Interpreter_Optimizations

Martin Desharnais

February 23, 2021

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

theory *Env*
imports *Main*
begin

1 Environment

locale *env* =
fixes
empty :: *'env* **and**
get :: *'env* ⇒ *'key* ⇒ *'val option* **and**
add :: *'env* ⇒ *'key* ⇒ *'val* ⇒ *'env* **and**
to-list :: *'env* ⇒ (*'key* × *'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* ≠ *y* ⟹ *get* (*add e x v*) *y* = *get e y* **and**
to-list-correct: *map-of* (*to-list e*) = *get e*

begin

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

definition *singleton* **where**
singleton ≡ *add empty*

fun *add-list* :: *'env* ⇒ (*'key* × *'val*) *list* ⇒ *'env* **where**
add-list e [] = *e* |
add-list e (*x* # *xs*) = *add* (*add-list e xs*) (*fst x*) (*snd x*)

definition *from-list* :: (*'key* × *'val*) *list* ⇒ *'env* **where**
from-list ≡ *add-list empty*

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

using *get-add-eq get-add-neq* **by** (*simp add*: *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*[*symmetric*]
unfolding *to-list-correct*[*symmetric*]
by *simp*
qed

end

end
theory *Env-list*
imports *Env*
begin

1.1 List-based implementation of environment

context
begin

qualified
datatype ('*key*, '*val*) *t* = *T* ('*key* × '*val*) *list*

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

qualified fun *get* :: ('*key*, '*val*) *t* ⇒ '*key* ⇒ '*val option* **where**
get (*T xs*) = *Map.map-of xs*

qualified fun *add* :: ('*key*, '*val*) *t* ⇒ '*key* ⇒ '*val* ⇒ ('*key*, '*val*) *t* **where**
add (*T xs*) *k v* = *T* ((*k*, *v*) # *xs*)

qualified fun *to-list* :: ('*key*, '*val*) *t* ⇒ ('*key* × '*val*) *list* **where**
to-list (*T xs*) = *xs*

lemma *get-empty*: *get empty x* = *None*
by (*simp add*: *empty-def*)

lemma *get-add-eq*: *get* (*add e x v*) *x* = *Some v*
by (*cases e*; *simp*)

lemma *get-add-neq*: *x* ≠ *y* ⟹ *get* (*add e x v*) *y* = *get e y*

```
  by (cases e; simp)

lemma to-list-correct: map-of (to-list e) = get e
proof (rule ext)
  fix k
  obtain xs where e = T xs
    by (cases e; simp)
  show map-of (to-list e) k = get e k
    unfolding ‹e = T xs›
    by (induction xs) simp-all
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)


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 ⟹ ∃ y ys'. ys = y # ys'
  same-length xs (y # ys) ⟹ ∃ x xs'. xs = x # xs'
proof −
  assume same-length (x # xs) ys
  then show ∃ y ys'. ys = y # ys'
    by (induction x # xs ys rule: same-length.induct) simp
next
  assume same-length xs (y # ys)
  then show ∃ x xs'. xs = x # xs'
    by (induction xs y # ys rule: same-length.induct) simp
qed

inductive for-all2 for r where
  for-all2-Nil: for-all2 r [] [] |
  for-all2-Cons: r x y ⟹ for-all2 r xs ys ⟹ for-all2 r (x # xs) (y # ys)

code-pred for-all2 .

declare for-all2-Nil[intro]
declare for-all2-Cons[intro]

lemma for-all2-refl: (∀ x. r x x) ⟹ for-all2 r xs xs
  by (induction xs) auto

lemma for-all2-same-length: for-all2 r xs ys ⟹ same-length xs ys
  by (induction rule: for-all2.induct) (auto intro: same-length.intros)

lemma for-all2-ConsD: for-all2 r (x # xs) (y # ys) ⟹ r x y ∧ for-all2 r xs ys
  using for-all2.cases by blast
```

2 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*]

3 Generic lemmas

lemma *map-list-update-id*:
f (*xs* ! *pc*) = *f instr* ⟹ *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* (λ*x*. *x* = *y*) *xs*
shows *xs* = *replicate* (*length xs*) *y*
using *assms*
by (*induction xs*; *simp*)

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

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

lemma *replicate-eq-map*:
assumes *list-all g* (*take n xs*) **and** *n* ≤ *length xs* **and** ∀ *y*. *g y* ⟶ *f y* = *x*
shows *replicate n x* = *map f* (*take n 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

end
theory *Option-applicative*
imports *Main*
begin

context begin

local-setup ‹
Local-Theory.map-background-naming (*Name-Space.add-path Option*)
›

fun *pure* **where**

```
  pure x = Some x

fun ap :: ('a ⇒ 'b) option ⇒ 'a option ⇒ 'b option (infixl <*> 51) where
  Some f <*> Some x = Some (f x) |
  - <*> - = None

end

lemma identity: pure id <*> x = x
  by (cases x) simp-all

lemma homomorphism: pure f <*> pure x = pure (f x)
  by simp

lemma interchange: f <*> pure x = pure (λg. g x) <*> f
  by (cases f) simp-all

lemma composition: pure (∘) <*> u <*> v <*> w = u <*> (v <*> w)
  apply (cases u, simp)
  apply (cases v, simp)
  apply (cases w, simp)
  by simp

end
theory Result
  imports
    Main
    HOL−Library.Monad-Syntax
begin

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

4 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*]: $x \gg\!= Ok = x$
by (*cases x*) *simp-all*

qualified lemma *bind-eq-Ok-conv*: $(x \gg\!= f = Ok\ z) = (\exists y.\ x = Ok\ y \wedge f\ y = Ok\ z)$
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*:
$x \gg\!= f = Error\ z \longleftrightarrow x = Error\ z \vee (\exists y.\ x = Ok\ y \wedge f\ y = Error\ z)$
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*]:
$(if\ a\ then\ b\ else\ Error\ c) = Ok\ d \longleftrightarrow a \wedge b = Ok\ d$
$(if\ a\ then\ Error\ c\ else\ b) = Ok\ d \longleftrightarrow \neg\ a \wedge b = Ok\ d$
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

5 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* ⇒ ('*err*, '*ok list*) *result* **where**
those [] = *Ok* [] |
those (*x* # *xs*) = *do* {
y ← *x*;
ys ← *those xs*;
Ok (*y* # *ys*)
}

qualified lemma *those-Cons-OkD*: *those* (*x* # *xs*) = *Ok ys* ⟹ ∃ *y ys*'. *ys* = *y* # *ys*' ∧ *x* = *Ok y* ∧ *those xs* = *Ok ys*'
by *auto*

end

end
theory *Global*
imports *Main Result*
begin

sledgehammer-params [*timeout* = *30*]
sledgehammer-params [*provers* = *cvc4 e spass vampire z3*]

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*

fun *traverse* :: ('*a* ⇒ '*b option*) ⇒ '*a list* ⇒ '*b list option* **where**

```
  traverse f [] = Some [] |
  traverse f (x # xs) = do {
    x' ← f x;
    xs' ← traverse f xs;
    Some (x' # xs')
  }

lemma traverse-length: traverse f xs = Some ys ⟹ length ys = length xs
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then show ?case
    by (auto simp add: Option.bind-eq-Some-conv)
qed

datatype 'instr fundef =
  Fundef (body: 'instr list) (arity: nat)

lemma rel-fundef-arities: rel-fundef r gd1 gd2 ⟹ arity gd1 = arity gd2
  by (simp add: fundef.rel-sel)

lemma rel-fundef-body-length[simp]:
  rel-fundef r fd1 fd2 ⟹ length (body fd1) = length (body fd2)
  by (auto intro: list-all2-lengthD simp add: fundef.rel-sel)

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

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

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

definition rewrite :: 'instr list ⇒ nat ⇒ 'instr ⇒ 'instr list where
  rewrite p pc i = list-update p pc i

fun rewrite-fundef-body :: 'instr fundef ⇒ nat ⇒ 'instr ⇒ 'instr fundef where
  rewrite-fundef-body (Fundef xs ar) n x = Fundef (rewrite xs n x) ar

lemmas length-rewrite[simp] = length-list-update[folded rewrite-def]
lemmas nth-rewrite-eq[simp] = nth-list-update-eq[folded rewrite-def]
lemmas nth-rewrite-neq[simp] = nth-list-update-neq[folded rewrite-def]
lemmas take-rewrite[simp] = take-update-cancel[folded rewrite-def]
lemmas take-rewrite-swap = take-update-swap[folded rewrite-def]
lemmas map-rewrite = map-update[folded rewrite-def]
lemmas list-all2-rewrite-cong[intro] = list-all2-update-cong[folded rewrite-def]
```

lemma *body-rewrite-fundef-body*[*simp*]: *body* (*rewrite-fundef-body fd n x*) = *rewrite* (*body fd*) *n x*
by (*cases fd*) *simp*

lemma *arity-rewrite-fundef-body*[*simp*]: *arity* (*rewrite-fundef-body fd n x*) = *arity fd*
by (*cases fd*) *simp*

lemma *if-eq-const-conv*: (*if x then y else z*) = *w* ⟷ *x* ∧ *y* = *w* ∨ ¬*x* ∧ *z* = *w*
by *simp*

end
theory *Op*
imports *Main*
begin

6 n-ary operations

locale *nary-operations* =
fixes
𝔒𝔭 :: *'op* ⇒ *'a list* ⇒ *'a* **and**
𝔄𝔯𝔦𝔱𝔶 :: *'op* ⇒ *nat*
assumes
𝔒𝔭-𝔄𝔯𝔦𝔱𝔶-*domain*: *length xs* = 𝔄𝔯𝔦𝔱𝔶 *op* ⟹ ∃ *y*. 𝔒𝔭 *op xs* = *y*

end
theory *OpInl*
imports *Op*
begin

7 n-ary operations

locale *nary-operations-inl* =
nary-operations 𝔒𝔭 𝔄𝔯𝔦𝔱𝔶
for
𝔒𝔭 :: *'op* ⇒ *'a list* ⇒ *'a* **and** 𝔄𝔯𝔦𝔱𝔶 +
fixes
𝔍𝔫𝔩𝔒𝔭 :: *'opinl* ⇒ *'a list* ⇒ *'a* **and**
𝔍𝔫𝔩 :: *'op* ⇒ *'a list* ⇒ *'opinl option* **and**
𝔍𝔰𝔍𝔫𝔩 :: *'opinl* ⇒ *'a list* ⇒ *bool* **and**
𝔇𝔢𝔍𝔫𝔩 :: *'opinl* ⇒ *'op*
assumes
𝔍𝔫𝔩-*invertible*: 𝔍𝔫𝔩 *op xs* = *Some opinl* ⟹ 𝔇𝔢𝔍𝔫𝔩 *opinl* = *op* **and**
𝔍𝔫𝔩𝔒𝔭-*correct*: *length xs* = 𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 *opinl*) ⟹ 𝔍𝔫𝔩𝔒𝔭 *opinl xs* = 𝔒𝔭 (𝔇𝔢𝔍𝔫𝔩 *opinl*) *xs* **and**
𝔍𝔫𝔩-𝔍𝔰𝔍𝔫𝔩: 𝔍𝔫𝔩 *op xs* = *Some opinl* ⟹ 𝔍𝔰𝔍𝔫𝔩 *opinl xs*

begin

lemma $\mathfrak{Inl}$-*inj-on*: *inj-on* $\mathfrak{Inl}$ { *op* | *op args*. $\mathfrak{Inl}$ *op args* ≠ *None* }
apply (*simp add*: *inj-on-def*)
proof (*intro allI impI*, *elim exE*)
fix op_1 op_2 $args_1$ $args_2$ $opinl_1$ $opinl_2$
assume *assms*: $\mathfrak{Inl}$ op_1 = $\mathfrak{Inl}$ op_2 $\mathfrak{Inl}$ op_1 $args_1$ = *Some* $opinl_1$ $\mathfrak{Inl}$ op_2 $args_2$ = *Some* $opinl_2$
hence $\mathfrak{DeInl}$ $opinl_1$ = op_1 $\mathfrak{DeInl}$ $opinl_1$ = op_2
by (*auto intro*: $\mathfrak{Inl}$-*invertible*)
thus op_1 = op_2
by *simp*
qed

abbreviation $\mathfrak{Inl}$-*dom* **where**
$\mathfrak{Inl}$-*dom* ≡ {*op* | *op args*. $\mathfrak{Inl}$ *op args* ≠ *None* }

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

end

end
theory *Dynamic*
imports *Main*
begin

locale *dynval* =
fixes
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

8 Inline caching

8.1 Static representation

datatype (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) *instr* =
IPush 'dyn |
IPop |
ILoad 'var |
IStore 'var |
IOp 'op |
IOpInl 'opinl |
ICJump nat |
ICall 'fun

8.2 Dynamic representation

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 is-true is-false +
nary-operations-inl 𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩
for
F-empty **and**
F-get :: *'fenv* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) *instr fundef option* **and**
F-add **and** *F-to-list* **and**
heap-empty **and**
heap-get :: *'henv* ⇒ *'var* × *'dyn* ⇒ *'dyn option* **and**
heap-add **and** *heap-to-list* **and**
is-true :: *'dyn* ⇒ *bool* **and** *is-false* **and**
𝔒𝔭 :: *'op* ⇒ *'dyn list* ⇒ *'dyn* **and** 𝔄𝔯𝔦𝔱𝔶 **and**
𝔍𝔫𝔩𝔒𝔭 **and** 𝔍𝔫𝔩 **and** 𝔍𝔰𝔍𝔫𝔩 **and** 𝔇𝔢𝔍𝔫𝔩 :: *'opinl* ⇒ *'op*
begin

inductive *final* :: (*'fenv*, *'henv*, (*'fun*, *'dyn*) *frame*) *state* ⇒ *bool* **where**
F-get F f = *Some fd* ⟹ *pc* = *length* (*body fd*) ⟹ *final* (*State F H* [*Frame f pc* Σ])

8.3 Semantics

inductive *step* ::
(*'fenv*, *'henv*, (*'fun*, *'dyn*) *frame*) *state* ⇒ (*'fenv*, *'henv*, (*'fun*, *'dyn*) *frame*) *state* ⇒ *bool* (**infix** → *55*) **where**

step-push:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IPush d* $\Longrightarrow$
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*d* # Σ) # *st*) |

step-pop:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IPop* $\Longrightarrow$
State F H (*Frame f pc* (*d* # Σ) # *st*) → *State F H* (*Frame f* (*Suc pc*) Σ # *st*) |

step-load:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *ILoad x* $\Longrightarrow$
heap-get H (*x*, *y*) = *Some d* $\Longrightarrow$
State F H (*Frame f pc* (*y* # Σ) # *st*) → *State F H* (*Frame f* (*Suc pc*) (*d* # Σ) # *st*) |

step-store:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IStore x* $\Longrightarrow$
heap-add H (*x*, *y*) *d* = *H'* $\Longrightarrow$
State F H (*Frame f pc* (*y* # *d* # Σ) # *st*) → *State F H'* (*Frame f* (*Suc pc*) Σ # *st*) |

step-op:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IOp op* $\Longrightarrow$
$\mathfrak{Arity}$ *op* = *ar* $\Longrightarrow$ *ar* ≤ *length* Σ $\Longrightarrow$ $\mathfrak{Inl}$ *op* (*take ar* Σ) = *None* $\Longrightarrow$
$\mathfrak{Op}$ *op* (*take ar* Σ) = *x* $\Longrightarrow$
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*x* # *drop ar* Σ) # *st*) |

step-op-inl:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IOp op* $\Longrightarrow$
$\mathfrak{Arity}$ *op* = *ar* $\Longrightarrow$ *ar* ≤ *length* Σ $\Longrightarrow$ $\mathfrak{Inl}$ *op* (*take ar* Σ) = *Some opinl* $\Longrightarrow$
$\mathfrak{InlOp}$ *opinl* (*take ar* Σ) = *x* $\Longrightarrow$
F-add F f (*rewrite-fundef-body fd pc* (*IOpInl opinl*)) = *F'* $\Longrightarrow$
State F H (*Frame f pc* Σ # *st*) → *State F' H* (*Frame f* (*Suc pc*) (*x* # *drop ar* Σ) # *st*) |

step-op-inl-hit:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IOpInl opinl* $\Longrightarrow$
$\mathfrak{Arity}$ ($\mathfrak{DeInl}$ *opinl*) = *ar* $\Longrightarrow$ *ar* ≤ *length* Σ $\Longrightarrow$ $\mathfrak{IsInl}$ *opinl* (*take ar* Σ) $\Longrightarrow$
$\mathfrak{InlOp}$ *opinl* (*take ar* Σ) = *x* $\Longrightarrow$
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*x* # *drop ar* Σ) # *st*) |

step-op-inl-miss:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IOpInl opinl* $\Longrightarrow$
$\mathfrak{Arity}$ ($\mathfrak{DeInl}$ *opinl*) = *ar* $\Longrightarrow$ *ar* ≤ *length* Σ $\Longrightarrow$ ¬ $\mathfrak{IsInl}$ *opinl* (*take ar* Σ) $\Longrightarrow$
$\mathfrak{InlOp}$ *opinl* (*take ar* Σ) = *x* $\Longrightarrow$
F-add F f (*rewrite-fundef-body fd pc* (*IOp* ($\mathfrak{DeInl}$ *opinl*))) = *F'* $\Longrightarrow$

State F H (Frame f pc Σ # st) → State F′ H (Frame f (Suc pc) (x # drop ar Σ) # st) |

step-cjump-true:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = ICJump n ⟹
is-true d ⟹
State F H (Frame f pc (d # Σ) # st) → State F H (Frame f n Σ # st) |

step-cjump-false:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = ICJump n ⟹
is-false d ⟹
State F H (Frame f pc (d # Σ) # st) → State F H (Frame f (Suc pc) Σ # st) |

step-fun-call:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = ICall g ⟹
F-get F g = Some gd ⟹ arity gd ≤ length Σ ⟹
frame$_f$ = Frame f pc Σ ⟹ frame$_g$ = Frame g 0 (take (arity gd) Σ) ⟹
State F H (frame$_f$ # st) → State F H (frame$_g$ # frame$_f$ # st) |

step-fun-end:
F-get F g = Some gd ⟹ arity gd ≤ length Σ$_f$ ⟹ pc$_g$ = length (body gd) ⟹
frame$_g$ = Frame g pc$_g$ Σ$_g$ ⟹ frame$_f$ = Frame f pc$_f$ Σ$_f$ ⟹
frame$_f$′ = Frame f (Suc pc$_f$) (Σ$_g$ @ drop (arity gd) Σ$_f$) ⟹
State F H (frame$_g$ # frame$_f$ # st) → State F H (frame$_f$′ # st)

lemma *step-deterministic*:
s1 → s2 ⟹ s1 → s3 ⟹ s2 = s3
by (*induction rule*: *step.cases*;
auto elim!: *step.cases dest*: *is-true-and-is-false-implies-False*)

lemma *final-finished*: *final s ⟹ finished step s*
unfolding *finished-def*
proof
assume *final s* **and** ∃ *x*. *step s x*
then obtain *s′* **where** *step s s′*
by *auto*
thus *False*
using ‹*final s*›
by (*auto elim*!: *step.cases final.cases*)
qed

sublocale *inca-sem*: *semantics step final*
using *final-finished step-deterministic*
by *unfold-locales*

inductive *load* :: (*'fenv*, *'a*, *'fun*) *prog* ⇒ (*'fenv*, *'a*, (*'fun*, *'b*) *frame*) *state* ⇒ *bool*
where

```
  F-get F main = Some fd ⟹ arity fd = 0 ⟹
  load (Prog F H main) (State F H [Frame main 0 []])

sublocale inca-lang: language step final 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 is-true is-false
  for is-true :: 'dyn ⇒ bool 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 ⇒ 'dyn ⇒ ('dyn, 'ubx1, 'ubx2) unboxed option  where
```

unbox Ubx1 = map-option OpUbx1 ∘ unbox-ubx1 |
unbox Ubx2 = map-option OpUbx2 ∘ unbox-ubx2

fun *cast-and-box* **where**
cast-and-box Ubx1 = map-option box-ubx1 ∘ cast-Ubx1 |
cast-and-box Ubx2 = map-option box-ubx2 ∘ 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 pc Σ) = Frame g pc (if f = g then map box-operand Σ else Σ)

definition *box-stack* **where**
box-stack f ≡ map (box-frame f)

end

end
theory *OpUbx*
imports *OpInl Unboxed*
begin

9 n-ary operations

locale *nary-operations-ubx* =
nary-operations-inl 𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩 +
unboxedval is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
for
𝔒𝔭 :: *'op ⇒ 'dyn list ⇒ 'dyn* **and** 𝔄𝔯𝔦𝔱𝔶 **and**
𝔍𝔫𝔩𝔒𝔭 **and** 𝔍𝔫𝔩 **and** 𝔍𝔰𝔍𝔫𝔩 **and** 𝔇𝔢𝔍𝔫𝔩 :: *'opinl ⇒ 'op* **and**
is-true :: *'dyn ⇒ bool* **and** *is-false* **and**
box-ubx1 :: *'ubx1 ⇒ 'dyn* **and** *unbox-ubx1* **and**
box-ubx2 :: *'ubx2 ⇒ 'dyn* **and** *unbox-ubx2* +
fixes
𝔘𝔟𝔵𝔒𝔭 :: *'opubx ⇒ ('dyn, 'ubx1, 'ubx2) unboxed list ⇒ ('dyn, 'ubx1, 'ubx2) unboxed option* **and**
𝔘𝔟𝔵 :: *'opinl ⇒ type option list ⇒ 'opubx option* **and**
𝔅𝔬𝔵 :: *'opubx ⇒ 'opinl* **and**
𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 :: *'opubx ⇒ type option list × type option*
assumes

𝔘𝔟𝔵-*invertible*:
𝔘𝔟𝔵 *opinl ts* = *Some opubx* ⟹ 𝔅𝔬𝔵 *opubx* = *opinl* **and**
𝔘𝔟𝔵𝔒𝔭-*correct*:
𝔘𝔟𝔵𝔒𝔭 *opubx* Σ = *Some z* ⟹ 𝔍𝔫𝔩𝔒𝔭 (𝔅𝔬𝔵 *opubx*) (*map norm-unboxed* Σ) = *norm-unboxed z* **and**
𝔘𝔟𝔵𝔒𝔭-*to*-𝔍𝔫𝔩:
𝔘𝔟𝔵𝔒𝔭 *opubx* Σ = *Some z* ⟹ 𝔍𝔫𝔩 (𝔇𝔢𝔍𝔫𝔩 (𝔅𝔬𝔵 *opubx*)) (*map norm-unboxed* Σ) = *Some* (𝔅𝔬𝔵 *opubx*) **and**

𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-𝔄𝔯𝔦𝔱𝔶:
𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 (𝔅𝔬𝔵 *opubx*)) = *length* (*fst* (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 *opubx*)) **and**
𝔘𝔟𝔵-*opubx-type*:
𝔘𝔟𝔵 *opinl ts* = *Some opubx* ⟹ *fst* (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 *opubx*) = *ts* **and**

𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-*correct*:
𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 *opubx* = (*map typeof xs*, τ) ⟹
∃ *y*. 𝔘𝔟𝔵𝔒𝔭 *opubx xs* = *Some y* ∧ *typeof y* = τ **and**
𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-*complete*:
𝔘𝔟𝔵𝔒𝔭 *opubx xs* = *Some y* ⟹ 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 *opubx* = (*map typeof xs*, *typeof y*)

begin

end

end
theory *Ubx*
imports *Global OpUbx Env*
VeriComp.Language
begin

10 Unboxed caching

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

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 𝔄𝔯𝔦𝔱𝔶
𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩

