

Interpreter_Optimizations

Martin Desharnais

March 17, 2025

Abstract

This Isabelle/HOL formalization builds on the *VeriComp* entry of the *Archive of Formal Proofs* to provide the following contributions:

- an operational semantics for a realistic virtual machine (Std) for dynamically typed programming languages;
- the formalization of an inline caching optimization (Inca), a proof of bisimulation with (Std), and a compilation function;
- the formalization of an unboxing optimization (Ubx), a proof of bisimulation with (Inca), and a simple compilation function.

This formalization was described in [1].

Contents

1	Generic lemmas	3
2	Environment	4
2.1	Generic lemmas	7
2.2	List-based implementation of environment	7
3	nth_opt	9
4	Generic lemmas	10
5	Non-empty list	13
6	Monadic bind	14
7	Conversion functions	15
8	pred_map	18

9 Rest	20
9.1 Function definition	21
9.2 Program	23
9.3 Stack frame	23
9.4 Dynamic state	24
10 n-ary operations	29
11 n-ary operations	30
12 Inline caching	31
12.1 Static representation	31
12.2 Semantics	32
13 n-ary operations	36
14 Unboxed caching	37
14.1 Static representation	37
14.2 Semantics	40
15 Locale imports	58
16 Normalization	59
17 Equivalence of call stacks	62
18 Simulation relation	67
19 Backward simulation	67
20 Forward simulation	84
21 Bisimulation	108
22 Strongest postcondition	108
23 Range validations	109
24 Basic block validation	110
25 Function definition validation	110
26 Program definition validation	110
27 Generic program rewriting	110
28 Lifting	112

29 Optimization	115
30 Compilation of function definition	121
31 Compilation of function environment	124
32 Compilation of program	126
32.1 Completeness of compilation	128
33 Dynamic values	131
34 Normal operations	132
35 Inlined operations	133
36 Unboxed operations	134
36.1 Typing	135
37 Generic definitions	139
38 Simulation relation	143
39 Backward simulation	143
40 Forward simulation	148
41 Bisimulation	153
41.1 Compilation of function definitions	153
41.2 Compilation of function environments	155
41.3 Compilation of programs	156
41.4 Completeness of compilation	157
theory <i>Env</i>	
imports Main HOL-Library.Library	
begin	

1 Generic lemmas

```

lemma map-of-list-allI:
  assumes  $\bigwedge k v. f k = \text{Some } v \implies P(k, v)$  and
          $\bigwedge k v. \text{map-of } kvs k = \text{Some } v \implies f k = \text{Some } v$  and
         distinct (map fst kvs)
  shows list-all P kvs
  using assms(2-)
proof (induction kvs)
  case Nil
  then show ?case by simp
next

```

```

case (Cons kv kvs)
from Cons.prems(1) have f (fst kv) = Some (snd kv)
  by simp
hence P kv
  using assms(1)
  by (cases kv; simp)
moreover have list-all P kvs
  using Cons.IH Cons.prems
  by (metis Some-eq-map-of-iff distinct.simps(2) list.set-intros(2) list.simps(9))
ultimately show ?case by simp
qed

```

2 Environment

```

locale env =
  fixes
    empty :: 'env and
    get :: 'env  $\Rightarrow$  'key  $\Rightarrow$  'val option and
    add :: 'env  $\Rightarrow$  'key  $\Rightarrow$  'val  $\Rightarrow$  'env and
    to-list :: 'env  $\Rightarrow$  ('key  $\times$  'val) list
  assumes
    get-empty: get empty x = None and
    get-add-eq: get (add e x v) x = Some v and
    get-add-neq: x  $\neq$  y  $\implies$  get (add e x v) y = get e y and
    to-list-correct: map-of (to-list e) = get e and
    to-list-distinct: distinct (map fst (to-list e))

begin

declare get-empty[simp]
declare get-add-eq[simp]
declare get-add-neq[simp]

definition singleton where
  singleton  $\equiv$  add empty

fun add-list :: 'env  $\Rightarrow$  ('key  $\times$  'val) list  $\Rightarrow$  'env where
  add-list e [] = e |
  add-list e (x # xs) = add (add-list e xs) (fst x) (snd x)

definition from-list :: ('key  $\times$  'val) list  $\Rightarrow$  'env where
  from-list  $\equiv$  add-list empty

lemma from-list-correct: get (from-list xs) = map-of xs
proof (induction xs)
  case Nil
  then show ?case
    using get-empty by (simp add: from-list-def)
next

```

```

case (Cons x xs)
show ?case
  using get-add-eq get-add-neq Cons.IH by (auto simp: from-list-def)
qed

lemma from-list-Nil[simp]: from-list [] = empty
  by (simp add: from-list-def)

lemma get-from-list-to-list: get (from-list (to-list e)) = get e
proof
  fix x
  show get (from-list (to-list e)) x = get e x
    unfolding from-list-correct
    unfolding to-list-correct
    by (rule refl)
qed

lemma to-list-list-allI:
  assumes  $\bigwedge k v. \text{get } e k = \text{Some } v \implies P(k, v)$ 
  shows list-all P (to-list e)
proof (rule map-of-list-allI[of get e])
  fix k v
  show get e k = Some v  $\implies P(k, v)$ 
    using assms by simp
next
  fix k v
  show map-of (to-list e) k = Some v  $\implies \text{get } e k = \text{Some } v$ 
    unfolding to-list-correct by assumption
next
  show distinct (map fst (to-list e))
    by (rule to-list-distinct)
qed

definition map-entry where
map-entry e k f ≡ case get e k of None ⇒ e | Some v ⇒ add e k (f v)

lemma get-map-entry-eq[simp]: get (map-entry e k f) k = map-option f (get e k)
  unfolding map-entry-def
  by (cases get e k) simp-all

lemma get-map-entry-neq[simp]: x ≠ y  $\implies \text{get } (\text{map-entry } e x f) y = \text{get } e y$ 
  unfolding map-entry-def
  by (cases get e x) simp-all

lemma dom-map-entry[simp]: dom (get (map-entry e k f)) = dom (get e)
  unfolding dom-def
  apply (rule Collect-cong)
  by (metis None-eq-map-option-iff get-map-entry-eq get-map-entry-neq)

```

```

lemma get-map-entry-conv:
  get (map-entry e x f) y = map-option (λv. if x = y then f v else v) (get e y)
  unfolding map-entry-def
  by (cases get e x; cases x = y; simp add: option.map-ident)

lemma map-option-comp-map-entry:
  assumes ∀x ∈ ran (get e). f (g x) = fx
  shows map-option f ∘ get (map-entry e k g) = map-option f ∘ get e
  proof (intro ext)
    fix x
    show (map-option f ∘ get (map-entry e k g)) x = (map-option f ∘ get e) x
    proof (cases k = x)
      case True
      thus ?thesis
        using assms(1)
        by (auto simp: get-map-entry-eq option.map-comp intro!: map-option-cong
          ranI)
      next
        case False
        then show ?thesis
        by (simp add: get-map-entry-neq)
      qed
    qed

lemma map-option-comp-get-add:
  assumes k ∈ dom (get e) and ∀x ∈ ran (get e). f v = fx
  shows map-option f ∘ get (add e k v) = map-option f ∘ get e
  proof (intro ext)
    fix x
    show (map-option f ∘ get (add e k v)) x = (map-option f ∘ get e) x
    proof (cases x = k)
      case True
      show ?thesis
      proof (cases get e x)
        case None
        thus ?thesis
        using True assms(1) by auto
      next
        case (Some v')
        thus ?thesis
        using True assms(2) by (auto intro: ranI)
      qed
    next
      case False
      thus ?thesis by simp
    qed
  qed

end

```

```

end
theory Env-list
  imports Env HOL-Library.Library
begin

```

2.1 Generic lemmas

```

lemma map-of-filter:
   $x \neq y \implies \text{map-of}(\text{filter}(\lambda z. \text{fst } z \neq y) \text{ zs}) x = \text{map-of} \text{ zs } x$ 
  by (induction zs) auto

```

2.2 List-based implementation of environment

```

context
begin

```

```

qualified type-synonym ('key, 'val) t = ('key × 'val) list

```

```

qualified definition empty :: ('key, 'val) t where
  empty ≡ []

```

```

qualified definition get :: ('key, 'val) t ⇒ 'key ⇒ 'val option where
  get ≡ map-of

```

```

qualified definition add :: ('key, 'val) t ⇒ 'key ⇒ 'val ⇒ ('key, 'val) t where
  add e k v ≡ AList.update k v e

```

```

term filter

```

```

qualified fun to-list :: ('key, 'val) t ⇒ ('key × 'val) list where
  to-list [] = [] |
  to-list (x # xs) = x # to-list (filter (\(k, v). k ≠ fst x) xs)

```

```

lemma get-empty: get empty x = None
  unfolding get-def empty-def
  by simp

```

```

lemma get-add-eq: get (add e x v) x = Some v
  unfolding get-def add-def
  by (simp add: update-Some-unfold)

```

```

lemma get-add-neq: x ≠ y ⇒ get (add e x v) y = get e y
  unfolding get-def add-def
  by (simp add: update-conv')

```

```

lemma to-list-correct: map-of (to-list e) = get e
  unfolding get-def
proof (induction e rule: to-list.induct)
  case 1

```

```

then show ?case by simp
next
  case (? x xs)
  show ?case
  proof (rule ext)
    fix k'
    show map-of (to-list (x # xs)) k' = map-of (x # xs) k'
    using 2.IH map-of-filter
    by (auto simp add: case-prod-beta')
  qed
qed

lemma set-to-list: set (to-list e) ⊆ set e
  by (induction e rule: to-list.induct) auto

lemma to-list-distinct: distinct (map fst (to-list e))
proof (induction e rule: to-list.induct)
  case 1
  then show ?case by simp
next
  case (? x xs)
  then show ?case
  using set-to-list
  by fastforce
qed

end

global-interpretation env-list:
  env Env-list.empty Env-list.get Env-list.add Env-list.to-list
  defines
    singleton = env-list.singleton and
    add-list = env-list.add-list and
    from-list = env-list.from-list
    apply (unfold-locales)
    by (simp-all add: get-empty get-add-eq get-add-neq to-list-correct to-list-distinct)

export-code Env-list.empty Env-list.get Env-list.add Env-list.to-list singleton add-list
from-list
  in SML module-name Env

end
theory List-util
  imports Main
begin

inductive same-length :: 'a list ⇒ 'b list ⇒ bool where

```

```

same-length-Nil: same-length [] []
same-length-Cons: same-length xs ys ==> same-length (x # xs) (y # ys)

code-pred same-length .

lemma same-length-iff-eq-lengths: same-length xs ys <=> length xs = length ys
proof
  assume same-length xs ys
  then show length xs = length ys
  by (induction xs ys rule: same-length.induct) simp-all
next
  assume length xs = length ys
  then show same-length xs ys
  proof (induction xs arbitrary: ys)
    case Nil
    then show ?case
    by (simp add: same-length-Nil)
next
  case (Cons x xs)
  then show ?case
  by (metis length-Suc-conv same-length-Cons)
qed
qed

lemma same-length-Cons:
  same-length (x # xs) ys ==>  $\exists y \text{ } ys'. \text{ } ys = y \# ys'$ 
  same-length xs (y # ys) ==>  $\exists x \text{ } xs'. \text{ } xs = x \# xs'$ 
proof -
  assume same-length (x # xs) ys
  then show  $\exists y \text{ } ys'. \text{ } ys = y \# ys'$ 
  by (induction x # xs ys rule: same-length.induct) simp
next
  assume same-length xs (y # ys)
  then show  $\exists x \text{ } xs'. \text{ } xs = x \# xs'$ 
  by (induction xs y # ys rule: same-length.induct) simp
qed

```

3 nth_opt

```

fun nth-opt where
  nth-opt (x # -) 0 = Some x |
  nth-opt (- # xs) (Suc n) = nth-opt xs n |
  nth-opt - - = None

lemma nth-opt-eq-Some-conv: nth-opt xs n = Some x <=> n < length xs ∧ xs ! n
= x
by (induction xs n rule: nth-opt.induct; simp)

lemmas nth-opt-eq-SomeD[dest] = nth-opt-eq-Some-conv[THEN iffD1]

```

4 Generic lemmas

```

lemma list-rel-imp-pred1:
  assumes
    list-all2 R xs ys and
     $\bigwedge x y. (x, y) \in \text{set}(\text{zip } xs \text{ } ys) \implies R x y \implies P x$ 
  shows list-all P xs
  using assms
  by (induction xs ys rule: list.rel-induct) auto

lemma list-rel-imp-pred2:
  assumes
    list-all2 R xs ys and
     $\bigwedge x y. (x, y) \in \text{set}(\text{zip } xs \text{ } ys) \implies R x y \implies P y$ 
  shows list-all P ys
  using assms
  by (induction xs ys rule: list.rel-induct) auto

lemma eq-append-conv-conj:  $(zs = xs @ ys) = (xs = \text{take}(\text{length } xs) \text{ } zs \wedge ys = \text{drop}(\text{length } xs) \text{ } zs)$ 
  by (metis append-eq-conv-conj)

lemma list-all-list-updateI: list-all P xs  $\implies$  P x  $\implies$  list-all P (list-update xs n x)
  by (induction xs arbitrary: n) (auto simp add: nat.split-sels(2))

lemmas list-all2-update1-cong = list-all2-update-cong[of - - ys - ys ! i i for ys i,
  simplified]
lemmas list-all2-update2-cong = list-all2-update-cong[of - xs - xs ! i - i for xs i,
  simplified]

lemma map-list-update-id:
  f (xs ! pc) = f instr  $\implies$  map f (xs[pc := instr]) = map f xs
  using list-update-id map-update by metis

lemma list-all-eq-const-imp-replicate:
  assumes list-all ( $\lambda x. x = y$ ) xs
  shows xs = replicate (length xs) y
  using assms
  by (induction xs; simp)

lemma list-all-eq-const-imp-replicate':
  assumes list-all ((=) y) xs
  shows xs = replicate (length xs) y
  using assms
  by (induction xs; simp)

lemma list-all-eq-const-replicate-lhs[intro]:
  list-all ( $\lambda x. y = x$ ) (replicate n y)
  by (simp add: list-all-length)

```

```

lemma list-all-eq-const-replicate-rhs[intro]:
  list-all ( $\lambda x. x = y$ ) (replicate n y)
  by (simp add: list-all-length)

lemma list-all-eq-const-replicate[simp]: list-all ((=) c) (replicate n c)
  by (simp add: list-all-length)

lemma replicate-eq-map:
  assumes n = length xs and  $\bigwedge y. y \in \text{set } xs \implies f y = x$ 
  shows replicate n x = map f xs
  using assms
  proof (induction xs arbitrary: n)
    case Nil
    thus ?case by simp
  next
    case (Cons x xs)
    thus ?case by (cases n; auto)
  qed

lemma replicate-eq-impl-Ball-eq:
  shows replicate n c = xs  $\implies (\forall x \in \text{set } xs. x = c)$ 
  by (meson in-set-replicate)

lemma rel-option-map-of:
  assumes list-all2 (rel-prod (=) R) xs ys
  shows rel-option R (map-of xs l) (map-of ys l)
  using assms
  proof (induction xs ys rule: list.rel-induct)
    case Nil
    thus ?case by simp
  next
    case (Cons x xs y ys)
    from Cons.hyps have fst x = fst y and R (snd x) (snd y)
      by (simp-all add: rel-prod-sel)
    show ?case
    proof (cases l = fst y)
      case True
      then show ?thesis
        by (simp add: <fst x = fst y> <R (snd x) (snd y)>)
    next
      case False
      then show ?thesis
        using Cons.IH
        by (simp add: <fst x = fst y> )
    qed
  qed

lemma list-all2-rel-prod-nth:

```

```

assumes list-all2 (rel-prod R1 R2) xs ys and n < length xs
shows R1 (fst (xs ! n)) (fst (ys ! n))  $\wedge$  R2 (snd (xs ! n)) (snd (ys ! n))
using assms
proof (induction n arbitrary: xs ys)
  case 0
  then show ?case
    using assms(1,2)
    by (auto elim: list.rel-cases)
  next
    case (Suc n)
    then obtain x xs' y ys' where xs-def[simp]: xs = x # xs' and ys-def[simp]: ys
    = y # ys'
    by (auto elim: list.rel-cases)
    show ?case
      using Suc.pirms Suc.IH[of xs' ys']
      by force
  qed

lemma list-all2-rel-prod-fst-hd:
  assumes list-all2 (rel-prod R1 R2) xs ys and xs  $\neq [] \vee ys \neq []$ 
  shows R1 (fst (hd xs)) (fst (hd ys))  $\wedge$  R2 (snd (hd xs)) (snd (hd ys))
  using assms
  by (auto simp: rel-prod-sel elim: list.rel-cases)

lemma list-all2-rel-prod-fst-last:
  assumes list-all2 (rel-prod R1 R2) xs ys and xs  $\neq [] \vee ys \neq []$ 
  shows R1 (fst (last xs)) (fst (last ys))  $\wedge$  R2 (snd (last xs)) (snd (last ys))
  proof –
    have xs  $\neq []$  and ys  $\neq []$ 
    using assms by (auto elim: list.rel-cases)
    moreover have length xs = length ys
    by (rule assms(1)[THEN list-all2-lengthD])
    ultimately show ?thesis
    using list-all2-rel-prod-nth[OF assms(1)]
    by (simp add: last-conv-nth)
  qed

lemma list-all-nthD[intro]: list-all P xs  $\implies$  n < length xs  $\implies$  P (xs ! n)
  by (simp add: list-all-length)

lemma list-all P xs  $\implies$   $\forall x \in \text{set } xs. P x$ 
  using list-all-iff list.pred-set
  by (simp add: list-all-iff)

lemma list-all-map-of-SomeD:
  assumes list-all P kvs and map-of kvs k = Some v
  shows P (k, v)
  using assms

```

```

unfolding list.pred-set
by (auto dest: map-of-SomeD)

lemma list-all-not-nthD:list-all P xs  $\implies \neg P (xs ! n) \implies \text{length } xs \leq n$ 
proof (induction xs arbitrary: n)
  case Nil
  then show ?case
    by simp
  next
    case (Cons x xs)
    then show ?case
      by (cases n) simp-all
  qed

lemma list-all-butlast-not-nthD: list-all P (butlast xs)  $\implies \neg P (xs ! n) \implies \text{length } xs \leq \text{Suc } n$ 
  using list-all-not-nthD[of - butlast xs for xs, simplified]
  by (smt (z3) One-nat-def Suc-pred le-Suc-eq length-butlast length-greater-0-conv
less-Suc-eq list.pred-inject(1) list-all-not-nthD not-le nth-butlast)

lemma list-all-replicateI[intro]: P x  $\implies$  list-all P (replicate n x)
  unfolding list.pred-set
  by simp

lemma map-eq-append-replicate-conv:
  assumes map f xs = replicate n x @ ys
  shows map f (take n xs) = replicate n x
  using assms
  by (metis append-eq-conv-conj length-replicate take-map)

lemma map-eq-replicate-imp-list-all-const:
  map f xs = replicate n x  $\implies n = \text{length } xs \implies$  list-all ( $\lambda y. f y = x$ ) xs
  by (induction xs arbitrary: n) simp-all

lemma map-eq-replicateI: length xs = n  $\implies (\bigwedge x. x \in \text{set } xs \implies f x = c) \implies$ 
map f xs = replicate n c
by (induction xs arbitrary: n) auto

lemma list-all-dropI[intro]: list-all P xs  $\implies$  list-all P (drop n xs)
by (metis append-take-drop-id list-all-append)

```

5 Non-empty list

```

type-synonym 'a nlist = 'a × 'a list

end
theory Result
imports
  Main

```

HOL-Library.Monad-Syntax

begin

```
datatype ('err, 'a) result =
  is-err: Error 'err |
  is-ok: Ok 'a
```

6 Monadic bind

context begin

```
qualified fun bind :: ('a, 'b) result ⇒ ('b ⇒ ('a, 'c) result) ⇒ ('a, 'c) result where
  bind (Error x) - = Error x |
  bind (Ok x) f = f x
```

end

adhoc-overloading

```
bind ⇔ Result.bind
```

context begin

```
qualified lemma bind-Ok[simp]: x ≈ Ok = x
  by (cases x) simp-all
```

```
qualified lemma bind-eq-Ok-conv: (x ≈ f = Ok z) = (exists y. x = Ok y ∧ f y = Ok z)
  by (cases x) simp-all
```

```
qualified lemmas bind-eq-OkD[dest!] = bind-eq-Ok-conv[THEN iffD1]
qualified lemmas bind-eq-OkE = bind-eq-OkD[THEN exE]
qualified lemmas bind-eq-OkI[intro] = conjI[THEN exI[THEN bind-eq-Ok-conv[THEN iffD2]]]
```

qualified lemma bind-eq-Error-conv:

```
x ≈ f = Error z ↔ x = Error z ∨ (exists y. x = Ok y ∧ f y = Error z)
  by (cases x) simp-all
```

```
qualified lemmas bind-eq-ErrorD[dest!] = bind-eq-Error-conv[THEN iffD1]
qualified lemmas bind-eq-ErrorE = bind-eq-ErrorD[THEN disjE]
qualified lemmas bind-eq-ErrorI =
  disjI1[THEN bind-eq-Error-conv[THEN iffD2]]
  conjI[THEN exI[THEN disjI2[THEN bind-eq-Error-conv[THEN iffD2]]]]
```

lemma if-then-else-Ok[simp]:

```
(if a then b else Error c) = Ok d ↔ a ∧ b = Ok d
(if a then Error c else b) = Ok d ↔ ¬ a ∧ b = Ok d
  by (cases a) simp-all
```

```

qualified lemma if-then-else-Error[simp]:
  (if a then Ok b else c) = Error d  $\longleftrightarrow \neg a \wedge c = Error d$ 
  (if a then c else Ok b) = Error d  $\longleftrightarrow a \wedge c = Error d$ 
  by (cases a) simp-all

qualified lemma map-eq-Ok-conv: map-result f g x = Ok y  $\longleftrightarrow (\exists x'. x = Ok x'$ 
 $\wedge y = g x')$ 
  by (cases x; auto)

qualified lemma map-eq-Error-conv: map-result f g x = Error y  $\longleftrightarrow (\exists x'. x =$ 
 $Error x' \wedge y = f x')$ 
  by (cases x; auto)

qualified lemmas map-eq-OkD[dest!] = iffD1[OF map-eq-Ok-conv]
qualified lemmas map-eq-ErrorD[dest!] = iffD1[OF map-eq-Error-conv]

end

```

7 Conversion functions

```
context begin
```

```

qualified fun of-option where
  of-option e None = Error e |
  of-option e (Some x) = Ok x

qualified lemma of-option-injective[simp]: (of-option e x = of-option e y) = (x
= y)
  by (cases x; cases y; simp)

qualified lemma of-option-eq-Ok[simp]: (of-option x y = Ok z) = (y = Some z)
  by (cases y) simp-all

qualified fun to-option where
  to-option (Error -) = None |
  to-option (Ok x) = Some x

qualified lemma to-option-Some[simp]: (to-option r = Some x) = (r = Ok x)
  by (cases r) simp-all

qualified fun those :: ('err, 'ok) result list  $\Rightarrow$  ('err, 'ok list) result where
  those [] = Ok []
  those (x # xs) = do {
    y  $\leftarrow$  x;
    ys  $\leftarrow$  those xs;
    Ok (y # ys)
  }

```

```
qualified lemma those-Cons-OkD: those (x # xs) = Ok ys  $\Longrightarrow \exists y ys'. ys = y \#$ 
```

```

 $ys' \wedge x = Ok y \wedge \text{those } xs = Ok ys'$ 
  by auto

end

end
theory Option-Extra
  imports Main
begin

fun ap-option (infixl  $\diamond$  60) where
   $(Some f) \diamond (Some x) = Some (f x)$  |
   $- \diamond - = None$ 

lemma ap-option-eq-Some-conv:  $f \diamond x = Some y \longleftrightarrow (\exists f' x'. f = Some f' \wedge x = Some x' \wedge y = f' x')$ 
  by (cases f; simp; cases x; auto)

definition ap-map-prod where
   $ap\text{-map}\text{-prod } f g p \equiv Some \text{Pair} \diamond f (\text{fst } p) \diamond g (\text{snd } p)$ 

lemma ap-map-prod-eq-Some-conv:
   $ap\text{-map}\text{-prod } f g p = Some p' \longleftrightarrow (\exists x y. p = (x, y) \wedge (\exists x' y'. p' = (x', y') \wedge f x = Some x' \wedge g y = Some y'))$ 
  proof (cases p)
    case (Pair x y)
    then show ?thesis
      unfolding ap-map-prod-def
      by (auto simp add: ap-map-prod-def ap-option-eq-Some-conv)
  qed

fun ap-map-list :: ('a  $\Rightarrow$  'b option)  $\Rightarrow$  'a list  $\Rightarrow$  'b list option where
   $ap\text{-map}\text{-list } [] = Some []$  |
   $ap\text{-map}\text{-list } f (x \# xs) = Some (\#) \diamond f x \diamond ap\text{-map}\text{-list } f xs$ 

lemma length-ap-map-list:  $ap\text{-map}\text{-list } f xs = Some ys \implies \text{length } ys = \text{length } xs$ 
  by (induction xs arbitrary: ys) (auto simp add: ap-option-eq-Some-conv)

lemma ap-map-list-imp-rel-option-map-of:
  assumes  $ap\text{-map}\text{-list } f xs = Some ys$  and
     $\bigwedge x y. (x, y) \in \text{set } (\text{zip } xs ys) \implies f x = Some y \implies \text{fst } x = \text{fst } y$ 
  shows rel-option ( $\lambda x y. f (k, x) = Some (k, y)$ ) (map-of xs k) (map-of ys k)
  using assms
  proof (induction xs arbitrary: ys)
    case Nil
    thus ?case by simp
  next
    case (Cons x xs)
    from Cons.preds show ?case
  qed

```

```

apply (auto simp add: ap-option-eq-Some-conv option-rel-Some1 elim!: Cons.IH)
  apply (metis prod.collapse)
  apply (metis prod.collapse)
  by blast
qed

lemma ap-map-list-ap-map-prod-imp-rel-option-map-of:
assumes ap-map-list (ap-map-prod Some f) xs = Some ys
shows rel-option (λx y. f x = Some y) (map-of xs k) (map-of ys k)
using assms
proof (induction xs arbitrary: ys)
  case Nil
  thus ?case by simp
next
  case (Cons x xs)
  from Cons.prem show ?case
  by (auto simp: ap-option-eq-Some-conv ap-map-prod-eq-Some-conv elim: Cons.IH[simplified])
qed

lemma ex-ap-map-list-eq-SomeI:
assumes list-all (λx. ∃y. f x = Some y) xs
shows ∃ys. ap-map-list f xs = Some ys
using assms
by (induction xs) auto

lemma ap-map-list-iff-list-all2:
ap-map-list f xs = Some ys ↔ list-all2 (λx y. f x = Some y) xs ys
proof (rule iffI)
  show ap-map-list f xs = Some ys ⇒ list-all2 (λx y. f x = Some y) xs ys
    by (induction xs arbitrary: ys) (auto simp: ap-option-eq-Some-conv)
next
  show list-all2 (λx y. f x = Some y) xs ys ⇒ ap-map-list f xs = Some ys
    by (induction xs ys rule: list.rel-induct) auto
qed

lemma ap-map-list-map-conv:
assumes
  ap-map-list f xs = Some ys and
  ∀x y. x ∈ set xs ⇒ f x = Some y ⇒ y = g x
shows ys = map g xs
using assms
by (induction xs arbitrary: ys) (auto simp: ap-option-eq-Some-conv)

end
theory Map-Extra
imports Main HOL-Library.Library
begin

lemmas map-of-eq-Some-imp-key-in-fst-dom[intro] =

```

domI[of map-of xs for xs, unfolded dom-map-of-conv-image-fst]

lemma *very-weak-map-of-SomeI*: $k \in fst \set kvs \implies \exists v. map-of kvs k = Some v$
by (induction kvs) auto

lemma *map-of-fst-hd-neq-Nil*[simp]:
assumes $xs \neq []$
shows $map-of xs (fst (hd xs)) = Some (snd (hd xs))$
using assms
by (cases xs) simp-all

definition *map-merge* **where**
map-merge f m1 m2 x =
 $(case (m1 x, m2 x) of$
 $(None, None) \Rightarrow None$
 $| (None, Some z) \Rightarrow Some z$
 $| (Some y, None) \Rightarrow Some y$
 $| (Some y, Some z) \Rightarrow f y z)$

lemma *option-case-cancel*[simp]: $(case opt of None \Rightarrow x | Some - \Rightarrow x) = x$
using option.case-eq-if
by (simp add: option.case-eq-if)

lemma *map-le-map-merge-Some-const*:
 $f \subseteq_m map-merge (\lambda x y. Some x) f g$ **and**
 $g \subseteq_m map-merge (\lambda x y. Some y) f g$
unfolding map-le-def atomize-conj
find-theorems - &&& -
proof (intro conjI ballI)
fix x
assume $x \in dom f$
then obtain y **where** $f x = Some y$
by blast
then show $f x = map-merge (\lambda x y. Some x) f g x$
unfolding map-merge-def
by simp
next
fix x
assume $x \in dom g$
then obtain z **where** $g x = Some z$
by blast
then show $g x = map-merge (\lambda x. Some) f g x$
unfolding map-merge-def
by (simp add: option.case-eq-if)
qed

8 pred_map

definition *pred-map* **where**

```

pred-map P m ≡ (forall x y. m x = Some y —> P y)

lemma pred-map-get:
  assumes pred-map P m and m x = Some y
  shows P y
  using assms unfolding pred-map-def by simp

end
theory AList-Extra
  imports HOL-Library.AList List-util
begin

lemma list-all2-rel-prod-updateI:
  assumes list-all2 (rel-prod (=) R) xs ys and R xval yval
  shows list-all2 (rel-prod (=) R) (AList.update k xval xs) (AList.update k yval ys)
  using assms(1,1,2)
  by (induction xs ys rule: list.rel-induct) auto

lemma length-map-entry[simp]: length (AList.map-entry k f al) = length al
  by (induction al) simp-all

lemma map-entry-id0[simp]: AList.map-entry k id = id
proof (rule ext)
  fix xs
  show AList.map-entry k id xs = id xs
    by (induction xs) auto
qed

lemma map-entry-id: AList.map-entry k id xs = xs
  by simp

lemma map-entry-map-of-Some-conv:
  map-of xs k = Some v ==> AList.map-entry k f xs = AList.update k (f v) xs
  by (induction xs) auto

lemma map-entry-map-of-None-conv:
  map-of xs k = None ==> AList.map-entry k f xs = xs
  by (induction xs) auto

lemma list-all2-rel-prod-map-entry:
  assumes
    list-all2 (rel-prod (=) R) xs ys and
    (forall xval yval. map-of xs k = Some xval ==> map-of ys k = Some yval ==> R (f xval) (g yval))
  shows list-all2 (rel-prod (=) R) (AList.map-entry k f xs) (AList.map-entry k g ys)
  using assms(1)[THEN rel-option-map-of, of k]
proof (cases rule: option.rel-cases)
  case None

```

```

thus ?thesis
  using assms(1) by (simp add: map-entry-map-of-None-conv)
next
  case (Some xval yval)
  then show ?thesis
    using assms(1,2)
    by (auto simp add: map-entry-map-of-Some-conv intro!: list-all2-rel-prod-updateI)
qed

lemmas list-all2-rel-prod-map-entry1 = list-all2-rel-prod-map-entry[where g =
id, simplified]
lemmas list-all2-rel-prod-map-entry2 = list-all2-rel-prod-map-entry[where f = id,
simplified]

lemma list-all-updateI:
  assumes list-all P xs and P (k, v)
  shows list-all P (AList.update k v xs)
  using assms
  by (induction xs) auto

lemma set-update: set (AList.update k v xs) ⊆ {(k, v)} ∪ set xs
  by (induction xs) auto

end
theory Global
  imports HOL-Library.Library Result Env List-util Option-Extra Map-Extra AL-
ist-Extra
begin

sledgehammer-params [timeout = 30]
sledgehammer-params [provers = cvc5 e spass vampire z3 zipperposition]

declare K-record-comp[simp]

lemma if-then-Some-else-None-eq[simp]:
  (if a then Some b else None) = Some c ↔ a ∧ b = c
  (if a then Some b else None) = None ↔ ¬ a
  by (cases a) simp-all

lemma if-then-else-distributive: (if a then f b else f c) = f (if a then b else c)
  by simp

```

9 Rest

```

lemma map-ofD:
  fixes xs k opt
  assumes map-of xs k = opt
  shows opt = None ∨ (∃ n < length xs. opt = Some (snd (xs ! n)))
  using assms

```

```

by (metis in-set-conv-nth map-of-SomeD not-Some-eq snd-conv)

lemma list-all2-assoc-map-rel-get:
  assumes list-all2 (=) (map fst xs) (map fst ys) and list-all2 R (map snd xs)
  (map snd ys)
  shows rel-option R (map-of xs k) (map-of ys k)
  using assms[unfolded list.rel-map]
proof (induction xs ys rule: list.rel-induct)
  case Nil
  thus ?case by simp
next
  case (Cons x xs y ys)
  thus ?case by (cases k = fst x) auto
qed

```

9.1 Function definition

```

datatype ('label, 'instr) fundef =
  Fundef (body: ('label × 'instr list) list) (arity: nat) (return: nat) (fundef-locals: nat)

lemma rel-fundef-arithes: rel-fundef r1 r2 gd1 gd2 ==> arity gd1 = arity gd2
  by (simp add: fundef.relsel)

lemma rel-fundef-return: rel-fundef R1 R2 gd1 gd2 ==> return gd1 = return gd2
  by (simp add: fundef.relsel)

lemma rel-fundef-locals: rel-fundef R1 R2 gd1 gd2 ==> fundef-locals gd1 = fundef-locals gd2
  by (simp add: fundef.relsel)

lemma rel-fundef-body-length[simp]:
  rel-fundef r1 r2 fd1 fd2 ==> length (body fd1) = length (body fd2)
  by (auto intro: list-all2-lengthD simp add: fundef.relsel)

definition funtype where
  funtype fd ≡ (arity fd, return fd)

lemma rel-fundef-funtype[simp]: rel-fundef R1 R2 fd1 fd2 ==> funtype fd1 = funtype fd2
  by (simp add: fundef.relsel funtype-def)

lemma rel-fundef-rel-fst-hd-bodies:
  assumes rel-fundef R1 R2 fd1 fd2 and body fd1 ≠ [] ∨ body fd2 ≠ []
  shows R1 (fst (hd (body fd1))) (fst (hd (body fd2)))
  using assms
  unfolding fundef.relsel
  by (auto intro: list-all2-rel-prod-fst-hd[THEN conjunct1])

```

```

lemma map-option-comp-conv:
  assumes
     $\bigwedge x. \text{rel-option } R (f x) (g x)$ 
     $\bigwedge fd1 fd2. fd1 \in \text{ran } f \implies fd2 \in \text{ran } g \implies R fd1 fd2 \implies h fd1 = i fd2$ 
    shows map-option  $h \circ f = \text{map-option } i \circ g$ 
  proof (intro ext)
    fix  $x$ 
    show (map-option  $h \circ f$ )  $x = (\text{map-option } i \circ g)$   $x$ 
      using assms(1)[of  $x$ ]
      by (cases rule: option.rel-cases) (auto intro: ranI assms(2))
  qed

lemma map-option-arity-comp-conv:
  assumes ( $\bigwedge x. \text{rel-option} (\text{rel-fundef } R S) (f x) (g x)$ )
  shows map-option arity  $\circ f = \text{map-option arity} \circ g$ 
  proof (rule map-option-comp-conv)
    fix  $x$  show rel-option (rel-fundef  $R S$ )  $(f x) (g x)$ 
      by (rule assms(1))
  next
    fix  $fd1 fd2$ 
    show rel-fundef  $R S fd1 fd2 \implies \text{arity } fd1 = \text{arity } fd2$ 
      by (rule rel-fundef-arithes)
  qed

definition wf-fundef where
  wf-fundef  $fd \longleftrightarrow \text{body } fd \neq []$ 

lemma wf-fundef-non-empty-bodyD[dest,intro]: wf-fundef  $fd \implies \text{body } fd \neq []$ 
  by (simp add: wf-fundef-def)

definition wf-fundefs where
  wf-fundefs  $F \longleftrightarrow (\forall f fd. F f = \text{Some } fd \longrightarrow \text{wf-fundef } fd)$ 

lemma wf-fundefsI:
  assumes  $\bigwedge f fd. F f = \text{Some } fd \implies \text{wf-fundef } fd$ 
  shows wf-fundefs  $F$ 
  using assms by (simp add: wf-fundefs-def)

lemma wf-fundefsI':
  assumes  $\bigwedge f. \text{pred-option wf-fundef } (F f)$ 
  shows wf-fundefs  $F$ 
  using assms unfolding wf-fundefs-def
  by (metis option.pred-inject(2))

lemma wf-fundefs-imp-wf-fundef:
  assumes wf-fundefs  $F$  and  $F f = \text{Some } fd$ 
  shows wf-fundef  $fd$ 
  using assms by (simp add: wf-fundefs-def)

```

```
hide-fact wf-fundefs-def
```

9.2 Program

```
datatype ('fenv, 'henv, 'fun) prog =
  Prog (prog-fundefs: 'fenv) (heap: 'henv) (main-fun: 'fun)

definition wf-prog where
  wf-prog Get p  $\longleftrightarrow$  wf-fundefs (Get (prog-fundefs p))
```

9.3 Stack frame

```
datatype ('fun, 'label, 'operand) frame =
  Frame 'fun 'label (prog-counter: nat) (regs: 'operand list) (operand-stack: 'operand
list)

definition instr-at where
  instr-at fd label pc =
    (case map-of (body fd) label of
      Some instrs  $\Rightarrow$ 
        if pc < length instrs then
          Some (instrs ! pc)
        else
          None
      | None  $\Rightarrow$  None)

lemma instr-atD:
  assumes instr-at fd l pc = Some instr
  shows  $\exists$  instrs. map-of (body fd) l = Some instrs  $\wedge$  pc < length instrs  $\wedge$  instrs
! pc = instr
  using assms
  by (cases map-of (body fd) l) (auto simp: instr-at-def)

lemma rel-fundef-imp-rel-option-instr-at:
  assumes rel-fundef (=) R fd1 fd2
  shows rel-option R (instr-at fd1 l pc) (instr-at fd2 l pc)
  using assms
  proof (cases rule: fundef.rel-cases)
    case (Fundef bblocks1 - - bblocks2 )
    then show ?thesis
    by (auto simp: instr-at-def list-all2-lengthD
      intro: list-all2-nthD2 elim!: option.rel-cases dest!: rel-option-map-of[of - - -
l])
  qed

definition next-instr where
  next-instr F f label pc  $\equiv$  do {
    fd  $\leftarrow$  F f;
    instr-at fd label pc
  }
```

```

lemma next-instr-eq-Some-conv:
  next-instr F f l pc = Some instr  $\longleftrightarrow$  ( $\exists fd. F f = \text{Some } fd \wedge \text{instr-at } fd l pc = \text{Some } instr$ )
  by (simp add: next-instr-def bind-eq-Some-conv)

lemma next-instrD:
  assumes next-instr F f l pc = Some instr
  shows  $\exists fd. F f = \text{Some } fd \wedge \text{instr-at } fd l pc = \text{Some } instr$ 
  using assms unfolding next-instr-def
  by (cases F f; simp)

lemma next-instr-pc-lt-length-instrsI:
  assumes
    next-instr F f l pc = Some instr and
    F f = Some fd and
    map-of (body fd) l = Some instrs
  shows pc < length instrs
  using assms
  by (auto dest!: next-instrD instr-atD)

lemma next-instr-get-map-ofD:
  assumes
    next-instr F f l pc = Some instr and
    F f = Some fd and
    map-of (body fd) l = Some instrs
  shows pc < length instrs and instrs ! pc = instr
  using assms
  by (auto dest!: next-instrD instr-atD)

lemma next-instr-length-instrs:
  assumes
    F f = Some fd and
    map-of (body fd) label = Some instrs
  shows next-instr F f label (length instrs) = None
  by (simp add: assms next-instr-def instr-at-def)

lemma next-instr-take-Suc-conv:
  assumes next-instr F f l pc = Some instr and
    F f = Some fd and
    map-of (body fd) l = Some instrs
  shows take (Suc pc) instrs = take pc instrs @ [instr]
  using assms
  by (auto simp: take-Suc-conv-app-nth dest!: next-instrD instr-atD)

```

9.4 Dynamic state

```

datatype ('fenv, 'henv, 'frame) state =
  State (state-fundefs: 'fenv) (heap: 'henv) (callstack: 'frame list)

```

```

definition state-ok where
  state-ok Get s  $\longleftrightarrow$  wf-fundefs (Get (state-fundefs s))

inductive final for F-get Return where
  finalI: next-instr (F-get F) f l pc = Some Return  $\implies$ 
    final F-get Return (State F H [Frame f l pc R Σ])

definition allocate-frame where
  allocate-frame f fd args uninitialized  $\equiv$ 
    Frame f (fst (hd (body fd))) 0 (args @ replicate (fundef-locals fd) uninitialized)
  []

inductive load for F-get Uninitialized where
  F-get F main = Some fd  $\implies$  arity fd = 0  $\implies$  body fd  $\neq$  []  $\implies$ 
    load F-get Uninitialized (Prog F H main) (State F H [allocate-frame main fd [] Uninitialized])

lemma length-neq-imp-not-list-all2:
  assumes length xs  $\neq$  length ys
  shows  $\neg$  list-all2 R xs ys
  proof (rule notI)
    assume list-all2 R xs ys
    hence length xs = length ys
      by (rule list-all2-lengthD)
    thus False
      using assms by contradiction
  qed

lemma list-all2-rel-prod-conv:
  list-all2 (rel-prod R S) xs ys  $\longleftrightarrow$ 
    list-all2 R (map fst xs) (map fst ys)  $\wedge$  list-all2 S (map snd xs) (map snd ys)
  proof (cases length xs = length ys)
    case True
      then show ?thesis
        by (induction xs ys rule: list-induct2) (auto simp: rel-prodsel)
    next
      case False
        hence neq-length-maps:
          length (map fst xs)  $\neq$  length (map fst ys)
          length (map snd xs)  $\neq$  length (map snd ys)
          by simp-all
        show ?thesis
          using length-neq-imp-not-list-all2[OF False]
          using neq-length-maps[THEN length-neq-imp-not-list-all2]
          by simp
  qed

definition rewrite-fundef-body ::
```

```

('label, 'instr) fundef  $\Rightarrow$  'label  $\Rightarrow$  nat  $\Rightarrow$  'instr  $\Rightarrow$  ('label, 'instr) fundef where
rewrite-fundef-body fd l n instr =
  (case fd of Fundef bblocks ar ret locals  $\Rightarrow$ 
    Fundef (AList.map-entry l (λinstrs. instrs[n := instr])) bblocks) ar ret locals)

lemma rewrite-fundef-body-cases[case-names invalid-label valid-label]:
assumes
   $\wedge_{bs \text{ ar ret locals}} fd = \text{Fundef } bs \text{ ar ret locals} \implies \text{map-of } bs \text{ } l = \text{None} \implies P$ 
   $\wedge_{bs \text{ ar ret locals}} instrs. fd = \text{Fundef } bs \text{ ar ret locals} \implies \text{map-of } bs \text{ } l = \text{Some } instrs \implies P$ 
shows P
using assms
by (cases fd; cases map-of (body fd) l) auto

lemma arity-rewrite-fundef-body[simp]: arity (rewrite-fundef-body fd l pc instr) =
arity fd
by (cases fd; simp add: rewrite-fundef-body-def option.case-eq-if)

lemma return-rewrite-fundef-body[simp]: return (rewrite-fundef-body fd l pc instr) =
return fd
by (cases fd) (simp add: rewrite-fundef-body-def option.case-eq-if)

lemma funtype-rewrite-fundef-body[simp]: funtype (rewrite-fundef-body fd l pc instr') =
funtype fd
by (simp add: funtype-def)

lemma length-body-rewrite-fundef-body[simp]:
length (body (rewrite-fundef-body fd l pc instr)) = length (body fd)
by (cases fd) (simp add: rewrite-fundef-body-def)

lemma prod-in-set-fst-image-conv: (x, y)  $\in$  set xys  $\implies$  x  $\in$  fst ` set xys
by (metis fstI image-eqI)

lemma map-of-body-rewrite-fundef-body-conv-neq[simp]:
assumes l  $\neq$  l'
shows map-of (body (rewrite-fundef-body fd l pc instr)) l' = map-of (body fd) l'
using assms
by (cases fd) (simp add: rewrite-fundef-body-def map-of-map-entry)

lemma map-of-body-rewrite-fundef-body-conv-eq[simp]:
map-of (body (rewrite-fundef-body fd l pc instr)) l =
map-option (λxs. xs[pc := instr]) (map-of (body fd) l)
by (cases fd) (simp add: rewrite-fundef-body-def map-of-map-entry map-option-case)

lemma instr-at-rewrite-fundef-body-conv:
instr-at (rewrite-fundef-body fd l' pc' instr') l pc =
map-option (λinstr. if l = l'  $\wedge$  pc = pc' then instr' else instr) (instr-at fd l pc)
proof (cases fd)
case (Fundef bblocks ar ret locals)

