A formal semantics for Ivory with proofs

Simon Winwood Trevor Elliott
Lee Pike
{sjw,trevor,leepike}@galois.com

July 21, 2014
git revision a298674

Contents

1 Introduction

This is a document produced automatically by the Isabelle theorem prover from the theory development available at

https://github.com/GaloisInc/ivory/tree/master/ivory-formal-model.

This document is provided primarily for those who do not wish to install Isabelle. We apologize for any illegibility caused by the document's automated generation.

2 General preliminaries

class *infinite* =
  fixes *fresh* :: *'a set* ⇒ *'a*
  assumes *fresh-not-in*: *finite S* ⟹ *fresh S* ∉ *S*

instantiation *nat* :: {*infinite*}
begin

definition
  nat-fresh-def: *fresh S* = (*if S* = {} *then 0 else Suc* (*Max S*))

instance
proof
  fix *S* :: *nat set*
  assume *finite S*
  show *fresh S* ∉ *S*
  proof
    assume *fresh S* ∈ *S*
    with ‹*finite S*› **have** *Suc* (*Max S*) ≤ *Max S*
      unfolding *nat-fresh-def*
      by (*auto dest*!: *Max-ge split*: *split-if-asm*)

    thus *False* **by** *simp*

qed
qed
end

definition
submap-st :: $('var \Rightarrow 'c\ option) \Rightarrow ('var \Rightarrow 'fun\ option) \Rightarrow ('fun \Rightarrow 'c \Rightarrow bool) \Rightarrow bool$
where
$submap\text{-}st\ m1\ m2\ P \equiv \forall x \in dom\ m1.\ x \in dom\ m2 \wedge P\ (the\ (m2\ x))\ (the\ (m1\ x))$

fun
inits :: $'a\ list \Rightarrow 'a\ list\ list$
where
$inits\ []\ = [[]]$
$|\ inits\ (x\ \#\ xs) = []\ \#\ map\ (op\ \#\ x)\ (inits\ xs)$

lemma *submap-st-empty* [*simp*]:
submap-st Map.empty m P
unfolding *submap-st-def* **by** *simp*

lemma *submap-stE* [*elim?*]:
assumes *major*: *submap-st m n P*
and *mx*: $m\ x = Some\ v$
and *rl*: $\bigwedge v'.\ [\![\ n\ x = Some\ v';\ P\ v'\ v]\!] \Longrightarrow R$
shows *R*
using *major mx rl*
unfolding *submap-st-def* **by** *fastforce*

lemma *submap-st-update*:
assumes *st*: $submap\text{-}st\ m\ m'\ P$
and *pvs*: $P\ v'\ v$
shows $submap\text{-}st\ (m(x \mapsto v))\ (m'(x \mapsto v'))\ P$
using *st pvs* **unfolding** *submap-st-def* **by** *auto*

lemma *submap-st-weaken*:
assumes *st*: $submap\text{-}st\ m\ m'\ P$
and *rl*: $\bigwedge x\ v\ v'.\ [\![\ m\ x = Some\ v;\ m'\ x = Some\ v';\ P\ v'\ v\ ]\!] \Longrightarrow P'\ v'\ v$
shows $submap\text{-}st\ m\ m'\ P'$
using *st rl*
unfolding *submap-st-def*
by (*auto dest*: *bspec*)

lemma *submap-st-dom*:
assumes *st*: $submap\text{-}st\ m\ m'\ P$
shows $dom\ m \subseteq dom\ m'$
using *st* **unfolding** *submap-st-def* **by** *auto*

lemma *submap-stI*:
  fixes *as* **and** *k*
  assumes *dom*: *dom* $M' \subseteq$ *dom* M
  and *rl*: $\bigwedge x\ v\ v'.\ [\![\ M'\ x = Some\ v';\ M\ x = Some\ v\ ]\!] \Longrightarrow P\ v\ v'$
  shows *submap-st* $M'\ M\ P$
  using *dom* **unfolding** *submap-st-def*
  by (*auto dest*!: *set-mp elim*!: *rl*)

lemma *map-leD*:
  assumes *mle*: $M \subseteq_m M'$
  and *mx*: $M\ x = Some\ y$
  shows $M'\ x = Some\ y$
  using *mle mx* **unfolding** *map-le-def* **by** (*auto simp*: *dom-def*)

lemma *map-le-map-upd-right*:
  assumes *map-le*: $M \subseteq_m M'$
  and *notin*: $x \notin dom\ M'$
  shows $M \subseteq_m M'(x \mapsto y)$
  using *map-le notin* **unfolding** *map-le-def*
  by (*auto simp*: *dom-def*)

lemma *ran-map-upd-subset*:
  $ran\ (M(x \mapsto v)) \subseteq ran\ M \cup \{v\}$
  unfolding *ran-def* **by** *auto*

lemma *ran-map-upds*:
  $ran\ [as\ [\mapsto]\ bs] \subseteq set\ bs$
  unfolding *ran-def map-upds-def*
  by (*clarsimp dest*!: *map-of-SomeD simp*: *set-zip*)

lemma *option-map-map-upds*:
  $Option.map\ f \circ [as\ [\mapsto]\ bs] = [as\ [\mapsto]\ map\ f\ bs]$
  unfolding *map-upds-def*
  by (*simp add*: *map-of-map* [*symmetric*] *rev-map* [*symmetric*] *zip-map2*)

lemma *map-of-apply*:
  $[\![k \in set\ (map\ fst\ xs);\ P\ (Some\ (hd\ (map\ snd\ (filter\ (\lambda x.\ fst\ x = k)\ xs))))\ ]\!]$
  $\Longrightarrow P\ (map\text{-}of\ xs\ k)$
  by (*induct xs*) (*auto split*: *split-if-asm*)

lemma *hd-filter-conv-nth*:
  assumes *non-empty*: *filter* $P\ xs \neq []$
  shows $R\ (hd\ (filter\ P\ xs)) = (\exists\, i < length\ xs.\ R\ (xs\ !\ i) \wedge P\ (xs\ !\ i) \wedge (\forall j <$
$i.\ \neg\ P\ (xs\ !\ j)))$ (**is** *?LHS xs* = *?RHS xs*)
  using *non-empty*
proof (*induction xs arbitrary*:)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons y ys*)

show *?case*
proof (*cases P y*)
case *True*
thus *?thesis* **by** *fastforce*
next
case *False*
hence *?LHS* (*y* # *ys*) = *?LHS ys* **by** *simp*
also have ... = *?RHS ys*
using *False Cons.prems* **by** (*auto intro*!: *Cons.IH*)
also have ... = *?RHS* (*y* # *ys*)
proof
assume *?RHS ys*
then obtain *i* **where** $i < \mathit{length}\ ys \land R\ (ys\ !\ i) \land P\ (ys\ !\ i) \land (\forall j{<}i.\ \neg\ P$
$(ys\ !\ j))$ **..**
thus *?RHS* (*y* # *ys*) **using** *False*
by − (*rule exI* [**where** *x* = *Suc i*], *clarsimp simp*: *less-Suc-eq-0-disj*)
next
assume *?RHS* (*y* # *ys*)
with *False* **obtain** *i* **where** $i < \mathit{length}\ ys \land R\ (ys\ !\ i) \land P\ (ys\ !\ i) \land (\forall j{<}i.$
$\neg\ P\ (ys\ !\ j))$
by (*fastforce simp*: *nth.simps gr0-conv-Suc split*: *nat.splits*)
thus *?RHS ys* **..**
qed
finally show *?thesis* **.**
qed
qed

lemma *map-of-apply-nth*:
assumes *key-in-list*: $k \in \mathit{set}\ (\mathit{map}\ \mathit{fst}\ xs)$
and *rl*: $\bigwedge i.\ [\![i < \mathit{length}\ xs;\ \mathit{fst}\ (xs\ !\ i) = k;\ (\forall j < i.\ \mathit{fst}\ (xs\ !\ j) \neq k)\ ]\!] \Longrightarrow$
$P\ (\mathit{Some}\ (\mathit{snd}\ (xs\ !\ i)))$
shows *P* (*map-of xs k*)
using *key-in-list*
proof (*rule map-of-apply*)
let $?i = \mathit{LEAST}\ i.\ \mathit{fst}\ (xs\ !\ i) = k$

from *key-in-list* **obtain** *i* **where**
ivs: $i < \mathit{length}\ xs\ \mathit{fst}\ (xs\ !\ i) = k$
by (*clarsimp simp*: *in-set-conv-nth split-def*)

hence $[x{\leftarrow}xs\ .\ \mathit{fst}\ x = k] \neq []$
by (*fastforce simp add*: *filter-empty-conv*)

moreover have $?i < \mathit{length}\ xs$ **using** *ivs*
by (*auto elim*!: *order-le-less-trans* [*rotated*] *intro*!: *Least-le*)
moreover from ‹*fst* (*xs* ! *i*) = *k*› **have** $\mathit{fst}\ (xs\ !\ ?i) = k$ **by** (*rule LeastI*)
moreover have $\forall j < ?i.\ \mathit{fst}\ (xs\ !\ j) \neq k$ **by** (*clarsimp dest*!: *not-less-Least*)

from ‹$[x{\leftarrow}xs\ .\ \mathit{fst}\ x = k] \neq []$› **have** $\mathit{hd}\ [x{\leftarrow}xs\ .\ \mathit{fst}\ x = k] = (xs\ !\ ?i)$

by (*rule iffD2* [*OF hd-filter-conv-nth*])
(*rule exI* [**where** x = *?i*], *intro conjI*, *simp-all*, *fact+*)

moreover have P (*Some* (*snd* (*xs* ! *?i*))) **by** (*rule rl*, *fact+*)
ultimately show P (*Some* (*hd* (*map snd* [x←*xs* . *fst* x = k])))
by (*simp add*: *hd-map*)
qed

lemma *map-of-apply-nth-LEAST* [*consumes 1*, *case-names LEAST*]:
fixes *xs* **and** k
defines *l-i* ≡ *LEAST* i. *fst* (*xs* ! i) = k
assumes *key-in-list*: k ∈ *set* (*map fst xs*)
and *rl*: ⟦*l-i* < *length xs*; *fst* (*xs* ! *l-i*) = k ⟧ ⟹ P (*Some* (*snd* (*xs* ! *l-i*)))
shows P (*map-of xs k*)
using *key-in-list*
proof (*rule map-of-apply*)
from *key-in-list* **obtain** i **where**
ivs: i < *length xs fst* (*xs* ! i) = k
by (*clarsimp simp*: *in-set-conv-nth split-def*)

hence [x←*xs* . *fst* x = k] ≠ []
by (*fastforce simp add*: *filter-empty-conv*)

moreover have *l-i* < *length xs* **using** *ivs* **unfolding** *l-i-def*
by (*auto simp add*: *l-i-def intro*: *order-le-less-trans* [*rotated*] *intro*!: *Least-le*)
moreover have *fst* (*xs* ! i) = k **by** *fact*
hence *fst* (*xs* ! *l-i*) = k
unfolding *l-i-def* **by** (*rule LeastI*)
moreover from ‹*l-i* < *length xs*›
have ∀ j < *l-i*. *fst* (*xs* ! j) ≠ k
unfolding *l-i-def*
by (*clarsimp dest*!: *not-less-Least*)
from ‹[x←*xs* . *fst* x = k] ≠ []› **have** *hd* [x←*xs* . *fst* x = k] = (*xs* ! *l-i*)
by (*rule iffD2* [*OF hd-filter-conv-nth*])
(*rule exI* [**where** x = *l-i*], *intro conjI*, *simp-all*, *fact+*)

moreover have P (*Some* (*snd* (*xs* ! *l-i*))) **by** (*rule rl*) *fact+*

ultimately show P (*Some* (*hd* (*map snd* [x←*xs* . *fst* x = k])))
by (*simp add*: *hd-map*)
qed

lemma *map-of-apply-nth-LEAST′* [*consumes 1*, *case-names LEAST*]:
fixes *as* **and** k
defines *l-i* ≡ *LEAST* i. *as* ! i = k
assumes *key-in-list*: k ∈ *set as*
and *lens*: *length as* = *length bs*
and *rl*: ⟦*l-i* < *length as*; *as* ! *l-i* = k ⟧ ⟹ P (*Some* (*bs* ! *l-i*))
shows P (*map-of* (*zip as bs*) k)

proof (*rule map-of-apply-nth*)
  from *key-in-list lens* **show** $k \in$ *set* (*map fst* (*zip as bs*)) **by** *clarsimp*
next
  fix *i*
  assume *ilt*: $i <$ *length* (*zip as bs*) **and** *fst* (*zip as bs* ! *i*) = *k*
    $\forall j<i.$ *fst* (*zip as bs* ! *j*) $\neq k$
  hence *as-v*: *as* ! *i* = *k* $\forall j<i.$ *as* ! *j* $\neq k$
    by *auto*
  hence *l-i-v*: *l-i* = *i* **unfolding** *l-i-def*
    by (*auto intro*!: *Least-equality simp*: *linorder-not-less* [*symmetric*])
  hence *P* (*Some* (*bs* ! *l-i*)) **using** *ilt lens as-v*
    by (*intro rl*, *simp-all*)
  thus *P* (*Some* (*snd* (*zip as bs* ! *i*))) **using** *lens ilt l-i-v* **by** *simp*
qed

lemma *option-bind-Some-iff*:
  (*Option.bind m f* = *Some v*) = ($\exists v'.$ *m* = *Some* $v' \wedge f\ v'$ = *Some v*)
  by (*cases m*) *simp-all*

lemma *list-all2-weaken* [*consumes 1*, *case-names P*]:
  assumes *lall*: *list-all2 P xs ys*
  and *rl*: $\bigwedge i.$ ⟦ $i <$ *length xs*; *length ys* = *length xs*; *P* (*xs* ! *i*) (*ys* ! *i*) ⟧ $\Longrightarrow$ *Q* (*xs* ! *i*) (*ys* ! *i*)
  shows *list-all2 Q xs ys*
  using *lall* **unfolding** *list-all2-conv-all-nth*
  by (*auto intro*: *rl*)

lemma *submap-st-list-all2I*:
  fixes *as* **and** *k*
  assumes *lens*: *length as* = *length bs length as* = *length cs*
  and *lall*: *list-all2 P cs bs*
  shows *submap-st* [*as* [$\mapsto$] *bs*] [*as* [$\mapsto$] *cs*] *P*
  unfolding *submap-st-def*
proof (*intro ballI conjI*)
  fix *x*
  assume *xin*: $x \in$ *dom* [*as* [$\mapsto$] *bs*]
  thus $x \in$ *dom* [*as* [$\mapsto$] *cs*] **using** *lens* **by** *simp*

  let *?i* = *LEAST i. rev as* ! *i* = *x*
  from *lall* **have** *list-all2 P* (*rev cs*) (*rev bs*) **by** *simp*
  hence *?i* $<$ *length* (*rev cs*) $\Longrightarrow$ *P* (*rev cs* ! *?i*) (*rev bs* ! *?i*)
    by (*rule list-all2-nthD*)
  thus *P* (*the* ([*as* [$\mapsto$] *cs*] *x*)) (*the* ([*as* [$\mapsto$] *bs*] *x*)) **using** *lens xin*
    by (*clarsimp simp*: *map-upds-def dom-map-of-zip zip-rev* [*symmetric*]) (*fastforce intro*: *map-of-apply-nth-LEAST'*)
qed

lemma *list-all2-ballE1*:

assumes *asms*: *list-all2 P as bs a* ∈ *set as*
and *rl*: $\bigwedge b$. ⟦ *b* ∈ *set bs*; *P a b* ⟧ ⟹ *R*
shows *R*
using *asms rl*
by (*fastforce simp*: *list-all2-conv-all-nth in-set-conv-nth*)

theory *Heaps*
imports *Lib*
begin

type-synonym *ridx* = *nat*

type-synonym *roff* = *nat*

definition
lookup-heap :: ((′*a* ⇒ ′*b option*) *list*) ⇒ *ridx* ⇒ ′*a* ⇒ ′*b option*
where
lookup-heap H region = (*if region* < *length H then* (*H* ! *region*) *else Map.empty*)

definition
update-heap :: ((′*a* ⇒ ′*b option*) *list*) ⇒ *ridx* ⇒ ′*a* ⇒ ′*b* ⇒ (′*a* ⇒ ′*b option*) *list option*
where
update-heap H region = (*if region* < *length H then* (λ*off v*. *Some* (*take region H* @ [(*H* ! *region*)(*off* ↦ *v*)] @ *drop* (*region* + *1*) *H*)) *else* (λ- -. *None*))

definition
push-heap :: ((′*a* ⇒ ′*b option*) *list*) ⇒ ((′*a* ⇒ ′*b option*) *list*)
where
push-heap H = *H* @ [*Map.empty*]

definition
pop-heap :: ((′*a* ⇒ ′*b option*) *list*) ⇒ ((′*a* ⇒ ′*b option*) *list*)
where
pop-heap H = *butlast H*

definition
fresh-in-heap :: ((′*a* :: *infinite* ⇒ ′*b option*) *list*) ⇒ *ridx* ⇒ ′*a*
where
fresh-in-heap H n = (*fresh* (*dom* (*H* ! *n*)))

definition
subheap :: (′*a* ⇒ ′*b option*) *list* ⇒ (′*a* ⇒ ′*b option*) *list* ⇒ *bool*

