZERO(?'a) (plift (hrs-mem-update (heap-update ?p ?v) ?h) ?q)) = ?f (the-default ZERO(?'a) (plift ?h ?q))

⟦length ?bs = size-of TYPE(?'b); ⟦PTR-VALID(?'a) (hrs-htd ?h) ?q;

hrs-htd ?h $\models_t$ *?q*$]\!] \Longrightarrow$ *?f* (*h-val* (*heap-update-padding ?p ?v ?bs* (*hrs-mem ?h*)) *?q*) = *?f* (*h-val* (*hrs-mem ?h*) *?q*)$]\!] \Longrightarrow$ *?f* (*the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update* (*heap-update-padding ?p ?v ?bs*) *?h*) *?q*)) = *?f* (*the-default ZERO*(*?'a*) (*plift ?h ?q*)).

thm *lift-global-heap-def*
thm *the-plift-hval-eqI*
thm *plift-eqI*
thm *plift-simps*

Treatment of Array Types At the core there is no special type-information for array types. It is treated analogously to structures. Each index gets an individual field name which is the unary encoding of the index. The good thing is that one does not need any new fundamental lemmas to deal with array types. But it is not a good idea to unfold the type-information for arrays and to work on that expanded version. On the one hand things like simplification might become slow, on the other hand we can make use of the regular structure of array types and once and forall derive general lemmas.

The central theorems to work with field-lookup in arrays are *?n* < *CARD*(*?'b*) $\Longrightarrow$ *field-lookup* (*typ-info-t TYPE*(*?'a*[*?'b*])) [*replicate ?n CHR ''1''*] *?i* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*?'a*)) (λ*x*. *x*.[*?n*]) (λ*x f*. *Arrays.update f ?n x*), *?i* + *?n* ∗ *size-of TYPE*(*?'a*)) and *?i* < *CARD*(*?'b*) $\Longrightarrow$ *array-ptr-index ?p False ?i* = *PTR*(*?'a*) &(*?p*→[*replicate ?i CHR ''1''*]). The latter one allows to convert between an array-index-arithmetic based view of array pointers *array-ptr-index p False i* and the field-name view *PTR*('*a*) &(*p*→[*replicate i CHR ''1''*]).

thm *field-lookup-array*
thm *array-ptr-index-field-lvalue-conv*

These theorems are used to normalise array accesses towards symbolic field name accesses of the format term>*Ptr* &(*p*→[*replicate i CHR ''1''*]). Note that we don't unfold the definition of *replicate* here, and index *i* stays abstract, constrained by precondition that limits its range.

thm *unpacked-C.ptr-valid.unfold*
thm *unpacked-C'array-2.ptr-valid.unfold*

The instances of *typ-heap-simulation* can be established from *typ-heap-simulation* of the element types. We introduce the locales *array-typ-heap-simulation* and *two-dimensional-array-typ-heap-simulation* to automatically provide the sublocale relations when the element type falls into *array-outer-max-size* or *array-inner-max-size* respectively.

Derived Heaps The commutation proofs for derived heaps follow another path. For derived heaps we can assume that we have all the instances of *typ-heap-simulation* of the component types already available. Unaddressable fields are never shared. Pointers only appear within a dedicated

enclosing parent structure. Validity of the pointer coincides with validity of the parent pointer. Pointers that are mapped by the field heap coincide with the parent pointers, there is no need to calculate the offset of the field. So deriving the commutation proof from the available theorems boils down to their composition. The central lemma connects an update of a derived heap to an fold over the updates of the toplevel fields: *c-guard ?p* $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[''*y4-C*'']))(*y4-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[''*y3-C*'']))(*y3-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[''*y2-C*'']))(*y2-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[''*y1-C*'']))(*y1-C ?x*) ∘ *heap-update* (*PTR*(*32 signed word*) &(*?p*→[''*x-C*'']))(*x-C ?x*))))

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[''*c2-C*'']))(*c2-C ?x*) ∘ *heap-update* (*PTR*(*32 word*) &(*?p*→[''*c1-C*'']))(*c1-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*closed-C*) &(*?p*→[''*fld4-C*'']))(*fld4-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[''*fld3-C*'']))(*fld3-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[''*fld2-C*'']))(*fld2-C ?x*) ∘ *heap-update* (*PTR*(*32 word*) &(*?p*→[''*fld1-C*'']))(*fld1-C ?x*)))

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[''*lng-C*'']))(*lng-C ?x*) ∘ *heap-update* (*PTR*(*8 word*) &(*?p*→[''*chr-C*'']))(*chr-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[''*fld-C*'']))(*outer-C.fld-C ?x*) ∘ *heap-update* (*PTR*(*inner-C*) &(*?p*→[''*inner-C*'']))(*outer-C.inner-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[''*count-C*'']))(*count-C ?x*) ∘ *heap-update* (*PTR*(*unpacked-C*[*2*]) &(*?p*→[''*elements-C*'']))(*elements-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*unpacked-C*) &(*?p*→[''*fy-C*'']))(*fy-C ?x*) ∘ (*heap-update* (*PTR*(*closed-C*) &(*?p*→[''*fz-C*'']))(*fz-C ?x*) ∘ *heap-update* (*PTR*(*32 word*) &(*?p*→[''*fx-C*'']))(*fx-C ?x*))

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*unpacked-C*[*3*][*2*]) &(*?p*→[''*matrix-C*'']))(*matrix-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[''*fld-C*'']))(*outer-array-C.fld-C ?x*) ∘ *heap-update* (*PTR*(*inner-C*[*5*]) &(*?p*→[''*inner-array-C*'']))(*inner-array-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*data-C*[*10*]) &(*?p*→[''*array-C*'']))(*data-array-C.array-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*data-C*) &(*?p*→[''*d2-C*'']))(*d2-C ?x*) ∘ *heap-update* (*PTR*(*data-C*) &(*?p*→[''*d1-C*'']))(*d1-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*data-struct1-C*) &(*?p*→[''*d-C*'']))(*d-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*[*12*]) &(*?p*→[''*arr2-C*'']))(*arr2-C ?x*) ∘ *heap-update* (*PTR*(*32 word*[*12*]) &(*?p*→[''*arr1-C*'']))(*arr1-C ?x*). Moreover, we establish that *heap-update-padding* is equivalent to *heap-update* under the state lifting function. Padding bytes become irrelevant in the split heap. For each toplevel field we have the com-

mutation proof connecting the monolithic byte-level heap update to the split-heap update collected in ⟦*IS-VALID*(*32 word*) (*lift-global-heap ?s*) *?p*; *length ?bs = size-of TYPE*(*32 word*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-w32-update* (λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*32 word*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-w32-update* (λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*32 word*) (*lift-global-heap ?s*) (*PTR-COERCE*(*32 signed word* → *32 word*) *?p*); *length ?bs = size-of TYPE*(*32 signed word*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-w32-update* (λ*h*. *h*(*PTR-COERCE*(*32 signed word* → *32 word*) *?p* := *UCAST*(*32 signed* → *32*) *?x*)) (*lift-global-heap ?s*)

IS-VALID(*32 word*) (*lift-global-heap ?s*) (*PTR-COERCE*(*32 signed word* → *32 word*) *?p*) ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-w32-update* (λ*h*. *h*(*PTR-COERCE*(*32 signed word* → *32 word*) *?p* := *UCAST*(*32 signed* → *32*) *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*8 word*) (*lift-global-heap ?s*) *?p*; *length ?bs = size-of TYPE*(*8 word*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-w8-update* (λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*8 word*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-w8-update* (λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*8 word*) (*lift-global-heap ?s*) (*PTR-COERCE*(*8 signed word* → *8 word*) *?p*); *length ?bs = size-of TYPE*(*8 signed word*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-w8-update* (λ*h*. *h*(*PTR-COERCE*(*8 signed word* → *8 word*) *?p* := *UCAST*(*8 signed* → *8*) *?x*)) (*lift-global-heap ?s*)

IS-VALID(*8 word*) (*lift-global-heap ?s*) (*PTR-COERCE*(*8 signed word* → *8 word*) *?p*) ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-w8-update* (λ*h*. *h*(*PTR-COERCE*(*8 signed word* → *8 word*) *?p* := *UCAST*(*8 signed* → *8*) *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*32 word ptr*) (*lift-global-heap ?s*) *?p*; *length ?bs = size-of TYPE*(*32 word ptr*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-w32′ptr-update* (λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*32 word ptr*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-w32′ptr-update* (λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*unit ptr*) (*lift-global-heap ?s*) *?p*; *length ?bs = size-of TYPE*(*unit ptr*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-unit′ptr-update* (λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*unit ptr*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-unit′ptr-update* (λ*h*. *h*(*?p*

:= ?x)) (lift-global-heap ?s)
⟦IS-VALID(data-C) (lift-global-heap ?s) ?p; length ?bs = size-of TYPE(data-C)⟧
⟹ lift-global-heap (t-hrs-′-update (hrs-mem-update (heap-update-padding
?p ?x ?bs)) ?s) = heap-data-C-update (λh. h(?p := ?x)) (lift-global-heap
?s)
IS-VALID(data-C) (lift-global-heap ?s) ?p ⟹ lift-global-heap (t-hrs-′-update
(hrs-mem-update (heap-update ?p ?x)) ?s) = heap-data-C-update (λh. h(?p
:= ?x)) (lift-global-heap ?s)
⟦IS-VALID(closed-C) (lift-global-heap ?s) ?p; length ?bs = size-of TYPE(closed-C)⟧
⟹ lift-global-heap (t-hrs-′-update (hrs-mem-update (heap-update-padding
?p ?x ?bs)) ?s) = heap-closed-C-update (λh. h(?p := ?x)) (lift-global-heap
?s)
IS-VALID(closed-C) (lift-global-heap ?s) ?p ⟹ lift-global-heap (t-hrs-′-update
(hrs-mem-update (heap-update ?p ?x)) ?s) = heap-closed-C-update (λh. h(?p
:= ?x)) (lift-global-heap ?s)
⟦IS-VALID(unpacked-C) (lift-global-heap ?s) ?p; length ?bs = size-of
TYPE(unpacked-C)⟧ ⟹ lift-global-heap (t-hrs-′-update (hrs-mem-update
(heap-update-padding ?p ?x ?bs)) ?s) = heap-unpacked-C-update (λh. h(?p
:= ?x)) (lift-global-heap ?s)
IS-VALID(unpacked-C) (lift-global-heap ?s) ?p ⟹ lift-global-heap (t-hrs-′-update
(hrs-mem-update (heap-update ?p ?x)) ?s) = heap-unpacked-C-update (λh.
h(?p := ?x)) (lift-global-heap ?s)
⟦IS-VALID(arr-struct-C) (lift-global-heap ?s) ?p; length ?bs = size-of
TYPE(arr-struct-C)⟧ ⟹ lift-global-heap (t-hrs-′-update (hrs-mem-update
(heap-update-padding ?p ?x ?bs)) ?s) = heap-arr-struct-C-update (λh. h(?p
:= ?x)) (lift-global-heap ?s)
IS-VALID(arr-struct-C) (lift-global-heap ?s) ?p ⟹ lift-global-heap (t-hrs-′-update
(hrs-mem-update (heap-update ?p ?x)) ?s) = heap-arr-struct-C-update (λh.
h(?p := ?x)) (lift-global-heap ?s)
⟦valid-array-base.valid-array (λh. IS-VALID(unpacked-C) h) (lift-global-heap
?s) ?p; length ?bs = size-of TYPE(unpacked-C[3])⟧ ⟹ lift-global-heap
(t-hrs-′-update (hrs-mem-update (heap-update-padding ?p ?x ?bs)) ?s) = un-
packed-C.heap-array-map ?p (λ-. ?x) (lift-global-heap ?s)
valid-array-base.valid-array (λh. IS-VALID(unpacked-C) h) (lift-global-heap
?s) ?p ⟹ lift-global-heap (t-hrs-′-update (hrs-mem-update (heap-update ?p
?x)) ?s) = unpacked-C.heap-array-map ?p (λ-. ?x) (lift-global-heap ?s)
⟦valid-array-base.valid-array (valid-array-base.valid-array (λh. IS-VALID(unpacked-C)
h)) (lift-global-heap ?s) ?p; length ?bs = size-of TYPE(unpacked-C[3][2])⟧
⟹ lift-global-heap (t-hrs-′-update (hrs-mem-update (heap-update-padding
?p ?x ?bs)) ?s) = unpacked-C.outer.heap-array-map ?p (λ-. ?x) (lift-global-heap
?s)
valid-array-base.valid-array (valid-array-base.valid-array (λh. IS-VALID(unpacked-C)
h)) (lift-global-heap ?s) ?p ⟹ lift-global-heap (t-hrs-′-update (hrs-mem-update
(heap-update ?p ?x)) ?s) = unpacked-C.outer.heap-array-map ?p (λ-. ?x)