```
    is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
    𝔘𝔟𝔵𝔒𝔭 𝔘𝔟𝔵 𝔅𝔬𝔵 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭
  for
    — Functions environment
    F-empty and
    F-get :: 'fenv ⇒ 'fun ⇒ ('dyn, 'var, 'fun, 'op, 'opinl, 'opubx, 'num, 'bool) 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

    — Unboxed values
    is-true :: 'dyn ⇒ bool and is-false and
    box-ubx1 and unbox-ubx1 and
    box-ubx2 and unbox-ubx2 and

    — n-ary operations
    𝔒𝔭 :: 'op ⇒ 'dyn list ⇒ 'dyn and 𝔄𝔯𝔦𝔱𝔶 and
    𝔍𝔫𝔩𝔒𝔭 and 𝔍𝔫𝔩 and 𝔍𝔰𝔍𝔫𝔩 and 𝔇𝔢𝔍𝔫𝔩 :: 'opinl ⇒ 'op and
   𝔘𝔟𝔵𝔒𝔭 :: 'opubx ⇒ ('dyn, 'num, 'bool) unboxed list ⇒ ('dyn, 'num, 'bool) unboxed
option and
    𝔘𝔟𝔵 :: 'opinl ⇒ type option list ⇒ 'opubx option and
    𝔅𝔬𝔵 :: 'opubx ⇒ 'opinl and
    𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭
begin

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

fun generalize-fundef where
  generalize-fundef (Fundef xs ar) = Fundef (map generalize-instr xs) ar

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 ∘ generalize-instr = generalize-instr
  by fastforce

lemma generalize-fundef-length[simp]: length (body (generalize-fundef fd)) = length
```

(*body fd*)
by (*cases fd*) *simp*

lemma *body-generalize-fundef*[*simp*]: *body* (*generalize-fundef fd*) = *map generalize-instr* (*body fd*)
by (*cases fd*) *simp*

lemma *arity-generalize-fundef*[*simp*]: *arity* (*generalize-fundef fd2*) = *arity fd2*
by (*cases fd2*) *simp*

inductive *final* **where**
F-get F f = *Some fd* ⟹ *pc* = *length* (*body fd*) ⟹ *final* (*State F H* [*Frame f pc* Σ])

10.1 Semantics

inductive *step* (**infix** → *55*) **where**
step-push:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IPush d* ⟹
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*OpDyn d* # Σ) # *st*) |

step-push-ubx1:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IPushUbx1 n* ⟹
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*OpUbx1 n* # Σ) # *st*) |

step-push-ubx2:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IPushUbx2 b* ⟹
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*OpUbx2 b* # Σ) # *st*) |

step-pop:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IPop* ⟹
State F H (*Frame f pc* (*x* # Σ) # *st*) → *State F H* (*Frame f* (*Suc pc*) Σ # *st*) |

step-load:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *ILoad x* ⟹
cast-Dyn i = *Some i′* ⟹ *heap-get H* (*x*, *i′*) = *Some d* ⟹
State F H (*Frame f pc* (*i* # Σ) # *st*) → *State F H* (*Frame f* (*Suc pc*) (*OpDyn d* # Σ) # *st*) |

step-load-ubx-hit:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *ILoadUbx* τ *x* ⟹
cast-Dyn i = *Some i′* ⟹ *heap-get H* (*x*, *i′*) = *Some d* ⟹ *unbox* τ *d* = *Some blob* ⟹

State F H (Frame f pc (i # Σ) # st) → State F H (Frame f (Suc pc) (blob # Σ) # st) |

step-load-ubx-miss:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = ILoadUbx τ x ⟹
cast-Dyn i = Some i' ⟹ heap-get H (x, i') = Some d ⟹ unbox τ d = None ⟹
F-add F f (generalize-fundef fd) = F' ⟹
State F H (Frame f pc (i # Σ) # st) → State F' H (box-stack f (Frame f (Suc pc) (OpDyn d # Σ) # st)) |

step-store:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = IStore x ⟹
cast-Dyn i = Some i' ⟹ cast-Dyn y = Some d ⟹ heap-add H (x, i') d = H' ⟹
State F H (Frame f pc (i # y # Σ) # st) → State F H' (Frame f (Suc pc) Σ # st) |

step-store-ubx:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = IStoreUbx τ x ⟹
cast-Dyn i = Some i' ⟹ cast-and-box τ blob = Some d ⟹ heap-add H (x, i') d = H' ⟹
State F H (Frame f pc (i # blob # Σ) # st) → State F H' (Frame f (Suc pc) Σ # st) |

step-op:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = IOp op ⟹
𝔄rity op = ar ⟹ ar ≤ length Σ ⟹
traverse cast-Dyn (take ar Σ) = Some Σ' ⟹
𝔍nl op Σ' = None ⟹ 𝔒p op Σ' = x ⟹
State F H (Frame f pc Σ # st) → State F H (Frame f (Suc pc) (OpDyn x # drop ar Σ) # st) |

step-op-inl:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = IOp op ⟹
𝔄rity op = ar ⟹ ar ≤ length Σ ⟹
traverse cast-Dyn (take ar Σ) = Some Σ' ⟹
𝔍nl op Σ' = Some opinl ⟹ 𝔍nl𝔒p opinl Σ' = x ⟹
F-add F f (rewrite-fundef-body fd pc (IOpInl opinl)) = F' ⟹
State F H (Frame f pc Σ # st) → State F' H (Frame f (Suc pc) (OpDyn x # drop ar Σ) # st) |

step-op-inl-hit:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = IOpInl opinl ⟹
𝔄rity (𝔇e𝔍nl opinl) = ar ⟹ ar ≤ length Σ ⟹
traverse cast-Dyn (take ar Σ) = Some Σ' ⟹

$\mathfrak{IsInl}$ *opinl* $\Sigma' \Longrightarrow$ $\mathfrak{InlOp}$ *opinl* $\Sigma' = x \Longrightarrow$
State F H (*Frame f pc* Σ # *st*) $\rightarrow$ *State F H* (*Frame f* (*Suc pc*) (*OpDyn x* # *drop ar* Σ) # *st*) |

step-op-inl-miss:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IOpInl opinl* $\Longrightarrow$
$\mathfrak{Arity}$ ($\mathfrak{DeInl}$ *opinl*) = *ar* $\Longrightarrow$ *ar* $\leq$ *length* $\Sigma \Longrightarrow$
traverse cast-Dyn (*take ar* Σ) = *Some* $\Sigma' \Longrightarrow$
$\neg$ $\mathfrak{IsInl}$ *opinl* $\Sigma' \Longrightarrow$ $\mathfrak{InlOp}$ *opinl* $\Sigma' = x \Longrightarrow$
F-add F f (*rewrite-fundef-body fd pc* (*IOp* ($\mathfrak{DeInl}$ *opinl*))) = *F'* $\Longrightarrow$
State F H (*Frame f pc* Σ # *st*) $\rightarrow$ *State F' H* (*Frame f* (*Suc pc*) (*OpDyn x* # *drop ar* Σ) # *st*) |

step-op-ubx:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *IOpUbx opubx* $\Longrightarrow$
$\mathfrak{DeInl}$ ($\mathfrak{Box}$ *opubx*) = *op* $\Longrightarrow$ $\mathfrak{Arity}$ *op* = *ar* $\Longrightarrow$ *ar* $\leq$ *length* $\Sigma \Longrightarrow$
$\mathfrak{UbxOp}$ *opubx* (*take ar* Σ) = *Some x* $\Longrightarrow$
State F H (*Frame f pc* Σ # *st*) $\rightarrow$ *State F H* (*Frame f* (*Suc pc*) (*x* # *drop ar* Σ) # *st*) |

step-cjump-true:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *ICJump n* $\Longrightarrow$
cast-Dyn y = *Some d* $\Longrightarrow$ *is-true d* $\Longrightarrow$
State F H (*Frame f pc* (*y* # Σ) # *st*) $\rightarrow$ *State F H* (*Frame f n* Σ # *st*) |

step-cjump-false:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *ICJump n* $\Longrightarrow$
cast-Dyn y = *Some d* $\Longrightarrow$ *is-false d* $\Longrightarrow$
State F H (*Frame f pc* (*y* # Σ) # *st*) $\rightarrow$ *State F H* (*Frame f* (*Suc pc*) Σ # *st*) |

step-fun-call:
F-get F f = *Some fd* $\Longrightarrow$ *pc* < *length* (*body fd*) $\Longrightarrow$ *body fd* ! *pc* = *ICall g* $\Longrightarrow$
F-get F g = *Some gd* $\Longrightarrow$ *arity gd* = *ar* $\Longrightarrow$ *ar* $\leq$ *length* $\Sigma \Longrightarrow$
$frame_f$ = *Frame f pc* $\Sigma \Longrightarrow$ *list-all is-dyn-operand* (*take ar* Σ) $\Longrightarrow$
$frame_g$ = *Frame g 0* (*take ar* Σ) $\Longrightarrow$
State F H ($frame_f$ # *st*) $\rightarrow$ *State F H* ($frame_g$ # $frame_f$ # *st*) |

step-fun-end:
F-get F g = *Some gd* $\Longrightarrow$ *arity gd* $\leq$ *length* $\Sigma_f \Longrightarrow$ pc_g = *length* (*body gd*) $\Longrightarrow$
$frame_g$ = *Frame g* pc_g $\Sigma_g \Longrightarrow$ $frame_f$ = *Frame f* pc_f $\Sigma_f \Longrightarrow$
$frame_f{}'$ = *Frame f* (*Suc* pc_f) (Σ_g @ *drop* (*arity gd*) Σ_f) $\Longrightarrow$
State F H ($frame_g$ # $frame_f$ # *st*) $\rightarrow$ *State F H* ($frame_f{}'$ # *st*)

theorem *step-deterministic*: *s1* $\rightarrow$ *s2* $\Longrightarrow$ *s1* $\rightarrow$ *s3* $\Longrightarrow$ *s2* = *s3*
by (*induction rule*: *step.induct*;

erule step.cases; *safe*;
auto dest: *is-true-and-is-false-implies-False*)

lemma *final-finished*: *final s* $\Longrightarrow$ *finished step s*
unfolding *finished-def*
proof
assume *final s* **and** $\exists\, x.$ *step s x*
then obtain s' **where** *step s* s'
by *auto*
thus *False*
using ‹*final s*›
by (*auto elim*!: *step.cases final.cases*)
qed

sublocale *ubx-sem*: *semantics step final*
using *final-finished step-deterministic*
by *unfold-locales*

inductive *load* :: ('*fenv*, '*a*, '*fun*) *prog* ⇒ ('*fenv*, '*a*, ('*fun*, '*b*) *frame*) *state* ⇒ *bool*
where
F-get F main = *Some fd* $\Longrightarrow$ *arity fd* = *0* $\Longrightarrow$
load (*Prog F H main*) (*State F H* [*Frame main 0* []])

sublocale *inca-lang*: *language step final load*
by *unfold-locales*

end

end
theory *Ubx-type-inference*
imports *Result Ubx*
HOL−Library.Monad-Syntax
begin

11 Type of operations

locale *ubx-sp* =
ubx
F-empty F-get F-add F-to-list
heap-empty heap-get heap-add heap-to-list
is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩 𝔘𝔟𝔵𝔒𝔭 𝔘𝔟𝔵 𝔅𝔬𝔵 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭
for
— Functions environment
F-empty **and** *F-get* **and** *F-add* **and** *F-to-list* **and**

— Memory heap
heap-empty **and** *heap-get* **and** *heap-add* **and** *heap-to-list* **and**

— Unboxed values
is-true **and** *is-false* **and**
box-ubx1 **and** *unbox-ubx1* **and**
box-ubx2 **and** *unbox-ubx2* **and**

— n-ary operations
𝔒𝔭 and 𝔄𝔯𝔦𝔱𝔶 and 𝔍𝔫𝔩𝔒𝔭 and 𝔍𝔫𝔩 and 𝔍𝔰𝔍𝔫𝔩 and 𝔇𝔢𝔍𝔫𝔩 and 𝔘𝔟𝔵𝔒𝔭 and 𝔘𝔟𝔵 and 𝔅𝔬𝔵 and 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭
begin

12 Strongest postcondition of instructions

fun *sp-gen-pop-push* **where**
sp-gen-pop-push (*domain*, *codom*) Σ = (
let ar = *length domain in*
if ar ≤ *length* Σ ∧ *take ar* Σ = *domain then*
Ok (*codom* # *drop ar* Σ)
else
Error ()
)

fun *sp-instr* :: (*'fun* ⇒ - *fundef option*) ⇒ - ⇒ *type option list* ⇒ (*unit*, *type option list*) *result* **where**
sp-instr - (*IPush* -) Σ = *Ok* (*None* # Σ) |
sp-instr - (*IPushUbx1* -) Σ = *Ok* (*Some Ubx1* # Σ) |
sp-instr - (*IPushUbx2* -) Σ = *Ok* (*Some Ubx2* # Σ) |
sp-instr - *IPop* (- # Σ) = *Ok* Σ |
sp-instr - (*ILoad* -) (*None* # Σ) = *Ok* (*None* # Σ) |
sp-instr - (*ILoadUbx* τ -) (*None* # Σ) = *Ok* (*Some* τ # Σ) |
sp-instr - (*IStore* -) (*None* # *None* # Σ) = *Ok* Σ |
sp-instr - (*IStoreUbx* τ_1 -) (*None* # *Some* τ_2 # Σ) = (*if* $\tau_1 = \tau_2$ *then Ok* Σ *else Error* ()) |
sp-instr - (*IOp op*) Σ = *sp-gen-pop-push* (*replicate* (𝔄𝔯𝔦𝔱𝔶 *op*) *None*, *None*) Σ |
sp-instr - (*IOpInl opinl*) Σ = *sp-gen-pop-push* (*replicate* (𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 *opinl*)) *None*, *None*) Σ |
sp-instr - (*IOpUbx opubx*) Σ = *sp-gen-pop-push* (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 *opubx*) Σ |
sp-instr F (*ICall f*) Σ = *do* {
fd ← *Result.of-option* () (*F f*);
sp-gen-pop-push (*replicate* (*arity fd*) *None*, *None*) Σ
} |
sp-instr - - - = *Error* ()

lemma *sp-instr-no-jump*: *sp-instr F instr* Σ = *Ok type* ⟹ ¬ *is-jump instr*
by (*induction instr* Σ *rule*: *sp-instr.induct*) *simp-all*

lemma *map-constant*[*simp*]: ∀ *x* ∈ *set xs*. *x* = *y* ⟹ *map* (λ-. *y*) *xs* = *xs*
by (*simp add*: *map-idI*)

lemma *sp-generalize-instr*:

assumes *sp-instr F x Σ = Ok Σ′*
shows *sp-instr F* (*generalize-instr x*) (*map* (λ-. *None*) Σ) = *Ok* (*map* (λ-. *None*) Σ′)
using *assms*
apply (*induction F x Σ rule*: *sp-instr.induct*;
(*auto simp*: *Let-def take-map drop-map*; *fail*)*?*)
subgoal for - *opubx*
apply (*cases* TypeOfOp *opubx*)
by (*auto simp add*: *Let-def take-map drop-map map-replicate-const* TypeOfOp-Arity)
done

lemma *map-typeof-box*: *map typeof* (*map box-operand* Σ) = *replicate* (*length* Σ) *None*
proof (*induction* Σ)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
then show *?case*
by (*cases x*) *simp-all*
qed

13 Strongest postcondition of function definitions

fun *sp* :: (′*fun* ⇒ - *fundef option*) ⇒ - *list* ⇒ *type option list* ⇒ (*unit*, *type option list*) *result* **where**
sp - [] Σ = *Ok* Σ |
sp F (*instr* # *p*) Σ = *do* {
Σ′ ← *sp-instr F instr* Σ;
sp F p Σ′
}

lemma *sp-no-jump*: *sp F xs* Σ = *Ok type* ⟹ ∀ *x* ∈ *set xs*. ¬ *is-jump x*
by (*induction xs arbitrary*: Σ; *auto dest*: *sp-instr-no-jump*)

lemma *sp-append*: *sp F* (*xs* @ *ys*) Σ = *sp F xs* Σ ⟜ *sp F ys*
proof (*induction xs arbitrary*: Σ)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
then show *?case*
by (*cases sp-instr F x* Σ; *simp*)
qed

lemmas *sp-eq-bind-take-drop* =
sp-append[*of* - *take n xs drop n xs* **for** *n xs*, *unfolded append-take-drop-id*]

lemma *sp-generalize*:

assumes *sp F xs* Σ = *Ok* Σ′
shows *sp F* (*map generalize-instr xs*) (*map* (λ-. *None*) Σ) = *Ok* (*map* (λ-. *None*) Σ′)
using *assms*
proof (*induction xs arbitrary*: Σ Σ′)
case *Nil*
then show *?case* **using** *assms*(*1*) **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof (*cases sp-instr F x* Σ)
case (*Error x1*)
then show *?thesis*
using *Cons.prems* **by** *auto*
next
case (*Ok* Σ′′)
show *?thesis*
using *Cons sp-generalize-instr*[*OF Ok*]
by (*simp add*: *Ok*)
qed
qed

lemma *sp-generalize2*:
assumes *sp F xs* (*replicate n None*) = *Ok* Σ′
shows *sp F* (*map generalize-instr xs*) (*replicate n None*) = *Ok* (*map* (λ-. *None*) Σ′)
using *assms*
using *sp-generalize* **by** *fastforce*

lemma *comp-K*[*simp*]: (λ-. *x*) ∘ *f* = (λ-. *x*)
by *auto*

lemma *is-ok-sp-generalize*:
assumes *is-ok* (*sp F xs* (*map* (λ-. *None*) Σ))
shows *is-ok* (*sp F* (*map generalize-instr xs*) (*map* (λ-. *None*) Σ))
proof −
from *assms* **obtain** *res* **where** *0*: *sp F xs* (*map* (λ-. *None*) Σ) = *Ok res*
by (*auto simp add*: *is-ok-def*)
show *?thesis*
using *sp-generalize*[*OF 0*] **by** *simp*
qed

lemma *is-ok-sp-generalize2*:
assumes *is-ok* (*sp F xs* (*replicate n None*))
shows *is-ok* (*sp F* (*map generalize-instr xs*) (*replicate n None*))
using *assms is-ok-sp-generalize*
by (*metis Ex-list-of-length map-replicate-const*)

lemma *sp-instr-op*:

assumes $\mathfrak{Defnl}$ *opinl* = *op*
shows *sp-instr F* (*IOp op*) Σ = *sp-instr F* (*IOpInl opinl*) Σ
by (*rule sp-instr.cases*[*of* (*F*, (*IOp op*), Σ)]; *simp add*: *Let-def assms*)

lemma *sp-list-update*:
assumes
n < *length xs* **and**
∀Σ. *sp-instr F* (*xs* ! *n*) Σ = *sp-instr F x* Σ
shows *sp F* (*xs*[*n* := *x*]) = *sp F xs*
proof (*intro ext*)
fix *ys*
have *sp F* (*take n xs* @ *x* # *drop* (*Suc n*) *xs*) *ys* = *sp F* (*take n xs* @ *xs* ! *n* # *drop* (*Suc n*) *xs*) *ys*
unfolding *sp-append*
using *assms*(*2*)
by (*cases sp F* (*take n xs*) *ys*; *simp*)
thus *sp F* (*xs*[*n* := *x*]) *ys* = *sp F xs ys*
unfolding *id-take-nth-drop*[*OF assms*(*1*), *symmetric*]
unfolding *upd-conv-take-nth-drop*[*OF assms*(*1*), *symmetric*]
by *assumption*
qed

lemma *sp-list-update-eq-Ok*:
assumes
n < *length xs* **and**
∀Σ. *sp-instr F* (*xs* ! *n*) Σ = *sp-instr F x* Σ **and**
sp F xs ys = *Ok zs*
shows *sp F* (*xs*[*n* := *x*]) *ys* = *Ok zs*
unfolding *sp-list-update*[*OF assms*(*1*,*2*)]
using *assms*(*3*)
by *assumption*

lemma *is-ok-sp-list-update*:
assumes
is-ok (*sp F xs types*) **and**
pc < *length xs* **and**
∀Σ. *sp-instr F* (*xs* ! *pc*) Σ = *sp-instr F instr* Σ
shows *is-ok* (*sp F* (*xs*[*pc* := *instr*]) *types*)
proof −
from *assms*(*1*) **obtain** *types'* **where** *sp F xs types* = *Ok types'*
unfolding *is-ok-def* **by** *auto*
thus *?thesis*
using *sp-list-update*[*OF assms*(*2*,*3*)] **by** *simp*
qed

lemmas *sp-rewrite* = *sp-list-update*[*folded rewrite-def*]
lemmas *sp-rewrite-eq-Ok* = *sp-list-update-eq-Ok*[*folded rewrite-def*]
lemmas *is-ok-sp-rewrite* = *is-ok-sp-list-update*[*folded rewrite-def*]

end

end
theory *Unboxed-lemmas*
imports *Unboxed*
begin

lemma *typeof-bind-OpDyn*[*simp*]: *typeof* ∘ *OpDyn* = (λ-. *None*)
by *auto*

lemma *is-dyn-operand-eq-typeof*: *is-dyn-operand* = (λ*x*. *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* ⟷ (∃ *d*. *x* = *OpDyn d*)
typeof x = *Some Ubx1* ⟷ (∃ *n*. *x* = *OpUbx1 n*)
typeof x = *Some Ubx2* ⟷ (∃ *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* ⟹ *x* = *OpDyn d*
cast-Ubx1 x = *Some n* ⟹ *x* = *OpUbx1 n*
cast-Ubx2 x = *Some b* ⟹ *x* = *OpUbx2 b*
by (*cases x*; *simp*)+

lemma *traverse-cast-Dyn-replicate*:
assumes *traverse 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.prems* **show** *?case*
by (*auto intro*: *Cons.IH dest*: *cast-inversions*(*1*) *simp add*: *bind-eq-Some-conv*)
qed

context *unboxedval* **begin**

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

lemma *cast-and-box-imp-typeof*[*simp*]: *cast-and-box* τ *blob* = *Some d* $\Longrightarrow$ *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* $\Longrightarrow$ *norm-unboxed blob* = *d*
using *box-unbox-inverse*
by (*cases* τ; *auto*)

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

lemma *typeof-and-norm-unboxed-imp-cast-and-box*:
assumes *typeof* x' = *Some* τ *norm-unboxed* x' = x
shows *cast-and-box* τ x' = *Some x*
using *assms*
by (*induction* τ; *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 x*) = *None*
by (*cases x*; *simp*)

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* = (λ-. *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 Option-applicative Result*
VeriComp.Simulation
Inca Ubx Ubx-type-inference Unboxed-lemmas
begin

14 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
is-true is-false
𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩 +
Subx: *ubx-sp*
Fubx-empty Fubx-get Fubx-add Fubx-to-list
heap-empty heap-get heap-add heap-to-list
is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩 𝔘𝔟𝔵𝔒𝔭 𝔘𝔟𝔵 𝔅𝔬𝔵 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭
for
— Functions environments
Finca-empty **and**
Finca-get :: *'fenv-inca* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) *Inca.instr fundef option* **and**
Finca-add **and** *Finca-to-list* **and**

Fubx-empty **and**
Fubx-get :: *'fenv-ubx* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'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* ⇒ *'var* × *'dyn* ⇒ *'dyn option* **and** *heap-add* **and** *heap-to-list* **and**

— Unboxed values
is-true **and** *is-false* **and**
box-ubx1 **and** *unbox-ubx1* **and**
box-ubx2 **and** *unbox-ubx2* **and**

— n-ary operations
𝔒𝔭 **and** 𝔄𝔯𝔦𝔱𝔶 **and** 𝔍𝔫𝔩𝔒𝔭 **and** 𝔍𝔫𝔩 **and** 𝔍𝔰𝔍𝔫𝔩 **and** 𝔇𝔢𝔍𝔫𝔩 **and** 𝔘𝔟𝔵𝔒𝔭 **and** 𝔘𝔟𝔵 **and** 𝔅𝔬𝔵 **and** 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭
begin

declare *Subx.sp-append*[*simp*]

15 Strongest postcondition

definition *sp-fundef* **where**
sp-fundef F fd xs ≡ *Subx.sp F xs* (*replicate* (*arity fd*) *None*)

lemmas *sp-fundef-generalize* =
Subx.sp-generalize[**where** Σ = *replicate* (*arity fd*) *None* **for** *fd*, *simplified*, *folded sp-fundef-def*]

lemma *eq-sp-to-eq-sp-fundef*:
assumes *Subx.sp F1* = (*Subx.sp F2* :: (*'dyn*, *'id*, -, -, -, -, *'num*, *'bool*) *Ubx.instr list* ⇒ -)
shows *sp-fundef F1* = (*sp-fundef F2* :: - ⇒ (*'dyn*, *'id*, -, -, -, -, *'num*, *'bool*) *Ubx.instr list* ⇒ -)
using *assms*(*1*)
by (*intro ext*; *simp add*: *sp-fundef-def*)

definition *sp-fundefs* **where**
sp-fundefs F ≡ ∀ *f fd*. *F f* = *Some fd* ⟶ *sp-fundef F fd* (*body fd*) = *Ok* [*None*]

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.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 n*) = *Inca.ICJump n* |
norm-instr (*Ubx.ICall x*) = *Inca.ICall x*

definition *rel-fundefs* **where**
rel-fundefs f g = (∀ *x*. *rel-option* (*rel-fundef* (λ*y z*. *y* = *norm-instr z*)) (*f x*) (*g x*))

abbreviation (*input*) *norm-eq* **where**
norm-eq x y ≡ *x* = *norm-instr y*

lemma *norm-generalize-instr*: *norm-instr* (*Subx.generalize-instr instr*) = *norm-instr instr*
by (*cases instr*) *simp-all*

lemma *rel-fundef-body-nth*:
assumes *rel-fundef norm-eq fd1 fd2* **and** *pc* < *length* (*body fd1*)