where
  subheap = *list-all2 map-le*

lemma *pop-push-heap*: *pop-heap* (*push-heap H*) = *H*
  by (*simp add*: *pop-heap-def push-heap-def*)

lemma *lookup-heap-Some-iff*:
  (*lookup-heap H region off* = *Some v*) = (*region* < *length H* ∧ (*H* ! *region*) *off* = *Some v*)
  unfolding *lookup-heap-def* **by** *simp*

lemma *lookup-heap-empty* [*simp*]:
  lookup-heap [] *n* = (λ-. *None*) **unfolding** *lookup-heap-def* **by** *simp*

lemma *lookup-heap-length-append* [*simp*]:
  lookup-heap (*H* @ [*R*]) (*length H*) = *R*
  unfolding *lookup-heap-def* **by** *simp*

lemma *update-heap-empty* [*simp*]:
  update-heap [] *n* = (λ- -. *None*) **unfolding** *update-heap-def* **by** *simp*

lemma *update-heap-idem*:
  shows (*update-heap H region off v* = *Some H*) = (*lookup-heap H region off* = *Some v*)
  apply (*simp add*: *lookup-heap-Some-iff*)
  apply (*clarsimp simp add*: *update-heap-def list-eq-iff-nth-eq*)
  apply *rule*
   apply (*clarsimp simp add*: *min-absorb1 min-absorb2 Suc-pred*)
   apply (*drule spec*, *drule* (*1*) *mp*)
   apply (*clarsimp simp*: *nth-append fun-upd-idem-iff*)
  apply (*clarsimp simp add*: *min-absorb1 min-absorb2*)
  apply *rule*
   apply (*rule Suc-pred*)
   apply *arith*
 apply (*auto simp*: *le-imp-diff-is-add nth-append nth.simps min-absorb1 min-absorb2 not-less add-ac split*: *nat.splits*)
  done

lemma *update-heap-into-lookup-heap*:
  assumes *upd*: *update-heap H region off v* = *Some H'*
  shows *lookup-heap H' region' off'* = (*if region'* = *region* ∧ *off'* = *off then Some v else lookup-heap H region' off'*)
  using *upd* **unfolding** *update-heap-def*
   by (*auto simp*: *lookup-heap-def min-absorb2 min-absorb1 not-less nth-append nth.simps*
      Suc-pred le-imp-diff-is-add add-ac
      split: *split-if-asm nat.splits*)

lemma *lookup-heap-into-update-heap-same*:

assumes *lup*: *lookup-heap H region off* = *Some v*
obtains H' **where** *update-heap H region off* v' = *Some* H'
using *lup* **unfolding** *update-heap-def*
by (*auto simp*: *lookup-heap-def min-absorb2 min-absorb1 not-less nth-append nth.simps*
Suc-pred le-imp-diff-is-add add-ac
split: *split-if-asm nat.splits*)

lemma *update-heap-Some-iff*:
(*update-heap H region off v* = *Some* H')
= (*region* < *length H* $\wedge$ H' = *take region H* @ [(*H* ! *region*)(*off* $\mapsto$ *v*)] @ *drop* (*region* + *1*) *H*)
unfolding *update-heap-def* **by** *auto*

lemma *update-heap-mono*:
assumes *upd*: *update-heap H region off v* = *Some* H'
shows *update-heap* (*H* @ *G*) *region off v* = *Some* (H' @ *G*)
using *upd* **unfolding** *update-heap-def*
by (*auto simp*: *nth-append split*: *split-if-asm*)

lemma *update-heap-shrink*:
assumes *upd*: *update-heap* (*H* @ *G*) *region off v* = *Some* (H' @ *G*)
and *region*: *region* < *length H*
shows *update-heap H region off v* = *Some* H'
using *upd region* **unfolding** *update-heap-def*
by (*clarsimp simp*: *not-less nth-append split*: *split-if-asm*)

lemma *update-heap-length*:
update-heap H n x v = *Some* H' $\Longrightarrow$ *length* H' = *length H*
by (*auto simp*: *update-heap-def split*: *split-if-asm*)

lemma *length-pop-heap* [*simp*]:
$H \neq []$ $\Longrightarrow$ *length* (*pop-heap H*) = *length H* − *1*
unfolding *pop-heap-def* **by** *simp*

lemma *length-push-heap* [*simp*]:
length (*push-heap H*) = *Suc* (*length H*)
unfolding *push-heap-def* **by** *simp*

lemma *length-pop-heap-le*:
length (*pop-heap H*) $\leq$ *length H*
unfolding *pop-heap-def* **by** *simp*

lemma *fresh-in-heap-fresh*:
assumes *finite*: *finite* (*dom* (*lookup-heap H n*))
shows *fresh-in-heap H n* $\notin$ *dom* (*lookup-heap H n*)
proof
let *?R* = *lookup-heap H n*
assume *fresh-in-heap H n* $\in$ *dom ?R*

hence *fresh* (*dom ?R*) $\in$ *dom ?R*
by (*auto split*: *split-if-asm simp*: *fresh-in-heap-def lookup-heap-def*)
moreover from *finite* **have** *fresh* (*dom ?R*) $\notin$ *dom ?R* **by** (*rule fresh-not-in*)
ultimately show *False* **by** *simp*
qed

lemma *subheap-singleton*:
subheap [*R*] *H* = ($\exists$ *R'*. *H* = [*R'*] $\wedge$ *R* $\subseteq_m$ *R'*)
unfolding *subheap-def*
by (*cases H*) *auto*

lemma *subheap-lengthD*:
subheap H H' $\Longrightarrow$ *length H* = *length H'* **unfolding** *subheap-def* **by** (*rule list-all2-lengthD*)

lemma *subheap-lookup-heapD*:
assumes *sh*: *subheap H H'*
and *lup*: *lookup-heap H region off* = *Some v*
shows *lookup-heap H' region off* = *Some v*
using *sh lup* **unfolding** *subheap-def*
by (*auto simp*: *lookup-heap-Some-iff list-all2-conv-all-nth intro*: *map-leD*)

lemma *subheap-mono-left*:
assumes *sh*: *subheap H H'*
shows *subheap* (*G* @ *H*) (*G* @ *H'*)
using *sh* **unfolding** *subheap-def*
by (*auto simp*: *list-all2-conv-all-nth nth-append*)

lemma *subheap-mono-right*:
assumes *sh*: *subheap H H'*
shows *subheap* (*H* @ *G*) (*H'* @ *G*)
using *sh* **unfolding** *subheap-def*
by (*auto simp*: *list-all2-conv-all-nth nth-append*)

lemma *subheap-refl*:
shows *subheap H H*
unfolding *subheap-def*
by (*auto simp*: *list-all2-conv-all-nth*)

lemma *subheap-trans* [*trans*]:
assumes *sh*: *subheap H H'*
and *sh'*: *subheap H' H''*
shows *subheap H H''*
using *sh sh'* **unfolding** *subheap-def*
by (*auto simp*: *list-all2-conv-all-nth nth-append elim*!: *map-le-trans dest*!: *spec*)

lemma *subheap-take-drop*:
assumes *xd*: *x* $\notin$ *dom* (*lookup-heap H n*)
and *mn*: *m* = *Suc n*

and *nv*: $n < length\ H$
shows *subheap H* (*take n H* @ [(*H* ! *n*)($x \mapsto y$)] @ *drop m H*) (**is** *subheap H* *?H'*)
using *mn nv xd* **unfolding** *subheap-def*
by (*auto intro*!: *nth-equalityI simp*: *lookup-heap-Some-iff list-all2-conv-all-nth le-imp-diff-is-add nth-append nth.simps min-absorb1 min-absorb2 not-less add-ac map-le-def split*: *nat.splits*)

end

3 Syntax

3.1 Types

datatype *prim* = *BoolT* | *NatT* | *UnitT*

datatype *area* = *Stored prim*

datatype *'r wtype* = *Prim prim* | *RefT 'r area*

abbreviation
NAT :: *'r wtype*
where
NAT ≡ *Prim NatT*

abbreviation
BOOL :: *'r wtype*
where
BOOL ≡ *Prim BoolT*

abbreviation
UNIT :: *'r wtype*
where
UNIT ≡ *Prim UnitT*

datatype *'r funtype* = *FunT 'r wtype 'r wtype list*

3.2 Expressions and statements

datatype *binop* = *add* | *sub* | *mult*
datatype *cmpop* = *lt* | *eq*

datatype *'var expr* =
  Var 'var
  | *Nat nat*
  | *Bool bool*
  | *Unit*
  | *BinCmp cmpop 'var expr 'var expr*
  | *BinOp binop 'var expr 'var expr*

datatype *'var impureexp* =
  Pure 'var expr
  | *NewRef 'var expr*
  | *ReadRef 'var expr*
  | *WriteRef 'var expr 'var expr*

datatype (*'var*, *'fun*) *stmt* =
  Skip
  | *Return 'var expr*
  | *Bind 'var 'var impureexp* (*'var*, *'fun*) *stmt*
  | *If 'var expr* (*'var*, *'fun*) *stmt* (*'var*, *'fun*) *stmt*
  | *For 'var 'var expr 'var expr 'var expr* (*'var*, *'fun*) *stmt*
  | *Seq* (*'var*, *'fun*) *stmt* (*'var*, *'fun*) *stmt* (**infixr** ;; *90*)
  | *Call 'var 'fun 'var expr list* (*'var*, *'fun*) *stmt*

fun
  is-terminal :: (*'var*, *'fun*) *stmt* ⇒ *bool*
where
  is-terminal (*Return e*) = *True*
| *is-terminal* - = *False*

lemmas *is-terminalE* [*consumes 1*, *case-names Return Skip*]
  = *is-terminal.elims*(*2*)

3.3 Values

datatype *prim-value* = *NatV nat* | *BoolV bool* | *UnitV*
datatype *wvalue* = *PrimV prim-value* | *RefV ridx roff*
datatype *hvalue* = *StoredV prim-value*

datatype (*'var*, *'fun*) *func* = *Func 'var list* (*'var*, *'fun*) *stmt*

type-synonym (*'var*, *'fun*) *funs* = *'fun* ⇒ (*'var*, *'fun*) *func option*

type-synonym *region* = *roff* ⇒ *hvalue option*

type-synonym *heap* = *region list*

type-synonym *'var store* = *'var* ⇒ *wvalue option*

datatype *'var frame-class* = *ReturnFrame 'var* | *SeqFrame*

fun
  isReturnFrame :: *'var frame-class* ⇒ *bool*
where
  isReturnFrame (*ReturnFrame* -) = *True*
| *isReturnFrame SeqFrame* = *False*

type-synonym (*'var*, *'fun*) *stack-frame* = *'var store* × (*'var*, *'fun*) *stmt* × *'var frame-class*

type-synonym (*'var*, *'fun*) *stack* = (*'var*, *'fun*) *stack-frame list*

record (*'var*, *'fun*) *state* =
  store :: *'var store*
  heap :: *heap*
  stack :: (*'var*, *'fun*) *stack*

4 Semantics

4.1 Expressions

fun
  cmpopV :: *cmpop* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
  cmpopV lt e_1 e_2 = ($e_1 < e_2$)
| *cmpopV eq* e_1 e_2 = ($e_1 = e_2$)

fun
  binopV :: *binop* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
  binopV add e_1 e_2 = ($e_1 + e_2$)
| *binopV sub* e_1 e_2 = ($e_1 - e_2$)
| *binopV mult* e_1 e_2 = ($e_1 * e_2$)

fun

$ExpV :: 'var\ store \Rightarrow 'var\ expr \Rightarrow wvalue\ option$
where
$ExpVar$: $ExpV\ G\ (Var\ x) = G\ x$
| $ExpNat$: $ExpV\ G\ (Nat\ n) = Some\ (PrimV\ (NatV\ n))$
| $ExpBool$: $ExpV\ G\ (Bool\ b) = Some\ (PrimV\ (BoolV\ b))$
| $ExpUnit$: $ExpV\ G\ Unit = Some\ (PrimV\ UnitV)$
| $ExpBinCmp$: $ExpV\ G\ (BinCmp\ bop\ e_1\ e_2) = (case\ (ExpV\ G\ e_1,\ ExpV\ G\ e_2)\ of$
$(Some\ (PrimV\ (NatV\ v_1)),\ Some\ (PrimV\ (NatV$
$v_2))) \Rightarrow Some\ (PrimV\ (BoolV\ (cmpopV\ bop\ v_1\ v_2)))$
$|\ \text{-} \Rightarrow None)$
| $ExpBinOp$: $ExpV\ G\ (BinOp\ bop\ e_1\ e_2) = (case\ (ExpV\ G\ e_1,\ ExpV\ G\ e_2)\ of$
$(Some\ (PrimV\ (NatV\ v_1)),\ Some\ (PrimV\ (NatV$
$v_2))) \Rightarrow Some\ (PrimV\ (NatV\ (binopV\ bop\ v_1\ v_2)))$
$|\ \text{-} \Rightarrow None)$

abbreviation
$expv\text{-}some :: 'var\ store \Rightarrow 'var\ expr \Rightarrow wvalue \Rightarrow bool\ (\text{-} \models \text{-} \downarrow \text{-})$
where
$G \models e \downarrow v \equiv ExpV\ G\ e = Some\ v$

fun
$wvalue\text{-}to\text{-}hvalue :: wvalue \Rightarrow hvalue\ option$
where
$wvalue\text{-}to\text{-}hvalue\ (PrimV\ v) = Some\ (StoredV\ v)$
| $wvalue\text{-}to\text{-}hvalue\ \text{-} \qquad = None$

fun
$hvalue\text{-}to\text{-}wvalue :: hvalue \Rightarrow wvalue\ option$
where
$hvalue\text{-}to\text{-}wvalue\ (StoredV\ v) = Some\ (PrimV\ v)$

fun
$ImpureExpV :: 'var\ store \Rightarrow heap \Rightarrow 'var\ impureexp \Rightarrow (heap \times wvalue)\ option$
where
$ExpPure$: $ImpureExpV\ G\ H\ (Pure\ e) = Option.map\ (\lambda v.\ (H,\ v))\ (ExpV\ G\ e)$

| $ExpNewRef$: $ImpureExpV\ G\ H\ (NewRef\ e) = (let\ region = length\ H - 1\ in$
$let\ off = fresh\text{-}in\text{-}heap\ H\ region\ in$
$Option.bind\ (ExpV\ G\ e)$
$(\lambda wv.\ Option.bind\ (wvalue\text{-}to\text{-}hvalue\ wv)$
$(\lambda v.\ Option.bind\ (update\text{-}heap\ H\ region\ off\ v)$
$(\lambda H'.\ Some\ (H',\ RefV\ region\ off)))))$
| $ExpReadRef$: $ImpureExpV\ G\ H\ (ReadRef\ e) = (case\ ExpV\ G\ e\ of\ Some\ (RefV$
$region\ off) \Rightarrow Option.bind\ (lookup\text{-}heap\ H\ region\ off)$
$(\lambda hv.$
$Option.bind\ (hvalue\text{-}to\text{-}wvalue\ hv)$
$(\lambda v.$
$Some\ (H,v)))$

| - ⇒ *None*)
| *ExpWriteRef*: *ImpureExpV G H* (*WriteRef* e_1 e_2) = (*case* (*ExpV G* e_1, *ExpV G* e_2) *of*
(*Some* (*RefV region off*), *Some wv*) ⇒ *Option.bind* (*wvalue-to-hvalue wv*)
(λ*v*. *Option.bind* (*update-heap H region off v*)
(λ*H'*. *Some* (*H'*, *PrimV UnitV*)))
| - ⇒ *None*)

abbreviation
impure-expv-some :: *'var store* ⇒ *heap* ⇒ *'var impureexp* ⇒ *heap* ⇒ *wvalue* ⇒ *bool* (- ⊨ -, - ⇓ -, - [*49*, *49*, *49*, *49*, *49*] *50*)
where
G ⊨ *H*, *e* ⇓ *H'*, *v* ≡ *ImpureExpV G H e* = *Some* (*H'*, *v*)