```

```

then show ?thesis
proof (cases instr-at fd l pc)
  case None
  then show ?thesis
    by (cases map-of bblocks l)
    (auto simp add: Fundef rewrite-fundef-body-def instr-at-def map-of-map-entry)
next
  case (Some instrs)
  then show ?thesis
    by (cases map-of bblocks l)
    (auto simp add: Fundef rewrite-fundef-body-def instr-at-def map-of-map-entry)
qed
qed

lemma fundef-locals-rewrite-fundef-body[simp]:
  fundef-locals (rewrite-fundef-body fd l pc instr) = fundef-locals fd
  by (cases fd; simp add: rewrite-fundef-body-def option.case-eq-if)

lemma rel-fundef-rewrite-body1:
assumes
  rel-fd1-fd2: rel-fundef (=) R fd1 fd2 and
  instr-at-l-pc: instr-at fd1 l pc = Some instr and
  R-iff:  $\bigwedge x. R \text{ instr } x \longleftrightarrow R \text{ instr}' x$ 
shows rel-fundef (=) R (rewrite-fundef-body fd1 l pc instr') fd2
using assms(1)
proof (cases rule: fundef.rel-cases)
  case (Fundef xs ar' ret' locals' ys ar ret locals)
  show ?thesis
    using Fundef(3,3,1,2,4-) instr-at-l-pc
  proof (induction xs ys arbitrary: fd1 fd2 rule: list.rel-induct)
    case Nil
    then show ?case by (simp add: rewrite-fundef-body-def)
next
  case (Cons x xs y ys)
  then show ?case
    apply (simp add: rewrite-fundef-body-def)
    apply safe
    using assms(3)
    apply (smt (verit, ccfv-SIG) fun-upd-apply fundef.sel(1) instr-atD list-all2-lengthD
      list-all2-nthD2 list-all2-update1-cong map-of.simps(2) option.simps(1)
      prod.sel(1,2)
      rel-prodsel)
    by (simp add: instr-at-def)
qed
qed

lemma rel-fundef-rewrite-body:
assumes rel-fd1-fd2: rel-fundef (=) R fd1 fd2 and R-i1-i2: R i1 i2
shows rel-fundef (=) R (rewrite-fundef-body fd1 l pc i1) (rewrite-fundef-body fd2

```

```

 $l \ pc \ i2)$ 
  using assms(1)
proof (cases rule: fundef.rel-cases)
  case (Fundef bblocks1 ar1 ret1 locals1 bblocks2 ar2 ret2 locals2)
  then show ?thesis
    using R-i1-i2
    by (auto simp: rewrite-fundef-body-def
      intro!: list-all2-update-cong
      intro!: list-all2-rel-prod-map-entry
      dest: rel-option-map-of[of --- l])
qed

lemma rewrite-fundef-body-triv:
  instr-at fd l pc = Some instr  $\implies$  rewrite-fundef-body fd l pc instr = fd
  by (cases fd)
    (auto simp add: rewrite-fundef-body-def map-entry-map-of-Some-conv update-triv
    dest: instr-atD)

lemma rel-fundef-rewrite-body2:
  assumes
    rel-fd1-fd2: rel-fundef (=) R fd1 fd2 and
    instr-at-l-pc: instr-at fd2 l pc = Some instr and
    R-iff:  $\bigwedge x. R x \text{ instr} \longleftrightarrow R x \text{ instr}'$ 
  shows rel-fundef (=) R fd1 (rewrite-fundef-body fd2 l pc instr')
  using assms(1)
proof (cases rule: fundef.rel-cases)
  case (Fundef xs ar' ret' locals' ys ar ret locals)
  moreover obtain instrs2 where
    map-of (body fd2) l = Some instrs2 and pc < length instrs2 and instrs2 ! pc
    = instr
    using instr-atD[OF instr-at-l-pc] by auto
  moreover then obtain instrs1 instr1 where
    map-of (body fd1) l = Some instrs1 and pc < length instrs1 and instrs1 ! pc
    = instr1
    using rel-fundef-imp-rel-option-instr-at[OF rel-fd1-fd2, of l pc]
    apply (auto simp add: instr-at-def option-rel-Some2)
    by (metis instr-atD instr-at-def)
  moreover have list-all2 R instrs1 instrs2
  using Fundef <map-of (body fd2) l = Some instrs2> <map-of (body fd1) l =
  Some instrs1>
  by (auto dest: rel-option-map-of[of --- l])
  moreover hence R (instrs1 ! pc) (instrs2 ! pc)
  using <pc < length instrs1>
  by (auto intro: list-all2-nthD)
  ultimately show ?thesis
  by (auto simp: rewrite-fundef-body-def R-iff
    intro!: list-all2-rel-prod-map-entry2
    intro: list-all2-update2-cong)
qed

```

```

lemma rel-fundef-rewrite-body2':
  assumes
    rel-fd1-fd2: rel-fundef (=) R fd1 fd2 and
    instr-at1: instr-at fd1 l pc = Some instr1 and
    R-instr1-instr2: R instr1 instr2'
  shows rel-fundef (=) R fd1 (rewrite-fundef-body fd2 l pc instr2')
  using assms(1)
  proof (cases rule: fundef.rel-cases)
    case (Fundef bblocks1 ar1 ret1 locals1 bblocks2 ar2 ret2 locals2)
    moreover obtain instrs1 where
      map-of bblocks1 l = Some instrs1 and pc < length instrs1 and instrs1 ! pc =
      instr1
      using Fundef instr-at1[THEN instr-atD] by auto

    moreover then obtain instrs2 instr2 where
      map-of (body fd2) l = Some instrs2 and pc < length instrs2 and instrs2 ! pc =
      instr2
      using Fundef rel-fd1-fd2
      apply (auto simp: option-rel-Some1 dest!: rel-option-map-of[where l = l])
      by (metis list-all2-lengthD)
    ultimately show ?thesis
    using R-instr1-instr2
    by (auto simp: rewrite-fundef-body-def
      intro!: list-all2-update2-cong list-all2-rel-prod-map-entry2
      dest: rel-option-map-of[of --- l])
  qed

thm rel-fundef-rewrite-body

lemma if-eq-const-conv: (if x then y else z) = w  $\longleftrightarrow$  x  $\wedge$  y = w  $\vee$   $\neg$  x  $\wedge$  z = w
  by simp

lemma const-eq-if-conv: w = (if x then y else z)  $\longleftrightarrow$  x  $\wedge$  w = y  $\vee$   $\neg$  x  $\wedge$  w = z
  by auto

end
theory Op
  imports Main
begin

```

10 n-ary operations

```

locale nary-operations =
  fixes
    Op :: 'op  $\Rightarrow$  'a list  $\Rightarrow$  'a and
    Arity :: 'op  $\Rightarrow$  nat
  assumes
    Op-Arity-domain: length xs = Arity op  $\Longrightarrow$   $\exists$  y. Op op xs = y

```

```

end
theory OpInl
  imports Op
begin

11 n-ary operations

locale nary-operations-inl =
  nary-operations Op Arith
  for
    Op :: 'op ⇒ 'a list ⇒ 'a and Arith +
  fixes
    InlOp :: 'opinl ⇒ 'a list ⇒ 'a and
    Inl :: 'op ⇒ 'a list ⇒ 'opinl option and
    IsInl :: 'opinl ⇒ 'a list ⇒ bool and
    DeInl :: 'opinl ⇒ 'op
  assumes
    Inl-invertible: Inl op xs = Some opinl ⇒ DeInl opinl = op and
    InlOp-correct: length xs = Arith (DeInl opinl) ⇒ InlOp opinl xs = Op (DeInl
      opinl) xs and
    Inl-IsInl: Inl op xs = Some opinl ⇒ IsInl opinl xs

begin

lemma Inl-inj-on: inj-on Inl { op | op args. Inl op args ≠ None }
  apply (simp add: inj-on-def)
proof (intro allI impI, elim exE)
  fix op1 op2 args1 args2 opinl1 opinl2
  assume assms: Inl op1 = Inl op2 Inl op1 args1 = Some opinl1 Inl op2 args2 =
    Some opinl2
  hence DeInl opinl1 = op1 DeInl opinl1 = op2
    by (auto intro: Inl-invertible)
  thus op1 = op2
    by simp
qed

abbreviation Inl-dom where
  Inl-dom ≡ { op | op args. Inl op args ≠ None }

lemma bij-betw Inl Inl-dom { Inl op | op. op ∈ Inl-dom}
  using bij-betw-def
  unfolding image-def
  using Inl-inj-on
  by (auto simp add: bij-betw-def)

```

```

end

end
theory Dynamic
imports Main
begin

locale dynval =
fixes
  uninitialized :: 'dyn and
  is-true :: 'dyn ⇒ bool and
  is-false :: 'dyn ⇒ bool
assumes
  not-true-and-false: ¬ (is-true d ∧ is-false d)
begin

lemma false-not-true: is-false d ⟹ ¬ is-true d
  using not-true-and-false by blast

lemma true-not-false: is-true d ⟹ ¬ is-false d
  using not-true-and-false by blast

lemma is-true-and-is-false-implies-False: is-true d ⟹ is-false d ⟹ False
  using true-not-false false-not-true by simp

end

end
theory Inca
imports Global OpInl Env Dynamic
  VeriComp.Language
begin

```

12 Inline caching

12.1 Static representation

```

datatype ('dyn, 'var, 'fun, 'label, 'op, 'opinl) instr =
  IPush 'dyn |
  IPop |
  IGet nat |
  ISet nat |
  ILoad 'var |
  IStore 'var |
  IOp 'op |
  IOpInl 'opinl |
  is-jump: ICJump 'label 'label |
  ICcall 'fun |

```

```

is-return: IReturn

locale inca =
Fenv: env F-empty F-get F-add F-to-list +
Henv: env heap-empty heap-get heap-add heap-to-list +
dynval uninitialized is-true is-false +
nary-operations-inl Op Arith InlOp Inl IsInl DeInl
for
  — Functions environment
  F-empty and
  F-get :: 'fenv => 'fun => ('label, ('dyn, 'var, 'fun, 'label, 'op, 'opinl) instr) fundef
option and
  F-add and F-to-list and

  — Memory heap
  heap-empty and
  heap-get :: 'henv => 'var × 'dyn => 'dyn option and
  heap-add and heap-to-list and

  — Dynamic values
  uninitialized :: 'dyn and is-true and is-false and

  — n-ary operations
  Op :: 'op => 'dyn list => 'dyn and Arith and
  InlOp and Inl and IsInl and DeInl :: 'opinl => 'op
begin

```

12.2 Semantics

```

inductive step (infix <-> 55) where
  step-push:
    next-instr (F-get F) f l pc = Some (IPush d) ==>
      State F H (Frame f l pc R Σ # st) → State F H (Frame f l (Suc pc) R (d # Σ) # st) |
  step-pop:
    next-instr (F-get F) f l pc = Some IPop ==>
      State F H (Frame f l pc R (d # Σ) # st) → State F H (Frame f l (Suc pc) R Σ # st) |
  step-get:
    next-instr (F-get F) f l pc = Some (IGet n) ==>
      n < length R ==> d = R ! n ==>
      State F H (Frame f l pc R Σ # st) → State F H (Frame f l (Suc pc) R (d # Σ) # st) |
  step-set:
    next-instr (F-get F) f l pc = Some (ISet n) ==>
      n < length R ==> R' = R[n := d] ==>
```

$\text{State } F H (\text{Frame } f l pc R (d \# \Sigma) \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R' \Sigma \# st) |$

step-load:

$\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (ILoad x) \implies$

$\text{heap-get } H (x, y) = \text{Some } d \implies$

$\text{State } F H (\text{Frame } f l pc R (y \# \Sigma) \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (d \# \Sigma) \# st) |$

step-store:

$\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (IStore x) \implies$

$\text{heap-add } H (x, y) d = H' \implies$

$\text{State } F H (\text{Frame } f l pc R (y \# d \# \Sigma) \# st) \rightarrow \text{State } F H' (\text{Frame } f l (\text{Suc } pc) R \Sigma \# st) |$

step-op:

$\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (IOp op) \implies$

$\text{Arith } op = ar \implies ar \leq \text{length } \Sigma \implies \text{Inl } op (\text{take } ar \Sigma) = \text{None} \implies$

$\text{Op } op (\text{take } ar \Sigma) = x \implies$

$\text{State } F H (\text{Frame } f l pc R \Sigma \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (x \# drop ar \Sigma) \# st) |$

step-op-inl:

$\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (IOp op) \implies$

$\text{Arith } op = ar \implies ar \leq \text{length } \Sigma \implies \text{Inl } op (\text{take } ar \Sigma) = \text{Some } opinl \implies$

$\text{InlOp } opinl (\text{take } ar \Sigma) = x \implies$

$F' = \text{Fenv.map-entry } F f (\lambda fd. \text{rewrite-fundef-body } fd l pc (IOpInl opinl)) \implies$

$\text{State } F H (\text{Frame } f l pc R \Sigma \# st) \rightarrow \text{State } F' H (\text{Frame } f l (\text{Suc } pc) R (x \# drop ar \Sigma) \# st) |$

step-op-inl-hit:

$\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (IOpInl opinl) \implies$

$\text{Arith } (\text{DefInl } opinl) = ar \implies ar \leq \text{length } \Sigma \implies \text{IsInl } opinl (\text{take } ar \Sigma) \implies$

$\text{InlOp } opinl (\text{take } ar \Sigma) = x \implies$

$\text{State } F H (\text{Frame } f l pc R \Sigma \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (x \# drop ar \Sigma) \# st) |$

step-op-inl-miss:

$\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (IOpInl opinl) \implies$

$\text{Arith } (\text{DefInl } opinl) = ar \implies ar \leq \text{length } \Sigma \implies \neg \text{IsInl } opinl (\text{take } ar \Sigma) \implies$

$\text{InlOp } opinl (\text{take } ar \Sigma) = x \implies$

$F' = \text{Fenv.map-entry } F f (\lambda fd. \text{rewrite-fundef-body } fd l pc (IOp (\text{DefInl } opinl))) \implies$

$\text{State } F H (\text{Frame } f l pc R \Sigma \# st) \rightarrow \text{State } F' H (\text{Frame } f l (\text{Suc } pc) R (x \# drop ar \Sigma) \# st) |$

step-cjump:

$\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (ICJump } l_t l_f) \implies$

$\text{is-true } d \wedge l' = l_t \vee \text{is-false } d \wedge l' = l_f \implies$

```

State F H (Frame f l pc R (d # Σ) # st) → State F H (Frame f l' 0 R Σ #
st) | 

step-call:
next-instr (F-get F) f l pc = Some (ICall g) ⇒
F-get F g = Some gd ⇒ arity gd ≤ length Σ ⇒
frameg = allocate-frame g gd (take (arity gd) Σ) uninitialized ⇒
State F H (Frame f l pc R Σ # st) → State F H (frameg # Frame f l pc R Σ
# st) | 

step-return:
next-instr (F-get F) g lg pcg = Some IReturn ⇒
F-get F g = Some gd ⇒ arity gd ≤ length Σf ⇒
length Σg = return gd ⇒
framef' = Frame f lf (Suc pcf) Rf (Σg @ drop (arity gd) Σf) ⇒
State F H (Frame g lg pcg Rg Σg # Frame f lf pcf Rf Σf # st) → State F H
(framef' # st)

lemma step-deterministic:
assumes s1 → s2 and s1 → s3
shows s2 = s3
using assms
apply (cases s1 s2 rule: step.cases)
apply (auto elim: step.cases dest: is-true-and-is-false-implies-False) [3]
done

lemma step-right-unique: right-unique step
using step-deterministic
by (rule right-uniqueI)

lemma final-finished:
assumes final F-get IReturn s
shows finished step s
unfolding finished-def
proof (rule notI)
assume ∃x. step s x
then obtain s' where step s s' by auto
thus False
using ⟨final F-get IReturn s⟩
by (auto simp: state-ok-def elim!: step.cases final.cases)
qed

sublocale semantics step final F-get IReturn
using final-finished step-deterministic
by unfold-locales

```

```

definition load where
  load ≡ Global.load F-get uninitialized

sublocale language step final F-get IReturn load
  by unfold-locales

end

end
theory Unboxed
  imports Global Dynamic
begin

  datatype type = Ubx1 | Ubx2

  datatype ('dyn, 'ubx1, 'ubx2) unboxed =
    is-dyn-operand: OpDyn 'dyn |
    OpUbx1 'ubx1 |
    OpUbx2 'ubx2

  fun typeof where
    typeof (OpDyn -) = None |
    typeof (OpUbx1 -) = Some Ubx1 |
    typeof (OpUbx2 -) = Some Ubx2

  fun cast-Dyn where
    cast-Dyn (OpDyn d) = Some d |
    cast-Dyn - = None

  fun cast-Ubx1 where
    cast-Ubx1 (OpUbx1 x) = Some x |
    cast-Ubx1 - = None

  fun cast-Ubx2 where
    cast-Ubx2 (OpUbx2 x) = Some x |
    cast-Ubx2 - = None

locale unboxedval = dynval uninitialized is-true is-false
  for uninitialized :: 'dyn and is-true and is-false +
  fixes
    box-ubx1 :: 'ubx1 ⇒ 'dyn and unbox-ubx1 :: 'dyn ⇒ 'ubx1 option and
    box-ubx2 :: 'ubx2 ⇒ 'dyn and unbox-ubx2 :: 'dyn ⇒ 'ubx2 option
  assumes
    box-unbox-inverse:
      unbox-ubx1 d = Some u1 ⇒ box-ubx1 u1 = d
      unbox-ubx2 d = Some u2 ⇒ box-ubx2 u2 = d
begin

```

```

fun unbox :: type  $\Rightarrow$  'dyn  $\Rightarrow$  ('dyn, 'ubx1, 'ubx2) unboxed option where
  unbox Ubx1 = map-option OpUbx1  $\circ$  unbox-ubx1 |
  unbox Ubx2 = map-option OpUbx2  $\circ$  unbox-ubx2

fun cast-and-box where
  cast-and-box Ubx1 = map-option box-ubx1  $\circ$  cast-Ubx1 |
  cast-and-box Ubx2 = map-option box-ubx2  $\circ$  cast-Ubx2

fun norm-unboxed where
  norm-unboxed (OpDyn d) = d |
  norm-unboxed (OpUbx1 x) = box-ubx1 x |
  norm-unboxed (OpUbx2 x) = box-ubx2 x

fun box-operand where
  box-operand (OpDyn d) = OpDyn d |
  box-operand (OpUbx1 x) = OpDyn (box-ubx1 x) |
  box-operand (OpUbx2 x) = OpDyn (box-ubx2 x)

fun box-frame where
  box-frame f (Frame g l pc R  $\Sigma$ ) = Frame g l pc R (iff f = g then map box-operand
 $\Sigma$  else  $\Sigma$ )

definition box-stack where
  box-stack f  $\equiv$  map (box-frame f)

end

end
theory OpUbx
  imports OpInl Unboxed
begin

```

13 n-ary operations

```

locale nary-operations-ubx =
  nary-operations-inl  $\mathfrak{Op}$  Arity InlOp Inl IsInl DeInl +
  unboxedval unitIALIZED is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
  for
     $\mathfrak{Op}$  :: 'op  $\Rightarrow$  'dyn list  $\Rightarrow$  'dyn and Arity and
    InlOp and Inl and IsInl and DeInl :: 'opinl  $\Rightarrow$  'op and
    unitIALIZED :: 'dyn and is-true and is-false and
    box-ubx1 :: 'ubx1  $\Rightarrow$  'dyn and unbox-ubx1 and
    box-ubx2 :: 'ubx2  $\Rightarrow$  'dyn and unbox-ubx2 +
  fixes
     $\mathfrak{UbxOp}$  :: 'opubx  $\Rightarrow$  ('dyn, 'ubx1, 'ubx2) unboxed list  $\Rightarrow$  ('dyn, 'ubx1, 'ubx2)
  unboxed option and
     $\mathfrak{Ubx}$  :: 'opinl  $\Rightarrow$  type option list  $\Rightarrow$  'opubx option and
     $\mathfrak{Box}$  :: 'opubx  $\Rightarrow$  'opinl and
     $\mathfrak{TypeOp}$  :: 'opubx  $\Rightarrow$  type option list  $\times$  type option

```

```

assumes
   $\text{Ubx-invertible}:$ 
     $\text{Ubx } opinl \ ts = Some \ opubx \implies \text{Bor } opubx = opinl \ \text{and}$ 
   $\text{UbxOp-correct}:$ 
     $\text{UbxOp } opubx \ \Sigma = Some \ z \implies \text{InlOp} (\text{Bor } opubx) \ (\text{map norm-unboxed } \Sigma) =$ 
     $\text{norm-unboxed } z \ \text{and}$ 
   $\text{UbxOp-to-Inl}:$ 
     $\text{UbxOp } opubx \ \Sigma = Some \ z \implies \text{Inl} (\text{DeInl} (\text{Bor } opubx)) \ (\text{map norm-unboxed } \Sigma) = Some (\text{Bor } opubx) \ \text{and}$ 

 $\text{TypeOfOp-Arith}:$ 
   $\text{Arith} (\text{DeInl} (\text{Bor } opubx)) = length (fst (\text{TypeOfOp } opubx)) \ \text{and}$ 
 $\text{Ubx-opubx-type}:$ 
   $\text{Ubx } opinl \ ts = Some \ opubx \implies fst (\text{TypeOfOp } opubx) = ts \ \text{and}$ 

 $\text{TypeOfOp-correct}:$ 
   $\text{TypeOfOp } opubx = (\text{map typeof } xs, \tau) \implies$ 
   $\exists y. \text{UbxOp } opubx \ xs = Some \ y \wedge \text{typeof } y = \tau \ \text{and}$ 
 $\text{TypeOfOp-complete}:$ 
   $\text{UbxOp } opubx \ xs = Some \ y \implies \text{TypeOfOp } opubx = (\text{map typeof } xs, \text{typeof } y)$ 

end
theory Ubx
imports Global OpUbx Env
VeriComp.Language
begin

```

14 Unboxed caching

14.1 Static representation

```

datatype ('dyn, 'var, 'fun, 'label, 'op, 'opinl, 'opubx, 'num, 'bool) instr =
  IPush 'dyn | IPushUbx1 'num | IPushUbx2 'bool |
  IPop |
  IGet nat | IGetUbx type nat |
  ISet nat | ISetUbx type nat |
  ILoad 'var | ILoadUbx type 'var |
  IStore 'var | IStoreUbx type 'var |
  IOp 'op | IOpInl 'opinl | IOpUbx 'opubx |
  is-jump: ICJump 'label 'label |
  is-fun-call: ICall 'fun |
  is-return: IReturn

locale ubx =
  Fenv: env F-empty F-get F-add F-to-list +
  Henv: env heap-empty heap-get heap-add heap-to-list +
  nary-operations-ubx
  Op Arith
  InlOp Inl IsInl DeInl

```

```

uninitialized is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
AbxOp Abx Box TypeOpOp
for
    — Functions environment
    F-empty and
    F-get :: 'fenv  $\Rightarrow$  'fun  $\Rightarrow$  ('label, ('dyn, 'var, 'fun, 'label, 'op, 'opinl, 'opubx,  

'num, 'bool) instr) fundef option and
    F-add and F-to-list and

    — Memory heap
    heap-empty and
    heap-get :: 'henv  $\Rightarrow$  'var  $\times$  'dyn  $\Rightarrow$  'dyn option and
    heap-add and heap-to-list and

    — Dynamic values
    uninitialized :: 'dyn and is-true and is-false and

    — Unboxed values
    box-ubx1 and unbox-ubx1 and
    box-ubx2 and unbox-ubx2 and

    — n-ary operations
    Op :: 'op  $\Rightarrow$  'dyn list  $\Rightarrow$  'dyn and Arity and
    InOp and Inl and IsInl and DeInl :: 'opinl  $\Rightarrow$  'op and
    AbxOp :: 'opubx  $\Rightarrow$  ('dyn, 'num, 'bool) unboxed list  $\Rightarrow$  ('dyn, 'num, 'bool) unboxed  

option and
    Abx :: 'opinl  $\Rightarrow$  type option list  $\Rightarrow$  'opubx option and
    Box :: 'opubx  $\Rightarrow$  'opinl and
    TypeOpOp
begin

```

```

lemmas map-option-funtype-comp-map-entry-rewrite-fundef-body[simp] =
Fenv.map-option-comp-map-entry[of - funtype  $\lambda fd.$  rewrite-fundef-body  $fd l pc$ 
instr for l pc instr, simplified]

```

```

fun generalize-instr where
    generalize-instr (IPushUbx1 n) = IPush (box-ubx1 n) |
    generalize-instr (IPushUbx2 b) = IPush (box-ubx2 b) |
    generalize-instr (IGetUbx - n) = IGet n |
    generalize-instr (ISetUbx - n) = ISet n |
    generalize-instr (ILoadUbx - x) = ILoad x |
    generalize-instr (IStoreUbx - x) = IStore x |
    generalize-instr (IOpUbx opubx) = IOpInl (Box opubx) |
    generalize-instr instr = instr

```

```

lemma instrsel-generalize-instr[simp]:
    is-jump (generalize-instr instr)  $\longleftrightarrow$  is-jump instr
    is-fun-call (generalize-instr instr)  $\longleftrightarrow$  is-fun-call instr

```

```

is-return (generalize-instr instr)  $\longleftrightarrow$  is-return instr
unfolding atomize-conj
by (cases instr; simp)

lemma instrsel-generalize-instr-comp[simp]:
  is-jump  $\circ$  generalize-instr = is-jump and
  is-fun-call  $\circ$  generalize-instr = is-fun-call and
  is-return  $\circ$  generalize-instr = is-return
unfolding atomize-conj
using instrsel-generalize-instr by auto

fun generalize-fundef where
  generalize-fundef (FunDef xs ar ret locals) =
    FunDef (map-ran ( $\lambda$ . map generalize-instr) xs) ar ret locals

lemma generalize-instr-idempotent[simp]:
  generalize-instr (generalize-instr x) = generalize-instr x
by (cases x) simp-all

lemma generalize-instr-idempotent-comp[simp]:
  generalize-instr  $\circ$  generalize-instr = generalize-instr
by fastforce

lemma length-body-generalize-fundef[simp]: length (body (generalize-fundef fd)) =
length (body fd)
by (cases fd) (simp add: map-ran-def)

lemma arity-generalize-fundef[simp]: arity (generalize-fundef fd) = arity fd
by (cases fd) simp

lemma return-generalize-fundef[simp]: return (generalize-fundef fd) = return fd
by (cases fd) simp

lemma fundef-locals-generalize[simp]: fundef-locals (generalize-fundef fd) = fun-
def-locals fd
by (cases fd; simp)

lemma funtype-generalize-fundef[simp]: funtype (generalize-fundef fd) = funtype
fd
by (cases fd; simp add: funtype-def)

lemmas map-option-comp-map-entry-generalize-fundef[simp] =
Fenv.map-option-comp-map-entry[of - funtype generalize-fundef, simplified]

lemma image-fst-set-body-generalize-fundef[simp]:
  fst ‘ set (body (generalize-fundef fd)) = fst ‘ set (body fd)
by (cases fd) (simp add: dom-map-ran)

lemma map-of-generalize-fundef-conv:

```

```

map-of (body (generalize-fundef fd)) l = map-option (map generalize-instr) (map-of
(body fd) l)
by (cases fd) (simp add: map-ran-conv)

lemma map-option-arity-get-map-entry-generalize-fundef[simp]:
map-option arity o F-get (Fenv.map-entry F2 f generalize-fundef) =
map-option arity o F-get F2
by (auto intro: Fenv.map-option-comp-map-entry)

lemma instr-at-generalize-fundef-conv:
instr-at (generalize-fundef fd) l = map-option generalize-instr o instr-at fd l
proof (intro ext)
fix n
show instr-at (generalize-fundef fd) l n = (map-option generalize-instr o instr-at
fd l) n
proof (cases fd)
case (Fundef bblocks ar locals)
then show ?thesis
by (cases map-of bblocks l) (simp-all add: instr-at-def map-ran-conv)
qed
qed

```

14.2 Semantics

```

inductive step (infix  $\leftrightarrow$  55) where
step-push:
next-instr (F-get F) f l pc = Some (IPush d)  $\implies$ 
State F H (Frame f l pc R  $\Sigma$  # st)  $\rightarrow$  State F H (Frame f l (Suc pc) R (OpDyn
d #  $\Sigma$ ) # st) |

step-push-ubx1:
next-instr (F-get F) f l pc = Some (IPushUbx1 n)  $\implies$ 
State F H (Frame f l pc R  $\Sigma$  # st)  $\rightarrow$  State F H (Frame f l (Suc pc) R (OpUbx1
n #  $\Sigma$ ) # st) |

step-push-ubx2:
next-instr (F-get F) f l pc = Some (IPushUbx2 b)  $\implies$ 
State F H (Frame f l pc R  $\Sigma$  # st)  $\rightarrow$  State F H (Frame f l (Suc pc) R (OpUbx2
b #  $\Sigma$ ) # st) |

step-pop:
next-instr (F-get F) f l pc = Some IPop  $\implies$ 
State F H (Frame f l pc R (x #  $\Sigma$ ) # st)  $\rightarrow$  State F H (Frame f l (Suc pc) R
 $\Sigma$  # st) |

step-get:
next-instr (F-get F) f l pc = Some (IGet n)  $\implies$ 
n < length R  $\implies$  cast-Dyn (R ! n) = Some d  $\implies$ 
State F H (Frame f l pc R  $\Sigma$  # st)  $\rightarrow$  State F H (Frame f l (Suc pc) R (OpDyn
d # st))

```

$d \# \Sigma) \# st) |$
step-get-ubx-hit:
 $\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (IGetUbx \tau n) \implies$
 $n < \text{length } R \implies \text{cast-Dyn } (R ! n) = \text{Some } d \implies \text{unbox } \tau d = \text{Some blob} \implies$
 $\text{State } F H (\text{Frame } f l pc R \Sigma \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (\text{blob} \# \Sigma) \# st) |$
step-get-ubx-miss:
 $\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (IGetUbx \tau n) \implies$
 $n < \text{length } R \implies \text{cast-Dyn } (R ! n) = \text{Some } d \implies \text{unbox } \tau d = \text{None} \implies$
 $F' = F\text{env.map-entry } F f \text{ generalize-fundef} \implies$
 $\text{State } F H (\text{Frame } f l pc R \Sigma \# st) \rightarrow \text{State } F' H (\text{box-stack } f (\text{Frame } f l (\text{Suc } pc) R (\text{OpDyn } d \# \Sigma) \# st)) |$
step-set:
 $\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (ISet n) \implies$
 $n < \text{length } R \implies \text{cast-Dyn blob} = \text{Some } d \implies R' = R[n := \text{OpDyn } d] \implies$
 $\text{State } F H (\text{Frame } f l pc R (\text{blob} \# \Sigma) \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R' \Sigma \# st) |$
step-set-ubx:
 $\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (ISetUbx \tau n) \implies$
 $n < \text{length } R \implies \text{cast-and-box } \tau \text{ blob} = \text{Some } d \implies R' = R[n := \text{OpDyn } d] \implies$
 $\text{State } F H (\text{Frame } f l pc R (\text{blob} \# \Sigma) \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R' \Sigma \# st) |$
step-load:
 $\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (ILoad } x) \implies$
 $\text{cast-Dyn } i = \text{Some } i' \implies \text{heap-get } H (x, i') = \text{Some } d \implies$
 $\text{State } F H (\text{Frame } f l pc R (i \# \Sigma) \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (\text{OpDyn } d \# \Sigma) \# st) |$
step-load-ubx-hit:
 $\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (ILoadUbx \tau x) \implies$
 $\text{cast-Dyn } i = \text{Some } i' \implies \text{heap-get } H (x, i') = \text{Some } d \implies \text{unbox } \tau d = \text{Some blob} \implies$
 $\text{State } F H (\text{Frame } f l pc R (i \# \Sigma) \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (\text{blob} \# \Sigma) \# st) |$
step-load-ubx-miss:
 $\text{next-instr } (F\text{-get } F) f l pc = \text{Some } (ILoadUbx \tau x) \implies$
 $\text{cast-Dyn } i = \text{Some } i' \implies \text{heap-get } H (x, i') = \text{Some } d \implies \text{unbox } \tau d = \text{None} \implies$
 $F' = F\text{env.map-entry } F f \text{ generalize-fundef} \implies$
 $\text{State } F H (\text{Frame } f l pc R (i \# \Sigma) \# st) \rightarrow \text{State } F' H (\text{box-stack } f (\text{Frame } f l (\text{Suc } pc) R (\text{OpDyn } d \# \Sigma) \# st)) |$

step-store:

$$\begin{aligned} \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{IStore } x) &\implies \\ \text{cast-Dyn } i = \text{Some } i' &\implies \text{cast-Dyn } y = \text{Some } d \implies \text{heap-add } H (x, i') d = \\ H' &\implies \\ \text{State } F H (\text{Frame } f l pc R (i \# y \# \Sigma) \# st) &\rightarrow \text{State } F H' (\text{Frame } f l (\text{Suc } pc) R \Sigma \# st) \mid \end{aligned}$$

step-store-ubx:

$$\begin{aligned} \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{IStoreUbx } \tau x) &\implies \\ \text{cast-Dyn } i = \text{Some } i' &\implies \text{cast-and-box } \tau blob = \text{Some } d \implies \text{heap-add } H (x, i') d = H' \implies \\ \text{State } F H (\text{Frame } f l pc R (i \# blob \# \Sigma) \# st) &\rightarrow \text{State } F H' (\text{Frame } f l (\text{Suc } pc) R \Sigma \# st) \mid \end{aligned}$$

step-op:

$$\begin{aligned} \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{IOp } op) &\implies \\ \text{Arith } op = ar &\implies ar \leq \text{length } \Sigma \implies \\ \text{ap-map-list cast-Dyn } (\text{take } ar \Sigma) = \text{Some } \Sigma' &\implies \\ \text{Inl } op \Sigma' = \text{None} &\implies \text{Op } op \Sigma' = x \implies \\ \text{State } F H (\text{Frame } f l pc R \Sigma \# st) &\rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (\text{OpDyn } x \# \text{drop } ar \Sigma) \# st) \mid \end{aligned}$$

step-op-inl:

$$\begin{aligned} \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{IOp } op) &\implies \\ \text{Arith } op = ar &\implies ar \leq \text{length } \Sigma \implies \\ \text{ap-map-list cast-Dyn } (\text{take } ar \Sigma) = \text{Some } \Sigma' &\implies \\ \text{Inl } op \Sigma' = \text{Some } opinl &\implies \text{InlOp } opinl \Sigma' = x \implies \\ F' = \text{Fenv.map-entry } F f (\lambda fd. \text{ rewrite-fundef-body } fd l pc (\text{IOpInl } opinl)) &\implies \\ \text{State } F H (\text{Frame } f l pc R \Sigma \# st) &\rightarrow \text{State } F' H (\text{Frame } f l (\text{Suc } pc) R (\text{OpDyn } x \# \text{drop } ar \Sigma) \# st) \mid \end{aligned}$$

step-op-inl-hit:

$$\begin{aligned} \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{IOpInl } opinl) &\implies \\ \text{Arith } (\text{DeInl } opinl) = ar &\implies ar \leq \text{length } \Sigma \implies \\ \text{ap-map-list cast-Dyn } (\text{take } ar \Sigma) = \text{Some } \Sigma' &\implies \\ \text{IsInl } opinl \Sigma' \implies \text{InlOp } opinl \Sigma' = x &\implies \\ \text{State } F H (\text{Frame } f l pc R \Sigma \# st) &\rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (\text{OpDyn } x \# \text{drop } ar \Sigma) \# st) \mid \end{aligned}$$

step-op-inl-miss:

$$\begin{aligned} \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{IOpInl } opinl) &\implies \\ \text{Arith } (\text{DeInl } opinl) = ar &\implies ar \leq \text{length } \Sigma \implies \\ \text{ap-map-list cast-Dyn } (\text{take } ar \Sigma) = \text{Some } \Sigma' &\implies \\ \neg \text{IsInl } opinl \Sigma' \implies \text{InlOp } opinl \Sigma' = x &\implies \\ F' = \text{Fenv.map-entry } F f (\lambda fd. \text{ rewrite-fundef-body } fd l pc (\text{IOp } (\text{DeInl } opinl))) &\implies \\ \text{State } F H (\text{Frame } f l pc R \Sigma \# st) &\rightarrow \text{State } F' H (\text{Frame } f l (\text{Suc } pc) R (\text{OpDyn } x \# \text{drop } ar \Sigma) \# st) \mid \end{aligned}$$

step-op-ubx:

$$\begin{aligned} & \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{IOpUbx } opubx) \implies \\ & \text{DefInl } (\text{Box } opubx) = op \implies \text{Arity } op = ar \implies ar \leq \text{length } \Sigma \implies \\ & \text{UbxOp } opubx (\text{take } ar \Sigma) = \text{Some } x \implies \\ & \text{State } F H (\text{Frame } f l pc R \Sigma \# st) \rightarrow \text{State } F H (\text{Frame } f l (\text{Suc } pc) R (x \# drop ar \Sigma) \# st) \mid \end{aligned}$$

step-cjump:

$$\begin{aligned} & \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{ICJump } l_t l_f) \implies \\ & \text{cast-Dyn } y = \text{Some } d \implies \text{is-true } d \wedge l' = l_t \vee \text{is-false } d \wedge l' = l_f \implies \\ & \text{State } F H (\text{Frame } f l pc R (y \# \Sigma) \# st) \rightarrow \text{State } F H (\text{Frame } f l' 0 R \Sigma \# st) \mid \end{aligned}$$

step-call:

$$\begin{aligned} & \text{next-instr } (\text{F-get } F) f l pc = \text{Some } (\text{ICall } g) \implies \\ & \text{F-get } F g = \text{Some } gd \implies \text{arity } gd \leq \text{length } \Sigma \implies \\ & \text{list-all is-dyn-operand } (\text{take } (\text{arity } gd) \Sigma) \implies \\ & \text{frame}_g = \text{allocate-frame } g gd (\text{take } (\text{arity } gd) \Sigma) (\text{OpDyn uninitialized}) \implies \\ & \text{State } F H (\text{Frame } f l pc R_f \Sigma \# st) \rightarrow \text{State } F H (\text{frame}_g \# \text{Frame } f l pc R_f \Sigma \# st) \mid \end{aligned}$$

step-return:

$$\begin{aligned} & \text{next-instr } (\text{F-get } F) g l_g pc_g = \text{Some } \text{IReturn} \implies \\ & \text{F-get } F g = \text{Some } gd \implies \text{arity } gd \leq \text{length } \Sigma_f \implies \\ & \text{length } \Sigma_g = \text{return } gd \implies \text{list-all is-dyn-operand } \Sigma_g \implies \\ & \text{frame}_f' = \text{Frame } f l_f (\text{Suc } pc_f) R_f (\Sigma_g @ \text{drop } (\text{arity } gd) \Sigma_f) \implies \\ & \text{State } F H (\text{Frame } g l_g pc_g R_g \Sigma_g \# \text{Frame } f l_f pc_f R_f \Sigma_f \# st) \rightarrow \text{State } F H (\text{frame}_f' \# st) \end{aligned}$$

lemma *step-deterministic:*

```

assumes s1 → s2 and s1 → s3
shows s2 = s3
using assms
apply (cases s1 s2 rule: step.cases)
apply (auto simp: next-instr-length-instrs elim: step.cases dest: is-true-and-is-false-implies-False)
[3]
apply (auto simp: next-instr-length-instrs elim: step.cases dest: is-true-and-is-false-implies-False)
[3]
apply (auto simp: next-instr-length-instrs elim: step.cases dest: is-true-and-is-false-implies-False)
[3]
apply (auto simp: next-instr-length-instrs elim: step.cases dest: is-true-and-is-false-implies-False)
[3]
apply (auto simp: next-instr-length-instrs elim: step.cases dest: is-true-and-is-false-implies-False)
[3]
apply (auto simp: next-instr-length-instrs elim: step.cases dest: is-true-and-is-false-implies-False)
[3]
apply (auto simp: next-instr-length-instrs elim!: step.cases dest: is-true-and-is-false-implies-False)
[3]
apply (auto simp: next-instr-length-instrs elim!: step.cases dest: is-true-and-is-false-implies-False)
```

```

[1]
done

lemma step-right-unique: right-unique step
  using step-deterministic
  by (rule right-uniqueI)

lemma final-finished:
  assumes final F-get IReturn s
  shows finished step s
  unfolding finished-def
proof (rule notI)
  assume  $\exists x. \text{step } s x$ 
  then obtain s' where step s s' by auto
  thus False
    using ⟨final F-get IReturn s⟩
    by (auto simp: state-ok-def elim!: step.cases final.cases)
qed

sublocale ubx-sem: semantics step final F-get IReturn
  using final-finished step-deterministic
  by unfold-locales

definition load where
  load ≡ Global.load F-get (OpDyn uninitialized)

sublocale inca-lang: language step final F-get IReturn load
  by unfold-locales

end

end
theory Ubx-Verification
  imports HOL-Library.Sublist Ubx Map-Extra
begin

lemma f-g-eq-f-imp-f-comp-g-eq-f[simp]:  $(\bigwedge x. f(g x) = f x) \implies (f \circ g) = f$ 
  by (simp add: comp-def)

context ubx begin

inductive sp-instr for F ret where
  Push:
    sp-instr F ret (IPush d)  $\Sigma$  (None #  $\Sigma$ ) |
  PushUbx1:
    sp-instr F ret (IPushUbx1 u)  $\Sigma$  (Some Ubx1 #  $\Sigma$ ) |
  PushUbx2:
    sp-instr F ret (IPushUbx2 u)  $\Sigma$  (Some Ubx2 #  $\Sigma$ ) |
  Pop:

```

```

sp-instr F ret IPop ( $\tau \# \Sigma$ )  $\Sigma$  |
Get:
  sp-instr F ret (IGet n)  $\Sigma$  (None  $\# \Sigma$ ) |
GetUbx:
  sp-instr F ret (IGetUbx  $\tau$  n)  $\Sigma$  (Some  $\tau \# \Sigma$ ) |
Set:
  sp-instr F ret (ISet n) (None  $\# \Sigma$ )  $\Sigma$  |
SetUbx:
  sp-instr F ret (ISetUbx  $\tau$  n) (Some  $\tau \# \Sigma$ )  $\Sigma$  |
Load:
  sp-instr F ret (ILoad x) (None  $\# \Sigma$ ) (None  $\# \Sigma$ ) |
LoadUbx:
  sp-instr F ret (ILoadUbx  $\tau$  x) (None  $\# \Sigma$ ) (Some  $\tau \# \Sigma$ ) |
Store:
  sp-instr F ret (IStore x) (None  $\# \Sigma$ )  $\Sigma$  |
StoreUbx:
  sp-instr F ret (IStoreUbx  $\tau$  x) (None  $\# \Sigma$ ) (Some  $\tau \# \Sigma$ )  $\Sigma$  |
Op:
   $\Sigma i = (\text{replicate } (\text{Arith } op) \text{ None} @ \Sigma) \Rightarrow$ 
  sp-instr F ret (IOp op)  $\Sigma i$  (None  $\# \Sigma$ ) |
OpInl:
   $\Sigma i = (\text{replicate } (\text{Arith } (\text{DeInl } opinl)) \text{ None} @ \Sigma) \Rightarrow$ 
  sp-instr F ret (IOpInl opinl)  $\Sigma i$  (None  $\# \Sigma$ ) |
OpUbx:
   $\Sigma i = \text{fst } (\text{TypeOfOp } opubx) @ \Sigma \Rightarrow result = \text{snd } (\text{TypeOfOp } opubx) \Rightarrow$ 
  sp-instr F ret (IOpUbx opubx)  $\Sigma i$  (result  $\# \Sigma$ ) |
CJump:
  sp-instr F ret (ICJump  $l_t l_f$ ) [None] [] |
Call:
   $F f = \text{Some } (ar, r) \Rightarrow \Sigma i = \text{replicate } ar \text{ None} @ \Sigma \Rightarrow \Sigma o = \text{replicate } r \text{ None}$ 
@  $\Sigma \Rightarrow$ 
  sp-instr F ret (ICall f)  $\Sigma i$   $\Sigma o$  |
Return:  $\Sigma i = \text{replicate } ret \text{ None} \Rightarrow$ 
  sp-instr F ret IReturn  $\Sigma i$  []

```

inductive *sp-instrs* **for** *F* *ret* **where**

```

Nil:
  sp-instrs F ret []  $\Sigma \Sigma$  |
Cons:
  sp-instr F ret instr  $\Sigma i \Sigma \Rightarrow$  sp-instrs F ret instrs  $\Sigma \Sigma o \Rightarrow$ 
  sp-instrs F ret (instr  $\#$  instrs)  $\Sigma i \Sigma o$ 

```

lemmas *sp-instrs-ConsE* = *sp-instrs.cases*[*of* - - *x* $\#$ *xs* **for** *x* *xs*, *simplified*]

lemma *sp-instrs-ConsD*:

```

assumes sp-instrs F ret (instr  $\#$  instrs)  $\Sigma i \Sigma o$ 
shows  $\exists \Sigma.$  sp-instr F ret instr  $\Sigma i \Sigma \wedge$  sp-instrs F ret instrs  $\Sigma \Sigma o$ 
using assms
by (auto elim: sp-instrs-ConsE)

```

```

lemma sp-instr-deterministic:
  assumes
    sp-instr F ret instr  $\Sigma i \Sigma o$  and
    sp-instr F ret instr  $\Sigma i \Sigma o'$ 
  shows  $\Sigma o = \Sigma o'$ 
  using assms by (auto elim!: sp-instr.cases)

lemma sp-instr-right-unique: right-unique ( $\lambda(\text{instr}, \Sigma i)$   $\Sigma o$ . sp-instr F ret instr  $\Sigma i$   $\Sigma o$ )
  by (auto intro!: right-uniqueI intro: sp-instr-deterministic)

lemma sp-instrs-deterministic:
  assumes
    sp-instrs F ret instr  $\Sigma i \Sigma o$  and
    sp-instrs F ret instr  $\Sigma i \Sigma o'$ 
  shows  $\Sigma o = \Sigma o'$ 
  using assms
  proof (induction instr  $\Sigma i \Sigma o$  arbitrary:  $\Sigma o'$  rule: sp-instrs.induct)
    case (Nil  $\Sigma$ )
    then show ?case
      by (auto elim: sp-instrs.cases)
  next
    case (Cons instr  $\Sigma i \Sigma$  instrs  $\Sigma o$ )
    from Cons.prem obtain  $\Sigma'$  where sp-instr F ret instr  $\Sigma i \Sigma'$  and sp-instrs F
    ret instrs  $\Sigma' \Sigma o'$ 
      by (auto elim: sp-instrs.cases)
    hence  $\Sigma' = \Sigma$  and sp-instrs F ret instrs  $\Sigma' \Sigma o'$ 
      using Cons.hyps
      by (auto intro: sp-instr-deterministic)
    then show ?case
      by (auto intro: Cons.IH)
  qed

fun fun-call-in-range where
  fun-call-in-range F (ICall f)  $\longleftrightarrow$   $f \in \text{dom } F$  |
  fun-call-in-range F instr  $\longleftrightarrow$  True

lemma fun-call-in-range-generalize-instr[simp]:
  fun-call-in-range F (generalize-instr instr)  $\longleftrightarrow$  fun-call-in-range F instr
  by (induction instr; simp)

lemma sp-instr-complete:
  assumes fun-call-in-range F instr
  shows  $\exists \Sigma i \Sigma o$ . sp-instr F ret instr  $\Sigma i \Sigma o$ 
  using assms
  by (cases instr; auto intro: sp-instr.intros)

lemma sp-instr-Op2I:

```

```

assumes Arit $\eta$  op = 2
shows sp-instr F ret (IOp op) (None # None #  $\Sigma$ ) (None #  $\Sigma$ )
using sp-instr.Op[of - op]
unfolding assms numeral-2-eq-2
by auto

lemma
assumes
  F-def: F = Map.empty and
  arity-op: Arit $\eta$  op = 2
shows sp-intrs F 1 [IPush x, IPush y, IOp op, IReturn] [] []
by (auto simp: arity-op numeral-2-eq-2
  intro!: sp-intrs.intros
  intro: sp-instr.intros sp-instr-Op2I)

lemma sp-intrs-NilD[dest]: sp-intrs F ret []  $\Sigma i \Sigma o \implies \Sigma i = \Sigma o$ 
by (auto elim: sp-intrs.cases)

lemma sp-intrs-list-update:
assumes
  sp-intrs F ret instrs  $\Sigma i \Sigma o$  and
  sp-instr F ret (instrs!n) = sp-instr F ret instr
shows sp-intrs F ret (instrs[n := instr])  $\Sigma i \Sigma o$ 
using assms
proof (induction instrs  $\Sigma i \Sigma o$  arbitrary: n rule: sp-intrs.induct)
  case (Nil  $\Sigma$ )
  thus ?case by (auto intro: sp-intrs.Nil)
next
  case (Cons instr  $\Sigma i \Sigma$  instrs  $\Sigma o$ )
  show ?case
  proof (cases n)
    case 0
    thus ?thesis
      using Cons.hyps Cons.preds[unfolded 0, simplified]
      by (auto intro: sp-intrs.Cons)
  next
    case (Suc n')
    then show ?thesis
    using Cons.hyps Cons.preds[unfolded Suc, simplified, THEN Cons.IH]
    by (auto intro: sp-intrs.Cons)
  qed
qed

lemma sp-intrs-appendD:
assumes sp-intrs F ret (instrs1 @ instrs2)  $\Sigma i \Sigma o$ 
shows  $\exists \Sigma$ . sp-intrs F ret instrs1  $\Sigma i \Sigma \wedge$  sp-intrs F ret instrs2  $\Sigma \Sigma o$ 
using assms
proof (induction instrs1 arbitrary:  $\Sigma i$ )
  case Nil

```

```

thus ?case by (auto intro: sp-instrs.Nil)
next
  case (Cons instr instrs1)
    then obtain  $\Sigma$  where sp-instr F ret instr  $\Sigma i \Sigma$  and sp-instrs F ret (instrs1 @
  instrs2)  $\Sigma \Sigma o$ 
    by (auto elim: sp-instrs.cases)
  thus ?case
    by (auto intro: sp-instrs.Cons dest: Cons.IH)
qed

lemma sp-instrs-appendD':
  assumes sp-instrs F ret (instrs1 @ instrs2)  $\Sigma i \Sigma o$  and sp-instrs F ret instrs1
 $\Sigma i \Sigma$ 
  shows sp-instrs F ret instrs2  $\Sigma \Sigma o$ 
proof -
  obtain  $\Sigma'$  where
    sp-instrs1: sp-instrs F ret instrs1  $\Sigma i \Sigma'$  and
    sp-instrs2: sp-instrs F ret instrs2  $\Sigma' \Sigma o$ 
    using sp-instrs-appendD[OF assms(1)]
    by auto
  have  $\Sigma' = \Sigma$ 
    using sp-instrs-deterministic[OF assms(2) sp-instrs1] by simp
  thus ?thesis
    using sp-instrs2 by simp
qed

lemma sp-instrs-appendI[intro]:
  assumes sp-instrs F ret instrs1  $\Sigma i \Sigma$  and sp-instrs F ret instrs2  $\Sigma \Sigma o$ 
  shows sp-instrs F ret (instrs1 @ instrs2)  $\Sigma i \Sigma o$ 
  using assms
  by (induction instrs1  $\Sigma i \Sigma$  rule: sp-instrs.induct) (auto intro: sp-instrs.Cons)

lemma sp-instrs-singleton-conv[simp]:
  sp-instrs F ret [instr]  $\Sigma i \Sigma o \longleftrightarrow$  sp-instr F ret instr  $\Sigma i \Sigma o$ 
  by (auto intro: sp-instrs.intros elim: sp-instrs.cases)

lemma sp-instrs-singletonI:
  assumes sp-instr F ret instr  $\Sigma i \Sigma o$ 
  shows sp-instrs F ret [instr]  $\Sigma i \Sigma o$ 
  using assms by (auto intro: sp-instrs.intros)

fun local-var-in-range where
  local-var-in-range n (IGet k)  $\longleftrightarrow$   $k < n$  |
  local-var-in-range n (IGetUbx  $\tau$  k)  $\longleftrightarrow$   $k < n$  |
  local-var-in-range n (ISet k)  $\longleftrightarrow$   $k < n$  |
  local-var-in-range n (ISetUbx  $\tau$  k)  $\longleftrightarrow$   $k < n$  |
  local-var-in-range - -  $\longleftrightarrow$  True

lemma local-var-in-range-generalize-instr[simp]:
```

local-var-in-range n (*generalize-instr* $instr$) \longleftrightarrow *local-var-in-range* n $instr$
by (*cases* $instr$; *simp*)

lemma *local-var-in-range-comp-generalize-instr*[*simp*]:
local-var-in-range $n \circ *generalize-instr* = *local-var-in-range* n
using *local-var-in-range-generalize-instr*
by *auto*$

fun *jump-in-range* **where**
jump-in-range L (*ICJump* $l_t l_f$) $\longleftrightarrow \{l_t, l_f\} \subseteq L \mid$
jump-in-range $L - \longleftrightarrow$ *True*

inductive *wf-basic-block* **for** $F L ret n$ **where**
intrs $\neq [] \implies$
list-all (*local-var-in-range* n) *intrs* \implies
list-all (*fun-call-in-range* F) *intrs* \implies
list-all (*jump-in-range* L) *intrs* \implies
list-all ($\lambda i. \neg is-jump i \wedge \neg is-return i$) (*butlast* *intrs*) \implies
sp-intrs $F ret$ *intrs* $[] \implies$
wf-basic-block $F L ret n$ (*label*, *intrs*)

lemmas *wf-basic-blockI* = *wf-basic-block.simps*[*THEN iffD2*]
lemmas *wf-basic-blockD* = *wf-basic-block.simps*[*THEN iffD1*]

definition *wf-fundef* **where**
wf-fundef $F fd \longleftrightarrow$
body $fd \neq [] \wedge$
list-all
 $(wf\text{-basic-block } F (fst ` set (body fd)) (return fd) (arity fd + fundef-locals fd))$
 $(body fd)$

lemmas *wf-fundefI* = *wf-fundef-def*[*THEN iffD2*, *OF conjI*]
lemmas *wf-fundefD* = *wf-fundef-def*[*THEN iffD1*]
lemmas *wf-fundef-body-neq-NilD* = *wf-fundefD*[*THEN conjunct1*]
lemmas *wf-fundef-all-wf-basic-blockD* = *wf-fundefD*[*THEN conjunct2*]

definition *wf-fundefs* **where**
wf-fundefs $F \longleftrightarrow (\forall f. pred-option (wf-fundef (map-option funtype \circ F)) (F f))$

lemmas *wf-fundefsI* = *wf-fundefs-def*[*THEN iffD2*]
lemmas *wf-fundefsD* = *wf-fundefs-def*[*THEN iffD1*]

lemma *wf-fundefs-getD*:
shows *wf-fundefs* $F \implies F f = Some fd \implies wf\text{-fundef} (map\text{-option funtype} \circ F)$
 fd
by (*auto dest!:* *wf-fundefsD*[*THEN spec*, *of - f*])

definition *wf-prog* **where**
wf-prog $p \longleftrightarrow wf\text{-fundefs} (F\text{-get} (prog-fundefs p))$

```

definition wf-state where
  wf-state s  $\longleftrightarrow$  wf-fundefs (F-get (state-fundefs s))

lemmas wf-stateI = wf-state-def[THEN iffD2]
lemmas wf-stateD = wf-state-def[THEN iffD1]

lemma sp-instr-generalize0:
  assumes sp-instr F ret instr  $\Sigma i \Sigma o$  and
     $\Sigma i' = map (\lambda-. None) \Sigma i$  and  $\Sigma o' = map (\lambda-. None) \Sigma o$ 
  shows sp-instr F ret (generalize-instr instr)  $\Sigma i' \Sigma o'$ 
  using assms
proof (induction instr  $\Sigma i \Sigma o$  rule: sp-instr.induct)
  case (OpUbx  $\Sigma i$  opubx  $\Sigma$  result)
  then show ?case
    apply simp
    unfolding map-replicate-const
    unfolding TypeOfOp-Arity[symmetric]
    by (auto intro: sp-instr.OpInl)
qed (auto simp: intro: sp-instr.intros)

lemma sp-instrs-generalize0:
  assumes sp-instrs F ret instrs  $\Sigma i \Sigma o$  and
     $\Sigma i' = map (\lambda-. None) \Sigma i$  and  $\Sigma o' = map (\lambda-. None) \Sigma o$ 
  shows sp-instrs F ret (map generalize-instr instrs)  $\Sigma i' \Sigma o'$ 
  using assms
proof (induction instrs  $\Sigma i \Sigma o$  arbitrary:  $\Sigma i' \Sigma o'$  rule: sp-instrs.induct)
  case (Nil  $\Sigma$ )
  then show ?case
    by (auto intro: sp-instrs.Nil)
next
  case (Cons instr  $\Sigma i$   $\Sigma$  instrs  $\Sigma o$ )
  then show ?case
    by (auto intro: sp-instrs.Cons sp-instr-generalize0)
qed

lemmas sp-instr-generalize = sp-instr-generalize0[OF - refl refl]
lemmas sp-instr-generalize-Nil-Nil = sp-instr-generalize[of - - - [] [], simplified]
lemmas sp-instrs-generalize = sp-instrs-generalize0[OF - refl refl]
lemmas sp-instrs-generalize-Nil-Nil = sp-instrs-generalize[of - - - [] [], simplified]

lemma jump-in-range-generalize-instr[simp]:
  jump-in-range L (generalize-instr instr)  $\longleftrightarrow$  jump-in-range L instr
  by (cases instr) simp-all

lemma wf-basic-block-map-generalize-instr:
  assumes wf-basic-block F L ret n (label, instrs)
  shows wf-basic-block F L ret n (label, map generalize-instr instrs)
  using assms

```

```

by (auto simp add: wf-basic-block.simps list.pred-map map-butlast[symmetric]
  intro: sp-instrs-generalize-Nil-Nil)

lemma list-all-wf-basic-block-generalize-fundef:
  assumes list-all (wf-basic-block F L ret n) (body fd)
  shows list-all (wf-basic-block F L ret n) (body (generalize-fundef fd))
proof (cases fd)
  case (Fundef bblocks ar ret locals)
  then show ?thesis
    using assms
    by (auto simp: map-ran-def list.pred-map comp-def case-prod-beta
      intro: wf-basic-block-map-generalize-instr
      elim!: list.pred-mono-strong)
qed

lemma wf-fundefs-map-entry:
  assumes wf-F: wf-fundefs (F-get F) and
    same-funtype:  $\bigwedge fd. fd \in ran(F\text{-get } F) \implies \text{funtype}(f fd) = \text{funtype} fd$  and
    same-arity:  $\bigwedge fd. F\text{-get } F x = \text{Some } fd \implies \text{arity}(f fd) = \text{arity } fd$  and
    same-return:  $\bigwedge fd. F\text{-get } F x = \text{Some } fd \implies \text{return}(f fd) = \text{return } fd$  and
    same-body-length:  $\bigwedge fd. F\text{-get } F x = \text{Some } fd \implies \text{length}(\text{body}(f fd)) = \text{length}(\text{body } fd)$  and
    samelocals:  $\bigwedge fd. F\text{-get } F x = \text{Some } fd \implies \text{fundef-locals}(f fd) = \text{fundef-locals } fd$  and
    samelabels:  $\bigwedge fd. F\text{-get } F x = \text{Some } fd \implies \text{fst} \cdot \text{set}(\text{body}(f fd)) = \text{fst} \cdot \text{set}(\text{body } fd)$  and
    list-all-wf-basic-block-f:  $\bigwedge fd.$ 
       $F\text{-get } F x = \text{Some } fd \implies$ 
        list-all (wf-basic-block (map-option funtype o F-get F) (fst · set (body fd)))
        (return fd)
        ( $\text{arity } fd + \text{fundef-locals } fd$ ) (body fd)  $\implies$ 
        list-all (wf-basic-block (map-option funtype o F-get F) (fst · set (body fd)))
        (return fd)
        ( $\text{arity } fd + \text{fundef-locals } fd$ ) (body (f fd))
    shows wf-fundefs (F-get (Fenv.map-entry F x f))
proof (intro wf-fundefsI allI)
  fix y
  let ?F' = F-get (Fenv.map-entry F x f)
  show pred-option (wf-fundef (map-option funtype o ?F')) (?F' y)
  proof (cases F-get F y)
    case None
    then show ?thesis
      by (simp add: Fenv.get-map-entry-conv)
  next
    case (Some gd)
    hence wf-gd: wf-fundef (map-option funtype o F-get F) gd
      using wf-F[THEN wf-fundefsD, THEN spec, of y] by simp
    then show ?thesis
    proof (cases x = y)

```

```

case True
moreover have wf-fundef (map-option funtype o F-get (Fenv.map-entry F y f)) (f gd)
proof (rule wf-fundefI)
  show body (f gd) ≠ []
    using same-body-length[unfolded True, OF Some]
    using wf-fundef-body-neq-NilD[OF wf-gd]
    by auto
next
  show list-all (wf-basic-block (map-option funtype o F-get (Fenv.map-entry F y f)))
    (fst `set (body (f gd))) (return (f gd)) (arity (f gd) + fundef-locals (f gd))
  (body (f gd))
    using Some True
    using wf-gd[THEN wf-fundefD] Some[unfolded True]
    by (auto simp: Fenv.map-option-comp-map-entry ranI assms(2–8) intro!: wf-fundefI)
qed
ultimately show ?thesis
  by (simp add: Some)
next
case False
then show ?thesis
  unfolding Fenv.get-map-entry-neq[OF False]
  unfolding Some option.pred-inject
  using wf-gd[THEN wf-fundefD]
  using same-funtype
  by (auto simp: Fenv.map-option-comp-map-entry intro!: wf-fundefI)
qed
qed
qed

lemma wf-fundefs-generalize:
  assumes wf-F: wf-fundefs (F-get F)
  shows wf-fundefs (F-get (Fenv.map-entry F f generalize-fundef))
  using wf-F
  apply (elim wf-fundefs-map-entry)
  by (auto elim: list-all-wf-basic-block-generalize-fundef)

lemma list-all-wf-basic-block-rewrite-fundef-body:
  assumes
    list-all (wf-basic-block F L ret n) (body fd) and
    instr-at fd l pc = Some instr and
    sp-instr-eq: sp-instr F ret instr = sp-instr F ret instr' and
    local-var-in-range-iff: local-var-in-range n instr' ↔ local-var-in-range n instr
  and
    fun-call-in-range-iff: fun-call-in-range F instr' ↔ fun-call-in-range F instr
  and
    jump-in-range-iff: jump-in-range L instr' ↔ jump-in-range L instr and

```

```

is-jump-iff: is-jump instr'  $\longleftrightarrow$  is-jump instr and
is-return-iff: is-return instr'  $\longleftrightarrow$  is-return instr
shows list-all (wf-basic-block F L ret n) (body (rewrite-fundef-body fd l pc instr'))
proof (cases fd)
  case (Fundef bblocks ar ret' locals)
  have wf-bblocks: list-all (wf-basic-block F L ret n) bblocks
    using assms(1) unfolding Fundef by simp

  moreover have wf-basic-block F L ret n (l, instrs[pc := instr'])
    if map-of bblocks l = Some instrs for instrs
    proof –
      have wf-basic-block-l-instrs: wf-basic-block F L ret n (l, instrs)
        by (rule list-all-map-of-SomeD[OF wf-bblocks that])

      have nth-instrs-pc: pc < length instrs ! pc = instr
        using assms(2)[unfolded instr-at-def Fundef, simplified, unfolded that, simplified]
        by simp-all

    show ?thesis
    proof (rule wf-basic-blockI, simp, intro conjI)
      show instrs  $\neq$  []
        using wf-basic-block-l-instrs
        by (auto elim: wf-basic-block.cases)
    next
      show list-all (local-var-in-range n) (instrs[pc := instr'])
        using wf-basic-block-l-instrs nth-instrs-pc
        by (auto simp: local-var-in-range-iff
          elim!: wf-basic-block.cases intro!: list-all-list-updateI)
    next
      show list-all ( $\lambda i. \neg$  is-jump i  $\wedge$   $\neg$  is-return i) (butlast (instrs[pc := instr']))
        using wf-basic-block-l-instrs nth-instrs-pc
        apply (auto simp: butlast-list-update elim!: wf-basic-block.cases)
        apply (rule list-all-list-updateI)
        apply assumption
        by (simp add: is-jump-iff is-return-iff list-all-length nth-butlast)
    next
      show sp-instrs F ret (instrs[pc := instr']) [] []
        using wf-basic-block-l-instrs nth-instrs-pc sp-instr-eq
        by (auto elim!: wf-basic-block.cases elim!: sp-instrs-list-update)
    next
      show list-all (fun-call-in-range F) (instrs[pc := instr'])
        using wf-basic-block-l-instrs nth-instrs-pc
        by (auto simp: fun-call-in-range-iff
          elim!: wf-basic-block.cases intro!: list-all-list-updateI)
    next
      show list-all (jump-in-range L) (instrs[pc := instr'])
        using wf-basic-block-l-instrs nth-instrs-pc
        by (auto simp: jump-in-range-iff elim!: wf-basic-block.cases intro!: list-all-list-updateI)

```

```

qed
qed

with Fundef show ?thesis
  using assms(1,2)
  by (auto simp add: rewrite-fundef-body-def map-entry-map-of-Some-conv
    intro: list-all-updateI dest!: instr-atD)
qed

lemma wf-fundefs-rewrite-body:
  assumes wf-fundefs (F-get F) and
    next-instr (F-get F) f l pc = Some instr and
    sp-instr-eq: ⋀ret.
      sp-instr (map-option funtype o F-get F) ret instr' =
      sp-instr (map-option funtype o F-get F) ret instr and
      local-var-in-range-iff: ⋀n. local-var-in-range n instr' ↔ local-var-in-range n
    instr and
    fun-call-in-range-iff:
      fun-call-in-range (map-option funtype o F-get F) instr' ↔
      fun-call-in-range (map-option funtype o F-get F) instr and
    jump-in-range-iff: ⋀L. jump-in-range L instr' ↔ jump-in-range L instr and
    is-jump-iff: is-jump instr' ↔ is-jump instr and
    is-return-iff: is-return instr' ↔ is-return instr
  shows wf-fundefs (F-get (Fenv.map-entry F f (λfd. rewrite-fundef-body fd l pc
  instr)))
proof -
  obtain fd where F-f: F-get F f = Some fd and instr-at-pc: instr-at fd l pc =
  Some instr
    using assms(2)[THEN next-instrD]
    by auto

  show ?thesis
    using assms(1)
  proof (rule wf-fundefs-map-entry)
    fix fd
    show fst ` set (body (rewrite-fundef-body fd l pc instr')) = fst ` set (body fd)
      by (cases fd) (simp add: rewrite-fundef-body-def dom-map-entry)
  next
    fix gd
    assume F-get F f = Some gd
    hence gd = fd
      using F-f by simp
    then show
      list-all (wf-basic-block (map-option funtype o F-get F) (fst ` set (body gd)))
    (return gd)
      (arity gd + fundef-locals gd)) (body gd) ==>
      list-all (wf-basic-block (map-option funtype o F-get F) (fst ` set (body gd)))
    (return gd)
      (arity gd + fundef-locals gd)) (body (rewrite-fundef-body gd l pc instr'))

```

```

using assms(1,3-)
using F-f instr-at-pc
by (elim list-all-wf-basic-block-rewrite-fundef-body) simp-all
qed simp-all
qed

lemma sp-instr-Op-OpInl-conv:
assumes op = Ⓜ opinl
shows sp-instr F ret (IOp op) = sp-instr F ret (IOpInl opinl)
by (auto simp: assms elim: sp-instr.cases intro: sp-instr.OpInl sp-instr.Op)

lemma wf-state-step-preservation:
assumes wf-state s and step s s'
shows wf-state s'
using assms(2,1)
proof (cases s s' rule: step.cases)
case (step-op-inl F f l pc op ar Σ Σ' opinl x F' H R st)
then show ?thesis
using assms(1)
by (auto intro!: wf-stateI intro: sp-instr-Op-OpInl-conv[symmetric]
      elim!: wf-fundefs-rewrite-body dest!: wf-stateD Inl-invertible)
next
case (step-op-inl-miss F f l pc opinl ar Σ Σ' x F' H R st)
then show ?thesis
using assms(1)
by (auto intro!: wf-stateI intro: sp-instr-Op-OpInl-conv
      elim!: wf-fundefs-rewrite-body dest!: wf-stateD)
qed (auto simp: box-stack-def
      intro!: wf-stateI wf-fundefs-generalize
      intro: sp-instr.intros
      dest!: wf-stateD)

end

end
theory Unboxed-lemmas
imports Unboxed
begin

lemma cast-Dyn-eq-Some-imp-typeof: cast-Dyn u = Some d ==> typeof u = None
by (auto elim: cast-Dyn.elims)

lemma typeof-bind-OpDyn[simp]: typeof o OpDyn = (λx. None)
by auto

lemma is-dyn-operand-eq-typeof: is-dyn-operand = (λx. typeof x = None)

```

```

proof (intro ext)
  fix x
  show is-dyn-operand x = (typeof x = None)
    by (cases x; simp)
qed

lemma is-dyn-operand-eq-typeof-Dyn[simp]: is-dyn-operand x <→ typeof x = None
  by (cases x; simp)

lemma typeof-unboxed-eq-const:
  fixes x
  shows
    typeof x = None  $\longleftrightarrow$  ( $\exists d. x = OpDyn d$ )
    typeof x = Some Ubx1  $\longleftrightarrow$  ( $\exists n. x = OpUbx1 n$ )
    typeof x = Some Ubx2  $\longleftrightarrow$  ( $\exists b. x = OpUbx2 b$ )
  by (cases x; simp)+

lemmas typeof-unboxed-inversion = typeof-unboxed-eq-const[ THEN iffD1 ]

lemma cast-inversions:
  cast-Dyn x = Some d  $\implies$  x = OpDyn d
  cast-Ubx1 x = Some n  $\implies$  x = OpUbx1 n
  cast-Ubx2 x = Some b  $\implies$  x = OpUbx2 b
  by (cases x; simp)+

lemma ap-map-list-cast-Dyn-replicate:
  assumes ap-map-list cast-Dyn xs = Some ys
  shows map typeof xs = replicate (length xs) None
  using assms
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  from Cons.preds show ?case
    by (auto intro: Cons.IH dest: cast-inversions(1) simp: ap-option-eq-Some-conv)
qed

context unboxedval begin

lemma unbox-typeof[simp]: unbox τ d = Some blob  $\implies$  typeof blob = Some τ
  by (cases τ; auto)

lemma cast-and-box-imp-typeof[simp]: cast-and-box τ blob = Some d  $\implies$  typeof blob = Some τ
  using cast-inversions[of blob]
  by (induction τ; auto dest: cast-inversions[of blob])

lemma norm-unbox-inverse[simp]: unbox τ d = Some blob  $\implies$  norm-unboxed blob

```

```

= d
  using box-unbox-inverse
  by (cases  $\tau$ ; auto)

lemma norm-cast-and-box-inverse[simp]:
  cast-and-box  $\tau$  blob = Some d  $\Rightarrow$  norm-unboxed blob = d
  by (induction  $\tau$ ; auto elim: cast-Dyn.elims cast-Ubx1.elims cast-Ubx2.elims)

lemma typeof-and-norm-unboxed-imp-cast-Dyn:
  assumes typeof  $x'$  = None norm-unboxed  $x' = x$ 
  shows cast-Dyn  $x' = \text{Some } x$ 
  using assms
  unfolding typeof-unboxed-eq-const
  by auto

lemma typeof-and-norm-unboxed-imp-cast-and-box:
  assumes typeof  $x' = \text{Some } \tau$  norm-unboxed  $x' = x$ 
  shows cast-and-box  $\tau x' = \text{Some } x$ 
  using assms
  by (induction  $\tau$ ; induction  $x'$ ; simp)

lemma norm-unboxed-bind-OpDyn[simp]: norm-unboxed  $\circ$  OpDyn = id
  by auto

lemmas box-stack-Nil[simp] = list.map(1)[of box-frame  $f$  for  $f$ , folded box-stack-def]
lemmas box-stack-Cons[simp] = list.map(2)[of box-frame  $f$  for  $f$ , folded box-stack-def]

lemma typeof-box-operand[simp]: typeof (box-operand  $u$ ) = None
  by (cases  $u$ ) simp-all

lemma typeof-box-operand-comp[simp]: typeof  $\circ$  box-operand = ( $\lambda\_. \text{None}$ )
  by auto

lemma is-dyn-box-operand: is-dyn-operand (box-operand  $x$ )
  by (cases  $x$ ) simp-all

lemma is-dyn-operand-comp-box-operand[simp]: is-dyn-operand  $\circ$  box-operand =
  ( $\lambda\_. \text{True}$ )
  using is-dyn-box-operand by auto

lemma norm-box-operand[simp]: norm-unboxed (box-operand  $x$ ) = norm-unboxed
 $x$ 
  by (cases  $x$ ) simp-all

end

end
theory Inca-to-Ubx-simulation
  imports List-util Result

```

VeriComp.Simulation
Inca Ubx Ubx-Verification Unboxed-lemmas

```

begin

lemma take:-Suc n = length xs ==> take n xs = butlast xs
  using butlast-conv-take diff-Suc-1 append-butlast-last-id
  by (metis butlast-conv-take diff-Suc-1)

lemma append-take-singleton-conv:Suc n = length xs ==> xs = take n xs @ [xs ! n]
proof (induction xs arbitrary: n)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then show ?case
  proof (cases n)
    case 0
    then show ?thesis
      using Cons
      by simp
  next
    case (Suc n')
    have Suc n' = length xs
      by (rule Cons.preds[unfolded Suc, simplified])
    from Suc show ?thesis
      by (auto intro: Cons.IH[OF ‹Suc n' = length xs›])
  qed
qed

```

15 Locale imports

```

locale inca-to-ubx-simulation =
  Sinca: inca
  Finca-empty Finca-get Finca-add Finca-to-list
  heap-empty heap-get heap-add heap-to-list
  uninitialized is-true is-false
  Ⓣ Arity Inl Ⓣ Inl IsInl DeInl +
  Subx: ubx
  Fubx-empty Fubx-get Fubx-add Fubx-to-list
  heap-empty heap-get heap-add heap-to-list
  uninitialized is-true is-false
  box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
  Ⓣ Arity Inl Ⓣ Inl IsInl DeInl Ubx Ⓣ Ubx Box TypeDefOp
for
  — Functions environments
  Finca-empty and
  Finca-get :: 'fenv-inca => 'fun => ('label, ('dyn, 'var, 'fun, 'label, 'op, 'opInl)
  Inca.instr) fundef option and

```

Finca-add and *Finca-to-list* and

Fubx-empty and

Fubx-get :: $'fenv\text{-}ubx \Rightarrow 'fun \Rightarrow ('label, ('dyn, 'var, 'fun, 'label, 'op, 'opinl, 'opubx, 'ubx1, 'ubx2) Ubx.instr) fundef option$ and
Fubx-add and *Fubx-to-list* and

— Memory heap

heap-empty and *heap-get* :: $'henv \Rightarrow 'var \times 'dyn \Rightarrow 'dyn option$ and *heap-add* and *heap-to-list* and

— Dynamic values

uninitialized :: $'dyn$ and *is-true* and *is-false* and

— Unboxed values

box-ubx1 and *unbox-ubx1* and
box-ubx2 and *unbox-ubx2* and

— n-ary operations

\mathcal{O} p and Arity and InOp and InI and IsInI and DeInI and UriOp and Uri and Box and TypeOp and begin

16 Normalization

fun *norm-instr* **where**

```

norm-instr (Ubx.IPush d) = Inca.IPush d |
norm-instr (Ubx.IPushUbx1 n) = Inca.IPush (box-ubx1 n) |
norm-instr (Ubx.IPushUbx2 b) = Inca.IPush (box-ubx2 b) |
norm-instr Ubx.IPop = Inca.IPop |
norm-instr (Ubx.IGet n) = Inca.IGet n |
norm-instr (Ubx.IGetUbx - n) = Inca.IGet n |
norm-instr (Ubx.ISet n) = Inca.ISet n |
norm-instr (Ubx.ISetUbx - n) = Inca.ISet n |
norm-instr (Ubx.ILoad x) = Inca.ILoad x |
norm-instr (Ubx.ILoadUbx - x) = Inca.ILoad x |
norm-instr (Ubx.IStore x) = Inca.IStore x |
norm-instr (Ubx.IStoreUbx - x) = Inca.IStore x |
norm-instr (Ubx.IOp op) = Inca.IOp op |
norm-instr (Ubx.IOpInl op) = Inca.IOpInl op |
norm-instr (Ubx.IOpUbx op) = Inca.IOpInl (Box op) |
norm-instr (Ubx.ICJump lt lf) = Inca.ICJump lt lf |
norm-instr (Ubx.ICall x) = Inca.ICall x |
norm-instr Ubx.IReturn = Inca.IReturn

```

lemma *norm-generalize-instr[simp]*: *norm-instr* (*Subx.generalize-instr instr*) = *norm-instr instr*

by (*cases instr*) *simp-all*

```

abbreviation norm-eq where
  norm-eq x y ≡ x = norm-instr y

definition rel-fundefs where
  rel-fundefs f g = (λ x. rel-option (rel-fundef (=) norm-eq) (f x) (g x))

lemma rel-fundefsI:
  assumes λ x. rel-option (rel-fundef (=) norm-eq) (F1 x) (F2 x)
  shows rel-fundefs F1 F2
  using assms
  by (simp add: rel-fundefs-def)

lemma rel-fundefsD:
  assumes rel-fundefs F1 F2
  shows rel-option (rel-fundef (=) norm-eq) (F1 x) (F2 x)
  using assms
  by (simp add: rel-fundefs-def)

lemma rel-fundefs-next-instr:
  assumes rel-F1-F2: rel-fundefs F1 F2
  shows rel-option norm-eq (next-instr F1 f l pc) (next-instr F2 f l pc)
  using rel-F1-F2[THEN rel-fundefsD, of f]
  proof (cases rule: option.rel-cases)
    case None
    thus ?thesis by (simp add: next-instr-def)
  next
    case (Some fd1 fd2)
    then show ?thesis
      by (auto simp: next-instr-def intro: rel-fundef-imp-rel-option-instr-at)
  qed

lemma rel-fundefs-next-instr1:
  assumes rel-F1-F2: rel-fundefs F1 F2 and next-instr1: next-instr F1 f l pc = Some instr1
  shows ∃ instr2. next-instr F2 f l pc = Some instr2 ∧ norm-eq instr1 instr2
  using rel-fundefs-next-instr[OF rel-F1-F2, of f l pc]
  unfolding next-instr1
  unfolding option-rel-Some1
  by assumption

lemma rel-fundefs-next-instr2:
  assumes rel-F1-F2: rel-fundefs F1 F2 and next-instr2: next-instr F2 f l pc = Some instr2
  shows ∃ instr1. next-instr F1 f l pc = Some instr1 ∧ norm-eq instr1 instr2
  using rel-fundefs-next-instr[OF rel-F1-F2, of f l pc]
  unfolding next-instr2
  unfolding option-rel-Some2
  by assumption

```

```

lemma rel-fundefs-empty: rel-fundefs ( $\lambda\_. \text{None}$ ) ( $\lambda\_. \text{None}$ )
  by (simp add: rel-fundefs-def)

lemma rel-fundefs-None1:
  assumes rel-fundefs f g and f x = None
  shows g x = None
  by (metis assms rel-fundefs-def rel-option-None1)

lemma rel-fundefs-None2:
  assumes rel-fundefs f g and g x = None
  shows f x = None
  by (metis assms rel-fundefs-def rel-option-None2)

lemma rel-fundefs-Some1:
  assumes rel-fundefs f g and f x = Some y
  shows  $\exists z. g x = \text{Some } z \wedge \text{rel-fundef } (=) \text{ norm-eq } y z$ 
  proof -
    from assms(1) have rel-option (rel-fundef (=) norm-eq) (f x) (g x)
      unfolding rel-fundefs-def by simp
      with assms(2) show ?thesis
        by (simp add: option-rel-Some1)
  qed

lemma rel-fundefs-Some2:
  assumes rel-fundefs f g and g x = Some y
  shows  $\exists z. f x = \text{Some } z \wedge \text{rel-fundef } (=) \text{ norm-eq } z y$ 
  proof -
    from assms(1) have rel-option (rel-fundef (=) norm-eq) (f x) (g x)
      unfolding rel-fundefs-def by simp
      with assms(2) show ?thesis
        by (simp add: option-rel-Some2)
  qed

lemma rel-fundefs-rel-option:
  assumes rel-fundefs f g and  $\bigwedge x y. \text{rel-fundef } (=) \text{ norm-eq } x y \implies h x y$ 
  shows rel-option h (f z) (g z)
  proof -
    have rel-option (rel-fundef (=) norm-eq) (f z) (g z)
      using assms(1)[unfolded rel-fundefs-def] by simp
    then show ?thesis
      unfolding rel-option-unfold
      by (auto simp add: assms(2))
  qed

lemma rel-fundef-generalizeI:
  assumes rel-fundef (=) norm-eq fd1 fd2
  shows rel-fundef (=) norm-eq fd1 (Subx.generalize-fundef fd2)
  using assms
  by (cases rule: fundef.rel-cases)

```

```

(auto simp: map-ran-def list.rel-map elim: list.rel-mono-strong)

lemma rel-fundefs-generalizeI:
  assumes rel-fundefs (Finca-get F1) (Fubx-get F2)
  shows rel-fundefs (Finca-get F1) (Fubx-get (Subx.Fenv.map-entry F2 f Subx.generalize-fundef))
proof (rule rel-fundefsI)
  fix x
  show rel-option (rel-fundef (=) norm-eq)
    (Finca-get F1 x) (Fubx-get (Subx.Fenv.map-entry F2 f Subx.generalize-fundef)
  x)
  unfolding Subx.Fenv.get-map-entry-conv
  unfolding option.rel-map
  using assms(1)[THEN rel-fundefsD, of x]
  by (auto intro: rel-fundef-generalizeI elim: option.rel-mono-strong)
qed

lemma rel-fundefs-rewriteI:
assumes
  rel-F1-F2: rel-fundefs (Finca-get F1) (Fubx-get F2) and
  norm-eq instr1' instr2'
shows rel-fundefs
  (Finca-get (Sinca.Fenv.map-entry F1 f (λfd. rewrite-fundef-body fd l pc instr1'))))
  (Fubx-get (Subx.Fenv.map-entry F2 f (λfd. rewrite-fundef-body fd l pc instr2'))))
  (is rel-fundefs (Finca-get ?F1') (Fubx-get ?F2'))
proof (rule rel-fundefsI)
  fix x
  show rel-option (rel-fundef (=) norm-eq) (Finca-get ?F1' x) (Fubx-get ?F2' x)
  proof (cases f = x)
    case True
    show ?thesis
      using rel-F1-F2[THEN rel-fundefsD, of x] True assms(2)
      by (cases rule: option.rel-cases) (auto intro: rel-fundef-rewrite-body)
  next
    case False
    then show ?thesis
      using rel-F1-F2[THEN rel-fundefsD, of x] by simp
  qed
qed

```

17 Equivalence of call stacks

```

definition norm-stack :: ('dyn, 'ubx1, 'ubx2) unboxed list ⇒ 'dyn list where
  norm-stack Σ ≡ List.map Subx.norm-unboxed Σ

```

```

lemma norm-stack-Nil[simp]: norm-stack [] = []
  by (simp add: norm-stack-def)

```

```

lemma norm-stack-Cons[simp]: norm-stack (d # Σ) = Subx.norm-unboxed d # norm-stack Σ

```

```

by (simp add: norm-stack-def)

lemma norm-stack-append: norm-stack (xs @ ys) = norm-stack xs @ norm-stack
ys
by (simp add: norm-stack-def)

lemmas drop-norm-stack = drop-map[where f = Subx.norm-unboxed, folded norm-stack-def]
lemmas take-norm-stack = take-map[where f = Subx.norm-unboxed, folded norm-stack-def]
lemmas norm-stack-map = map-map[where f = Subx.norm-unboxed, folded norm-stack-def]

lemma norm-box-stack[simp]: norm-stack (map Subx.box-operand Σ) = norm-stack
Σ
by (induction Σ) (auto simp: norm-stack-def)

lemma length-norm-stack[simp]: length (norm-stack xs) = length xs
by (simp add: norm-stack-def)

definition is-valid-fun-call where
is-valid-fun-call F f l pc Σ g ≡ next-instr F f l pc = Some (ICall g) ∧
(∃ gd. F g = Some gd ∧ arity gd ≤ length Σ ∧ list-all is-dyn-operand (take
(arity gd) Σ))

lemma is-valid-funcall-map-entry-generalize-fundefI:
assumes is-valid-fun-call (Fubx-get F2) g l pc Σ z
shows is-valid-fun-call (Fubx-get (Subx.Fenv.map-entry F2 f Subx.generalize-fundef))
g l pc Σ z
proof (cases f = z)
case True
then show ?thesis
using assms
by (cases z = g)
(auto simp: is-valid-fun-call-def next-instr-def Subx.instr-at-generalize-fundef-conv)
next
case False
then show ?thesis
using assms
by (cases Fubx-get F2 g)
(auto simp: is-valid-fun-call-def next-instr-def
Subx.instr-at-generalize-fundef-conv Subx.Fenv.get-map-entry-conv)
qed

lemma is-valid-fun-call-map-box-operandI:
assumes is-valid-fun-call (Fubx-get F2) g l pc Σ z
shows is-valid-fun-call (Fubx-get F2) g l pc (map Subx.box-operand Σ) z
using assms
unfolding is-valid-fun-call-def
by (auto simp: take-map list.pred-map list.pred-True)

lemma inst-at-rewrite-fundef-body-disj:

```

```

instr-at (rewrite-fundef-body fd l pc instr) l pc = Some instr ∨
  instr-at (rewrite-fundef-body fd l pc instr) l pc = None
proof (cases fd)
  case (Fundef bblocks ar locals)
  show ?thesis
    proof (cases map-of bblocks l)
      case None
      thus ?thesis
        using Fundef
        by (simp add: rewrite-fundef-body-def instr-at-def map-entry-map-of-None-conv)
    next
      case (Some instr')
      moreover hence l ∈ fst ` set bblocks
        by (meson domI domIff map-of-eq-None-iff)
      ultimately show ?thesis
        using Fundef
        apply (auto simp add: rewrite-fundef-body-def instr-at-def map-entry-map-of-Some-conv)
        by (smt (verit, ccfv-threshold) length-list-update nth-list-update-eq option.case-eq-if
          option.distinct(1) option.sel update-Some-unfold)
      qed
    qed

lemma is-valid-fun-call-map-entry-conv:
  assumes next-instr (Fubx-get F2) f l pc = Some instr ∨ is-fun-call instr ∨
  is-fun-call instr'
  shows
    is-valid-fun-call (Fubx-get (Subx.Fenv.map-entry F2 f (λfd. rewrite-fundef-body
      fd l pc instr')))=
      is-valid-fun-call (Fubx-get F2)
  proof (intro ext)
    fix f' l' pc' Σ g
    show
      is-valid-fun-call (Fubx-get (Subx.Fenv.map-entry F2 f (λfd. rewrite-fundef-body
        fd l pc instr'))) f' l' pc' Σ g =
        is-valid-fun-call (Fubx-get F2) f' l' pc' Σ g
    proof (cases f = f')
      case True
      then show ?thesis
        using assms
        apply (cases f = g)
        by (auto simp: is-valid-fun-call-def next-instr-eq-Some-conv
          instr-at-rewrite-fundef-body-conv if-split-eq1)
    next
      case False
      then show ?thesis
        using assms
        apply (cases f = g)
        by (auto simp: is-valid-fun-call-def next-instr-eq-Some-conv)
    qed

```

qed

```
lemma is-valid-fun-call-map-entry-neq-f-neq-l:
  assumes f ≠ g l ≠ l'
  shows
    is-valid-fun-call (Fubx-get (Subx.Fenv.map-entry F2 f (λfd. rewrite-fundef-body
    fd l pc instr'))) g l' =
      is-valid-fun-call (Fubx-get F2) g l'
    apply (intro ext)
    unfolding is-valid-fun-call-def
    using assms
    apply (simp add: next-instr-eq-Some-conv)
    apply (rule iffI; simp)
    unfolding Subx.Fenv.get-map-entry-conv
    apply simp
    apply (metis arity-rewrite-fundef-body)
    apply safe
    by simp

inductive rel-stacktraces for F where
  rel-stacktraces-Nil:
    rel-stacktraces F opt [] []
  rel-stacktraces-Cons:
    rel-stacktraces F (Some f) st1 st2 ==>
    Σ1 = map Subx.norm-unboxed Σ2 ==>
    R1 = map Subx.norm-unboxed R2 ==>
    list-all is-dyn-operand R2 ==>
    F f = Some fd2 ==> map-of (body fd2) l = Some instrs ==>
    Subx.sp-intrs (map-option funtype ∘ F) (return fd2) (take pc instrs) [] (map
    typeof Σ2) ==>
    pred-option (is-valid-fun-call F f l pc Σ2) opt ==>
    rel-stacktraces F opt (Frame f l pc R1 Σ1 # st1) (Frame f l pc R2 Σ2 # st2)

lemma rel-stacktraces-map-entry-generalize-fundefI[intro]:
  assumes rel-stacktraces (Fubx-get F2) opt st1 st2
  shows rel-stacktraces (Fubx-get (Subx.Fenv.map-entry F2 f Subx.generalize-fundef))
    opt st1 (Subx.box-stack f st2)
  using assms(1)
proof (induction opt st1 st2 rule: rel-stacktraces.induct)
  case (rel-stacktraces-Nil opt)
  thus ?case
    by (auto intro: rel-stacktraces.rel-stacktraces-Nil)
next
  case (rel-stacktraces-Cons g st1 st2 Σ1 Σ2 R1 R2 gd2 l instrs pc opt)
  show ?case
  proof (cases f = g)
    case True
    then show ?thesis
```

```

using rel-stacktraces-Cons
apply auto
apply (rule rel-stacktraces.rel-stacktraces-Cons)
  apply assumption
  apply simp
  apply (rule refl)
  apply assumption
  apply simp
by (auto simp add: take-map Subx.map-of-generalize-fundef-conv
    intro!: Subx.sp-instrs-generalize0
    intro!: is-valid-funcall-map-entry-generalize-fundefI is-valid-fun-call-map-box-operandI
    elim!: option.pred-mono-strong)
next
case False
then show ?thesis
using rel-stacktraces-Cons
by (auto intro: rel-stacktraces.intros is-valid-funcall-map-entry-generalize-fundefI
    elim!: option.pred-mono-strong)
qed
qed

lemma rel-stacktraces-map-entry-rewrite-fundef-body:
assumes
  rel-stacktraces (Fubx-get F2) opt st1 st2 and
  next-instr (Fubx-get F2) f l pc = Some instr and
  ∃ret. Subx.sp-instr (map-option funtype o Fubx-get F2) ret instr =
    Subx.sp-instr (map-option funtype o Fubx-get F2) ret instr' and
    ¬ is-fun-call instr ∨ is-fun-call instr'
shows rel-stacktraces
  (Fubx-get (Subx.Fenv.map-entry F2 f (λfd. rewrite-fundef-body fd l pc instr'))))
opt st1 st2
using assms(1)
proof (induction opt st1 st2 rule: rel-stacktraces.induct)
  case (rel-stacktraces-Nil opt)
  then show ?case
    by (auto intro: rel-stacktraces.rel-stacktraces-Nil)
next
  case (rel-stacktraces-Cons g st1 st2 Σ1 Σ2 R1 R2 gd2 l' instrs pc' opt)
  show ?case (is rel-stacktraces (Fubx-get ?F2') opt ?st1 ?st2)
  proof (cases f = g)
    case True
    show ?thesis
  proof (cases l' = l)
    case True
    show ?thesis
    apply (rule rel-stacktraces.rel-stacktraces-Cons)
    using rel-stacktraces-Cons.IH apply simp
    using rel-stacktraces-Cons.hyps apply simp
    using rel-stacktraces-Cons.hyps apply simp

```

```

using rel-stacktraces-Cons.hyps apply simp
using rel-stacktraces-Cons.hyps ⟨f = g⟩ apply simp
using rel-stacktraces-Cons.hyps True apply simp
using rel-stacktraces-Cons.hyps apply simp
using rel-stacktraces-Cons.hyps assms ⟨f = g⟩ True
apply (cases pc' ≤ pc) []
apply (auto simp add: take-update-swap intro!: Subx.sp-instrs-list-update
      dest!: next-instrD instr-atD) [2]
using rel-stacktraces-Cons.hyps
unfolding is-valid-fun-call-map-entry-conv[OF assms(2,4,5)]
by simp
next
case False
show ?thesis
proof (rule rel-stacktraces.rel-stacktraces-Cons)
show Fubx-get ?F2' g = Some (rewrite-fundef-body gd2 l pc instr')
  unfolding ⟨f = g⟩
  using rel-stacktraces-Cons.hyps by simp
next
show pred-option (is-valid-fun-call (Fubx-get ?F2') g l' pc' Σ2) opt
  unfolding is-valid-fun-call-map-entry-conv[OF assms(2,4,5)]
  using rel-stacktraces-Cons.hyps by simp
qed (insert rel-stacktraces-Cons False, simp-all)
qed
next
case False
then show ?thesis
  using rel-stacktraces-Cons
  by (auto simp: is-valid-fun-call-map-entry-conv[OF assms(2,4,5)])
    intro!: rel-stacktraces.rel-stacktraces-Cons)
qed
qed

```

18 Simulation relation

```

inductive match (infix ‘~’ 55) where
matchI: Subx.wf-state (State F2 H st2) ==>
  rel-fundefs (Finca-get F1) (Fubx-get F2) ==>
  rel-stacktraces (Fubx-get F2) None st1 st2 ==>
  match (State F1 H st1) (State F2 H st2)

```

```

lemmas matchI[consumes 0, case-names wf-state rel-fundefs rel-stacktraces] =
match.intros(1)

```

19 Backward simulation

```

lemma map-eq-append-map-drop:
map f xs = ys @ map f (drop n xs) ←→ map f (take n xs) = ys

```

```

by (metis append-same-eq append-take-drop-id map-append)

lemma ap-map-list-cast-Dyn-to-map-norm:
  assumes ap-map-list cast-Dyn xs = Some ys
  shows ys = map Subx.norm-unboxed xs
proof -
  from assms have list-all2 (λx y. cast-Dyn x = Some y) xs ys
    by (simp add: ap-map-list-iff-list-all2)
  thus ?thesis
    by (induction xs ys rule: list.rel-induct) (auto dest: cast-inversions)
qed

lemma ap-map-list-cast-Dyn-to-all-Dyn:
  assumes ap-map-list cast-Dyn xs = Some ys
  shows list-all (λx. typeof x = None) xs
proof -
  from assms have list-all2 (λx y. cast-Dyn x = Some y) xs ys
    by (simp add: ap-map-list-iff-list-all2)
  hence list-all2 (λx y. typeof x = None) xs ys
    by (auto intro: list.rel-mono-strong cast-Dyn-eq-Some-imp-typeof)
  thus ?thesis
    by (induction xs ys rule: list.rel-induct) auto
qed

lemma ap-map-list-cast-Dyn-map-typeof-replicate-conv:
  assumes ap-map-list cast-Dyn xs = Some ys and n = length xs
  shows map typeof xs = replicate n None
  using assms
  by (auto simp: list.pred-set intro!: replicate-eq-map[symmetric]
    dest!: ap-map-list-cast-Dyn-to-all-Dyn)

lemma cast-Dyn-eq-Some-conv-norm-unboxed[simp]: cast-Dyn i = Some i' ⟹
Subx.norm-unboxed i = i'
  by (cases i) simp-all

lemma cast-Dyn-eq-Some-conv-typeof[simp]: cast-Dyn i = Some i' ⟹ typeof i =
None
  by (cases i) simp-all

lemma backward-lockstep-simulation:
  assumes match s1 s2 and Subx.step s2 s2'
  shows ∃ s1'. Sinca.step s1 s1' ∧ match s1' s2'
  using assms
proof (induction s1 s2 rule: match.induct)
  case (matchI F2 H st2 F1 st1)
  from matchI(3,1,2,4) show ?case
  proof (induction None :: 'fun option st1 st2 rule: rel-stacktraces.induct)
    case rel-stacktraces-Nil
    hence False by (auto elim: Subx.step.cases)
  qed
qed

```

```

thus ?case by simp
next
  case (rel-stacktraces-Cons f st1 st2 Σ1 Σ2 R1 R2 fd2 l instrs pc)
  note hyps = rel-stacktraces-Cons.hyps
  note prems = rel-stacktraces-Cons.prems
  have wf-state2: Subx.wf-state (State F2 H (Frame f l pc R2 Σ2 # st2)) using
    prems by simp
  have rel-F1-F2: rel-fundefs (Finca-get F1) (Fubx-get F2) using prems by simp
  have rel-st1-st2: rel-stacktraces (Fubx-get F2) (Some f) st1 st2 using hyps by
    simp
  have Σ1-def: Σ1 = map Subx.norm-unboxed Σ2 using hyps by simp
  have R1-def: R1 = map Subx.norm-unboxed R2 using hyps by simp
  have all-dyn-R2: list-all is-dyn-operand R2 using hyps by simp
  have F2-f: Fubx-get F2 f = Some fd2 using hyps by simp
  have map-of-fd2-l: map-of (body fd2) l = Some instrs using hyps by simp
  have sp-instrs-prefix: Subx.sp-instrs (map-option funtype o Fubx-get F2) (return
    fd2)
    (take pc instrs) [] (map typeof Σ2)
    using hyps by simp

  note next-instr2 = rel-fundefs-next-instr2[OF rel-F1-F2]
  note sp-instrs-prefix' =
    Subx.sp-instrs-singletonI[THEN Subx.sp-instrs-appendI[OF sp-instrs-prefix]]

  obtain fd1 where
    F1-f: Finca-get F1 f = Some fd1 and rel-fd1-fd2: rel-fundef (=) norm-eq fd1
    fd2
    using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto

  have wf-F2: Subx.wf-fundefs (Fubx-get F2)
    by (rule wf-state2[THEN Subx.wf-stateD, simplified])

  have wf-fd2: Subx.wf-fundef (map-option funtype o Fubx-get F2) fd2
    using F2-f wf-F2[THEN Subx.wf-fundefsD, THEN spec, of f]
    by simp

  have
    instrs-neq-Nil: instrs ≠ [] and
    all-jumps-in-range: list-all (Subx.jump-in-range (fst ` set (body fd2))) instrs
  and
    sp-instrs-instrs: Subx.sp-instrs (map-option funtype o Fubx-get F2) (return
    fd2) instrs [] []
    using list-all-map-of-SomeD[OF wf-fd2[THEN Subx.wf-fundef-all-wf-basic-blockD]
    map-of-fd2-l]
    by (auto dest: Subx.wf-basic-blockD)

  have sp-instrs-instrs': Subx.sp-instrs (map-option funtype o Fubx-get F2) (return
    fd2)
    (butlast instrs @ [instrs ! pc]) [] [] if pc-def: pc = length instrs - 1

```

```

unfolding pc-def last-conv-nth[OF instrs-neq-Nil, symmetric]
unfolding append-butlast-last-id[OF instrs-neq-Nil]
by (rule sp-instrs-instrs)

have sp-instr-last: Subx.sp-instr (map-option funtype o Fubx-get F2) (return
fd2)
  (instrs ! pc) (map typeof  $\Sigma 2$ ) [] if pc-def: pc = length instrs - 1
  using sp-instrs-instrs'[OF pc-def]
  using sp-instrs-prefix[unfolded pc-def butlast-conv-take[symmetric]]
  by (auto dest!: Subx.sp-instrs-appendD')

from list-all-map-of-SomeD[OF wf-fd2[THEN Subx.wf-fundef-all-wf-basic-blockD]
map-of-fd2-l]
  have is-jump-nthD:  $\bigwedge n$ . is-jump (instrs ! n)  $\implies n < \text{length } \textit{instrs} \implies n = \text{length } \textit{instrs} - 1$ 
  by (auto dest!: Subx.wf-basic-blockD
    list-all-butlast-not-nthD[of  $\lambda i. \neg \text{is-jump } i \wedge \neg \text{Ubx.instr.is-return } i$ ,
    simplified, OF - disjI])

note wf-s2' = Subx.wf-state-step-preservation[OF wf-state2 prems(3)]

from prems(3) show ?case
  using wf-s2'
proof (induction State F2 H (Frame f l pc R2  $\Sigma 2 \# st2$ ) s2' rule: Subx.step.induct)
  case (step-push d)
    let ?st1' = Frame f l (Suc pc) R1 (d #  $\Sigma 1$ ) # st1
    let ?s1' = State F1 H ?st1'
    show ?case (is  $\exists x. \text{?STEP } x \wedge \text{?MATCH } x (\text{State } F2 H ?st2')$ )
    proof (intro exI conjI)
      show ?STEP ?s1'
        using step-push.hyps
        by (auto intro!: Sinca.step-push dest: next-instr2)
  next
    have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
      using step-push.hyps rel-stacktraces-Cons
      using Subx.sp-instr.Push[THEN sp-instrs-prefix']
      by (auto simp: take-Suc-conv-app-nth
        intro!: rel-stacktraces.intros dest!: next-instrD instr-atD)
    thus ?MATCH ?s1' (State F2 H ?st2')
      using step-push.prem rel-F1-F2
      by (auto intro: match.intros)
  qed
next
  case (step-push-ubx1 n)
    let ?st1' = Frame f l (Suc pc) R1 (box-ubx1 n #  $\Sigma 1$ ) # st1
    let ?s1' = State F1 H ?st1'
    show ?case (is  $\exists x. \text{?STEP } x \wedge \text{?MATCH } x (\text{State } F2 H ?st2')$ )
    proof (intro exI conjI)
      show ?STEP ?s1'

```

```

using step-push-ubx1.hyps
by (auto intro!: Sinca.step-push dest: next-instr2)
next
have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
  using step-push-ubx1.hyps rel-stacktraces-Cons
  using Subx.sp-instr.PushUbx1[THEN sp-intrs-prefix']
  by (auto simp: take-Suc-conv-app-nth
        intro!: rel-stacktraces.intros dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H ?st2')
  using step-push-ubx1.prem rel-F1-F2
  by (auto intro!: match.intros)
qed
next
case (step-push-ubx2 b)
let ?st1' = Frame f l (Suc pc) R1 (box-ubx2 b #  $\Sigma$ ) # st1
let ?s1' = State F1 H ?st1'
show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH x (State F2 H ?st2'))
proof (intro exI conjI)
  show ?STEP ?s1'
    using step-push-ubx2.hyps
    by (auto intro!: Sinca.step-push dest: next-instr2)
next
have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
  using step-push-ubx2.hyps rel-stacktraces-Cons
  using Subx.sp-instr.PushUbx2[THEN sp-intrs-prefix']
  by (auto simp: take-Suc-conv-app-nth
        intro!: rel-stacktraces.intros dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H ?st2')
  using step-push-ubx2.prem rel-F1-F2
  by (auto intro!: match.intros)
qed
next
case (step-pop d  $\Sigma$ )
let ?st1' = Frame f l (Suc pc) R1 (map Subx.norm-unboxed  $\Sigma$ ) # st1
let ?s1' = State F1 H ?st1'
show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH x (State F2 H ?st2'))
proof (intro exI conjI)
  show ?STEP ?s1'
    unfolding  $\Sigma$ -def
    using step-pop.hyps
    by (auto intro!: Sinca.step-pop dest: next-instr2)
next
have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
  using step-pop.hyps rel-stacktraces-Cons
  by (auto simp: take-Suc-conv-app-nth
        intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.Pop
        dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H ?st2')
  using step-pop.prem rel-F1-F2

```

```

    by (auto intro!: match.intros)
qed
next
case (step-get n d)
let ?st1' = Frame f l (Suc pc) R1 (R1 ! n # map Subx.norm-unboxed Σ2)
# st1
let ?s1' = State F1 H ?st1'
show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2 H ?st2'))
proof (intro exI conjI)
show ?STEP ?s1'
unfolding Σ1-def R1-def
using step-get.hyps
by (auto intro!: Sinca.step-get dest: next-instr2)
next
have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
using step-get.hyps rel-stacktraces-Cons
by (auto simp: take-Suc-conv-app-nth
intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.Get
dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H ?st2')
using step-get.prems rel-F1-F2
by (auto intro!: match.intros)
qed
next
case (step-get-ubx-hit τ n d blob)
let ?st1' = Frame f l (Suc pc) R1 (R1 ! n # map Subx.norm-unboxed Σ2)
# st1
let ?s1' = State F1 H ?st1'
show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2 H ?st2'))
proof (intro exI conjI)
show ?STEP ?s1'
unfolding Σ1-def R1-def
using step-get-ubx-hit.hyps
by (auto intro!: Sinca.step-get dest: next-instr2)
next
have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
using step-get-ubx-hit.hyps rel-stacktraces-Cons
by (auto simp: take-Suc-conv-app-nth
intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.GetUbx
dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H ?st2')
using step-get-ubx-hit.prems rel-F1-F2
by (auto intro!: match.intros)
qed
next
case (step-get-ubx-miss τ n d F2')
hence R1 ! n = d
by (simp add: R1-def)
let ?st1' = Frame f l (Suc pc) R1 (R1 ! n # map Subx.norm-unboxed Σ2)

```

```

# st1
let ?s1' = State F1 H ?st1'
show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2' H ?st2'))
proof (intro exI conjI)
  show ?STEP ?s1'
    unfolding Σ1-def R1-def
    using step-get-ubx-miss.hyps
    by (auto intro!: Sinca.step-get dest: next-instr2)
next
  have rel-stacktraces (Fubx-get F2') None ?st1' ?st2'
    apply simp
  proof (rule rel-stacktraces.intros)
    show rel-stacktraces (Fubx-get F2') (Some f) st1 (Subx.box-stack f st2)
      unfolding step-get-ubx-miss.hyps
      using rel-st1-st2
      by (rule rel-stacktraces-map-entry-generalize-fundefI)
  next
    show Fubx-get F2' f = Some (Subx.generalize-fundef fd2)
      unfolding step-get-ubx-miss.hyps
      using F2-f
      by simp
  next
    show map-of (body (Subx.generalize-fundef fd2)) l = Some (map Subx.generalize-instr
      unfolding Subx.map-of-generalize-fundef-conv
      unfolding map-of-fd2-l
      by simp
  next
    show Subx.sp-intrs (map-option funtype o Fubx-get F2')
      (return (Subx.generalize-fundef fd2))
      (take (Suc pc) (map Subx.generalize-instr instrs))
      [] (map typeof (OpDyn d # map Subx.box-operand Σ2))
      using step-get-ubx-miss.hyps F2-f map-of-fd2-l
      by (auto simp: take-map take-Suc-conv-app-nth simp del: map-append
        intro!: sp-intrs-prefix'[THEN Subx.sp-intrs-generalize0] Subx.sp-instr.GetUbx
        dest!: next-instrD instr-atD)
    qed (insert R1-def all-dyn-R2 ⟨R1 ! n = d, simp-all)
    thus ?MATCH ?s1' (State F2' H ?st2')
      using step-get-ubx-miss.preds rel-F1-F2
      unfolding step-get-ubx-miss.hyps
      by (auto intro!: match.intros rel-fundefs-generalizeI)
  qed
next
  case (step-set n blob d R2' Σ2')
  let ?st1' = Frame fl (Suc pc) (map Subx.norm-unboxed R2') (map Subx.norm-unboxed
    Σ2') # st1
  let ?s1' = State F1 H ?st1'
  show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2 H ?st2'))
  proof (intro exI conjI)

```

```

show ?STEP ?s1'
  unfolding Σ1-def R1-def
  using step-set.hyps
  by (auto simp: map-update intro!: Sinca.step-set dest!: next-instr2)
next
  have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
    using step-set.hyps rel-stacktraces-Cons
    by (auto simp: take-Suc-conv-app-nth
      intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.Set
      intro: list-all-list-updateI
      dest!: next-instrD instr-atD)
  thus ?MATCH ?s1' (State F2 H ?st2')
    using step-set.preds rel-F1-F2
    by (auto intro!: match.intros)
qed
next
  case (step-set-ubx τ n blob d R2' Σ2')
  let ?st1' = Frame f l (Suc pc) (map Subx.norm-unboxed R2') (map Subx.norm-unboxed
Σ2') # st1
    let ?s1' = State F1 H ?st1'
    show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2 H ?st2'))
    proof (intro exI conjI)
      show ?STEP ?s1'
        unfolding Σ1-def R1-def
        using step-set-ubx.hyps
        by (auto simp: map-update intro!: Sinca.step-set dest!: next-instr2)
    next
      have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
        using step-set-ubx.hyps rel-stacktraces-Cons
        by (auto simp: take-Suc-conv-app-nth
          intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.SetUbx
          intro: list-all-list-updateI
          dest!: next-instrD instr-atD)
      thus ?MATCH ?s1' (State F2 H ?st2')
        using step-set-ubx.preds rel-F1-F2
        by (auto intro!: match.intros)
    qed
  next
    case (step-load x i i' d Σ2')
    let ?st1' = Frame f l (Suc pc) R1 (d # map Subx.norm-unboxed Σ2') # st1
    let ?s1' = State F1 H ?st1'
    show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2 H ?st2'))
    proof (intro exI conjI)
      show ?STEP ?s1'
        unfolding Σ1-def
        using step-load.hyps
        by (auto intro!: Sinca.step-load dest!: next-instr2)
    next
      have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'

```

```

using step-load.hyps rel-stacktraces-Cons
by (auto simp: take-Suc-conv-app-nth
      intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.Load
      dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H ?st2')
  using step-load.prem rel-F1-F2
  by (auto intro!: match.intros)
qed
next
case (step-load-ubx-hit  $\tau$  x i i' d blob  $\Sigma 2'$ )
let ?st1' = Frame f l (Suc pc) R1 (d # map Subx.norm-unboxed  $\Sigma 2'$ ) # st1
let ?s1' = State F1 H ?st1'
show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH x (State F2 H ?st2'))
proof (intro exI conjI)
  show ?STEP ?s1'
    unfolding  $\Sigma 1$ -def
    using step-load-ubx-hit.hyps
    by (auto intro!: Sinca.step-load dest!: next-instr2)
next
have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
  using step-load-ubx-hit.hyps rel-stacktraces-Cons
  by (auto simp: take-Suc-conv-app-nth
      intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.LoadUbx
      dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H ?st2')
  using step-load-ubx-hit.prem rel-F1-F2
  by (auto intro!: match.intros)
qed
next
case (step-load-ubx-miss  $\tau$  x i i' d F2'  $\Sigma 2'$ )
let ?st1' = Frame f l (Suc pc) R1 (d # map Subx.norm-unboxed  $\Sigma 2'$ ) # st1
let ?s1' = State F1 H ?st1'
show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH x (State F2' H ?st2'))
proof (intro exI conjI)
  show ?STEP ?s1'
    unfolding  $\Sigma 1$ -def
    using step-load-ubx-miss.hyps
    by (auto intro!: Sinca.step-load dest!: next-instr2)
next
have rel-stacktraces (Fubx-get F2') None ?st1' ?st2'
  apply simp
proof (rule rel-stacktraces.intros)
  show rel-stacktraces (Fubx-get F2') (Some f) st1 (Subx.box-stack f st2)
  unfolding step-load-ubx-miss
  using rel-st1-st2
  by (rule rel-stacktraces-map-entry-generalize-fundefI)
next
show Fubx-get F2' f = Some (Subx.generalize-fundef fd2)
  unfolding step-load-ubx-miss.hyps

```

```

  using F2-f by (simp add: Subx.map-of-generalize-fundef-conv)
next
  show map-of (body (Subx.generalize-fundef fd2)) l =
    Some (map Subx.generalize-instr instrs)
  unfolding Subx.map-of-generalize-fundef-conv
  using step-load-ubx-miss.hyps F2-f map-of-fd2-l
  by simp
next
  show Subx.sp-instrs (map-option funtype o Fubx-get F2') (return (Subx.generalize-fundef
fd2))
    (take (Suc pc) (map Subx.generalize-instr instrs))
    [] (map typeof (OpDyn d # map Subx.box-operand Σ2'))
  using step-load-ubx-miss.hyps F2-f map-of-fd2-l
  by (auto simp: take-map take-Suc-conv-app-nth simp del: map-append
intro!: sp-instrs-prefix'[THEN Subx.sp-instrs-generalize0] Subx.sp-instr.LoadUbx
dest!: next-instrD instr-atD)
qed (insert all-dyn-R2, simp-all add: R1-def)
thus ?MATCH ?s1' (State F2' H ?st2')
  using step-load-ubx-miss.prem rel-F1-F2
  unfolding step-load-ubx-miss.hyps
  by (auto intro: match.intros rel-fundefs-generalizeI)
qed
next
  case (step-store x i i' y d H' Σ2')
  let ?st1' = Frame f l (Suc pc) R1 (map Subx.norm-unboxed Σ2') # st1
  let ?s1' = State F1 H' ?st1'
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x (State F2 H' ?st2'))
  proof (intro exI conjI)
    show ?STEP ?s1'
      unfolding Σ1-def
      using step-store.hyps
      by (auto intro: Sinca.step-store dest!: next-instr2)
  next
    have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
      using step-store.hyps rel-stacktraces-Cons
      by (auto simp: take-Suc-conv-app-nth
intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.Store
dest!: next-instrD instr-atD)
    thus ?MATCH ?s1' (State F2 H' ?st2')
      using step-store.prem rel-F1-F2
      by (auto intro: match.intros)
  qed
next
  case (step-store-ubx τ x i i' blob d H' Σ2')
  let ?st1' = Frame f l (Suc pc) R1 (map Subx.norm-unboxed Σ2') # st1
  let ?s1' = State F1 H' ?st1'
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x (State F2 H' ?st2'))
  proof (intro exI conjI)
    show ?STEP ?s1'

```

```

unfolding  $\Sigma_1\text{-def}$ 
using step-store-ubx.hyps
by (auto intro: Sinca.step-store dest!: next-instr2)
next
have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
using step-store-ubx.hyps rel-stacktraces-Cons
by (auto simp: take-Suc-conv-app-nth
    intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.StoreUbx
    dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H' ?st2')
using step-store-ubx.prem rel-F1-F2
by (auto intro: match.intros)
qed
next
case (step-op op ar  $\Sigma_2'$  x)
let ?st1' = Frame f l (Suc pc) R1 (x # drop ar (map Subx.norm-unboxed
 $\Sigma_2$ )) # st1
let ?s1' = State F1 H ?st1'
show ?case (is  $\exists x. ?STEP x \wedge ?MATCH x (State F2 H ?st2')$ )
proof (intro exI conjI)
show ?STEP ?s1'
unfolding  $\Sigma_1\text{-def}$ 
using step-op.hyps
by (auto simp: take-map ap-map-list-cast-Dyn-to-map-norm[symmetric]
    intro!: Sinca.step-op dest!: next-instr2)
next
have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
using step-op.hyps rel-stacktraces-Cons
by (auto simp: take-Suc-conv-app-nth drop-map map-eq-append-map-drop
    simp: ap-map-list-cast-Dyn-map-typeof-replicate-conv
    intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.Op
    dest!: next-instrD instr-atD)
thus ?MATCH ?s1' (State F2 H ?st2')
using step-op.prem rel-F1-F2
by (auto intro: match.intros)
qed
next
case (step-op-inl op ar  $\Sigma_2'$  opinl x F2')
let ?F1' = Sinca.Fenv.map-entry F1 f ( $\lambda fd. rewrite-fundef-body fd l pc$ 
(Inca.IOpInl opinl))
let ?st1' = Frame f l (Suc pc) R1 (x # drop ar (map Subx.norm-unboxed
 $\Sigma_2$ )) # st1
let ?s1' = State ?F1' H ?st1'
show ?case (is  $\exists x. ?STEP x \wedge ?MATCH x (State F2' H ?st2')$ )
proof (intro exI conjI)
show ?STEP ?s1'
unfolding  $\Sigma_1\text{-def}$ 
using step-op-inl.hyps
by (auto simp: take-map ap-map-list-cast-Dyn-to-map-norm[symmetric])

```

```

    intro!: Sinca.step-op-inl dest!: next-instr2)
next
  show ?MATCH ?s1' (State F2' H ?st2')
    using step-op-inl.prem
  proof (rule match.intros)
    show rel-fundefs (Finca-get ?F1') (Fubx-get F2')
      unfolding step-op-inl.hyps
      using rel-F1-F2
      by (auto intro: rel-fundefs-rewriteI)
next
  let ?fd2' = rewrite-fundef-body fd2 l pc (Ubx.instr.IOpInl opinl)
  let ?instrs' = instrs[pc := Ubx.instr.IOpInl opinl]
  show rel-stacktraces (Fubx-get F2') None ?st1' ?st2'
  proof (rule rel-stacktraces.intros)
    show rel-stacktraces (Fubx-get F2') (Some f) st1 st2
      using step-op-inl.hyps rel-st1-st2 Sinca.Jnl-invertible
      by (auto simp: Subx.sp-instr-Op-OpInl-conv
           intro: rel-stacktraces-map-entry-rewrite-fundef-body)
next
  show Fubx-get F2' f = Some ?fd2'
    unfolding step-op-inl.hyps
    using F2-f by simp
next
  show map-of (body ?fd2') l = Some ?instrs'
    using map-of-fd2-l by simp
next
  show Subx.sp-instrs (map-option funtype o Fubx-get F2')
    (return (rewrite-fundef-body fd2 l pc (Ubx.instr.IOpInl opinl)))
     (take (Suc pc) ?instrs') [] (map typeof (OpDyn x # drop ar Σ2)))
    using rel-stacktraces-Cons step-op-inl.hyps
  by (auto simp add: take-Suc-conv-app-nth Subx.Fenv.map-option-comp-map-entry
       map-eq-append-map-drop ap-map-list-cast-Dyn-map-typeof-replicate-conv
       Sinca.Jnl-invertible
       intro!: sp-instrs-prefix' Subx.sp-instr.OpInl
       dest!: next-instrD instr-atD)
    qed (insert all-dyn-R2 R1-def, simp-all add: drop-map)
  qed
qed
next
  case (step-op-inl-hit opinl ar Σ2' x)
  let ?st1' = Frame f l (Suc pc) R1 (x # drop ar (map Subx.norm-unboxed
Σ2)) # st1
  let ?s1' = State F1 H ?st1'
  show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2 H ?st2'))
  proof (intro exI conjI)
    show ?STEP ?s1'
      unfolding Σ1-def
      using step-op-inl-hit.hyps
      by (auto simp: take-map ap-map-list-cast-Dyn-to-map-norm[symmetric])

```

```

    intro!: Sinca.step-op-inl-hit dest!: next-instr2)
next
  show ?MATCH ?s1' (State F2 H ?st2')
    using step-op-inl-hit.prems rel-F1-F2
  proof (rule match.intros)
    show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
      using step-op-inl-hit.hyps rel-stacktraces-Cons
    by (auto simp: take-Suc-conv-app-nth drop-map map-eq-append-map-drop
          simp: ap-map-list-cast-Dyn-map-typeof-replicate-conv
          intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.OpInl
          dest!: next-instrD instr-atD)
    qed
  qed
next
  case (step-op-inl-miss opinl ar Σ2' x F2')
    let ?F1' = Sinca.Fenv.map-entry F1 f (λfd. rewrite-fundef-body fd l pc
    (Inca.IOp (DeInl opinl)))
    let ?st1' = Frame f l (Suc pc) R1 (x # drop ar (map Subx.norm-unboxed
    Σ2)) # st1
    let ?s1' = State ?F1' H ?st1'
    show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2' H ?st2'))
    proof (intro exI conjI)
      show ?STEP ?s1'
        unfolding Σ1-def
        using step-op-inl-miss.hyps
      by (auto simp: take-map ap-map-list-cast-Dyn-to-map-norm[symmetric]
            intro!: Sinca.step-op-inl-miss dest!: next-instr2)
next
  show ?MATCH ?s1' (State F2' H ?st2')
    using step-op-inl-miss.prems
  proof (rule match.intros)
    show rel-fundefs (Finca-get ?F1') (Fubx-get F2')
      unfolding step-op-inl-miss.hyps
      using rel-F1-F2
      by (auto intro: rel-fundefs-rewriteI)
next
  let ?fd2' = rewrite-fundef-body fd2 l pc (Ubx.instr.IOp (DeInl opinl))
  let ?intrs' = intrs[pc := Ubx.instr.IOp (DeInl opinl)]
  show rel-stacktraces (Fubx-get F2') None ?st1' ?st2'
  proof (rule rel-stacktraces.intros)
    show rel-stacktraces (Fubx-get F2') (Some f) st1 st2
      using step-op-inl-miss.hyps rel-st1-st2 Sinca.Inl-invertible
      by (auto intro: rel-stacktraces-map-entry-rewrite-fundef-body
            Subx.sp-instr-Op-OpInl-conv[OF refl, symmetric])
next
  show Fubx-get F2' f = Some ?fd2'
    unfolding step-op-inl-miss.hyps
    using F2-f by simp
next

```

```

show map-of (body ?fd2') l = Some ?instrs'
  using map-of-fd2-l by simp
next
  show Subx.sp-instrs (map-option funtype o Fubx-get F2')
    (return (rewrite-fundef-body fd2 l pc (Ubx.instr.IOp (DeJnl opinl))))
    (take (Suc pc) ?instrs') [] (map typeof (OpDyn x # drop ar Σ2))
  using rel-stacktraces-Cons step-op-inl-miss.hyps
by (auto simp add: take-Suc-conv-app-nth Subx.Fenv.map-option-comp-map-entry
    map-eq-append-map-drop ap-map-list-cast-Dyn-map-typeof-replicate-conv
    Sinca.Jnl-invertible
    intro!: sp-instrs-prefix' Subx.sp-instr.Op
    dest!: next-instrD instr-atD)
qed (insert all-dyn-R2 R1-def, simp-all add: drop-map)
qed
qed
next
case (step-op-ubx opubx op ar x)
let ?st1' = Frame f l (Suc pc) R1 (Subx.norm-unboxed x # drop ar (map
Subx.norm-unboxed Σ2)) # st1
let ?s1' = State F1 H ?st1'
show ?case (is ∃ x. ?STEP x ∧ ?MATCH x (State F2 H ?st2'))
proof (intro exI conjI)
  show ?STEP ?s1'
    unfolding Σ1-def
    using step-op-ubx.hyps
    by (auto simp: take-map
        intro!: Sinca.step-op-inl-hit
        intro: Subx.UbxOp-to-Jnl[THEN Sinca.Jnl-IsJnl] Subx.UbxOp-correct
        dest: next-instr2)
next
show ?MATCH ?s1' (State F2 H ?st2')
  using step-op-ubx.prem rel-F1-F2
proof (rule match.intros)
  show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
    using step-op-ubx.hyps rel-stacktraces-Cons
  by (auto simp: take-Suc-conv-app-nth drop-map map-eq-append-map-drop
      intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.Op Ubx
      dest!: next-instrD instr-atD
      dest!: Subx.TypeOp-complete)
qed
qed
next
case (step-cjump lt lf x d l' Σ2')
hence instr-at fd2 l pc = Some (Ubx.instr.ICJump lt lf)
  using F2-f by (auto dest!: next-instrD)
hence pc-in-dom: pc < length instrs AND nth-instrs-pc: instrs ! pc = Ubx.instr.ICJump
lt lf
  using map-of-fd2-l by (auto dest!: instr-atD)
hence {lt, lf} ⊆ fst ` set (body fd2)

```

```

using all-jumps-in-range by (auto simp: list-all-length)
moreover have  $l' \in \{l_t, l_f\}$ 
  using step-cjump.hyps by auto
ultimately have  $l' \in \text{fst}^{\text{'}} \text{set}(\text{body } fd2)$ 
  by blast
then obtain  $\text{instrs}'$  where  $\text{map-of-}l': \text{map-of } (\text{body } fd2) l' = \text{Some } \text{instrs}'$ 
  by (auto dest: weak-map-of-SomeI)

have  $pc\text{-def}: pc = \text{length } \text{instrs} - 1$ 
  using is-jump-nthD[ $OF - pc\text{-in-dom}$ ] nth-intrs-pc by simp
have  $\Sigma 2'\text{-eq-Nil}: \Sigma 2' = []$ 
  using sp-instr-last[ $OF pc\text{-def}$ ] step-cjump.hyps
  by (auto simp: nth-intrs-pc elim!: Subx.sp-instr.cases)

let ?st1' = Frame f l' 0 R1 (map Subx.norm-unboxed  $\Sigma 2'$ ) # st1
let ?s1' = State F1 H ?st1'
show ?case (is  $\exists x. ?STEP x \wedge ?MATCH x (\text{State } F2 H ?st2')$ )
proof (intro exI conjI)
  show ?STEP ?s1'
    unfolding  $\Sigma 1\text{-def}$ 
    using step-cjump.hyps
    by (auto intro!: Sinca.step-cjump dest: next-instr2)
next
show ?MATCH ?s1' (State F2 H ?st2')
  using step-cjump.prem rel-F1-F2
proof (rule match.intros)
  show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
    using map-of-l' rel-stacktraces-Cons(1,3-5)
    by (auto simp:  $\Sigma 2'\text{-eq-Nil}$  intro!: rel-stacktraces.intros intro: Subx.sp-intrs.Nil)
qed
qed
next
case (step-call g gd2 frame_g)
then obtain gd1 where
   $F1\text{-}g: Finca\text{-get } F1 g = \text{Some } gd1 \text{ and } rel\text{-}gd1\text{-}gd2: rel\text{-}fundef (=) norm\text{-}eq$ 
  gd1 gd2
  using rel-fundefs-Some2[ $OF rel\text{-}F1\text{-}F2$ ] by auto

have wf-gd2: Subx.wf-fundef (map-option funtype o Fubx-get F2) gd2
  using Subx.wf-fundefs-getD[ $OF wf\text{-}F2$ ] step-call.hyps by simp

obtain intrs_g where gd2-fst-bblock:  $\text{map-of } (\text{body } gd2) (\text{fst } (\text{hd } (\text{body } gd2)))$ 
= Some intrs_g
  using Subx.wf-fundef-body-neq-NilD[ $OF wf\text{-}gd2$ ]
  by (metis hd-in-set map-of-eq-None-iff not-Some-eq prod.collapse prod-in-set-fst-image-conv)

let ?frame_g = allocate-frame g gd1 (take (arity gd1)  $\Sigma 1$ ) uninitialized
let ?st1' = ?frame_g # Frame f l pc R1  $\Sigma 1$  # st1
let ?s1' = State F1 H ?st1'

```

```

show ?case (is ∃x. ?STEP x ∧ ?MATCH x (State F2 H ?st2'))
proof (intro exI conjI)
  show ?STEP ?s1'
    unfolding Σ1-def
    using step-call.hyps F1-g rel-gd1-gd2
    by (auto simp: rel-fundef-arities intro!: Sinca.step-call dest: next-instr2)
next
  show ?MATCH ?s1' (State F2 H ?st2')
    using step-call.prem rel-F1-F2
  proof (rule match.intros)
    have FOO: fst (hd (body gd1)) = fst (hd (body gd2))
      apply (rule rel-fundef-rel-fst-hd-bodies[OF rel-gd1-gd2])
      using Subx.wf-fundefs-getD[OF wf-F2 ⟨Fubx-get F2 g = Some gd2⟩]
      by (auto dest: Subx.wf-fundef-body-neq-NilD)
    show rel-stacktraces (Fubx-get F2) None ?s1' ?st2'
      unfolding step-call allocate-frame-def FOO
    proof (rule rel-stacktraces.intros(2))
      show rel-stacktraces (Fubx-get F2) (Some g)
        (Frame f l pc R1 Σ1 # st1) (Frame f l pc R2 Σ2 # st2)
        using step-call rel-stacktraces-Cons
        by (auto simp: is-valid-fun-call-def intro: rel-stacktraces.intros)
    next
      show take (arity gd1) Σ1 @ replicate (fundef-locals gd1) uninitialized =
        map Subx.norm-unboxed (take (arity gd2) Σ2 @
        replicate (fundef-locals gd2) (OpDyn uninitialized))
        using rel-gd1-gd2
        by (simp add: rel-fundef-arities rel-fundef-locals take-map Σ1-def)
    next
      show list-all is-dyn-operand (take (arity gd2) Σ2 @
        replicate (fundef-locals gd2) (OpDyn uninitialized))
        using step-call.hyps by auto
      qed (insert step-call gd2-fst-bblock, simp-all add: Subx.sp-intrs.Nil)
    qed
  qed
next
  case (step-return fd2' Σ2_g frame_g' g l_g pc_g R2_g st2')
  hence fd2-fd2'[simp]: fd2' = fd2
    using F2-f by simp
  then obtain fd1 where
    F1-f: Finca-get F1 f = Some fd1 and rel-fd1-fd2: rel-fundef (=) norm-eq
  fd1 fd2
    using F2-f rel-fundefs-Some2[OF rel-F1-F2] by auto
  show ?case
  using rel-st1-st2 unfolding ⟨Frame g l_g pc_g R2_g Σ2_g # st2' = st2⟩[symmetric]
  proof (cases rule: rel-stacktraces.cases)
    case (rel-stacktraces-Cons st1' Σ1_g R1_g gd2 instrs)
    hence is-valid-call-f: is-valid-fun-call (Fubx-get F2) g l_g pc_g Σ2_g f
      by simp
    let ?s1' = State F1 H (Frame g l_g (Suc pc_g) R1_g (Σ1 @ drop (arity fd2)