```
  shows body fd1 ! pc = norm-instr (body fd2 ! pc)
  using assms
  by (auto dest: list-all2-nthD simp: fundef.rel-sel)

lemma rel-fundef-rewrite-body:
  assumes
    rel-fundef norm-eq fd1 fd2 and
    norm-instr (body fd2 ! pc) = norm-instr instr
  shows rel-fundef norm-eq fd1 (rewrite-fundef-body fd2 pc instr)
  using assms(1)
proof (cases rule: fundef.rel-cases)
  case (Fundef xs ar' ys ar)
  hence length xs = length ys
    by (simp add: list-all2-conv-all-nth)
  hence length xs = length (rewrite ys pc instr)
    by simp
  hence list-all2 norm-eq xs (rewrite ys pc instr)
  proof (elim list-all2-all-nthI)
    fix n
    assume n < length xs
    hence n < length ys
      by (simp add: ‹length xs = length ys›)
    thus xs ! n = norm-instr (rewrite ys pc instr ! n)
      using list-all2-nthD[OF ‹list-all2 norm-eq xs ys› ‹n < length xs›, symmetric]
      using assms(2)[unfolded Fundef(2), simplified]
      by (cases pc = n; simp)
  qed
  thus ?thesis
    using Fundef by simp
qed

lemma rel-fundef-generalize:
  assumes rel-fundef norm-eq fd1 fd2
  shows rel-fundef norm-eq fd1 (Subx.generalize-fundef fd2)
  using assms(1)
proof (cases rule: fundef.rel-cases)
  case (Fundef xs ar' ys ar)
  hence list-all2 (λx y. x = norm-instr y) xs (map Subx.generalize-instr ys)
    by (auto elim!: list-all2-mono simp: list.rel-map norm-generalize-instr)
  thus ?thesis
    using Fundef by simp
qed

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

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 get fd2 n Σ*2 g* ≡ *n* < *length* (*body fd2*) ∧ *body fd2* ! *n* = *ICall g* ∧
      (∃ *gd*. *get g* = *Some gd* ∧ *list-all is-dyn-operand* (*take* (*arity gd*) Σ*2*))

inductive *rel-stacktraces* **for** *get* **where**
  rel-stacktraces-Nil:
    rel-stacktraces get [] [] *opt* |

  rel-stacktraces-Cons:
    rel-stacktraces get st1 st2 (*Some f*) ⟹
    Σ*1* = *norm-stack* Σ*2* ⟹
    get f = *Some fd2* ⟹
    sp-fundef get fd2 (*take n* (*body fd2*)) = *Ok* (*map typeof* Σ*2*) ⟹
    pred-option (*is-valid-fun-call get fd2 n* Σ*2*) *opt* ⟹
    rel-stacktraces get (*Frame f n* Σ*1* # *st1*) (*Frame f n* Σ*2* # *st2*) *opt*

definition *all-same-arities* **where**
  all-same-arities F1 F2 ≡ ∀ *f*. *rel-option* (λ*fd gd*. *arity fd* = *arity gd*) (*F1 f*) (*F2 f*)

lemma *all-same-arities-commutative*: *all-same-arities F1 F2* = *all-same-arities F2 F1*
proof
  assume *all-same-arities F1 F2*
  then show *all-same-arities F2 F1*
    unfolding *all-same-arities-def*
    by (*simp add*: *rel-option-unfold*)
next
  assume *all-same-arities F2 F1*
  then show *all-same-arities F1 F2*
    unfolding *all-same-arities-def*
    by (*simp add*: *rel-option-unfold*)

qed

lemma *sp-instr-same-arities*:
all-same-arities F1 F2 $\Longrightarrow$ *Subx.sp-instr F1 x ys* = *Subx.sp-instr F2 x ys*
proof (*induction F1 x ys rule*: *Subx.sp-instr.induct*)
fix *F1 f* Σ
assume *assms*: *all-same-arities F1 F2*
show *Subx.sp-instr F1* (*Ubx.ICall f*) Σ = *Subx.sp-instr F2* (*Ubx.ICall f*) Σ
proof (*cases F1 f*)
case *None*
then have *F2 f* = *None*
using *HOL.spec*[*OF assms*[*unfolded all-same-arities-def*], *of f*] **by** *simp*
with *None* **show** *?thesis* **by** *simp*
next
case (*Some a*)
then obtain *b* **where** *F2 f* = *Some b* **and** *arity a* = *arity b*
using *HOL.spec*[*OF assms*[*unfolded all-same-arities-def*], *of f*] **by** (*auto simp*: *option-rel-Some1*)
with *Some* **show** *?thesis* **by** *simp*
qed
qed *simp-all*

lemma *sp-same-arities*:
assumes *all-same-arities F1 F2*
shows *Subx.sp F1* = *Subx.sp F2*
proof (*intro ext allI*)
fix *xs ys*
show *Subx.sp F1 xs ys* = *Subx.sp F2 xs ys*
proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
apply *simp*
apply (*cases* (*F1*, *x*, *ys*) *rule*: *Subx.sp-instr.cases*;
simp add: *Cons.IH Let-def*)
subgoal for *opubx*
apply (*cases* $\mathfrak{TypeOfOp}$ *opubx*)
by (*auto simp*: *Cons.IH Let-def*)
subgoal for *f*
apply (*rule option.rel-induct*[*of* (λ*fd gd*. *arity fd* = *arity gd*) *F1 f F2 f*])
using *assms*(*1*)[*unfolded all-same-arities-def*]
by (*auto simp add*: *Cons.IH*)
done
qed
qed

lemmas *sp-fundef-same-arities* = *sp-same-arities*[*THEN eq-sp-to-eq-sp-fundef*]

lemma *all-same-arities-add*:
assumes *Fubx-get F f = Some fd1* **and** *arity fd1 = arity fd2*
shows *all-same-arities* (*Fubx-get F*) (*Fubx-get* (*Fubx-add F f fd2*))
unfolding *all-same-arities-def*
proof (*intro allI*)
fix *g*
show *rel-option* (λ*fd gd*. *arity fd = arity gd*)
(*Fubx-get F g*)
(*Fubx-get* (*Fubx-add F f fd2*) *g*)
using *assms*
by (*cases f = g*) (*simp-all add*: *option.rel-refl*)
qed

lemma *all-same-arities-generalize-fundef*:
assumes *Fubx-get F f = Some fd*
shows *all-same-arities* (*Fubx-get F*) (*Fubx-get* (*Fubx-add F f* (*Subx.generalize-fundef fd*)))
using *all-same-arities-add*[*OF assms*(*1*)]
using *ubx.arity-generalize-fundef*
by *simp*

lemma *rel-stacktraces-box*:
assumes
rel-stacktraces F1 xs ys opt **and**
F2 f = map-option Subx.generalize-fundef (*F1 f*) **and**
⋀*g*. *f ≠ g* ⟹ *F2 g = F1 g* **and**
all-same-arities F1 F2
shows *rel-stacktraces F2 xs* (*Subx.box-stack f ys*) *opt*
using *assms*(*1*)
proof (*induction xs ys opt rule*: *rel-stacktraces.induct*)
case *rel-stacktraces-Nil*
then show *?case*
by (*auto intro*: *rel-stacktraces.intros*)
next
case (*rel-stacktraces-Cons st1 st2 g Σ1 Σ2 gd n opt*)
note *sp-F1-eq-sp-F2*[*simp*] = *sp-same-arities*[*OF assms*(*4*)]
note *sp-fundef-F1-eq-sp-fundef-F2*[*simp*] = *eq-sp-to-eq-sp-fundef*[*OF sp-F1-eq-sp-F2*]
show *?case*
apply *simp*
proof (*intro conjI impI*)
assume *f = g*
then have *get2-g*: *F2 g = Some* (*Subx.generalize-fundef gd*)
using *assms*(*2*) ‹*F1 g = Some gd*› **by** *simp*

show *rel-stacktraces F2* (*Frame g n Σ1 # st1*)
(*Frame g n* (*map Subx.box-operand Σ2*) *# Subx.box-stack g st2*) *opt*
proof (*intro rel-stacktraces.intros*)
show *rel-stacktraces F2 st1* (*Subx.box-stack g st2*) (*Some g*)

```
      using rel-stacktraces-Cons.IH
      unfolding ‹f = g›
      by simp
  next
    show Σ1 = norm-stack (map Subx.box-operand Σ2)
      using ‹Σ1 = norm-stack Σ2› by simp
  next
    show F2 g = Some (Subx.generalize-fundef gd)
      using get2-g .
  next
  show sp-fundef F2 (Subx.generalize-fundef gd) (take n (body (Subx.generalize-fundef
gd))) =
       Ok (map typeof (map Subx.box-operand Σ2))
      using sp-fundef-generalize[OF rel-stacktraces-Cons.hyps(4)]
      by (simp add: take-map sp-fundef-def)
  next
    show pred-option
      (is-valid-fun-call F2 (Subx.generalize-fundef gd) n (map Subx.box-operand
Σ2)) opt
    proof (cases opt)
      case (Some h)
      then show ?thesis
        using rel-stacktraces-Cons.hyps(5)
      apply (simp add: take-map list.pred-map list.pred-True is-valid-fun-call-def)
        using ‹f = g› assms(3) get2-g by fastforce
    qed simp
  qed
next
  assume f ≠ g
  show rel-stacktraces F2 (Frame g n Σ1 # st1) (Frame g n Σ2 # Subx.box-stack
f st2) opt
    using rel-stacktraces-Cons assms(3)[OF ‹f ≠ g›]
    apply (auto intro!: rel-stacktraces.intros;
        cases opt; simp add: sp-fundef-def is-valid-fun-call-def)
    subgoal for h
      using assms(2,3) by (cases f = h; auto)
    done
qed
qed

lemma rel-stacktraces-generalize:
  assumes
    rel-stacktraces (Fubx-get F) st1 st2 (Some f) and
    Fubx-get F f = Some fd  and
     sp-prefix: sp-fundef (Fubx-get F) fd (take pc (body fd)) = Ok (None # map
typeof Σ2) and
    norm-stacks: Σ1 = norm-stack Σ2 and
    pc-in-range: pc < length (body fd) and
    sp-instr: Subx.sp-instr (Fubx-get F) (Subx.generalize-instr (body fd ! pc))
```

```
    (None # map (λ-. None) Σ2) = Ok (None # map (typeof ∘ Subx.box-operand)
Σ2)
  shows rel-stacktraces (Fubx-get (Fubx-add F f (Subx.generalize-fundef fd)))
         (Frame f (Suc pc) (d # Σ1) # st1)
        (Frame f (Suc pc) (OpDyn d # map Subx.box-operand Σ2) # Subx.box-stack
f st2) None
proof −
  let ?fd′ = Subx.generalize-fundef fd
  let ?F′ = Fubx-add F f ?fd′
  show ?thesis
  proof (intro rel-stacktraces-Cons)
    show rel-stacktraces (Fubx-get ?F′) st1 (Subx.box-stack f st2) (Some f)
      using assms(2) all-same-arities-generalize-fundef
      by (auto intro: rel-stacktraces-box[OF assms(1)])
  next
    show d # Σ1 = norm-stack (OpDyn d # map Subx.box-operand Σ2)
      using norm-stacks by simp
  next
    show Fubx-get ?F′ f = Some ?fd′
      by simp
  next
    show sp-fundef (Fubx-get ?F′) ?fd′ (take (Suc pc) (body ?fd′)) =
      Ok (map typeof (OpDyn d # map Subx.box-operand Σ2))
      unfolding all-same-arities-generalize-fundef[THEN sp-fundef-same-arities,
          OF assms(2), symmetric]
      using sp-fundef-generalize[OF sp-prefix] sp-instr
    by (auto simp add: sp-fundef-def take-map take-Suc-conv-app-nth[OF pc-in-range])
  qed simp-all
qed

lemma rel-fundefs-rewrite:
  assumes
    rel-F1-F2: rel-fundefs (Finca-get F1) (Fubx-get F2) and
    F2-get-f: Fubx-get F2 f = Some fd2 and
    F2-add-f: Fubx-add F2 f (rewrite-fundef-body fd2 pc instr) = F2′ and
    eq-norm: norm-instr (body fd2 ! pc) = norm-instr instr
  shows rel-fundefs (Finca-get F1) (Fubx-get F2′)
  unfolding rel-fundefs-def
proof
  fix x
  show rel-option (rel-fundef norm-eq) (Finca-get F1 x) (Fubx-get F2′ x)
  proof (cases x = f)
    case True
    then have F2′-get-x: Fubx-get F2′ x = Some (rewrite-fundef-body fd2 pc instr)
      using F2-add-f by auto
     obtain fd1 where 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-get-f] by auto
    thus ?thesis
```

```
    unfolding F2'-get-x option-rel-Some2
    using True rel-fundef-rewrite-body[OF rel-fd1-fd2 eq-norm]
    by auto
  next
    case False
    then have Fubx-get F2' x = Fubx-get F2 x
      using F2-add-f by auto
    then show ?thesis
      using rel-F1-F2 rel-fundefs-def
      by fastforce
  qed
qed

lemma rel-fundef-rewrite-both:
  assumes rel-fundef norm-eq fd1 fd2 and norm-instr y = x
  shows rel-fundef norm-eq (rewrite-fundef-body fd1 pc x) (rewrite-fundef-body fd2
pc y)
  using assms by (auto simp: fundef.rel-sel)

lemma rel-fundefs-rewrite-both:
  assumes
    rel-init: rel-fundefs (Finca-get F1) (Fubx-get F2) and
    rel-new: rel-fundef norm-eq fd1 fd2
  shows rel-fundefs (Finca-get (Finca-add F1 f fd1)) (Fubx-get (Fubx-add F2 f fd2))
  unfolding rel-fundefs-def
proof
  fix x
  show rel-option (rel-fundef norm-eq) (Finca-get (Finca-add F1 f fd1) x) (Fubx-get
(Fubx-add F2 f fd2) x)
  proof (cases x = f)
    case True
    then show ?thesis
      using rel-new by simp
  next
    case False
    then show ?thesis
      using rel-init
      by (simp add: rel-fundefs-def)
  qed
qed

lemma rel-fundefs-generalize:
  assumes
    rel-F1-F2: rel-fundefs (Finca-get F1) (Fubx-get F2) and
    F2-get-f: Fubx-get F2 f = Some fd2
  shows rel-fundefs (Finca-get F1) (Fubx-get (Fubx-add F2 f (Subx.generalize-fundef
fd2)))
  unfolding rel-fundefs-def
proof
```

```
  let ?F2' = (Fubx-add F2 f (Subx.generalize-fundef fd2))
  fix x
  show rel-option (rel-fundef norm-eq) (Finca-get F1 x) (Fubx-get ?F2' x)
  proof (cases x = f)
    case True
    then have F2'-get-x: Fubx-get ?F2' x = Some (Subx.generalize-fundef fd2)
      using Subx.Fenv.get-add-eq by auto
    obtain fd1 where Finca-get F1 f = Some fd1 and rel-fundef norm-eq fd1 fd2
      using rel-fundefs-Some2[OF rel-F1-F2 F2-get-f] by auto
    thus ?thesis
      unfolding F2'-get-x option-rel-Some2
      using True rel-fundef-generalize
      by auto
  next
    case False
    then have Fubx-get ?F2' x = Fubx-get F2 x
      using Subx.Fenv.get-add-neq by auto
    then show ?thesis
      using rel-F1-F2 rel-fundefs-def
      by fastforce
  qed
qed

lemma rel-stacktraces-rewrite-fundef:
  assumes
    rel-stacktraces (Fubx-get F2) xs ys opt and
    Fubx-get F2 f = Some fd and
    pc < length (body fd) and
   ∀Σ. Subx.sp-instr (Fubx-get F2) (body fd ! pc) Σ = Subx.sp-instr (Fubx-get F2)
instr Σ and
    ¬ is-fun-call (body fd ! pc)
  shows rel-stacktraces
     (Fubx-get (Fubx-add F2 f (rewrite-fundef-body fd pc instr))) xs ys opt
  using assms(1)
proof (induction xs ys opt rule: rel-stacktraces.induct)
  case rel-stacktraces-Nil
  then show ?case
    by (auto intro: rel-stacktraces.intros)
next
  case (rel-stacktraces-Cons st1 st2 g Σ1 Σ2 gd n opt)
  have same-arities: all-same-arities (Fubx-get F2)
    (Fubx-get (Fubx-add F2 f (rewrite-fundef-body fd pc instr)))
    using all-same-arities-add[OF assms(2)]
    by simp

  show ?case
  proof (cases g = f)
    case True
    then have gd = fd
```

```
      using assms(2) rel-stacktraces-Cons.hyps(3) by auto
    thus ?thesis
      using True rel-stacktraces-Cons
      apply (auto intro!: rel-stacktraces.intros
        simp: sp-fundef-same-arities[OF same-arities, symmetric])
      apply (cases n ≤ pc; simp add: sp-fundef-def)
      subgoal
        using assms(3,4)
        by (simp add: Subx.sp-rewrite-eq-Ok take-rewrite-swap)
      subgoal
         using rel-stacktraces-Cons.hyps(5)
       proof (induction opt)
         case (Some h)
         hence rewrite (body fd) pc instr ! n = Ubx.ICall h
           using ‹gd = fd›
           apply (simp add: is-valid-fun-call-def)
           by (metis assms(5) instr.disc nth-rewrite-neq)
         then show ?case
           using Some ‹gd = fd›
           apply (simp add: is-valid-fun-call-def)
       by (metis arity-rewrite-fundef-body assms(2) option.inject Subx.Fenv.get-add-eq
Subx.Fenv.get-add-neq)
      qed simp
      done
  next
    case False
    thus ?thesis
      using rel-stacktraces-Cons
      apply (auto intro!: rel-stacktraces.intros
        simp: sp-fundef-same-arities[OF same-arities, symmetric])
    proof (induction opt)
      case (Some h)
      then show ?case
        using Some assms(2)
        by (cases h = f; simp add: is-valid-fun-call-def)
    qed simp
  qed
qed
```