4.2 Statement evaluation

datatype (*'var*, *'fun*) *StepResult* = *Normal* (*'var*, *'fun*) *state* × (*'var*, *'fun*) *stmt*
| *Finished wvalue*

inductive
Step :: (*'var*, *'fun*) *funs* ⇒ (*'var*, *'fun*) *state* × (*'var*, *'fun*) *stmt* ⇒ (*'var*, *'fun*) *StepResult* ⇒ *bool* (- ⊨ - ▷ - [*49*, *49*, *49*] *50*)
where
StepBind: ⟦ *store S* ⊨ *heap S*, *e* ⇓ *H'*, *v* ⟧ ⟹ *F* ⊨ (*S*, *Bind x e s*) ▷ *Normal* (*S*⦇ *store* := (*store S*)(*x* ↦ *v*), *heap* := *H'* ⦈, *s*)
| *StepIf*: ⟦ *store S* ⊨ *e* ↓ *PrimV* (*BoolV b*) ⟧ ⟹ *F* ⊨ (*S*, *If e* s_1 s_2) ▷ *Normal* (*S*, *if b then* s_1 *else* s_2)

| *StepFor*: ⟦ *store S* ⊨ e_I ↓ *v* ⟧ ⟹ *F* ⊨ (*S*, *For x* e_I e_B e_S *s*) ▷ *Normal* (*S*⦇ *store* := (*store S*)(*x* ↦ *v*) ⦈, *If* e_B (*s* ;; *For x* e_S e_B e_S *s*) *Skip*)
| *StepSeq*: *F* ⊨ (*S*, s_1 ;; s_2) ▷ *Normal* (*S* ⦇ *stack* := (*store S*, s_2, *SeqFrame*) # *stack S* ⦈, s_1)
| *StepSkip*: *stack S* = (*st'*, *cont*, *SeqFrame*) # *stack'* ⟹ *F* ⊨ (*S*, *Skip*) ▷ *Normal* (*S* ⦇ *store* := *st'*, *stack* := *stack'* ⦈, *cont*)
| *StepReturnSeq*: *stack S* = (*st'*, *cont*, *SeqFrame*) # *stack'* ⟹ *F* ⊨ (*S*, *Return e*) ▷ *Normal* (*S* ⦇ *stack* := *stack'* ⦈, *Return e*)
| *StepReturnFun*: ⟦ *stack S* = (*store'*, *cont*, *ReturnFrame x*) # *stack'*; *store S* ⊨ *e* ↓ *v* ⟧
⟹ *F* ⊨ (*S*, *Return e*) ▷ *Normal* (⦇ *store* = *store'*(*x* ↦ *v*), *heap* = *pop-heap* (*heap S*), *stack* = *stack'* ⦈, *cont*)
| *StepCall*: ⟦ *F f* = *Some* (*Func as body*); *length as* = *length es*; *pre-vs* = *map* (*ExpV* (*store S*)) *es*; ∀ *v* ∈ *set pre-vs*. *v* ≠ *None* ⟧
⟹ *F* ⊨ (*S*, *Call x f es s*) ▷ *Normal* (⦇ *store* = [*as* [↦] *map the pre-vs*], *heap* = *push-heap* (*heap S*), *stack* = (*store S*, *s*, *ReturnFrame x*) # (*stack*

S) ⦈, *body*)
| *StepReturnFin*: $\llbracket$ *stack* S = []; *store* $S \models e \downarrow v$ $\rrbracket \Longrightarrow F \models (S,$ *Return* $e) \rhd$ *Finished* v

inductive-cases *StepSkipE*: $F \models (S,$ *Skip*$) \rhd$ *Normal* (S', s')
inductive-cases *StepReturnE* [*consumes 1*, *case-names SeqFrame ReturnFrame Finish*]: $F \models (S,$ *Return* $e) \rhd R$

inductive-cases *StepCallE*: $F \models (S,$ *Call x args body s*$) \rhd R$
inductive-cases *StepSeqE*: $F \models (S, s_1 ;; s_2) \rhd R$

inductive
StepN :: ('*var*, '*fun*) *funs* ⇒ *nat* ⇒ ('*var*, '*fun*) *state* × ('*var*, '*fun*) *stmt* ⇒ ('*var*, '*fun*) *StepResult* ⇒ *bool* (-, - ⊨ - $\rhd^*$ - [*49*, *49*, *49*, *49*] *50*)
where
Step1: F, *0* $\models S \rhd^*$ *Normal* S
| *StepN*: $\llbracket$ $F, n \models S \rhd^*$ *Normal* S'; $F \models S' \rhd S''$ $\rrbracket \Longrightarrow F$, *Suc* $n \models S \rhd^* S''$

lemma *StepN-add-head*:
assumes *s1*: $F \models S \rhd$ *Normal* S'
and *sn*: $F, n \models S' \rhd^* S''$
shows F, *Suc* $n \models S \rhd^* S''$
using *sn s1*
by *induction* (*auto intro*: *StepN.intros*)

5 Well formed programs

5.1 Preliminaries

type-synonym *regionT* = *roff* ⇒ *area option*

type-synonym *heapT* = *regionT list*

type-synonym ('*fun*, '*r*) *funsT* = '*fun* ⇒ '*r funtype option*

type-synonym ('*var*, '*r*) *storeT* = '*var* ⇒ '*r wtype option*

type-synonym '*r region-env* = '*r* ⇒ *ridx option*

5.2 Region type variable substitutions

type-synonym $'r\ tsubst = 'r \Rightarrow 'r$

fun
$tsubst :: 'r\ tsubst \Rightarrow 'r\ wtype \Rightarrow 'r\ wtype$
where
$tsubst\ \vartheta\ (RefT\ \varrho\ \tau) = RefT\ (\vartheta\ \varrho)\ \tau$
$|\ tsubst\ \vartheta\ \tau \qquad = \tau$

fun
$tfrees :: 'r\ wtype \Rightarrow 'r\ set$
where
$tfrees\ (RefT\ \varrho\ \tau) = \{\varrho\}$
$|\ tfrees\ \tau \qquad = \{\}$

definition
$tfrees\text{-}set :: 'r\ wtype\ set \Rightarrow 'r\ set$
where
$tfrees\text{-}set\ ts = \bigcup (tfrees\ `\ ts)$

5.3 Expressions

inductive
$WfE :: ('var, 'r)\ storeT \Rightarrow 'var\ expr \Rightarrow 'r\ wtype \Rightarrow bool\ (\text{-} \vdash \text{-} : \text{-}\ [49, 49, 49]\ 50)$
where
$wfVar: \quad \Gamma\ v = Some\ \tau \Longrightarrow \Gamma \vdash Var\ v : \tau$
$|\ wfNat: \quad \Gamma \vdash Nat\ n : NAT$
$|\ wfBool: \quad \Gamma \vdash Bool\ b : BOOL$
$|\ wfUnit: \quad \Gamma \vdash Unit : UNIT$
$|\ wfBinCmp: \quad [\![\ \Gamma \vdash e_1 : NAT; \Gamma \vdash e_2 : NAT\ ]\!] \Longrightarrow \Gamma \vdash BinCmp\ bop\ e_1\ e_2 : BOOL$
$|\ wfBinOp: \quad [\![\ \Gamma \vdash e_1 : NAT; \Gamma \vdash e_2 : NAT\ ]\!] \Longrightarrow \Gamma \vdash BinOp\ bop\ e_1\ e_2 : NAT$

5.4 Impure expressions

inductive
$WfImpureExpr :: ('var, 'r)\ storeT \Rightarrow 'r \Rightarrow 'var\ impureexp \Rightarrow 'r\ wtype \Rightarrow bool\ (\text{-}, \text{-} \vdash I\ \text{-} : \text{-}\ [49, 49, 49]\ 50)$
where
$wfPure: \quad [\![\ \Gamma \vdash e : \tau\ ]\!] \Longrightarrow \Gamma, \varrho \vdash I\ Pure\ e : \tau$
$|\ wfNewRef: \quad [\![\ \Gamma \vdash e : Prim\ \tau\ ]\!] \Longrightarrow \Gamma, \varrho \vdash I\ NewRef\ e : RefT\ \varrho\ (Stored\ \tau)$
$|\ wfReadRef: \quad [\![\ \Gamma \vdash e : RefT\ \gamma\ (Stored\ \tau)\ ]\!] \Longrightarrow \Gamma, \varrho \vdash I\ ReadRef\ e : Prim\ \tau$

$|\ wfWriteRef: [\![\ \Gamma \vdash e_2 : Prim\ \tau; \ \Gamma \vdash e_1 : RefT\ \gamma\ (Stored\ \tau)\ ]\!] \Longrightarrow \Gamma, \varrho \vdash I\ WriteRef\ e_1\ e_2 : UNIT$

5.5 Statements

inductive
$WfStmt$ :: $('var,\ 'r)\ storeT \Rightarrow ('fun,\ 'r)\ funsT \Rightarrow 'r \Rightarrow ('var,\ 'fun)\ stmt \Rightarrow 'r\ wtype \Rightarrow bool \Rightarrow bool$ (-, -, - ⊢ - : -, - [49, 49, 49, 49] 50)
where
$wfSkip$: $\Gamma,\ \Psi,\ \varrho \vdash Skip : \tau,\ False$
| $wfReturn$: $\Gamma \vdash e : \tau \Longrightarrow \Gamma,\ \Psi,\ \varrho \vdash Return\ e : \tau,\ b$
| $wfBind$: $[\![\ \Gamma,\ \varrho \vdash I\ e : \tau';\ \Gamma(v \mapsto \tau'),\ \Psi,\ \varrho \vdash s : \tau,\ b\ ]\!] \Longrightarrow \Gamma,\ \Psi,\ \varrho \vdash Bind\ v\ e\ s : \tau,\ b$
| $wfIf$: $[\![\ \Gamma \vdash e : BOOL;\ \Gamma,\ \Psi,\ \varrho \vdash s_1 : \tau,\ b\ ;\ \Gamma,\ \Psi,\ \varrho \vdash s_2 : \tau,\ b\ ]\!] \Longrightarrow \Gamma,\ \Psi,\ \varrho \vdash If\ e\ s_1\ s_2 : \tau,\ b$
| $wfWhile$: $[\![\ \Gamma \vdash e_I : \tau';\ \Gamma(v \mapsto \tau') \vdash e_B : BOOL;\ \Gamma(v \mapsto \tau') \vdash e_S : \tau';\ \Gamma(v \mapsto \tau'),\ \Psi,\ \varrho \vdash s : \tau,\ b\ ]\!] \Longrightarrow \Gamma,\ \Psi,\ \varrho \vdash For\ v\ e_I\ e_B\ e_S\ s : \tau,\ False$
| $wfSeq$: $[\![\ \Gamma,\ \Psi,\ \varrho \vdash s_1 : \tau,\ False;\ \Gamma,\ \Psi,\ \varrho \vdash s_2 : \tau,\ b\ ]\!] \Longrightarrow \Gamma,\ \Psi,\ \varrho \vdash s_1\ ;;\ s_2 : \tau,\ b$
| $wfCall$: $[\![\ \Psi\ f = Some\ (FunT\ \sigma\ ts);$
$list\text{-}all2\ (\lambda e\ \tau.\ \Gamma \vdash e : tsubst\ \vartheta\ \tau)\ es\ ts;$
$\Gamma(x \mapsto tsubst\ \vartheta\ \sigma),\ \Psi,\ \varrho \vdash s : \tau,\ b\ ]\!] \Longrightarrow \Gamma,\ \Psi,\ \varrho \vdash (Call\ x\ f\ es\ s) : \tau,\ b$

inductive-cases $wfReturnE$: $\Gamma,\ \Psi,\ \varrho \vdash Return\ e : \tau,\ b$

5.6 Functions

inductive
$WfFunc$ :: $('fun,\ 'r)\ funsT \Rightarrow ('var,\ 'fun)\ func \Rightarrow 'r\ funtype \Rightarrow bool$
where
$wfFunc$: $[\![\ length\ args = length\ ts;$
$\varrho \notin tfrees\text{-}set\ (set\ ts);\ [args\ [\mapsto]\ ts],\ \Psi,\ \varrho \vdash s : \tau,\ True;$
$tfrees\ \tau \subseteq tfrees\text{-}set\ (set\ ts)\ ]\!]$ (* *no polymorphic return* *)
$\Longrightarrow WfFunc\ \Psi\ (Func\ args\ s)\ (FunT\ \tau\ ts)$

6 Well-formed values and programs

inductive
$WfPrimValue$ :: $prim\text{-}value \Rightarrow prim \Rightarrow bool$
where
$wfNatV$: $WfPrimValue\ (NatV\ n)\ NatT$
| $wfBoolV$: $WfPrimValue\ (BoolV\ b)\ BoolT$
| $wfUnitV$: $WfPrimValue\ UnitV\ UnitT$

inductive-cases $WfPNatVE$: $WfPrimValue\ v\ NatT$
inductive-cases $WfPBoolVE$: $WfPrimValue\ v\ BoolT$

inductive
$WfWValue$ :: $'r\ region\text{-}env \Rightarrow heapT \Rightarrow wvalue \Rightarrow 'r\ wtype \Rightarrow bool$
where
$wfPrimV$: $WfPrimValue\ v\ \tau \Longrightarrow WfWValue\ \Delta\ \Theta\ (PrimV\ v)\ (Prim\ \tau)$
| $wfRefV$: $[\![\ \Delta\ \varrho = Some\ region;\ lookup\text{-}heap\ \Theta\ region\ off = Some\ \tau\ ]\!] \Longrightarrow$

WfWValue Δ Θ (*RefV region off*) (*RefT* ϱ τ)

inductive-cases *WfNatVE*: *WfWValue* Δ Θ *v NAT*
inductive-cases *WfBoolVE*: *WfWValue* Δ Θ *v BOOL*
inductive-cases *WfRefVE*: *WfWValue* Δ Θ *v* (*RefT* ϱ τ)

inductive
WfHValue :: *hvalue* $\Rightarrow$ *area* $\Rightarrow$ *bool*
where
wfStoredV: *WfPrimValue v* τ $\Longrightarrow$ *WfHValue* (*StoredV v*) (*Stored* τ)

inductive
WfHeap :: *heap* $\Rightarrow$ *heapT* $\Rightarrow$ *bool*
where
wfHeapNil: *WfHeap* [] []
| *wfHeapCons*:$\llbracket$ *WfHeap H* Θ; *submap-st* Σ *R WfHValue*; *finite* (*dom R*) $\rrbracket$ $\Longrightarrow$ *WfHeap* (*H* @ [*R*]) (Θ @ [Σ])

inductive
WfFuns :: (*'var*, *'fun*) *funs* $\Rightarrow$ (*'fun*, *'r*) *funsT* $\Rightarrow$ *bool*
where
WfFuns: *submap-st* Ψ *F* (*WfFunc* Ψ) $\Longrightarrow$ *WfFuns F* Ψ

inductive
WfStore :: *'r region-env* $\Rightarrow$ *heapT* $\Rightarrow$ *'var store* $\Rightarrow$ (*'var*, *'r*) *storeT* $\Rightarrow$ *bool*
where
WfStore: *submap-st* Γ *G* (*WfWValue* Δ Θ) $\Longrightarrow$ *WfStore* Δ Θ *G* Γ

inductive
WfFrees :: *'r region-env* $\Rightarrow$ (*'var*, *'r*) *storeT* $\Rightarrow$ *'r* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
WfFrees: $\llbracket$ Δ ϱ = *Some n*; $\forall k \in$ *ran* Δ. $k \leq n$; *tfrees-set* (*ran* Γ) $\subseteq$ *dom* Δ; *finite* (*dom* Δ)$\rrbracket$ $\Longrightarrow$ *WfFrees* Δ Γ ϱ *n*

inductive-cases *WfFreesE* [*elim?*]: *WfFrees* Δ Γ ϱ *n*

inductive
WfStack :: (*'fun*, *'r*) *funsT* $\Rightarrow$ *'r region-env* $\Rightarrow$ *heapT* $\Rightarrow$ (*'var*, *'fun*) *stack* $\Rightarrow$ *'r wtype* $\Rightarrow$ *bool* $\Rightarrow$ *'r* $\Rightarrow$ *bool*
where
wfStackNil: *WfStack* Ψ Δ [Σ] [] *NAT True* ϱ
| *wfStackFun*: $\llbracket$ *WfStack* Ψ Δ' Θ *st* τ' b' γ; *WfStore* Δ' Θ *store'* Γ; $\Gamma(x \mapsto \tau)$, Ψ, $\gamma \vdash$ *cont* : τ', b';
WfFrees Δ' ($\Gamma(x \mapsto \tau)$) γ (*length* Θ $-$ *1*); $\Delta' \subseteq_m \Delta$ $\rrbracket$
$\Longrightarrow$ *WfStack* Ψ Δ (Θ @ [Σ]) ((*store'*, *cont*, *ReturnFrame x*) # *st*) τ *True* ϱ
| *wfStackSeq*: $\llbracket$ *WfStack* Ψ Δ Θ *st* τ b' ϱ; *WfStore* Δ Θ *store'* Γ; Γ, Ψ, $\varrho \vdash$ *cont* : τ, b'; *tfrees-set* (*ran* Γ) $\subseteq$ *dom* Δ $\rrbracket$

$\Longrightarrow$ *WfStack* Ψ Δ Θ $((store', cont, SeqFrame) \# st)$ τ b ϱ

inductive-cases *wfStackFalseE*: *WfStack* Ψ Δ Θ *st* τ *False* ϱ

inductive-cases *WfStackConsE*: *WfStack* Ψ Δ Θ $(s\#st)$ τ b ϱ
inductive-cases *WfStackSeqE*: *WfStack* Ψ Δ Θ $((st, s, SeqFrame) \# st')$ τ b ϱ

declare *One-nat-def Un-insert-right* [*simp del*]
inductive-cases *WfStackFunE*: *WfStack* Ψ Δ Θ $((st, s, ReturnFrame\ x) \# st')$ τ b ϱ
declare *One-nat-def Un-insert-right* [*simp*]

inductive
WfState :: $('var, 'fun)$ *state* $\Rightarrow$ $('var, 'r)$ *storeT* $\Rightarrow$ $('fun, 'r)$ *funsT* $\Rightarrow$ $'r$ *wtype* $\Rightarrow$ *bool* $\Rightarrow$ $'r$ $\Rightarrow$ *bool*
where
WfState: $[\![$ *WfStore* Δ Θ (*store* S) Γ; *WfHeap* (*heap* S) Θ; *WfStack* Ψ Δ Θ (*stack* S) τ b ϱ; *WfFrees* Δ Γ ϱ (*length* $\Theta - 1$) $]\!]$
$\Longrightarrow$ *WfState* S Γ Ψ τ b ϱ

declare *One-nat-def* [*simp del*]
inductive-cases *WfStateE* [*elim*]: *WfState* S Γ Ψ τ b ϱ
declare *One-nat-def* [*simp*]

inductive
WfProgram :: $('fun, 'r)$ *funsT* $\Rightarrow$ $('var, 'fun)$ *state* $\Rightarrow$ $('var, 'fun)$ *stmt* $\Rightarrow$ *bool*
where
wfProgramI: $[\![$ *WfState* S Γ Ψ τ b ϱ; $\Gamma, \Psi, \varrho \vdash s : \tau, b$ $]\!]$ $\Longrightarrow$ *WfProgram* Ψ S s