(*lift-global-heap ?s*)

⟦*IS-VALID*(*two-dimensional-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*two-dimensional-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-two-dimensional-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*two-dimensional-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-two-dimensional-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*data-struct1-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-struct1-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-data-struct1-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*data-struct1-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-data-struct1-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*data-struct2-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-struct2-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-data-struct2-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*data-struct2-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-data-struct2-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*inner-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*inner-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-inner-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*inner-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-inner-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*valid-array-base.valid-array* (λ*h*. *IS-VALID*(*inner-C*) *h*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*inner-C*[5])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *inner-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

valid-array-base.valid-array (λ*h*. *IS-VALID*(*inner-C*) *h*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *inner-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*outer-array-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*outer-array-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-outer-array-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*outer-array-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-outer-array-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*valid-array-base.valid-array* (λ*h*. *IS-VALID*(*data-C*) *h*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-C*[10])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update*

(*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *data-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

valid-array-base.valid-array (λ*h*. *IS-VALID*(*data-C*) *h*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *data-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*data-array-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-array-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-data-array-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*data-array-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-data-array-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*outer-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*outer-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-outer-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*outer-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-outer-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*other-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*other-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-other-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*other-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-other-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*valid-array-base.valid-array* (λ*h*. *IS-VALID*(*unpacked-C*) *h*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unpacked-C*[*2*])⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *unpacked-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

valid-array-base.valid-array (λ*h*. *IS-VALID*(*unpacked-C*) *h*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *unpacked-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*array-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*array-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-array-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*array-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-array-C-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*unpacked-C*[*3*][*2*]) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unpacked-C*[*3*][*2*])⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *unpacked-C.outer.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*unpacked-C*[*3*][*2*]) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *unpacked-C.outer.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*inner-C*[5]) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*inner-C*[5])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *inner-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*inner-C*[5]) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *inner-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*data-C*[10]) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-C*[10])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *data-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*data-C*[10]) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *data-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*unpacked-C*[2]) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unpacked-C*[2])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *unpacked-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)

IS-VALID(*unpacked-C*[2]) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *unpacked-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*).

thm *heap-update-fold-toplevel-fields-pointless*
thm *plift-heap-update-padding-heap-update-pointless-conv*
thm *lift-heap-update-padding-heap-update-conv*
thm *write-commutes*

No restriction to *packed-type***!**

In the original version of AutoCorres there was a restriction of heap lifting to types that are also in *packed-type*. Those are types that internally have no padding fields at all. The reason was that certain lemmas especially regarding heap update become more involved in the presence of padding fields. A *heap-update* preserves the value of all padding bytes, whereas the abstract value obtained from a corresponding *h-val* is independent of the value of the padding bytes. So in principle these notions fit together well but it seems that some formal language was missing to reason about padding bytes. Meanwhile we added a theory to reason about those padding bytes in *AutoCorres2.Padding-Equivalence*. This made it possible to liberate heap lifting from the restriction to *packed-type*. For example *heap-update ?p ?arr* = (λ*s*. *foldl* (λ*s n*. *heap-update* (*array-ptr-index ?p False n*) (*?arr*.[*n*]) *s*) *s* [*0..<CARD*(*?'b*)]) for *packed-type* holds in general *c-guard ?p* ⟹ *heap-update ?p ?arr ?h* = *fold* (λ*i*. *heap-update* (*array-ptr-index ?p False i*) (*?arr*.[*i*])) [*0..<CARD*(*?'b*)] *?h*.

thm *heap-update-array*
thm *heap-update-Array*

Padding Equivalence and *xmem-type*

Padding bytes and padding fields are introduced via the construction of new C-Types via the combinators *ti-pad-combine* and *ti-typ-pad-combine* to satisfy alignment properties. However, the concept of a padding byte is not a first-class citizen of a ('*a*, '*b*) *typ-desc*, but just happens to be a field with some special properties. In practice these fields are generated by *ti-pad-combine*, so we know that the field name starts with ''!*pad*'', and the associated 'lense' for lookup and update have the 'passthrough' properties of a padding field.

We made these properties explicit by supplying a theory to identify padding-bytes in a list of bytes associated with an C-Type, and having notions to compare such byte-lists, e.g. telling if two byte lists are equal up-to the padding bytes, or if they agree on the padding bytes. This information is also backed into the properties of *xmem-type*, which is a subclass of *mem-type*. All primitive word and pointer types as well as all types that are constructed by the UMM module of the C-Parser are proved to belong to that class. The construction of array types propagates the class as expected.

Again the notions come in pairs of a 'typed' and a 'untyped' version. The 'typed' version associated with *typ-info-t* is a lense based formalisation, the 'untyped' version associated with *typ-uinfo-t* is based on byte lists.

The lense based version is introduced by *padding-lense*. Access and update follow the approach of '*a field-desc*. Lookup / access has type '*a* ⇒ *byte list* ⇒ *byte list*. It takes an abstract value and a supply list of padding bytes and retrieves the bytes encoding the field. The intuition of the padding byte list is the following. It is supposed to bridge the gap between the abstract value and the byte representation. The abstract value is independent of the padding bytes. Hence in general there is no one-to-one correspondence between a byte list encoding of an abstract value and the abstract value. When converting from a byte list to an abstract value this is fine, the padding bytes are just ignored. But when we go the other way we have to invent the padding bytes. One solution could be to just fill up with zeros. Another solution could be to yield a non-deterministic result for the padding bytes. The UMM model chose another way. The possible valuation for padding bytes is supplied as an additional argument. So whenever a padding byte is needed we just take it from the position in that list. The update has type *byte list* ⇒ '*a* ⇒ '*a*. It takes the value of a field, encoded as a byte list, and transforms it to an update on the abstract value.

context *padding-lense*
begin

The concept of a *padding-lense* follows the signature of '*a field-desc* and describes the notions of padding bytes as semantic properties of the access function *acc* and the update function *upd*. Based on them it introduces the notions of

- *local.is-padding-byte*, is a byte a padding with respect to the lense?
- *local.is-value-byte*, is a byte a value byte with respect to the lense, i.e. does the abstract value associated with the byte list depend on that byte.
- *local.eq-padding*, access cannot distinguish the byte lists.
- *local.eq-upto-padding*, update cannot distinguish the byte list.

thm *is-padding-byte-def*
thm *is-value-byte-def*
thm *eq-padding-def*
thm *eq-upto-padding-def*

As the definitions are semantically defined the effect on *acc* and *upd* are rather immediate. For example, if two byte lists are equal upto padding, then an update with *upd* yields the same result. If the padding bytes within two supply byte lists are the same, then *acc v* yields the same result for both byte lists.

end

In the untyped *typ-uinfo-t* we define the corresponding notions, as properties of the byte list instead of the access and update functions.

- *is-padding-byte*, a byte is a padding byte in a byte list, if normalisation of the byte list is independent of its value. Normalisation *norm-tu* is defined on *typ-uinfo-t* and sets all padding bytes to zero.
- *is-value-byte*, normalisation depends on the value of the byte.
- *eq-padding*, all padding bytes are the same.
- *eq-upto-padding*, all value bytes are the same.

thm *is-padding-byte-def*
thm *is-value-byte-def*
thm *eq-padding-def*
thm *eq-upto-padding-def*

For instances of *xmem-type* we can go back and forth between both characterisations.

thm *is-padding-byte-lense-conv*
thm *field-lookup-is-padding-byte-lense-conv*
thm *is-value-byte-lense-conv*
thm *field-lookup-is-value-byte-lense-conv*
thm *eq-padding-lense-conv*
thm *field-lookup-eq-padding-lense-conv*

thm *eq-upto-padding-lense-conv*
thm *field-lookup-eq-upto-padding-lense-conv*

With those theorems in place it is easy to show that padding fields only consist of padding bytes and thus do not account to the abstract value.

thm *field-lookup-access-ti-to-bytes-field-conv*
thm *access-ti-update-ti-eq-upto-padding*
thm *field-lookup-qualified-padding-field-name(1)*
thm *is-padding-tag-def padding-tag-def padding-desc-def*

UMM-Simprocs Cache

To solve sideconditions on UMM-Types we have implemented some simprocs within `UMM_Proofs`. Currently their scope is limited to the usecases we have in the proofs described above. It should be straightforward to extend them to more usecases. Their purpose is not to support abstract reasoning on types but to provide properties of concrete instances of C-types, for example deciding whether $TYPE(32\ word) \leq_\tau TYPE(outer\text{-}C)$ holds.

The benefit of the simprocs is that we do not have to guess and prove lots of lemmas already during the definition of a new UMM type, but can postpone it until we actually need them. Once they are proven they are added to the cache, so they are only proven once. In our use-case the lemmas we need depend on the addressable fields the user specifies on an **autocorres** invocation.