18 Matching relation

```
lemma sp-fundefs-get:
  assumes sp-fundefs F and F f = Some fd
  shows sp-fundef F fd (body fd) = Ok [None]
  using assms by (simp add: sp-fundefs-def)

lemma sp-fundefs-generalize:
  assumes sp-fundefs (Fubx-get F) and Fubx-get F f = Some fd
  shows sp-fundefs (Fubx-get (Fubx-add F f (Subx.generalize-fundef fd)))
```

```
  unfolding sp-fundefs-def
proof (intro allI impI)
  fix g gd
  assume get-g: Fubx-get (Fubx-add F f (Subx.generalize-fundef fd)) g = Some gd
   note sp-F-eq-sp-F' = all-same-arities-generalize-fundef[OF assms(2), THEN
sp-same-arities, symmetric]
  show sp-fundef (Fubx-get (Fubx-add F f (Subx.generalize-fundef fd))) gd (body
gd) = Ok [None]
  proof (cases f = g)
    case True
    then have gd = Subx.generalize-fundef fd
      using get-g by simp
    then show ?thesis
      using assms
      by (simp add: sp-fundefs-def sp-fundef-def sp-F-eq-sp-F' Subx.sp-generalize2)
  next
    case False
    then show ?thesis
      using get-g assms
      by (simp add: sp-fundefs-def sp-fundef-def sp-F-eq-sp-F')
  qed
qed

lemma sp-fundefs-add:
  assumes
    sp-fundefs (Fubx-get F) and
    sp-fundef (Fubx-get F) fd (body fd) = Ok [None] and
    all-same-arities (Fubx-get F) (Fubx-get (Fubx-add F f fd))
  shows sp-fundefs (Fubx-get (Fubx-add F f fd))
  unfolding sp-fundefs-def
proof (intro allI impI)
  fix g gd
  assume Fubx-get (Fubx-add F f fd) g = Some gd
  show sp-fundef (Fubx-get (Fubx-add F f fd)) gd (body gd) = Ok [None]
  proof (cases f = g)
    case True
    then have gd = fd
      using ‹Fubx-get (Fubx-add F f fd) g = Some gd› by simp
    then show ?thesis
      unfolding sp-fundef-same-arities[OF assms(3), symmetric]
      using assms(2) by simp
  next
    case False
    then show ?thesis
      unfolding sp-fundef-same-arities[OF assms(3), symmetric]
      using ‹Fubx-get (Fubx-add F f fd) g = Some gd›
      using sp-fundefs-get[OF assms(1)]
      by simp
  qed
```

qed

inductive *match* (**infix** $\sim$ *55*) **where**
rel-fundefs (*Finca-get F1*) (*Fubx-get F2*) $\Longrightarrow$
sp-fundefs (*Fubx-get F2*) $\Longrightarrow$
rel-stacktraces (*Fubx-get F2*) *st1 st2 None* $\Longrightarrow$
match (*State F1 H st1*) (*State F2 H st2*)

19 Backward simulation

lemma *traverse-cast-Dyn-to-norm*: *traverse cast-Dyn xs* = *Some ys* $\Longrightarrow$ *norm-stack xs* = *ys*
proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons.prems* **show** *?case*
by (*auto intro*: *Cons.IH elim*: *cast-Dyn.elims simp*: *Option.bind-eq-Some-conv*)
qed

lemma *traverse-cast-Dyn-to-all-Dyn*:
traverse cast-Dyn xs = *Some ys* $\Longrightarrow$ *list-all* (λx. *typeof x* = *None*) *xs*
proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons.prems* **show** *?case*
by (*auto intro*: *Cons.IH elim*: *cast-Dyn.elims simp*: *Option.bind-eq-Some-conv*)
qed

lemma *backward-lockstep-simulation*:
assumes *Subx.step s2 s2′* **and** *s1* $\sim$ *s2*
shows $\exists$ *s1′*. *Sinca.step s1 s1′* $\wedge$ *s1′* $\sim$ *s2′*
using *assms*(*2*,*1*)
proof (*induction s1 s2 rule*: *match.induct*)
case (*1 F1 F2 st1 st2 H*)
have *rel-F1-F2*: *rel-fundefs* (*Finca-get F1*) (*Fubx-get F2*) **using** *1* **by** *simp*
have *sp-F2*: *sp-fundefs* (*Fubx-get F2*) **using** *1* **by** *simp*

from *1*(*3*,*4*) **show** *?case*
proof (*induction st1 st2 None* :: *'fun option rule*: *rel-stacktraces.induct*)
case *rel-stacktraces-Nil*
hence *False* **by** (*auto elim*: *Subx.step.cases*)
thus *?case* **by** *simp*
next
case (*rel-stacktraces-Cons st1 st2 f* Σ*1* Σ*2 fd2 pc*)
have *F2-f*: *Fubx-get F2 f* = *Some fd2*

```
    using rel-stacktraces-Cons by simp
  have rel-st1-st2: rel-stacktraces (Fubx-get F2) st1 st2 (Some f)
    using rel-stacktraces-Cons by simp
  have sp-fundef-prefix: sp-fundef (Fubx-get F2) fd2 (take pc (body fd2)) = Ok
(map typeof Σ2)
    using rel-stacktraces-Cons by simp
  have Σ1-def: Σ1 = norm-stack Σ2
    using rel-stacktraces-Cons by simp

  note sp-fundef-def[simp]
  note sp-prefix = sp-fundef-prefix[unfolded sp-fundef-def]

  have sp-generalized: Subx.sp (Fubx-get (Fubx-add F2 f (Subx.generalize-fundef
fd2)))
      (map Subx.generalize-instr (take pc (body fd2))) (replicate (arity fd2) None)
=
    Ok (map (λ-. None) Σ2)
    using Subx.sp-generalize[OF sp-prefix, simplified]
    using all-same-arities-generalize-fundef[OF F2-f]
    by (simp add: sp-same-arities)

  from rel-stacktraces-Cons.prems(1) show ?case
  proof (induction State F2 H (Frame f pc Σ2 # st2) s2' rule: Subx.step.induct)
    case (step-push fd2' d)
    then have [simp]: fd2' = fd2 using F2-f by simp
    obtain fd1 where fd1-thms: Finca-get F1 f = Some fd1 rel-fundef norm-eq
fd1 fd2
      using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F1 H (Frame f (Suc pc) (d # Σ1) # st1)
      have ?STEP ?s1'
        using step-push fd1-thms
        by (auto intro: Sinca.step-push simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1'
        using step-push rel-stacktraces-Cons rel-F1-F2 sp-F2
     by (auto intro!: match.intros intro: rel-stacktraces.intros simp: take-Suc-conv-app-nth)
      ultimately show ?thesis by blast
    qed
  next
    case (step-push-ubx1 fd2' n)
    then have [simp]: fd2' = fd2 using F2-f by simp
    obtain fd1 where fd1-thms: Finca-get F1 f = Some fd1 rel-fundef norm-eq
fd1 fd2
      using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F1 H (Frame f (Suc pc) (box-ubx1 n # Σ1) # st1)
      have ?STEP ?s1'
```

```
        using step-push-ubx1 fd1-thms
        by (auto intro: Sinca.step-push simp: rel-fundef-body-nth)
       moreover have ?MATCH ?s1'
        using step-push-ubx1 rel-stacktraces-Cons rel-F1-F2 sp-F2
      by (auto intro!: match.intros intro: rel-stacktraces.intros simp: take-Suc-conv-app-nth)
       ultimately show ?thesis by blast
      qed
    next
      case (step-push-ubx2 fd2' b)
      then have [simp]: fd2' = fd2 using F2-f by simp
      obtain fd1 where fd1-thms: Finca-get F1 f = Some fd1 rel-fundef norm-eq
fd1 fd2
        using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto
      show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
      proof −
        let ?s1' = State F1 H (Frame f (Suc pc) (box-ubx2 b # Σ1) # st1)
        have ?STEP ?s1'
          using step-push-ubx2 fd1-thms
          by (auto intro: Sinca.step-push simp: rel-fundef-body-nth)
        moreover have ?MATCH ?s1'
          using step-push-ubx2 rel-stacktraces-Cons rel-F1-F2 sp-F2
      by (auto intro!: match.intros intro: rel-stacktraces.intros simp: take-Suc-conv-app-nth)
        ultimately show ?thesis by blast
      qed
    next
      case (step-pop fd2' x Σ2')
      then have [simp]: fd2' = fd2 using F2-f by simp
      obtain fd1 where fd1-thms: Finca-get F1 f = Some fd1 rel-fundef norm-eq
fd1 fd2
        using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto
      show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
      proof −
        let ?s1' = State F1 H (Frame f (Suc pc) (norm-stack Σ2') # st1)
        have ?STEP ?s1'
          unfolding Σ1-def
          using step-pop fd1-thms
          by (auto intro!: Sinca.step-pop simp: rel-fundef-body-nth)
        moreover have ?MATCH ?s1'
          using step-pop rel-stacktraces-Cons rel-F1-F2 sp-F2
      by (auto intro!: match.intros intro: rel-stacktraces.intros simp: take-Suc-conv-app-nth)
        ultimately show ?thesis by blast
      qed
    next
      case (step-load fd2' x i i' d Σ2')
      then have [simp]: fd2' = fd2 using F2-f by simp
      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
```

```
  show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
  proof −
    let ?s1' = State F1 H (Frame f (Suc pc) (d # norm-stack Σ2') # st1)
    have pc-in-range: pc < length (body fd1)
      using rel-fd1-fd2 step-load(2) by simp
    have ?STEP ?s1'
      using step-load rel-fundef-body-nth[OF rel-fd1-fd2 pc-in-range]
      by (auto intro!: Sinca.step-load[OF F1-f pc-in-range, of x]
            dest: cast-inversions(1)
            simp: Σ1-def)
    moreover have ?MATCH ?s1'
      using step-load refl rel-stacktraces-Cons rel-F1-F2 sp-F2
     by (auto intro!: match.intros intro: rel-stacktraces.intros(2)[OF rel-st1-st2]
          dest!: cast-inversions(1)
          simp: take-Suc-conv-app-nth[OF step-load(2), simplified])
    ultimately show ?thesis by blast
  qed
next
  case (step-load-ubx-hit fd2' τ x i i' d blob Σ2')
  then have [simp]: fd2' = fd2 using F2-f by simp
  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
  show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
  proof −
    let ?s1' = State F1 H (Frame f (Suc pc) (d # norm-stack Σ2') # st1)
    have pc-in-range: pc < length (body fd1)
      using rel-fd1-fd2 step-load-ubx-hit(2) by simp
    have ?STEP ?s1'
      using step-load-ubx-hit rel-fundef-body-nth[OF rel-fd1-fd2 pc-in-range]
      by (auto intro!: Sinca.step-load[OF F1-f pc-in-range, of x]
          dest: cast-inversions(1)
          simp: rel-fundef-body-nth Σ1-def)
    moreover have ?MATCH ?s1'
      using step-load-ubx-hit rel-stacktraces-Cons rel-F1-F2 sp-F2
     by (auto intro!: match.intros intro: rel-stacktraces.intros(2)[OF rel-st1-st2]
          dest!: cast-inversions(1)
          simp: take-Suc-conv-app-nth[OF step-load-ubx-hit(2), simplified])
    ultimately show ?thesis by blast
  qed
next
  case (step-load-ubx-miss fd2' τ x i i' d F2' Σ2')
  then have fd2'-def[simp]: fd2' = fd2 using F2-f by simp
  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
  show ?case (is ∃x. ?STEP x ∧ ?MATCH x)
```

```
    proof −
      let ?s1′ = State F1 H (Frame f (Suc pc) (d # norm-stack Σ2′) # st1)
      have pc-in-range: pc < length (body fd1)
        using rel-fd1-fd2 step-load-ubx-miss(2) by simp
      have ?STEP ?s1′
        using step-load-ubx-miss F1-f rel-fd1-fd2
        by (auto intro!: Sinca.step-load[OF F1-f pc-in-range, of x]
            dest!: cast-inversions(1)
            simp: rel-fundef-body-nth Σ1-def)
      moreover have ?MATCH ?s1′
      proof (rule match.intros)
        show rel-fundefs (Finca-get F1) (Fubx-get F2′)
          unfolding step-load-ubx-miss.hyps(7)[symmetric]
          using rel-fundefs-generalize[OF rel-F1-F2 F2-f]
          by simp
      next
        show sp-fundefs (Fubx-get F2′)
          unfolding step-load-ubx-miss.hyps(7)[symmetric]
          using sp-fundefs-generalize[OF sp-F2 F2-f]
          by simp
      next
        show rel-stacktraces (Fubx-get F2′)
          (Frame f (Suc pc) (d # norm-stack Σ2′) # st1)
          (Subx.box-stack f (Frame f (Suc pc) (OpDyn d # Σ2′) # st2)) None
          using step-load-ubx-miss sp-fundef-prefix
              by (auto intro!: rel-stacktraces-generalize[OF rel-st1-st2 F2-f] dest:
cast-inversions)
      qed
      ultimately show ?thesis by blast
    qed
  next
    case (step-store fd2′ x i i′ y d H′ Σ2′)
    then have [simp]: fd2′ = fd2 using F2-f by simp
     obtain fd1 where fd1-thms: Finca-get F1 f = Some fd1 rel-fundef norm-eq
fd1 fd2
      using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1′ = State F1 H′ (Frame f (Suc pc) (norm-stack Σ2′) # st1)
      have ?STEP ?s1′
        unfolding Σ1-def
        using step-store fd1-thms
        by (auto intro!: Sinca.step-store dest!: cast-inversions
            simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1′
        using step-store rel-stacktraces-Cons rel-F1-F2 sp-F2
        by (auto intro!: match.intros rel-stacktraces.intros dest!: cast-inversions
            simp: take-Suc-conv-app-nth)
      ultimately show ?thesis by blast
```

```
    qed
  next
    case (step-store-ubx fd2' τ x i i' blob d H' Σ2')
    then have [simp]: fd2' = fd2 using F2-f by simp
    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
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F1 H' (Frame f (Suc pc) (norm-stack Σ2') # st1)
      have pc-in-range: pc < length (body fd1)
        using rel-fd1-fd2 step-store-ubx.hyps(2) by auto
      have ?STEP ?s1'
        unfolding Σ1-def ‹i # blob # Σ2' = Σ2›[symmetric]
        using step-store-ubx pc-in-range
        by (auto intro!: Sinca.step-store[OF F1-f]
            dest!: cast-inversions
            simp: rel-fundef-body-nth[OF rel-fd1-fd2])
      moreover have ?MATCH ?s1'
        using step-store-ubx rel-stacktraces-Cons rel-F1-F2 sp-F2
        by (auto intro!: match.intros rel-stacktraces.intros
            dest!: cast-inversions
            simp: take-Suc-conv-app-nth)
      ultimately show ?thesis by blast
    qed
  next
    case (step-op fd2' op ar Σ2' x)
    then have [simp]: fd2' = fd2 using F2-f by simp
     obtain fd1 where fd1-thms: Finca-get F1 f = Some fd1 rel-fundef norm-eq
fd1 fd2
      using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
     let ?s1' = State F1 H (Frame f (Suc pc) (x # drop ar (norm-stack Σ2)) #
st1)
      have ?STEP ?s1'
        unfolding Σ1-def
        using step-op fd1-thms take-norm-stack traverse-cast-Dyn-to-norm
        by (auto intro!: Sinca.step-op simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1'
        using step-op rel-stacktraces-Cons rel-F1-F2 sp-F2
        by (auto intro!: match.intros rel-stacktraces.intros
          simp: take-Suc-conv-app-nth drop-norm-stack Let-def take-map drop-map
            traverse-cast-Dyn-replicate)
      ultimately show ?thesis by blast
    qed
  next
    case (step-op-inl fd2' op ar Σ2' opinl x F2')
```

```
    then have [simp]: fd2' = fd2 using F2-f by simp
    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
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?F1' = Finca-add F1 f (rewrite-fundef-body fd1 pc (Inca.IOpInl opinl))
      let ?s1' = State ?F1' H (Frame f (Suc pc) (x # drop ar (norm-stack Σ2))
# st1)
      have ?STEP ?s1'
        unfolding Σ1-def
       using step-op-inl F1-f rel-fd1-fd2 take-norm-stack traverse-cast-Dyn-to-norm
        by (auto intro!: Sinca.step-op-inl simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1'
      proof
        show rel-fundefs (Finca-get ?F1') (Fubx-get F2')
          using step-op-inl.hyps(9)
          using rel-fundefs-rewrite-both[OF rel-F1-F2]
          using rel-fundef-rewrite-both[OF rel-fd1-fd2]
          by auto
      next
        show sp-fundefs (Fubx-get F2')
          using step-op-inl.hyps all-same-arities-add sp-fundefs-get[OF sp-F2]
          using Sinca.𝔍𝔫𝔩-invertible
          by (auto intro!: sp-fundefs-add[OF sp-F2] Subx.sp-rewrite-eq-Ok
              simp: Subx.sp-instr-op Let-def)
      next
        have sp-F2-F2': Subx.sp (Fubx-get F2) = Subx.sp (Fubx-get F2')
          using step-op-inl.hyps(1,9)
          by (auto intro!: sp-same-arities all-same-arities-add)

        let ?fd2' = rewrite-fundef-body fd2' pc (Ubx.IOpInl opinl)

        show rel-stacktraces (Fubx-get F2')
          (Frame f (Suc pc) (x # drop ar (norm-stack Σ2)) # st1)
          (Frame f (Suc pc) (OpDyn x # drop ar Σ2) # st2) None
          using step-op-inl
        proof (intro rel-stacktraces.intros)
          show rel-stacktraces (Fubx-get F2') st1 st2 (Some f)
            using step-op-inl Sinca.𝔍𝔫𝔩-invertible
              by (auto intro!: rel-stacktraces-rewrite-fundef[OF rel-st1-st2] simp:
Subx.sp-instr-op)
        next
          show sp-fundef (Fubx-get F2') ?fd2' (take (Suc pc) (body ?fd2')) =
            Ok (map typeof (OpDyn x # drop ar Σ2))
            using step-op-inl Sinca.𝔍𝔫𝔩-invertible sp-prefix
            using traverse-cast-Dyn-to-all-Dyn
         by (auto simp: take-Suc-conv-app-nth[of pc] sp-F2-F2'[symmetric] Let-def
```

```
                take-map drop-map traverse-cast-Dyn-replicate)
        qed (auto simp add: drop-norm-stack)
       qed
       ultimately show ?thesis by blast
     qed
   next
     case (step-op-inl-hit fd2' opinl ar Σ2' x)
     then have [simp]: fd2' = fd2 using F2-f by simp
     obtain fd1 where fd1-thms: Finca-get F1 f = Some fd1 rel-fundef norm-eq
fd1 fd2
       using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto
     show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
     proof −
       let ?s1' = State F1 H (Frame f (Suc pc) (x # drop ar (norm-stack Σ2)) #
st1)
       have ?STEP ?s1'
         unfolding Σ1-def
         using step-op-inl-hit fd1-thms take-norm-stack traverse-cast-Dyn-to-norm
         by (auto intro!: Sinca.step-op-inl-hit simp: rel-fundef-body-nth)
       moreover have ?MATCH ?s1'
         using step-op-inl-hit rel-stacktraces-Cons rel-F1-F2 sp-F2
      apply (auto intro!: match.intros rel-stacktraces.intros simp: take-Suc-conv-app-nth)
         using drop-norm-stack apply blast
         by (simp add: Let-def drop-map take-map traverse-cast-Dyn-replicate)
       ultimately show ?thesis by blast
     qed
   next
     case (step-op-inl-miss fd2' opinl ar Σ2' x F2')
     then have [simp]: fd2' = fd2 using F2-f by simp
     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
     show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
     proof −
       let ?F1' = Finca-add F1 f (rewrite-fundef-body fd1 pc (Inca.IOp (𝔇𝔢𝔍𝔫𝔩
opinl)))
       let ?s1' = State ?F1' H (Frame f (Suc pc) (x # drop ar (norm-stack Σ2))
# st1)
       have ?STEP ?s1'
         unfolding Σ1-def
      using step-op-inl-miss F1-f rel-fd1-fd2 take-norm-stack traverse-cast-Dyn-to-norm
         by (auto intro!: Sinca.step-op-inl-miss simp: rel-fundef-body-nth)
       moreover have ?MATCH ?s1'
       proof
         show rel-fundefs (Finca-get ?F1') (Fubx-get F2')
           using step-op-inl-miss
           using rel-fundefs-rewrite-both[OF rel-F1-F2]
           using rel-fundef-rewrite-both[OF rel-fd1-fd2]
```

```
        by auto
      next
        show sp-fundefs (Fubx-get F2')
         using step-op-inl-miss.hyps all-same-arities-add sp-fundefs-get[OF sp-F2]

        by (auto intro!: sp-fundefs-add[OF sp-F2] Subx.sp-rewrite-eq-Ok
            simp: Subx.sp-instr-op[symmetric])
      next
        have sp-F2-F2': Subx.sp (Fubx-get F2) = Subx.sp (Fubx-get F2')
          using step-op-inl-miss
          by (auto intro!: sp-same-arities all-same-arities-add)

        let ?fd2' = rewrite-fundef-body fd2' pc (Ubx.IOp (𝔇𝔢𝔍𝔫𝔩 opinl))

        show rel-stacktraces (Fubx-get F2')
          (Frame f (Suc pc) (x # drop ar (norm-stack Σ2)) # st1)
          (Frame f (Suc pc) (OpDyn x # drop ar Σ2) # st2) None
        proof
          show rel-stacktraces (Fubx-get F2') st1 st2 (Some f)
            using step-op-inl-miss.hyps
              by (auto intro: rel-stacktraces-rewrite-fundef[OF rel-st1-st2] simp:
Subx.sp-instr-op[symmetric])
        next
          show Fubx-get F2' f = Some ?fd2'
            using step-op-inl-miss Subx.Fenv.get-add-eq by blast
        next
          show sp-fundef (Fubx-get F2') ?fd2' (take (Suc pc) (body ?fd2')) =
            Ok (map typeof (OpDyn x # drop ar Σ2))
            using ‹pc < length (body fd2')›
            using sp-prefix step-op-inl-miss traverse-cast-Dyn-to-all-Dyn
            by (auto simp add: take-Suc-conv-app-nth[of pc] sp-F2-F2'[symmetric]
                traverse-cast-Dyn-replicate take-map drop-map)
        qed (simp-all add: drop-norm-stack)
      qed
      ultimately show ?thesis by blast
    qed
  next
    case (step-op-ubx fd2' opubx op ar x)
    then have [simp]: fd2' = fd2 using F2-f by simp
     obtain fd1 where fd1-thms: Finca-get F1 f = Some fd1 rel-fundef norm-eq
fd1 fd2
      using rel-fundefs-Some2[OF rel-F1-F2 F2-f] by auto
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F1 H (Frame f (Suc pc) (Subx.norm-unboxed x # drop ar
(norm-stack Σ2)) # st1)
      have pc < length (body fd1)
        using fd1-thms(2) step-op-ubx.hyps(2) by auto
      hence nth-fd1: body fd1 ! pc = Inca.instr.IOpInl (𝔅𝔬𝔵 opubx)
```

```
        using rel-fundef-body-nth[OF fd1-thms(2)]
        using ‹body fd2' ! pc = IOpUbx opubx›
        by simp
      have IsInl (Box opubx) (take ar (map Subx.norm-unboxed Σ2))
        using step-op-ubx
        by (auto intro: Sinca.Inl-IsInl Subx.UbxOp-to-Inl simp: take-map)
      hence ?STEP ?s1'
        unfolding norm-stack-def Σ1-def
        using step-op-ubx fd1-thms(1) nth-fd1 ‹pc < length (body fd1)›
        using Subx.UbxOp-correct
        by (auto intro!: Sinca.step-op-inl-hit simp: take-map)
      moreover have ?MATCH ?s1'
     using step-op-ubx rel-stacktraces-Cons rel-F1-F2 sp-F2 Subx.TypeOfOp-complete
        by (auto intro!: match.intros rel-stacktraces.intros
          simp: take-Suc-conv-app-nth drop-norm-stack Let-def drop-map take-map
min.absorb2)
      ultimately show ?thesis by blast
    qed
  next
    case (step-cjump-true fd2' n x Σ2')
    then have False
   using F2-f sp-fundefs-get[OF sp-F2, unfolded sp-fundef-def, THEN Subx.sp-no-jump]
nth-mem
      by force
    thus ?case by simp
  next
    case (step-cjump-false fd2' n x Σ2')
    then have False
   using F2-f sp-fundefs-get[OF sp-F2, unfolded sp-fundef-def, THEN Subx.sp-no-jump]
nth-mem
      by force
    thus ?case by simp
  next
    case (step-fun-call f' fd2' pc' g gd2 ar Σ2' frame_g)
    then have [simp]: f' = f pc' = pc fd2' = fd2 using F2-f by auto
     obtain fd1 where 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
    obtain gd1 where Finca-get F1 g = Some gd1 and rel-gd1-gd2: rel-fundef
norm-eq gd1 gd2
      using step-fun-call rel-fundefs-Some2[OF rel-F1-F2] by blast
    have pc-in-range: pc < length (body fd1)
      using ‹pc' < length (body fd2')› rel-fundef-body-length[OF rel-fd1-fd2]
      by simp
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
     let ?s1' = State F1 H (Frame g 0 (take (arity gd1) Σ1) # Frame f pc Σ1 #
st1)
      have ?STEP ?s1'
```

```
      using step-fun-call rel-stacktraces-Cons.hyps(2) pc-in-range
      using ‹Finca-get F1 f = Some fd1› ‹Finca-get F1 g = Some gd1›
      using rel-fundef-body-nth[OF rel-fd1-fd2]
      using rel-fundef-arities[OF rel-gd1-gd2, symmetric]
      by (auto intro!: Sinca.step-fun-call)
    moreover have ?MATCH ?s1'
    proof
      show rel-fundefs (Finca-get F1) (Fubx-get F2)
        using rel-F1-F2 .
    next
      show sp-fundefs (Fubx-get F2)
        using sp-F2 .
    next
      show rel-stacktraces (Fubx-get F2)
        (Frame g 0 (take (arity gd1) Σ1) # Frame f pc Σ1 # st1)
        (frame_g # Frame f pc Σ2 # st2) None
        unfolding step-fun-call(9)
      proof (rule rel-stacktraces.intros(2))
        show rel-stacktraces (Fubx-get F2) (Frame f pc Σ1 # st1) (Frame f pc
Σ2 # st2) (Some g)
          using rel-st1-st2 F2-f Σ1-def sp-prefix
          using step-fun-call
          by (auto intro!: rel-stacktraces.intros
              elim!: list.pred-mono-strong simp: is-valid-fun-call-def)
      next
        show take (arity gd1) Σ1 = norm-stack (take ar Σ2')
          using Σ1-def rel-fundef-arities[OF rel-gd1-gd2]
          using step-fun-call by (simp add: take-norm-stack)
      next
        show Fubx-get F2 g = Some gd2
          using step-fun-call by simp
      next
        show sp-fundef (Fubx-get F2) gd2 (take 0 (body gd2)) = Ok (map typeof
(take ar Σ2'))
          using step-fun-call
          by (auto elim: replicate-eq-map)
      qed simp-all
    qed
    ultimately show ?thesis by blast
  qed
 next
  case (step-fun-end f' fd2' Σ2_g pc' Σ2' frame_g g pc_g frame_g' st2')
  then have [simp]: f' = f fd2' = fd2 pc' = pc Σ2' = Σ2 using F2-f by auto
   obtain fd1 where 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
  obtain gd2 Σ1_g st1' where
    st1-def: st1 = Frame g pc_g Σ1_g # st1' and
    Σ1_g = norm-stack Σ2_g and
```