7 Type system properties

7.1 General properties

lemma *WfHeap-length*:
assumes *wfh*: *WfHeap* H Θ
shows *length* H = *length* Θ
using *wfh*
by *induct auto*

lemma *WfHeap-dom*:
assumes *wfh*: *WfHeap* H Θ
and *nv*: $n <$ *length* H
shows *dom* $(\Theta\ !\ n) \subseteq$ *dom* $(H\ !\ n)$
using *wfh nv WfHeap-length* [*OF wfh*]
by *induct* (*auto simp*: *nth-append not-less dest*!: *submap-st-dom*)

lemma *WfHeap-dom'*:
  assumes *wfh*: *WfHeap H* Θ
  shows *dom* (*lookup-heap* Θ *n*) $\subseteq$ *dom* (*lookup-heap H n*)
  using *wfh WfHeap-length* [*OF wfh*]
  by *induct* (*auto simp*: *nth-append not-less lookup-heap-def dest*!: *submap-st-dom split*: *split-if-asm*)

lemma *WfStack-heap-length*:
  assumes *wfst*: *WfStack* Ψ Δ Θ *st* τ *b* ϱ
  shows *length* Θ = *Suc* (*length* (*filter* (λ(-, -, *f*). *isReturnFrame f*) *st*))
  using *wfst*
  by *induct auto*

lemma *WfStack-heap-not-empty*:
  assumes *wfst*: *WfStack* Ψ Δ Θ *st* τ *b* ϱ
  and *wfh*: *WfHeap H* Θ
  shows $H \neq []$
  using *wfst wfh*
  by (*auto dest*!: *WfStack-heap-length WfHeap-length*)

lemma *WfFrees-domD*:
  assumes *wfe*: *WfFrees* Δ Γ ϱ *n*
  shows $\varrho \in$ *dom* Δ
  using *wfe*
  by (*rule WfFreesE*, *auto*)

lemma *WfStore-lift-weak*:
  assumes *wfst*: *WfStore* Δ Θ *st* Γ
  and *rl*: $\bigwedge v\ \tau.$ *WfWValue* Δ Θ *v* $\tau \Longrightarrow$ *WfWValue* Δ' Θ' *v* τ
  shows *WfStore* Δ' Θ' *st* Γ
  using *wfst*
  apply (*clarsimp elim*!: *WfStore.cases intro*!: *WfStore.intros*)
  apply (*erule submap-st-weaken*)
  apply (*erule rl*)
  done

lemma *WfHeap-inversionE*:
  assumes *wfh*: *WfHeap H* Θ
  and *lup*: *lookup-heap* Θ *region off* = *Some* τ
  obtains *v* **where** *lookup-heap H region off* = *Some v* **and** *WfHValue v* τ
  using *wfh lup*
proof (*induction arbitrary*: *thesis*)
  case *wfHeapNil* **thus** *?case* **by** *simp*
next
  case (*wfHeapCons H* Θ Σ *R*)

  note *heap-len* = *WfHeap-length* [*OF wfHeapCons.hyps*(*1*)]

show *?case*
proof (*cases region* = *length H*)
case *True*
thus *?thesis* **using** *wfHeapCons.prems wfHeapCons.hyps heap-len*
by (*auto simp add*: *lookup-heap-Some-iff nth-append elim*!: *submap-stE*)
next
case *False*
with *wfHeapCons.prems heap-len* **have** *lookup-heap* Θ *region off* = *Some* τ
by (*clarsimp simp*: *lookup-heap-Some-iff nth-append*)

then obtain v **where** *lookup-heap H region off* = *Some* v **and**
WfHValue v τ **by** (*rule wfHeapCons.IH* [*rotated*])

show *?thesis*
proof (*rule wfHeapCons.prems*(*1*))
from ‹*lookup-heap H region off* = *Some* v›
show *lookup-heap* (*H* @ [*R*]) *region off* = *Some* v
by (*simp add*: *lookup-heap-Some-iff nth-append*)
qed *fact*
qed
qed

7.2 Returns tag weakening

lemma *WfStmt-weaken-returns*:
assumes *wfs*: Γ, Ψ, $\varrho \vdash s : \tau$, b
and *brl*: $b' \longrightarrow b$
shows Γ, Ψ, $\varrho \vdash s : \tau$, b'
using *wfs brl*
by (*induct arbitrary*: b') (*auto intro*: *WfStmt.intros*)

lemma *WfStack-weaken-returns*:
assumes *wfst*: *WfStack* Ψ Δ Θ *st* τ b' ϱ
and *brl*: $b' \longrightarrow b$
shows *WfStack* Ψ Δ Θ *st* τ b ϱ
using *wfst brl*
by *induct* (*auto intro*!: *WfStack.intros elim*: *WfStmt-weaken-returns*)

7.3 Type substitution and free type variables

lemma *tsubst-twice*:
tsubst ϑ (*tsubst* ϑ' τ) = *tsubst* ($\vartheta \circ \vartheta'$) τ
by (*induct* τ) *simp-all*

lemma *tfrees-tsubst*:
tfrees (*tsubst* ϑ τ) = ϑ ‘ *tfrees* τ
by (*cases* τ, *simp-all*)

lemma *tfrees-set-tsubst*:

tfrees-set (*tsubst* ϑ ‘ *S*) = ϑ ‘ *tfrees-set* *S*
unfolding *tfrees-set-def*
by (*auto simp*: *tfrees-tsubst*)

lemma *tfrees-set-Un*:
tfrees-set ($S \cup S'$) = *tfrees-set* $S \cup$ *tfrees-set* S'
unfolding *tfrees-set-def* **by** *simp*

lemma *tfrees-set-singleton* [*simp*]:
tfrees-set $\{\tau\}$ = *tfrees* τ
unfolding *tfrees-set-def* **by** *simp*

lemma *tsubst-cong*:
$[\![(\bigwedge x.\ x \in tfrees\ \tau \Longrightarrow \vartheta\ x = \vartheta'\ x);\ \tau = \tau'\ ]\!] \Longrightarrow tsubst\ \vartheta\ \tau = tsubst\ \vartheta'\ \tau'$
by (*induct* τ) *auto*

lemma *tfrees-set-conv-bex*:
($x \in$ *tfrees-set* *S*) = ($\exists \tau \in S.\ x \in$ *tfrees* τ)
unfolding *tfrees-set-def* **by** *auto*

7.4 Type judgements and free variables

lemma *Expr-tfrees*:
assumes *wf*: $\Gamma \vdash e : \tau$
shows *tfrees* $\tau \subseteq$ *tfrees-set* (*ran* Γ)
using *wf*
by *induction* (*auto simp*: *tfrees-set-def ran-def*)

lemma *ImpureExpr-tfrees*:
assumes *wf*: $\Gamma, \varrho \vdash I\ e : \tau$
shows *tfrees* $\tau \subseteq$ (*tfrees-set* (*ran* Γ) $\cup \{\varrho\}$)
using *wf*
by (*induction*) (*auto dest*: *Expr-tfrees*)

lemma *tfrees-update-storeT*:
assumes $\Gamma \vdash e : \tau$
shows *tfrees-set* (*ran* ($\Gamma(x \mapsto \tau)$)) $\subseteq$ *tfrees-set* (*ran* Γ)
proof −
from ‹$\Gamma \vdash e : \tau$› **have** *t-sub*: *tfrees* $\tau \subseteq$ *tfrees-set* (*ran* Γ) **by** (*rule Expr-tfrees*)

have *tfrees-set* (*ran* ($\Gamma(x \mapsto \tau)$)) $\subseteq$ *tfrees-set* (*ran* $\Gamma \cup \{\tau\}$)
by (*auto simp*: *tfrees-set-def dest*!: *set-mp* [*OF ran-map-upd-subset*])
also have ... = *tfrees-set* (*ran* Γ) **using** *t-sub*
by (*auto simp*: *tfrees-set-def*)
finally show *?thesis* **.**
qed

lemma *tfrees-update-storeT'*:

assumes $\Gamma,\ \varrho \vdash I\ e : \tau$
shows *tfrees-set* (*ran* ($\Gamma(x \mapsto \tau)$)) $\subseteq$ *tfrees-set* (*ran* Γ) $\cup$ $\{\varrho\}$
proof –
from ‹$\Gamma,\ \varrho \vdash I\ e : \tau$› **have** *t-sub*: *tfrees* $\tau \subseteq$ *tfrees-set* (*ran* Γ) $\cup$ $\{\varrho\}$ **by** (*rule ImpureExpr-tfrees*)

have *tfrees-set* (*ran* ($\Gamma(x \mapsto \tau)$)) $\subseteq$ *tfrees-set* (*ran* $\Gamma \cup \{\tau\}$)
by (*auto simp*: *tfrees-set-def dest*!: *set-mp* [*OF ran-map-upd-subset*])
also have ... $\subseteq$ *tfrees-set* (*ran* Γ) $\cup$ $\{\varrho\}$ **using** *t-sub*
by (*auto simp*: *tfrees-set-def*)
finally show *?thesis* **.**
qed

lemma *all-WfE-into-tfrees-set*:
assumes *lall*: *list-all2* ($\lambda e\ \tau.\ \Gamma \vdash e : tsubst\ \vartheta\ \tau$) *es ts*
shows *tfrees-set* (*set* (*map* (*tsubst* ϑ) *ts*)) $\subseteq$ *tfrees-set* (*ran* Γ)
proof –
{
fix *i*
assume $i <$ *length ts*
with *lall* **have** $\Gamma \vdash es\ !\ i : tsubst\ \vartheta\ (ts\ !\ i)$
by (*rule list-all2-nthD2*)
hence *tfrees* (*tsubst* ϑ (*ts* ! *i*)) $\subseteq$ *tfrees-set* (*ran* Γ)
by (*rule Expr-tfrees*)
} **thus** *?thesis* **unfolding** *tfrees-set-def*
by (*fastforce simp*: *list-all2-conv-all-nth in-set-conv-nth*)
qed

lemma *tfrees-set-mono*:
assumes *ss*: $S \subseteq S'$
shows *tfrees-set* $S \subseteq$ *tfrees-set* S'
using *ss*
unfolding *tfrees-set-def*
by *auto*

7.5 Type judgements and substitution

lemma *WfExpr-tsubst*:
assumes *wf*: $\Gamma \vdash e : \tau$
shows (*Option.map* (*tsubst* ϑ) $\circ$ Γ) $\vdash e : tsubst\ \vartheta\ \tau$
using *wf*
by *induction* (*auto intro*: *WfE.intros*)

lemma *WfImpureExpr-tsubst*:
notes *o-apply* [*simp del*]
assumes *wf*: $\Gamma,\ \varrho \vdash I\ e : \tau$
shows (*Option.map* (*tsubst* ϑ) $\circ$ Γ), ($\vartheta\ \varrho$) $\vdash I\ e : tsubst\ \vartheta\ \tau$

using *wf*
by *induction* (*auto intro*!: *WfImpureExpr.intros dest*: *WfExpr-tsubst*)

lemmas *WfExpr-tsubst-Prim* = *WfExpr-tsubst* [**where** τ = *Prim* τ, *simplified*, *standard*]

lemma *WfStmt-tsubst*:
notes *o-apply* [*simp del*]
assumes *wfs*: $\Gamma,\ \Psi,\ \varrho \vdash e : \tau,\ b$
shows $(\mathit{Option.map}\ (\mathit{tsubst}\ \vartheta) \circ \Gamma),\ \Psi,\ \vartheta\ \varrho \vdash e : \mathit{tsubst}\ \vartheta\ \tau,\ b$
using *wfs*
proof (*induction*)
note [*simp del*] = *option-map-o-map-upd fun-upd-apply*
case (*wfBind* $\Gamma\ \varrho\ e\ \tau'\ v\ \Psi\ s\ \tau\ b$)

from ‹$\Gamma,\ \varrho \vdash I\ e : \tau'$›
have *t-sub*: $\mathit{tfrees}\ \tau' \subseteq \mathit{tfrees\text{-}set}\ (\mathit{ran}\ \Gamma) \cup \{\varrho\}$ **by** (*rule ImpureExpr-tfrees*)

show *?case*
proof
have $\mathit{Option.map}\ (\mathit{tsubst}\ \vartheta) \circ \Gamma(v \mapsto \tau'),\ \Psi,\ \vartheta\ \varrho \vdash s : \mathit{tsubst}\ \vartheta\ \tau,\ b$
by (*rule wfBind.IH*)
thus $(\mathit{Option.map}\ (\mathit{tsubst}\ \vartheta) \circ \Gamma)(v \mapsto \mathit{tsubst}\ \vartheta\ \tau'),\ \Psi,\ \vartheta\ \varrho \vdash s : \mathit{tsubst}\ \vartheta\ \tau,\ b$
by (*simp add*: *option-map-o-map-upd*)

from ‹$\Gamma,\ \varrho \vdash I\ e : \tau'$›
show $\mathit{Option.map}\ (\mathit{tsubst}\ \vartheta) \circ \Gamma,\ \vartheta\ \varrho \vdash I\ e : \mathit{tsubst}\ \vartheta\ \tau'$
by (*rule WfImpureExpr-tsubst*)
qed
next
case (*wfWhile* $\Gamma\ e_I\ \tau'\ v\ e_B\ e_S\ \Psi\ \varrho\ s\ \tau\ b$)
note [*simp del*] = *option-map-o-map-upd fun-upd-apply*

show *?case* **using** *wfWhile.hyps*
proof (*intro WfStmt.intros*)
let $?\Gamma = (\mathit{Option.map}\ (\mathit{tsubst}\ \vartheta) \circ \Gamma)$

from ‹$\Gamma \vdash e_I : \tau'$›
have *t-sub*: $\mathit{tfrees}\ \tau' \subseteq \mathit{tfrees\text{-}set}\ (\mathit{ran}\ \Gamma)$ **by** (*rule Expr-tfrees*)

have $\mathit{Option.map}\ (\mathit{tsubst}\ \vartheta) \circ \Gamma(v \mapsto \tau'),\ \Psi,\ \vartheta\ \varrho \vdash s : \mathit{tsubst}\ \vartheta\ \tau,\ b$
by (*rule wfWhile.IH*)
thus $?\Gamma(v \mapsto \mathit{tsubst}\ \vartheta\ \tau'),\ \Psi,\ \vartheta\ \varrho \vdash s : \mathit{tsubst}\ \vartheta\ \tau,\ b$
by (*simp add*: *option-map-o-map-upd*)
qed (*auto simp*: *option-map-o-map-upd* [*symmetric*]
intro: *WfExpr-tsubst-Prim WfExpr-tsubst*)
next
case (*wfCall* $\Psi\ f\ \sigma\ ts\ \Gamma\ \vartheta'\ es\ x\ \varrho\ s\ \tau\ b$)

note [*simp del*] = *option-map-o-map-upd fun-upd-apply*

let *?both* = $(\vartheta \circ \vartheta')$

from ‹Ψ *f* = *Some* (*FunT* σ *ts*)›
show *?case*
proof
  from ‹*list-all2* ($\lambda e\ \tau$. $\Gamma \vdash e$: *tsubst* ϑ' τ) *es ts*›
  show *list-all2* ($\lambda e\ \tau$. *Option.map* (*tsubst* ϑ) $\circ\ \Gamma \vdash e$: *tsubst ?both* τ) *es ts*
  proof
    fix e' τ'
    assume $\Gamma \vdash e'$: *tsubst* ϑ' τ'
    thus *Option.map* (*tsubst* ϑ) $\circ\ \Gamma \vdash e'$: *tsubst ?both* τ'
      by − (*drule WfExpr-tsubst*, *simp add*: *tsubst-twice*)
  qed
  have *Option.map* (*tsubst* ϑ) $\circ\ \Gamma(x \mapsto$ *tsubst* ϑ' σ), Ψ, ϑ $\varrho \vdash s$: *tsubst* ϑ τ, *b*
    by (*rule wfCall.IH*)
  thus (*Option.map* (*tsubst* ϑ) $\circ\ \Gamma$)($x \mapsto$ *tsubst* $(\vartheta \circ \vartheta')$ σ), Ψ, ϑ $\varrho \vdash s$: *tsubst*
ϑ τ, *b*
    by (*simp add*: *option-map-o-map-upd tsubst-twice*)
qed
qed (*auto simp add*: *tsubst-twice*
  intro: *WfStmt.intros*
  dest: *WfExpr-tsubst WfImpureExpr-tsubst*
  WfExpr-tsubst-Prim)