ML-val ‹
val - = ***simproc*** ‹*field-lookup*›
val - = ***simproc*** ‹*type-calculations*›
val - = ***simproc*** ‹*typuinfo-calculations*›
›

ML-val ‹
Cached-Theory-Simproc.trace-cache @{*context*}
›

The simprocs make use of a common cache. The cache itself is implemented as a simpset. Cache lookup means we try to rewrite with the cache:

- cache miss: term is unchanged.
- cache hit: term is changed.

  In case of a cache hit we are finished and return the resulting equation. In case of a cache miss we invoke the simplifier with a taylored simpset to derive a new equation. This equation is then added to the cache as well as returned from the simproc.

  Some fine points of this setup are related to context management. Conceptually we only prove theory level theorems about an UMM

type. So even if we prove them after the definition of the UMM type they should all be applicable as if we have immediately proven them at the point where the UMM type was generated. However, the simproc is invoked in some context later on where the theory has already advanced. In order to produce a result after a cache miss we have to somehow 'travel back in time' to the original theory context after the definition of the UMM type, such that the simplifier properly works on it and the result is reusable in other contexts. We maintain that original theory state and recertify terms before simplifying them.

Another point is, that the terms we attempt to simplify and cache might depend on each other. In order not to miss to cache intermediate results one has to carefully craft the simpsets.

In general the simplifier tries simprocs only after unconditional and conditional rules. So when a rewrite rule has already transformed the redex the simproc will never see that redex. This has two main implications:

- In order to have the simproc-based cache applied, there must not be a rule in the simpset that removes the redex before the cache actually has a chance to see it. This is why we maintain several simpsets to control that behaviour.

- When a cache miss was encountered and we apply the simplifier recursively to rewrite the redex we have to take care that the simproc is not invoked again on the same redex. When there is an unconditional rule in the simpset that rewrites that redex we are on the safe side. But beware of *conditional* rules. When the solver fails to solve the conditions the rule can fail and then the simproc is invoked again on the same redex. To prevent the setup from looping in these cases we maintain the redexes the simproc has already seen.

Here are some principles in the design of the simprocs and their simpsets:

- The type descriptions obtained by *typ-info-t* are not fully expanded. Instead we use a simplifier setup that works on the combinators, like *empty-typ-info*, *final-pad*, *ti-typ-pad-combine*. This maintains the more compact representation of a type-descriptor as a combinator expression, which is anyways the way it is originally defined.

- Fieldnames for array indexes stay symbolic: *replicate i ''1''*. We employ derived rules for array indexes and do not have to expand the type-descriptor of arrays. For these rules to apply we typically need the information that the index is in the bounds of the array type. In those cases the simproc generates a conditional rewrite rule.

- To test equality on *export-uinfo* t = *export-uinfo* s we normalise towards *export-uinfo* (*typ-info-t* $TYPE('a)$) = *export-uinfo* (*typ-info-t* $TYPE('b)$), where both $'a$ and $'b$ are concrete instances. Then when both expressions happen to be the same we have proved equality. Note that this approach is incomplete and in particular not sufficient to disprove the equality and prove the inequality. However, as these equality often appears as sidecondition in a conditional rewrite rule, not being able to prove the equality is somehow "equivalent" to disproving it: If we cannot prove the sidecondition the rule cannot be applied. The most prominent pattern for this is a *field-lookup* yielding the type-description of the selected field which is then compared to some type-description that is derived from a pointer type. To also disprove equality we make use of rule (*export-uinfo* (*typ-info-t* $TYPE(?'a)$) = *export-uinfo* (*typ-info-t* $TYPE(?'b)$)) = ($TYPE(?'a) \leq_\tau TYPE(?'b)$ $\wedge$ $TYPE(?'b) \leq_\tau TYPE(?'a)$) and use the simproc on *sub-typ*.

- To decide whether e.g. $TYPE(32\ word) \leq_\tau TYPE(outer\text{-}C)$ holds or not, we first try to disprove it by using type name information $[\![typ\text{-}name\ (typ\text{-}info\text{-}t\ TYPE(?'a)) \neq pad\text{-}typ\text{-}name; typ\text{-}name\ (typ\text{-}info\text{-}t\ TYPE(?'a)) \notin td\text{-}names\ (typ\text{-}info\text{-}t\ TYPE(?'b))]\!] \Longrightarrow \neg\ TYPE(?'a) \leq_\tau TYPE(?'b)$. both *typ-name* and *td-names* are supplied by the UMM package: *typ-name* (*typ-info-t* $TYPE(arr\text{-}struct\text{-}C)) \equiv ''arr\text{-}struct\text{-}C''$

  typ-name (*typ-info-t* $TYPE(data\text{-}struct2\text{-}C)) \equiv ''data\text{-}struct2\text{-}C''$

  typ-name (*typ-info-t* $TYPE(data\text{-}struct1\text{-}C)) \equiv ''data\text{-}struct1\text{-}C''$

  typ-name (*typ-info-t* $TYPE(data\text{-}array\text{-}C)) \equiv ''data\text{-}array\text{-}C''$

  typ-name (*typ-info-t* $TYPE(outer\text{-}array\text{-}C)) \equiv ''outer\text{-}array\text{-}C''$

  typ-name (*typ-info-t* $TYPE(two\text{-}dimensional\text{-}C)) \equiv ''two\text{-}dimensional\text{-}C''$

  typ-name (*typ-info-t* $TYPE(other\text{-}C)) \equiv ''other\text{-}C''$

  typ-name (*typ-info-t* $TYPE(array\text{-}C)) \equiv ''array\text{-}C''$

  typ-name (*typ-info-t* $TYPE(outer\text{-}C)) \equiv ''outer\text{-}C''$

  typ-name (*typ-info-t* $TYPE(unpacked\text{-}C)) \equiv ''unpacked\text{-}C''$

  typ-name (*typ-info-t* $TYPE(inner\text{-}C)) \equiv ''inner\text{-}C''$

  typ-name (*typ-info-t* $TYPE(closed\text{-}C)) \equiv ''closed\text{-}C''$

  typ-name (*typ-info-t* $TYPE(data\text{-}C)) \equiv ''data\text{-}C''$, *td-names* (*typ-info-t* $?x) \equiv \{''word0000010'', ''arr\text{-}struct\text{-}C'', ''word0000010\text{-}array\text{-}00110''\}$

  td-names (*typ-info-t* $?x) \equiv \{''data\text{-}C'', ''word0000010'', ''data\text{-}struct2\text{-}C'', ''data\text{-}struct1\text{-}C''\}$

  td-names (*typ-info-t* $?x) \equiv \{''data\text{-}C'', ''word0000010'', ''data\text{-}struct1\text{-}C''\}$

  td-names (*typ-info-t* $?x) \equiv \{''data\text{-}C'', ''word0000010'', ''data\text{-}array\text{-}C'', ''data\text{-}C\text{-}array\text{-}01010''\}$

td-names (*typ-info-t ?x*) ≡ {*''inner-C''*, *''closed-C''*, *''word0000010''*, *''outer-array-C''*, *''inner-C-array-1010''*}

td-names (*typ-info-t ?x*) ≡ {*''word00010''*, *''unpacked-C''*, *''word0000010''*, *''two-dimensional-C''*, *''unpacked-C-array-110''*, *''unpacked-C-array-110-array-010''*}

td-names (*typ-info-t ?x*) ≡ {*''other-C''*, *''closed-C''*, *''word00010''*, *''unpacked-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''array-C''*, *''word00010''*, *''unpacked-C''*, *''word0000010''*, *''unpacked-C-array-010''*}

td-names (*typ-info-t ?x*) ≡ {*''inner-C''*, *''outer-C''*, *''closed-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''word00010''*, *''unpacked-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''inner-C''*, *''closed-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''closed-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''data-C''*, *''word0000010''*}. By construction every distinct type gets an individual name. If that does not work we try proving it instead, by using a tansitivity prover on the single step sub-type relations provided by $TYPE(\textit{32 signed word}) \leq_\tau TYPE(\textit{data-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{data-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{closed-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{inner-C})$

$TYPE(\textit{closed-C}) \leq_\tau TYPE(\textit{inner-C})$

$TYPE(\textit{8 word}) \leq_\tau TYPE(\textit{unpacked-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{unpacked-C})$

$TYPE(\textit{inner-C}) \leq_\tau TYPE(\textit{outer-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{outer-C})$

$TYPE(\textit{unpacked-C}[2]) \leq_\tau TYPE(\textit{array-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{array-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{other-C})$

$TYPE(\textit{closed-C}) \leq_\tau TYPE(\textit{other-C})$

$TYPE(\textit{unpacked-C}) \leq_\tau TYPE(\textit{other-C})$

$TYPE(\textit{unpacked-C}[3][2]) \leq_\tau TYPE(\textit{two-dimensional-C})$

$TYPE(\textit{inner-C}[5]) \leq_\tau TYPE(\textit{outer-array-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{outer-array-C})$

$TYPE(\textit{data-C}[10]) \leq_\tau TYPE(\textit{data-array-C})$

$TYPE(\textit{data-C}) \leq_\tau TYPE(\textit{data-struct1-C})$

$TYPE(data\text{-}struct1\text{-}C) \leq_\tau TYPE(data\text{-}struct2\text{-}C)$

$TYPE(32\ word[12]) \leq_\tau TYPE(arr\text{-}struct\text{-}C)$ and the rule for arrays $TYPE(?'a) \leq_\tau TYPE(?'a[?'b])$.

thm *not-sub-typ-via-td-name*
thm *typ-name-simps*
thm *td-names-simps*
thm *sub-typ-simps*
thm *element-typ-subtyp-array-typ*