rel-st1′-st2′: *rel-stacktraces* (*Fubx-get F2*) *st1′ st2′* (*Some g*) **and**
Fubx-get F2 g = *Some gd2* **and**
sp-prefix-gd2: *Subx.sp* (*Fubx-get F2*) (*take* pc_g (*body gd2*))
(*replicate* (*arity gd2*) *None*) = *Ok* (*map typeof* $\Sigma 2_g$) **and**
pc_g*-in-range*: pc_g < *length* (*body gd2*) **and**
body gd2 ! pc_g = *Ubx.ICall f* **and**
prefix-all-dyn-$\Sigma 2_g$: *list-all is-dyn-operand* (*take* (*arity fd2*) $\Sigma 2_g$)
using *rel-st1-st2 step-fun-end*
by (*auto elim*: *rel-stacktraces.cases simp*: *is-valid-fun-call-def*)
have *pc-at-end*: *pc* = *length* (*body fd1*)
using ‹*pc′* = *length* (*body fd2′*)›
using *rel-fundef-body-length*[*OF rel-fd1-fd2*]
by *simp*
show *?case* (**is** ∃ *x*. *?STEP x* ∧ *?MATCH x*)
proof −
let *?s1′* = *State F1 H* (*Frame g* (*Suc* pc_g) (Σ*1* @ *drop* (*arity fd1*) $\Sigma 1_g$) # *st1′*)
have *?STEP ?s1′*
using *step-fun-end.hyps*(*2*)
using *st1-def* ‹$\Sigma 1_g$ = *norm-stack* $\Sigma 2_g$›
using ‹*Finca-get F1 f* = *Some fd1*› *pc-at-end*
using *rel-fundef-arities*[*OF rel-fd1-fd2*, *symmetric*]
by (*auto intro*!: *Sinca.step-fun-end*)
moreover have *?MATCH ?s1′*
using *rel-F1-F2 sp-F2*
proof
show *rel-stacktraces* (*Fubx-get F2*)
(*Frame g* (*Suc* pc_g) (Σ*1* @ *drop* (*arity fd1*) $\Sigma 1_g$) # *st1′*)
($frame_g$′ # *st2′*) *None*
unfolding *step-fun-end*(*6*)
using *rel-st1′-st2′*
proof (*rule rel-stacktraces.rel-stacktraces-Cons*)
show Σ*1* @ *drop* (*arity fd1*) $\Sigma 1_g$ = *norm-stack* (Σ*2′* @ *drop* (*arity fd2′*) $\Sigma 2_g$)
using Σ*1-def* ‹$\Sigma 1_g$ = *norm-stack* $\Sigma 2_g$›
using *rel-fundef-arities*[*OF rel-fd1-fd2*]
by (*simp add*: *norm-stack-append drop-norm-stack*)
next
show *Fubx-get F2 g* = *Some gd2*
using ‹*Fubx-get F2 g* = *Some gd2*› **.**
next
have *arity fd2* ≤ *length* $\Sigma 2_g$
using *step-fun-end.hyps*(*2*) **by** *simp*
moreover have *list-all* (λ*x*. *x* = *None*) (*take* (*arity fd2*) (*map typeof* $\Sigma 2_g$))
using *prefix-all-dyn-*$\Sigma 2_g$
by (*auto elim*!: *list.pred-mono-strong simp*: *take-map list.pred-map is-dyn-operand-def*)
moreover have *map typeof* Σ*2* = [*None*]

```
            using ‹pc' = length (body fd2')› sp-prefix
            using sp-fundefs-get[OF sp-F2 F2-f]
            by simp
            ultimately show sp-fundef (Fubx-get F2) gd2 (take (Suc pc_g) (body
gd2)) =
            Ok (map typeof (Σ2' @ drop (arity fd2') Σ2_g))
            using sp-prefix-gd2
            using ‹body gd2 ! pc_g = Ubx.ICall f› F2-f
            by (auto dest: list-all-eq-const-imp-replicate
                simp: take-Suc-conv-app-nth[OF pc_g-in-range] Let-def drop-map)
          qed simp
        qed
        ultimately show ?thesis by blast
      qed
    qed
  qed
qed

lemma match-final-backward:
  s1 ∼ s2 ⟹ Subx.final s2 ⟹ Sinca.final s1
proof (induction s1 s2 rule: match.induct)
  case (1 F1 F2 st1 st2 H)
  obtain f fd2 pc Σ2 where
    st2-def: st2 = [Frame f pc Σ2] and Fubx-get F2 f = Some fd2 and pc = length
(body fd2)
    using ‹Subx.final (State F2 H st2)›
    by (auto elim!: Subx.final.cases)
  obtain Σ1 where st1-def: st1 = [Frame f pc Σ1]
    using ‹rel-stacktraces (Fubx-get F2) st1 st2 None›
    unfolding st2-def
    by (auto elim!: rel-stacktraces.cases)
  obtain fd1 where Finca-get F1 f = Some fd1 and rel-fd1-fd2: rel-fundef norm-eq
fd1 fd2
    using ‹rel-fundefs (Finca-get F1) (Fubx-get F2)› ‹Fubx-get F2 f = Some fd2›
    using rel-fundefs-Some2
    by fastforce
  have length (body fd1) = length (body fd2)
    using rel-fd1-fd2 by simp
  thus ?case
    unfolding st1-def
    using ‹Finca-get F1 f = Some fd1› ‹pc = length (body fd2)›
    by (auto intro: Sinca.final.intros)
qed

sublocale inca-to-ubx-simulation:
   backward-simulation Sinca.step Subx.step Sinca.final Subx.final λ- -. False λ-.
match
  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 *traverse-cast-Dyn-eq-norm-stack*:
assumes *list-all* (λ*x*. *x* = *None*) (*map typeof xs*)
shows *traverse cast-Dyn xs* = *Some* (*norm-stack xs*)
using *assms*
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons.prems* **have**
typeof-x: *typeof x* = *None* **and**
typeof-xs: *list-all* (λ*x*. *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 *Sinca.step s1 s1′* **and** *s1* ∼ *s2*
shows ∃ *s2′*. *Subx.step s2 s2′* ∧ *s1′* ∼ *s2′*
using *assms*(*2*,*1*)
proof (*induction s1 s2 rule*: *match.induct*)
case (*1 F1 F2 st1 st2 H*)
have *rel-F1-F2*: *rel-fundefs* (*Finca-get F1*) (*Fubx-get F2*) **using** *1* **by** *simp*
have *sp-F2*: *sp-fundefs* (*Fubx-get F2*) **using** *1* **by** *simp*

note *rel-fundefs-rewrite-both′* =
rel-fundef-rewrite-both[*THEN rel-fundefs-rewrite-both*[*OF rel-F1-F2*]]

from *1*(*3*,*4*) **show** *?case*
proof (*induction st1 st2 None* :: *'fun option rule*: *rel-stacktraces.induct*)
case *rel-stacktraces-Nil*
hence *False* **by** (*auto elim*: *Sinca.step.cases*)
thus *?case* **by** *simp*
next
case (*rel-stacktraces-Cons st1 st2 f* Σ*1* Σ*2 fd2 pc*)
have *F2-f*: *Fubx-get F2 f* = *Some fd2*
using *rel-stacktraces-Cons* **by** *simp*
have *rel-st1-st2*: *rel-stacktraces* (*Fubx-get F2*) *st1 st2* (*Some f*)
using *rel-stacktraces-Cons* **by** *simp*
have *sp-fundef-prefix*: *sp-fundef* (*Fubx-get F2*) *fd2* (*take pc* (*body fd2*)) = *Ok* (*map typeof* Σ*2*)

```
  using rel-stacktraces-Cons by simp
have Σ1-def: Σ1 = norm-stack Σ2
  using rel-stacktraces-Cons by simp

note sp-fundef-def[simp]
note sp-prefix = sp-fundef-prefix[unfolded sp-fundef-def]

note ex-F1-f = rel-fundefs-Some2[OF rel-F1-F2 F2-f]
note sp-fundef-full = sp-fundefs-get[OF sp-F2 ‹Fubx-get F2 f = Some fd2›]
note sp-full = sp-fundef-full[unfolded sp-fundef-def]
hence sp-sufix: Subx.sp (Fubx-get F2)
  (body fd2 ! pc # drop (Suc pc) (body fd2)) (map typeof Σ2) = Ok [None]
  if pc-in-range: pc < length (body fd2)
  unfolding Cons-nth-drop-Suc[OF pc-in-range]
  unfolding Subx.sp-eq-bind-take-drop[of - body fd2 - pc]
  unfolding sp-prefix
  by simp

from rel-stacktraces-Cons.prems(1) show ?case
proof (induction State F1 H (Frame f pc Σ1 # st1) s1' rule: Sinca.step.induct)
  case (step-push fd1 d)
  hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
    using ex-F1-f by simp
  hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
    using step-push rel-fundef-body-nth by metis
  have pc-in-range: pc < length (body fd2)
    using step-push rel-fundef-body-length[OF rel-fd1-fd2] by simp

  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    using step-push norm-instr-nth-body
  proof (cases body fd2 ! pc)
    case (IPush d2)
    then have d-def: d = d2
      using norm-instr-nth-body step-push.hyps(3) by auto
    let ?s2' = State F2 H (Frame f (Suc pc) (OpDyn d2 # Σ2) # st2)
    have ?STEP ?s2'
      using IPush step-push d-def F2-f pc-in-range
      by (auto intro: Subx.step-push)
    moreover have ?MATCH ?s2'
      using rel-F1-F2 sp-F2 rel-st1-st2 F2-f Σ1-def
      using IPush d-def sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
      by (auto intro!: match.intros rel-stacktraces.intros)
    ultimately show ?thesis by auto
  next
    case (IPushUbx1 n)
    then have d-def: d = box-ubx1 n
      using norm-instr-nth-body step-push.hyps(3) by auto
    let ?s2' = State F2 H (Frame f (Suc pc) (OpUbx1 n # Σ2) # st2)
    have ?STEP ?s2'
```

```
      using IPushUbx1 step-push d-def F2-f pc-in-range
      by (auto intro: Subx.step-push-ubx1)
    moreover have ?MATCH ?s2'
      using rel-F1-F2 sp-F2 rel-st1-st2 F2-f Σ1-def
      using IPushUbx1 d-def sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
      by (auto intro!: match.intros rel-stacktraces.intros)
    ultimately show ?thesis by auto
  next
    case (IPushUbx2 b)
    then have d-def: d = box-ubx2 b
      using norm-instr-nth-body step-push.hyps(3) by auto
    let ?s2' = State F2 H (Frame f (Suc pc) (OpUbx2 b # Σ2) # st2)
    have ?STEP ?s2'
      using IPushUbx2 step-push d-def F2-f pc-in-range
      by (auto intro: Subx.step-push-ubx2)
    moreover have ?MATCH ?s2'
      using rel-F1-F2 sp-F2 rel-st1-st2 F2-f Σ1-def
      using IPushUbx2 d-def sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
      by (auto intro!: match.intros rel-stacktraces.intros)
    ultimately show ?thesis by auto
  qed simp-all
next
  case (step-pop fd1 x Σ1')
  hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
    using ex-F1-f by simp
  hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
    using step-pop rel-fundef-body-nth by metis
  have pc-in-range: pc < length (body fd2)
    using step-pop rel-fundef-body-length[OF rel-fd1-fd2] by simp
  obtain x' Σ2' where Σ2-def: Σ2 = x' # Σ2' and Σ1' = norm-stack Σ2'
    using step-pop ‹Σ1 = norm-stack Σ2› norm-stack-def by auto

  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    using step-pop norm-instr-nth-body
  proof (cases body fd2 ! pc)
    case IPop
    let ?s2' = State F2 H (Frame f (Suc pc) Σ2' # st2)
    have ?STEP ?s2'
      using IPop step-pop F2-f pc-in-range Σ2-def
      by (auto intro: Subx.step-pop)
    moreover have ?MATCH ?s2'
      using rel-F1-F2 sp-F2 rel-st1-st2 F2-f ‹Σ1' = norm-stack Σ2'›
      using IPop sp-prefix take-Suc-conv-app-nth[OF pc-in-range] Σ2-def
      by (auto intro!: match.intros rel-stacktraces.intros)
    ultimately show ?thesis by auto
  qed simp-all
next
  case (step-load fd1 var i d Σ1')
  hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
```

```
    using ex-F1-f by simp
  hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
    using step-load rel-fundef-body-nth by metis
  have pc-in-range: pc < length (body fd2)
    using step-load rel-fundef-body-length[OF rel-fd1-fd2] by simp
  obtain i' Σ2' where
      Σ2-def: Σ2 = i' # Σ2' and norm-i': Subx.norm-unboxed i' = i and
norm-stack Σ2' = Σ1'
    using step-load Σ1-def norm-stack-def by auto

  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    using step-load norm-instr-nth-body
  proof (cases body fd2 ! pc)
    case (ILoad var')
    then have [simp]: var' = var
      using norm-instr-nth-body step-load.hyps(3) by auto
    let ?s2' = State F2 H (Frame f (Suc pc) (OpDyn d # Σ2') # st2)
    have i' = OpDyn i
      using sp-sufix[OF pc-in-range, unfolded Σ2-def ILoad, simplified]
      using norm-i'
      by (cases i'; simp)
    hence ?STEP ?s2'
      using ILoad step-load F2-f pc-in-range
      by (auto intro!: Subx.step-load simp: Σ2-def)
    moreover have ?MATCH ?s2'
      using rel-F1-F2 sp-F2 rel-st1-st2 F2-f Σ1-def
      using ILoad sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
      by (auto intro!: match.intros rel-stacktraces.intros
          simp: ‹norm-stack Σ2' = Σ1'› ‹i' = OpDyn i› Σ2-def)
    ultimately show ?thesis by auto
  next
    case (ILoadUbx τ var')
    then have [simp]: var' = var
      using norm-instr-nth-body step-load.hyps(3) by auto
    have [simp]: i' = OpDyn i
      using sp-sufix[OF pc-in-range, unfolded Σ2-def ILoadUbx, simplified]
      using norm-i'
      by (cases i'; simp)
    show ?thesis
    proof (cases Subx.unbox τ d)
      case None
      let ?F2' = Fubx-add F2 f (Subx.generalize-fundef fd2)
      let ?frame = Frame f (Suc pc) (OpDyn d # Σ2')
      let ?s2' = State ?F2' H (Subx.box-stack f (?frame # st2))
      have ?STEP ?s2'
        using None ILoadUbx
        using step-load pc-in-range F2-f
        by (auto intro!: Subx.step-load-ubx-miss
            simp add: Σ2-def
```