7.6 Type judgements and store (type) updates

lemma *WfStore-upd*:
  assumes *wfst*: *WfStore* Δ Θ *G* Γ
  and *wfwv*: *WfWValue* Δ Θ *v* τ
  shows *WfStore* Δ Θ ($G(x \mapsto v)$) ($\Gamma(x \mapsto \tau)$)
  using *wfst wfwv*
  by (*auto elim*!: *WfStore.cases submap-st-update intro*!: *WfStore*)

lemma *WfFrees-upd-storeT*:
  assumes *wffr*: *WfFrees* Δ Γ ϱ *n*
  and *t-sub*: *tfrees* $\tau \subseteq$ *tfrees-set* (*ran* Γ) $\cup$ $\{\varrho\}$
  shows *WfFrees* Δ ($\Gamma(x \mapsto \tau)$) ϱ *n*
  using *wffr*
proof (*rule WfFreesE*, *intro WfFrees*)
  assume Δ ϱ = *Some n*
  $\forall k \in$ *ran* Δ. $k \leq n$
  tfrees-set (*ran* Γ) $\subseteq$ *dom* Δ *finite* (*dom* Δ)

  thus Δ ϱ = *Some n* **and** $\forall k \in$ *ran* Δ. $k \leq n$
    and *finite* (*dom* Δ) **by** *simp-all*

  have *tfrees-set* (*ran* ($\Gamma(x \mapsto \tau)$)) $\subseteq$ *tfrees-set* (*ran* $\Gamma \cup \{\tau\}$)

by (*auto simp*: *tfrees-set-def dest*!: *set-mp* [*OF ran-map-upd-subset*])
also have ... $\subseteq$ *tfrees-set* (*ran* Γ) $\cup$ $\{\varrho\}$ **using** *t-sub*
by (*auto simp*: *tfrees-set-def*)
also have ... $\subseteq$ (*dom* Δ) **using**
⟨*tfrees-set* (*ran* Γ) $\subseteq$ *dom* Δ⟩⟨Δ ϱ = *Some n*⟩
by (*simp add*: *domI*)
finally show *tfrees-set* (*ran* ($\Gamma(x \mapsto \tau)$)) $\subseteq$ *dom* Δ **.**
qed

7.7 Type judgements and heap (type) updates

lemma *WfWValue-region-extend*:
assumes *wfwv*: *WfWValue* Δ (Θ @ [Σ]) *v* τ'
and *notin*: $x \notin$ *dom* Σ
shows *WfWValue* Δ (Θ @ [$\Sigma(x \mapsto \tau)$]) *v* τ'
using *wfwv notin*
by *cases* (*auto simp*: *lookup-heap-Some-iff nth-append not-less intro*!: *WfWValue.intros split*: *split-if-asm*)

lemma *WfWValue-heap-monotone*:
assumes *wfwv*: *WfWValue* Δ Θ *v* τ'
shows *WfWValue* Δ (Θ @ Θ') *v* τ'
using *wfwv*
by *cases* (*auto intro*!: *WfWValue.intros simp*: *lookup-heap-Some-iff nth-append*)

lemma *WfWValue-heap-mono*:
assumes *wfst*: *WfWValue* Δ Θ *v* τ
and *sub*: *subheap* Θ Θ'
shows *WfWValue* Δ Θ' *v* τ
using *wfst sub*
proof *induction*
case (*wfRefV* Δ ϱ *region* Θ *off* τ)
show *?case*
proof
from ⟨*subheap* Θ Θ'⟩ ⟨*lookup-heap* Θ *region off* = *Some* τ⟩
show *lookup-heap* Θ' *region off* = *Some* τ **by** (*rule subheap-lookup-heapD*)
qed *fact*
qed (*auto intro*: *WfWValue.intros*)

lemma *WfStore-heap-mono*:
assumes *wfst*: *WfStore* Δ Θ *G* Γ
and *sub*: *subheap* Θ Θ'
shows *WfStore* Δ Θ' *G* Γ
proof (*rule*, *rule submap-st-weaken*)
from *wfst* **show** *submap-st* Γ *G* (*WfWValue* Δ Θ) **by** (*auto elim*: *WfStore.cases*)
next
fix *mv nv*
assume *wfwv*: *WfWValue* Δ Θ *mv nv*
thus *WfWValue* Δ Θ' *mv nv* **using** *sub*

by (*rule WfWValue-heap-mono*)
qed

lemma *WfStack-mono*:
assumes *wfst*: *WfStack* Ψ Δ Θ *st* τ *b* ϱ
and *sub*: *subheap* Θ Θ'
shows *WfStack* Ψ Δ Θ' *st* τ *b* ϱ
using *wfst sub*
proof (*induction arbitrary*: Θ')
case *wfStackNil* **thus** *?case* **by** (*clarsimp simp*: *subheap-singleton intro*!: *WfStack.intros*)
next
case (*wfStackFun* Ψ Δ' Θ *st* τ' b' γ *store'* Γ *x* τ *cont* Δ Σ ϱ Θ')

from ‹*subheap* (Θ @ [Σ]) Θ'› **obtain** Σ'
where Θ' = *butlast* Θ' @ [Σ'] **and** *subheap* Θ (*butlast* Θ')
unfolding *subheap-def*
by (*clarsimp simp add*: *list-all2-append1 butlast-append list-all2-Cons1*
cong: *rev-conj-cong*)

moreover have *WfStack* Ψ Δ (*butlast* Θ' @ [Σ']) ((*store'*, *cont*, *ReturnFrame x*) # *st*) τ *True* ϱ
proof
from ‹*subheap* Θ (*butlast* Θ')› **show** *WfStack* Ψ Δ' (*butlast* Θ') *st* τ' b' γ
by (*rule wfStackFun.IH*)

from ‹*WfStore* Δ' Θ *store'* Γ› ‹*subheap* Θ (*butlast* Θ')›
show *WfStore* Δ' (*butlast* Θ') *store'* Γ **by** (*rule WfStore-heap-mono*)

from ‹Θ' = *butlast* Θ' @ [Σ']› ‹*WfFrees* Δ' ($\Gamma(x \mapsto \tau)$) γ (*length* Θ − *1*)› ‹*subheap* Θ (*butlast* Θ')›
show *WfFrees* Δ' ($\Gamma(x \mapsto \tau)$) γ (*length* (*butlast* Θ') − *1*)
by (*clarsimp dest*!: *subheap-lengthD*)
qed *fact+*
ultimately show *?case* **by** *simp*
next
case (*wfStackSeq* Ψ Δ Θ *st* τ b' ϱ *store'* Γ *cont b* Θ')

show *?case*
proof
from ‹*subheap* Θ Θ'›**show** *WfStack* Ψ Δ Θ' *st* τ b' ϱ **by** (*rule wfStackSeq.IH*)
from ‹*WfStore* Δ Θ *store'* Γ› ‹*subheap* Θ Θ'›
show *WfStore* Δ Θ' *store'* Γ **by** (*rule WfStore-heap-mono*)
qed *fact+*
qed

lemma *WfWValue-push-heap*:
assumes *wfst*: *WfWValue* Δ Θ *v* τ
shows *WfWValue* Δ (*push-heap* Θ) *v* τ

using *wfst*
by *induction* (*auto intro*!: *WfWValue.intros simp*: *lookup-heap-Some-iff push-heap-def nth-append*)

lemma *WfStore-push-heap*:
assumes *wfst*: *WfStore* Δ Θ *G* Γ
shows *WfStore* Δ (*push-heap* Θ) *G* Γ
proof (*rule*, *rule submap-st-weaken*)
from *wfst* **show** *submap-st* Γ *G* (*WfWValue* Δ Θ) **by** (*auto elim*: *WfStore.cases*)
next
fix *mv nv*
assume *wfwv*: *WfWValue* Δ Θ *mv nv*
thus *WfWValue* Δ (*push-heap* Θ) *mv nv*
by (*rule WfWValue-push-heap*)
qed

lemma *WfStore-upd-heapT*:
assumes *wfst*: *WfStore* Δ Θ *G* Γ
and *new-T*: *update-heap* Θ *n x* τ = *Some* Θ'
and *x-not-in*: $x \notin$ *dom* (*lookup-heap* Θ *n*)
shows *WfStore* Δ Θ' *G* Γ
proof (*rule*, *rule submap-st-weaken*)
from *wfst* **show** *submap-st* Γ *G* (*WfWValue* Δ Θ)
by (*auto elim*: *WfStore.cases*)
next
fix *mv nv*
assume *wfwv*: *WfWValue* Δ Θ *mv nv*

from *wfwv*
show *WfWValue* Δ Θ' *mv nv*
proof *cases*
case (*wfRefV* ϱ *region off* τ)

hence *lookup-heap* Θ' *region off* = *Some* τ **using** *x-not-in new-T*
by (*cases n* = *region*)
(*fastforce simp*: *lookup-heap-Some-iff nth-append update-heap-def not-less min-absorb2*
split: *split-if-asm*)+
thus *?thesis* **using** *wfRefV*
by (*auto simp add*: *lookup-heap-Some-iff intro*!: *WfWValue.intros*)
qed (*auto intro*: *WfWValue.intros*)
qed

lemma *WfHeap-upd*:
assumes *wfh*: *WfHeap H* Θ
and *wfwv*: *WfHValue v* τ
and *nv*: *n* = *length H* − *1*
and *new-H*: *update-heap H n x v* = *Some H'*

```
  and    new-T: update-heap Θ n x τ = Some Θ'
  and    notin: x ∉ dom (lookup-heap H n)
  shows   WfHeap H' Θ'
  using wfh wfwv notin WfHeap-dom' [where n = n, OF wfh] nv new-H new-T
proof induction
  case wfHeapNil thus ?case by simp
next
  case (wfHeapCons H Θ Σ R)

  note heap-len = WfHeap-length [OF wfHeapCons.hyps(1)]

  have x ∉ dom R using wfHeapCons.prems by (simp add: lookup-heap-def)
  hence x ∉ dom Σ using wfHeapCons.prems heap-len by auto
  with ‹WfHeap H Θ› ‹submap-st Σ R WfHValue› ‹finite (dom R)›
  show ?case using wfHeapCons.prems heap-len
    unfolding update-heap-def
    by (auto simp add: update-heap-def nth-append
      intro!: WfHeap.intros submap-st-update
      elim!: submap-st-weaken WfWValue-region-extend)
qed

lemma WfHeap-upd-same-type:
  assumes wfh: WfHeap H Θ
  and     wfwv: WfHValue v τ
  and    new-H: update-heap H n x v = Some H'
  and      lup: lookup-heap Θ n x = Some τ
  shows   WfHeap H' Θ
  using wfh wfwv new-H lup
proof (induction arbitrary: H')
  case wfHeapNil thus ?case by simp
next
  case (wfHeapCons H Θ Σ R H')

  note heap-len = WfHeap-length [OF wfHeapCons.hyps(1)]

  show ?case
  proof (cases n = length Θ)
    case True thus ?thesis using wfHeapCons.prems wfHeapCons.hyps heap-len
      by (auto simp add: update-heap-def nth-append submap-st-def
            intro!:  WfHeap.intros)
  next
    case False
    with ‹lookup-heap (Θ @ [Σ]) n x = Some τ› have n < length Θ
      by (simp add: lookup-heap-Some-iff)
    hence n < length H using heap-len by simp

    from ‹update-heap (H @ [R]) n x v = Some H'›
    have H' = (butlast H') @ [R]
      unfolding update-heap-def using False heap-len
```

by (*auto simp*: *nth-append butlast-snoc not-less butlast-append*
split: *split-if-asm*)
moreover have *WfHeap* (*butlast* H' @ [R]) (Θ @ [Σ])
proof
have *WfHValue* v τ **by** *fact*
moreover
from ‹*update-heap* (H @ [R]) n x v = *Some* H'› ‹H' = (*butlast* H') @ [R]›
have *update-heap* (H @ [R]) n x v = *Some* (*butlast* H' @ [R]) **by** *simp*
hence *update-heap* H n x v = *Some* (*butlast* H') **using** ‹n < *length* H›
by (*rule update-heap-shrink*)
moreover
from ‹*lookup-heap* (Θ @ [Σ]) n x = *Some* τ› ‹n < *length* Θ›
have *lookup-heap* Θ n x = *Some* τ
by (*simp add*: *lookup-heap-Some-iff nth-append*)

ultimately show *WfHeap* (*butlast* H') Θ
by (*rule wfHeapCons.IH*)
qed *fact*+
ultimately show *?thesis* **by** *simp*
qed
qed

7.8 Region environment updates

lemma *WfWValue-renv-mono*:
assumes *wfwv*: *WfWValue* Δ Θ v τ
and *sub*: $\Delta \subseteq_m \Delta'$
shows *WfWValue* Δ' Θ v τ
using *wfwv sub*
by *induct* (*auto intro*!: *WfWValue.intros*
dest!: *map-leD*)

lemma *WfStack-renv-mono*:
notes *fun-upd-apply* [*simp del*]
assumes *wfst*: *WfStack* Ψ Δ Θ *st* τ b ϱ
and *sub*: $\Delta \subseteq_m \Delta'$
shows *WfStack* Ψ Δ' Θ *st* τ b ϱ
using *wfst sub*
proof *induction*
case *wfStackNil* **show** *?case* **..**
next
case *wfStackFun*
thus *?case* **by** (*auto intro*!: *WfStack.intros elim*: *map-le-trans*)
next
case (*wfStackSeq* Ψ Δ Θ *st* τ b' ϱ *store'* Γ *cont* b)

show *?case*
proof

from ‹*WfStore* Δ Θ *store′* Γ› ‹$\Delta \subseteq_m \Delta'$›
show *WfStore* Δ' Θ *store′* Γ
by (*auto intro*!: *map-le-map-upd-right elim*!: *WfStore-lift-weak WfWValue-renv-mono*
)
show *WfStack* Ψ Δ' Θ *st* τ *b′* ϱ **by** (*rule wfStackSeq.IH*) *fact*

from ‹*tfrees-set* (*ran* Γ) $\subseteq$ *dom* Δ›
show *tfrees-set* (*ran* Γ) $\subseteq$ *dom* Δ'
proof (*rule order-trans*)
from ‹$\Delta \subseteq_m \Delta'$› **show** *dom* Δ $\subseteq$ *dom* Δ' **by** (*rule map-le-implies-dom-le*)
qed
qed *fact*+
qed

theory *EvalSafe*
imports *Semantics TypeSystemProps*
begin

lemma *Expr-safe*:
assumes *wfe*: $\Gamma \vdash e : \tau$
and *wfg*: *WfStore* Δ Θ *G* Γ
shows $\exists v.\ G \models e \downarrow v \wedge$ *WfWValue* Δ Θ *v* τ
using *wfe wfg*
proof *induct*
case (*wfVar* Γ *x* τ)
have *WfStore* Δ Θ *G* Γ **and** Γ *x* = *Some* τ **by** *fact*+
then obtain *v* **where** *G x* = *Some v* **and** *WfWValue* Δ Θ *v* τ
by (*auto elim*!: *submap-stE WfStore.cases*)
thus *?case* **by** *simp*
next
case (*wfBinCmp* Γ e_1 e_2 *bop*)
thus *?case*
by (*clarsimp elim*!: *WfNatVE WfPNatVE intro*!: *WfWValue.intros WfPrim-Value.intros*)
next
case (*wfBinOp* Γ e_1 e_2 *bop*)
thus *?case*
by (*clarsimp elim*!: *WfNatVE WfPNatVE intro*!: *WfWValue.intros WfPrim-Value.intros*)
qed (*auto intro*: *WfWValue.intros WfPrimValue.intros*)

lemma *Expr-safeE*:
assumes *wfe*: $\Gamma \vdash e : \tau$

and *wfg*: *WfStore* Δ Θ *G* Γ
and *rl*: $\bigwedge v.$ ⟦ $G \models e \downarrow v$; *WfWValue* Δ Θ v τ ⟧ $\Longrightarrow R$
shows R
using *wfe wfg* **by** (*auto dest*!: *Expr-safe intro*: *rl*)

lemma *ImpureExpr-safeE*:
notes *subheap-refl* [*intro*]
fixes τ :: *'r wtype*
assumes *wfe*: Γ, $\varrho \vdash I$ e : τ
and *wfs*: *WfStore* Δ Θ *st* Γ *WfHeap* H Θ $H \neq []$ *WfFrees* Δ Γ ϱ (*length* Θ − *1*)
obtains H' Θ' v **where** $st \models H, e \Downarrow H', v$ *WfHeap* H' Θ' *WfWValue* Δ Θ' v τ *subheap* Θ Θ'
using *wfe wfs*
proof (*induction*)
case (*wfPure* Γ e τ ϱ)
note *that* = *wfPure.prems*(*1*)

have $\Gamma \vdash e : \tau$ **by** *fact+*
then obtain v **where** $st \models e \downarrow v$ *WfWValue* Δ Θ v τ **using** ‹*WfStore* Δ Θ *st* Γ›
by (*auto elim*!: *Expr-safeE*)

thus *?case* **using** ‹*WfHeap* H Θ›
by (*auto intro*!: *that*)
next
case (*wfNewRef* Γ e τ ϱ)
note *that* = *wfNewRef.prems*(*1*)