Here some examples

lemma *export-uinfo* (*typ-info-t TYPE*(*array-C*)) = *export-uinfo* (*typ-info-t TYPE*(*other-C*))
apply *simp*
oops

schematic-goal *field-lookup* (*typ-info-t TYPE*(*outer-C*)) [''*inner-C*'', ''*fld1-C*''] *0* = *?X*
apply *simp*
done

lemma $TYPE(8\ word) \leq_\tau TYPE(outer\text{-}C) = False$
by *simp*

schematic-goal $i < 2 \Longrightarrow j < 3 \Longrightarrow$
field-ti TYPE(*two-dimensional-C*) [''*matrix-C*'', *replicate i CHR* ''*1*'', *replicate j CHR* ''*1*''] = *?X*
apply *simp*
done

lemma $TYPE(8\ word) \leq_\tau TYPE(8\ word[42])$
apply *simp*
done

ML-val ‹
Cached-Theory-Simproc.trace-cache @{*context*}
›

26.21.6 Examples for normalisation of array indexes

lemma
fixes *p*::*array-C ptr*
assumes *bnd*[*simp*]: $i < 2$
shows (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate i CHR* ''*1*'']) = *PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) $+_p$ *int i*
supply [[*simp-trace*=*false*, *verbose*=*5*]]
apply (*array-index-to-ptr-arith-simp*)
done

lemma
fixes *p*::*array-C ptr*
assumes *bnd*: (*i*::*32 word*) < *2*
shows (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate* (*unat i*) *CHR* ''*1*'']))
= *PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *uint i*
supply [[*simp-trace*=*false*, *verbose*=*5*]]
apply (*array-index-to-ptr-arith-simp simp*: *bnd*)
done

lemma
fixes *p*::*array-C ptr*
shows *do* {- <− *guard* (λ-. *unat* (*i*::*32 word*) < *2*);
return ((*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate* (*unat i*) *CHR* ''*1*''])))}
=
do {
- ← *guard* (λ*s*. *unat i* < *2*);
return (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *uint i*)
}
supply [[*simp-trace*=*false*, *verbose*=*6*]]
apply *array-index-to-ptr-arith-simp*
done

lemma
fixes *p*::*array-C ptr*
shows *do* {- <− *guard* (λ-. (*i*::*32 word*) < *2*);
return ((*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate* (*unat i*) *CHR* ''*1*''])))}
=
do {
- ← *guard* (λ*s*. *i* < *2*);
return (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *uint i*)
}
supply [[*simp-trace*=*false*, *verbose*=*6*]]
apply *array-index-to-ptr-arith-simp*
done

lemma
fixes *p*::*array-C ptr*
assumes *bnd*[*simp*]: *i* < *2*
shows (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate i CHR* ''*1*'', ''*chr-C*'']))
=
PTR(*unpacked-C*)
&(*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *int i*→[''*chr-C*''])
supply [[*simp-trace*=*false*, *verbose*=*5*]]
apply (*array-index-to-ptr-arith-simp*)
done

lemma (*i*::*32 word*) < *4* ⟹ *unat i* < *4*
supply [[*array-bound-mksimps*, *verbose*=*5*]]

apply (*simp*)
done

lemma (*i*::*32 signed word*) $<_s$ *4* $\wedge$ *0* $\leq_s$ *i* $\Longrightarrow$ *nat* (*sint i*) < *4*
supply [[*array-bound-mksimps*, *verbose=5*]]
apply (*simp*)
done

26.21.7 Essence of Heap Lifting

Some birds eye view on what heap lifting is about. Consider the following C program.

```
typedef struct foo {
  int myint;
  bool mybool;
} foo;

int * p;
foo * q;

(*p) = 42;
i = q->myint;
b = q->mybool;
```

Is value *i* and *b* affected by update to $*p$
From the C-Parser we only get *c-guard p* and *c-guard q*:

```
{c_guard p \<and> c_guard q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

This only tells us something about alignment of pointers and that the pointer span does not overflow, but nothing about disjointness of the pointer spans.

What about stronger guarantees, if we assume well-typedness of the pointers:

```
{d \<Turnstile>\<^sub>t p \<and> d \<Turnstile>\<^sub>t p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

Now we know that *p* might alias with *q−>myint* as they have the same type, but that pointer span *p* is actually disjoint form pointer span *q−>mybool*. Hence for the read of *b* we can skip the heap update via *p*.

Now consider that the structure *foo* is a closed structure in the split heap model. This means that we have even stronger assumptions (which are guaranteed by guards in the lifted code):

```
{root_ptr_valid d p \<and> root_ptr_valid p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

From this we can infer that the pointer spans p and the entire pointer span q don't overlap. hence reading i as well as b can skip the heap update via p.

In general the heap lifting phase introduces the potentially weaker guards for open types:

```
{ptr_valid d p \<and> ptr_valid p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

So whether i might be affected by p depends on whether field *myint* is addressable or not.

The essential semantic part of heap lifting is to introduce these guards at every pointer access. In a second step these guards then allow a change in a representation from the monolitic heap to the split heap, with a separate heap for each atomic type. But in a sense this second step could be viewed as optional or cosmetic in the semantic sense. There is all the information available to perform the "skipping" steps as sketched above in the monolithic heap. This is what is performed in the simulation proof. So the question is if we could omit introduction of "lifted globals" completely and provide the necessary automation to the user to simplify accesses on the monolithic heap that mimic that behaviour in the split heap. In a sense instead of doing the simulation proof upfront it is done adhoc at every heap access / update. This avoids to provide quadratically many commutation lemmas (in the number of fields of open types) that are introduces for the lifted globals.

From the perspective of the simulation proof alone such a "virtual" split heap, which does not introduce a new lifted-globals type, seems to be equivalent to the one which introduces a new type with concrete distinct record fields for each fundamental heap. The simulation proof only works when every pointer access is guarded and under that assumption the two representations behave the same.

However, besides the simulation there are other aspects of the split heap that might pay off for verification or further abstractions. The two models

behave differently for invalid pointers: In the split-heap (with different record fields), updates to one heap still do not affect other heaps but we cannot properly model such updates in the monolithic heap at all.

Another aspect was introduced later on for stack allocation: non-deterministic padding bytes. The *typ-heap-simulation* was generalised to state that lifting to the split heap is invariant for arbitrary padding bytes, i.e. *write-simulation*. The lifted heap only cares about the abstract values were the values of the padding bytes become irrelevant.

thm *h-val-heap-update-padding*
thm *plift-heap-update-padding-heap-update-pointless-conv*
thm *lift-heap-update-padding-heap-update-conv*

context *open-struct-all-corres*
begin
thm *ts-def*
thm *ac-corres*
end

context *ts-definition-access-arr-struct*
begin
thm *access-arr-struct'-def* [*no-vars*]
lemma *access-arr-struct' p* ≡ *do* {
  i ← *whileLoop* (λ*i s*. *i* < *6*)
        (λ*i*. *do* {
              - ← *guard* (λ*s*. *IS-VALID*(*arr-struct-C*) *s p*);
              - ← *modify* (*heap-arr-struct-C-update* (λ*a*. *a*(*p* := *arr2-C-update* (λ*a*.
Arrays.update a (*unat i*) *3*) (*a p*))));
              - ← *guard* (λ*s*. *i* + *1* < *0xC*);
                  - ← *modify* (*heap-arr-struct-C-update* (λ*a*. *a*(*p* := *arr2-C-update*
(*fupdate* (*unat* (*i* + *1*)) (λ*v*. *v* + *i*)) (*a p*))));
              return (*i* + *1*)
            })
        0;
  return 1
}
  unfolding *access-arr-struct'-def*.
end

end
theory *AutoCorres-Documentation*
imports
  AutoCorresInfrastructure
  pointers-to-locals
  In-Out-Parameters-Ex
  fnptr
  union-ac
  open-struct
begin

end

Chapter 27

C-Translation Infrastructure

theory *CTranslationInfrastructure*
imports
CTranslation
begin

term *guarded-spec-body*

27.1 Local Variables

Local variables in the SIMPL outcome of the C-Parser are represented in *locals*. The natural number in the domain encodes the canonical position in the scope of the C-function: input arguments, then return variable, then local variables. The byte list in the range of the function is the encoding of the value according to the UMM (unified memory model) *c-type*. The typing of variables is maintained as data in `CLocals` and a typed view is achieved via *clookup* and *cupdate*.

In previous attempts local variables were represented first as a **record** and more recently as **statespace**. The representation as *locals* tries to combine the best of all previous approaches:

- Uniform (program independent) representation of local variables.
- Positional 'naming' enables a canonical parameter passing for function calls, even if the names of parameters are unknown, as in the case of function-pointers.
- Lightweight "typing" via ML infrastructure instead of the 'fixes' of **statespace**. Unfortunately, the more flexible **statespace** representation for local variables turned out to reveal some performance bottle neck in the locale infrastructure, which itself boils down to the resources consumed by the omnipresent instantiation of types and terms.

declare [[*ML-print-depth = 1000*]]

27.1.1 Basic ML primitives

To define new variables the basic primitive is `CLocals.define_locals` which takes a scope (currently a two element list of qualifiers: program-name followed by function-name) and the list of local variable declarations (name, type and kind). Note that the order is relevant as it represents the canonical positional ordering.

setup ‹*CLocals.define-locals* [*prog*, *myfun*] [
(*x*, @{*typ* ‹*32 word*›}, *NameGeneration.Loc*),
(*p*, @{*typ* ‹*32 word ptr*›}, *NameGeneration.Loc*),
(*z*, @{*typ* ‹*64 word*›}, *NameGeneration.Loc*)] ›

ML ‹*CLocals.Data.get* (*Context.Proof* @{*context*})›

For each local variable a constant is defined with the position of the variable. The constant name encodes the original name of the local variable. Whenever available we use this symbolic name in favour of the plain literal number. However, at some places we use the plain numbers, in particular for parameter passing. There is some simplifier setup which we explain later that allows to use either representation.

To enable syntax translations from short names to the symbolic constants together with their typing one has to enter the correct scope, e.g.:

local-setup ‹
Local-Theory.declaration {*pervasive=false*, *syntax=true*, *pos* = ***here***} (*fn* - =>
CLocals.scope-map (*K* [*prog*, *myfun*]))
›

term *prog.myfun.p-′*
thm *prog.myfun.p-′-def*

Some remarks on the naming scheme for those symbolic constants. The qualifiers for both program-name and function-name are mandatory. The name mangling with the suffix serves two purposes:

- Avoid name clashes with user input via variable names in the C sources (in particular for the artificial return variable)
- Avoid name clashes for bound or free variables (or new constants) in HOL. Note that even a constant with mandatory qualifier e.g. *foo.c* would force the pretty printer to avoid a bound variable being named just as *c*. The pretty printer only takes the `Long_Name.base_name` into account.

  The general suffix *-′* avoids clashes with C variable names as the prime is not a legal character for C identifiers. Syntax translations will truncate this suffix. The artificial return variable *ret* that the C-Parser introduces is internally *ret′-′* to avoid a potential conflict with a local variable named *ret*.

ML ‹*CLocals.Data.get* (*Context.Proof* @{*context*})›

27.1.2 Syntax

Plain lookup and update in locals.

term ‹$l \cdot p$›
ML-val ‹***term*** ‹$l \cdot p$››

term ‹$s\langle x := y\rangle$›
term ‹$s\langle x := y,\ p := k\rangle$›

ML-val ‹***term*** ‹$s\langle x := y\rangle$››
ML-val ‹***term*** ‹$s\langle x := y,\ p := k\rangle$››

Lookup and update in a complete Simpl state, selecting the *locals*

term ‹$s \cdot_{\mathcal{L}} p$›
ML-val ‹@{*term* ‹$s \cdot_{\mathcal{L}} p$›}›

term ‹$s\langle x :=_{\mathcal{L}} y\rangle$›
term ‹$s\langle x :=_{\mathcal{L}} y,\ p :=_{\mathcal{L}} k\rangle$›

27.1.3 Simplifier setup

The simplifier is setup to be able to deal with symbolic as well as literal numerals. Symbolic ones are expanded on the fly employing the infrastructure form the code generator. The trigger for this expansion is via a simplification procedure that. Conditional rules can be configured via the attribute:

ML ‹@{*attributes* [*code-simproc*]}›

The simplification procedure first checks wheter the precondition evaluates to true, via the code-generator. In case of success the simplifier is invoked to replay that proof and trigger the rule. `Code_Simproc.code_simp_prems`, `Code_Simproc.code_prove`.