```

```

 $\Sigma 1_g) \# st1'$ 
  show ?thesis (is  $\exists x. ?STEP x \wedge ?MATCH x (State F2 H ?st2')$ )
  proof (intro exI conjI)
    show ?STEP ?s1'
      unfolding rel-stacktraces-Cons
      proof (rule Sinca.step.step-return)
        show next-instr (Finca-get F1) f l pc = Some Inca.instr.IReturn
        using <next-instr (Fubx-get F2) f l pc = Some Ubx.instr.IReturn>
        using rel-fundefs-next-instr2[OF rel-F1-F2]
        by force
    next
      show Finca-get F1 f = Some fd1
      by (rule F1-f)
    qed (insert step-return.hyps rel-fd1-fd2,
           simp-all add:  $\Sigma 1$ -def rel-fundef-arithes rel-fundef-return)
  next
    show ?MATCH ?s1' (State F2 H ?st2')
    unfolding step-return.hyps
    proof (rule match.intros)
      have next-instr (Fubx-get F2) g lg pcg = Some (Ubx.instr.ICall f) and
        arity fd2  $\leq$  length  $\Sigma 2_g$  and list-all is-dyn-operand (take (arity fd2)
 $\Sigma 2_g)$ 
      using is-valid-call-f[unfolded is-valid-fun-call-def] F2-f
      by simp-all
      hence
        pcg-in-range: pcg  $<$  length instrs and
        nth-instrs-pcg; instrs ! pcg = Ubx.instr.ICall f
        using rel-stacktraces-Cons
        by (auto dest!: next-instrD instr-atD)
        have replicate-None: replicate (arity fd2) None = map typeof (take (arity
fd2)  $\Sigma 2_g)$ 
        using <arity fd2  $\leq$  length  $\Sigma 2_g$  > <list-all is-dyn-operand (take (arity fd2)
 $\Sigma 2_g)$ >
        by (auto simp: is-dyn-operand-eq-typeof list-all-intro!: replicate-eq-map)
        show rel-stacktraces (Fubx-get F2) None
          (Frame g lg (Suc pcg) R1g ( $\Sigma 1$  @ drop (arity fd2)  $\Sigma 1_g) \# st1'$ )
          (Frame g lg (Suc pcg) R2g ( $\Sigma 2$  @ drop (arity fd2)  $\Sigma 2_g) \# st2'$ )
        using rel-stacktraces-Cons
        apply (auto simp:  $\Sigma 1$ -def drop-map take-Suc-conv-app-nth[OF
pcg-in-range] nth-instrs-pcg
                     intro!: rel-stacktraces.intros elim!: Subx.sp-instrs-appendI)
        apply (rule Subx.sp-instr.Call[of - - - - map typeof (drop (arity fd2)
 $\Sigma 2_g))])
        apply (simp add: F2-f functype-def)
        apply (simp add: replicate-None map-append[symmetric])
        using <length  $\Sigma 2$  = return fd2'> <list-all is-dyn-operand  $\Sigma 2$ >
        by (auto simp: list.pred-set intro: replicate-eq-map[symmetric])
    qed (insert step-return rel-F1-F2, simp-all)
  qed$ 
```

```

qed
qed
qed
qed

lemma match-final-backward:
  assumes match s1 s2 and final-s2: final Fubx-get Ubx.IReturn s2
  shows final Finca-get Inca.IReturn s1
  using ⟨match s1 s2⟩
proof (cases s1 s2 rule: match.cases)
  case (matchI F2 H st2 F1 st1)
  show ?thesis
  using final-s2[unfolded matchI]
proof (cases - - State F2 H st2 rule: final.cases)
  case (finalI f l pc R Σ)
  then show ?thesis
  using matchI
  by (auto intro!: final.intros elim!: rel-stacktraces.cases dest: rel-fundefs-next-instr2)
qed
qed

sublocale inca-to-ubx-simulation: backward-simulation where
  step1 = Sinca.step and final1 = final Finca-get Inca.IReturn and
  step2 = Subx.step and final2 = final Fubx-get Ubx.IReturn and
  match = λ_. match and order = λ_. False
  using match-final-backward backward-lockstep-simulation
  lockstep-to-plus-backward-simulation[of match Subx.step Sinca.step]
  by unfold-locales auto

```

20 Forward simulation

```

lemma ap-map-list-cast-Dyn-eq-norm-stack:
  assumes list-all (λx. x = None) (map typeof xs)
  shows ap-map-list cast-Dyn xs = Some (map Subx.norm-unboxed xs)
  using assms
proof (induction xs)
  case Nil
  thus ?case by simp
next
  case (Cons x xs)
  from Cons.prem have
    typeof-x: typeof x = None and
    typeof-xs: list-all (λx. x = None) (map typeof xs)
    by simp-all
  obtain x' where x = OpDyn x'
  using typeof-unboxed-inversion(1)[OF typeof-x] by auto
  then show ?case
  using Cons.IH[OF typeof-xs]
  by simp

```

qed

```

lemma forward-lockstep-simulation:
  assumes match s1 s2 and Sinca.step s1 s1'
  shows  $\exists s2'. Subx.step s2 s2' \wedge \text{match } s1' s2'$ 
  using assms(1)
proof (cases s1 s2 rule: match.cases)
  case (matchI F2 H st2 F1 st1)
  have s2-def:  $s2 = \text{Global.state.State } F2 H st2$  using matchI by simp
  have rel-F1-F2: rel-fundefs (Finca-get F1) (Fubx-get F2) using matchI by simp
  have wf-s2: Subx.wf-state s2 using matchI by simp
  hence wf-F2: Subx.wf-fundefs (Fubx-get F2) by (auto simp: s2-def dest: Subx.wf-stateD)

  note wf-s2'I = Subx.wf-state-step-preservation[OF wf-s2]

  from ⟨rel-stacktraces (Fubx-get F2) None st1 st2⟩ show ?thesis
  proof (cases Fubx-get F2 None :: 'fun option st1 st2 rule: rel-stacktraces.cases)
    case rel-stacktraces-Nil
    with matchI assms(2) show ?thesis by (auto elim: Sinca.step.cases)
    next
      case (rel-stacktraces-Cons f st1' st2' Σ1 Σ2 R1 R2 fd2 l instrs pc)
      have rel-st1'-st2': rel-stacktraces (Fubx-get F2) (Some f) st1' st2'
        using rel-stacktraces-Cons by simp
      have st2-def:  $st2 = \text{Frame } f l pc R2 \Sigma2 \# st2'$  using rel-stacktraces-Cons by simp
        have Σ1-def:  $\Sigma1 = \text{map Subx.norm-unboxed } \Sigma2$  using rel-stacktraces-Cons by simp
        have all-dyn-R2: list-all is-dyn-operand R2 using rel-stacktraces-Cons by simp
        have F2-f: Fubx-get F2 f = Some fd2 using rel-stacktraces-Cons by simp
        have map-of-fd2-l: map-of (body fd2) l = Some instrs using rel-stacktraces-Cons by simp
        have sp-instrs-prefix: Subx.sp-instrs (map-option funtype o Fubx-get F2) (return fd2)
          (take pc instrs) [] (map typeof Σ2)
          using rel-stacktraces-Cons by simp
        note sp-instrs-prefix' =
          Subx.sp-instrs-singletonI[THEN Subx.sp-instrs-appendI[OF sp-instrs-prefix]]

        have wf(fd2): Subx.wf-fundef (map-option funtype o Fubx-get F2) fd2
          using wf-F2 F2-f by (auto dest: Subx.wf-fundefs-getD)
        hence sp-instrs-instrs:
          Subx.sp-instrs (map-option funtype o Fubx-get F2) (return fd2) instrs [] []
          using wf-fd2[THEN Subx.wf-fundef-all-wf-basic-blockD] map-of-fd2-l
          by (auto dest: list-all-map-of-SomeD[OF - map-of-fd2-l] dest: Subx.wf-basic-blockD)
        hence sp-instrs-sufix: Subx.sp-instrs (map-option funtype o Fubx-get F2) (return fd2)
          (instrs ! pc # drop (Suc pc) instrs) (map typeof Σ2) [] if pc < length instrs
          using that Subx.sp-instrs-appendD'[OF - sp-instrs-prefix]
          by (simp add: Cons-nth-drop-Suc)

```

```

have
  instrs-neq-Nil: instrs  $\neq \emptyset$  and
  all-jumps-in-range: list-all (Subx.jump-in-range (fst ` set (body fd2))) instrs
and
  sp-instrs-instrs: Subx.sp-instrs (map-option funtype  $\circ$  Fubx-get F2) (return
fd2) instrs  $\emptyset$  []
  using list-all-map-of-SomeD[OF wf-fd2[THEN Subx.wf-fundef-all-wf-basic-blockD]
map-of-fd2-l]
  by (auto dest: Subx.wf-basic-blockD)

from assms(2)[unfolded matchI rel-stacktraces-Cons] show ?thesis
proof (induction State F1 H (Frame f l pc (map Subx.norm-unboxed R2) (map
Subx.norm-unboxed  $\Sigma 2$ ) # st1') s1' rule: Sinca.step.induct)
case (step-push d)
then obtain instr2 where
  next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
  norm-eq-instr1-instr2: norm-eq (Inca.IPush d) instr2
  by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
then show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH (State F1 H ?st1') x)
proof (cases instr2)
  case (IPush d2)
  let ?st2' = Frame f l (Suc pc) R2 (OpDyn d #  $\Sigma 2$ ) # st2'
  let ?s2' = State F2 H ?st2'
  show ?thesis
  proof (intro exI conjI)
    show ?STEP s2'
    using next-instr2 norm-eq-instr1-instr2
    unfolding IPush
    by (auto simp: s2-def st2-def intro: Subx.step-push)
next
  show ?MATCH (State F1 H ?st1') ?s2'
  proof (rule match.intros)
    show Subx.wf-state s2'
    using wf-F2 by (auto intro: Subx.wf-stateI)
next
  show rel-stacktraces (Fubx-get F2) None st1' st2'
  using next-instr2 rel-stacktraces-Cons
  unfolding IPush
  by (auto simp: next-instr-take-Suc-conv
    intro!: rel-stacktraces.intros sp-instrs-prefix' intro: Subx.sp-instr.Push)
  qed (simp-all add: rel-F1-F2)
qed
next
  case (IPushUbx1 u)
  let ?st2' = Frame f l (Suc pc) R2 (OpUbx1 u #  $\Sigma 2$ ) # st2'
  let ?s2' = State F2 H ?st2'
  show ?thesis
  proof (intro exI conjI)

```

```

show ?STEP ?s2'
  using next-instr2 norm-eq-instr1-instr2
  unfolding IPushUbx1
  by (auto simp: s2-def st2-def intro: Subx.step-push-ubx1)
next
  show ?MATCH (State F1 H ?st1') ?s2'
  proof (rule match.intros)
    show Subx.wf-state ?s2'
      using wf-F2 by (auto intro: Subx.wf-stateI)
    show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
      using next-instr2 norm-eq-instr1-instr2 rel-stacktraces-Cons
      unfolding IPushUbx1
      by (auto simp: next-instr-take-Suc-conv
        intro!: rel-stacktraces.intros sp-instrs-prefix' intro: Subx.sp-instr.PushUbx1)
    qed (simp-all add: rel-F1-F2)
  qed
next
  case (IPushUbx2 u)
  let ?st2' = Frame f l (Suc pc) R2 (OpUbx2 u # Σ2) # st2'
  let ?s2' = State F2 H ?st2'
  show ?thesis
  proof (intro exI conjI)
    show ?STEP ?s2'
      using next-instr2 norm-eq-instr1-instr2
      unfolding IPushUbx2
      by (auto simp: s2-def st2-def intro: Subx.step-push-ubx2)
next
  show ?MATCH (State F1 H ?st1') ?s2'
  proof (rule match.intros)
    show Subx.wf-state ?s2'
      using wf-F2 by (auto intro: Subx.wf-stateI)
    show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
      using next-instr2 norm-eq-instr1-instr2 rel-stacktraces-Cons
      unfolding IPushUbx2
      by (auto simp: next-instr-take-Suc-conv
        intro!: rel-stacktraces.intros sp-instrs-prefix' intro: Subx.sp-instr.PushUbx2)
    qed (simp-all add: rel-F1-F2)
  qed
qed simp-all
next
  case (step-pop d Σ1')
  then obtain u Σ2' where
    Σ2-def: Σ2 = u # Σ2' and d = Subx.norm-unboxed u and
    Σ1'-def: Σ1' = map Subx.norm-unboxed Σ2'
    by auto
  from step-pop obtain instr2 where
    next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
    norm-eq-instr1-instr2: norm-eq Inca.IPop instr2
    by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])

```

```

then show ?case (is  $\exists x. \ ?STEP\ x \wedge \ ?MATCH\ (State\ F1\ H\ ?st1')\ x$ )
proof (cases instr2)
  case IPop
  let ?st2' = Frame f l (Suc pc) R2  $\Sigma 2' \# st2'$ 
  let ?s2' = State F2 H ?st2'
  show ?thesis
  proof (intro exI conjI)
    show ?STEP ?s2'
    using next-instr2 norm-eq-instr1-instr2
    unfolding IPop
    by (auto simp: s2-def st2-def  $\Sigma 2$ -def intro: Subx.step-pop)
  next
    show ?MATCH (State F1 H ?st1') ?s2'
    proof (rule match.intros)
      show Subx.wf-state ?s2'
      using wf-F2 by (auto intro: Subx.wf-stateI)
    next
      show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
      using next-instr2 rel-stacktraces-Cons
      unfolding IPop
      by (auto simp:  $\Sigma 2$ -def  $\Sigma 1$ '-def next-instr-take-Suc-conv
           intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.Pop)
      qed (simp-all add: rel-F1-F2)
    qed
    qed simp-all
  next
    case (step-get n d)
    hence nth-R2-n:  $R2 ! n = OpDyn\ d$ 
    using all-dyn-R2
    by (metis Subx.norm-unboxed.simps(1) is-dyn-operand-def length-map
        list-all-length nth-map)
    from step-get obtain instr2 where
      next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
      norm-eq-instr1-instr2: norm-eq (Inca.IGet n) instr2
      by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
    then show ?case (is  $\exists x. \ ?STEP\ x \wedge \ ?MATCH\ (State\ F1\ H\ ?st1')\ x$ )
    proof (cases instr2)
      case (IGet n')
      hence n' = n using norm-eq-instr1-instr2 by simp
      let ?st2' = Frame f l (Suc pc) R2 (OpDyn d #  $\Sigma 2$ ) # st2'
      let ?s2' = State F2 H ?st2'
      show ?thesis
      proof (intro exI conjI)
        show ?STEP ?s2'
        using step-get nth-R2-n
        using next-instr2 norm-eq-instr1-instr2
        unfolding IGet
        by (auto simp: s2-def st2-def intro: Subx.step-get)
      next

```

```

show ?MATCH (State F1 H ?st1') ?s2'
proof (rule match.intros)
  show Subx.wf-state ?s2'
    using wf-F2 by (auto intro: Subx.wf-stateI)
next
  show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
    using next-instr2 rel-stacktraces-Cons
    unfolding IGet
    by (auto simp: next-instr-take-Suc-conv
        intro!: rel-stacktraces.intros sp-instrs-prefix' intro: Subx.sp-instr.Get)
  qed (simp-all add: rel-F1-F2)
qed
next
case (IGetUbx τ n')
hence n' = n using norm-eq-instr1-instr2 by simp
show ?thesis
proof (cases Subx.unbox τ d)
  case None
  let ?F2' = Subx.Fenv.map-entry F2 f Subx.generalize-fundef
  let ?st2' = Subx.box-stack f (Frame f l (Suc pc) R2 (OpDyn d # Σ2) #
st2')
  let ?s2' = State ?F2' H ?st2'
  show ?thesis
  proof (intro exI conjI)
    show ?STEP ?s2'
      using step-get nth-R2-n None
      using next-instr2 norm-eq-instr1-instr2
      unfolding IGetUbx
      by (auto simp: s2-def st2-def intro: Subx.step-get-ubx-miss[simplified])
  next
    show ?MATCH (State F1 H ?st1') ?s2'
    proof (rule match.intros)
      show Subx.wf-state ?s2'
        using wf-F2
        by (auto intro!: Subx.wf-stateI intro: Subx.wf-fundefs-generalize)
    next
      show rel-fundefs (Finca-get F1) (Fubx-get ?F2')
        using rel-F1-F2
        by (auto intro: rel-fundefs-generalizeI)
    next
      have sp-instrs-gen: Subx.sp-instrs (map-option funtype o Fubx-get F2)
      (return fd2)
        (take (Suc pc) (map Subx.generalize-instr instrs)) [] (map Map.empty
        (OpDyn d # Σ2))
        using rel-stacktraces-Cons step-get.hyps
        using IGetUbx ⟨n' = n⟩
        using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
        by (auto simp: take-Suc-conv-app-nth take-map
            intro!: Subx.sp-instrs-appendI

```

```

    intro: Subx.sp-instrs-generalize0 Subx.sp-instr.Get)
then show rel-stacktraces (Fubx-get ?F2') None ?st1' ?st2'
    apply simp
    proof (rule rel-stacktraces.intros)
        show rel-stacktraces (Fubx-get ?F2') (Some f) st1' (Subx.box-stack f
st2')
        using rel-st1'-st2' rel-stacktraces-map-entry-generalize-fundefI by
simp
next
    show Fubx-get ?F2' f = Some (Subx.generalize-fundef fd2)
    using F2-f by simp
next
    show map-of (body (Subx.generalize-fundef fd2)) l =
Some (map Subx.generalize-instr instrs)
    using map-of-fd2-l by (simp add: Subx.map-of-generalize-fundef-conv)
qed (insert all-dyn-R2 map-of-fd2-l, simp-all add: Subx.map-of-generalize-fundef-conv)
qed
qed
next
case (Some u)
let ?st2' = Frame f l (Suc pc) R2 (u # Σ2) # st2'
let ?s2' = State F2 H ?st2'
show ?thesis
proof (intro exI conjI)
    show ?STEP ?s2'
        using step-get nth-R2-n Some
        using next-instr2 norm-eq-instr1-instr2
        unfolding IGetUbx
        by (auto simp: s2-def st2-def intro: Subx.step-get-ubx-hit)
next
    show ?MATCH (State F1 H ?st1') ?s2'
    proof (rule match.intros)
        show Subx.wf-state ?s2'
            using wf-F2 by (auto intro: Subx.wf-stateI)
next
    show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
        using next-instr2 rel-stacktraces-Cons
        using Some
        unfolding IGetUbx
        by (auto simp: next-instr-take-Suc-conv
intro!: rel-stacktraces.intros sp-instrs-prefix' intro: Subx.sp-instr.GetUbx)
qed (simp-all add: rel-F1-F2)
qed
qed
qed simp-all
next
case (step-set n R1' d Σ1')
then obtain u Σ2' where
Σ2-def: Σ2 = u # Σ2' and d-def: d = Subx.norm-unboxed u and

```

```

 $\Sigma 1' \text{-def: } \Sigma 1' = \text{map } Subx.\text{norm-unboxed } \Sigma 2'$ 
by auto
from step-set obtain instr2 where
next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
norm-eq-instr1-instr2: norm-eq (Inca.ISet n) instr2
by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
by simp-all
from next-instr2 norm-eq-instr1-instr2
show ?case (is  $\exists x. ?\text{STEP } x \wedge ?\text{MATCH } (\text{State } F1 H ?st1') x$ )
proof (cases instr2)
  case (ISet n')
  hence n' = n using norm-eq-instr1-instr2 by simp
  have typeof-u: typeof u = None
  using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc ISet, simplified]
  by (auto simp: Sigma2-def elim: Subx.sp-instrs.cases Subx.sp-instr.cases)
  hence cast-Dyn-u: cast-Dyn u = Some d
  by (auto simp add: d-def dest: Subx.typeof-and-norm-unboxed-imp-cast-Dyn)
  let ?R2' = R2[n := OpDyn d]
  let ?st2' = Frame f l (Suc pc) ?R2' Sigma2' # st2'
  let ?s2' = State F2 H ?st2'
  show ?thesis
  proof (intro exI conjI)
    show ?STEP ?s2'
    using step-set.hyps cast-Dyn-u
    using next-instr2 norm-eq-instr1-instr2
    unfolding ISet
    by (auto simp: s2-def st2-def Sigma2-def intro: Subx.step-set)
  next
    show ?MATCH (State F1 H ?st1') ?s2'
    proof (rule match.intros)
      show Subx.wf-state ?s2'
      using wf-F2 by (auto intro: Subx.wf-stateI)
    next
      show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
      using next-instr2 rel-stacktraces-Cons
      unfolding ISet
      using step-set.hyps cast-Dyn-u
      by (auto simp: Sigma1'-def Sigma2-def map-update next-instr-take-Suc-conv
           intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.Set
           intro: list-all-list-updateI)
      qed (simp-all add: rel-F1-F2)
    qed
  next
    case (ISetUbx tau n')
    hence n' = n using norm-eq-instr1-instr2 by simp
    have typeof-u: typeof u = Some tau
    using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc ISetUbx,

```

simplified]

```

by (auto simp: Σ2-def elim: Subx.sp-instrs.cases Subx.sp-instr.cases)
hence cast-and-box-u: Subx.cast-and-box τ u = Some d
by (auto simp add: d-def dest: Subx.typeof-and-norm-unboxed-imp-cast-and-box)
let ?R2' = R2[n := OpDyn d]
let ?st2' = Frame f l (Suc pc) ?R2' Σ2' # st2'
let ?s2' = State F2 H ?st2'
show ?thesis
proof (intro exI conjI)
show ?STEP ?s2'
using step-set cast-and-box-u
using next-instr2 norm-eq-instr1-instr2
unfolding ISetUbx
by (auto simp: s2-def st2-def Σ2-def intro: Subx.step-set-ubx)
next
show ?MATCH (State F1 H ?st1') ?s2'
proof (rule match.intros)
show Subx.wf-state ?s2'
using wf-F2 by (auto intro: Subx.wf-stateI)
next
show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
using next-instr2 rel-stacktraces-Cons
unfolding ISetUbx
using step-set.hyps cast-and-box-u
by (auto simp: Σ1'-def Σ2-def map-update next-instr-take-Suc-conv
intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.SetUbx
intro: list-all-list-updateI)
qed (simp-all add: rel-F1-F2)
qed
qed simp-all
next
case (step-load x y d Σ1')
then obtain u Σ2' where
Σ2-def: Σ2 = u # Σ2' and d-def: y = Subx.norm-unboxed u and
Σ1'-def: Σ1' = map Subx.norm-unboxed Σ2'
by auto
from step-load obtain instr2 where
next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
norm-eq-instr1-instr2: norm-eq (Inca.ILoad x) instr2
by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
by simp-all
from next-instr2 norm-eq-instr1-instr2
show ?case (is ∃x. ?STEP x ∧ ?MATCH (State F1 H ?st1') x)
proof (cases instr2)
case (ILoad x')
hence x' = x using norm-eq-instr1-instr2 by simp
have typeof-u: typeof u = None

```

```

using sp-intrs-sufix[OF pc-in-range, unfolded nth-intrs-pc ILoad, simplified]
  by (auto simp: Σ2-def elim: Subx.sp-intrs.cases Subx.sp-instr.cases)
hence cast-Dyn-u: cast-Dyn u = Some y
by (auto simp add: d-def dest: Subx.typeof-and-norm-unboxed-imp-cast-Dyn)
let ?st2' = Frame f l (Suc pc) R2 (OpDyn d # Σ2') # st2'
let ?s2' = State F2 H ?st2'
show ?thesis
proof (intro exI conjI)
  show ?STEP ?s2'
    using step-load.hyps cast-Dyn-u next-instr2
    unfolding ILoad ⟨x' = x⟩
    by (auto simp: s2-def st2-def Σ2-def intro: Subx.step-load)
next
  show ?MATCH (State F1 H ?st1') ?s2'
  proof (rule match.intros)
    show Subx.wf-state ?s2'
    using wf-F2 by (auto intro: Subx.wf-stateI)
next
  show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
  using next-instr2 rel-stacktraces-Cons
  unfolding ILoad ⟨x' = x⟩
  using step-load.hyps cast-Dyn-u
  by (auto simp: Σ1'-def Σ2-def map-update next-instr-take-Suc-conv
        intro: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.Load
        intro: list-all-list-updateI)
  qed (simp-all add: rel-F1-F2)
qed
next
  case (ILoadUbx τ x')
  hence x' = x using norm-eq-instr1-instr2 by simp
  have typeof-u: typeof u = None
  using sp-intrs-sufix[OF pc-in-range, unfolded nth-intrs-pc ILoadUbx, simplified]
    by (auto simp: Σ2-def elim: Subx.sp-intrs.cases Subx.sp-instr.cases)
  hence cast-Dyn-u: cast-Dyn u = Some y
  by (auto simp add: d-def dest: Subx.typeof-and-norm-unboxed-imp-cast-Dyn)
  show ?thesis
  proof (cases Subx.unbox τ d)
    case None
    let ?F2' = Subx.Fenv.map-entry F2 f Subx.generalize-fundef
    let ?st2' = Subx.box-stack f (Frame f l (Suc pc) R2 (OpDyn d # Σ2') #
      st2')
    let ?s2' = State ?F2' H ?st2'
    show ?thesis
    proof (intro exI conjI)
      show ?STEP ?s2'
        using step-load.hyps next-instr2 cast-Dyn-u None
        unfolding ILoadUbx ⟨x' = x⟩

```

```

by (auto simp add: s2-def st2-def Σ2-def intro: Subx.step-load-ubx-miss[simplified])
next
  show ?MATCH (State F1 H ?st1') ?s2'
  proof (rule match.intros)
    show Subx.wf-state ?s2'
    using wf-F2
    by (auto intro!: Subx.wf-stateI intro: Subx.wf-fundefs-generalize)
  next
    show rel-fundefs (Finca-get F1) (Fubx-get ?F2')
    using rel-F1-F2
    by (auto intro: rel-fundefs-generalizeI)
  next
    have Subx.sp-intrs (map-option funtype o Fubx-get F2) (return fd2)
      (take (Suc pc) (map Subx.generalize-instr intrs)) [] (None # map
      Map.empty Σ2')
    using rel-stacktraces-Cons step-load.hyps
    using pc-in-range nth-intrs-pc
    using ILoadUbx <x' = x>
    by (auto simp: Σ2-def take-Suc-conv-app-nth take-map
      intro!: Subx.sp-intrs-appendI
      intro: Subx.sp-intrs-generalize0 Subx.sp-instr.Load)
  thus rel-stacktraces (Fubx-get ?F2') None ?st1' ?st2'
    apply simp
    proof (rule rel-stacktraces.intros)
      show Fubx-get ?F2' f = Some (Subx.generalize-fundef fd2)
      using F2-f by simp
  next
    show map-of (body (Subx.generalize-fundef fd2)) l =
      Some (map Subx.generalize-instr intrs)
    using map-of-fd2-l
    by (simp add: Subx.map-of-generalize-fundef-conv)
    qed (insert rel-F1-F2 all-dyn-R2 F2-f map-of-fd2-l rel-st1'-st2', auto
      simp: Σ1'-def)
    qed
  qed
  next
    case (Some u2)
    let ?st2' = Frame f l (Suc pc) R2 (u2 # Σ2') # st2'
    let ?s2' = State F2 H ?st2'
    show ?thesis
    proof (intro exI conjI)
      show ?STEP ?s2'
      using step-load.hyps next-instr2 cast-Dyn-u Some
      unfolding ILoadUbx <x' = x>
    by (auto simp add: s2-def st2-def Σ2-def intro: Subx.step-load-ubx-hit[simplified])
  next
    show ?MATCH (State F1 H ?st1') ?s2'
    proof (rule match.intros)
      show Subx.wf-state ?s2'

```

```

    using wf-F2
    by (auto intro!: Subx.wf-stateI intro: Subx.wf-fundefs-generalize)
next
    show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
        using next-instr2 rel-stacktraces-Cons
        using Some typeof-u
        unfolding ILoadUbx
        by (auto simp: Σ2-def Σ1'-def next-instr-take-Suc-conv
              intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.LoadUbx)
    qed (insert rel-F1-F2, simp-all)
    qed
    qed
    qed simp-all
next
    case (step-store x d1 d2 H' Σ1')
    then obtain u1 u2 Σ2' where
        Σ2-def: Σ2 = u1 # u2 # Σ2' and
        d1-def: d1 = Subx.norm-unboxed u1 and
        d2-def: d2 = Subx.norm-unboxed u2 and
        Σ1'-def: Σ1' = map Subx.norm-unboxed Σ2'
        by auto
    from step-store obtain instr2 where
        next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
        norm-eq-instr1-instr2: norm-eq (Inca.instr.IStore x) instr2
        by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
    have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
        using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
        by simp-all
    from next-instr2 norm-eq-instr1-instr2
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH (State F1 H' ?st1') x)
    proof (cases instr2)
        case (IStore x')
        hence x' = x using norm-eq-instr1-instr2 by simp
        have casts: cast-Dyn u1 = Some d1 cast-Dyn u2 = Some d2
            unfolding atomize-conj
            using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc IStore, simplified]
        by (auto simp: d1-def d2-def Σ2-def elim: Subx.sp-instrs.cases Subx.sp-instr.cases
              intro: Subx.typeof-and-norm-unboxed-imp-cast-Dyn)
        let ?st2' = Frame f l (Suc pc) R2 Σ2' # st2'
        let ?s2' = State F2 H' ?st2'
        show ?thesis
        proof (intro exI conjI)
            show ?STEP ?s2'
                using step-store.hyps casts next-instr2
                unfolding IStore ⟨x' = x⟩
                by (auto simp: s2-def st2-def Σ2-def intro: Subx.step-store)
next
    show ?MATCH (State F1 H' ?st1') ?s2'

```

```

proof (rule match.intros)
  show Subx.wf-state ?s2'
    using wf-F2 by (auto intro: Subx.wf-stateI)
next
  show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
    using next-instr2 rel-stacktraces-Cons
    unfolding IStore ⟨x' = xusing step-store.hyps casts
    by (auto simp: Σ1'-def Σ2-def map-update next-instr-take-Suc-conv
          intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.Store
          intro: list-all-list-updateI)
    qed (insert rel-F1-F2, simp-all)
  qed
next
  case (IStoreUbx τ x')
  hence x' = x using norm-eq-instr1-instr2 by simp
  have casts: cast-Dyn u1 = Some d1 Subx.cast-and-box τ u2 = Some d2
    unfolding atomize-conj
    using sp-intrs-sufix[OF pc-in-range, unfolded nth-intrs-pc IStoreUbx,
simplified]
    by (auto simp: d1-def d2-def Σ2-def elim: Subx.sp-intrs.cases Subx.sp-instr.cases
          intro: Subx.typeof-and-norm-unboxed-imp-cast-Dyn
          intro: Subx.typeof-and-norm-unboxed-imp-cast-and-box)
  let ?st2' = Frame f l (Suc pc) R2 Σ2' # st2'
  let ?s2' = State F2 H' ?st2'
  show ?thesis
  proof (intro exI conjI)
    show ?STEP ?s2'
      using step-store.hyps casts next-instr2
      unfolding IStoreUbx ⟨x' = xby (auto simp: s2-def st2-def Σ2-def intro: Subx.step-store-ubx)
next
  show ?MATCH (State F1 H' ?st1') ?s2'
  proof (rule match.intros)
    show Subx.wf-state ?s2'
      using wf-F2 by (auto intro: Subx.wf-stateI)
next
  show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
    using next-instr2 rel-stacktraces-Cons
    unfolding IStoreUbx ⟨x' = xusing step-store.hyps casts
    by (auto simp: Σ1'-def Σ2-def map-update next-instr-take-Suc-conv
          intro!: rel-stacktraces.intros sp-intrs-prefix' Subx.sp-instr.StoreUbx
          intro: list-all-list-updateI)
    qed (insert rel-F1-F2, simp-all)
  qed
  qed simp-all
next
  case (step-op op ar x)

```

```

then obtain instr2 where
  next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
    norm-eq-instr1-instr2: norm-eq (Inca.IOp op) instr2
    by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
  have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
    using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
    by simp-all
  from next-instr2 norm-eq-instr1-instr2
  show ?case (is  $\exists x. \text{?STEP } x \wedge \text{?MATCH} (\text{State } F1 H \text{?st1}') x$ )
  proof (cases instr2)
    case (IOp op')
    hence op' = op using norm-eq-instr1-instr2 by simp
    have casts:
      ap-map-list cast-Dyn (take ar  $\Sigma 2$ ) = Some (take ar (map Subx.norm-unboxed
 $\Sigma 2$ ))
      using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc IOp, simplified]
      using step-op.hyps
      by (auto simp: op' = op take-map
        elim!: Subx.sp-instrs.cases[of - - x # xs for x xs, simplified] Subx.sp-instr.cases
        intro!: ap-map-list-cast-Dyn-eq-norm-stack[of take (Aritop op)  $\Sigma 2$ ]
        dest!: map-eq-append-replicate-conv)
    let ?st2' = Frame f l (Suc pc) R2 (OpDyn x # drop ar  $\Sigma 2$ ) # st2'
    let ?s2' = State F2 H ?st2'
    show ?thesis
    proof (intro exI conjI)
      show ?STEP ?s2'
        using step-op.hyps casts next-instr2
        unfolding IOp op' = op
        by (auto simp: s2-def st2-def intro: Subx.step-op)
    next
      show ?MATCH (State F1 H ?st1') ?s2'
      proof (rule match.intros)
        show Subx.wf-state ?s2'
          using wf-F2 by (auto intro: Subx.wf-stateI)
    next
      show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
      using step-op.hyps casts next-instr2 rel-stacktraces-Cons
      unfolding IOp op' = op
      by (auto simp: min-absorb2 take-map[symmetric] drop-map[symmetric]
        simp: next-instr-take-Suc-conv
        intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.Op
        intro: list-all-list-updateI
        dest!: ap-map-list-cast-Dyn-replicate[symmetric])
      qed (insert rel-F1-F2, simp-all)
    qed
  qed simp-all
next
  case (step-op-inl op ar opinl x F1')
  then obtain instr2 where

```

```

next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
norm-eq-instr1-instr2: norm-eq (Inca.IOp op) instr2
by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
  using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
  by simp-all
from next-instr2 norm-eq-instr1-instr2
show ?case (is  $\exists x. \text{?STEP } x \wedge \text{?MATCH} (\text{State } F1' H \text{?st1}') x$ )
proof (cases instr2)
  case (IOp op')
  hence op' = op using norm-eq-instr1-instr2 by simp
  have casts:
    ap-map-list cast-Dyn (take ar  $\Sigma 2$ ) = Some (take ar (map Subx.norm-unboxed
 $\Sigma 2$ ))
    using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc IOp, simplified]
    using step-op-inl.hyps
    by (auto simp: op' = op take-map
      elim!: Subx.sp-instrs.cases[of - - x # xs for x xs, simplified] Subx.sp-instr.cases
        intro!: ap-map-list-cast-Dyn-eq-norm-stack[of take (Arit $\eta$  op)  $\Sigma 2$ ]
        dest!: map-eq-append-replicate-conv)
  let ?st2' = Frame f l (Suc pc) R2 (OpDyn x # drop ar  $\Sigma 2$ ) # st2'
  let ?F2' = Subx.Fenv.map-entry F2 f (λ fd. rewrite-fundef-body fd l pc
  (Ubx.IOpInl opinl))
  let ?s2' = State ?F2' H ?st2'
  have step-s2-s2': ?STEP ?s2'
    using step-op-inl.hyps casts next-instr2
    unfolding IOp op' = op
    by (auto simp: s2-def st2-def intro: Subx.step-op-inl)
  show ?thesis
  proof (intro exI conjI)
    show ?STEP ?s2' by (rule step-s2-s2')
  next
    show ?MATCH (State F1' H ?st1') ?s2'
    proof (rule match.intros)
      show Subx.wf-state ?s2'
        by (rule Subx.wf-state-step-preservation[OF wf-s2 step-s2-s2'])
    next
      show rel-fundefs (Finca-get F1') (Fubx-get ?F2')
        using rel-F1-F2 step-op-inl
        by (auto intro: rel-fundefs-rewriteI)
    next
      have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
        using rel-st1'-st2'
        apply (rule rel-stacktraces.intros)
        using step-op-inl.hyps casts next-instr2 rel-stacktraces-Cons
        unfolding IOp op' = op
        by (auto simp: min-absorb2 take-map[symmetric] drop-map[symmetric]
          simp: next-instr-take-Suc-conv
          intro!: sp-instrs-prefix' Subx.sp-instr.Op

```

```

dest!: ap-map-list-cast-Dyn-replicate[symmetric])
then show rel-stacktraces (Fubx-get ?F2') None ?st1' ?st2'
  apply (rule rel-stacktraces-map-entry-rewrite-fundef-body)
  apply (rule next-instr2)
  unfolding IOp <op' = op>
  using Sinca.Inl-invertible Subx.sp-instr-Op-OpInl-conv step-op-inl.hyps(4)
apply blast
  apply simp
  apply simp
  done
qed
qed
qed simp-all
next
case (step-op-inl-hit opinl ar x)
then obtain instr2 where
  next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
  norm-eq-instr1-instr2: norm-eq (Inca.IOpInl opinl) instr2
  by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
  using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
  by simp-all
from next-instr2 norm-eq-instr1-instr2
show ?case (is ∃ x. ?STEP x ∧ ?MATCH (State F1 H ?st1') x)
proof (cases instr2)
  case (IOpInl opinl')
  hence opinl' = opinl using norm-eq-instr1-instr2 by simp
  have casts:
    ap-map-list cast-Dyn (take ar Σ2) = Some (take ar (map Subx.norm-unboxed
Σ2))
    using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc IOpInl, sim-
plified]
    using step-op-inl-hit.hyps
    by (auto simp: <opinl' = opinl> take-map
      elim!: Subx.sp-instrs.cases[of - - x # xs for x xs, simplified] Subx.sp-instr.cases
      intro!: ap-map-list-cast-Dyn-eq-norm-stack[of take (Arith (DeInl opinl))]
Σ2]
    dest!: map-eq-append-replicate-conv)
  let ?st2' = Frame f l (Suc pc) R2 (OpDyn x # drop ar Σ2) # st2'
  let ?s2' = State F2 H ?st2'
  have step-s2-s2': ?STEP ?s2'
    using step-op-inl-hit.hyps casts next-instr2
    unfolding IOpInl <opinl' = opinl>
    by (auto simp: s2-def st2-def intro: Subx.step-op-inl-hit)
  show ?thesis
  proof (intro exI conjI)
    show ?STEP ?s2' by (rule step-s2-s2')
  next
    show ?MATCH (State F1 H ?st1') ?s2'

```

```

proof (rule match.intros)
  show Subx.wf-state ?s2'
    by (rule Subx.wf-state-step-preservation[OF wf-s2 step-s2-s2'])
next
  show rel-stacktraces(Fubx-get F2) None ?st1' ?st2'
    using step-op-inl-hit.hyps casts next-instr2 rel-stacktraces-Cons
    unfolding IOpInl <opinl' = opinl'
    by (auto simp: min-absorb2 take-map[symmetric] drop-map[symmetric]
          simp: next-instr-take-Suc-conv
          intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.OpInl
          intro: list-all-list-updateI
          dest!: ap-map-list-cast-Dyn replicate[symmetric])
    qed (insert rel-F1-F2, simp-all)
  qed
next
  case (IOpUbx opubx)
  hence opinl = Box opubx using norm-eq-instr1-instr2 by simp
  let ?ar = Arity (DeInl (Box opubx))

  obtain codom where typeof-opubx: TypeOp opubx = (map typeof (take
  ?ar Σ2), codom)
    using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc IOpUbx,
simplified]
    by (cases TypeOp opubx
          (auto simp: eq-append-conv-conj Subx.TypeOp-Arity take-map
          dest!: Subx.sp-instrs-ConsD elim!: Subx.sp-instr.cases)

  obtain u where
    eval-opubx: UbxOp opubx (take ?ar Σ2) = Some u and typeof-u: typeof u
    = codom
    using Subx.TypeOp-correct[OF typeof-opubx] by auto
    hence x-def: x = Subx.norm-unboxed u
    using step-op-inl-hit.hyps
    using Subx.UbxOp-correct[OF eval-opubx]
    unfolding <opinl = Box opubx> take-map
    by simp

  let ?st2' = Frame f l (Suc pc) R2 (u # drop ?ar Σ2) # st2'
  let ?s2' = State F2 H ?st2'

  have step-s2-s2': ?STEP ?s2'
    using step-op-inl-hit.hyps next-instr2
    unfolding IOpUbx <opinl = Box opubx>
    using eval-opubx
    by (auto simp: s2-def st2-def intro!: Subx.step-op-ubx)
  show ?thesis
  proof (intro exI conjI)
    show ?STEP ?s2' by (rule step-s2-s2')
  next

```

```

show ?MATCH (State F1 H ?st1') ?s2'
proof (rule match.intros)
  show Subx.wf-state ?s2'
    by (rule Subx.wf-state-step-preservation[OF wf-s2 step-s2-s2'])
next
  show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
    using step-op-inl-hit.hyps next-instr2 rel-stacktraces-Cons
    unfolding IOpUbx `opinl = Ⓜ opubx x-def
    by (auto simp: typeof-opubx typeof-u
        simp: min-absorb2 take-map[symmetric] drop-map[symmetric]
        simp: next-instr-take-Suc-conv
        intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.OpUbx
        dest!: ap-map-list-cast-Dyn replicate[symmetric])
  qed (insert rel-F1-F2, simp-all)
qed
qed simp-all
next
case (step-op-inl-miss opinl ar x F1')
then obtain instr2 where
  next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
  norm-eq-instr1-instr2: norm-eq (Inca.IOpInl opinl) instr2
  by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
  using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
  by simp-all
from next-instr2 norm-eq-instr1-instr2
show ?case (is ∃ x. ?STEP x ∧ ?MATCH (State F1' H ?st1') x)
proof (cases instr2)
  case (IOpInl opinl')
  hence opinl' = opinl using norm-eq-instr1-instr2 by simp
  have casts:
    ap-map-list cast-Dyn (take ar Σ2) = Some (take ar (map Subx.norm-unboxed
Σ2))
    using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc IOpInl, sim-
plified]
    using step-op-inl-miss.hyps
    by (auto simp: `opinl' = opinl` take-map
      elim!: Subx.sp-instrs.cases[of - - x # xs for x xs, simplified] Subx.sp-instr.cases
      intro!: ap-map-list-cast-Dyn-eq-norm-stack[of take (Arith (DeInl opinl))])
Σ2]
    dest!: map-eq-append-replicate-conv)
  let ?st2' = Frame f l (Suc pc) R2 (OpDyn x # drop ar Σ2) # st2'
    let ?F2' = Subx.Fenv.map-entry F2 f (λfd. rewrite-fundef-body fd l pc
(Ubx.IOp (DeInl opinl)))
    let ?s2' = State ?F2' H ?st2'
    have step-s2-s2': ?STEP ?s2'
      using step-op-inl-miss.hyps casts next-instr2
      unfolding IOpInl `opinl' = opinl
      by (auto simp: s2-def st2-def intro: Subx.step-op-inl-miss)