from ‹$\Gamma \vdash e$: *Prim* τ› ‹*WfStore* Δ Θ *st* Γ› **obtain** v **where** $st \models e \downarrow$ *PrimV* v *WfPrimValue* v τ
by (*auto elim*!: *Expr-safeE WfWValue.cases*)

show *?case*
proof (*rule that*)
let *?region* = (*length* H − *1*)
let *?off* = *fresh-in-heap* H *?region*
let *?H'* = *take ?region* H @ [(H ! *?region*)(*?off* $\mapsto$ *StoredV* v)]
let *?*Θ' = *take ?region* Θ @ [(Θ ! *?region*)(*?off* $\mapsto$ *Stored* τ)]

from ‹*WfHeap* H Θ› **have** *fin*: *finite* (*dom* (*lookup-heap* H *?region*))
by (*auto elim*: *WfHeap.cases*)

with ‹*WfHeap* H Θ›
have *?off* $\notin$ *dom* (*lookup-heap* Θ *?region*)
by (*rule contra-subsetD* [*OF WfHeap-dom' fresh-in-heap-fresh*])

from ‹$st \models e \downarrow$ *PrimV* v› ‹$H \neq []$›

show *st* $\models$ *H*, *NewRef e* $\Downarrow$ *?H'*, *RefV ?region ?off*
by (*clarsimp simp*: *Let-def update-heap-def*)

from ‹*WfHeap H* Θ› **have** *length* Θ = *length H* **by** (*rule WfHeap-length* [*symmetric*])
with ‹*H* $\neq$ []› **have** *update-heap* Θ *?region ?off* (*Stored* τ) = *Some ?*Θ'
by (*auto simp add*: *update-heap-def diff-Suc-less*)

from ‹*WfPrimValue v* τ› **have** *WfHValue* (*StoredV v*) (*Stored* τ) **..**
with ‹*WfHeap H* Θ› **show** *WfHeap ?H' ?*Θ'
proof (*rule WfHeap-upd* [*OF - - refl*])
from ‹*H* $\neq$ []› **show** *update-heap H ?region ?off* (*StoredV v*) = *Some ?H'*
by (*simp add*: *update-heap-def*)
from *fin* **show** *?off* $\notin$ *dom* (*lookup-heap H ?region*)
by (*rule fresh-in-heap-fresh*)
qed *fact*

have *notin*: *?off* $\notin$ *dom* (*lookup-heap* Θ *?region*) **by** *fact*
show *subheap* Θ *?*Θ' **using** ‹*length* Θ = *length H*› ‹*H* $\neq$ []›
using *subheap-take-drop* [*OF notin refl*] **by** *simp*

show *WfWValue* Δ *?*Θ' (*RefV ?region ?off*) (*RefT* ϱ (*Stored* τ))
proof
from ‹*length* Θ = *length H*›
show *lookup-heap ?*Θ' *?region ?off* = *Some* (*Stored* τ)
by (*auto simp*: *lookup-heap-Some-iff nth-append min-absorb2*)

from ‹*WfFrees* Δ Γ ϱ (*length* Θ − *1*)› ‹*length* Θ = *length H*›
show Δ ϱ = *Some ?region* **by** (*clarsimp elim*!: *WfFreesE*)
qed
qed
next
case (*wfReadRef* Γ *e* γ τ ϱ)
note *that* = *wfReadRef*.*prems*(*1*)

from ‹Γ $\vdash$ *e* : *RefT* γ (*Stored* τ)› ‹*WfStore* Δ Θ *st* Γ›
obtain *v* **where** *st* $\models$ *e* $\downarrow$ *v* *WfWValue* Δ Θ *v* (*RefT* γ (*Stored* τ))
by (*erule Expr-safeE*)

moreover from ‹*WfWValue* Δ Θ *v* (*RefT* γ (*Stored* τ))›
obtain *region off* **where** *v* = *RefV region off*
lookup-heap Θ *region off* = *Some* (*Stored* τ) **by** (*rule WfRefVE*)

from ‹*WfHeap H* Θ› ‹*lookup-heap* Θ *region off* = *Some* (*Stored* τ)›
obtain *v'* **where** *lookup-heap H region off* = *Some* (*StoredV v'*) **and** *WfPrimValue v'* τ
by (*auto elim*!: *WfHeap-inversionE WfHValue*.*cases*)

show *?case*

proof (*rule that*)
from ‹$st \models e \downarrow v$› ‹$v = RefV\ region\ off$› ‹$lookup\text{-}heap\ H\ region\ off = Some\ (StoredV\ v')$›
show $st \models H,\ ReadRef\ e \Downarrow H,\ PrimV\ v'$ **by** *simp*

from ‹$WfPrimValue\ v'\ \tau$› **show** $WfWValue\ \Delta\ \Theta\ (PrimV\ v')\ (Prim\ \tau)$ **..**
qed *fact+*
next
case ($wfWriteRef\ \Gamma\ e_2\ \tau\ e_1\ \gamma\ \varrho$)
note $that = wfWriteRef.prems(1)$

from ‹$\Gamma \vdash e_1 : RefT\ \gamma\ (Stored\ \tau)$› ‹$WfStore\ \Delta\ \Theta\ st\ \Gamma$›
obtain v **where** $st \models e_1 \downarrow v$ $WfWValue\ \Delta\ \Theta\ v\ (RefT\ \gamma\ (Stored\ \tau))$
by (*erule Expr-safeE*)

from ‹$WfWValue\ \Delta\ \Theta\ v\ (RefT\ \gamma\ (Stored\ \tau))$›
obtain *region off* **where** $v = RefV\ region\ off$
$lookup\text{-}heap\ \Theta\ region\ off = Some\ (Stored\ \tau)$ **by** (*rule WfRefVE*)

from ‹$\Gamma \vdash e_2 : Prim\ \tau$› ‹$WfStore\ \Delta\ \Theta\ st\ \Gamma$›
obtain v' **where** $st \models e_2 \downarrow PrimV\ v'$ $WfPrimValue\ v'\ \tau$
by (*auto elim*!: *Expr-safeE WfWValue.cases*)

from ‹$WfHeap\ H\ \Theta$› ‹$lookup\text{-}heap\ \Theta\ region\ off = Some\ (Stored\ \tau)$›
obtain *hv* **where** $lookup\text{-}heap\ H\ region\ off = Some\ (StoredV\ hv)$
by (*auto elim*!: *WfHeap-inversionE WfHValue.cases*)
then obtain H' **where** $update\text{-}heap\ H\ region\ off\ (StoredV\ v') = Some\ H'$
by (*rule lookup-heap-into-update-heap-same*)

show *?case*
proof (*rule that*)
from ‹$st \models e_1 \downarrow v$› ‹$v = RefV\ region\ off$› ‹$st \models e_2 \downarrow PrimV\ v'$›
‹$update\text{-}heap\ H\ region\ off\ (StoredV\ v') = Some\ H'$›
show $st \models H,\ WriteRef\ e_1\ e_2 \Downarrow H',\ PrimV\ UnitV$
by *clarsimp*

show $WfWValue\ \Delta\ \Theta\ (PrimV\ UnitV)\ (Prim\ UnitT)$ **by** (*intro WfWValue.intros WfPrimValue.intros*)

from ‹$WfHeap\ H\ \Theta$› ‹$WfPrimValue\ v'\ \tau$›
‹$update\text{-}heap\ H\ region\ off\ (StoredV\ v') = Some\ H'$›
‹$lookup\text{-}heap\ \Theta\ region\ off = Some\ (Stored\ \tau)$›
show $WfHeap\ H'\ \Theta$
by (*rule WfHeap-upd-same-type* [*OF - wfStoredV*])
qed *rule*
qed

```
lemma ImpureExpr-safe-stateE:
  notes subheap-refl [intro]
  assumes wfe: Γ, ϱ ⊢I e : τ
  and     wfs: WfState S Γ Ψ τ′ b ϱ
  obtains H′ Θ′ Δ v where store S ⊨ heap S, e ⇓ H′, v WfHeap H′ Θ′
   WfStore Δ Θ′ (store S) Γ  WfWValue Δ Θ′ v τ  WfStack Ψ Δ Θ′ (stack S) τ′ b
ϱ
   WfFrees Δ Γ ϱ (length Θ′ − 1)
proof −
  from wfs obtain Θ Δ where
    WfStore Δ Θ (store S) Γ
    WfHeap (heap S) Θ
    WfStack Ψ Δ Θ (stack S) τ′ b ϱ
    WfFrees Δ Γ ϱ (length Θ − 1)
    ..
  moreover
  from ‹WfStack Ψ Δ Θ (stack S) τ′ b ϱ› ‹WfHeap (heap S) Θ› have heap S ≠ []
    by (rule WfStack-heap-not-empty)

  ultimately obtain H′ Θ′ v where store S ⊨ heap S, e ⇓ H′, v WfHeap H′ Θ′
    WfWValue Δ Θ′ v τ subheap Θ Θ′
    using wfe
    by (auto elim!: ImpureExpr-safeE dest: WfHeap-length)

  from ‹subheap Θ Θ′› have length Θ = length Θ′ by (rule subheap-lengthD)

  show ?thesis
  proof (rule that)
    show WfStore Δ Θ′ (store S) Γ by (rule WfStore-heap-mono) fact+
    show WfStack Ψ Δ Θ′ (stack S) τ′ b ϱ by (rule WfStack-mono) fact+

    from ‹length Θ = length Θ′› show WfFrees Δ Γ ϱ (length Θ′ − 1)
      by (rule subst) fact
  qed fact+
qed

end
```

8 Progress

```
lemma Progress:
  assumes wff: WfFuns F Ψ
  and     wfst: WfState S Γ Ψ τ b ϱ
  and     wfs: Γ, Ψ, ϱ ⊢ s : τ, b
  shows ∃R. F ⊨ (S, s) ⊳ R
  using wfs wfst wff
proof (induct arbitrary: S)
```

case (*wfSkip* Γ Ψ ϱ τ *S*)
from ‹*WfState S* Γ Ψ τ *False* ϱ›
obtain Δ Θ **where** *WfStack* Ψ Δ Θ (*stack S*) τ *False* ϱ **..**
then obtain *store′ cont st* **where** *stack S* = (*store′*, *cont*, *SeqFrame*) # *st*
  by (*rule wfStackFalseE*)
show *?case* **by** (*rule exI StepSkip*)+ *fact*

next
case (*wfReturn* Γ *e* τ Ψ ϱ *b S*)
from ‹$\Gamma \vdash e : \tau$› ‹*WfState S* Γ Ψ τ *b* ϱ›
obtain *v* **where** *e-to-v*: *store S* $\models e \downarrow v$
  by (*auto elim*: *Expr-safeE elim*!: *WfState.cases*)

show *?case*
proof (*cases stack S* = [])
  case *True* **thus** *?thesis* **using** *e-to-v* **by** (*auto intro*: *Step.intros*)
next
  case *False*
  then obtain *st cont fclass stack′*
    where *stackS*: *stack S* = (*st*, *cont*, *fclass*) # *stack′*
    by (*fastforce simp*: *neq-Nil-conv*)

  thus *?thesis* **using** *e-to-v*
    by (*cases fclass*) (*auto intro*: *Step.intros*)
qed

next
case (*wfBind* Γ ϱ *e* τ' *v* Ψ *s* τ *b S*)

from ‹$\Gamma, \varrho \vdash I\ e : \tau'$› ‹*WfState S* Γ Ψ τ *b* ϱ›
obtain H' Θ' Δ v' **where** *eval*: *store S* $\models$ *heap S*, $e \Downarrow H', v'$
  and *wfh′*: *WfHeap* H' Θ'
  and *wfs′*: *WfStore* Δ Θ' (*store S*) Γ
  and *wfwv′*: *WfWValue* Δ Θ' v' τ'
  by (*auto elim*!: *ImpureExpr-safe-stateE*)

from *wfs′ wfwv′* **have** *wfs-upd*: *WfStore* Δ Θ' (*store* $S(x \mapsto v')$) ($\Gamma(x \mapsto \tau')$)
  by (*rule WfStore-upd*)

with *eval* **show** *?case*
  by (*auto intro*: *exI Step.intros*)

next
case (*wfIf* Γ *e* Ψ ϱ s_1 τ *b* s_2 *S*)
from ‹$\Gamma \vdash e$: *BOOL*› ‹*WfState S* Γ Ψ τ *b* ϱ›
obtain *bv* **where** *e-to-v*: *store S* $\models e \downarrow$ *PrimV* (*BoolV bv*)
  by (*auto elim*!: *Expr-safeE WfPBoolVE WfBoolVE elim*!: *WfState.cases*)

thus *?case*
  by (*auto intro*: *exI Step.intros*)

next
case (*wfWhile* Γ e_I τ' *v* e_B e_S Ψ ϱ *s* τ *b S*)

from ⟨$\Gamma \vdash e_I : \tau'$⟩ ⟨*WfState S* Γ Ψ τ *False* ϱ⟩
obtain *v* **where** *e-to-v*: *store* $S \models e_I \downarrow v$
  by (*auto elim*: *Expr-safeE elim*!: *WfState.cases*)

thus *?case* **by** (*auto intro*: *exI Step.intros*)
next
  case (*wfSeq* Γ Ψ ϱ s_1 τ s_2 *b S*)
  show *?case* **by** *rule rule*
next
  case (*wfCall* Ψ *f* σ *ts* Γ ϑ *es x* ϱ *s* τ *b S*)

  from ⟨Ψ *f* = *Some* (*FunT* σ *ts*)⟩ ⟨*WfFuns F* Ψ⟩
  obtain *fbody* **where** *F f* = *Some fbody WfFunc* Ψ *fbody* (*FunT* σ *ts*)
    by (*auto elim*!: *WfFuns.cases submap-stE*)
  then obtain *as body* **where**
    things: *F f* = *Some* (*Func as body*)
    length as = *length ts*
    by (*cases fbody*, *auto elim*!: *WfFunc.cases*)

  show *?case*
  proof (*rule exI*, *rule StepCall* [*OF* - - *refl*])
    show *F f* = *Some* (*Func as body*) **by** *fact*
  next
    have *list-all2* ($\lambda e\ \tau.\ \Gamma \vdash e : tsubst\ \vartheta\ \tau$) *es ts* **by** *fact*
    hence *length es* = *length ts* **..**
    with *things* **show** *length as* = *length es* **by** *simp*
  next
    show $\forall v \in set\ (map\ (ExpV\ (store\ S))\ es).\ v \neq None$
    proof
      fix *v*
      assume $v \in set\ (map\ (ExpV\ (store\ S))\ es)$
      then obtain *e* **where** *ein*: $e \in set\ es$ **and** *ev*: *ExpV* (*store S*) *e* = *v*
        by *clarsimp*

      have *list-all2* ($\lambda e\ \tau.\ \Gamma \vdash e : tsubst\ \vartheta\ \tau$) *es ts* **by** *fact*
      then obtain *t* **where** $\Gamma \vdash e : tsubst\ \vartheta\ t$ **using** *ein*
        by (*rule list-all2-ballE1*)

      moreover have *WfState S* Γ Ψ τ *b* ϱ **by** *fact*
      ultimately show $v \neq None$ **using** *ev*
        by (*auto elim*: *Expr-safeE elim*!: *WfState.cases*)
    qed
  qed
qed