Note that the attribute also consumes an optional name to index the simprocs. This name is taken from **named-theorems**.

The relevant setup of the rules can be found in `CLocals.thy`.

schematic-goal ‹(*clookup 2* (*cupdate prog.myfun.z-'* (λ-. *2*::*64 word*) *l*)) = (*?X*::*64 word*)›
  apply *simp*
  done

schematic-goal ‹(*clookup prog.myfun.z-'* (*cupdate prog.myfun.z-'* (λ-. *2*::*64 word*) *l*)) = (*?X*::*64 word*)›
  apply *simp*
  done
schematic-goal $s\langle x := \lambda\text{-}.\ 2\rangle \cdot x = ?X$

apply *simp*
done

schematic-goal $s\langle x := \lambda\text{-}.\ 2\rangle \cdot p = ?X$
apply *simp*
done

install-C-file *ex.c*

thm *size-td-simps*
thm *my-struct-C-update-const*

The root locale storing the global content *ex-global-addresses*. This is also the locale ware the bodies of the imported functions are defined.

context *ex-global-addresses*
begin

The usage of global variables is analysed and three cases are distinguesed:

- static variables do not change there value. So they are defined as HOL constant with their initial value. In case no initialiser is given they are zero initialised.
- 'ordinary' global variables that are also updated somewhere in the code are stored in the record of global variables *globals*. They are accessed by the lookup and update function of the **record** package.
- In case the address of a global variable is taken somewhere in the code, we declare a global constant for a pointer to that variable. Lookup and update is then indirectly via the heap. Note that the pointer is only declared. There is no defining equation. Currently there are also no assumptions maintained about distinctness of global variable pointers. This is up to the user.

thm *global-const-defs*
term *g-static*
term *g-ordinary-'* **term** *g-ordinary-'-update*
term *g-addressed-'*
thm *main-body-def*
thm *globals.typing.get-upd*
thm *state.typing.get-upd*

Functions also result in a declaration of a constant representing the global function pointer. *ex.add*. They have the type *unit ptr*.

term *ex.add*

When the code actually uses function pointers to perform indirect calls, some more infrastructure is provided, cf `../doc/fnptr.thy`

When looking into the function bodies the symbolic names for the positional local variables are displayed, fully qualified with program name as well as function name.

thm *add-body-def*

To have short names printed, one can either enter the scope of the function by activating the corresponding *variables* bundle, or use the option to always display short names.

context includes *add-variables*
begin
thm *add-body-def*
end

declare [[*clocals-short-names*]]
thm *add-body-def*
end

The canonical locale to do verification of a procedure is the *impl* locale. This activates the scope of the function and also stores the equation that the lookup of the function pointer in the environment Γ retrieves the expected body.

context *add-impl*
begin
thm *add-body-def*
thm *add-impl*

The symbolic names can be folded and unfolded by an attribute.

thm *add-body-def* [*unfold-locals*]
thm *add-body-def* [*unfold-locals*, *fold-locals*]

These attributes can also take qualifier in case an alternative scope should be added

thm *call-add-body-def*
thm *call-add-body-def* [*unfold-locals*] — nothing happens as we are in the wrong scope
thm *call-add-body-def* [*unfold-locals call-add*] — now they are expanded as we qualify the scope

end

context *call-add-impl*
begin
thm *call-add-body-def*
declare [[*hoare-use-call-tr'* = *false*]]
thm *call-add-body-def*
end

All the implementations of the program are gathered in the *simpl* locale

context *ex-simpl*
begin
thm *add-impl*
thm *call-add-impl*
end

27.2 Infrastructure for states

type-synonym *state* = (*globals*, *locals*, *32 signed word*) *CProof.state*

context *add-impl*
begin

As a final part of the verification generator generator terms containing local and global variables are generalised to a 'splitted' view, where each component becomes has a bound variable.

lemma
∃ *s*::*state*. *s* ·$_{\mathcal{L}}$ *n* = *3*
apply (*tactic* ‹*HPInter.GENERALISE* @{*context*} *1*›)
by *simp*

lemma
∃ *s*::*state*. *s* ·$_{\mathcal{L}}$ *n* = *3*
apply (*tactic* ‹*HPInter.GENERALISE* @{*context*} *1*›)
by *simp*

lemma
∃ *s*::*state*. *s* ·$_{\mathcal{L}}$ *n* = *3* ∧ *s* ·$_{\mathcal{L}}$ *m* = *3*
apply (*tactic* ‹*HPInter.GENERALISE* @{*context*} *1*›)
by *simp*

lemma
∃ *s*::*state*. *s* ·$_{\mathcal{L}}$ *n* = *3* ∧ *s* ·$_{\mathcal{L}}$ *m* = *3* ∧ *g-'* (*globals s*) = *4* ∧ *global-exn-var'-'* *s* = *Break*
apply (*tactic* ‹*HPInter.GENERALISE* @{*context*} *1*›)
by *simp*

Simpl syntax

ML ‹
@{*term* ‹{| ´*n* = *x* |}›}
›
term ‹{| ´*n* = *x* |}›
ML ‹
@{*term* ‹´*n* :== *x* ›}
›
term ‹´*n* :== *x* ›

term ‹´n :== *CALL add*(*x*, *y*)›
term ‹´n :== *CALL ex.add*(*x*, *y*)›
ML ‹@{*term* ‹´n :== *CALL add*(*x*, *y*)›}›

ML ‹@{*term* ‹´*ret'* :== *PROC add*(*2*, *3*)›}›

term ‹´*ret'* :== *PROC add*(*2*, *3*)›

end

27.3 Cached simproc examples

Some more explanation on this is in `../doc/AutoCorresInfrastructure.thy`

schematic-goal *TYPE*(*addr-bitsize word*) $\leq_\tau$ *TYPE*(*unpacked-C*[*12*])
apply *simp*
done

lemma *TYPE*(*8 word*) $\leq_\tau$ *TYPE*(*array-C*)
supply [[*verbose=3*]]
apply *simp*
done

lemma *TYPE*(*8 word*) $<_\tau$ *TYPE*(*array-C*)
supply [[*verbose=3*]]
apply *simp*
done

lemma *TYPE*(*addr-bitsize word*) $\leq_\tau$ *TYPE*(*matrix-C*)
supply [[*verbose=3*]]
apply *simp*
done

lemma *TYPE*(*8 word*) $\leq_\tau$ *TYPE*(*matrix-C*)
apply *simp*
done

ML ‹
Cached-Theory-Simproc.trace-cache @{*context*}
›

schematic-goal ‹
field-names-no-padding (*typ-info-t TYPE*(*outer-C*)) (*export-uinfo* (*typ-info-t TYPE*(*inner-C*)))
= *?XX*›
supply [[*simp-trace=true*, *verbose=5*]]
apply *simp*

done
ML ‹
Cached-Theory-Simproc.trace-cache @{*context*}
›

schematic-goal ‹
set (*field-names-no-padding* (*typ-info-t TYPE*(*outer-C*)) (*export-uinfo* (*typ-info-t TYPE*(*inner-C*)))) = *?XX*›
supply [[*simp-trace*=*true*, *verbose*=*5*]]
apply *simp*
done
ML ‹
Cached-Theory-Simproc.trace-cache @{*context*}
›

lemma *export-uinfo* (*typ-info-t TYPE*(*unpacked-C*)) = *export-uinfo* (*typ-info-t* (*TYPE*(*32 word*)))
supply [[*simp-trace*=*true*,*verbose*=*3*]]
apply *simp*
oops

lemma *typ-uinfo-t TYPE*(*unpacked-C*) = *export-uinfo* (*typ-info-t* (*TYPE*(*32 word*)))
apply *simp*
oops

lemma *typ-uinfo-t TYPE*(*32 signed word*) = *export-uinfo* (*typ-info-t* (*TYPE*(*32 word*)))
supply [[*simp-trace*=*true*,*verbose*=*3*]]
apply *simp*
done

lemma *typ-uinfo-t TYPE*(*32 signed word*[*3*]) = *export-uinfo* (*typ-info-t* (*TYPE*(*32 word*[*3*])))
supply [[*simp-trace*=*true*,*verbose*=*3*]]
apply *simp*
done

schematic-goal ‹
set (*field-names-no-padding* (*typ-info-t TYPE*(*unpacked-C*)) (*export-uinfo* (*typ-info-t TYPE*(*8 word*)))) = *?XX*›
supply [[*simp-trace*=*true*, *verbose*=*3*]]
apply *simp*
done

lemma *export-uinfo* (*typ-info-t TYPE*(*unpacked-C*)) = *export-uinfo* (*typ-info-t* (*TYPE*(*unpacked-C*)))
supply [[*simp-trace*=*true*,*verbose*=*3*]]

apply *simp*
done

ML ‹
Cached-Theory-Simproc.trace-cache @{*context*}
›
lemma *TYPE* (*32 word*) $\leq_\tau$ *TYPE*(*32 word*[*256*])
supply [[*simp-trace=true*,*verbose=3*]]
apply *simp*
done

lemma *TYPE* (*32 word*) $\leq_\tau$ *TYPE*(*32 word*[*256*])
supply [[*simp-trace=true*,*verbose=3*]]
apply *simp*
done
ML ‹
Cached-Theory-Simproc.trace-cache @{*context*}
›

schematic-goal *n* < *12* ⟹ *field-lookup* (*typ-info-t TYPE*(*array-C*)) [*''elements-C''*, *replicate n CHR ''1''*, *''chr-C''*] *0* = *?X*
supply [[*simp-trace=true*, *simp-trace-depth-limit=1*, *verbose=5*]]
apply *simp*
done

schematic-goal *n* < *12* ⟹ *field-lookup* (*typ-uinfo-t TYPE*(*array-C*)) [*''elements-C''*, *replicate n CHR ''1''*, *''chr-C''*] *0* = *?X*
supply [[*simp-trace=true*, *simp-trace-depth-limit=1*, *verbose=5*]]
apply *simp*
done

schematic-goal *n* < *12* ⟹ *field-ti TYPE*(*array-C*) [*''elements-C''*, *replicate n CHR ''1''*, *''chr-C''*] = *?X*
supply [[*simp-trace=false*, *simp-trace-depth-limit=1*, *verbose=3*]]
apply *simp*
done

schematic-goal *n* < *12* ⟹ *field-ti TYPE*(*array-C*) [*''elements-C''*, *replicate n CHR ''1''*, *''chr-C''*] = *?X*
supply [[*simp-trace=false*, *simp-trace-depth-limit=1*, *verbose=3*]]
apply *simp*
done

lemma *TYPE*(*unpacked-C*[*12*]) $\leq_\tau$ *TYPE*(*array-C*[*2*])
supply [[*simp-trace=false*, *simp-trace-depth-limit=1*, *verbose=3*]]
apply *simp*
done

lemma ¬ *TYPE*(*unpacked-C*[*12*]) $\leq_\tau$ *TYPE*(*unpacked-C*[*24*])

supply [[*simp-trace=false*, *simp-trace-depth-limit=2*, *verbose=3*]]
apply *simp*
done

lemma *TYPE*(*unpacked-C*[*12*]) $\leq_\tau$ *TYPE*(*array-C*)
apply *simp*
done

lemma *typ-uinfo-t TYPE*(*unit ptr*) $\neq$ *typ-uinfo-t TYPE*(*32 word ptr*)
apply *simp*
done