```

```

show ?thesis
proof (intro exI conjI)
  show ?STEP ?s2' by (rule step-s2-s2')
next
  show ?MATCH (State F1' H ?st1') ?s2'
  proof (rule match.intros)
    show Subx.wf-state ?s2'
    by (rule Subx.wf-state-step-preservation[OF wf-s2 step-s2-s2'])
next
  show rel-fundefs (Finca-get F1') (Fubx-get ?F2')
  using rel-F1-F2 step-op-inl-miss.hyps
  by (auto intro: rel-fundefs-rewriteI)
next
  have rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
  using rel-st1'-st2'
  apply (rule rel-stacktraces.intros)
  using step-op-inl-miss.hyps casts next-instr2 rel-stacktraces-Cons
  unfolding IOpInl <opinl' = opinl>
  by (auto simp: min-absorb2 take-map[symmetric] drop-map[symmetric]
    simp: next-instr-take-Suc-conv
    intro!: sp-intrs-prefix' Subx.sp-instr.OpInl
    dest!: ap-map-list-cast-Dyn-replicate[symmetric])
  then show rel-stacktraces (Fubx-get ?F2') None ?st1' ?st2'
  apply (rule rel-stacktraces-map-entry-rewrite-fundef-body)
  apply (rule next-instr2)
  unfolding IOpInl <opinl' = opinl>
  using Sinca.Inl-invertible Subx.sp-instr-Op-OpInl-conv step-op-inl-miss.hyps
apply metis
  apply simp
  apply simp
  done
qed
qed
next
  case (IOpUbx opubx)
  hence opinl = ⌈opubx⌉ using norm-eq-instr1-instr2 by simp
  let ?ar = ⌈arity(DeInl(⌈opubx⌉))⌉
    obtain codom where typeof-opubx: TypeOp opubx = (map typeof (take
      ?ar Σ2), codom)
      using sp-intrs-sufix[OF pc-in-range, unfolded nth-intrs-pc IOpUbx,
      simplified]
      by (cases TypeOp opubx)
        (auto simp: eq-append-conv-conj Subx.TypeOp-Arith take-map
        dest!: Subx.sp-intrs-ConsD elim!: Subx.sp-instr.cases)
    obtain u where UbrOp opubx (take ?ar Σ2) = Some u
      using Subx.TypeOp-correct[OF typeof-opubx] by auto
    hence IsInl opinl (take ?ar Σ1)
      unfolding Σ1-def

```

```

by (auto simp: `opinl = Bx opubx` take-map dest: Subx.UbxOp-to-Inl[THEN
Inca.Inl-IsInl])
hence False
using step-op-inl-miss.hyps
by (simp add: Σ1-def `opinl = Bx opubx`)
thus ?thesis by simp
qed simp-all
next
case (step-cjump lt lf d l' Σ1')
then obtain u Σ2' where
Σ2-def: Σ2 = u # Σ2' and
d-def: d = Subx.norm-unboxed u and
Σ1'-def: Σ1' = map Subx.norm-unboxed Σ2'
by auto
from step-cjump.hyps obtain instr2 where
next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
norm-eq-instr1-instr2: norm-eq (Inca.ICJump lt lf) instr2
by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
by simp-all

from next-instr2 norm-eq-instr1-instr2
show ?case (is ∃ x. ?STEP x ∧ ?MATCH (State F1 H ?st1') x)
proof (cases instr2)
case (ICJump lt' lf)
hence lt' = lt and lf' = lf using norm-eq-instr1-instr2 by simp-all
hence {lt, lf} ⊆ fst ` set (body fd2)
using all-jumps-in-range pc-in-range nth-instrs-pc ICJump by (auto simp:
list-all-length)
moreover have l' ∈ {lt, lf}
using step-cjump.hyps by auto
ultimately have l' ∈ fst ` set (body fd2)
by blast
then obtain instrs' where map-of-l': map-of (body fd2) l' = Some instrs'
by (auto dest: weak-map-of-SomeI)

have sp-instrs-instrs': Subx.sp-instrs (map-option funtype o Fubx-get F2)
{return fd2}
(butlast instrs @ [instrs ! pc]) [] [] if pc-def: pc = length instrs - 1
unfolding pc-def last-conv-nth[OF instrs-neq-Nil, symmetric]
unfolding append-butlast-last-id[OF instrs-neq-Nil]
by (rule sp-instrs-instrs)

have sp-instr-last: Subx.sp-instr (map-option funtype o Fubx-get F2) (return
fd2)
(instrs ! pc) (map typeof Σ2) [] [] if pc-def: pc = length instrs - 1
using sp-instrs-instrs'[OF pc-def]
using sp-instrs-prefix[unfolded pc-def butlast-conv-take[symmetric]]

```

```

by (auto dest!: Subx.sp-instrs-appendD')

have is-jump-nthD:  $\bigwedge n. \text{is-jump } (\text{instrs} ! n) \implies n < \text{length } \text{instrs} \implies n = \text{length } \text{instrs} - 1$ 
  using list-all-map-of-SomeD[ $\text{OF wf-fd2[THEN Subx.wf-fundef-all-wf-basic-blockD]}$ ]
  map-of-fd2-l]
  by (auto dest!: Subx.wf-basic-blockD
    list-all-butlast-not-nthD[of  $\lambda i. \neg \text{is-jump } i \wedge \neg \text{Ubx.instr.is-return } i$ ,
    simplified,  $\text{OF} - \text{disjII}]]$ )

have pc-def:  $\text{pc} = \text{length } \text{instrs} - 1$ 
  using is-jump-nthD[ $\text{OF} - \text{pc-in-range}$ ] nth-instrs-pc ICJump by simp
have  $\Sigma 2' \text{-eq-Nil}: \Sigma 2' = []$ 
  using sp-instr-last[ $\text{OF pc-def}$ ] step-cjump.hyps
by (auto simp:  $\Sigma 2\text{-def } d\text{-def }$  nth-instrs-pc ICJump elim!: Subx.sp-instr.cases)

have cast:  $\text{cast-Dyn } u = \text{Some } d$ 
  using sp-instrs-sufix[ $\text{OF pc-in-range, unfolded nth-instrs-pc ICJump}$ ,
  simplified]
  by (auto simp:  $\Sigma 2\text{-def } d\text{-def }$  dest!: Subx.sp-instrs-ConsD elim!: Subx.sp-instr.cases
    intro: Subx.typeof-and-norm-unboxed-imp-cast-Dyn)

let ?st2' = Frame f l' 0 R2  $\Sigma 2' \# st2'$ 
let ?s2' = State F2 H ?st2'

have step-s2-s2': ?STEP ?s2'
  using step-cjump.hyps cast next-instr2
  unfolding ICJump  $\langle l_t' = l_t \rangle \langle l_f' = l_f \rangle$ 
  by (auto simp: s2-def st2-def  $\Sigma 2\text{-def }$  intro!: Subx.step-cjump)
show ?thesis
proof (intro exI conjI)
  show ?STEP ?s2' by (rule step-s2-s2')
next
  show ?MATCH (State F1 H ?st1') ?s2'
    proof (rule match.intros)
      show Subx.wf-state ?s2'
        by (rule Subx.wf-state-step-preservation[ $\text{OF wf-s2 step-s2-s2'}$ ])
    next
      show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
        using step-cjump.hyps cast next-instr2 rel-stacktraces-Cons
        using map-of-l'
        unfolding ICJump  $\langle l_t' = l_t \rangle \langle l_f' = l_f \rangle$   $\Sigma 1'\text{-def } \Sigma 2' \text{-eq-Nil}$ 
        by (auto simp: min-absorb2 take-map[symmetric] drop-map[symmetric]
          simp: next-instr-take-Suc-conv
          intro!: rel-stacktraces.intros sp-instrs-prefix' Subx.sp-instr.CJump
          intro: Subx.sp-instrs.Nil
          dest!: ap-map-list-cast-Dyn-replicate[symmetric]))
      qed (insert rel-F1-F2, simp-all)
    qed
  qed

```

```

qed simp-all
next
case (step-call g gd1 frameg)
then obtain instr2 where
  next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
  norm-eq-instr1-instr2: norm-eq (Inca.ICall g) instr2
  by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
have pc-in-range: pc < length instrs and nth-instrs-pc: instrs ! pc = instr2
  using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
  by simp-all

from step-call.hyps obtain gd2 where
  F2-g: Fubx-get F2 g = Some gd2 and rel-gd1-gd2: rel-fundef (=) norm-eq
gd1 gd2
  using rel-fundefs-Some1[OF rel-F1-F2] by auto

have wf-gd2: Subx.wf-fundef (map-option funtype o Fubx-get F2) gd2
  by (rule Subx.wf-fundefs-getD[OF wf-F2 F2-g])

obtain intrsg where gd2-fst-bblock: map-of (body gd2) (fst (hd (body gd2)))
= Some intrsg
  using Subx.wf-fundef-body-neq-NilD[OF wf-gd2]
  by (metis hd-in-set map-of-eq-None-iff not-Some-eq prod.collapse prod-in-set-fst-image-conv)

from norm-eq-instr1-instr2
show ?case (is  $\exists x.$  ?STEP x  $\wedge$  ?MATCH (State F1 H ?st1') x)
proof (cases instr2)
  case (ICall g')
  hence g' = g using norm-eq-instr1-instr2 by simp
  hence all-dyn-args: list-all is-dyn-operанд (take (arity gd2)  $\Sigma 2$ )
  using sp-instrs-sufix[OF pc-in-range, unfolded nth-instrs-pc ICall, simplified]
  using F2-g
  by (auto simp: funtype-def eq-append-conv-conj take-map list.pred-set
dest!: Subx.sp-instrs-ConsD replicate-eq-impl-Ball-eq elim!: Subx.sp-instr.cases)

let ?frameg = allocate-frame g gd2 (take (arity gd2)  $\Sigma 2$ ) (OpDyn uninitialized)
let ?st2' = ?frameg # Frame f l pc R2  $\Sigma 2$  # st2'
let ?s2' = State F2 H ?st2'

have step-s2-s2': ?STEP ?s2'
  using step-call.hyps next-instr2 F2-g rel-gd1-gd2 all-dyn-args
  unfolding ICall `g' = g`
  by (auto simp: s2-def st2-def rel-fundef-arithies intro!: Subx.step-call)
show ?thesis
proof (intro exI conjI)
  show ?STEP ?s2' by (rule step-s2-s2')
next
  show ?MATCH (State F1 H ?st1') ?s2'

```

```

proof (rule match.intros)
  show Subx.wf-state ?s2'
    by (rule Subx.wf-state-step-preservation[OF wf-s2 step-s2-s2'])
next
  have FOO: fst (hd (body gd1)) = fst (hd (body gd2))
    apply (rule rel-fundef-rel-fst-hd-bodies[OF rel-gd1-gd2])
    using Subx.wf-fundefs-getD[OF wf-F2 <Fubx-get F2 g = Some gd2>]
    by (auto dest: Subx.wf-fundef-body-neq-NilD)
  show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
    unfolding step-call.hyps allocate-frame-def FOO
  proof (rule rel-stacktraces.intros)
    show Fubx-get F2 g = Some gd2
      by (rule F2-g)
  next
    show rel-stacktraces (Fubx-get F2) (Some g)
    (Frame f l pc (map Subx.norm-unboxed R2) (map Subx.norm-unboxed
      $\Sigma_2)$  # st1')
      (Frame f l pc R2  $\Sigma_2$  # st2')
      using step-call.hyps rel-stacktraces-Cons next-instr2 F2-g rel-gd1-gd2
      all-dyn-args
        unfolding ICall  $\langle g' = g \rangle$ 
        by (auto simp: is-valid-fun-call-def rel-fundef-arithes introl: rel-stacktraces.intros)
        qed (insert rel-gd1-gd2 all-dyn-args gd2-fst-bblock,
          simp-all add: take-map rel-fundef-arithes rel-fundef-locals Subx.sp-intrs.Nil
          list-all-replicateI)
        qed (insert rel-F1-F2, simp-all)
        qed
      qed simp-all
  next
  case (step-return fd1  $\Sigma_1$  frameg' g lg pcg R1g st1')
  then obtain instr2 where
    next-instr2: next-instr (Fubx-get F2) f l pc = Some instr2 and
    norm-eq-instr1-instr2: norm-eq Inca.IReturn instr2
    by (auto dest: rel-fundefs-next-instr1[OF rel-F1-F2])
  have pc-in-range: pc < length instrs and nth-intrs-pc: instrs ! pc = instr2
    using next-instr-get-map-ofD[OF next-instr2 F2-f map-of-fd2-l]
    by simp-all

  from step-return.hyps have rel-fd1-fd2: rel-fundef (=) norm-eq fd1 fd2
    using rel-fundefsD[OF rel-F1-F2, of f] F2-f by simp

  from norm-eq-instr1-instr2
  show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH (State F1 H ?st1') x)
  proof (cases instr2)
    case IReturn
      have map-typeof- $\Sigma_2$ : map typeof  $\Sigma_2$  = replicate (return fd2) None
      using sp-intrs-suffix[OF pc-in-range, unfolded nth-intrs-pc IReturn,
      simplified]
      by (auto elim: Subx.sp-instr.cases dest: Subx.sp-intrs-ConsD)

```

```

hence all-dyn- $\Sigma 2$ : list-all is-dyn-operand  $\Sigma 2$ 
  by (auto simp: list.pred-set dest: replicate-eq-impl-Ball-eq[OF sym])
  show ?thesis
    using rel-st1'-st2' unfolding <Frame g lg pcg R1g  $\Sigma 1_g$  # st1'' =
    st1'[symmetric]
    proof (cases rule: rel-stacktraces.cases)
      case (rel-stacktraces-Cons st2''  $\Sigma 2_g$  R2g gd2 instrsg)
      let ?st2' = Frame g lg (Suc pcg) R2g ( $\Sigma 2$  @ drop (arity fd2)  $\Sigma 2_g$ ) # st2''
      let ?s2' = State F2 H ?st2'
      have step-s2-s2': ?STEP ?s2'
        using step-return.hyps next-instr2 F2-f rel-fd1-fd2 rel-stacktraces-Cons
      all-dyn- $\Sigma 2$ 
        unfolding IReturn
        by (auto simp: s2-def st2-def rel-fundef-arities rel-fundef-return
              intro: Subx.step-return)
        show ?thesis
        proof (intro exI conjI)
          show ?STEP ?s2' by (rule step-s2-s2')
        next
          show ?MATCH (State F1 H ?st1') ?s2'
          proof (rule match.intros)
            show Subx.wf-state ?s2'
              by (rule Subx.wf-state-step-preservation[OF wf-s2 step-s2-s2'])
          next
            have Subx.sp-instr (map-option funtype o Fubx-get F2) (return gd2)
              (Ubx.instr.ICall f) (map typeof  $\Sigma 2_g$ )
              (replicate (return fd2) None @ map typeof (drop (arity fd2)  $\Sigma 2_g$ ))
              using rel-stacktraces.Cons F2-f
              using replicate-eq-map[of arity fd2 take (arity fd2)  $\Sigma 2_g$  typeof None]
              by (auto simp: funtype-def is-valid-fun-call-def
                    simp: min-absorb2 list.pred-set take-map[symmetric] drop-map[symmetric]
                    intro!: Subx.sp-instr.Call[where  $\Sigma = \text{map typeof} (\text{drop} (\text{arity fd2})$ 
                     $\Sigma 2_g)]])
            then show rel-stacktraces (Fubx-get F2) None ?st1' ?st2'
              unfolding step-return.hyps
              using rel-stacktraces-Cons rel-fd1-fd2 F2-f
              using map-typeof- $\Sigma 2$ 
              by (auto simp: drop-map rel-fundef-arities is-valid-fun-call-def
                    simp: next-instr-take-Suc-conv funtype-def
                    intro!: rel-stacktraces.intros Subx.sp-instrs-appendI[where  $\Sigma =$ 
                    map typeof  $\Sigma 2_g)])
              qed (insert rel-F1-F2, simp-all)
            qed
            qed
            qed simp-all
            qed
            qed
            qed
            qed$$ 
```

```

lemma match-final-forward:
  assumes match s1 s2 and final-s1: final Finca-get Inca.IReturn s1
  shows final Fubx-get Ubx.IReturn s2
  using ⟨match s1 s2⟩
  proof (cases s1 s2 rule: match.cases)
    case (matchI F2 H st2 F1 st1)
    show ?thesis
    using final-s1[unfolded matchI]
    proof (cases - - State F1 H st1 rule: final.cases)
      case (finalI f l pc R Σ)
      then show ?thesis
      using matchI
      by (auto intro!: final.intros elim: rel-stacktraces.cases norm-instr.elims[OF
sym]
          dest: rel-fundefs-next-instr1)
    qed
  qed

sublocale inca-ubx-forward-simulation: forward-simulation where
  step1 = Sinca.step and final1 = final Finca-get Inca.IReturn and
  step2 = Subx.step and final2 = final Fubx-get Ubx.IReturn and
  match = λ-. match and order = λ- -. False
  using match-final-forward forward-lockstep-simulation
  using lockstep-to-plus-forward-simulation[of match Sinca.step - Subx.step]
  by unfold-locales auto

```

21 Bisimulation

```

sublocale inca-ubx-bisimulation: bisimulation where
  step1 = Sinca.step and final1 = final Finca-get Inca.IReturn and
  step2 = Subx.step and final2 = final Fubx-get Ubx.IReturn and
  match = λ-. match and orderf = λ- -. False and orderb = λ- -. False
  by unfold-locales

end

end
theory Inca-Verification
  imports Inca
begin

context inca begin

```

22 Strongest postcondition

```

inductive sp-instr for F ret where
  Push:
    sp-instr F ret (IPush d) Σ (Suc Σ) |

```

Pop:

$$sp\text{-}instr F \text{ ret } IPop (Suc \Sigma) \Sigma \mid$$

Get:

$$sp\text{-}instr F \text{ ret } (IGet n) \Sigma (Suc \Sigma) \mid$$

Set:

$$sp\text{-}instr F \text{ ret } (ISet n) (Suc \Sigma) \Sigma \mid$$

Load:

$$sp\text{-}instr F \text{ ret } (ILoad x) (Suc \Sigma) (Suc \Sigma) \mid$$

Store:

$$sp\text{-}instr F \text{ ret } (IStore x) (Suc (Suc \Sigma)) \Sigma \mid$$

Op:

$$\Sigma i = \text{Arity } op + \Sigma \implies$$

$$sp\text{-}instr F \text{ ret } (IOp op) \Sigma i (Suc \Sigma) \mid$$

OpInl:

$$\Sigma i = \text{Arity } (\text{OpInl } opinl) + \Sigma \implies$$

$$sp\text{-}instr F \text{ ret } (IOpInl opinl) \Sigma i (Suc \Sigma) \mid$$

CJump:

$$sp\text{-}instr F \text{ ret } (ICJump l_t l_f) 1 0 \mid$$

Call:

$$F f = Some (ar, r) \implies \Sigma i = ar + \Sigma \implies \Sigma o = r + \Sigma \implies$$

$$sp\text{-}instr F \text{ ret } (ICall f) \Sigma i \Sigma o \mid$$

Return:

$$\Sigma i = ret \implies$$

$$sp\text{-}instr F \text{ ret } IReturn \Sigma i 0$$

sp-instr calculates the strongest postcondition of the arity of the operand stack.

inductive sp-instrs for F ret where

Nil:

$$sp\text{-}instrs F \text{ ret } [] \Sigma \Sigma \mid$$

Cons:

$$sp\text{-}instr F \text{ ret } instr \Sigma i \Sigma \implies sp\text{-}instrs F \text{ ret } instrs \Sigma \Sigma o \implies$$

$$sp\text{-}instrs F \text{ ret } (instr \# instrs) \Sigma i \Sigma o$$

23 Range validations

```
fun local-var-in-range where
  local-var-in-range n (IGet k)  $\longleftrightarrow$   $k < n$  |
  local-var-in-range n (ISet k)  $\longleftrightarrow$   $k < n$  |
  local-var-in-range - -  $\longleftrightarrow$  True

fun jump-in-range where
  jump-in-range L (ICJump l_t l_f)  $\longleftrightarrow$   $\{l_t, l_f\} \subseteq L$  |
  jump-in-range L -  $\longleftrightarrow$  True

fun fun-call-in-range where
  fun-call-in-range F (ICall f)  $\longleftrightarrow$   $f \in \text{dom } F$  |
  fun-call-in-range F instr  $\longleftrightarrow$  True
```

24 Basic block validation

```
definition wf-basic-block where
  wf-basic-block F L ret n bblock  $\longleftrightarrow$ 
    (let (label, instrs) = bblock in
      list-all (local-var-in-range n) instrs  $\wedge$ 
      list-all (jump-in-range L) instrs  $\wedge$ 
      list-all (fun-call-in-range F) instrs  $\wedge$ 
      list-all ( $\lambda i. \neg Inca.\text{is-jump } i \wedge \neg Inca.\text{is-return } i$ ) (butlast instrs)  $\wedge$ 
      instrs  $\neq [] \wedge$ 
      sp-instrs F ret instrs 0 0)
```

25 Function definition validation

```
definition wf-fundef where
  wf-fundef F fd  $\longleftrightarrow$ 
    body fd  $\neq [] \wedge$ 
    list-all
      (wf-basic-block F (fst `set (body fd)) (return fd) (arity fd + fundef-locals fd))
      (body fd))
```

26 Program definition validation

```
definition wf-prog where
  wf-prog p  $\longleftrightarrow$ 
    (let F = F-get (prog-fundefs p) in
      pred-map (wf-fundef (map-option funtype o F)) F)

end

end
theory Inca-to-Ubx-compiler
imports Inca-to-Ubx-simulation Result
Inca-Verification
VeriComp.Compiler
HOL-Library.Monad-Syntax
begin
```

27 Generic program rewriting

```
primrec monadic-fold-map where
  monadic-fold-map f acc [] = Some (acc, [])
  monadic-fold-map f acc (x # xs) = do {
    (acc', x')  $\leftarrow$  f acc x;
    (acc'', xs')  $\leftarrow$  monadic-fold-map f acc' xs;
    Some (acc'', x' # xs')
  }
```

```

lemma monadic-fold-map-length:
  monadic-fold-map f acc xs = Some (acc', xs')  $\implies$  length xs = length xs'
  by (induction xs arbitrary: acc xs') (auto simp: bind-eq-Some-conv)

lemma monadic-fold-map-ConsD[dest]:
  assumes monadic-fold-map f a (x # xs) = Some (c, ys)
  shows  $\exists y \ ys' \ b. \ ys = y \# ys' \wedge f a x = Some (b, y) \wedge$  monadic-fold-map f b xs
  = Some (c, ys')
  using assms
  by (auto simp add: bind-eq-Some-conv)

lemma monadic-fold-map-eq-Some-conv:
  monadic-fold-map f a (x # xs) = Some (c, ys)  $\longleftrightarrow$ 
  ( $\exists y \ ys' \ b. \ f a x = Some (b, y) \wedge$  monadic-fold-map f b xs = Some (c, ys')  $\wedge$  ys
  = y # ys')
  by (auto simp add: bind-eq-Some-conv)

lemma monadic-fold-map-eq-Some-conv':
  monadic-fold-map f a (x # xs) = Some p  $\longleftrightarrow$ 
  ( $\exists y \ ys' \ b. \ f a x = Some (b, y) \wedge$  monadic-fold-map f b xs = Some (fst p, ys')
   $\wedge$  snd p = y # ys')
  by (cases p) (auto simp add: bind-eq-Some-conv)

lemma monadic-fold-map-list-all2:
  assumes monadic-fold-map f acc xs = Some (acc', ys) and
   $\wedge acc \ acc' \ x \ y. \ f acc x = Some (acc', y) \implies P x y$ 
  shows list-all2 P xs ys
  using assms(1)
  proof (induction xs arbitrary: acc ys)
    case Nil
    then show ?case by simp
  next
    case (Cons x xs)
    show ?case
      using Cons.preds
      by (auto simp: bind-eq-Some-conv intro: assms(2) Cons.IH)
  qed

lemma monadic-fold-map-list-all:
  assumes monadic-fold-map f acc xs = Some (acc', ys) and
   $\wedge acc \ acc' \ x \ y. \ f acc x = Some (acc', y) \implies P y$ 
  shows list-all P ys
  proof -
    have list-all2 ( $\lambda -. \ P$ ) xs ys
    using assms
    by (auto elim: monadic-fold-map-list-all2)
    thus ?thesis
      by (auto elim: list-rel-imp-pred2)
  qed

```

```

fun gen-pop-push where
  gen-pop-push instr (domain, codomain)  $\Sigma$  = (
    let ar = length domain in
    if ar  $\leq$  length  $\Sigma$   $\wedge$  take ar  $\Sigma$  = domain then
      Some (instr, codomain @ drop ar  $\Sigma$ )
    else
      None
  )

context inca-to-ubx-simulation begin

28 Lifting

fun lift-instr :: -  $\Rightarrow$  -  $\Rightarrow$  -  $\Rightarrow$  -  $\Rightarrow$  -  $\Rightarrow$  -
  ((-, -, -, -, -, 'opubx, 'ubx1, 'ubx2) Ubx.instr  $\times$  -) option where
    lift-instr F L ret N (Inca.IPush d)  $\Sigma$  = Some (IPush d, None #  $\Sigma$ ) |
    lift-instr F L ret N Inca.IPop (- #  $\Sigma$ ) = Some (IPop,  $\Sigma$ ) |
    lift-instr F L ret N (Inca.IGet n)  $\Sigma$  = (if n < N then Some (IGet n, None #  $\Sigma$ )
    else None) |
    lift-instr F L ret N (Inca.ISet n) (None #  $\Sigma$ ) = (if n < N then Some (ISet n,
 $\Sigma$ ) else None) |
    lift-instr F L ret N (Inca.ILoad x) (None #  $\Sigma$ ) = Some (ILoad x, None #  $\Sigma$ ) |
    lift-instr F L ret N (Inca.IStore x) (None # None #  $\Sigma$ ) = Some (IStore x,  $\Sigma$ ) |
    lift-instr F L ret N (Inca.IOp op)  $\Sigma$  =
      gen-pop-push (IOp op) (replicate (Aritiy op) None, [None])  $\Sigma$  |
      lift-instr F L ret N (Inca.IOpInl opinl)  $\Sigma$  =
        gen-pop-push (IOpInl opinl) (replicate (Aritiy (OpInl opinl)) None, [None])  $\Sigma$  |
        lift-instr F L ret N (Inca.ICJump lt lf) [None] =
          (if List.member L lt  $\wedge$  List.member L lf then Some (ICJump lt lf, []) else None)
    |
    lift-instr F L ret N (Inca.ICall f)  $\Sigma$  = do {
      (ar, ret)  $\leftarrow$  F f;
      gen-pop-push (ICall f) (replicate ar None, replicate ret None)  $\Sigma$ 
    } |
    lift-instr F L ret N Inca.IReturn  $\Sigma$  =
      (if  $\Sigma$  = replicate ret None then Some (IReturn, []) else None) |
    lift-instr - - - - - = None

definition lift-instrs where
  lift-instrs F L ret N  $\equiv$ 
    monadic-fold-map ( $\lambda \Sigma$  instr. map-option prod.swap (lift-instr F L ret N instr
 $\Sigma$ ))

lemma lift-instrs-length:
  assumes lift-instrs F L ret N  $\Sigma$ i xs = Some ( $\Sigma$ o, ys)
  shows length xs = length ys
  using assms unfolding lift-instrs-def
  by (auto intro: monadic-fold-map-length)

```

```

lemma lift-intrs-not-Nil: lift-intrs F L ret N  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)  $\implies$  xs  $\neq []$ 
 $\longleftrightarrow$  ys  $\neq []$ 
using lift-intrs-length by fastforce

lemma lift-intrs-NilD[dest]:
assumes lift-intrs F L ret N  $\Sigma i$  [] = Some ( $\Sigma o$ , ys)
shows  $\Sigma o = \Sigma i \wedge ys = []$ 
using assms
by (simp-all add: lift-intrs-def)

lemmas Some-eq-bind-conv =
bind-eq-Some-conv[unfolded eq-commute[of Option.bind f g Some x for f g x]]

lemma lift-instr-is-jump:
assumes lift-instr F L ret N x  $\Sigma i$  = Some (y,  $\Sigma o$ )
shows Inca.is-jump x  $\longleftrightarrow$  Ubx.is-jump y
using assms
by (rule lift-instr.elims)
(auto simp add: if-split-eq2 Let-def Some-eq-bind-conv)

lemma lift-instr-is-return:
assumes lift-instr F L ret N x  $\Sigma i$  = Some (y,  $\Sigma o$ )
shows Inca.is-return x  $\longleftrightarrow$  Ubx.is-return y
using assms
by (rule lift-instr.elims)
(auto simp add: if-split-eq2 Let-def Some-eq-bind-conv)

lemma lift-intrs-all-not-jump-not-return:
assumes lift-intrs F L ret N  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
shows
list-all ( $\lambda i. \neg$  Inca.is-jump i  $\wedge$   $\neg$  Inca.is-return i) xs  $\longleftrightarrow$ 
list-all ( $\lambda i. \neg$  Ubx.is-jump i  $\wedge$   $\neg$  Ubx.is-return i) ys
using assms
proof (induction xs arbitrary:  $\Sigma i$   $\Sigma o$  ys)
case Nil
then show ?case by (simp add: lift-intrs-def)
next
case (Cons x xs)
from Cons.preds show ?case
apply (simp add: lift-intrs-def bind-eq-Some-conv)
apply (fold lift-intrs-def)
by (auto simp add: Cons.IH lift-instr-is-jump lift-instr-is-return)
qed

lemma lift-intrs-all-butlast-not-jump-not-return:
assumes lift-intrs F L ret N  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
shows
list-all ( $\lambda i. \neg$  Inca.is-jump i  $\wedge$   $\neg$  Inca.is-return i) (butlast xs)  $\longleftrightarrow$ 

```

```

list-all ( $\lambda i. \neg Ubx.is\text{-}jump i \wedge \neg Ubx.is\text{-}return i$ ) (butlast ys)
using lift-intrs-length[OF assms(1)] assms unfolding lift-intrs-def
proof (induction xs ys arbitrary:  $\Sigma i \Sigma o$  rule: list-induct2)
  case Nil
  then show ?case by simp
next
  case (Cons x xs y ys)
  thus ?case
    by (auto simp add: bind-eq-Some-conv lift-instr-is-jump lift-instr-is-return)
qed

lemma lift-instr-sp:
  assumes lift-instr F L ret N x  $\Sigma i = Some (y, \Sigma o)$ 
  shows Subx.sp-instr F ret y  $\Sigma i \Sigma o$ 
  using assms
  apply (induction F L ret N x  $\Sigma i$  rule: lift-instr.induct;
    auto simp: Let-def intro: Subx.sp-instr.intros)
  apply (rule Subx.sp-instr.Op, metis append-take-drop-id)
  apply (rule Subx.sp-instr.OpInl, metis append-take-drop-id)
  apply (auto simp add: bind-eq-Some-conv intro!: Subx.sp-instr.Call, metis append-take-drop-id)
  done

lemma lift-intrs-sp:
  assumes lift-intrs F L ret N  $\Sigma i xs = Some (\Sigma o, ys)$ 
  shows Subx.sp-intrs F ret ys  $\Sigma i \Sigma o$ 
  using assms unfolding lift-intrs-def
proof (induction xs arbitrary:  $\Sigma i \Sigma o ys$ )
  case Nil
  thus ?case by (auto intro: Subx.sp-intrs.Nil)
next
  case (Cons x xs)
  from Cons.premis show ?case
    by (auto simp add: bind-eq-Some-conv intro: Subx.sp-intrs.Cons lift-instr-sp
      Cons.IH)
qed

lemma lift-instr-fun-call-in-range:
  assumes lift-instr F L ret N x  $\Sigma i = Some (y, \Sigma o)$ 
  shows Subx.fun-call-in-range F y
  using assms
  by (induction F L ret N x  $\Sigma i$  rule: lift-instr.induct) (auto simp: Let-def bind-eq-Some-conv)

lemma lift-intrs-all-fun-call-in-range:
  assumes lift-intrs F L ret N  $\Sigma i xs = Some (\Sigma o, ys)$ 
  shows list-all (Subx.fun-call-in-range F) ys
  using assms unfolding lift-intrs-def
  by (auto intro!: monadic-fold-map-list-all intro: lift-instr-fun-call-in-range)

```

```

lemma lift-instr-local-var-in-range:
  assumes lift-instr F L ret N x  $\Sigma i = \text{Some } (y, \Sigma o)$ 
  shows Subx.local-var-in-range N y
  using assms
  by (induction F L ret N x  $\Sigma i$  rule: lift-instr.induct) (auto simp: Let-def bind-eq-Some-conv)

lemma lift-instrs-all-local-var-in-range:
  assumes lift-instrs F L ret N  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
  shows list-all (Subx.local-var-in-range N) ys
  using assms unfolding lift-instrs-def
  by (auto intro!: monadic-fold-map-list-all intro: lift-instr-local-var-in-range)

lemma lift-instr-jump-in-range:
  assumes lift-instr F L ret N x  $\Sigma i = \text{Some } (y, \Sigma o)$ 
  shows Subx.jump-in-range (set L) y
  using assms
  by (induction F L ret N x  $\Sigma i$  rule: lift-instr.induct)
    (auto simp: Let-def bind-eq-Some-conv in-set-member)

lemma lift-instrs-all-jump-in-range:
  assumes lift-instrs F L ret N  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
  shows list-all (Subx.jump-in-range (set L)) ys
  using assms unfolding lift-instrs-def
  by (auto intro!: monadic-fold-map-list-all intro: lift-instr-jump-in-range)

lemma lift-instr-norm:
  lift-instr F L ret N instr1  $\Sigma 1 = \text{Some } (\text{instr2}, \Sigma 2) \implies \text{norm-eq instr1 instr2}$ 
  by (induction instr1  $\Sigma 1$  rule: lift-instr.induct) (auto simp: Let-def bind-eq-Some-conv)

lemma lift-instrs-all-norm:
  assumes lift-instrs F L ret N  $\Sigma 1$  instrs1 = Some ( $\Sigma 2$ , instrs2)
  shows list-all2 norm-eq instrs1 instrs2
  using assms unfolding lift-instrs-def
  by (auto simp: lift-instr-norm elim!: monadic-fold-map-list-all2)

```

29 Optimization

```

context
  fixes load-oracle :: nat  $\Rightarrow$  type option
begin

definition orelse :: 'a option  $\Rightarrow$  'a option  $\Rightarrow$  'a option (infixr <orelse> 55) where
  x orelse y = (case x of Some x'  $\Rightarrow$  Some x' | None  $\Rightarrow$  y)

lemma None-orelse[simp]: None orelse y = y
  by (simp add: orelse-def)

lemma orelse-None[simp]: x orelse None = x
  by (cases x) (simp-all add: orelse-def)

```

```

lemma Some-orelse[simp]: Some x orelse y = Some x
  by (simp add: orelse-def)

lemma orelse-eq-Some-conv:
  x orelse y = Some z  $\longleftrightarrow$  (x = Some z  $\vee$  x = None  $\wedge$  y = Some z)
  by (cases x) simp-all

lemma orelse-eq-SomeE:
  assumes
    x orelse y = Some z and
    x = Some z  $\Longrightarrow$  P and
    x = None  $\Longrightarrow$  y = Some z  $\Longrightarrow$  P
  shows P
  using assms(1)
  unfolding orelse-def
  by (cases x; auto intro: assms(2,3))

fun drop-prefix where
  drop-prefix [] ys = Some ys |
  drop-prefix (x # xs) (y # ys) = (if x = y then drop-prefix xs ys else None) |
  drop-prefix -- = None

lemma drop-prefix-append-prefix[simp]: drop-prefix xs (xs @ ys) = Some ys
  by (induction xs) simp-all

lemma drop-prefix-eq-Some-conv: drop-prefix xs ys = Some zs  $\longleftrightarrow$  ys = xs @ zs
  by (induction xs ys arbitrary: zs rule: drop-prefix.induct)
  (auto simp: if-split-eq1)

fun optim-instr where
  optim-instr -- (IPush d)  $\Sigma$  =
    Some Pair  $\diamond$  (Some IPushUbx1  $\diamond$  (unbox-ubx1 d))  $\diamond$  Some (Some Ubx1 #  $\Sigma$ )
  orelse
    Some Pair  $\diamond$  (Some IPushUbx2  $\diamond$  (unbox-ubx2 d))  $\diamond$  Some (Some Ubx2 #  $\Sigma$ )
  orelse
    Some (IPush d, None #  $\Sigma$ )
  |
  optim-instr -- (IPushUbx1 n)  $\Sigma$  = Some (IPushUbx1 n, Some Ubx1 #  $\Sigma$ ) |
  optim-instr -- (IPushUbx2 b)  $\Sigma$  = Some (IPushUbx2 b, Some Ubx2 #  $\Sigma$ ) |
  optim-instr -- IPop (- #  $\Sigma$ ) = Some (IPop,  $\Sigma$ ) |
  optim-instr -- pc (IGet n)  $\Sigma$  =
    map-option ( $\lambda\tau.$  (IGetUbx  $\tau$  n, Some  $\tau$  #  $\Sigma$ )) (load-oracle pc) orelse
    Some (IGet n, None #  $\Sigma$ ) |
  optim-instr -- pc (IGetUbx  $\tau$  n)  $\Sigma$  = Some (IGetUbx  $\tau$  n, Some  $\tau$  #  $\Sigma$ ) |
  optim-instr -- (ISet n) (None #  $\Sigma$ ) = Some (ISet n,  $\Sigma$ ) |
  optim-instr -- (ISet n) (Some  $\tau$  #  $\Sigma$ ) = Some (ISetUbx  $\tau$  n,  $\Sigma$ ) |
  optim-instr -- (ISetUbx - n) (None #  $\Sigma$ ) = Some (ISet n,  $\Sigma$ ) |
  optim-instr -- (ISetUbx - n) (Some  $\tau$  #  $\Sigma$ ) = Some (ISetUbx  $\tau$  n,  $\Sigma$ ) |

```

```

optim-instr -- pc (ILoad x) (None # Σ) =
  map-option (λτ. (ILoadUbx τ x, Some τ # Σ)) (load-oracle pc) orelse
  Some (ILoad x, None # Σ) |
optim-instr --- (ILoadUbx τ x) (None # Σ) = Some (ILoadUbx τ x, Some τ # Σ) |
  optim-instr --- (IStore x) (None # None # Σ) = Some (IStore x, Σ) |
  optim-instr --- (IStore x) (None # Some τ # Σ) = Some (IStoreUbx τ x, Σ) |
  optim-instr --- (IStoreUbx - x) (None # None # Σ) = Some (IStore x, Σ) |
  optim-instr --- (IStoreUbx - x) (None # Some τ # Σ) = Some (IStoreUbx τ x, Σ) |
  optim-instr --- (IOp op) Σ =
    map-option (λΣo. (IOp op, None # Σo)) (drop-prefix (replicate (Arith op)
  None) Σ) |
  optim-instr --- (IOpInl opinl) Σ =
    let ar = Arith (DeInl opinl) in
    if ar ≤ length Σ then
      case Ubx opinl (take ar Σ) of
        None ⇒ map-option (λΣo. (IOpInl opinl, None # Σo)) (drop-prefix (replicate
  ar None) Σ) |
        Some opubx ⇒ map-option (λΣo. (IOpUbx opubx, snd (TypeOfOp opubx)
  # Σo))
          (drop-prefix (fst (TypeOfOp opubx)) Σ)
        else
          None
    ) |
  optim-instr --- (IOpUbx opubx) Σ =
    let p = TypeOfOp opubx in
    map-option (λΣo. (IOpUbx opubx, snd p # Σo)) (drop-prefix (fst p) Σ)) |
  optim-instr --- (ICJump lt lf) [None] = Some (ICJump lt lf, [])
  optim-instr F -- (ICall f) Σ = do {
    (ar, ret) ← F f;
    Σo ← drop-prefix (replicate ar None) Σ;
    Some (ICall f, replicate ret None @ Σo)
  } |
  optim-instr - ret - IReturn Σ = (if Σ = replicate ret None then Some (IReturn,
  []) else None) |
  optim-instr ----- = None

```

definition optim-instrs **where**

```

optim-instrs F ret ≡ λpc Σi instrs.
  map-option (λ((-, Σo), instrs'). (Σo, instrs'))
    (monadic-fold-map (λ(pc, Σ) instr.
      map-option (λ(instr', Σo). ((Suc pc, Σo), instr')) (optim-instr F ret pc instr
Σ)))
    (pc, Σi) instrs)

```

lemma optim-instrs-Cons-eq-Some-conv:

```

optim-instrs F ret pc Σi (instr # instrs) = Some (Σo, ys) ←→ (∃ y ys' Σ.
  ys = y # ys' ∧

```

```

 $optim\text{-}instr F \text{ret } pc \text{ instr } \Sigma i = Some (y, \Sigma) \wedge$ 
 $optim\text{-}instrs F \text{ret } (\text{Suc } pc) \Sigma \text{ instrs} = Some (\Sigma o, ys')$ 
unfolding optim-instrs-def
by (auto simp: bind-eq-Some-conv)

lemma optim-instrs-length:
assumes optim-instrs F ret pc  $\Sigma i \ xs = Some (\Sigma o, ys)$ 
shows length xs = length ys
using assms unfolding optim-instrs-def
by (auto intro: monadic-fold-map-length)

lemma optim-instrs-not-Nil: optim-instrs F ret pc  $\Sigma i \ xs = Some (\Sigma o, ys) \implies xs \neq [] \longleftrightarrow ys \neq []$ 
using optim-instrs-length by fastforce

lemma optim-instrs-NilD[dest]:
assumes optim-instrs F ret pc  $\Sigma i [] = Some (\Sigma o, ys)$ 
shows  $\Sigma o = \Sigma i \wedge ys = []$ 
using assms
by (simp-all add: optim-instrs-def)

lemma optim-instrs-ConsD[dest]:
assumes optim-instrs F ret pc  $\Sigma i (x \# xs) = Some (\Sigma o, ys)$ 
shows  $\exists y \ ys' \ \Sigma. \ ys = y \# ys' \wedge$ 
 $optim\text{-}instr F \text{ret } pc \ x \ \Sigma i = Some (y, \Sigma) \wedge$ 
 $optim\text{-}instrs F \text{ret } (\text{Suc } pc) \Sigma \ xs = Some (\Sigma o, ys')$ 
using assms
unfolding optim-instrs-def
by (auto simp: bind-eq-Some-conv)

lemma optim-instr-norm:
assumes optim-instr F ret pc instr1  $\Sigma 1 = Some (instr2, \Sigma 2)$ 
shows norm-instr instr1 = norm-instr instr2
using assms
by (cases (F, ret, pc, instr1,  $\Sigma 1$ ) rule: optim-instr.cases)
    (auto simp: ap-option-eq-Some-conv Let-def if-split-eq1 bind-eq-Some-conv option.case-eq-if
      orelse-eq-Some-conv
      dest!: Subx.box-unbox-inverse dest: Subx. $\lambda b x. b \neq x$ -invertible)

lemma optim-instrs-all-norm:
assumes optim-instrs F ret pc  $\Sigma 1 \ instrs1 = Some (\Sigma 2, instrs2)$ 
shows list-all2 ( $\lambda i1 \ i2. \ norm\text{-}instr i1 = norm\text{-}instr i2$ ) instrs1 instrs2
using assms unfolding optim-instrs-def
by (auto simp: optim-instr-norm elim!: monadic-fold-map-list-all2)

lemma optim-instr-is-jump:
assumes optim-instr F ret pc x  $\Sigma i = Some (y, \Sigma o)$ 
shows is-jump x  $\longleftrightarrow$  is-jump y