9 Preservation

lemma *ImpureExp-length*:
  assumes *eval*: $st \models H,\ e \Downarrow H',\ v$
  shows *length* H' = *length* H
  using *eval*
  by (*induction rule*: *ImpureExpV.induct*)
    (*auto simp add*: *Let-def option-bind-Some-iff update-heap-length split*: *option.splits wvalue.splits*)

lemma *WfValue-return*:
  assumes *wfwv*: *WfWValue* $\Delta\ \Theta\ v\ \tau$
  and *frees*: *WfFrees* $\Delta\ \ \Gamma\ \varrho$ (*length* $\Theta - 1$) *WfFrees* Δ' $(\Gamma'(x \mapsto \tau))$ γ (*length* (*butlast* Θ) $- 1$) $\Delta' \subseteq_m \Delta$
  and *len*: *length* $\Theta > 1$
  shows *WfWValue* Δ' (*butlast* Θ) $v\ \tau$
  using *wfwv frees len*
proof *induction*
  case *wfPrimV* **show** *?case* **by** *rule fact*
next
  case (*wfRefV* $\Delta\ \varrho'$ *region* Θ *off* τ)

  show *?case*
  proof
    have $\varrho' \in$ *tfrees-set* (*ran* $(\Gamma'(x \mapsto$ *RefT* $\varrho'\ \tau)))$
      unfolding *tfrees-set-def ran-def* **by** *auto*
    also from ‹*WfFrees* Δ' $(\Gamma'(x \mapsto$ *RefT* $\varrho'\ \tau))$ γ (*length* (*butlast* Θ) $- 1$)›
    have ... $\subseteq$ *dom* Δ' **by** (*auto elim*!: *WfFreesE*)
    finally show $\Delta'\ \varrho'$ = *Some region*
      using ‹$\Delta' \subseteq_m \Delta$› ‹$\Delta\ \varrho'$ = *Some region*›
      by (*auto dest*: *map-leD*)

    from ‹*WfFrees* Δ' $(\Gamma'(x \mapsto$ *RefT* $\varrho'\ \tau))$ γ (*length* (*butlast* Θ) $- 1$)›
    have $\forall k \in$ *ran* Δ'. $k \leq$ *length* (*butlast* Θ) $- 1$ **..**

    with ‹$\Delta'\ \varrho'$ = *Some region*› ‹*length* $\Theta > 1$›
    have *region* < *length* (*butlast* Θ)
      by (*auto simp*: *ran-def dest*!: *set-mp*)

    thus *lookup-heap* (*butlast* Θ) *region off* = *Some* τ
      using ‹*lookup-heap* Θ *region off* = *Some* τ›
      by (*simp add*: *lookup-heap-Some-iff nth-butlast*)
  qed
qed

lemma *Preservation*:
  fixes τ :: $'r$ :: {*infinite*} *wtype*

assumes *wff*: *WfFuns* $F\ \Psi$
and *wfst*: *WfState* $S\ \Gamma\ \Psi\ \tau\ b\ \varrho$
and *wfs*: $\Gamma,\ \Psi,\ \varrho \vdash s : \tau,\ b$
and *step*: $F \models (S,\ s) \triangleright$ *Normal* $(S',\ s')$
shows $\exists \Gamma'\ \tau'\ b'\ \gamma.$ *WfState* $S'\ \Gamma'\ \Psi\ \tau'\ b'\ \gamma \wedge \Gamma',\ \Psi,\ \gamma \vdash s' : \tau',\ b'$
using *wfs step wfst wff*
proof (*induction arbitrary*: $S\ s'$)
case (*wfSkip* $\Gamma\ \Psi\ \varrho\ \tau\ S\ s'$)

from $\langle F \models (S,\ Skip) \triangleright Normal\ (S',\ s')\rangle$
obtain *store′ stack′* **where**
Sv′: $S' = S\ (\!|\ store := store',\ stack := stack'\ |\!)$
and *stackv*: *stack* $S = (store',\ s',\ SeqFrame)\ \#\ stack'$
by (*rule StepSkipE*)

with $\langle WfState\ S\ \Gamma\ \Psi\ \tau\ False\ \varrho\rangle$
obtain $\Theta\ \Delta\ \Gamma'\ b'$ **where**
WfHeap (*heap* S) Θ
WfStack $\Psi\ \Delta\ \Theta$ *stack′* $\tau\ b'\ \varrho$
WfStore $\Delta\ \Theta$ *store′* Γ'
tfrees-set (*ran* Γ') $\subseteq$ *dom* Δ
WfFrees $\Delta\ \Gamma\ \varrho$ (*length* $\Theta - 1$)
$\Gamma',\ \Psi,\ \varrho \vdash s' : \tau,\ b'$
by (*auto elim*!: *WfStateE WfStackSeqE*)
moreover from $\langle$*tfrees-set* (*ran* Γ') $\subseteq$ *dom* $\Delta\rangle$ $\langle$*WfFrees* $\Delta\ \Gamma\ \varrho$ (*length* $\Theta - 1$)$\rangle$
have *WfFrees* $\Delta\ \Gamma'\ \varrho$ (*length* $\Theta - 1$)
by (*auto elim*!: *WfFreesE intro*!: *WfFrees*)

ultimately show *?case* **using** *Sv′*
by (*auto intro*: *WfState intro*!: *exI*)
next
case (*wfReturn* $\Gamma\ e\ \tau\ \Psi\ \varrho\ b\ S\ s'$)

from $\langle F \models (S,\ Return\ e) \triangleright Normal\ (S',\ s')\rangle$
show *?case*
proof (*cases rule*: *StepReturnE*)
case (*SeqFrame st cont stack′*)
with *wfReturn* **show** *?thesis*
by (*clarsimp elim*!: *WfStateE WfStackSeqE*)
(*auto intro*: *WfState intro*!: *WfStmt.intros exI*)
next
case (*ReturnFrame store′ cont x stack′ v*)

hence *Sv′*: $S' = (\!|store = store'(x \mapsto v),\ heap = pop\text{-}heap\ (heap\ S),\ stack = stack'|\!)$
and *sv′*: $s' = cont$ **by** *simp-all*

from $\langle WfState\ S\ \Gamma\ \Psi\ \tau\ b\ \varrho\rangle$
obtain $\Theta\ \Delta$ **where** *WfStore* $\Delta\ \Theta$ (*store* S) Γ

WfHeap (*heap* S) Θ
WfStack Ψ Δ Θ (*stack* S) τ b ϱ
WfFrees Δ Γ ϱ (*length* Θ $-$ *1*)
..

from ‹*WfStack* Ψ Δ Θ (*stack* S) τ b ϱ› ‹*stack* S = (*store*′, *cont*, *ReturnFrame* x) # *stack*′›
obtain τ' b' Δ' Γ' γ **where** *WfStack* Ψ Δ' (*butlast* Θ) *stack*′ τ' b' γ
WfStore Δ' (*butlast* Θ) *store*′ Γ' $\Gamma'(x \mapsto \tau)$, Ψ, γ $\vdash$ *cont* : τ', b'
WfFrees Δ' ($\Gamma'(x \mapsto \tau)$) γ (*length* (*butlast* Θ) $-$ *1*) $\Delta' \subseteq_m \Delta$
by (*auto elim*!: *WfStackFunE*)

show *?thesis*
proof (*intro exI conjI*)
show *WfState* S' ($\Gamma'(x \mapsto \tau)$) Ψ τ' b' γ **unfolding** *Sv*′
proof (*rule*, *simp-all del*: *One-nat-def*)
from ‹*WfStore* Δ' (*butlast* Θ) *store*′ Γ'›
show *WfStore* Δ' (*butlast* Θ) (*store*′$(x \mapsto v)$) ($\Gamma'(x \mapsto \tau)$)
proof (*rule WfStore-upd*)
from ‹*WfStore* Δ Θ (*store* S) Γ› ‹*store* S $\models$ e $\downarrow$ v› ‹Γ $\vdash$ e : τ›
have *WfWValue* Δ Θ v τ **by** (*auto elim*: *Expr-safeE*)

thus *WfWValue* Δ' (*butlast* Θ) v τ
proof (*rule WfValue-return*)
from ‹*WfStack* Ψ Δ Θ (*stack* S) τ b ϱ›
‹*stack* S = (*store*′, *cont*, *ReturnFrame* x) # *stack*′›
show *1* < *length* Θ
by (*auto dest*: *WfStack-heap-length*)
qed *fact*+
qed

from ‹*WfStack* Ψ Δ Θ (*stack* S) τ b ϱ› ‹*WfHeap* (*heap* S) Θ›
have *heap* S $\neq$ [] **by** (*rule WfStack-heap-not-empty*)
with ‹*WfHeap* (*heap* S) Θ› **show** *WfHeap* (*pop-heap* (*heap* S)) (*butlast* Θ)
by (*auto simp*: *pop-heap-def elim*: *WfHeap.cases*)
qed *fact*+

from ‹$\Gamma'(x \mapsto \tau)$, Ψ, γ $\vdash$ *cont* : τ', b'› *sv*′ **show** $\Gamma'(x \mapsto \tau)$, Ψ, γ $\vdash$ s' : τ', b' **by** *simp*
qed
next
case *Finish*
thus *?thesis* **using** *wfReturn* **by** *simp*
qed
next
case (*wfBind* Γ ϱ e τ' x Ψ s τ b S s')

from ‹F $\models$ (S, *Bind* x e s) $\triangleright$ *Normal* (S', s')›
obtain H' v' **where**

```
    eval: store S ⊨ heap S, e ⇓ H′, v′
    and Sv′: S′ = S⦇store := store S(x ↦ v′), heap := H′⦈
    and ss′: s′ = s
    by cases

  show ?case
  proof (intro exI conjI)
    from ‹s′ = s› show Γ(x ↦ τ′), Ψ, ϱ ⊢ s′ : τ, b
      by (rule ssubst) fact

    from ‹Γ, ϱ ⊢I e : τ′› ‹WfState S Γ Ψ τ b ϱ›
    obtain H″ Θ′ Δ v″ where eval′: store S ⊨ heap S, e ⇓ H″, v″
      and wfh′: WfHeap H″ Θ′
      and wfs′: WfStore Δ Θ′ (store S) Γ
      and wfst′: WfStack Ψ Δ Θ′ (stack S) τ b ϱ
      and wfwv′: WfWValue Δ Θ′ v″ τ′
      and WfFrees Δ Γ ϱ (length Θ′ − 1)
      by (fastforce elim!: ImpureExpr-safe-stateE)

    moreover from ‹WfState S Γ Ψ τ b ϱ›
    have heap S ≠ []
      by (rule WfStateE) (erule (1) WfStack-heap-not-empty)

    moreover
    from ‹WfFrees Δ Γ ϱ (length Θ′ − 1)›
    have WfFrees Δ (Γ(x ↦ τ′)) ϱ (length Θ′ − 1)
    proof (rule WfFrees-upd-storeT)
      from ‹Γ, ϱ ⊢I e : τ′›
      show tfrees τ′ ⊆ tfrees-set (ran Γ) ∪ {ϱ}
        by (rule ImpureExpr-tfrees)
    qed
    ultimately show WfState S′ (Γ(x ↦ τ′)) Ψ τ b ϱ
      using wfst′ Sv′ eval eval′
      by (auto intro!: WfStore-upd WfState dest!: ImpureExp-length)
  qed
next
  case (wfIf Γ e Ψ ϱ s₁ τ b s₂ S s′)

  from ‹F ⊨ (S, stmt.If e s₁ s₂) ▷ Normal (S′, s′)›
  obtain bl where
    store S ⊨ e ↓ PrimV (BoolV bl)
    and ss: S′ = S s′ = (if bl then s₁ else s₂)
    by cases

  show ?case
  proof (intro exI conjI)
    from wfIf.hyps ss show Γ, Ψ, ϱ ⊢ s′ : τ, b
      by simp
    from ‹S′= S› show WfState S′ Γ Ψ τ b ϱ by (rule ssubst) fact
```

qed
next
case $(wfWhile\ \Gamma\ e_I\ \tau'\ x\ e_B\ e_S\ \Psi\ \varrho\ s\ \tau\ b\ S\ s')$

from ‹$F \models (S,\ For\ x\ e_I\ e_B\ e_S\ s) \rhd Normal\ (S',\ s')$›
obtain v **where**
eval: $store\ S \models e_I \downarrow v$
and *ss*: $S' = S(\!|store := store\ S(x \mapsto v)|\!)$
$s' = stmt.If\ e_B\ (s\ ;;\ For\ x\ e_S\ e_B\ e_S\ s)\ Skip$
by *cases*

from ‹$WfState\ S\ \Gamma\ \Psi\ \tau\ False\ \varrho$›
obtain $\Theta\ \Delta$ **where** *wfs*: $WfStore\ \Delta\ \Theta\ (store\ S)\ \Gamma$
$WfHeap\ (heap\ S)\ \Theta$
$WfStack\ \Psi\ \Delta\ \Theta\ (stack\ S)\ \tau\ False\ \varrho$
$WfFrees\ \Delta\ \Gamma\ \varrho\ (length\ \Theta - 1)$
..

show *?case*
proof (*intro conjI exI*)
show $\Gamma(x \mapsto \tau'),\ \Psi,\ \varrho \vdash s' : \tau,\ False$ **using** *ss wfWhile.hyps*
by (*auto intro*!: *WfStmt.intros elim*: *WfStmt-weaken-returns*)

show $WfState\ S'\ (\Gamma(x \mapsto \tau'))\ \Psi\ \tau\ False\ \varrho$
proof
from ‹$\Gamma \vdash e_I : \tau'$› **have** $tfrees\ \tau' \subseteq tfrees\text{-}set\ (ran\ \Gamma)$
by (*rule Expr-tfrees*)

hence $tfrees\ \tau' \subseteq tfrees\text{-}set\ (ran\ \Gamma) \cup \{\varrho\}$ **by** *auto*
with ‹$WfFrees\ \Delta\ \Gamma\ \varrho\ (length\ \Theta - 1)$›
show $WfFrees\ \Delta\ (\Gamma(x \mapsto \tau'))\ \varrho\ (length\ \Theta - 1)$
by (*rule WfFrees-upd-storeT*)

from *wfs ss* **show** $WfHeap\ (heap\ S')\ \Theta$
and $WfStack\ \Psi\ \Delta\ \Theta\ (stack\ S')\ \tau\ False\ \varrho$ **by** *simp-all*

from ‹$WfStore\ \Delta\ \Theta\ (store\ S)\ \Gamma$› ‹$\Gamma \vdash e_I : \tau'$› *ss eval*
show $WfStore\ \Delta\ \Theta\ (store\ S')\ (\Gamma(x \mapsto \tau'))$
by (*auto intro*!: *WfStore-upd elim*!: *Expr-safeE*)
qed
qed
next
case $(wfSeq\ \Gamma\ \Psi\ \varrho\ s_1\ \tau\ s_2\ b\ S\ s')$

with ‹$F \models (S,\ s_1\ ;;\ s_2) \rhd Normal\ (S',\ s')$›
have *Sv'*: $S' = S(\!|stack := (store\ S,\ s_2,\ SeqFrame)\ \#\ stack\ S|\!)$
and *ss*: $s' = s_1$
by (*auto elim*: *StepSeqE*)

show *?case*
proof (*intro exI conjI*)
from ‹*WfState S* Γ Ψ τ *b* ϱ› ‹Γ, Ψ, $\varrho \vdash s_2 : \tau$, *b*›
show *WfState S'* Γ Ψ τ *False* ϱ **using** *Sv'*
by (*auto elim*!: *WfStateE WfStackFunE elim*: *WfFreesE intro*!: *WfState WfStack.intros elim*: *WfStmt-weaken-returns*)

show Γ, Ψ, $\varrho \vdash s' : \tau$, *False* **using** *ss* **by** (*rule ssubst*) *fact*
qed
next
case (*wfCall* Ψ *f* σ *ts* Γ ϑ *es x* ϱ *s* τ *b S s'*)

from ‹*F* $\models$ (*S*, *Call x f es s*) $\triangleright$ *Normal* (*S'*, *s'*)›
obtain *args body* **where**
Sv': *S'* =
(|*store* = [*args* [$\mapsto$] *map* (*the* $\circ$ *ExpV* (*store S*)) *es*],
heap = *push-heap* (*heap S*),
stack = (*store S*, *s*, *ReturnFrame x*) # *stack S*|)
and *Ff*: *F f* = *Some* (*Func args body*)
and *largs1*: *length args* = *length es*
and *all-eval*: $\forall$ *e*$\in$*set es*. $\exists$ *y*. *store S* $\models$ *e* $\downarrow$ *y*
and *ss*: *s'* = *body*
by (*rule StepCallE*) *auto*

from ‹*WfState S* Γ Ψ τ *b* ϱ› *Sv' ss*
obtain Θ Δ **where**
wfs: *WfStore* Δ Θ (*store S*) Γ
WfHeap (*heap S*) Θ
WfStack Ψ Δ Θ (*stack S*) τ *b* ϱ
WfFrees Δ Γ ϱ (*length* Θ − *1*)
by (*auto elim*!: *WfStateE*)

from ‹*WfFuns F* Ψ› ‹*F f* = *Some* (*Func args body*)› ‹Ψ *f* = *Some* (*FunT* σ *ts*)›
obtain γ **where** *largs2*: *length args* = *length ts*
and $\gamma \in$ − *tfrees-set* (*set ts*)
[*args* [$\mapsto$] *ts*], Ψ, $\gamma \vdash$ *body* : σ, *True*
tfrees $\sigma \subseteq$ *tfrees-set* (*set ts*)
by (*auto elim*!: *WfFuns.cases WfFunc.cases submap-stE*)

show *?case*
proof (*intro exI conjI*)
let *?*γ = *fresh* (*dom* Δ)

def *pi-def*: $\pi \equiv \lambda\varrho$. *if* $\varrho = \gamma$ *then ?*γ *else if* $\varrho \in$ *tfrees-set* (*set ts*) *then* ϑ ϱ *else* ϱ
let *?*Γ = [*args* [$\mapsto$] *map* (*tsubst* π) *ts*]

let *?vs* = *map* (*the* $\circ$ *ExpV* (*store S*)) *es*

let *?G* = [*args* [$\mapsto$] *?vs*]
let *?*Δ = (Δ(*?*$\gamma \mapsto$ *length* Θ))

from ‹*WfFrees* Δ Γ ϱ (*length* Θ − *1*)› **have** *finite* (*dom* Δ)**..**
hence *?*$\gamma \notin$ *dom* Δ
by (*rule fresh-not-in*)

have *pi-gamma*: π γ = *?*γ **unfolding** *pi-def* **by** *simp*

from ‹*list-all2* (λe τ. $\Gamma \vdash e$: *tsubst* ϑ τ) *es ts*›
have *list-all2* (λe τ. $\Gamma \vdash e$: *tsubst* π τ) *es ts*
proof (*cases rule*: *list-all2-weaken*)
case (*P i*)
thus $\Gamma \vdash$ (*es* ! *i*) : *tsubst* π (*ts* ! *i*) **using** ‹$\gamma \in -$ *tfrees-set* (*set ts*)›
unfolding *pi-def*
apply −
apply (*clarsimp simp*: *all-set-conv-all-nth tfrees-set-conv-bex in-set-conv-nth split*: *split-if-asm split-if*)
apply (*subst tsubst-cong* [**where** $\vartheta' = \vartheta$])
apply (*auto simp*: *tfrees-set-conv-bex cong*: *tsubst-cong*)
done
qed