```
        simp del: Subx.box-stack-Cons)
    moreover have ?MATCH ?s2'
      using rel-fundefs-generalize[OF rel-F1-F2 F2-f]
      using sp-fundefs-generalize[OF sp-F2 F2-f]
      using sp-fundef-prefix ILoadUbx
      using pc-in-range
    by (auto intro!: match.intros rel-stacktraces-generalize[OF rel-st1-st2 F2-f]
        simp: Σ2-def ‹norm-stack Σ2' = Σ1'›)
    ultimately show ?thesis by auto
  next
    case (Some blob)
    let ?s2' = State F2 H (Frame f (Suc pc) (blob # Σ2') # st2)
    have ?STEP ?s2'
      using Some ILoadUbx step-load F2-f pc-in-range
      by (auto intro: Subx.step-load-ubx-hit simp: Σ2-def)
    moreover have ?MATCH ?s2'
      using rel-F1-F2 sp-F2 rel-st1-st2 F2-f Σ1-def
      using Some ILoadUbx sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
      by (auto intro!: match.intros rel-stacktraces.intros
        simp: ‹norm-stack Σ2' = Σ1'› Σ2-def)
    ultimately show ?thesis by auto
  qed
 qed simp-all
next
case (step-store fd1 var i x H' Σ1')
 hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
   using ex-F1-f by simp
 hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
   using step-store rel-fundef-body-nth by metis
 have pc-in-range: pc < length (body fd2)
   using step-store rel-fundef-body-length[OF rel-fd1-fd2] by simp
 obtain i' x' Σ2' where
   Σ2-def: Σ2 = i' # x' # Σ2' and
   norm-i': Subx.norm-unboxed i' = i and
   norm-x': Subx.norm-unboxed x' = x and
   norm-stack Σ2' = Σ1'
   using step-store Σ1-def norm-stack-def by auto

 show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
   using step-store norm-instr-nth-body
 proof (cases body fd2 ! pc)
   case (IStore var')
   then have [simp]: var' = var
     using norm-instr-nth-body step-store.hyps(3) by auto
  note sp-sufix' = sp-sufix[OF pc-in-range, unfolded Σ2-def IStore, simplified]
   have [simp]: i' = OpDyn i
     using norm-i' sp-sufix' by (cases i'; simp)
   have [simp]: x' = OpDyn x
     using norm-x' sp-sufix' by (cases x'; simp)
```

```
      let ?s2' = State F2 (heap-add H (var, i) x) (Frame f (Suc pc) Σ2' # st2)
      have ?STEP ?s2'
        using IStore Σ2-def pc-in-range F2-f
        by (auto intro: Subx.step-store)
      moreover have ?MATCH ?s2'
        using rel-F1-F2 sp-F2 rel-st1-st2 F2-f Σ2-def
        using ‹norm-stack Σ2' = Σ1'› ‹heap-add H (var, i) x = H'›
        using IStore sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
        by (auto intro!: match.intros rel-stacktraces.intros)
      ultimately show ?thesis by auto
    next
      case (IStoreUbx τ var')
      then have [simp]: var' = var
        using norm-instr-nth-body step-store.hyps(3) by auto
          note sp-sufix' = sp-sufix[OF pc-in-range, unfolded Σ2-def IStoreUbx,
simplified]
      have [simp]: i' = OpDyn i
        using norm-i' sp-sufix' by (cases i'; simp)
      have typeof-x'[simp]: typeof x' = Some τ
            using sp-sufix' apply (auto simp add: Result.bind-eq-Ok-conv elim!:
Subx.sp-instr.elims)
        by (metis result.simps(4))
      let ?s2' = State F2 (heap-add H (var, i) x) (Frame f (Suc pc) Σ2' # st2)
      have ?STEP ?s2'
        unfolding Σ2-def
        using F2-f pc-in-range IStoreUbx
     using Subx.typeof-and-norm-unboxed-imp-cast-and-box[OF typeof-x' norm-x']
        by (auto intro!: Subx.step-store-ubx)
      moreover have ?MATCH ?s2'
        using rel-F1-F2 sp-F2 rel-st1-st2 F2-f Σ2-def
        using ‹norm-stack Σ2' = Σ1'› ‹heap-add H (var, i) x = H'›
        using IStoreUbx sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
        by (auto intro!: match.intros rel-stacktraces.intros)
      ultimately show ?thesis by auto
    qed simp-all
  next
    case (step-op fd1 op ar x)
    hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
      using ex-F1-f by simp
    hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
      using step-op rel-fundef-body-nth by metis
    have pc-in-range: pc < length (body fd2)
      using step-op rel-fundef-body-length[OF rel-fd1-fd2] by simp

    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
      using step-op norm-instr-nth-body
    proof (cases body fd2 ! pc)
      case (IOp op')
      then have op' = op
```

```
    using norm-instr-nth-body step-op.hyps(3) by auto
  hence ar ≤ length Σ2 and
    all-arg-Dyn: list-all (λx. x = None) (map typeof (take ar Σ2))
    using IOp step-op sp-sufix[OF pc-in-range]
    by (auto simp: take-map)
  let ?s2' = State F2 H (Frame f (Suc pc) (OpDyn x # drop ar Σ2) # st2)
  have ?STEP ?s2'
    using IOp step-op ‹op' = op› Σ1-def pc-in-range F2-f
    using traverse-cast-Dyn-eq-norm-stack[OF all-arg-Dyn]
    by (auto intro!: Subx.step-op simp: take-norm-stack)
  moreover have ?MATCH ?s2'
    using rel-F1-F2 sp-F2 rel-st1-st2 F2-f
    using IOp sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
    using arg-cong[OF Σ1-def, of drop ar] ‹ar ≤ length Σ2›
    using ‹op' = op› all-arg-Dyn Σ1-def ‹𝔄𝔯𝔦𝔱𝔶 op = ar›
    by (auto intro!: match.intros rel-stacktraces.intros
        dest: list-all-eq-const-imp-replicate
        simp: drop-norm-stack[symmetric] Let-def take-map drop-map)
  ultimately show ?thesis by auto
 qed simp-all
next
 case (step-op-inl fd1 op ar opinl x F1')
 hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
   using ex-F1-f by simp
 hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
   using step-op-inl rel-fundef-body-nth by metis
 have pc-in-range: pc < length (body fd2)
   using step-op-inl rel-fundef-body-length[OF rel-fd1-fd2] by simp

 show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
   using step-op-inl norm-instr-nth-body
 proof (cases body fd2 ! pc)
   case (IOp op')
   then have op' = op
     using norm-instr-nth-body step-op-inl.hyps(3) by simp
   hence ar ≤ length Σ2 and
     all-arg-Dyn: list-all (λx. x = None) (map typeof (take ar Σ2))
     using IOp step-op-inl sp-sufix[OF pc-in-range]
     by (auto simp: take-map)
   let ?fd2' = rewrite-fundef-body fd2 pc (Ubx.IOpInl opinl)
   let ?F2' = Fubx-add F2 f ?fd2'
   let ?frame = Frame f (Suc pc) (OpDyn x # drop ar Σ2)
   let ?s2' = State ?F2' H (?frame # st2)
   have ?STEP ?s2'
     using IOp step-op-inl ‹op' = op›
     using Σ1-def pc-in-range ‹Fubx-get F2 f = Some fd2›
     using all-arg-Dyn take-norm-stack traverse-cast-Dyn-eq-norm-stack
     by (auto intro!: Subx.step-op-inl)
   moreover have ?MATCH ?s2'
```

```
      proof
        show rel-fundefs (Finca-get F1') (Fubx-get ?F2')
          using step-op-inl rel-fd1-fd2
          by (auto intro: rel-fundefs-rewrite-both')
      next
        have arity fd2 = arity ?fd2' by simp
        hence all-same-arities (Fubx-get F2) (Fubx-get ?F2')
          using all-same-arities-add[OF F2-f] by simp
        thus sp-fundefs (Fubx-get ?F2')
          apply (auto intro!: sp-fundefs-add[OF sp-F2])
          apply (rule Subx.sp-rewrite-eq-Ok[OF pc-in-range - sp-full])
          apply (rule allI)
          unfolding IOp
          apply (rule Subx.sp-instr-op)
          using Sinca.Inl-invertible ‹op' = op› step-op-inl.hyps(6) by blast
      next
        have rel-stacktraces (Fubx-get F2)
          (Frame f (Suc pc) (x # drop ar Σ1) # st1) (?frame # st2) None
          using rel-st1-st2 sp-prefix F2-f
          using IOp take-Suc-conv-app-nth[OF pc-in-range]
          using arg-cong[OF Σ1-def, of drop ar] ‹ar ≤ length Σ2›
          using ‹op' = op› step-op-inl.hyps(4) all-arg-Dyn
          by (auto intro!: rel-stacktraces.intros
              dest: list-all-eq-const-imp-replicate
              simp: drop-norm-stack[symmetric] Let-def take-map drop-map)
        thus rel-stacktraces (Fubx-get ?F2')
        (Frame f (Suc pc) (x # drop ar Σ1) # st1) (?frame # st2) None
          using F2-f pc-in-range IOp
          using Sinca.Inl-invertible ‹op' = op› step-op-inl.hyps(6)
            by (auto intro!: rel-stacktraces-rewrite-fundef simp: Subx.sp-instr-op
Let-def)
      qed
      ultimately show ?thesis by auto
    qed simp-all
  next
    case (step-op-inl-hit fd1 opinl ar x)
    hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
      using ex-F1-f by simp
    hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
      using step-op-inl-hit rel-fundef-body-nth by metis
    have pc-in-range: pc < length (body fd2)
      using step-op-inl-hit rel-fundef-body-length[OF rel-fd1-fd2] by simp

    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
      using step-op-inl-hit norm-instr-nth-body
    proof (cases body fd2 ! pc)
      case (IOpInl opinl')
      then have opinl' = opinl
        using norm-instr-nth-body step-op-inl-hit.hyps(3) by simp
```

```
      hence ar ≤ length Σ2 and
        all-arg-Dyn: list-all (λx. x = None) (map typeof (take ar Σ2))
        using IOpInl step-op-inl-hit sp-sufix[OF pc-in-range]
        by (auto simp: take-map)

      let ?s2' = State F2 H (Frame f (Suc pc) (OpDyn x # drop ar Σ2) # st2)
      have ?STEP ?s2'
        using IOpInl step-op-inl-hit ‹opinl' = opinl›
        using Σ1-def pc-in-range F2-f all-arg-Dyn
        by (auto intro: Subx.step-op-inl-hit traverse-cast-Dyn-eq-norm-stack
            simp: take-norm-stack)
      moreover have ?MATCH ?s2'
        using rel-F1-F2 sp-F2 rel-st1-st2 F2-f
        using IOpInl sp-prefix take-Suc-conv-app-nth[OF pc-in-range]
        using arg-cong[OF Σ1-def, of drop ar]
      using ‹ar ≤ length Σ2› ‹opinl' = opinl› step-op-inl-hit.hyps(4,5) all-arg-Dyn
        by (auto intro!: match.intros rel-stacktraces.intros
            dest: list-all-eq-const-imp-replicate
            simp: drop-norm-stack[symmetric] Let-def take-map drop-map)
      ultimately show ?thesis by auto
    next
      case (IOpUbx opubx)
      then have 𝔅𝔬𝔵 opubx = opinl
        using norm-instr-nth-body step-op-inl-hit.hyps(3) by simp
      hence 𝔄𝔯𝔦𝔱𝔶-opubx[simp]: 𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 (𝔅𝔬𝔵 opubx)) = ar
        using step-op-inl-hit.hyps(4,5) by simp
        note sp-sufix' = sp-sufix[OF pc-in-range, unfolded IOpUbx, simplified,
unfolded Let-def, simplified]
      obtain dom codom where typeof-opubx: 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx = (dom, codom)
        using prod.exhaust-sel by blast
      hence dom-def: dom = map typeof (take ar Σ2)
        unfolding 𝔄𝔯𝔦𝔱𝔶-opubx[unfolded Subx.𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-𝔄𝔯𝔦𝔱𝔶, symmetric]
        using sp-sufix'
        by (auto simp: Let-def take-map)

      obtain x' where
        eval-op: 𝔘𝔟𝔵𝔒𝔭 opubx (take ar Σ2) = Some x' and
        typeof-x': typeof x' = codom
        using typeof-opubx[unfolded dom-def, THEN Subx.𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-correct]
        by auto

      let ?s2' = State F2 H (Frame f (Suc pc) (x' # drop ar Σ2) # st2)
      have ?STEP ?s2'
        using Σ1-def step-op-inl-hit eval-op 𝔄𝔯𝔦𝔱𝔶-opubx
        using pc-in-range IOpUbx F2-f
        by (auto intro!: Subx.step-op-ubx)
      moreover have ?MATCH ?s2'
      proof −
        have x = Subx.norm-unboxed x'
```

```
      using Subx.𝔘𝔟𝔵𝔒𝔭-correct[OF eval-op, unfolded ‹𝔅𝔬𝔵 opubx = opinl›]
      using step-op-inl-hit Σ1-def
      by (simp add: take-map norm-stack-def)
    thus ?thesis
      using rel-F1-F2 sp-F2 rel-st1-st2 F2-f Σ1-def
     using sp-prefix take-Suc-conv-app-nth[OF pc-in-range, unfolded IOpUbx]
      using step-op-inl-hit
      using typeof-opubx typeof-x'
      by (auto intro!: match.intros rel-stacktraces.intros
       simp: dom-def drop-norm-stack Let-def min.absorb2 take-map drop-map)
  qed
  ultimately show ?thesis by auto
qed simp-all
next
  case (step-op-inl-miss fd1 opinl ar x F1')
  hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
    using ex-F1-f by simp
  hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
    using step-op-inl-miss rel-fundef-body-nth by metis
  have pc-in-range: pc < length (body fd2)
    using step-op-inl-miss rel-fundef-body-length[OF rel-fd1-fd2] by simp

  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    using step-op-inl-miss norm-instr-nth-body
  proof (cases body fd2 ! pc)
    case (IOpInl opinl')
    then have opinl' = opinl
      using norm-instr-nth-body step-op-inl-miss.hyps(3) by simp
    hence ar ≤ length Σ2 and
      all-arg-Dyn: list-all (λx. x = None) (map typeof (take ar Σ2))
      using IOpInl step-op-inl-miss sp-sufix[OF pc-in-range]
      by (auto simp: take-map)

    let ?fd2' = rewrite-fundef-body fd2 pc (Ubx.IOp (𝔇𝔢𝔍𝔫𝔩 opinl))
    let ?F2' = Fubx-add F2 f ?fd2'
    let ?frame = Frame f (Suc pc) (OpDyn x # drop ar Σ2)
    let ?s2' = State ?F2' H (?frame # st2)
    have ?STEP ?s2'
      using IOpInl step-op-inl-miss ‹opinl' = opinl›
      using Σ1-def pc-in-range F2-f
      using traverse-cast-Dyn-eq-norm-stack[OF all-arg-Dyn]
      by (auto intro!: Subx.step-op-inl-miss simp: take-norm-stack)
    moreover have ?MATCH ?s2'
    proof
      show rel-fundefs (Finca-get F1') (Fubx-get ?F2')
        using step-op-inl-miss rel-fd1-fd2
        by (auto intro: rel-fundefs-rewrite-both')
    next
      show sp-fundefs (Fubx-get ?F2')
```

```
      using IOpInl ‹opinl' = opinl› all-same-arities-add[OF F2-f] sp-full
      by (auto intro: sp-fundefs-add[OF sp-F2]
          simp: Subx.sp-instr-op[symmetric] Subx.sp-rewrite[OF pc-in-range])
  next
    have rel-stacktraces (Fubx-get F2)
      (Frame f (Suc pc) (x # drop ar Σ1) # st1) (?frame # st2) None
      using rel-st1-st2 sp-prefix F2-f
      using IOpInl Σ1-def take-Suc-conv-app-nth[OF pc-in-range]
      using ‹ar ≤ length Σ2› ‹opinl' = opinl› step-op-inl-miss.hyps(4,5)
      using all-arg-Dyn
      by (auto intro!: rel-stacktraces.intros
          dest: list-all-eq-const-imp-replicate
          simp: drop-norm-stack Let-def take-map drop-map)

    thus rel-stacktraces (Fubx-get ?F2')
      (Frame f (Suc pc) (x # drop ar Σ1) # st1) (?frame # st2) None
      using F2-f pc-in-range IOpInl
      by (auto intro: rel-stacktraces-rewrite-fundef
          simp: ‹opinl' = opinl› Let-def Subx.sp-instr-op[symmetric])
  qed
  ultimately show ?thesis by auto
next
  case (IOpUbx opubx)
  then have 𝔅𝔬𝔵 opubx = opinl
    using norm-instr-nth-body step-op-inl-miss.hyps(3) by simp
  hence 𝔄𝔯𝔦𝔱𝔶-opubx: 𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 (𝔅𝔬𝔵 opubx)) = ar
    using step-op-inl-miss.hyps(4,5) by simp
  obtain dom codom where 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx = (dom, codom)
    using prod.exhaust-sel by blast
  hence 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx = (map typeof (take ar Σ2), codom)
    unfolding 𝔄𝔯𝔦𝔱𝔶-opubx[unfolded Subx.𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-𝔄𝔯𝔦𝔱𝔶, symmetric]
    using sp-sufix[OF pc-in-range]
    unfolding IOpUbx
    by (auto simp: Let-def case-prod-beta take-map)
  then obtain x where eval-op: 𝔘𝔟𝔵𝔒𝔭 opubx (take ar Σ2) = Some x
    using Subx.𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-correct by blast
  hence 𝔍𝔰𝔍𝔫𝔩 opinl (take ar Σ1)
    unfolding Σ1-def norm-stack-def
    using ‹𝔅𝔬𝔵 opubx = opinl›
    by (auto intro: Sinca.𝔍𝔫𝔩-𝔍𝔰𝔍𝔫𝔩 Subx.𝔘𝔟𝔵𝔒𝔭-to-𝔍𝔫𝔩 simp: take-map)
  hence False
    using step-op-inl-miss by auto
  thus ?thesis by simp
qed simp-all
next
  case (step-cjump-true fd1 n Σ1')
  hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
    using ex-F1-f by simp
  hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
```

```
      using step-cjump-true rel-fundef-body-nth by metis
    have pc-in-range: pc < length (body fd2)
      using step-cjump-true rel-fundef-body-length[OF rel-fd1-fd2] by simp

    show ?case
      using step-cjump-true norm-instr-nth-body
    proof (cases body fd2 ! pc)
      case (ICJump n')
      then have False
          using F2-f sp-fundefs-get[OF sp-F2, unfolded sp-fundef-def, THEN
Subx.sp-no-jump]
        using nth-mem pc-in-range by fastforce
      thus ?thesis by simp
    qed simp-all
  next
    case (step-cjump-false fd1 n Σ1')
    hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
      using ex-F1-f by simp
    hence norm-instr-nth-body: norm-instr (body fd2 ! pc) = body fd1 ! pc
      using step-cjump-false rel-fundef-body-nth by metis
    have pc-in-range: pc < length (body fd2)
      using step-cjump-false rel-fundef-body-length[OF rel-fd1-fd2] by simp

    show ?case
      using step-cjump-false norm-instr-nth-body
    proof (cases body fd2 ! pc)
      case (ICJump n')
      then have False
          using F2-f sp-fundefs-get[OF sp-F2, unfolded sp-fundef-def, THEN
Subx.sp-no-jump]
        using nth-mem pc-in-range by fastforce
      thus ?thesis by simp
    qed simp-all
  next
    case (step-fun-call f' fd1 pc' g gd1 Σ1' frame_g)
    hence [simp]: f' = f pc' = pc Σ1' = Σ1 by auto
    hence rel-fd1-fd2: rel-fundef norm-eq fd1 fd2
      using step-fun-call ex-F1-f by simp
     obtain gd2 where Fubx-get F2 g = Some gd2 and rel-gd1-gd2: rel-fundef
norm-eq gd1 gd2
      using step-fun-call rel-fundefs-Some1[OF rel-F1-F2] by blast
    have pc-in-range: pc < length (body fd2)
      using step-fun-call rel-fd1-fd2 by simp

    have nth-body2: body fd2 ! pc = Ubx.ICall g
     using rel-fundef-body-nth[OF rel-fd1-fd2 ‹pc' < length (body fd1)›, symmetric]
      unfolding ‹body fd1 ! pc' = Inca.ICall g›
      by (cases body fd2 ! pc'; simp)
```

```
    have prefix-Σ2-all-Dyn: list-all (λx. x = None) (map typeof (take (arity gd2)
Σ2))
      using ‹Fubx-get F2 g = Some gd2› sp-sufix[OF pc-in-range] nth-body2
      by (auto simp: Let-def take-map)
    hence all-dyn-args: list-all is-dyn-operand (take (arity gd2) Σ2)
      by (auto elim: list.pred-mono-strong simp add: list.pred-map comp-def)