Matching of the interpreteted locale *stack-heap-raw-state*

ML-val ‹
val cterm-match = With-Fresh-Stack-Ptr.cterm-match @{context}
val {n=n1, init=init1, c=c1, instantiate=inst1, ...} = cterm-match @{cterm state.With-Fresh-Stack-Ptr n i1 c1}
val {n=n2, init=init2, c=c2, instantiate=inst2, ...} = cterm-match @{cterm state.With-Fresh-Stack-Ptr n i2 c2}
val t = inst1 {n=n2, init = init2, c= c2}
›

ML-val ‹
val match = With-Fresh-Stack-Ptr.match @{context}
val {n = n1, init=init1, c=c1, instantiate=inst1, ...} = match @{term globals.With-Fresh-Stack-Ptr n i1 c1}
val {n = n2, init=init2, c=c2, instantiate=inst2, ...} = match @{term globals.With-Fresh-Stack-Ptr n i2 c2}
val t = inst1 {n=n2, init = init2, c= c2}
›

thm *zero-simps*
thm *make-zero*
thm *size-simps*
thm *size-align-simps*

thm *h-val-fields*
thm *heap-update-fields*
thm *fg-cons-simps*
thm *fl-ti-simps*
thm *fl-Some-simps*

end

Chapter 28

Translation of the StrictC Dialect (Outdated)

Michael Norrish

28.1 Introduction

This report describes the translation program that imports C source into a running Isabelle/HOL process, making a series of Isabelle definitions in the process, as well as discharging a number of (relatively minor) proof obligations that arise along the way.

The source code for the translator is found in the directory `c-parser`. Source files, *e.g.*, `file.ML`, are found in this directory unless otherwise noted.

The translation expects its input to be a well-formed C source file. Such a source file must additionally satisfy a number of other constraints, giving rise to a subset of C that is here called "StrictC". Files in StrictC may also include special annotations intended only for consumption by Isabelle (and the human code-verifier).

In fact, there are two important StrictC programs: the translation program, and the analysis program. The analysis program is entirely independent of Isabelle, and can be used to check that a source file conforms to the StrictC subset. It also implements a number of analyses that can be performed on source code. For example, it can output the input file's call-graph, and can list the globals that are read or modified in each function. The additional source-files supporting the analysis program are found in

```
c-parser/standalone-parser
```

The rest of this report describes both the functionality provided by these programs (focussing on the translator), how this functionality is implemented, and *where* it is implemented. The aim is to give a picture of the systems' design in a way that should make future modification of the code possible.

28.1.1 StrictC Subset Summary

This is a brief list summarising the simplest restrictions imposed by the StrictC subset.

- No `goto` statements.
- No fall-through cases in `switch` statements. Cases can be terminated with `continue` and `return`, as well as `break`.
- Labels for `switch` statement cases must all appear at the syntactic level immediately below the block statement that must appear below the `switch` statement.
- No unions. (These are handled by a separate tool; see Cock [1].)
- No `struct`, `enum` or `typedef` declarations anywhere except at the top, global, level.

28.2 Abstract Syntax

The core data types in the translator represent the input program. These abstract syntax values are the product of parsing the concrete syntax (see Appendix 28.6 for the grammar used), and is the subsequent input to all further analyses and translation. The abstract syntax declarations are given in `Absyn.ML`. For example, the definition of the syntax type corresponding to C statements is given in Figure 28.1. There are also definitions for C types, expressions, and declarations in `Absyn.ML`.

The type `statement` is actually a `statement_node` wrapped inside a "region" (see `Region.ML` and Section 28.2.1 below), which provides information about where the original concrete syntax originated. This is used for providing error messages.

All strings in the statement declaration correspond to Isabelle terms (*e.g.*, the loop invariant in the `While` case). These will be parsed as such later in the translation process, but are just uninterpreted strings when the C parser finishes. Function calls can return their results into l-values, have the return value ignored, or have the return value itself `return`-ed. The first option corresponds to having the `expr option` argument be `SOME e` in the `AssignFnCall` constructor, the second would have that parameter be `NONE`, and the last is handled by the `ReturnFnCall` constructor.

Syntactically, these options correspond to writing

```
var = f(x,y);
```

or

```
f(x,y);
```

```
datatype statement_node =
      Assign of expr * expr
    | AssignFnCall of expr option * expr * expr list
    | EmbFnCall of expr * expr * expr list
    | Block of block_item list
    | While of expr * string wrap option * statement
    | Trap of trappable * statement
    | Return of expr option
    | ReturnFnCall of expr * expr list
    | Break
    | Continue
    | IfStmt of expr * statement * statement
    | Switch of expr * (expr option list * block_item list) list
    | EmptyStmt
    | Auxupd of string
    | Spec of ((string * string) * statement list * string)
    | AsmStmt of {volatilep : bool, asmblock : asmblock}
and statement = Stmt of statement_node Region.Wrap.t
and block_item =
    BI_Stmt of statement
  | BI_Decl of declaration wrap
```

Figure 28.1: The Abstract Syntax Data Type for C Statements

or

```
return f(x,y);
```

C99 Block Items Conforming to the C99 grammar, the input language allows declarations at any point inside a block, not just in a sequence at the head of the block. In other words, the following

```
{
  x = f(z);
  int y = x + 1;
  while (x < y) { ... }
}
```

is legal in C99. This means that a block has to take a list of `block_item` values as an argument, where a `block_item` is either a statement or a declaration.

Syntactic Sugar for Loops The abstract syntax has just one form of loop, the `While` constructor. The optional string argument to `While` is used to represent any user-supplied invariant. Together with the `Trap` constructor, this is used to implement all three forms of C loop. The translation follows the model from Norrish's PhD [6, p60]. It also supports `for`-loops with declarations in the first position. This latter, for example,

```
for (int i = 3; i < 10; i++) ...
```

is a feature of C99.

Breaking from C, the grammar has the third component of the `for` loop form be a restricted form of statement, rather than an expression. The parser syntax only allows comma-separated increments (*i.e.*, `++`), decrements and assignments.

The Isabelle translation eventually compiles all loops to one underlying the loop primitive in the VCG environment called `While`. This form does not handle exceptional control-flow forms like `break` and `continue`. These are instead handled by the `Trap` constructor, mapping to the VCG language's `TRY-CATCH` form.

28.2.1 Regions

The `Region` module implements a method for annotating arbitrary data types with location information. This module has been taken from the `MLton` compiler project (which has a BSD-style open source licence). It is used a great deal in the system, and its use could probably be extended still further. Region information is used to produce good error messages.

The basic type is that of the `region` which is essentially a pair of "source positions" (which are in turn implemented in `SourcePos.ML`). One useful source position is `SourcePos.bogus`, corresponding to "nowhere" (perhaps because some syntax has been conjured out of nowhere and doesn't really exist in a file).

Regions are then used to implement the concept of a "wrap" (SML type `Region.Wrap.t`), a polymorphic data type. The file `Absyn.ML` declares the following abbreviation:

```
type 'a wrap = 'a Region.Wrap.t
```

An `'a wrap` (read "alpha wrap") is an `'a` value coupled with a region. The important functions for manipulating wraps are

```
val wrap    : 'a * SourcePos.t * SourcePos.t -> 'a wrap
val bogwrap : 'a -> 'a wrap
val left    : 'a wrap -> SourcePos.t
val right   : 'a wrap -> SourcePos.t
val node    : 'a wrap -> 'a
val apnode  : ('a -> 'b) -> 'a wrap -> 'b wrap
```

For example, the grammar code in `StrictC.grm` manipulates a number of values of type `string wrap`. If an error relating to this value arises, both the string and its position can be reported to the user.

Things become more complicated when the type to be wrapped is recursive. The standard idiom in the project is illustrated with the definition of the `statement` data type (shown in Figure 28.1). The constructors for values of the type are actually given in an auxiliary type (`statement_node`), but the recursive constructors take arguments of type `statement`. The type `statement` is then a type that is mutually recursive with `statement_node`, and which has just one constructor, `Stmt`.[1]

Because a `statement` is not a wrap, the project cannot directly call functions like `node` on `statement` values. Instead, helper functions are declared:

```
val sleft  : statement -> SourcePos.t
val sright : statement -> SourcePos.t
val snode  : statement -> statement_node
val swrap  : SourcePos.t * SourcePos.t * statement_node ->
             statement
```

When code wishes to pattern-match against the multiple possible forms a statement `s` may have, the idiom is to write

[1] It would be nice if one could just declare `statement` to be an abbreviation of `statement_node wrap`, but SML doesn't permit this. It must be a `datatype` itself, and thus must have at least one constructor.

```
case snode s of
  EmptyStmt => ...
| While(g,inv,body) => ...
| IfStmt(g,ts,es) => ...
| ...
```

The strength of this idiom is that one *always* manipulates values of type `statement`. In particular, if the case analysis above is to make recursive calls of its analysis on sub-statements such as `body`, `ts` and `es`, these values are of the correct type for this to be done immediately.

The `expr` (expression) type (home to constructors such as `Deref`, `Var` and `TypeCast`) is set up in the same style, giving rise to functions `eleft`, `eright`, `enode`, `ewrap` and `ebogwrap`.

The type of C types The type of C types is `'a ctype`, with constructors such as `Void` and `Ptr`. Though recursive, it doesn't use the wrap idiom. On the other hand, this type is polymorphic. The `'a` parameter is instantiated with an SML type that corresponds to the forms that give the size of arrays when they are declared. This type parameter is instantiated with `expr` when the input file is first parsed. In this way, a declaration like

```
unsigned char array[EnumConst1 * sizeof(int*)];
```

can be handled. Thus, the `VarDecl` constructor of the `declaration` type

```
val VarDecl :
      expr ctype * string wrap * bool * initializer option ->
      declaration
```

takes an `expr ctype` as its first parameter. Within subsequent phases of the analysis and translation, it is much more convenient to work with values of type `int ctype`, where the (constant) expression has been evaluated. For example, the `get_rettype` function from `program_analysis.ML`, takes a `csenv` value (see Section 28.3 below) and the name of a function, and returns the return type of the function. The value returned is an `int ctype`. The conversion of an `expr ctype` into an `int ctype` is done by the function `Absyn.constify_abtype`.