from ‹[*args* [$\mapsto$] *ts*], Ψ, $\gamma \vdash$ *body* : σ, *True*›
have *?*Γ, Ψ, π $\gamma \vdash s'$: *tsubst* π σ, *True* **using** *ss*
by (*simp add*: *option-map-map-upds* [*symmetric*]) (*erule WfStmt-tsubst*)
thus *?*Γ, Ψ, *?*$\gamma \vdash s'$: *tsubst* π σ, *True* **using** *pi-gamma* **by** *simp*

have *delta-subm*: $\Delta \subseteq_m$ *?*Δ **using** ‹*?*$\gamma \notin$ *dom* Δ›
unfolding *map-le-def* **by** *simp*

show *WfState* S' *?*Γ Ψ (*tsubst* π σ) *True* *?*γ
unfolding Sv'
proof (*rule*, *simp-all*)
have *WfStore ?*Δ Θ *?G ?*Γ
proof (*rule*, *rule submap-st-list-all2I*)

show *list-all2* (*WfWValue ?*Δ Θ) *?vs* (*map* (*tsubst* π) *ts*)
proof (*rule list-all2-all-nthI*)
show *length ?vs* = *length* (*map* (*tsubst* π) *ts*) **using** *largs1 largs2* **by** *simp*
next
fix *n*
assume *n* < *length* (*map* (*the* $\circ$ *ExpV* (*store S*)) *es*)
hence *lts*: *n* < *length es n* < *length ts* **using** *largs1 largs2* **by** *auto*

from ‹*list-all2* (λe τ. $\Gamma \vdash e$: *tsubst* π τ) *es ts*› ‹*n* < *length es*›
have $\Gamma \vdash$ *es* ! *n* : *tsubst* π (*ts* ! *n*) **by** (*rule list-all2-nthD*)
with ‹*WfStore* Δ Θ (*store S*) Γ› **obtain** *v* **where** *store S* $\models$ *es* ! *n* $\downarrow$ *v WfWValue* Δ Θ *v* (*tsubst* π (*ts* ! *n*))

```
        by (auto elim!: Expr-safeE )
      thus WfWValue ?Δ Θ (?vs ! n) (map (tsubst π) ts ! n)
        using lts delta-subm
        apply (simp add: nth-map )
        apply (erule (1) WfWValue-renv-mono)
        done
    qed

    show length args = length (map (tsubst π) ts) using largs2 by simp
    show length args = length ?vs using largs1 by simp
  qed
  thus WfStore ?Δ (push-heap Θ) ?G ?Γ by (rule WfStore-push-heap)

  from ‹WfHeap (heap S) Θ›
  show WfHeap (push-heap (heap S)) (push-heap Θ)
    unfolding push-heap-def
    by (rule wfHeapCons) simp-all

  have ts-delta: π ‘ tfrees-set (set ts) ⊆ dom Δ
  proof −
    from ‹list-all2 (λe τ. Γ ⊢ e : tsubst π τ) es ts›
    have tfrees-set (set (map (tsubst π) ts)) ⊆ tfrees-set (ran Γ)
      by (rule all-WfE-into-tfrees-set)
    moreover
    from ‹WfFrees Δ Γ ϱ (length Θ − 1)› have tfrees-set (ran Γ) ⊆ dom Δ ..
    ultimately show ?thesis
      by (simp add: tfrees-set-tsubst)
  qed

  show WfStack Ψ ?Δ (push-heap Θ) ((store S, s, ReturnFrame x) # stack S)
(tsubst π σ) True ?γ
    unfolding push-heap-def
  proof (rule wfStackFun)
    from ‹WfFrees Δ Γ ϱ (length Θ − 1)›
    show WfFrees Δ (Γ(x ↦ tsubst π σ)) ϱ (length Θ − 1)
    proof (rule WfFrees-upd-storeT)
      from ‹tfrees σ ⊆ tfrees-set (set ts)›
    have tfrees (tsubst π σ) ⊆ π ‘ tfrees-set (set ts) by (auto simp: tfrees-tsubst)
      also have ...⊆ tfrees-set (ran Γ)
        using ‹list-all2 (λe τ. Γ ⊢ e : tsubst π τ) es ts›
        by (auto dest!: all-WfE-into-tfrees-set simp add: tfrees-set-tsubst)
      finally show tfrees (tsubst π σ) ⊆ tfrees-set (ran Γ) ∪ {ϱ}
        by auto
    qed

    from ‹Γ(x ↦ tsubst ϑ σ), Ψ, ϱ ⊢ s : τ, b›
         ‹tfrees σ ⊆ tfrees-set (set ts)›
         ‹γ ∈ − tfrees-set (set ts)›
    show Γ(x ↦ tsubst π σ), Ψ, ϱ ⊢ s : τ, b
```

unfolding *pi-def*
by (*subst tsubst-cong* [**where** $\vartheta' = \vartheta$]) (*auto simp*: *tfrees-set-conv-bex*)
qed *fact+*

show *WfFrees* $?\Delta$ $?\Gamma$ $?\gamma$ (*length* (*push-heap* Θ) $-$ *Suc 0*)
proof
show *tfrees-set* (*ran* $?\Gamma$) $\subseteq$ *dom* $?\Delta$ **using** *ts-delta*
apply *simp*
apply (*rule order-trans*)
apply (*rule tfrees-set-mono*)
apply (*rule ran-map-upds*)
apply (*auto simp add*: *tfrees-set-tsubst*)
done

show $?\Delta$ $?\gamma$ = *Some* (*length* (*push-heap* Θ) $-$ *Suc 0*) **by** *simp*

from ‹*finite* (*dom* Δ)› **show** *finite* (*dom* $?\Delta$) **by** *simp*

from ‹*WfFrees* Δ Γ ϱ (*length* Θ $-$ *1*)› **have** $\forall k \in$ *ran* Δ. $k \leq$ *length* Θ $-$ *1* **..**
thus $\forall k \in$ *ran* $?\Delta$. $k \leq$ *length* (*push-heap* Θ) $-$ *Suc 0* **using** ‹$?\gamma \notin$ *dom* Δ›
by (*auto simp*: *domIff*)
qed
qed
qed
qed

10 Soundness

lemma *Soundness*:
fixes Ψ :: ('*fun*, '*r* :: {*infinite*}) *funsT*
assumes *wfp*: *WfProgram* Ψ *S s*
and *wff*: *WfFuns F* Ψ
shows ($\exists$ *m nv*. $m \leq n \wedge F, m \models (S, s) \triangleright^*$ *Finished* (*PrimV* (*NatV nv*)))
$\vee$ ($\exists S'$ s'. $F, n \models (S, s) \triangleright^*$ *Normal* (S', s') $\wedge$ *WfProgram* Ψ S' s')
using *wfp*
proof (*induction n arbitrary*: *S s*)
case *0*

show *?case*
proof (*intro disjI2 exI conjI*)
show F, *0* $\models (S, s) \triangleright^*$ *Normal* (S, s) **..**
qed *fact*
next
case (*Suc n*)

from ‹*WfProgram* Ψ *S s*› **obtain** Γ *b* τ ϱ **where**
WfState *S* Γ Ψ τ *b* ϱ **and** Γ, Ψ, $\varrho \vdash s : \tau$, *b*
by (*auto elim*!: *WfProgram.cases*)

with ‹*WfFuns F* Ψ› **obtain** *R* **where** $F \models (S, s) \rhd R$
by (*auto dest*!: *Progress*)

show *?case*
proof (*cases R*)
case (*Finished v*)
with ‹$F \models (S, s) \rhd R$› **have** $F \models (S, s) \rhd$ *Finished v* **by** *simp*
then obtain *e* **where** $s =$ *Return e stack* $S = []$ *store* $S \models e \downarrow v$
by *cases simp-all*

from ‹$s =$ *Return e*› ‹Γ, Ψ, $\varrho \vdash s : \tau$, *b*› **have** $\Gamma \vdash e : \tau$
by (*auto elim*: *WfStmt.cases*)

from ‹*WfState S* Γ Ψ τ *b* ϱ› ‹*stack* $S = []$›
obtain Θ Δ **where** *WfStore* Δ Θ (*store S*) Γ
and $\tau =$ *NAT* **and** $b =$ *True*
by (*auto elim*: *WfStack.cases elim*!: *WfStateE*)

from ‹$\Gamma \vdash e : \tau$› ‹*WfStore* Δ Θ (*store S*) Γ› ‹*store* $S \models e \downarrow v$› ‹$\tau =$ *NAT*›
obtain *nv* **where** $v =$ *PrimV* (*NatV nv*)
by (*auto elim*!: *Expr-safeE WfNatVE WfPNatVE*)
with ‹$F \models (S, s) \rhd$ *Finished v*› **have** $F \models (S, s) \rhd$ *Finished* (*PrimV* (*NatV nv*))
by *simp*

show *?thesis*
proof (*intro exI disjI1 conjI*)
show *Suc 0* $\leq$ *Suc n*
by *simp*

from ‹$F \models (S, s) \rhd$ *Finished* (*PrimV* (*NatV nv*))›
show *F*, *Suc 0* $\models (S, s) \rhd^*$ *Finished* (*PrimV* (*NatV nv*))
by (*auto intro*: *StepN.intros*)
qed
next
case (*Normal Ss*)
with ‹$F \models (S, s) \rhd R$› **obtain** S' s' **where** $F \models (S, s) \rhd$ *Normal* (S', s') **by** (*cases Ss*, *auto*)

with ‹*WfFuns F* Ψ› ‹*WfState S* Γ Ψ τ *b* ϱ› ‹Γ, Ψ, $\varrho \vdash s : \tau$, *b*›
have *WfProgram* Ψ S' s'
by (*auto dest*!: *Preservation intro*: *WfProgram.intros*)

hence ($\exists\, m$ *nv*. $m \leq n \wedge F$, $m \models (S', s') \rhd^*$ *Finished* (*PrimV* (*NatV nv*))) $\vee$

$(\exists\, S''\ s''.\ F,\ n \models (S',\ s') \rhd^{*}\ \mathit{Normal}\ (S'',\ s'') \land \mathit{WfProgram}\ \Psi\ S''\ s'')$
by $(\mathit{rule\ Suc.IH})$

thus $\mathit{?thesis}$
proof $(\mathit{elim\ disjE\ conjE\ exE})$
fix $m\ nv$
assume $m \leq n\ F,\ m \models (S',\ s') \rhd^{*}\ \mathit{Finished}\ (\mathit{PrimV}\ (\mathit{NatV}\ nv))$

show $\mathit{?thesis}$
proof $(\mathit{intro\ exI\ disjI1\ conjI})$
from $\langle m \leq n\rangle$ **show** $\mathit{Suc}\ m \leq \mathit{Suc}\ n$ **..**

from $\langle F \models (S,\ s) \rhd \mathit{Normal}\ (S',\ s')\rangle$
$\langle F,\ m \models (S',\ s') \rhd^{*}\ \mathit{Finished}\ (\mathit{PrimV}\ (\mathit{NatV}\ nv))\rangle$
show $F,\ \mathit{Suc}\ m \models (S,\ s) \rhd^{*}\ \mathit{Finished}\ (\mathit{PrimV}\ (\mathit{NatV}\ nv))$
by $(\mathit{rule\ StepN\text{-}add\text{-}head})$
qed
next
fix $S''\ s''$
assume $F,\ n \models (S',\ s') \rhd^{*}\ \mathit{Normal}\ (S'',\ s'')\ \mathit{WfProgram}\ \Psi\ S''\ s''$
show $\mathit{?thesis}$
proof $(\mathit{intro\ exI\ disjI2\ conjI})$
from $\langle F \models (S,\ s) \rhd \mathit{Normal}\ (S',\ s')\rangle$
$\langle F,\ n \models (S',\ s') \rhd^{*}\ \mathit{Normal}\ (S'',\ s'')\rangle$
show $F,\ \mathit{Suc}\ n \models (S,\ s) \rhd^{*}\ \mathit{Normal}\ (S'',\ s'')$
by $(\mathit{rule\ StepN\text{-}add\text{-}head})$
qed *fact*
qed
qed
qed

lemma *Initial-program*:
fixes $\Psi :: ('\mathit{fun},\ '\mathit{r} :: \{\mathit{infinite}\})\ \mathit{funsT}$
defines $S \equiv (\!|\ \mathit{store} = \mathit{Map.empty},\ \mathit{heap} = [\mathit{Map.empty}],\ \mathit{stack} = []\ |\!)$
defines $(\gamma{::}'r) \equiv \mathit{fresh}\ \{\}$
assumes *wff*: $\mathit{WfFuns}\ F\ \Psi$
and *main*: $\Psi\ \mathit{main} = \mathit{Some}\ (\mathit{FunT}\ \mathit{NAT}\ [])$
shows $\mathit{WfProgram}\ \Psi\ S\ (\mathit{Call}\ x\ \mathit{main}\ []\ (\mathit{Return}\ (\mathit{Var}\ x)))$
proof
from *main*
show $\mathit{Map.empty},\ \Psi,\ \gamma \vdash \mathit{Call}\ x\ \mathit{main}\ []\ (\mathit{Return}\ (\mathit{Var}\ x)) : \mathit{NAT},\ \mathit{True}$
by $(\mathit{auto\ intro!}{:}\ \mathit{WfStmt.intros\ WfE.intros})$

show $\mathit{WfState}\ S\ \mathit{Map.empty}\ \Psi\ \mathit{NAT}\ \mathit{True}\ \gamma$
proof
let $?\Delta = [\gamma \mapsto 0]$
let $?\Theta = [\mathit{Map.empty}]$

show $\mathit{WfStore}\ ?\Delta\ ?\Theta\ (\mathit{store}\ S)\ \mathit{Map.empty}$

by *rule simp*

have *WfHeap* ([] @ [*Map.empty*]) ([] @ [*Map.empty*])
apply (*rule WfHeap.intros*)
apply *rule*
apply *simp-all*
done
thus *WfHeap* (*heap S*) *?*Θ **unfolding** *S-def* **by** *simp*

show *WfStack* Ψ *?*Δ *?*Θ (*stack S*) *NAT True* γ
unfolding *S-def* **by** *simp* (*rule wfStackNil*)

show *WfFrees ?*Δ *Map.empty* γ (*length* [*Map.empty*] − *1*)
by *rule* (*simp-all add*: *tfrees-set-def*)
qed
qed

lemma *Initial-program-result*:
fixes Ψ :: (*'fun*, *'r* :: {*infinite*}) *funsT*
and *main* :: *'fun* **and** *x* :: *'var*
defines *S* ≡ (⦇ *store* = *Map.empty*, *heap* = [*Map.empty*], *stack* = [] ⦈)
defines *s* ≡ *Call x main* [] (*Return* (*Var x*))
assumes *wff*: *WfFuns F* Ψ
and *main*: Ψ *main* = *Some* (*FunT NAT* [])
obtains (*Terminates*) *n nv* **where** *F*, *n* ⊨ (*S*, *s*) ▷* *Finished* (*PrimV* (*NatV nv*))
| (*Diverges*) ∀ *n*. (∃ *S' s'*. *F*, *n* ⊨ (*S*, *s*) ▷* *Normal* (*S'*, *s'*) ∧ *WfProgram* Ψ *S' s'*)
proof −
from *wff main* **have** *WfProgram* Ψ *S s*
unfolding *S-def s-def*
by (*rule Initial-program*)

have ∀ *n*. (∃ *n nv*. *F*, *n* ⊨ (*S*, *s*) ▷* *Finished* (*PrimV* (*NatV nv*)))
∨ (∃ *S' s'*. *F*, *n* ⊨ (*S*, *s*) ▷* *Normal* (*S'*, *s'*) ∧ *WfProgram* Ψ *S' s'*)
(**is** ∀ *n*. *?FINISHES n* ∨ *?DIVERGES n*)
proof
fix *n*

from ‹*WfProgram* Ψ *S s*› *wff*
have (∃ *m nv*. *m* ≤ *n* ∧ *F*, *m* ⊨ (*S*, *s*) ▷* *Finished* (*PrimV* (*NatV nv*)))
∨ (∃ *S' s'*. *F*, *n* ⊨ (*S*, *s*) ▷* *Normal* (*S'*, *s'*) ∧ *WfProgram* Ψ *S' s'*)
by (*rule Soundness*)

thus *?FINISHES n* ∨ *?DIVERGES n*
proof (*elim exE disjE conjE*)
fix *m nv*
assume *m* ≤ *n F*, *m* ⊨ (*S*, *s*) ▷* *Finished* (*PrimV* (*NatV nv*))
hence *?FINISHES n* **by** *auto*

thus *?thesis* **..**
next
fix S' s'
assume $F,\ n \models (S,\ s) \rhd^* \mathit{Normal}\ (S',\ s')$ $\mathit{WfProgram}\ \Psi\ S'\ s'$
hence *?DIVERGES* n **by** *auto*
thus *?thesis* **..**
qed
qed
thus *?thesis*
by (*auto intro*: *Terminates Diverges*)
qed