```

```

using assms
by (cases (F, ret, pc, x,  $\Sigma i$ ) rule: optim-instr.cases;
    simp add: orelse-eq-Some-conv ap-option-eq-Some-conv bind-eq-Some-conv
    Let-def if-split-eq1 option.case-eq-if;
    safe; simp)

lemma optim-instr-is-return:
assumes optim-instr F ret pc x  $\Sigma i$  = Some (y,  $\Sigma o$ )
shows is-return x  $\longleftrightarrow$  is-return y
using assms
by (cases (F, ret, pc, x,  $\Sigma i$ ) rule: optim-instr.cases;
    simp add: orelse-eq-Some-conv ap-option-eq-Some-conv bind-eq-Some-conv
    Let-def if-split-eq1 option.case-eq-if;
    safe; simp)

lemma optim-instrs-all-butlast-not-jump-not-return:
assumes optim-instrs F ret pc  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
shows
  list-all ( $\lambda i. \neg$  is-jump i  $\wedge$   $\neg$  is-return i) (butlast xs)  $\longleftrightarrow$ 
  list-all ( $\lambda i. \neg$  is-jump i  $\wedge$   $\neg$  is-return i) (butlast ys)
using optim-instrs-length[OF assms(1)] assms
proof (induction xs ys arbitrary: pc  $\Sigma i$   $\Sigma o$  rule: list-induct2)
  case Nil
  thus ?case by simp
next
  case (Cons x xs y ys)
  from Cons.preds obtain  $\Sigma$  where
    optim-x: optim-instr F ret pc x  $\Sigma i$  = Some (y,  $\Sigma$ ) and
    optim-xs: optim-instrs F ret (Suc pc)  $\Sigma$  xs = Some ( $\Sigma o$ , ys)
    by auto
  show ?case
    using Cons.hyps
    using optim-x optim-xs
    apply (simp add: Cons.IH optim-instr-is-jump optim-instr-is-return)
    by fastforce
qed

lemma optim-instr-jump-in-range:
assumes optim-instr F ret pc x  $\Sigma i$  = Some (y,  $\Sigma o$ )
shows Subx.jump-in-range L x  $\longleftrightarrow$  Subx.jump-in-range L y
using assms
by (cases (F, ret, pc, x,  $\Sigma i$ ) rule: optim-instr.cases)
  (auto simp: ap-option-eq-Some-conv Let-def if-split-eq1 option.case-eq-if
   bind-eq-Some-conv orelse-eq-Some-conv)

lemma optim-instrs-all-jump-in-range:
assumes optim-instrs F ret pc  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
shows list-all (Subx.jump-in-range L) xs  $\longleftrightarrow$  list-all (Subx.jump-in-range L) ys
using assms

```

```

by (induction xs arbitrary: pc  $\Sigma i \Sigma o$  ys) (auto simp: optim-instr-jump-in-range)

lemma optim-instr-fun-call-in-range:
assumes optim-instr F ret pc x  $\Sigma i$  = Some (y,  $\Sigma o$ )
shows Subx.fun-call-in-range F x  $\longleftrightarrow$  Subx.fun-call-in-range F y
using assms
by (cases (F, ret, pc, x,  $\Sigma i$ ) rule: optim-instr.cases)
(auto simp: ap-option-eq-Some-conv Let-def if-split-eq1 option.case-eq-if
bind-eq-Some-conv orelse-eq-Some-conv)

lemma optim-instrs-all-fun-call-in-range:
assumes optim-instrs F ret pc  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
shows list-all (Subx.fun-call-in-range F) xs  $\longleftrightarrow$  list-all (Subx.fun-call-in-range
F) ys
using assms
by (induction xs arbitrary: pc  $\Sigma i \Sigma o$  ys) (auto simp: optim-instr-fun-call-in-range)

lemma optim-instr-local-var-in-range:
assumes optim-instr F ret pc x  $\Sigma i$  = Some (y,  $\Sigma o$ )
shows Subx.local-var-in-range N x  $\longleftrightarrow$  Subx.local-var-in-range N y
using assms
by (cases (F, ret, pc, x,  $\Sigma i$ ) rule: optim-instr.cases)
(auto simp: ap-option-eq-Some-conv Let-def if-split-eq1 option.case-eq-if
bind-eq-Some-conv orelse-eq-Some-conv)

lemma optim-instrs-all-local-var-in-range:
assumes optim-instrs F ret pc  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
shows list-all (Subx.local-var-in-range N) xs  $\longleftrightarrow$  list-all (Subx.local-var-in-range
N) ys
using assms
by (induction xs arbitrary: pc  $\Sigma i \Sigma o$  ys) (auto simp: optim-instr-local-var-in-range)

lemma optim-instr-sp:
assumes optim-instr F ret pc x  $\Sigma i$  = Some (y,  $\Sigma o$ )
shows Subx.sp-instr F ret y  $\Sigma i \Sigma o$ 
using assms
by (cases (F, ret, pc, x,  $\Sigma i$ ) rule: optim-instr.cases)
(auto simp add: Let-def if-split-eq1 option.case-eq-if
simp: ap-option-eq-Some-conv orelse-eq-Some-conv drop-prefix-eq-Some-conv
bind-eq-Some-conv
intro: Subx.sp-instr.intros)

lemma optim-instrs-sp:
assumes optim-instrs F ret pc  $\Sigma i$  xs = Some ( $\Sigma o$ , ys)
shows Subx.sp-instrs F ret ys  $\Sigma i \Sigma o$ 
using assms
by (induction xs arbitrary: pc  $\Sigma i \Sigma o$  ys)
(auto intro!: Subx.sp-instrs.intros optim-instr-sp)

```

30 Compilation of function definition

```

definition compile-basic-block where
  compile-basic-block F L ret N ≡
    ap-map-prod Some (λi1. do {
      - ← if i1 ≠ [] then Some () else None;
      - ← if list-all (λi. ¬ Inca.is-jump i ∧ ¬ Inca.is-return i) (butlast i1) then
        Some () else None;
      (Σo, i2) ← lift-instrs F L ret N ([] :: type option list) i1;
      if Σo = [] then
        case optim-instrs F ret 0 ([] :: type option list) i2 of
          Some (Σo', i2') ⇒ Some (if Σo' = [] then i2' else i2) |
          None ⇒ Some i2
      else
        None
    })
}

lemma compile-basic-block-rel-prod-all-norm-eq:
  assumes compile-basic-block F L ret N bblock1 = Some bblock2
  shows rel-prod (=) (list-all2 norm-eq) bblock1 bblock2
  using assms
  unfolding compile-basic-block-def
  apply (auto simp add: ap-map-prod-eq-Some-conv bind-eq-Some-conv
    simp: if-split-eq1
    intro: lift-instrs-all-norm
    dest!: optim-instrs-all-norm lift-instrs-all-norm)
  subgoal premises prems for - xs zs ys
    using ⟨list-all2 norm-eq xs ys⟩
    proof (rule list-all2-trans[of norm-eq λi. norm-eq (norm-instr i) norm-eq xs ys
      zs, simplified])
    show list-all2 (λi. norm-eq (norm-instr i)) ys zs
    proof (cases optim-instrs F ret 0 [] ys)
      case None
      with prems show ?thesis by (simp add: list.rel-refl)
    next
      case (Some p)
      with prems show ?thesis
        by (cases p) (auto simp: list.rel-refl intro: optim-instrs-all-norm)
    qed
  qed
  done

lemma list-all-iff-butlast-last:
  assumes xs ≠ []
  shows list-all P xs ↔ list-all P (butlast xs) ∧ P (last xs)
  using assms
  by (induction xs) auto

lemma compile-basic-block-wf:

```

```

assumes compile-basic-block F L ret N x = Some y
shows Subx.wf-basic-block F (set L) ret N y
proof -
obtain f instrs1 instrs2 instrs3 where
  x-def: x = (f, instrs1) and
  y-def: y = (f, instrs3) and
  instrs1 ≠ [] and
  all-not-jump-not-return-intrs1:
    list-all (λi. ¬ Inca.is-jump i ∧ ¬ Inca.is-return i) (butlast instrs1) and
    lift-intrs1: lift-intrs F L ret N ([] :: type option list) instrs1 = Some ([],
  instrs2) and
  instr4-defs: instrs3 = instrs2 ∨
  optim-intrs F ret 0 ([] :: type option list) instrs2 = Some ([], instrs3)
using assms
unfolding compile-basic-block-def
apply (auto simp: ap-map-prod-eq-Some-conv bind-eq-Some-conv if-split-eq1
option.case-eq-if)
by blast

have instrs3 ≠ []
using instr4-defs ⟨instrs1 ≠ []⟩
using lift-intrs-not-Nil[OF lift-intrs1]
by (auto simp: optim-intrs-not-Nil)
moreover have list-all (Subx.local-var-in-range N) instrs3
using instr4-defs lift-intrs1
by (auto dest: lift-intrs-all-local-var-in-range simp: optim-intrs-all-local-var-in-range)
moreover have list-all (Subx.fun-call-in-range F) instrs3
using instr4-defs lift-intrs1
by (auto dest: lift-intrs-all-fun-call-in-range simp: optim-intrs-all-fun-call-in-range)
moreover have list-all (Subx.jump-in-range (set L)) instrs3
using instr4-defs lift-intrs1
by (auto dest: lift-intrs-all-jump-in-range simp: optim-intrs-all-jump-in-range)
moreover have list-all (λi. ¬ Ubx.instr.is-jump i ∧ ¬ Ubx.instr.is-return i)
(butlast instrs3)
using instr4-defs lift-intrs1 all-not-jump-not-return-intrs1
by (auto simp:
  lift-intrs-all-butlast-not-jump-not-return
  optim-intrs-all-butlast-not-jump-not-return)
moreover have Subx.sp-intrs F ret instrs3 [] []
using instr4-defs lift-intrs1
by (auto intro: lift-intrs-sp optim-intrs-sp)
ultimately show ?thesis
by (auto simp: y-def intro!: Subx.wf-basic-blockI)
qed

fun compile-fundef where
compile-fundef F (Fundef bblocks1 ar ret locals) = do {
  - ← if bblocks1 = [] then None else Some ();
  bblocks2 ← ap-map-list (compile-basic-block F (map fst bblocks1)) ret (ar +

```

```

locals)) bblocks1;
  Some (Fundef bblocks2 ar ret locals)
}

lemma compile-fundef-arithes: compile-fundef F fd1 = Some fd2  $\implies$  arity fd1 =
arity fd2
by (cases fd1) (auto simp: bind-eq-Some-conv)

lemma compile-fundef-returns: compile-fundef F fd1 = Some fd2  $\implies$  return fd1 =
return fd2
by (cases fd1) (auto simp: bind-eq-Some-conv)

lemma compile-fundef-locals:
compile-fundef F fd1 = Some fd2  $\implies$  fundef-locals fd1 = fundef-locals fd2
by (cases fd1) (auto simp: bind-eq-Some-conv)

lemma if-then-None-else-Some-eq[simp]:
(if a then None else Some b) = Some c  $\longleftrightarrow$   $\neg a \wedge b = c$ 
(if a then None else Some b) = None  $\longleftrightarrow$  a
by (cases a) simp-all

lemma
assumes compile-fundef F fd1 = Some fd2
shows
  rel-compile-fundef: rel-fundef (=) norm-eq fd1 fd2 (is ?REL) and
  wf-compile-fundef: Subx.wf-fundef F fd2 (is ?WF)
  unfolding atomize-conj
proof (cases fd1)
  case (Fundef bblocks1 ar ret locals)
  with assms obtain bblocks2 where
    bblocks1  $\neq$  [] and
    lift-bblocks1:
      ap-map-list (compile-basic-block F (map fst bblocks1) ret (ar + locals)) bblocks1
= Some bblocks2 and
    fd2-def: fd2 = Fundef bblocks2 ar ret locals
    by (auto simp add: bind-eq-Some-conv)

  show ?REL  $\wedge$  ?WF
  proof (rule conjI)
    show ?REL
      unfolding Fundef fd2-def
      proof (rule fundef.rel-intros)
        show list-all2 (rel-prod (=) (list-all2 norm-eq)) bblocks1 bblocks2
        using lift-bblocks1
        unfolding ap-map-list-iff-list-all2
        by (auto elim: list.rel-mono-strong intro: compile-basic-block-rel-prod-all-norm-eq)
    qed simp-all
  next
    have bblocks2  $\neq$  []

```

```

using <bblocks1 ≠ []> length-ap-map-list[OF lift-bblocks1] by force
moreover have list-all (Subx.wf-basic-block F (fst ` set bblocks1) ret (ar +
locals)) bblocks2
  using lift-bblocks1
  unfolding ap-map-list-iff-list-all2
  by (auto elim!: list-rel-imp-pred2 dest: compile-basic-block-wf)
moreover have fst ` set bblocks1 = fst ` set bblocks2
  using lift-bblocks1
  unfolding ap-map-list-iff-list-all2
  by (induction bblocks1 bblocks2 rule: list.rel-induct)
    (auto simp add: compile-basic-block-def ap-map-prod-eq-Some-conv)
ultimately show ?WF
  unfolding fd2-def
  by (auto intro: Subx.wf-fundefI)
qed
qed

end

end

locale inca-ubx-compiler =
  inca-to-ubx-simulation Finca-empty Finca-get
  for
    Finca-empty and
    Finca-get :: - ⇒ 'fun ⇒ - option +
fixes
  load-oracle :: 'fun ⇒ nat ⇒ type option
begin

```

31 Compilation of function environment

```

definition compile-env-entry where
  compile-env-entry F ≡ λp. ap-map-prod Some (compile-fundef (load-oracle (fst
p)) F) p

lemma rel-compile-env-entry:
  assumes compile-env-entry F (f, fd1) = Some (f, fd2)
  shows rel-fundef (=) norm-eq fd1 fd2
  using assms unfolding compile-env-entry-def
  by (auto simp: ap-map-prod-eq-Some-conv intro!: rel-compile-fundef)

definition compile-env where
  compile-env e ≡ do {
    let fundefs1 = Finca-to-list e;
    fundefs2 ← ap-map-list (compile-env-entry (map-option funtype ∘ Finca-get e))
    fundefs1;
    Some (Subx.Fenv.from-list fundefs2)
  }

```

```

lemma rel-ap-map-list-ap-map-list-compile-env-entries:
  assumes ap-map-list (compile-env-entry F) xs = Some ys
  shows rel-fundefs (Finca-get (Sinca.Fenv.from-list xs)) (Fubx-get (Subx.Fenv.from-list
ys))
  using assms
proof (induction xs arbitrary: ys)
  case Nil
  thus ?case
    using rel-fundefs-empty by simp
next
  case (Cons x xs)
  from Cons.preds obtain y ys' where
    ys-def: ys = y # ys' and
    compile-env-x: compile-env-entry F x = Some y and
    compile-env-xs: ap-map-list (compile-env-entry F) xs = Some ys'
    by (auto simp add: ap-option-eq-Some-conv)

  obtain f fd1 fd2 where
    prods: x = (f, fd1) y = (f, fd2) and compile-fundef (load-oracle f) F fd1 =
  Some fd2
    using compile-env-x
    by (cases x) (auto simp: compile-env-entry-def eq-fst-iff ap-map-prod-eq-Some-conv)

  have rel-fundef (=) norm-eq fd1 fd2
    using compile-env-x[unfolded prods]
    by (auto intro: rel-compile-env-entry)
  thus ?case
    using Cons.IH[OF compile-env-xs, THEN rel-fundefsD]
    unfolding prods ys-def
    unfolding Sinca.Fenv.from-list-correct Subx.Fenv.from-list-correct
    by (auto intro: rel-fundefsI)
qed

lemma rel-fundefs-compile-env:
  assumes compile-env F1 = Some F2
  shows rel-fundefs (Finca-get F1) (Fubx-get F2)
proof -
  from assms obtain xs where
    ap-map-list-F1: ap-map-list (compile-env-entry (map-option funtype o Finca-get
F1)) (Finca-to-list F1) = Some xs and
    F2-def: F2 = Subx.Fenv.from-list xs
    by (auto simp: compile-env-def bind-eq-Some-conv)

  show ?thesis
    using rel-ap-map-list-ap-map-list-compile-env-entries[OF ap-map-list-F1]
    unfolding F2-def Sinca.Fenv.get-from-list-to-list
    by assumption
qed

```

32 Compilation of program

```

fun compile where
  compile (Prog F1 H f) = Some Prog  $\diamond$  compile-env F1  $\diamond$  Some H  $\diamond$  Some f

lemma ap-map-list-cong:
  assumes  $\bigwedge x. x \in set ys \implies f x = g x$  and  $xs = ys$ 
  shows ap-map-list f xs = ap-map-list g ys
  using assms
  by (induction xs arbitrary: ys) auto

lemma compile-env-wf-fundefs:
  assumes compile-env F1 = Some F2
  shows Subx.wf-fundefs (Fubx-get F2)
proof (intro Subx.wf-fundefsI allI)
  fix f
  obtain xs where
    ap-map-list-F1: ap-map-list (compile-env-entry (map-option funtype  $\circ$  Finca-get F1)) (Finca-to-list F1) = Some xs and
    F2-def: F2 = Subx.Fenv.from-list xs
  using assms by (auto simp: compile-env-def bind-eq-Some-conv)

  have rel-map-of-F1-xs:
     $\bigwedge f. rel\text{-option} (\lambda x y. compile\text{-fundef} (load\text{-oracle} f) (map\text{-option funtype} \circ Finca\text{-get} F1) x = Some y)$ 
    (map-of (Finca-to-list F1) f) (map-of xs f)
  using ap-map-list-F1
  by (auto simp: compile-env-entry-def ap-map-prod-eq-Some-conv
        dest: ap-map-list-imp-rel-option-map-of)

  have funtype-F1-eq-funtype-F2:
    map-option funtype  $\circ$  Finca-get F1 = map-option funtype  $\circ$  Fubx-get F2
  proof (rule ext, simp)
    fix x
    show map-option funtype (Finca-get F1 x) = map-option funtype (Fubx-get F2 x)
  using unfolding F2-def Subx.Fenv.from-list-correct Finca.Fenv.to-list-correct[symmetric]
  using rel-map-of-F1-xs[of x]
  by (cases rule: option.rel-cases)
    (simp-all add: funtype-def compile-fundef-arith compile-fundef-returns)
  qed

  show pred-option (Subx.wf-fundef (map-option funtype  $\circ$  Fubx-get F2)) (Fubx-get F2 f)
  proof (cases map-of (Finca-to-list F1) f)
    case None
    thus ?thesis
      using rel-map-of-F1-xs[of f, unfolded None]
      by (simp add: F2-def Subx.Fenv.from-list-correct)

```

```

next
  case (Some fd1)
    show ?thesis
      using rel-map-of-F1-xs[of f, unfolded Some option-rel-Some1]
      unfolding funtype-F1-eq-funtype-F2 F2-def Subx.Fenv.from-list-correct
      by (auto intro: wf-compile-fundef)
  qed
qed

lemma compile-load:
assumes
  compile-p1: compile p1 = Some p2 and
  load: Subx.load p2 s2
shows  $\exists s1. \text{Sinca.load } p1 s1 \wedge \text{match } s1 s2$ 
proof -
  obtain F1 H main where p1-def: p1 = Prog F1 H main
  by (cases p1) simp
  then obtain F2 where
    compile-F1: compile-env F1 = Some F2 and
    p2-def: p2 = Prog F2 H main
    using compile-p1
    by (auto simp: ap-option-eq-Some-conv)

note rel-F1-F2 = rel-fundefs-compile-env[OF compile-F1]

show ?thesis
  using assms(2) unfolding p2-def Subx.load-def
  proof (cases - - - s2 rule: Global.load.cases)
  case (1 fd2)
  then obtain fd1 where
    F1-main: Finca-get F1 main = Some fd1 and rel-fd1-fd2: rel-fundef (=)
    norm-eq fd1 fd2
    using rel-fundefs-Some2[OF rel-F1-F2]
    by auto

let ?s1 = State F1 H [allocate-frame main fd1 [] uninitialized]

show ?thesis
proof (intro exI conjI)
  show Sinca.load p1 ?s1
    unfolding Sinca.load-def p1-def
    using 1 F1-main rel-fd1-fd2
    by (auto simp: rel-fundef-arities intro!: Global.load.intros dest: rel-fundef-body-length)
next
  have Subx.wf-state s2
  unfolding 1
  using compile-F1
  by (auto intro!: Subx.wf-stateI intro: compile-env-wf-fundefs)
  then show match ?s1 s2

```

```

using 1 rel-F1-F2 rel-fd1-fd2
by (auto simp: allocate-frame-def rel-fundef-locales
    simp: rel-fundef-rel-fst-hd-bodies[OF rel-fd1-fd2 disjI2]
    intro!: match.intros rel-stacktraces.intros intro: Subx.sp-intrs.Nil)
qed
qed
qed

interpretation std-to-inca-compiler: compiler where
  step1 = Sinca.step and final1 = final Finca-get Inca.IReturn and load1 =
  Sinca.load and
  step2 = Subx.step and final2 = final Fubx-get Ubx.IReturn and load2 = Subx.load
  and
  match = λ-. match and order = λ- -. False and
  compile = compile
using compile-load
by unfold-locales auto

```

32.1 Completeness of compilation

```

lemma lift-instr-None-preservation:
  assumes lift-instr F L ret N instr Σ = Some (instr', Σ') and list-all ((=) None) Σ
  shows list-all ((=) None) Σ'
  using assms
  by (cases (F, L, ret, N, instr, Σ) rule: lift-instr.cases)
    (auto simp: Let-def bind-eq-Some-conv)

lemma lift-instr-complete:
  assumes
    Sinca.local-var-in-range N instr and
    Sinca.jump-in-range (set L) instr and
    Sinca.fun-call-in-range F instr and
    Sinca.sp-instr F ret instr (length Σ) k and
    list-all ((=) None) Σ
  shows ∃ instr' Σ'. lift-instr F L ret N instr Σ = Some (instr', Σ') ∧ length Σ' = k
  using assms
  by (cases (F, L, ret, N, instr, Σ) rule: lift-instr.cases)
    (auto simp add: in-set-member Let-def
      dest: Map.domD dest!: list-all-eq-const-imp-replicate' elim: Sinca.sp-instr.cases)

lemma lift-intrs-complete:
  fixes Σ :: type option list
  assumes
    list-all (Sinca.local-var-in-range N) instrs and
    list-all (Sinca.jump-in-range (set L)) instrs and
    list-all (Sinca.fun-call-in-range F) instrs and
    Sinca.sp-intrs F ret instrs (length Σ) k and

```

```

list-all ((=) None) Σ
shows ∃Σ' instrs'. lift-instrs F L ret N Σ instrs = Some (Σ', instrs') ∧ length
Σ' = k
using assms
proof (induction instrs arbitrary: Σ)
case Nil
thus ?case
  unfolding lift-instrs-def
  by (auto elim: Sinca.sp-instrs.cases)
next
case (Cons instr instrs')
from Cons.prems(4) obtain k' where
  sp-head: Sinca.sp-instr F ret instr (length Σ) k' and
  sp-tail: Sinca.sp-instrs F ret instrs' k' k
  by (cases rule: Sinca.sp-instrs.cases) simp

have inv-instrs':
  list-all (Sinca.local-var-in-range N) instrs'
  list-all (Sinca.jump-in-range (set L)) instrs'
  list-all (Sinca.fun-call-in-range F) instrs'
using Cons.prems(1–3) by simp-all

from Cons.prems(1–3,5) obtain instr2 Σtmp where
  lift-head: lift-instr F L ret N instr Σ = Some (instr2, Σtmp) and
  length Σtmp = k'
  using lift-instr-complete[OF - - - sp-head, of N L] by auto
hence list-all ((=) None) Σtmp
  by (meson Cons.prems(5) lift-instr-None-preservation)
then obtain instrs2 and Σ' :: type option list where
  lift-tail: lift-instrs F L ret N Σtmp instrs' = Some (Σ', instrs2) and
  length Σ' = k
  using Cons.IH[OF inv-instrs', of Σtmp] sp-tail
  unfolding `length Σtmp = k'`
  by auto
show ?case
proof (intro exI conjI)
  show lift-instrs F L ret N Σ (instr # instrs') = Some (Σ', instr2 # instrs2)
    using lift-head lift-tail
    by (simp add: lift-instrs-def)
next
show length Σ' = k
  by (rule `length Σ' = k`)
qed
qed

lemma optim-instr-complete:
assumes sp: Subx.sp-instr F ret instr Σ Σ'
shows ∃Σ'' instr'. optim-instr O F ret pc instr Σ = Some (instr', Σ'') ∧ length
Σ' = length Σ''

```

```

using sp
proof (cases F ret instr  $\Sigma$   $\Sigma'$  rule: Subx.sp-instr.cases)
  case (Push d)
  thus ?thesis
    by (cases unbox-ubx1 d; cases unbox-ubx2 d) simp-all
next
  case (Get n)
  thus ?thesis
    by (cases O pc) simp-all
next
  case (Load x  $\Sigma$ )
  then show ?thesis
    by (cases O pc) simp-all
next
  case (OpInl opinl  $\Sigma$ )
  then show ?thesis
    by (cases Ubx opinl (replicate (Arity (DefInl opinl)) None))
      (simp-all add: Let-def Subx.Ubx-opubx-type)
qed simp-all

lemma compile-basic-block-complete:
assumes wf-bblock1: Sinca.wf-basic-block F (set L) ret n bblock1
shows  $\exists$  bblock2. compile-basic-block O F L ret n bblock1 = Some bblock2
proof (cases bblock1)
  case (Pair label instrs1)
  moreover obtain instrs2 where
    lift-intrs F L ret n ([] :: type option list) instrs1 = Some ([] , instrs2)
    using wf-bblock1[unfolded Pair, simplified]
    using lift-intrs-complete[of n instrs1 L F ret [] 0]
    by (auto simp: Sinca.wf-basic-block-def)
  ultimately show ?thesis
    using wf-bblock1[unfolded Pair, simplified]
    apply (simp add: compile-basic-block-def ap-map-prod-eq-Some-conv)
    by (cases optim-intrs O F ret 0 [] instrs2) (auto simp: Sinca.wf-basic-block-def)
qed

lemma bind-eq-map-option[simp]:  $x \gg= (\lambda y. \text{Some } (f y)) = \text{map-option } f x$ 
by (cases x) simp-all

lemma compile-fundef-complete:
assumes wf-fd1: Sinca.wf-fundef F fd1
shows  $\exists$  fd2. compile-fundef O F fd1 = Some fd2
proof (cases fd1)
  case (Fundef bblocks ar ret locals)
  then obtain bblock bblocks' where bblocks-def: bblocks = bblock # bblocks'
    using wf-fd1 by (cases bblocks; auto simp: Sinca.wf-fundef-def)
  obtain label instrs where bblock = (label, instrs)
    by (cases bblock) simp
  show ?thesis

```

```

using wf-fd1
by (auto simp add: Fundef Sinca.wf-fundef-def
    intro!: ex-ap-map-list-eq-SomeI intro: compile-basic-block-complete
    elim!: list.pred-mono-strong)
qed

lemma compile-env-entry-complete:
assumes wf-fd1: Sinca.wf-fundef F fd1
shows  $\exists$  fd2. compile-env-entry F (f, fd1) = Some fd2
using compile-fundef-complete[OF wf-fd1]
by (simp add: compile-env-entry-def ap-map-prod-eq-Some-conv)

lemma compile-env-complete:
assumes wf-F1: pred-map (Sinca.wf-fundef (map-option funtype o Finca-get F1))
(Finca-get F1)
shows  $\exists$  F2. compile-env F1 = Some F2
proof -
show ?thesis
using wf-F1
by (auto simp add: compile-env-def
    intro: ex-ap-map-list-eq-SomeI Sinca.Fenv.to-list-list-allI compile-env-entry-complete
    pred-map-get)
qed

theorem compile-complete:
assumes wf-p1: Sinca.wf-prog p1
shows  $\exists$  p2. compile p1 = Some p2
proof (cases p1)
case (Prog F1 H main)
then show ?thesis
using wf-p1 unfolding Sinca.wf-prog-def
by (auto simp: Let-def dest: compile-env-complete)
qed

end

end
theory Op-example
imports OpUbx Global Unboxed-lemmas
begin

```

33 Dynamic values

```

datatype dynamic = DNil | DBool bool | DNum int

definition is-true where
  is-true d  $\equiv$  (d = DBool True)

definition is-false where

```

```

is-false d ≡ (d = DBool False)

interpretation dynval-dynamic: dynval DNil is-true is-false
proof unfold-locales
fix d
show ¬(is-true d ∧ is-false d)
by (cases d) (simp-all add: is-true-def is-false-def)
qed

fun unbox-num :: dynamic ⇒ int option where
unbox-num (DNum n) = Some n |
unbox-num - = None

fun unbox-bool :: dynamic ⇒ bool option where
unbox-bool (DBool b) = Some b |
unbox-bool - = None

interpretation unboxed-dynamic:
unboxedval DNil is-true is-false DNum unbox-num DBool unbox-bool
proof (unfold-locales)
fix d n
show unbox-num d = Some n ⇒ DNum n = d
by (cases d; simp add: )
next
fix d b
show unbox-bool d = Some b ⇒ DBool b = d
by (cases d; simp add: )
qed

```

34 Normal operations

```

datatype op =
OpNeg |
OpAdd |
OpMul

fun ar :: op ⇒ nat where
ar OpNeg = 1 |
ar OpAdd = 2 |
ar OpMul = 2

fun eval-Neg :: dynamic list ⇒ dynamic where
eval-Neg [DBool b] = DBool (¬b) |
eval-Neg [-] = DNil

fun eval-Add :: dynamic list ⇒ dynamic where
eval-Add [DBool x, DBool y] = DBool (x ∨ y) |
eval-Add [DNum x, DNum y] = DNum (x + y) |
eval-Add [-, -] = DNil

```

```

fun eval-Mul :: dynamic list  $\Rightarrow$  dynamic where
  eval-Mul [DBool x, DBool y] = DBool (x  $\wedge$  y) |
  eval-Mul [DNum x, DNum y] = DNum (x * y) |
  eval-Mul [-, -] = DNil

fun eval :: op  $\Rightarrow$  dynamic list  $\Rightarrow$  dynamic where
  eval OpNeg = eval-Neg |
  eval OpAdd = eval-Add |
  eval OpMul = eval-Mul

lemma eval-arith-domain: length xs = ar op  $\implies \exists y. eval op xs = y$ 
  by simp

interpretation op-Op: nary-operations eval ar
  using eval-arith-domain
  by (unfold-locales; simp)

```

35 Inlined operations

```

datatype opinl =
  OpAddNum |
  OpMulNum

fun inl :: op  $\Rightarrow$  dynamic list  $\Rightarrow$  opinl option where
  inl OpAdd [DNum -, DNum -] = Some OpAddNum |
  inl OpMul [DNum -, DNum -] = Some OpMulNum |
  inl - - = None

inductive isinl :: opinl  $\Rightarrow$  dynamic list  $\Rightarrow$  bool where
  isinl OpAddNum [DNum -, DNum -] |
  isinl OpMulNum [DNum -, DNum -]

fun deinl :: opinl  $\Rightarrow$  op where
  deinl OpAddNum = OpAdd |
  deinl OpMulNum = OpMul

lemma inl-inj: inj inl
  unfolding inj-def
  proof (intro allI impI)
    fix x y
    assume inl x = inl y
    thus x = y
      unfolding fun-eq-iff
      by (cases x; cases y; auto elim: inl.elims simp: HOL.eq-commute[of None])
  qed

lemma inl-invertible: inl op xs = Some opinl  $\implies$  deinl opinl = op
  by (induction op xs rule: inl.induct; simp)

```

```

fun eval-AddNum :: dynamic list  $\Rightarrow$  dynamic where
  eval-AddNum [DNum x, DNum y] = DNum (x + y) |
  eval-AddNum [DBool x, DBool y] = DBool (x  $\vee$  y) |
  eval-AddNum [-, -] = DNil

fun eval-MulNum :: dynamic list  $\Rightarrow$  dynamic where
  eval-MulNum [DNum x, DNum y] = DNum (x * y) |
  eval-MulNum [DBool x, DBool y] = DBool (x  $\wedge$  y) |
  eval-MulNum [-, -] = DNil

fun eval-inl :: opinl  $\Rightarrow$  dynamic list  $\Rightarrow$  dynamic where
  eval-inl OpAddNum = eval-AddNum |
  eval-inl OpMulNum = eval-MulNum

lemma eval-AddNum-correct:
  length xs = 2  $\Longrightarrow$  eval-AddNum xs = eval-Add xs
  by (induction xs rule: eval-AddNum.induct; simp)

lemma eval-MulNum-correct:
  length xs = 2  $\Longrightarrow$  eval-MulNum xs = eval-Mul xs
  by (induction xs rule: eval-MulNum.induct; simp)

lemma eval-inl-correct:
  length xs = ar (deinl opinl)  $\Longrightarrow$  eval-inl opinl xs = eval (deinl opinl) xs
  using eval-AddNum-correct eval-MulNum-correct
  by (cases opinl; simp)

lemma inl-isinl:
  inl op xs = Some opinl  $\Longrightarrow$  isinl opinl xs
  by (induction op xs rule: inl.induct; auto intro: isinl.intros)

interpretation op-OpInl: nary-operations-inl eval ar eval-inl inl isinl deinl
  using inl-invertible
  using eval-inl-correct
  using inl-isinl
  by (unfold-locales; simp)

```

36 Unboxed operations

```

datatype opubx =
  OpAddNumUbx

fun ubx :: opinl  $\Rightarrow$  type option list  $\Rightarrow$  opubx option where
  ubx OpAddNum [Some Ubx1, Some Ubx2] = Some OpAddNumUbx |
  ubx - - = None

fun deubx :: opubx  $\Rightarrow$  opinl where
  deubx OpAddNumUbx = OpAddNum

```

```

lemma ubx-invertible: ubx opinl xs = Some opubx  $\Rightarrow$  deubx opubx = opinl
by (induction opinl xs rule: ubx.induct; simp)

fun eval-AddNumUbx where
  eval-AddNumUbx [OpUbx1 x, OpUbx1 y] = Some (OpUbx1 (x + y)) |
  eval-AddNumUbx - = None

fun eval-ubx where
  eval-ubx OpAddNumUbx = eval-AddNumUbx

lemma eval-ubx-correct:
  eval-ubx opubx xs = Some z  $\Rightarrow$ 
    eval-inl (deubx opubx) (map unboxed-dynamic.norm-unboxed xs) = unboxed-dynamic.norm-unboxed
z
  apply (cases opubx; simp)
  by (induction xs rule: eval-AddNumUbx.induct) auto

lemma eval-ubx-to-inl:
  assumes eval-ubx opubx  $\Sigma$  = Some z
  shows inl (deinl (deubx opubx)) (map unboxed-dynamic.norm-unboxed  $\Sigma$ ) =
Some (deubx opubx)
  proof (cases opubx)
    case OpAddNumUbx
    then have eval-AddNumUbx  $\Sigma$  = Some z
      using assms by simp
    then show ?thesis
      using OpAddNumUbx
      by (induction  $\Sigma$  rule: eval-AddNumUbx.induct) simp-all
  qed

```

36.1 Typing

```

fun typeof-opubx :: opubx  $\Rightarrow$  type option list  $\times$  type option where
  typeof-opubx OpAddNumUbx = ([Some Ubx1, Some Ubx1], Some Ubx1)

lemma ubx-imp-typeof-opubx:
  ubx opinl ts = Some opubx  $\Rightarrow$  fst (typeof-opubx opubx) = ts
  by (induction opinl ts rule: ubx.induct; simp)

lemma typeof-opubx-correct:
  typeof-opubx opubx = (map typeof xs, codomain)  $\Rightarrow$ 
     $\exists y.$  eval-ubx opubx xs = Some y  $\wedge$  typeof y = codomain
  proof (induction opubx)
    case OpAddNumUbx
    obtain n1 n2 where xs-def: xs = [OpUbx1 n1, OpUbx1 n2] and codomain =
Some Ubx1
      using OpAddNumUbx[symmetric]
      by (auto dest!: typeof-unboxed-inversion)

```

```

then show ?case by simp
qed

lemma typeof-opubx-complete:
  eval-ubx opubx xs = Some y  $\implies$ 
    typeof-opubx opubx = (map typeof xs, typeof y)
proof (induction opubx)
  case OpAddNumUbx
  then show ?case
    by (auto elim: eval-AddNumUbx.elims)
qed

lemma typeof-opubx-ar: length (fst (typeof-opubx opubx)) = ar (deinl (deubx op-ubx))
  by (induction opubx; simp)

interpretation op-OpUbx:
  nary-operations-ubx
  eval ar eval-inl inl isinl deinl
  DNil is-true is-false DNum unbox-num DBool unbox-bool
  eval-ubx ubx deubx typeof-opubx
  using ubx-invertible
  using eval-ubx-correct
  using eval-ubx-to-inl
  using ubx-imp-typeof-opubx
  using typeof-opubx-correct
  using typeof-opubx-complete
  using typeof-opubx-ar
  by (unfold-locales; (simp; fail) ?)

end
theory Std
  imports List-util Global Op Env Dynamic
  VeriComp.Language
begin

datatype ('dyn, 'var, 'fun, 'label, 'op) instr =
  IPush 'dyn |
  IPop |
  IGet nat |
  ISet nat |
  ILoad 'var |
  IStore 'var |
  IOp 'op |
  ICJump 'label 'label |
  ICcall 'fun |
  is-return: IReturn

```

```

locale std =
  Fenv: env F-empty F-get F-add F-to-list +
  Henv: env heap-empty heap-get heap-add heap-to-list +
  dynval uninitialized is-true is-false +
  nary-operations Op Arity
  for
    — Functions environment
    F-empty and
    F-get :: 'fenv  $\Rightarrow$  'fun  $\Rightarrow$  ('label, ('dyn, 'var, 'fun, 'label, 'op) instr) fundef option
    and
    F-add and F-to-list and

    — Memory heap
    heap-empty and
    heap-get :: 'henv  $\Rightarrow$  'var  $\times$  'dyn  $\Rightarrow$  'dyn option and
    heap-add and heap-to-list and

    — Dynamic values
    uninitialized :: 'dyn and is-true and is-false and

    — n-ary operations
    Op :: 'op  $\Rightarrow$  'dyn list  $\Rightarrow$  'dyn and Arity
begin

  inductive step (infix  $\leftrightarrow$  55) where
    step-push:
      next-instr (F-get F) f l pc = Some (IPush d)  $\Rightarrow$ 
      State F H (Frame f l pc R  $\Sigma \# st$ )  $\rightarrow$  State F H (Frame f l (Suc pc) R (d #  $\Sigma$ )  $\# st$ ) |
    step-pop:
      next-instr (F-get F) f l pc = Some (IPop)  $\Rightarrow$ 
      State F H (Frame f l pc R (d #  $\Sigma$ )  $\# st$ )  $\rightarrow$  State F H (Frame f l (Suc pc) R  $\Sigma \# st$ ) |
    step-get:
      next-instr (F-get F) f l pc = Some (IGet n)  $\Rightarrow$ 
      n < length R  $\Rightarrow$  d = R ! n  $\Rightarrow$ 
      State F H (Frame f l pc R  $\Sigma \# st$ )  $\rightarrow$  State F H (Frame f l (Suc pc) R (d #  $\Sigma$ )  $\# st$ ) |
    step-set:
      next-instr (F-get F) f l pc = Some (ISet n)  $\Rightarrow$ 
      n < length R  $\Rightarrow$  R' = R[n := d]  $\Rightarrow$ 
      State F H (Frame f l pc R (d #  $\Sigma$ )  $\# st$ )  $\rightarrow$  State F H (Frame f l (Suc pc) R'  $\Sigma \# st$ ) |
    step-load:
      next-instr (F-get F) f l pc = Some (ILoad x)  $\Rightarrow$ 

```

$\text{heap-get } H \ (x, y) = \text{Some } d \implies$
 $\text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ (y \ \# \ \Sigma) \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } pc) \ R \ (d \ \# \ \Sigma) \ \# \ st) \mid$

step-store:
 $\text{next-instr } (F\text{-get } F) \ f \ l \ pc = \text{Some } (\text{IStore } x) \implies$
 $\text{heap-add } H \ (x, y) \ d = H' \implies$
 $\text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ (y \ \# \ d \ \# \ \Sigma) \ \# \ st) \rightarrow \text{State } F \ H' \ (\text{Frame } f \ l \ (\text{Suc } pc) \ R \ \Sigma \ \# \ st) \mid$

step-op:
 $\text{next-instr } (F\text{-get } F) \ f \ l \ pc = \text{Some } (\text{IOp } op) \implies$
 $\text{Arith } op = ar \implies ar \leq \text{length } \Sigma \implies \text{Op } op \ (\text{take } ar \ \Sigma) = x \implies$
 $\text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } pc) \ R \ (x \ \# \ \text{drop } ar \ \Sigma) \ \# \ st) \mid$

step-cjump:
 $\text{next-instr } (F\text{-get } F) \ f \ l \ pc = \text{Some } (\text{ICJump } l_t \ l_f) \implies$
 $\text{is-true } d \wedge l' = l_t \vee \text{is-false } d \wedge l' = l_f \implies$
 $\text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ (d \ \# \ \Sigma) \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l' \ 0 \ R \ \Sigma \ \# \ st) \mid$

step-call:
 $\text{next-instr } (F\text{-get } F) \ f \ l \ pc = \text{Some } (\text{ICall } g) \implies$
 $F\text{-get } F \ g = \text{Some } gd \implies \text{arity } gd \leq \text{length } \Sigma \implies$
 $\text{frame}_g = \text{allocate-frame } g \ gd \ (\text{take } (\text{arity } gd) \ \Sigma) \ \text{uninitialized} \implies$
 $\text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \rightarrow \text{State } F \ H \ (\text{frame}_g \ \# \ \text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \mid$

step-return:
 $\text{next-instr } (F\text{-get } F) \ g \ l_g \ pc_g = \text{Some } \text{IReturn} \implies$
 $F\text{-get } F \ g = \text{Some } gd \implies \text{arity } gd \leq \text{length } \Sigma_f \implies$
 $\text{length } \Sigma_g = \text{return } gd \implies$
 $\text{frame}_f' = \text{Frame } f \ l_f \ (\text{Suc } pc_f) \ R_f \ (\Sigma_g @ \text{drop } (\text{arity } gd) \ \Sigma_f) \implies$
 $\text{State } F \ H \ (\text{Frame } g \ l_g \ pc_g \ R_g \ \Sigma_g \ \# \ \text{Frame } f \ l_f \ pc_f \ R_f \ \Sigma_f \ \# \ st) \rightarrow \text{State } F \ H \ (\text{frame}_f' \ \# \ st)$

lemma step-deterministic:
assumes $s1 \rightarrow s2$ **and** $s1 \rightarrow s3$
shows $s2 = s3$
using *assms*
apply (*cases* $s1 \ s2$ *rule*: *step.cases*)
apply (*auto elim*: *step.cases dest*: *is-true-and-is-false-implies-False*) [3]
apply (*auto elim*: *step.cases dest*: *is-true-and-is-false-implies-False*) [3]
apply (*auto elim*: *step.cases dest*: *is-true-and-is-false-implies-False*) [3]
apply (*auto elim*: *step.cases dest*: *is-true-and-is-false-implies-False*) [1]
done

lemma step-right-unique: *right-unique step*

```

using step-deterministic
by (rule right-uniqueI)

lemma final-finished:
  assumes final F-get IReturn s
  shows finished step s
  unfolding finished-def
  proof (rule notI)
    assume  $\exists x.$  step s x
    then obtain s' where step s s' by auto
    thus False
    using ⟨final F-get IReturn s⟩
    by (auto simp: state-ok-def elim!: step.cases final.cases)
  qed

sublocale semantics step final F-get IReturn
  using final-finished step-deterministic
  by unfold-locales

definition load where
  load ≡ Global.load F-get uninitialized

sublocale language step final F-get IReturn load
  by unfold-locales

end

end
theory Std-to-Inca-simulation
  imports Global List-util Std Inca
    VeriComp.Simulation
begin