28.3 The Symbol Table

All of the work done in analysis and translation of StrictC revolves around the information stored in two important data structures implemented in the module `program-analysis.ML`. The first of these is the `var_info` type. This stores information about individual variables. The second type, `csenv` ("C state environment"), stores information about the program as a whole,

including its collection of variables, but also recording details such as where variables are read and modified, and the program's call-graph structure.

The `var_info` type stores information about declared identifiers living in the name-space that encompasses normal objects, functions and enumeration constants. In addition to the type of the variable (*e.g.*, `int`, `char *` *etc.*), the `var_info` also includes information about where the variable was declared in terms of program locations, and also in terms of scope (it might be global, or declared local to a particular function).

The `csenv` type accumulates its information about the program by performing traversals of the abstract syntax tree. **The StrictC translator makes no effort to be a one-pass compiler**, but the number of traversals is no greater than three, and will probably be reduced in future versions of the implementation. These traversals are performed after the parser has constructed all of the tree. There are also places in this analysis where **the translator assumes that it has seen the whole program**. In particular, the translator cannot be used to translate *translation units* that are to be separately compiled. It must be presented with a concatenation of the complete sources.

The bulk of the API for manipulating values of type `csenv` is concerned with pulling information out of the symbol table. For example, it is possible to calculate the type of a C expression with the function

```
val cse_typing : csenv -> expr -> int ctype
```

In addition, `program-analysis.ML` contains the one entry-point for taking a sequence of external declarations (once parsed) and creating a `csenv` value:

```
val process_decls :
  Absyn.ext_decl list ->
  ((Absyn.ext_decl list * Absyn.statement list) * csenv)
```

The return type includes a modified version of the syntax that was provided as input, a list of the initialising assignments for the global variables, and the `csenv` value.

28.3.1 Functional Record Updates in SML

The code in `program-analysis.ML` uses a powerful, but cryptic SML idiom that makes it easy to define SML records along with functions for updating their fields. Done naïvely, writing code to do this represents a quadratic amount of work for the programmer. Put another way, the naïve approach requires $O(n)$ much typing whenever a field is added to or removed from a record definition of n fields.

The cryptic technique is fully described at

http://mlton.org/FunctionalRecordUpdate

and allows the addition or deletion of a field to be done with $O(1)$ much typing. Supporting code is in `FunctionalRecordUpdate.ML`.

The cryptic code is isolated within `program-analysis.ML`, where it is used to define update functions that are subsequently used exclusively. In general, when one of the two types has a field `fld` of type τ, then there will typically be a function {`cse`|`vi`}`_fupd_fld` defined, with type

$$(\tau \to \tau) \to \mathsf{rcd} \to \mathsf{rcd}$$

where rcd is either `var_info` or `csenv`. Such functions can be used to update the fields of a record: the user provides a function that is given the old value of the field, and which returns the new value.

28.4 Creation of the Hoare Environment State

The major oddity about Norbert Schirmer's Hoare environment, into which we are translating our programs, is that all local variables, including function parameters, have to be part of the "global state". This state must be declared before any functions can be translated, because a function becomes an Isabelle definition (conceptually at least) that operates over that state space.

Slightly simplifying, the state space is an Isabelle type that is a record with fields for every local and global variable. Each field has a type (Isabelle/HOL is a typed logic after all), which means that all local variables of the same name in the same StrictC translation unit must have the same type. This is rather an arbitrary requirement, but easy both to enforce and to comply with. Thankfully, signed and unsigned variants of the same underlying type (such as `signed short` and `unsigned short`) are given the same Isabelle/HOL type, so there is a little leeway. Nonetheless, if `i` is of type `int` in one function, it can not be of type `char` in another.[2]

The arrangement of global and local variables is actually slightly complicated. In essence, the state-space is set up to look like:

```
statespace = record
   globals :: global_var_type
   local_var1 :: lvar1_type
   local_var2 :: lvar2_type
   ...
   local_varn :: lvarn_type
```

[2]If this is attempted, the system will "munge" one of the variables so that it has a different name when translated into Isabelle. The "munged" name is stored in the variable's `var_info` record.

where the field `globals` is of a custom record type `global_var_type` that in turn contains all of the global variables. In addition to the user program's own globals, the translation process adds two extra global variables of its own. These variables are used for handling "exceptional exits" (such as those caused by the `break`, `continue` and `return` statements), and for modelling the global heap. The exact names of these variables is not important, here we will refer to them as `global_exn_var`, and `global_heap`.

These two special variables are part of `global_var_type` and so must have Isabelle types themselves. The type of `global_exn_var` is the enumerated type `c_exntype`, defined in the Isabelle theory `CProof` to have three possible values, `Break`, `Return` and `Continue`. The type of the global heap `global_heap` field is a product of underlying heap contents (a map of type `addr` $\rightarrow$ `word8`) and a special-purpose data type to store type information about the heap memory (see Harvey Tuch's PhD thesis [9] for more on this).

28.4.1 Representing Values in Memory

There is one important requirement that must be met by all object types that occur in C programs: they must be representable in memory. Alternatively, it must be possible to encode values of the types in a program as sequences of bytes, and to then decode those same bytes back into the original values. One approach to modelling this fundamental requirement might be to have Isabelle functions that manipulated only lists of bytes. Working at the level of this untyped, and very concrete, view of the program state would be an extremely poor state of affairs (the C programmer would have a more abstract view of the program than the verifier).

Our approach is to use Isabelle type-classes to encode the fact that an Isabelle type can be represented in a consistent amount of "C memory". When an Isabelle type `'a` is in the class `mem_type`[3], it supports functions

```
to_bytes   :: "'a::mem_type => byte list"
from_bytes :: "byte list => 'a::mem_type"
```

as well as a number of other supporting functions that record details like the fields that occur in compound `struct` types, and the (constant) length of the byte-lists that encode the values in the type.

All of the atomic types manipulated by our programs are fixed-width words (*e.g.*, 32 bit words for integers). It is easy to demonstrate that these types are indeed in the `mem_type` class. In particular, we do *not* pretend that programs manipulate infinite precision integers, and then worry about whether or not these integers can be pushed into and pulled out of memory. All of the arithmetic performed by the programs we verify is done at fixed widths, respecting the underlying machine's operations. Additionally, using

[3] See `umm_heap/CTypes.thy`.

the techniques from Section 28.5.1, we trap the undefined behaviour caused by overflow on signed values.

28.4.2 Pointers

Pointers are always represented as words of a particular size (regardless of type being pointed to). This is not required by the C standard, which only requires pointers to `void` be capable of storing all other non-function pointer values, and that all function pointer values be inter-convertible. Again, our decision to specialise on particular, and reasonable, target architectures makes life simpler.

Pointers retain type information by using "phantom" type variables. The Isabelle declaration is

```
datatype 'a ptr = Ptr addr
```

Then, if the C program under analysis calls for a variable of type `char *`, the Isabelle environment will include a variable of Isabelle type `byte ptr`. In this way our pointers are typed, even if their underlying encoding makes it trivial to view a pointer to one type as a pointer to another type.

Because the underlying representation is always the same, all pointer types are proved to be in the class `mem_type` once and for all (in theory `CTypes`). Pointers to `void` are represented as values in the Isabelle type `unit Ptr`, where `unit` is the standard singleton type. The `unit` type is not shown to be in the class `mem_type`.

28.4.3 Arrays

C arrays are lists of values of fixed length. A faithful representation of this type requires a novel Isabelle type. We build on Anthony Fox's implementation of John Harrison's "finite Cartesian products" idea [4]. Syntactic trickery within Isabelle allows us to write types like

```
nat[10]
```

which is an array of 10 natural numbers. There are operators for updating and indexing into arrays. Note that the type `10` that appears above is an Isabelle *type*, not a term.

If type τ is a `mem_type`, then an array of τ values will also be a `mem_type`, as long as the size of the array is not so large that the array would not fit into memory. This condition is discharged as the StrictC program is translated.

For technical reasons due to the implementation of type classes in Isabelle, we need to fix separate limits, ahead of time, on the number of elements in an array and the size of each element. Currently, for 32-bit ARM, our model fixes a maximum of:

```
struct listnode {
   int node_data;
   struct listnode *next;
};
```

```
struct node2;
struct node1 {
   struct node2 *data;
   struct node1 *next, *prev;
   int someflag;
};
struct node2 {
   struct node1 *owner;
   char stringdata[100];
};
```

Figure 28.2: Examples of Recursive `struct` Declarations Accepted in StrictC

- 2^{13} (8192) elements in each array; and
- 2^{19} bytes (512 KiB) in each array element

For x86-64, the limits are:

- 2^{20} (1048576) elements in each array; and
- 2^{26} bytes (64 MiB) in each array element

One would prefer to be able to multiply the size of the element type by the number of the elements, but the type system does not permit this (for good technical reasons).

28.4.4 C `struct` Types

From the point of view of the translation into Isabelle, `struct` types do not pose any great conceptual difficulties. A `struct` type is clearly very similar to an Isabelle record, which in turn is conceptually the same as a tuple. The first complication that arises is that Isabelle tuples can not be recursive, whereas C `struct` types are often recursive (as when implementing linked structures in the heap).

This required the implementation of an alternative record definition package, allowing (possibly multiple) recursive types. This then allows Isabelle types to be defined that correspond to C declarations such as those shown in Figure 28.2.

Confirming that a `struct` type really is representable in memory, requires the definition of functions for converting Isabelle records into lists of bytes and *vice versa*. The size of the converted value must also be checked to be no bigger than the size of memory. Both of these actions require knowledge of how the fields of the `struct` are laid out in memory, which is in turn a function of the padding that can be inserted between the fields. Such calculations are architecture dependent.

28.5 Translating Expressions

Expression translation is implemented in `expression_translation.ML`.

Fundamental Concepts StrictC expressions are essentially a subset of C expressions, and are fairly easy to translate to corresponding Isabelle expressions that manipulate Isabelle-encoded values. There are two fundamental concepts to grasp of expression translation. First, when being evaluated ("read to determine a value") a C expression of type τ becomes an Isabelle expression of type `statespace` $\rightarrow [\![\tau]\!]$, where $[\![\tau]\!]$ is the translated, Isabelle version of C type τ.