    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?args = map OpDyn (norm-stack (take (arity gd2) Σ2))
      let ?frame = Frame g 0 ?args
      let ?s1' = State F2 H (?frame # Frame f pc Σ2 # st2)
      have ?STEP ?s1'
      proof −
        have arity gd2 ≤ length Σ2
          using ‹arity gd1 ≤ length Σ1'›
          using rel-fundef-arities[OF rel-gd1-gd2] Σ1-def
          by simp
        thus ?thesis
          using pc-in-range nth-body2
          using ‹Fubx-get F2 g = Some gd2› rel-stacktraces-Cons.hyps(3)
          using all-dyn-args
          by (auto intro!: Subx.step-fun-call[of - - fd2] nth-equalityI
              simp: typeof-unboxed-eq-const list-all-length norm-stack-def)
      qed
      moreover have ?MATCH ?s1'
        using rel-F1-F2 sp-F2
      proof
        show rel-stacktraces (Fubx-get F2)
         (frame_g # Frame f pc Σ1 # st1)
         (?frame # Frame f pc Σ2 # st2) None
          unfolding step-fun-call(7)
        proof (rule rel-stacktraces.intros)
          show rel-stacktraces (Fubx-get F2) (Frame f pc Σ1 # st1) (Frame f pc
Σ2 # st2)
            (Some g)
            using pc-in-range nth-body2 rel-stacktraces-Cons
            using ‹Fubx-get F2 g = Some gd2› all-dyn-args
            by (auto intro!: rel-stacktraces.intros simp: is-valid-fun-call-def)
        next
          show take (arity gd1) Σ1' = norm-stack ?args
            using Σ1-def
            using rel-fundef-arities[OF rel-gd1-gd2]
            by (simp add: norm-stack-map take-norm-stack)
        next
          show Fubx-get F2 g = Some gd2
            using ‹Fubx-get F2 g = Some gd2› .
        next
          show sp-fundef (Fubx-get F2) gd2 (take 0 (body gd2)) = Ok (map typeof
```

?args)
            using *Σ1-def step-fun-call.hyps*(*5*)
            using *rel-fundef-arities*[*OF rel-gd1-gd2*]
            by (*simp add*: *map-replicate-const*)
          qed *simp-all*
        qed
        ultimately show *?thesis* **by** *blast*
      qed
    next
      case (*step-fun-end f' fd1 Σ1$_g$ pc' Σ1' frame$_g$ g pc$_g$ frame$_g$' st1'*)
      hence [*simp*]: *f' = f pc' = pc Σ1' = Σ1* **by** *auto*
      hence *rel-fd1-fd2*: *rel-fundef norm-eq fd1 fd2*
        using *step-fun-end ex-F1-f* **by** *simp*

      note *arities-fd1-fd2 = rel-fundef-arities*[*OF rel-fd1-fd2*]

      obtain *Σ2$_g$ st2'* **where**
        st2-def: *st2 = Frame g pc$_g$ Σ2$_g$ # st2'*
        using *step-fun-end rel-st1-st2*
        by (*auto elim*: *rel-stacktraces.cases*)

       hence *all-dyn-prefix-Σ2$_g$*: *list-all* (*λx. x = None*) (*map typeof* (*take* (*arity fd2*) *Σ2$_g$*))
        using *step-fun-end rel-st1-st2 F2-f*
        by (*auto*
            elim!: *rel-stacktraces.cases*[*of - - # -*] *list.pred-mono-strong*
            simp: *list.pred-map is-valid-fun-call-def*)

      show *?case* (**is** *∃ x. ?STEP x ∧ ?MATCH x*)
      proof −
        let *?s1' = State F2 H* (*Frame g* (*Suc pc$_g$*) (*Σ2 @ drop* (*arity fd2*) *Σ2$_g$*) *# st2'*)
        have *?STEP ?s1'*
          unfolding *st2-def*
          using *step-fun-end st2-def rel-st1-st2*
          using *arities-fd1-fd2*
          using *rel-fundef-body-length*[*OF rel-fd1-fd2*]
          by (*auto intro*!: *Subx.step-fun-end*[*OF ‹Fubx-get F2 f = Some fd2›*]
              elim!: *rel-stacktraces.cases*[*of - - # -*])
        moreover have *?MATCH ?s1'*
        proof −
          have *map typeof Σ2 = [None]*
            using *step-fun-end st2-def rel-st1-st2*
            using *rel-fd1-fd2 sp-full sp-prefix*
            by (*auto elim*!: *rel-stacktraces.cases*[*of - - # -*])

          thus *?thesis*
            using *step-fun-end st2-def rel-st1-st2*
            using *arities-fd1-fd2 Σ1-def F2-f all-dyn-prefix-Σ2$_g$*

```
          by (auto intro!: match.intros rel-stacktraces.intros
              intro: rel-F1-F2 sp-F2
              elim!: rel-stacktraces.cases[of - - # -]
              dest: list-all-eq-const-imp-replicate
             simp: take-Suc-conv-app-nth Let-def drop-norm-stack take-map drop-map
              simp: is-valid-fun-call-def)
       qed
       ultimately show ?thesis by auto
     qed
   qed
  qed
qed

lemma match-final-forward:
  s1 ∼ s2 ⟹ Sinca.final s1 ⟹ Subx.final s2
proof (induction s1 s2 rule: match.induct)
  case (1 F1 F2 st1 st2 H)
  obtain f fd1 pc Σ1 where
    st1-def: st1 = [Frame f pc Σ1] and
    F1-f: Finca-get F1 f = Some fd1 and
    pc-def: pc = length (body fd1)
    using ‹Sinca.final (State F1 H st1)›
    by (auto elim: Sinca.final.cases)
  obtain Σ2 where st2-def: st2 = [Frame f pc Σ2]
    using ‹rel-stacktraces (Fubx-get F2) st1 st2 None›
    unfolding st1-def
    by (auto elim: rel-stacktraces.cases)
  obtain fd2 where F2-f: Fubx-get F2 f = Some fd2 and rel-fd1-fd2: rel-fundef
norm-eq fd1 fd2
    using rel-fundefs-Some1[OF ‹rel-fundefs (Finca-get F1) (Fubx-get F2)› F1-f]
    by auto
  have length (body fd1) = length (body fd2)
    using rel-fd1-fd2 by simp
  thus ?case
    unfolding st2-def pc-def
    using F2-f by (auto intro: Subx.final.intros)
qed

sublocale inca-ubx-forward-simulation:
 forward-simulation Sinca.step Subx.step Sinca.final Subx.final λ- -. False λ-. match
  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 Sinca.step Subx.step Sinca.final Subx.final λ- -. False λ-. match
  by unfold-locales
```

end

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

22 Generic program rewriting

context
  fixes *rewrite-instr* :: *nat* ⇒ *'a* ⇒ *'stack* ⇒ (*'err*, *'b* × *'stack*) *result*
begin

fun *rewrite-prog* :: *nat* ⇒ *'a list* ⇒ *'stack* ⇒ (*'err*, *'b list* × *'stack*) *result* **where**
  rewrite-prog - [] Σ = Ok ([], Σ) |
  rewrite-prog n (*x* # *xs*) Σ = *do* {
    (*x'*, Σ') ← *rewrite-instr n x* Σ;
    (*xs'*, Σ'') ← *rewrite-prog* (*Suc n*) *xs* Σ';
    Ok (*x'* # *xs'*, Σ'')
  }

lemma *rewrite-prog-map-f*:
  assumes ⋀*x* Σ*1 n x'* Σ*2*. *rewrite-instr n x* Σ*1* = *Ok* (*x'*, Σ*2*) ⟹ *f x'* = *x*
  shows *rewrite-prog n xs* Σ*1* = *Ok* (*ys*, Σ*2*) ⟹ *map f ys* = *xs*
  by (*induction xs arbitrary*: Σ*1 n ys*; *auto simp*: *assms*)

end

fun *gen-pop-push* **where**
  gen-pop-push instr (*domain*, *codomain*) Σ = (
    let ar = *length domain in*
    if ar ≤ *length* Σ ∧ *take ar* Σ = *domain then*
      Ok (*instr*, *codomain* # *drop ar* Σ)
    else
      Error ()
)

context *inca-to-ubx-simulation* **begin**

lemma *sp-rewrite-prog*:
  assumes
    rewrite-prog f n p1 Σ*1* = *Ok* (*p2*, Σ*2*) **and**
    ⋀*x* Σ*1 n x'* Σ*2*. *f n x* Σ*1* = *Ok* (*x'*, Σ*2*) ⟹ *Subx.sp-instr F x'* Σ*1* = *Ok* Σ*2*
  shows *Subx.sp F p2* Σ*1* = *Ok* Σ*2*
  using *assms*(*1*)
  by (*induction p1 arbitrary*: Σ*1 n p2*; *auto simp*: *assms*(*2*))

23 Lifting

context
  fixes
    get-arity :: *'fun* ⇒ *nat option* **and**
    load-oracle :: *nat* ⇒ *type option*
begin

fun *lift-instr* **where**
  lift-instr (*Inca.IPush x*) Σ = *Ok* (*IPush x*, *None* # Σ) |
  lift-instr Inca.IPop (- # Σ) = *Ok* (*IPop*, Σ) |
  lift-instr (*Inca.ILoad x*) (*None* # Σ) = *Ok* (*ILoad x*, *None* # Σ) |
  lift-instr (*Inca.IStore x*) (*None* # *None* # Σ) = *Ok* (*IStore x*, Σ) |
  lift-instr (*Inca.IOp op*) Σ = *gen-pop-push* (*IOp op*) (*replicate* (𝔄𝔯𝔦𝔱𝔶 *op*) *None*, *None*) Σ |
  lift-instr (*Inca.IOpInl opinl*) Σ =
    gen-pop-push (*IOpInl opinl*) (*replicate* (𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 *opinl*)) *None*, *None*) Σ |
  lift-instr (*Inca.ICall f*) Σ = *do* {
    ar ← *Result.of-option* () (*get-arity f*);
    gen-pop-push (*ICall f*) (*replicate ar None*, *None*) Σ
  } |
  lift-instr - - = *Error* ()

definition *lift* **where**
  lift = *rewrite-prog* (λ-. *lift-instr*)

lemma *sp-lift-instr*:
  assumes
    lift-instr instr Σ*1* = *Ok* (*instr'*, Σ*2*) **and**
    ⋀*x*. *rel-option* (λ*ar fd*. *arity fd* = *ar*) (*get-arity x*) (*F x*)
  shows *Subx.sp-instr F instr'* Σ*1* = *Ok* Σ*2*
  using *assms*(*1*)
proof (*induction instr* Σ*1 rule*: *lift-instr.induct*)
  fix *f* Σ *n*
  assume *lift-instr* (*Inca.ICall f*) Σ = *Ok* (*instr'*, Σ*2*)
  thus *Subx.sp-instr F instr'* Σ = *Ok* Σ*2*
    using *assms*(*2*)[*of f*]
    by (*auto simp*: *option-rel-Some1*)
qed (*auto simp add*: *Let-def*)

lemma *sp-lift*:
  assumes
    lift n p1 Σ*1* = *Ok* (*p2*, Σ*2*) **and**
    ⋀*x*. *rel-option* (λ*ar fd*. *arity fd* = *ar*) (*get-arity x*) (*F x*)
  shows *Subx.sp F p2* Σ*1* = *Ok* Σ*2*
  using *assms*
  by (*auto elim*!: *sp-rewrite-prog sp-lift-instr simp*: *lift-def*)

lemma *norm-lift-instr*: *lift-instr x* Σ*1* = *Ok* (*x'*, Σ*2*) ⟹ *norm-instr x'* = *x*

by (*induction x Σ1 rule*: *lift-instr.induct*;
auto simp: *Let-def*)

lemma *norm-lift*:
assumes *lift n xs Σ1* = *Ok* (*ys*, *Σ2*)
shows *map norm-instr ys* = *xs*
by (*auto intro*!: *rewrite-prog-map-f*[*OF - assms*[*unfolded lift-def*]] *simp*: *norm-lift-instr*)

24 Optimization

fun *result-alternative* :: ('*a*, '*b*) *result* ⇒ ('*a*, '*b*) *result* ⇒ ('*a*, '*b*) *result* (**infixr** <|> *51*) **where**
result-alternative (*Ok x*) - = *Ok x* |
result-alternative - (*Ok x*) = *Ok x* |
result-alternative (*Error e*) - = *Error e*

definition *try-unbox* **where**
try-unbox τ x Σ unbox mk-instr ≡
(*case unbox x of Some n* ⇒ *Ok* (*mk-instr n*, *Some τ # Σ*) | *None* ⇒ *Error* ())

fun *optim-instr* **where**
optim-instr - (*IPush d*) *Σ* =
try-unbox Ubx1 d Σ unbox-ubx1 IPushUbx1 <|>
try-unbox Ubx2 d Σ unbox-ubx2 IPushUbx2 <|>
Ok (*IPush d*, *None # Σ*)
|
optim-instr - (*IPushUbx1 n*) *Σ* = *Ok* (*IPushUbx1 n*, *Some Ubx1 # Σ*) |
optim-instr - (*IPushUbx2 b*) *Σ* = *Ok* (*IPushUbx2 b*, *Some Ubx2 # Σ*) |
optim-instr - *IPop* (- # *Σ*) = *Ok* (*IPop*, *Σ*) |
optim-instr n (*ILoad x*) (*None # Σ*) = (
case load-oracle n of
Some τ ⇒ *Ok* (*ILoadUbx τ x*, *Some τ # Σ*) |
- ⇒ *Ok* (*ILoad x*, *None # Σ*)
) |
optim-instr - (*ILoadUbx τ x*) (*None # Σ*) = *Ok* (*ILoadUbx τ x*, *Some τ # Σ*) |
optim-instr - (*IStore x*) (*None # None # Σ*) = *Ok* (*IStore x*, *Σ*) |
optim-instr - (*IStore x*) (*None # Some τ # Σ*) = *Ok* (*IStoreUbx τ x*, *Σ*) |
optim-instr - (*IStoreUbx* τ_1 *x*) (*None # Some* τ_2 *# Σ*) = (*if* τ_1 = τ_2 *then Ok* (*IStoreUbx* τ_1 *x*, *Σ*) *else Error* ()) |
optim-instr - (*IOp op*) *Σ* = *gen-pop-push* (*IOp op*) (*replicate* (𝔄𝔯𝔦𝔱𝔶 *op*) *None*, *None*) *Σ* |
optim-instr - (*IOpInl opinl*) *Σ* = (
let ar = 𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 *opinl*) *in*
if ar ≤ *length Σ then*
case 𝔘𝔟𝔵 *opinl* (*take ar Σ*) *of*
None ⇒ *gen-pop-push* (*IOpInl opinl*) (*replicate ar None*, *None*) *Σ* |
Some opubx ⇒ *Ok* (*IOpUbx opubx*, *snd* (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 *opubx*) *# drop ar Σ*)
else
Error ()

```
  ) |
   optim-instr - (IOpUbx opubx) Σ = gen-pop-push (IOpUbx opubx) (TypeOfOp
opubx) Σ |
  optim-instr - (ICall f) Σ = do {
    ar ← Result.of-option () (get-arity f);
    gen-pop-push (ICall f) (replicate ar None, None) Σ
  } |
  optim-instr - - - = Error ()

definition optim where
  optim ≡ rewrite-prog optim-instr

lemma
  assumes
    optim 0 [IPush d1, IPush d2, IStore y] [] = Ok (xs, ys)
    unbox-ubx1 d1 = Some x
    unbox-ubx1 d2 = None unbox-ubx2 d2 = None
  shows xs = [IPushUbx1 x, IPush d2, IStoreUbx Ubx1 y] ∧ ys = []
  using assms(1)
  by (simp add: assms optim-def try-unbox-def)

lemma norm-optim-instr: optim-instr n x Σ1 = Ok (x', Σ2) ⟹ norm-instr x' =
norm-instr x
    for x Σ1 n x' Σ2
proof (induction n x Σ1 rule: optim-instr.induct)
  fix d Σ1 n
  assume optim-instr n (Ubx.IPush d) Σ1 = Ok (x', Σ2)
  thus norm-instr x' = norm-instr (Ubx.instr.IPush d)
    using Subx.box-unbox-inverse
    by (auto elim!: result-alternative.elims simp: try-unbox-def option.case-eq-if)
next
  fix x Σ1 n
  assume assms: optim-instr n (Ubx.ILoad x) (None # Σ1) = Ok (x', Σ2)
  thus norm-instr x' = norm-instr (Ubx.ILoad x)
  proof (cases load-oracle n)
    case None
    then show ?thesis using assms by simp
  next
    case (Some x)
    then show ?thesis
      using assms by (cases x) auto
  qed
next
  fix opinl Σ1 n
  assume assms: optim-instr n (IOpInl opinl) Σ1 = Ok (x', Σ2)
  then have arity-le-Σ1: Arity (DeInl opinl) ≤ length Σ1
    using prod.inject by fastforce

  show norm-instr x' = norm-instr (IOpInl opinl)
```

```
  proof (cases 𝔘𝔟𝔵 opinl (take (𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 opinl)) Σ1))
    case None
    then show ?thesis
      using assms arity-le-Σ1
      by (simp add: Let-def)
  next
    case (Some opubx)
    then show ?thesis
      using assms arity-le-Σ1
      by (auto simp: case-prod-beta Subx.𝔘𝔟𝔵-invertible)
  qed
next
  fix opubx Σ1 n
  assume assms: optim-instr n (IOpUbx opubx) Σ1 = Ok (x', Σ2)
  then show norm-instr x' = norm-instr (IOpUbx opubx)
    by (cases 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx; auto simp: Let-def)
qed (auto simp: Let-def)

lemma norm-optim:
  assumes optim n xs Σ1 = Ok (ys, Σ2)
  shows map norm-instr ys = map norm-instr xs
  using assms
  unfolding optim-def
  by (induction xs arbitrary: Σ1 n ys Σ2; auto simp: norm-optim-instr)

lemma sp-optim-instr:
  assumes
    optim-instr n instr Σ1 = Ok (instr', Σ2) and
    ⋀x. rel-option (λar fd. arity fd = ar) (get-arity x) (F x)
  shows Subx.sp-instr F instr' Σ1 = Ok Σ2
  using assms(1)
  apply (induction n instr Σ1 rule: optim-instr.induct;
      (auto simp: Let-def; fail)?)
  subgoal for - d
    by (auto elim!: result-alternative.elims simp: try-unbox-def option.case-eq-if)
  subgoal for n
    by (cases load-oracle n; auto)
  subgoal for - opinl Σ
    apply (simp add: Let-def)
    apply (cases 𝔘𝔟𝔵 opinl (take (𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 opinl)) Σ))
    apply (auto dest: list-all-eq-const-imp-replicate
        simp: case-prod-beta Let-def min.absorb2)
    subgoal for opubx
      using Subx.𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-𝔄𝔯𝔦𝔱𝔶[of opubx]
    by (cases 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx; auto simp: Subx.𝔘𝔟𝔵-invertible dest: Subx.𝔘𝔟𝔵-opubx-type)
    done
  subgoal for - opubx
    by (cases 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx; auto simp add: Let-def)
  subgoal for - f
```

using *assms(2)*[*of f*]
by (*auto simp*: *option-rel-Some1*)
done

lemma *sp-optim*:
assumes
optim n p1 Σ1 = Ok (*p2*, *Σ2*) **and**
⋀*x*. *rel-option* (λ*ar fd*. *arity fd* = *ar*) (*get-arity x*) (*F x*)
shows *Subx.sp F p2 Σ1 = Ok Σ2*
using *assms*
by (*auto elim*!: *sp-rewrite-prog sp-optim-instr simp*: *optim-def*)

25 Compilation of function definition

fun *compile-fundef* **where**
compile-fundef (*Fundef instrs ar*) = *do* {
(*xs*, Σ_1) ← *lift 0 instrs* (*replicate ar None* :: *type option list*);

— Ensure that the function returns a single dynamic result
() ← *if* Σ_1 = [*None*] *then Ok* () *else Error* ();

(*ys*, Σ_2) ← *optim 0 xs* (*replicate ar None*);

Ok (*Fundef* (
if Σ_2 = [*None*] *then*
ys — use optimization
else
xs — cancel optimization
) *ar*)
}

lemma *rel-compile-fundef*:
assumes *compile-fundef fd1 = Ok fd2*
shows *rel-fundef norm-eq fd1 fd2*
proof (*cases fd1*)
case (*Fundef xs ar*)
obtain *ys zs Σ2* **where**
lift-xs: *lift 0 xs* (*replicate ar None* :: *type option list*) = *Ok* (*ys*, [*None*]) **and**
optim-ys: *optim 0 ys* (*replicate ar None* :: *type option list*) = *Ok* (*zs*, *Σ2*) **and**
check: *fd2 = Fundef* (*if Σ2* = [*None*] *then zs else ys*) *ar*
using *assms* **unfolding** *Fundef*
unfolding *compile-fundef.simps*
by (*auto simp*: *Nitpick.case-unit-unfold if-then-else-distributive*)

then have *map norm-instr ys = xs*
by (*auto intro*: *norm-lift*)

show *?thesis*
proof (*cases Σ2* = [*None*])

```
    case True
    then show ?thesis
      using True Fundef
      using check norm-lift[OF lift-xs, symmetric]
      using norm-optim[OF optim-ys, symmetric]
      by (simp add: list.rel-map list.rel-refl)
  next
    case False
    then show ?thesis
      using Fundef check norm-lift[OF lift-xs, symmetric]
      by (simp add: list.rel-map list.rel-refl)
  qed
qed

lemma sp-compile-fundef:
  assumes
    compile-fundef fd1 = Ok fd2 and
    ⋀x. rel-option (λar fd. arity fd = ar) (get-arity x) (F x)
  shows sp-fundef F fd2 (body fd2) = Ok [None]
proof (cases fd1)
  case (Fundef xs ar)
  with assms show ?thesis
     by (auto elim!: sp-optim sp-lift simp: sp-fundef-def Nitpick.case-unit-unfold
if-eq-const-conv)
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
```

26 Compilation of function environment