```

37 Generic definitions

```

locale std-inca-simulation =
  Sstd: std
  Fstd-empty Fstd-get Fstd-add Fstd-to-list
  heap-empty heap-get heap-add heap-to-list
  uninitialized is-true is-false
  Ⓣp Ⓣarity +
  Sinca: inca
  Finca-empty Finca-get Finca-add Finca-to-list
  heap-empty heap-get heap-add heap-to-list
  uninitialized is-true is-false
  Ⓣp Ⓣarity Inl Ⓣp Inl IsInl DeInl
for
  — Functions environments
  Fstd-empty and

```

```

Fstd-get :: 'fenv-std  $\Rightarrow$  'fun  $\Rightarrow$  ('label, ('dyn, 'var, 'fun, 'label, 'op) Std.instr)
fundef option and
Fstd-add and Fstd-to-list and

Finca-empty and
Finca-get :: 'fenv-inca  $\Rightarrow$  'fun  $\Rightarrow$  ('label, ('dyn, 'var, 'fun, 'label, 'op, 'opinl)
Inca.instr) fundef option and
Finca-add and Finca-to-list and

— Memory heap
heap-empty and
heap-get :: 'henv  $\Rightarrow$  'var  $\times$  'dyn  $\Rightarrow$  'dyn option and
heap-add and heap-to-list and

— Dynamic values
uninitialized :: 'dyn and is-true and is-false and

— n-ary operations
Op :: 'op  $\Rightarrow$  'dyn list  $\Rightarrow$  'dyn and Arity and
InOp and Inl and IsInl and DeInl :: 'opinl  $\Rightarrow$  'op
begin

fun norm-instr where
norm-instr (Inca.IPush d) = Std.IPush d |
norm-instr Inca.IPop = Std.IPop |
norm-instr (Inca.IGet n) = Std.IGet n |
norm-instr (Inca.ISet n) = Std.ISet n |
norm-instr (Inca.ILoad x) = Std.ILoad x |
norm-instr (Inca.IStore x) = Std.IStore x |
norm-instr (Inca.IOp op) = Std.IOp op |
norm-instr (Inca.IOpInl opinl) = Std.IOp (DeInl opinl) |
norm-instr (Inca.ICJump lt lf) = Std.ICJump lt lf |
norm-instr (Inca.ICall x) = Std.ICall x |
norm-instr Inca.IReturn = Std.IReturn

abbreviation norm-eq where
norm-eq x y  $\equiv$  norm-instr y = x

definition rel-fundefs where
rel-fundefs f g = ( $\forall x.$  rel-option (rel-fundef (=) norm-eq) (f x) (g x))

lemma rel-fundefsI:
assumes  $\bigwedge x.$  rel-option (rel-fundef (=) norm-eq) (F1 x) (F2 x)
shows rel-fundefs F1 F2
using assms
by (simp add: rel-fundefs-def)

lemma rel-fundefsD:
assumes rel-fundefs F1 F2

```

```

shows rel-option (rel-fundef (=) norm-eq) (F1 x) (F2 x)
using assms
by (simp add: rel-fundefs-def)

lemma rel-fundefs-next-instr:
assumes rel-F1-F2: rel-fundefs F1 F2
shows rel-option norm-eq (next-instr F1 f l pc) (next-instr F2 f l pc)
proof -
have rel-option (rel-fundef (=) norm-eq) (F1 f) (F2 f)
  using rel-F1-F2[unfolded rel-fundefs-def] by simp
thus ?thesis
proof (cases rule: option.rel-cases)
case None
then show ?thesis by (simp add: next-instr-def)
next
case (Some fd1 fd2)
hence rel-option (list-all2 norm-eq) (map-of (body fd1) l) (map-of (body fd2)
l)
  by (auto elim!: fundef.rel-cases intro: rel-option-map-of)
then show ?thesis
  by (cases rule: option.rel-cases;
simp add: next-instr-def instr-at-def Some list-all2-conv-all-nth)
qed
qed

lemma rel-fundefs-next-instr1:
assumes rel-F1-F2: rel-fundefs F1 F2 and next-instr1: next-instr F1 f l pc =
Some instr1
shows ∃ instr2. next-instr F2 f l pc = Some instr2 ∧ norm-eq instr1 instr2
using rel-fundefs-next-instr[OF rel-F1-F2, of f l pc]
 unfolding next-instr1
 unfolding option-rel-Some1
by assumption

lemma rel-fundefs-next-instr2:
assumes rel-F1-F2: rel-fundefs F1 F2 and next-instr2: next-instr F2 f l pc =
Some instr2
shows ∃ instr1. next-instr F1 f l pc = Some instr1 ∧ norm-eq instr1 instr2
using rel-fundefs-next-instr[OF rel-F1-F2, of f l pc]
 unfolding next-instr2
 unfolding option-rel-Some2
by assumption

lemma rel-fundefs-empty: rel-fundefs (λ-. None) (λ-. None)
by (simp add: rel-fundefs-def)

lemma rel-fundefs-None1:
assumes rel-fundefs f g and f x = None
shows g x = None

```

```

by (metis assms rel-fundefs-def rel-option-None1)

lemma rel-fundefs-None2:
assumes rel-fundefs f g and g x = None
shows f x = None
by (metis assms rel-fundefs-def rel-option-None2)

lemma rel-fundefs-Some1:
assumes rel-fundefs f g and f x = Some y
shows ∃ z. g x = Some z ∧ rel-fundef (=) norm-eq y z
proof -
from assms(1) have rel-option (rel-fundef (=) norm-eq) (f x) (g x)
  unfolding rel-fundefs-def by simp
with assms(2) show ?thesis
  by (simp add: option-rel-Some1)
qed

lemma rel-fundefs-Some2:
assumes rel-fundefs f g and g x = Some y
shows ∃ z. f x = Some z ∧ rel-fundef (=) norm-eq z y
proof -
from assms(1) have rel-option (rel-fundef (=) norm-eq) (f x) (g x)
  unfolding rel-fundefs-def by simp
with assms(2) show ?thesis
  by (simp add: option-rel-Some2)
qed

lemma rel-fundefs-rel-option:
assumes rel-fundefs f g and ⋀ x y. rel-fundef (=) norm-eq x y ==> h x y
shows rel-option h (f z) (g z)
proof -
have rel-option (rel-fundef (=) norm-eq) (f z) (g z)
  using assms(1)[unfolded rel-fundefs-def] by simp
then show ?thesis
  unfolding rel-option-unfold
  by (auto simp add: assms(2))
qed

lemma rel-fundefs-rewriteI2:
assumes
  rel-F1-F2: rel-fundefs (Fstd-get F1) (Finca-get F2) and
  next-instr1: next-instr (Fstd-get F1) f l pc = Some instr1
  norm-eq instr1 instr2'
shows rel-fundefs (Fstd-get F1)
  (Finca-get (Sinca.Fenv.map-entry F2 f (λfd. rewrite-fundef-body fd l pc instr2'))))
  (is rel-fundefs (Fstd-get ?F1') (Finca-get ?F2'))
proof (rule rel-fundefsI)
fix x
show rel-option (rel-fundef (=) norm-eq) (Fstd-get ?F1' x) (Finca-get ?F2' x)

```

```

proof (cases  $f = x$ )
  case True
    show ?thesis
      using rel-F1-F2[THEN rel-fundefsD, of  $x$ ] True next-instr1 assms(3)
      by (cases rule: option.rel-cases)
        (auto intro!: rel-fundef-rewrite-body2' dest!: next-instrD)
  next
    case False
      then show ?thesis
        using rel-F1-F2[THEN rel-fundefsD, of  $x$ ] by simp
      qed
  qed

```

38 Simulation relation

```

inductive match (infix  $\sim$  55) where
  wf-fundefs (Fstd-get  $F_1$ )  $\implies$ 
  rel-fundefs (Fstd-get  $F_1$ ) (Finca-get  $F_2$ )  $\implies$ 
  (State  $F_1 H st$ )  $\sim$  (State  $F_2 H st$ )

```

39 Backward simulation

```

lemma backward-lockstep-simulation:
  assumes Sinca.step  $s_2 s_2'$  and match  $s_1 s_2$ 
  shows  $\exists s_1'. Sstd.step s_1 s_1' \wedge \text{match } s_1' s_2'$ 
proof -
  from assms(2) obtain  $F_1 F_2 H st$  where
     $s_1\text{-def: } s_1 = \text{State } F_1 H st$  and
     $s_2\text{-def: } s_2 = \text{State } F_2 H st$  and
    ok-F1: wf-fundefs (Fstd-get  $F_1$ ) and
    rel-F1-F2: rel-fundefs (Fstd-get  $F_1$ ) (Finca-get  $F_2$ )
  by (auto elim: match.cases)

note rel-F1-F2-next-instr = rel-fundefs-next-instr[OF rel-F1-F2]

from assms(1) show ?thesis
  unfolding  $s_1\text{-def } s_2\text{-def}$ 
  proof (induction State  $F_2 H st s_2'$  rule: Sinca.step.induct)
    case (step-push  $f l pc d R \Sigma st'$ )
    show ?case (is  $\exists x. ?STEP x \wedge ?MATCH x$ )
    proof -
      let  $?s_1' = \text{State } F_1 H (\text{Frame } f l (\text{Suc } pc) R (d \# \Sigma) \# st')$ 
      have ?STEP ?s1'
      using step-push rel-F1-F2-next-instr[of  $f l pc$ ]
      by (auto intro!: Sstd.step-push simp: option-rel-Some2)
      moreover have ?MATCH ?s1'
      using ok-F1 rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by blast

```

```

qed
next
case (step-pop f l pc R d Σ st')
show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof -
let ?s1' = State F1 H (Frame f l (Suc pc) R Σ # st')
have ?STEP ?s1'
using step-pop rel-F1-F2-next-instr[of f l pc]
by (auto intro!: Sstd.step-pop simp: option-rel-Some2)
moreover have ?MATCH ?s1'
using ok-F1 rel-F1-F2 by (auto intro: match.intros)
ultimately show ?thesis by blast
qed
next
case (step-get f l pc n R d Σ st')
show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof -
let ?s1' = State F1 H (Frame f l (Suc pc) R (d # Σ) # st')
have ?STEP ?s1'
using step-get rel-F1-F2-next-instr[of f l pc]
by (auto intro: Sstd.step-get simp: option-rel-Some2)
moreover have ?MATCH ?s1'
using ok-F1 rel-F1-F2 by (auto intro: match.intros)
ultimately show ?thesis by blast
qed
next
case (step-set f l pc n R R' d Σ st')
show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof -
let ?s1' = State F1 H (Frame f l (Suc pc) R' Σ # st')
have ?STEP ?s1'
using step-set rel-F1-F2-next-instr[of f l pc]
by (auto intro: Sstd.step-set simp: option-rel-Some2)
moreover have ?MATCH ?s1'
using ok-F1 rel-F1-F2 by (auto intro: match.intros)
ultimately show ?thesis by blast
qed
next
case (step-load f l pc x y d R Σ st')
show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof -
let ?s1' = State F1 H (Frame f l (Suc pc) R (d # Σ) # st')
have ?STEP ?s1'
using step-load rel-F1-F2-next-instr[of f l pc]
by (auto intro!: Sstd.step-load simp: option-rel-Some2)
moreover have ?MATCH ?s1'
using ok-F1 rel-F1-F2 by (auto intro: match.intros)
ultimately show ?thesis by blast
qed

```

```

next
case (step-store f l pc x y d H' R Σ st')
show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof -
  let ?s1' = State F1 H' (Frame f l (Suc pc) R Σ # st')
  have ?STEP ?s1'
    using step-store rel-F1-F2-next-instr[of f l pc]
    by (auto intro!: Sstd.step-store simp: option-rel-Some2)
  moreover have ?MATCH ?s1'
    using ok-F1 rel-F1-F2 by (auto intro: match.intros)
  ultimately show ?thesis by blast
qed
next
case (step-op f l pc op ar Σ x R st')
show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof -
  let ?s1' = State F1 H (Frame f l (Suc pc) R (x # drop ar Σ) # st')
  have ?STEP ?s1'
    using step-op rel-F1-F2-next-instr[of f l pc]
    by (auto intro!: Sstd.step-op simp: option-rel-Some2)
  moreover have ?MATCH ?s1'
    using ok-F1 rel-F1-F2 by (auto intro: match.intros)
  ultimately show ?thesis by blast
qed
next
case (step-op-inl f l pc op ar Σ opinl x F2' R st')
then obtain instr1 where
  next-instr1: next-instr (Fstd-get F1) f l pc = Some instr1 and
  norm-eq-instr1-instr2: norm-eq instr1 (Inca.IOp op)
  using rel-F1-F2-next-instr[of f l pc] by (simp add: option-rel-Some2)
then show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof (cases instr1)
  case (IOp op')
  hence op' = op using norm-eq-instr1-instr2 by simp
  let ?s1' = State F1 H (Frame f l (Suc pc) R (x # drop ar Σ) # st')
  have ?STEP ?s1'
    using step-op-inl next-instr1 IOp `op' = op`
    using Sinca.InlOp-correct Sinca.Inl-invertible
    by (auto intro: Sstd.step-op)
  moreover have ?MATCH ?s1'
    using ok-F1 step-op-inl next-instr1 IOp `op' = op`
    using Sinca.Inl-invertible
    by (auto intro!: match.intros intro: rel-fundefs-rewriteI2[OF rel-F1-F2])
  ultimately show ?thesis by blast
qed simp-all
next
case (step-op-inl-hit f l pc opinl ar Σ x R st')
show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof -

```

```

let ?s1' = State F1 H (Frame f l (Suc pc) R (x # drop ar Σ) # st')
have ?STEP ?s1'
  using step-op-inl-hit rel-F1-F2-next-instr[of f l pc]
  using Sinca.InlOp-correct
  by (auto intro: Sstd.step-op simp: option-rel-Some2)
moreover have ?MATCH ?s1'
  using ok-F1 rel-F1-F2 by (auto intro: match.intros)
ultimately show ?thesis by blast
qed
next
case (step-op-inl-miss f l pc opinl ar Σ x F' R st')
then obtain instr1 where
  next-instr1: next-instr (Fstd-get F1) f l pc = Some instr1 and
  norm-eq-instr1-instr2: norm-eq instr1 (Inca.IOpInl opinl)
  using rel-F1-F2-next-instr[of f l pc] by (simp add: option-rel-Some2)
then show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
proof (cases instr1)
  case (IOp op)
  hence op = DeInl opinl using norm-eq-instr1-instr2 by simp
  let ?s1' = State F1 H (Frame f l (Suc pc) R (x # drop ar Σ) # st')
  have ?STEP ?s1'
    using step-op-inl-miss next-instr1 IOp `op = DeInl opinl`
    using Sinca.InlOp-correct
    by (auto intro: Sstd.step-op)
moreover have ?MATCH ?s1'
  using ok-F1 step-op-inl-miss next-instr1 IOp `op = DeInl opinl`
  using Sinca.Inl-invertible
  by (auto intro!: match.intros intro: rel-fundefs-rewriteI2[OF rel-F1-F2])
ultimately show ?thesis by blast
qed simp-all
next
case (step-cjump f l pc lt lf d l' R Σ st')
show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
proof -
  let ?s1' = State F1 H (Frame f l' 0 R Σ # st')
  have ?STEP ?s1'
    using step-cjump rel-F1-F2-next-instr[of f l pc]
    by (auto intro: Sstd.step-cjump simp: option-rel-Some2)
moreover have ?MATCH ?s1'
  using ok-F1 rel-F1-F2 by (auto intro: match.intros)
ultimately show ?thesis by blast
qed
next
case (step-call f lf pcf g gd2 Σ_f frame_g R_f st')
then obtain gd1 where
  F1-g: Fstd-get F1 g = Some gd1 and
  rel-gd1-gd2: rel-fundef (=) norm-eq gd1 gd2
  using rel-fundefs-Some2[OF rel-F1-F2] by blast

```

```

have same-fst-label-gd1-gd2: fst (hd (body gd1)) = fst (hd (body gd2))
  using wf-fundefs-imp-wf-fundef[OF ok-F1 F1-g, unfolded wf-funef-def] rel-gd1-gd2
  by (auto elim!: fundef.rel-cases dest: list-all2-rel-prod-fst-hd)

show ?case (is  $\exists x. \ ?STEP\ x \wedge \ ?MATCH\ x$ )
proof -
  let  $\Sigma_g = take (arity\ gd1) \Sigma_f$ 
  let  $?lg = (fst (hd (body\ gd1)))$ 
  let  $?s1' = State\ F1\ H (frame_g \# Frame\ f\ l_f\ pc_f\ R_f\ \Sigma_f \# st')$ 
  have  $?STEP\ ?s1'$ 
    using step-call rel-F1-F2-next-instr[of\ l_f\ pc_f]
    using F1-g same-fst-label-gd1-gd2 rel-gd1-gd2
    by (auto intro!: Sstd.step-call
      simp: option-rel-Some2 allocate-frame-def rel-funef-arithies rel-funef-locals)
  moreover have  $?MATCH\ ?s1'$ 
    using ok-F1 rel-F1-F2
    by (auto intro: match.intros)
  ultimately show ?thesis by auto
qed
next
  case (step-return g l_g pc_g gd2  $\Sigma_f \Sigma_g$  frame_f' f l_f pc_f R_f R_g st')
  then obtain gd1 where
    F1-g: Fstd-get F1 g = Some gd1 and
    rel-gd1-gd2: rel-funef (=) norm-eq gd1 gd2
    using rel-fundefs-Some2[OF rel-F1-F2] by blast
  obtain instr1 where
    next-instr1: next-instr (Fstd-get F1) g l_g pc_g = Some instr1 and
    norm-eq-instr1-instr2: norm-eq instr1 Inca.IReturn
  using step-return rel-F1-F2-next-instr[of\ l_g\ pc_g] by (simp add: option-rel-Some2)
  then show ?case (is  $\exists x. \ ?STEP\ x \wedge \ ?MATCH\ x$ )
  proof (cases instr1)
    case IReturn
    let  $?s1' = State\ F1\ H (frame_f' \# st')$ 
    have  $?STEP\ ?s1'$ 
      using step-return next-instr1 IReturn
      using F1-g rel-gd1-gd2
      by (auto intro!: Sstd.step-return simp: rel-funef-arithies rel-funef-return)
    moreover have  $?MATCH\ ?s1'$ 
      using ok-F1 rel-F1-F2 by (auto intro: match.intros)
    ultimately show ?thesis by auto
  qed simp-all
qed
lemma match-final-backward:
  assumes match s1 s2 and final-s2: final Finca-get Inca.IReturn s2
  shows final Fstd-get Std.IReturn s1
  using ⟨match s1 s2⟩
  proof (cases s1 s2 rule: match.cases)

```

```

case (1 F1 F2 H st)
show ?thesis
  using final-s2[unfolded 1]
proof (cases - - State F2 H st rule: final.cases)
  case (finalI f l pc R Σ)
  then show ?thesis
    using 1
    by (auto intro!: final.intros dest: rel-fundefs-next-instr2)
  qed
qed

sublocale std-inca-simulation:
  backward-simulation where
    step1 = Sstd.step and final1 = final Fstd-get Std.IReturn and
    step2 = Sinca.step and final2 = final Finca-get Inca.IReturn and
    order = λ- -. False and match = λ-. match
  using match-final-backward backward-lockstep-simulation
  lockstep-to-plus-backward-simulation[of match Sinca.step Sstd.step]
  by unfold-locales auto

```

40 Forward simulation

```

lemma forward-lockstep-simulation:
  assumes Sstd.step s1 s1' and match s1 s2
  shows ∃ s2'. Sinca.step s2 s2' ∧ s1' ~ s2'
proof -
  from assms(2) obtain F1 F2 H st where
    s1-def: s1 = State F1 H st and
    s2-def: s2 = State F2 H st and
    ok-F1: wf-fundefs (Fstd-get F1) and
    rel-F1-F2: rel-fundefs (Fstd-get F1) (Finca-get F2)
  by (auto elim: match.cases)

note rel-F1-F2-next-instr = rel-fundefs-next-instr[OF rel-F1-F2]

from assms(1) show ?thesis
  unfolding s1-def s2-def
proof(induction State F1 H st s1' rule: Sstd.step.induct)
  case (step-push f l pc d R Σ st')
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
proof -
  let ?s1' = State F2 H (Frame f l (Suc pc) R (d # Σ) # st')
  have ?STEP ?s1'
  using step-push rel-F1-F2-next-instr[of f l pc]
  by (auto intro!: Sinca.step-push elim: norm-instr.elims simp: option-rel-Some1)
  moreover have ?MATCH ?s1'
  using ok-F1 rel-F1-F2 by (auto intro: match.intros)
  ultimately show ?thesis by blast
qed

```

```

next
  case (step-pop f l pc R d  $\Sigma$  st')
  show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH x)
  proof -
    let ?s1' = State F2 H (Frame f l (Suc pc) R  $\Sigma$  # st')
    have ?STEP ?s1'
      using step-pop rel-F1-F2-next-instr[of f l pc]
      by (auto intro: Sinca.step-pop elim: norm-instr.elims simp: option-rel-Some1)
      moreover have ?MATCH ?s1'
        using ok-F1 rel-F1-F2 by (auto intro: match.intros)
        ultimately show ?thesis by blast
  qed
next
  case (step-get f l pc n R d  $\Sigma$  st')
  show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH x)
  proof -
    let ?s1' = State F2 H (Frame f l (Suc pc) R (d #  $\Sigma$ ) # st')
    have ?STEP ?s1'
      using step-get rel-F1-F2-next-instr[of f l pc]
      by (auto intro: Sinca.step-get elim: norm-instr.elims simp: option-rel-Some1)
      moreover have ?MATCH ?s1'
        using ok-F1 rel-F1-F2 by (auto intro: match.intros)
        ultimately show ?thesis by blast
  qed
next
  case (step-set f l pc n R R' d  $\Sigma$  st')
  show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH x)
  proof -
    let ?s1' = State F2 H (Frame f l (Suc pc) R'  $\Sigma$  # st')
    have ?STEP ?s1'
      using step-set rel-F1-F2-next-instr[of f l pc]
      by (auto intro: Sinca.step-set elim: norm-instr.elims simp: option-rel-Some1)
      moreover have ?MATCH ?s1'
        using ok-F1 rel-F1-F2 by (auto intro: match.intros)
        ultimately show ?thesis by blast
  qed
next
  case (step-load f l pc x y d R  $\Sigma$  st')
  show ?case (is  $\exists x$ . ?STEP x  $\wedge$  ?MATCH x)
  proof -
    let ?s1' = State F2 H (Frame f l (Suc pc) R (d #  $\Sigma$ ) # st')
    have ?STEP ?s1'
      using step-load rel-F1-F2-next-instr[of f l pc]
      by (auto intro: Sinca.step-load elim: norm-instr.elims simp: option-rel-Some1)
      moreover have ?MATCH ?s1'
        using ok-F1 rel-F1-F2 by (auto intro: match.intros)
        ultimately show ?thesis by blast
  qed
next

```

```

case (step-store f l pc x y d H' R Σ st')
show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
proof -
  let ?s1' = State F2 H' (Frame f l (Suc pc) R Σ # st')
  have ?STEP ?s1'
    using step-store rel-F1-F2-next-instr[of f l pc]
    by (auto intro: Sinca.step-store elim: norm-instr.elims simp: option-rel-Some1)
    moreover have ?MATCH ?s1'
      using ok-F1 rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by blast
  qed
next
  case (step-op f l pc op ar Σ x R st')
  then obtain instr2 where
    next-instr2: next-instr (Finca-get F2) f l pc = Some instr2 and
    norm-eq-instr2: norm-eq (Std.IOp op) instr2
    using rel-F1-F2-next-instr[of f l pc] by (auto simp: option-rel-Some1)
  thus ?case (is ∃x. ?STEP x ∧ ?MATCH x)
  proof (cases instr2)
    case (IOp op')
    then have op' = op using norm-eq-instr2 by simp
    show ?thesis
    proof (cases Inl op' (take ar Σ))
      case None
      let ?s2' = State F2 H (Frame f l (Suc pc) R (x # drop ar Σ) # st')
      have ?STEP ?s2'
        using None IOp step-op ⟨op' = op⟩ next-instr2
        by (auto intro: Sinca.step-op)
      moreover have ?MATCH ?s2'
        using ok-F1 rel-F1-F2 by (auto intro: match.intros)
        ultimately show ?thesis by auto
    next
      case (Some opinl)
        let ?F2' = Sinca.Fenv.map-entry F2 f (λfd. rewrite-fundef-body fd l pc
        (IOpInl opinl))
        let ?s2' = State ?F2' H (Frame f l (Suc pc) R (x # drop ar Σ) # st')
        have ?STEP ?s2'
          using Some IOp step-op ⟨op' = op⟩
          using Sinca.InlOp-correct Sinca.Inl-invertible next-instr2
          by (auto intro: Sinca.step-op-inl)
        moreover have ?MATCH ?s2'
          using ok-F1 step-op IOp ⟨op' = op⟩ Some
          by (auto intro!: match.intros intro: rel-fundefs-rewriteI2[OF rel-F1-F2]
            simp: Sinca.Inl-invertible)
        ultimately show ?thesis by auto
    qed
next
  case (IOpInl opinl)
  hence op = DeInl opinl using norm-eq-instr2 by simp

```

```

show ?thesis
proof (cases ?s2 opinl (take ar Σ))
  case True
    let ?s2' = State F2 H (Frame f l (Suc pc) R (x # drop ar Σ) # st')
    have ?STEP ?s2'
      using step-op IOpInl ⟨op = DeInl opinl⟩ True
      using next-instr2 Sinca.InlOp-correct
      by (auto intro: Sinca.step-op-inl-hit)
    moreover have ?MATCH ?s2'
      using ok-F1 rel-F1-F2 by (auto intro: match.intros)
    ultimately show ?thesis by auto
  next
    case False
    let ?F2' = Sinca.Fenv.map-entry F2 f (λfd. rewrite-fundef-body fd l pc (IOp
      (DeInl opinl)))
      let ?s2' = State ?F2' H (Frame f l (Suc pc) R (x # drop ar Σ) # st')
      have ?STEP ?s2'
        using step-op IOpInl ⟨op = DeInl opinl⟩ False
        using next-instr2 Sinca.InlOp-correct
        by (auto intro: Sinca.step-op-inl-miss)
      moreover have ?MATCH ?s2'
        using ok-F1 step-op ⟨op = DeInl opinl⟩
        by (auto intro!: match.intros intro: rel-fundefs-rewriteI2[OF rel-F1-F2])
      ultimately show ?thesis by auto
    qed
    qed simp-all
  next
    case (step-cjump f l pc lt lf d l' R Σ st')
    show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
    proof –
      let ?s1' = State F2 H (Frame f l' 0 R Σ # st')
      have ?STEP ?s1'
        using step-cjump rel-F1-F2-next-instr[of f l pc]
        by (auto intro: Sinca.step-cjump elim: norm-instr.elims simp: option-rel-Some1)
      moreover have ?MATCH ?s1'
        using ok-F1 rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by blast
    qed
  next
    case (step-call f lf pcf g gd1 Σf frameg Rf st')
    then obtain gd2 where
      F2-g: Finca-get F2 g = Some gd2 and
      rel-gd1-gd2: rel-fundef (=) norm-eq gd1 gd2
      using rel-fundefs-Some1[OF rel-F1-F2] by blast

    have same-fst-label-gd1-gd2: fst (hd (body gd1)) = fst (hd (body gd2))
      using rel-gd1-gd2
      using wf-fundefs-imp-wf-fundef[OF ok-F1 ⟨Fstd-get F1 g = Some gd1⟩,
      unfolded wf-fundef-def]

```

```

by (auto elim: fundef.rel-cases dest!: list-all2-rel-prod-fst-hd)

show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
proof -
  let ?Σg = take (arity gd2) Σf
  let ?s2' = State F2 H (frameg # Frame f lf pc f Rf Σf # st')
  have ?STEP ?s2'
    using step-call F2-g rel-gd1-gd2 rel-F1-F2-next-instr[of f lf pc f]
    using same-fst-label-gd1-gd2
    by (auto intro!: Sinca.step-call elim: norm-instr.elims
      simp: option-rel-Some1 rel-fundef-arithes rel-fundef-locals allocate-frame-def)
  moreover have ?MATCH ?s2'
    using ok-F1 rel-F1-F2
    by (auto intro: match.intros)
  ultimately show ?thesis by auto
qed
next
  case (step-return g lg pcg gd1 Σf Σg framef' f lf pc f Rf Rg st')
  then obtain gd2 where
    F2-g: Finca-get F2 g = Some gd2 and
    rel-gd1-gd2: rel-fundef (=) norm-eq gd1 gd2
    using rel-fundefs-Some1[OF rel-F1-F2] by blast
  obtain instr2 where
    next-instr2: next-instr (Finca-get F2) g lg pcg = Some instr2 and
    norm-eq-instr1-instr2: norm-eq Std.IReturn instr2
    using step-return rel-F1-F2-next-instr[of g lg pcg] by (auto simp: option-rel-Some1)
    thus ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
  proof (cases instr2)
    case IReturn
    let ?s1' = State F2 H (framef' # st')
    have ?STEP ?s1'
      using step-return next-instr2 IReturn
      using F2-g rel-gd1-gd2
      by (auto intro!: Sinca.step-return simp: rel-fundef-arithes rel-fundef-return)
    moreover have ?MATCH ?s1'
      using ok-F1 rel-F1-F2 by (auto intro: match.intros)
    ultimately show ?thesis by auto
  qed simp-all
qed
qed

lemma match-final-forward:
assumes match s1 s2 and final-s1: final Fstd-get Std.IReturn s1
shows final Finca-get Inca.IReturn s2
using ⟨match s1 s2⟩
proof (cases s1 s2 rule: match.cases)
  case (1 F1 F2 H st)
  show ?thesis
    using final-s1[unfolded 1]

```

```

proof (cases -- State F1 H st rule: final.cases)
  case (finalI f l pc R Σ)
    then show ?thesis
      using 1
      by (auto intro: final.intros elim: norm-instr.elims dest: rel-fundefs-next-instr1)
  qed
qed

sublocale std-inca-forward-simulation:
  forward-simulation where
    step1 = Sstd.step and final1 = final Fstd-get Std.IReturn and
    step2 = Sinca.step and final2 = final Finca-get Inca.IReturn and
    order = λ- -. False and match = λ-. match
  using match-final-forward forward-lockstep-simulation
  lockstep-to-plus-forward-simulation[of match Sstd.step - Sinca.step]
  by unfold-locales auto

```

41 Bisimulation

```

sublocale std-inca-bisimulation:
  bisimulation where
    step1 = Sstd.step and final1 = final Fstd-get Std.IReturn and
    step2 = Sinca.step and final2 = final Finca-get Inca.IReturn and
    orderf = λ- -. False and orderb = λ- -. False and match = λ-. match
    by unfold-locales
  end

end
theory Std-to-Inca-compiler
  imports Std-to-Inca-simulation
  VeriComp.Compiler
begin

```

41.1 Compilation of function definitions

```

fun compile-instr where
  compile-instr (Std.IPush d) = Inca.IPush d |
  compile-instr Std.IPop = Inca.IPop |
  compile-instr (Std.IGet n) = Inca.IGet n |
  compile-instr (Std.ISet n) = Inca.ISet n |
  compile-instr (Std.ILoad x) = Inca.ILoad x |
  compile-instr (Std.IStore x) = Inca.IStore x |
  compile-instr (Std.IOp op) = Inca.IOp op |
  compile-instr (Std.ICJump lt lf) = Inca.ICJump lt lf |
  compile-instr (Std.ICall f) = Inca.ICall f |
  compile-instr Std.IReturn = Inca.IReturn

fun compile-fundef where

```

```

compile-fundef (Fundef [] --) = None |  

compile-fundef (Fundef bblocks ar ret locals) =  

  Some (Fundef (map-ran (λ-. map compile-instr) bblocks) ar ret locals)

context std-inca-simulation begin

lemma lambda-eq-eq[simp]: ( $\lambda x y. y = x$ ) = (=)
  apply (intro ext)
  by blast

lemma norm-compile-instr:
  norm-instr (compile-instr instr) = instr
  by (cases instr) auto

lemma rel-compile-fundef:
  assumes compile-fundef fd1 = Some fd2
  shows rel-fundef (=) norm-eq fd1 fd2
  using assms
proof (cases rule: compile-fundef.elims)
  case (? x xs ar)
  then show ?thesis
  by (auto intro: list.rel-refl
    simp: const-eq-if-conv list.rel-map norm-compile-instr map-ran-def case-prod-beta
    rel-prodsel)
qed simp-all

lemma rel-fundef-imp-fundef-ok-iff:
  assumes rel-fundef (=) norm-eq fd1 fd2
  shows wf-fundef fd1  $\longleftrightarrow$  wf-fundef fd2
  unfolding wf-fundef-def
  using assms
  by (auto simp: list-all2-rel-prod-conv list.rel-eq elim: fundef.rel-cases)

lemma rel-fundefs-imp-wf-fundefs-iff:
  assumes rel-f-g: rel-fundefs f g
  shows wf-fundefs f  $\longleftrightarrow$  wf-fundefs g
proof (rule iffI)
  assume wf-fundefs f show wf-fundefs g
  proof (rule wf-fundefsI)
    fix x fd
    assume g x = Some fd thus wf-fundef fd
      using wf-fundefs f
      by (meson rel-f-g rel-fundef-imp-fundef-ok-iff rel-fundefs-Some2 wf-fundefs-imp-wf-fundef)
  qed
next
  show wf-fundefs g  $\Longrightarrow$  wf-fundefs f
    using rel-f-g
    by (meson rel-fundefs-imp-fundef-ok-iff rel-fundefs-Some1 wf-fundefs-imp-wf-fundef
      wf-fundefsI)

```

qed

```
lemma compile-fundef-wf:
  assumes compile-fundef fd = Some fd'
  shows wf-fundef fd'
  using assms
proof (cases rule: compile-fundef.elims)
  case (? x xs ar)
  then show ?thesis
    by (simp add: const-eq-if-conv wf-fundef-def map-ran-Cons-sel)
qed simp-all
```

41.2 Compilation of function environments

```
definition compile-env where
  compile-env e ≡
    Some Sinca.Fenv.from-list ∘ ap-map-list (ap-map-prod Some compile-fundef)
    (Fstd-to-list e)
```

```
lemma compile-env-imp-rel-option:
  assumes compile-env F1 = Some F2
  shows rel-option (λfd1 fd2. compile-fundef fd1 = Some fd2) (Fstd-get F1 f)
    (Finca-get F2 f)
proof –
  from assms(1) obtain xs where
    ap-map-list-F1: ap-map-list (ap-map-prod Some compile-fundef) (Fstd-to-list
    F1) = Some xs and
    F2-def: F2 = Sinca.Fenv.from-list xs
    by (auto simp add: compile-env-def ap-option-eq-Some-conv)
```

```
show ?thesis
  unfolding F2-def Sinca.Fenv.from-list-correct[symmetric]
  unfolding Sinca.Fenv.from-list-correct
  unfolding Sstd.Fenv.to-list-correct[symmetric]
  using ap-map-list-F1
  by (simp add: ap-map-list-ap-map-prod-imp-rel-option-map-of)
qed
```

```
lemma Finca-get-compile:
  assumes compile-F1: compile-env F1 = Some F2
  shows Finca-get F2 f = Fstd-get F1 f ≈ compile-fundef
  using compile-env-imp-rel-option[OF assms(1), of f]
  by (auto elim!: option.rel-cases)
```

```
lemma compile-env-rel-fundefs:
  assumes compile-env F1 = Some F2
  shows rel-fundefs (Fstd-get F1) (Finca-get F2)
  unfolding rel-fundefs-def
proof (rule allI)
```

```

fix f
show rel-option (rel-fundef (=) norm-eq) (Fstd-get F1 f) (Finca-get F2 f)
  using compile-env-imp-rel-option[OF assms(1), of f]
  by (auto elim!: option.rel-cases intro: rel-compile-fundef)
qed

lemma compile-env-imp-wf-fundefs2:
assumes compile-env F1 = Some F2
shows wf-fundefs (Finca-get F2)
unfolding Finca-get-compile[OF assms(1)]
by (auto simp: bind-eq-Some-conv intro!: wf-fundefsI intro: compile-fundef-wf)

```

41.3 Compilation of programs

```

fun compile where
compile (Prog F1 H f) = Some Prog ◊ compile-env F1 ◊ Some H ◊ Some f

theorem compile-load:
assumes
  compile-p1: compile p1 = Some p2 and
  load: Sinca.load p2 s2
shows ∃ s1. Sstd.load p1 s1 ∧ match s1 s2
proof (cases p1)
  case (Prog F1 H main)
  with assms(1) obtain F2 where
    compile-F1: compile-env F1 = Some F2 and
    p2-def: p2 = Prog F2 H main
    by (auto simp: ap-option-eq-Some-conv)

  show ?thesis
    using assms(2) unfolding p2-def Sinca.load-def
    proof (cases --- s2 rule: load.cases)
      case (1 fd2)
      from 1 obtain fd1 where
        F1-main: Fstd-get F1 main = Some fd1 and compile-fd1: compile-fundef fd1
        = Some fd2
        using Finca-get-compile[OF compile-F1] by (auto simp: bind-eq-Some-conv)

      note rel-fd1-fd2 = rel-compile-fundef[OF compile-fd1]
      hence first-label-fd1-fd2: fst (hd (body fd1)) = fst (hd (body fd2))
        using 1
        by (auto elim!: fundef.rel-cases dest: list-all2-rel-prod-fst-hd)

      have bodies-fd1-fd2: body fd1 = [] ↔ body fd2 = []
        using rel-fundef-body-length[OF rel-fd1-fd2]
        by auto

      let ?s1 = State F1 H [allocate-frame main fd1 [] uninitialized]

```

```

show ?thesis
proof (intro exI conjI)
  show Sstd.load p1 ?s1
    unfolding Prog
    using 1 F1-main rel-fd1-fd2 bodies-fd1-fd2
    by (auto simp: rel-fundef-arities Sstd.load-def intro!: load.intros)
next
  note rel-F1-F2 = compile-env-rel-fundefs[OF compile-F1]
  moreover have wf-fundefs (Fstd-get F1)
    proof (rule wf-fundefsI)
      fix f fd1
      assume Fstd-get F1 f = Some fd1
      then show wf-fundef fd1
        by (metis (mono-tags, lifting) Finca-get-compile bind.bind-lunit compile-F1
            rel-F1-F2 compile-fundef-wf rel-fundef-imp-fundef-ok-iff
            rel-fundefs-Some1)
    qed
    ultimately show match ?s1 s2
      unfolding 1
      using rel-fd1-fd2
      by (auto simp: allocate-frame-def first-label-fd1-fd2 rel-fundef-locales
                     intro: match.intros)
    qed
  qed
qed

```

sublocale std-to-inca-compiler:

compiler where

step1 = Sstd.step and final1 = final Fstd-get Std.IReturn and load1 = Sstd.load and

step2 = Sinca.step and final2 = final Finca-get Inca.IReturn and load2 = Sinca.load and

order = λ- -. False and match = λ-. match and compile = compile

using compile-load

by unfold-locales auto

41.4 Completeness of compilation

```

lemma compile-fundef-complete:
  assumes wf-fundef fd1
  shows ∃fd2. compile-fundef fd1 = Some fd2
proof (cases fd1)
  case (Fundef bbs ar)
  then obtain bb bbs' where bbs-def: bbs = bb # bbs'
    using assms(1) by (cases bbs; auto)
  show ?thesis
    using assms
    unfolding Fundef bbs-def
    by simp

```

qed

lemma *compile-env-complete*:

assumes *wf-F1*: *wf-fundefs* (*Fstd-get F1*)
 shows $\exists F2.$ *compile-env F1 = Some F2*

proof –

show *?thesis*

using *wf-F1*

by (*auto simp: compile-env-def ap-option-eq-Some-conv ap-map-prod-eq-Some-conv intro!: ex-ap-map-list-eq-SomeI Sstd.Fenv.to-list-list-allI compile-fundef-complete intro: wf-fundefs-imp-wf-fundef*)

qed

theorem *compile-complete*:

assumes *wf-p1*: *wf-prog Fstd-get p1*
 shows $\exists p2.$ *compile p1 = Some p2*

proof (*cases p1*)

case (*Prog x1 x2 x3*)

then show *?thesis*

using *wf-p1 unfolding wf-prog-def*

by (*auto dest: compile-env-complete*)

qed

theorem *compile-load-forward*:

assumes

wf-p1: wf-prog Fstd-get p1 and load-p1: Sstd.load p1 s1

shows $\exists p2 s2.$ *compile p1 = Some p2 \wedge Sinca.load p2 s2 \wedge match s1 s2*

proof –

from *load-p1 obtain F1 H main fd1 where*

p1-def: p1 = Prog F1 H main and

s1-def: s1 = Global.state.State F1 H [allocate-frame main fd1 [] uninitialized]

and

F1-main: Fstd-get F1 main = Some fd1 and

arity-fd1: arity fd1 = 0

by (*auto simp: Sstd.load-def elim: load.cases*)

obtain *F2 where compile-F1: compile-env F1 = Some F2*

using *wf-p1[unfolded p1-def wf-prog-def, simplified]*

using *compile-env-complete[of F1]*

by *auto*

obtain *fd2 where*

F2-main: Finca-get F2 main = Some fd2 and compile-fd1: compile-fundef fd1

= Some fd2

using *compile-env-imp-rel-option[OF compile-F1, of main]*

unfolding *F1-main option-rel-Some1*

by *auto*

hence *rel-fd1-fd2: rel-fundef (=) norm-eq fd1 fd2*

```

using rel-compile-fundef by auto

note wf-fd2 = compile-fundef-wf[OF compile-fd1]

let ?s2 = State F2 H [allocate-frame main fd2 [] uninitialized]

show ?thesis
proof (intro exI conjI)
  show compile p1 = Some (Prog F2 H main)
    by (simp add: p1-def compile-F1)
next
  show Sinca.load (Prog F2 H main) ?s2
    using F2-main arity-fd1
    using rel-fundef-arities[OF rel-fd1-fd2]
    using compile-fundef-wf[OF compile-fd1]
    by (auto simp: wf-fundef-def Sinca.load-def intro: load.intros)
next
  have fst (hd (body fd1)) = fst (hd (body fd2))
    using rel-fd1-fd2 wf-fd2
    by (auto simp add: fundef.rel-sel wf-fundef-def list-all2-rel-prod-fst-hd)
  moreover have wf-fundefs (Fstd-get F1)
    using compile-env-imp-wf-fundefs2[OF compile-F1]
  using compile-env-rel-fundefs[OF compile-F1, THEN rel-fundefs-imp-wf-fundefs-iff]
    by simp
  ultimately show match s1 ?s2
    unfolding s1-def
    using rel-fd1-fd2 compile-env-rel-fundefs[OF compile-F1]
    by (auto simp: allocate-frame-def rel-fundef-locals intro!: match.intros)
  qed
qed

end

end

```

References

- [1] M. Desharnais and S. Brunthaler. Towards efficient and verified virtual machines for dynamic languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021. Association for Computing Machinery, 2021.