This must be done to make sense of expressions that read memory: an expression such as `s.arrayfld[3]` only has a specific value in the context of a specific state of memory. This use of a "lifted" function space to represent the expression is a standard technique in denotational semantics. In the example given, the value of the expression is a function that looks at the statespace to determine what data is stored at variable `s`. As the statespace evolves, the value returned from this function changes, but the function's value is the same.

Expressions do not just determine values however, they can also denote an *l-value*, something denoting a "place in memory" that is to be updated. This is done when an address is taken, or when an assignment is to be performed. In a simple language, l-values might only be variable names, but C allows for complicated expressions on both sides of an assignment. The example above (`s.arrayfld[3]`) might just as well be assigned to as read. So, the l-value of an expression e that has type τ will be an Isabelle value of type `statespace` $\rightarrow \tau \rightarrow$ `statespace`, a function that takes a new value and a statespace to change, and returns the updated statespace.

Not all expressions are l-values (the expression `3` is not, for example), so the translation of any one expression can return one or two different values, an "r-value" as well as an optional l-value.

In addition, because of the way the translation does not put local variables into memory, the translation provides another separate optional value, that of the expression's address. If everything did live in memory, all l-values would have addresses, and one could dispense with the separate calculation of l-values. Thus the first three lines of Figure 28.3.

The SML types given to the `lval`, `addr` and `rval` fields in the declaration of `expr_info` are themselves function spaces at the SML level. These function spaces manipulate values of SML type `term`. The way in which typing at the C level is reflected at the Isabelle level is mainly hidden at the SML level, where the programmer just manipulates terms (the Isabelle types are internal to those values).

However, the function spaces *can* be made visible at the SML level. This is done mainly to reduce the number of β-redexes that would otherwise be

```
datatype expr_info =
  EI of {lval   : (term -> term -> term) option,
         addr   : (term -> term) option,
         rval   : term -> term,
         cty    : int ctype,
         ibool  : bool,
         lguard : (term -> term * term) list,
         guard  : (term -> term * term) list,
         left   : SourcePos.t,
         right  : SourcePos.t }
```

Figure 28.3: The `expr_info` type, into which C expressions are translated internally.

created in the resulting term. For example, translating the C expression `x + 3` will first create r-values

$$\begin{aligned} [\![\mathtt{x}]\!] &= (\lambda\sigma.\ \mathtt{x}(\sigma)) \\ [\![\mathtt{3}]\!] &= (\lambda\sigma.\ 3) \end{aligned}$$

where the application of the `x` function to a statespace σ pulls out the value of variable `x` in σ.

When translating an expression $e_1 + e_2$, one naturally creates the value

$$\lambda\sigma.\ [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma)$$

If this was done within Isabelle `term` values, the result would be a `term` full of expressions of the form $(\lambda v.\ M)N$. By lifting the Isabelle λ to the SML level, only one abstraction need to be created, at the very top-level in the translation. Thus, the translation of the addition becomes the SML expression

```
(fn s => mk_plus (rval_of e1 s) (rval_of e2 s))
```

where `rval_of` returns the `rval` field of an `expr_info` and the β-reduction happens within SML.

The fourth line of the record in `expr_info` values stores the type of the expression, something that informs the translation of many different C expressions. For example, additions are not as simple as just presented because of the possibility that one of the arguments might be a pointer. The `left` and `right` fields of the `expr_info` record store the source-code position of the original expression. The two guard fields are explained below in Section 28.5.1.

The `ibool` field of the `expr_info` records whether or not the term being generated in the r-value is of Isabelle's boolean type. This is done so that

translation can avoid some conversions between Isabelle words and booleans. For example, if the expression being translated is `x < 6 && y > 10`, the resulting Isabelle term will include a use of the two comparison operators on words. Strictly, one should then turn the boolean results of these operators into either a one or zero. But, as these results are then combined with boolean conjunction, the values will immediately be converted back into booleans.

Of course, in an expression such as `x && x->fld > 6`, the first operand to the conjunction does not have Isabelle boolean type, and will be converted to a boolean value by comparing it with the null pointer. (In fact, this particular example causes a more complicated translation effect to occur; see Section 28.5.1 below on undefined behaviour.) Dually, if the code puts a boolean value into memory, the translation has to make sure that the Isabelle term has the appropriate word type once more. There are two functions used here:

```
mk_isabool : expr_info -> expr_info
strip_kb   : expr_info -> expr_info
```

The function `mk_isabool` produces an `expr_info` value that does have Isabelle boolean type (it will be the identity function on an r-value that is already known to be boolean). The function `strip_kb` reverses this, turning an Isabelle boolean into an Isabelle word if necessary.

28.5.1 Undefined Behaviour

There are three classes of under-specification in the C standard. Those classed as *implementation defined* are behaviours or values that are supposed to be fixed and documented by particular implementations. For example, the number of bits in an `int` (a number that must be at least 16), is a fixed value that implementations are allowed to choose themselves. Because StrictC targets a particular architecture, most implementation-defined aspects of C can be "baked into" the translation. (The design attempts to have the translation sources be easy to modify to account for different architectures; see for example the `ImplementationNumbers` structure in each of the `TargetNumbers.ML` files.)

The second class of under-specification comprises *unspecified* behaviours. The most important of these is the order of evaluation of arguments to operators and function calls. Therefore, one should avoid writing code that depends on unspecified evaluation order. Otherwise, the compiled executable might have different semantics compared to the Simpl specification that we generate. For example:

```
int x = f() + g();
```

where `f` and `g` both have side effects, is currently allowed but should be avoided. In the future, we plan to forbid or restrict programs that may have unspecified evaluation order. The standalone analysis tool can check for this problem using the `-embedded_fncalls` option.

The third class of under-specification is *undefined behaviour*. In essence, undefined behaviours are runtime errors (such as dereferencing a null pointer). They are undefined because the standard does not require the implementation to catch them, or to realise that they have occurred. Rather, if an undefined behaviour occurs, the user can no longer rely on their program to do anything sensible. In effect, implementations are given licence to blunder on however they like when an undefined behaviour occurs.

Thus, it is critical that the C code we verify never exhibits any undefined behaviours. We do this with a feature of the Hoare environment called the *guard*. A guard is a boolean condition g attached to a statement s, with the combination written $g \to s$. If the guard g is true when the combined statement is to be executed, then s is allowed to execute. If it is false, the underlying semantics defines the result to be a "fault". When a program faults, nothing more can happen, so it has effectively aborted. Verifying a program with guarded statements thus requires proofs that the guards are always true.

Most guards arise in expressions, and the translation process accumulates these in one top-level guard at the statement level. For example, in the statement

```
x = *p >> i;
```

there will be four guards attached to the statement that will need to be discharged: that `p` is not null, that `*p` is not negative, that `i` is not negative, and that `i` is less than 32.

One elegant feature of guards in the Hoare environment is that they can be selectively disabled for the purposes of verification. For example, the C standard's requirement that right shift operations might not be performed on negative numbers (that it results in undefined behaviour) is too strict, given a particular C compiler and target architecture. In this situation, it is possible to prove Hoare-triples where that particular guard is not used, so that the semantics of $g \to s$ is simply that of s.

28.6 Concrete Syntax: Parsing and Lexing

The grammar for StrictC is given in the file `StrictC.grm`, which is given as input to the standard tool `mlyacc`. The format of an `mlyacc` file looks quite similar to the format accepted by `yacc`. A typical grammar rule is

```
init_declarator_list
: init_declarator
```

```
                ([init_declarator])
| init_declarator YCOMMA init_declarator_list
                (init_declarator :: init_declarator_list)
```

where a non-terminal appears before a colon, and multiple possible right-hand sides are separated by the pipe or vertical bar character. The code for a production appears in parentheses after each right-hand-side. The code's convention is to have token names (*e.g.*, `YCOMMA` above) be upper-case.

Apart from the changes that turn assignment expressions into statements, the grammar for StrictC attempts to be as close as possible to the grammar of the C standard. In general, the names for non-terminals in `StrictC.grm` are taken from the standard, so it should be fairly clear how the standard's grammar has been mapped into `StrictC.grm`.

28.6.1 Lexing and `typedef` Names

The standard problem in lexing and parsing C is that the grammar is ambiguous: the non-terminal *typedef-name* is defined to simply be the same as an *identifier*. When the parser encounters

```
x * f(y);
```

it can't know if this is meant to be a multiplication of variable `x` by a function call expression, or whether it is the (prototype) declaration of a function called `f` taking an argument of type `y` and returning a value of type `x`.

To resolve this, the lexer must be able to classify identifier tokens as normal identifiers or *typedef-names*. The StrictC translator's approach to this problem is not typical, because of the strange way in which `mlyacc` combines handling of side effects with its error correction. Normally, one would have some sort of updatable symbol table that the parser would write to when it encountered a `typedef` declaration. The lexer would read from the same table as it encountered identifiers, allowing appropriate categorisation.

The approach taken in the StrictC translator is to have the lexer do all of the work, without reference to the parser (see `StrictC.lex`). This *is* possible, but is also convoluted, and involves many updatable variables that are internal to the lexer. The basic idea is that when the lexer sees a `typedef` token, it switches to the `TDEF` state. When an identifier is seen in this state, the identifier is added to the list of *typedef-names*, and lexing can continue. The complications arrive in declarations like

```
typedef struct s { int fld1; char fld2; } s_t;
```

where the identifiers `s`, `fld1` and `fld2` have to be ignored, and `s_t` taken as the new *typedef-name*. This requires the lexer to handle the matching brace characters, and partly motivates the requirement that `typedef` declarations all occur at the top-level (and not be nested).

28.6.2 GCC `__attribute__` Declarations

The parser handles, but mostly ignores, various `gcc`-specific extensions, such as `__attribute__`. These are tricky to parse (something admitted by the relevant `gcc` documentation): users are given almost unlimited liberty to put their `__attribute__` strings anywhere within a declaration.

For example, these three

```
int f(int) __attribute__((__const__));
__attribute__((__const__)) int f(int);
int __attribute__((__const__)) f(int);
```

are all supposed to parse successfully (and have the same meaning). Making this work is rather involved. Most attributes are ignored, but the standalone analysis tool does check that `const` and `pure` attributes are reasonable, given what it knows of how functions may modify and read the global state.

Bibliography

[1] D. Cock. Bitfields and tagged unions in C: Verification through automatic generation. In B. Beckert and G. Klein, editors, *Proc, 5th VERIFY*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, Aug. 2008.

[2] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. 7406:99–115, Aug. 2012.

[3] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of C code without the pain. pages 429–439, June 2014.

[4] J. Harrison. A HOL theory of Euclidean space. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2005.

[5] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.

[6] M. Norrish. *C Formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998. Also published as Technical Report 453, available from http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf.

[7] M. Norrish. C-to-Isabelle parser, version 1.13.0, May 2013. Accessed May 2016.

[8] N. Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, Feb. 2008. Formal proof development.

[9] H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Aug 2008.