```
definition compile-env-entry where
  compile-env-entry 𝒜 𝒪 x = map-result id (Pair (fst x)) (compile-fundef 𝒜 (𝒪
(fst x)) (snd x))

lemma rel-compile-env-entry:
  assumes compile-env-entry 𝒪 𝒜 (f, fd1) = Ok (f, fd2)
  shows rel-fundef norm-eq fd1 fd2
  using assms unfolding compile-env-entry-def
  using rel-compile-fundef
```

by *auto*

lemma *sp-compile-env-entry*:
assumes
compile-env-entry $\mathcal{A}$ $\mathcal{O}$ (*f*, *fd1*) = *Ok* (*f*, *fd2*) **and**
$\bigwedge$*x*. *rel-option* (λ*ar fd*. *arity fd* = *ar*) ($\mathcal{A}$ *x*) (*F x*)
shows *sp-fundef F fd2* (*body fd2*) = *Ok* [*None*]
using *assms* **unfolding** *compile-env-entry-def*
by (*auto elim*: *sp-compile-fundef*)

definition *compile-env* **where**
compile-env $\mathcal{A}$ $\mathcal{O}$ *e* ≡
map-result id Subx.Fenv.from-list
(*Result.those* (*map* (*compile-env-entry* $\mathcal{A}$ $\mathcal{O}$) (*Finca-to-list e*)))

lemma *list-assoc-those-compile-env-entries*:
Result.those (*map* (*compile-env-entry* $\mathcal{A}$ $\mathcal{O}$) *xs*) = *Ok ys* ⟹
rel-option (λ*fd1 fd2*. *compile-fundef* $\mathcal{A}$ ($\mathcal{O}$ *f*) *fd1* = *Ok fd2*) (*map-of xs f*) (*map-of ys f*)
proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
then obtain *y ys′* **where**
xs-def: *ys* = *y* # *ys′* **and**
comp-x: *compile-env-entry* $\mathcal{A}$ $\mathcal{O}$ *x* = *Ok y* **and**
comp-xs: *Result.those* (*map* (*compile-env-entry* $\mathcal{A}$ $\mathcal{O}$) *xs*) = *Ok ys′*
by *auto*

obtain *g fd1 fd2* **where** *prods*: *x* = (*g*, *fd1*) *y* = (*g*, *fd2*)
using *comp-x*
by (*cases x*) (*auto simp*: *compile-env-entry-def*)

have *compile-fundef* $\mathcal{A}$ ($\mathcal{O}$ *g*) *fd1* = *Ok fd2*
using *comp-x* **unfolding** *prods compile-env-entry-def*
by *auto*

then show *?case*
unfolding *prods xs-def*
using *Cons.IH*[*OF comp-xs*]
by *simp*
qed

lemma *compile-env-rel-compile-fundef*:
assumes *compile-env* $\mathcal{A}$ $\mathcal{O}$ *F1* = *Ok F2*
shows *rel-option* (λ*fd1 fd2*. *compile-fundef* $\mathcal{A}$ ($\mathcal{O}$ *f*) *fd1* = *Ok fd2*) (*Finca-get F1 f*) (*Fubx-get F2 f*)
proof −

```
 obtain xs where those-xs: Result.those (map (compile-env-entry 𝒜 𝒪) (Finca-to-list
F1)) = Ok xs
    using assms[unfolded compile-env-def]
    by auto
  hence F2-def: F2 = Subx.Fenv.from-list xs
    using assms[unfolded compile-env-def]
    by simp
  show ?thesis
  proof (cases map-of (Finca-to-list F1) f)
    case None
    then show ?thesis
      unfolding F2-def Subx.Fenv.from-list-correct[symmetric]
      unfolding Sinca.Fenv.to-list-correct[symmetric]
      using list-assoc-those-compile-env-entries[OF those-xs, of f]
      by simp
  next
    case (Some fd1)
    then show ?thesis
      unfolding F2-def Subx.Fenv.from-list-correct[symmetric]
      unfolding Sinca.Fenv.to-list-correct[symmetric]
      using list-assoc-those-compile-env-entries[OF those-xs, of f]
      by simp
  qed
qed

lemma rel-those-compile-env-entries:
  Result.those (map (compile-env-entry 𝒜 𝒪) xs) = Ok ys ⟹
   rel-fundefs (Finca-get (Sinca.Fenv.from-list xs)) (Fubx-get (Subx.Fenv.from-list
ys))
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case
    using rel-fundefs-empty by simp
next
  case (Cons x xs)
  then obtain y ys' where
    xs-def: ys = y # ys' and
    comp-x: compile-env-entry 𝒜 𝒪 x = Ok y and
    comp-xs: Result.those (map (compile-env-entry 𝒜 𝒪) xs) = Ok ys'
    by auto

  obtain f fd1 fd2 where prods: x = (f, fd1) y = (f, fd2)
    using comp-x
    by (cases x) (auto simp: compile-env-entry-def)

  have rel-fundef norm-eq fd1 fd2
    using rel-compile-env-entry[OF comp-x[unfolded prods]] .

  then show ?case
```

```
    unfolding prods xs-def
   unfolding Sinca.Fenv.from-list-correct[symmetric] Subx.Fenv.from-list-correct[symmetric]
    unfolding rel-fundefs-def
    apply auto
    using Cons.IH[OF comp-xs]
   unfolding Sinca.Fenv.from-list-correct[symmetric] Subx.Fenv.from-list-correct[symmetric]
    by (simp add: rel-fundefs-def)
qed

lemma rel-fundefs-compile-env:
  assumes compile-env 𝒜 𝒪 e = Ok e'
  shows rel-fundefs (Finca-get e) (Fubx-get e')
proof −
  obtain xs where
     FOO: Result.those (map (compile-env-entry 𝒜 𝒪) (Finca-to-list e)) = Ok xs
and
    BAR: e' = Subx.Fenv.from-list xs
    using assms
    by (auto simp: compile-env-def)

  show ?thesis
    using rel-those-compile-env-entries[OF FOO]
    unfolding BAR
    unfolding Sinca.Fenv.get-from-list-to-list
    .
qed

lemma sp-fundefs-compile-env:
  assumes
    compile-env 𝒜 𝒪 F1 = Ok F2 and
    ⋀x. rel-option (λar fd. arity fd = ar) (𝒜 x) (Finca-get F1 x)
  shows sp-fundefs (Fubx-get F2)
  unfolding sp-fundefs-def
  proof (intro allI impI)
    fix f fd2
    assume F2-f: Fubx-get F2 f = Some fd2
    then obtain fd1 where
      Finca-get F1 f = Some fd1 and
      compile-fd1: compile-fundef 𝒜 (𝒪 f) fd1 = Ok fd2
      using compile-env-rel-compile-fundef[OF assms(1), of f]
      by (auto simp: option-rel-Some2)

    note rel-F1-F2 = rel-fundefs-compile-env[OF assms(1)]

    have rel-option (λar fd. arity fd = ar) (𝒜 f) (Fubx-get F2 f) for f
      using assms(2)[of f]
    proof (cases rule: option.rel-cases)
      case None
      then show ?thesis
```

```
      using rel-fundefs-None1[OF rel-F1-F2]
      by simp
    next
     case (Some ar fd1)
     then obtain fd2 where
      Fubx-get F2 f = Some fd2 and
      rel-fd1-fd2: rel-fundef (λx y. x = norm-instr y) fd1 fd2
      using rel-fundefs-Some1[OF rel-F1-F2]
      by (meson rel-fundefs-Some1)
     with Some show ?thesis
      by (simp add: rel-fundef-arities)
    qed
    thus sp-fundef (Fubx-get F2) fd2 (body fd2) = Ok [None]
     by (auto intro!: sp-compile-fundef[OF compile-fd1])
  qed
```

27 Compilation of program

```
fun compile where
  compile (Prog F H f) = Result.to-option (do {
    F' ← compile-env (map-option arity ∘ Finca-get F) load-oracle F;
    Ok (Prog F' H f)
  })

lemma compile-load:
  assumes
    compile-p1: compile p1 = Some p2 and
    load: Subx.load p2 s2
  shows ∃ s1. Sinca.load p1 s1 ∧ 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 (map-option arity ∘ Finca-get F1) load-oracle F1 = Ok
F2 and
    p2-def: p2 = Prog F2 H main
    using compile-p1
    by auto

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

  show ?thesis
    using assms(2) unfolding p2-def
  proof (cases - s2 rule: Subx.load.cases)
    case (1 fd2)
    let ?s1 = State F1 H [Frame main 0 []]

    from 1 obtain fd1 where
      Fstd-main: Finca-get F1 main = Some fd1 and
```

compile-fundef (*map-option arity* ∘ *Finca-get F1*) (*load-oracle main*) *fd1* = *Ok fd2*
using *compile-env-rel-compile-fundef*[*OF compile-F1*, *of main*]
by (*auto simp*: *option-rel-Some2*)
with *1* **have** *arity fd1* = *0*
using *fundef.rel-sel rel-compile-fundef* **by** *metis*
hence *Sinca.load p1 ?s1*
unfolding *p1-def*
by (*auto intro*!: *Sinca.load.intros*[*OF Fstd-main*])
moreover have *?s1* ∼ *s2*
proof −
have *sp-fundefs* (*Fubx-get F2*)
by (*auto intro*: *sp-fundefs-compile-env*[*OF compile-F1*] *simp*: *option.rel-map option.rel-refl*)
moreover have *rel-stacktraces* (*Fubx-get F2*) [*Frame main 0* []] [*Frame main 0* []] *None*
using *1* **by** (*auto intro*!: *rel-stacktraces.intros simp*: *sp-fundef-def*)
ultimately show *?thesis*
unfolding *1*
using *rel-F1-F2*
by (*auto intro*!: *match.intros*)
qed
ultimately show *?thesis* **by** *auto*
qed
qed

interpretation *std-to-inca-compiler*:
compiler Sinca.step Subx.step Sinca.final Subx.final Sinca.load Subx.load
λ- -. *False* λ-. *match compile*
using *compile-load*
by *unfold-locales auto*

end

end
theory *Op-example*
imports *OpUbx Ubx-type-inference Global Unboxed-lemmas*
begin

28 Dynamic values

datatype *dynamic* = *DNil* | *DBool bool* | *DNum nat*

definition *is-true* **where**
is-true d = (*d* = *DBool True*)

definition *is-false* **where**
is-false d = (*d* = *DBool False*)

definition *box-bool* :: *bool* ⇒ *dynamic* **where**
box-bool = *DBool*

definition *box-num* :: *nat* ⇒ *dynamic* **where**
box-num = *DNum*

fun *unbox-num* :: *dynamic* ⇒ *nat 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 is-true is-false box-num unbox-num box-bool unbox-bool
proof (*unfold-locales*; (*simp add*: *is-true-def is-false-def*)*?*)
fix *d n*
show *unbox-num d* = *Some n* ⟹ *box-num n* = *d*
by (*cases d*; *simp add*: *box-num-def*)
next
fix *d b*
show *unbox-bool d* = *Some b* ⟹ *box-bool b* = *d*
by (*cases d*; *simp add*: *box-bool-def*)
qed

29 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* ⇒ *dynamic* **where**
eval-Mul [*DBool x*, *DBool y*] = *DBool* (*x* ∧ *y*) |

```
  eval-Mul [DNum x, DNum y] = DNum (x * y) |
  eval-Mul [-, -] = DNil

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

lemma eval-arith-domain: length xs = ar op ⟹ ∃ y. eval op xs = y
  by simp

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

30 Inlined operations

```
datatype opinl =
  OpAddNum |
  OpMulNum

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

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

fun deinl :: opinl ⇒ 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 ⟹ deinl opinl = op
  by (induction op xs rule: inl.induct; simp)

fun eval-AddNum :: dynamic list ⇒ dynamic where
  eval-AddNum [DNum x, DNum y] = DNum (x + y) |
```

eval-AddNum [*DBool x*, *DBool y*] = *DBool* (*x* ∨ *y*) |
eval-AddNum [-, -] = *DNil*

fun *eval-MulNum* :: *dynamic list* ⇒ *dynamic* **where**
eval-MulNum [*DNum x*, *DNum y*] = *DNum* (*x* ∗ *y*) |
eval-MulNum [*DBool x*, *DBool y*] = *DBool* (*x* ∧ *y*) |
eval-MulNum [-, -] = *DNil*

fun *eval-inl* :: *opinl* ⇒ *dynamic list* ⇒ *dynamic* **where**
eval-inl OpAddNum = *eval-AddNum* |
eval-inl OpMulNum = *eval-MulNum*

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

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

lemma *eval-inl-correct*:
length xs = *ar* (*deinl opinl*) ⟹ *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* ⟹ *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*)

31 Unboxed operations

datatype *opubx* =
OpAddNumUbx

fun *ubx* :: *opinl* ⇒ *type option list* ⇒ *opubx option* **where**
ubx OpAddNum [*Some Ubx1*, *Some Ubx1*] = *Some OpAddNumUbx* |
ubx - - = *None*

fun *deubx* :: *opubx* ⇒ *opinl* **where**
deubx OpAddNumUbx = *OpAddNum*

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

fun *eval-AddNumUbx* :: (*dynamic*, *nat*, *bool*) *unboxed list* ⇒ (*dynamic*, *nat*, *bool*) *unboxed option* **where**
  eval-AddNumUbx [*OpUbx1 x*, *OpUbx1 y*] = *Some* (*OpUbx1* (*x* + *y*)) |
  eval-AddNumUbx - = *None*

fun *eval-ubx* :: *opubx* ⇒ (*dynamic*, *nat*, *bool*) *unboxed list* ⇒ (*dynamic*, *nat*, *bool*) *unboxed option* **where**
  eval-ubx OpAddNumUbx = *eval-AddNumUbx*

lemma *eval-ubx-correct*:
  eval-ubx opubx xs = *Some z* ⟹
   eval-inl (*deubx opubx*) (*map unboxed-dynamic.norm-unboxed xs*) = *unboxed-dynamic.norm-unboxed z*
  apply (*cases opubx*; *simp*)
  apply (*induction xs rule*: *eval-AddNumUbx.induct*; *auto*)
  by (*simp add*: *box-num-def*)

lemma *eval-ubx-to-inl*:
  assumes *eval-ubx opubx* Σ = *Some z*
 shows *inl* (*deinl* (*deubx opubx*)) (*map unboxed-dynamic.norm-unboxed* Σ) = *Some* (*deubx opubx*)
proof (*cases opubx*)
  case *OpAddNumUbx*
  then have *eval-AddNumUbx* Σ = *Some z*
    using *assms* **by** *simp*
  then show *?thesis*
    using *OpAddNumUbx*
    apply (*induction* Σ *rule*: *eval-AddNumUbx.induct*; *simp*)
    by (*simp add*: *box-num-def*)
qed

31.1 Abstract interpretation

fun *typeof-opubx* :: *opubx* ⇒ *type option list* × *type option* **where**
  typeof-opubx OpAddNumUbx = ([*Some Ubx1*, *Some Ubx1*], *Some Ubx1*)

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

lemma *typeof-opubx-correct*:
  typeof-opubx opubx = (*map typeof xs*, *codomain*) ⟹
    ∃ *y*. *eval-ubx opubx xs* = *Some y* ∧ *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 ⟹
    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
    is-true is-false box-num unbox-num box-bool 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 Env-list Dynamic
    VeriComp.Language
begin

datatype ('dyn, 'var, 'fun, 'op) instr =
  IPush 'dyn |
  IPop |
  ILoad 'var |
  IStore 'var |
  IOp 'op |
  ICJump nat |
  ICall 'fun

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 is-true is-false +
nary-operations $\mathfrak{Op}$ $\mathfrak{Arity}$
for
F-empty **and**
F-get :: *'fenv* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*) *instr fundef option* **and**
F-add **and** *F-to-list* **and**
heap-empty **and**
heap-get :: *'henv* ⇒ *'var* × *'dyn* ⇒ *'dyn option* **and**
heap-add **and** *heap-to-list* **and**
is-true :: *'dyn* ⇒ *bool* **and** *is-false* **and**
$\mathfrak{Op}$ :: *'op* ⇒ *'dyn list* ⇒ *'dyn* **and** $\mathfrak{Arity}$
begin

inductive *final* :: (*'fenv*, *'henv*, (*'fun*, *'dyn*) *frame*) *state* ⇒ *bool* **where**
F-get F f = *Some fd* ⟹ *pc* = *length* (*body fd*) ⟹ *final* (*State F H* [*Frame f pc* Σ])

inductive *step* (**infix** → *55*) **where**
step-push:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IPush d* ⟹
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*d* # Σ) # *st*) |

step-pop:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IPop* ⟹
State F H (*Frame f pc* (*d* # Σ) # *st*) → *State F H* (*Frame f* (*Suc pc*) Σ # *st*) |

step-load:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *ILoad x* ⟹
heap-get H (*x*, *y*) = *Some d* ⟹
State F H (*Frame f pc* (*y* # Σ) # *st*) → *State F H* (*Frame f* (*Suc pc*) (*d* # Σ) # *st*) |

step-store:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IStore x* ⟹
heap-add H (*x*, *y*) *d* = *H'* ⟹
State F H (*Frame f pc* (*y* # *d* # Σ) # *st*) → *State F H'* (*Frame f* (*Suc pc*) Σ # *st*) |

step-op:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IOp op* ⟹
$\mathfrak{Arity}$ *op* = *ar* ⟹ *ar* ≤ *length* Σ ⟹ $\mathfrak{Op}$ *op* (*take ar* Σ) = *x* ⟹
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*x* # *drop ar* Σ) # *st*) |

step-cjump-true:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *ICJump n* ⟹
is-true d ⟹
State F H (*Frame f pc* (*d* # Σ) # *st*) → *State F H* (*Frame f n* Σ # *st*) |

step-cjump-false:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = ICJump n
⟹
is-false d ⟹
State F H (Frame f pc (d # Σ) # st) → State F H (Frame f (Suc pc) Σ # st) |

step-fun-call:
F-get F f = Some fd ⟹ pc < length (body fd) ⟹ body fd ! pc = ICall g ⟹
F-get F g = Some gd ⟹ arity gd ≤ length Σ ⟹
$frame_f$ *= Frame f pc Σ ⟹* $frame_g$ *= Frame g 0 (take (arity gd) Σ) ⟹*
State F H ($frame_f$ *# st) → State F H* ($frame_g$ *#* $frame_f$ *# st)* |

step-fun-end:
F-get F g = Some gd ⟹ arity gd ≤ length Σ_f *⟹* pc_g *= length (body gd) ⟹*
$frame_g$ *= Frame g* pc_g Σ_g *⟹* $frame_f$ *= Frame f* pc_f Σ_f *⟹*
$frame_f{}'$ *= Frame f (Suc* pc_f*) (*Σ_g *@ drop (arity gd)* Σ_f*) ⟹*
State F H ($frame_g$ *#* $frame_f$ *# st) → State F H* ($frame_f{}'$ *# st)*

lemma *step-deterministic*: *s1 → s2 ⟹ s1 → s3 ⟹ s2 = s3*
by (*induction rule*: *step.cases*;
auto elim!: *step.cases dest*: *is-true-and-is-false-implies-False*)

lemma *final-finished*: *final s ⟹ finished step s*
unfolding *finished-def*
proof
assume *final s* **and** ∃ *x*. *step s x*
then obtain s' **where** *step s* s'
by *auto*
thus *False*
using ‹*final s*›
by (*auto elim*!: *step.cases final.cases*)
qed

sublocale *semantics step final*
using *final-finished step-deterministic*
by *unfold-locales*

inductive *load* :: ('*fenv*, '*a*, '*fun*) *prog* ⇒ ('*fenv*, '*a*, ('*fun*, '*b*) *frame*) *state* ⇒ *bool*
where
F-get F main = Some fd ⟹ arity fd = 0 ⟹
load (Prog F H main) (State F H [*Frame main 0* []])

sublocale *language step final load*
by *unfold-locales*

end

end

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

32 Generic definitions

print-locale *std*

locale *std-inca-simulation* =
Sstd: *std*
Fstd-empty Fstd-get Fstd-add Fstd-to-list
heap-empty heap-get heap-add heap-to-list
is-true is-false
𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 +
Sinca: *inca*
Finca-empty Finca-get Finca-add Finca-to-list
heap-empty heap-get heap-add heap-to-list
is-true is-false
𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩
for
— Functions environments
Fstd-empty **and**
Fstd-get :: *'fenv-std* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*) *Std.instr fundef option*
and
Fstd-add **and** *Fstd-to-list* **and**

Finca-empty **and**
Finca-get :: *'fenv-inca* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) *Inca.instr fundef option* **and**
Finca-add **and** *Finca-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
is-true :: *'dyn* ⇒ *bool* **and** *is-false* **and**

— n-ary operations
𝔒𝔭 :: *'op* ⇒ *'dyn list* ⇒ *'dyn* **and** 𝔄𝔯𝔦𝔱𝔶 **and**
𝔍𝔫𝔩𝔒𝔭 **and** 𝔍𝔫𝔩 **and** 𝔍𝔰𝔍𝔫𝔩 **and** 𝔇𝔢𝔍𝔫𝔩 :: *'opinl* ⇒ *'op*
begin

fun *norm-instr* **where**
norm-instr (*Inca.IPush d*) = *Std.IPush d* |
norm-instr Inca.IPop = *Std.IPop* |
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 (𝔇𝔢𝔍𝔫𝔩 opinl) |
  norm-instr (Inca.ICJump n) = Std.ICJump n |
  norm-instr (Inca.ICall x) = Std.ICall x

abbreviation (input) norm-eq where
  norm-eq x y ≡ x = norm-instr y

definition rel-fundefs where
  rel-fundefs f g = (∀ x. rel-option (rel-fundef (λx y. x = norm-instr y)) (f x) (g x))

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-fundef-body-nth:
  assumes rel-fundef norm-eq fd1 fd2 and pc < length (body fd1)
  shows body fd1 ! pc = norm-instr (body fd2 ! pc)
  using assms
  by (auto dest: list-all2-nthD simp: fundef.rel-sel)

lemma rel-fundef-rewrite-body:
  assumes
    rel-fundef norm-eq fd1 fd2 and
    norm-instr (body fd2 ! pc) = norm-instr instr
  shows rel-fundef norm-eq fd1 (rewrite-fundef-body fd2 pc instr)
  using assms(1)
proof (cases rule: fundef.rel-cases)
  case (Fundef xs ar' ys ar)
  hence length xs = length ys
    by (simp add: list-all2-conv-all-nth)
  hence length xs = length (rewrite ys pc instr)
    by simp
```

```
  hence list-all2 norm-eq xs (rewrite ys pc instr)
  proof (elim list-all2-all-nthI)
    fix n
    assume n < length xs
    hence n < length ys
      by (simp add: ‹length xs = length ys›)
    thus xs ! n = norm-instr (rewrite ys pc instr ! n)
      using list-all2-nthD[OF ‹list-all2 norm-eq xs ys› ‹n < length xs›, symmetric]
      using assms(2)[unfolded Fundef(2), simplified]
      by (cases pc = n; simp)
  qed
  thus ?thesis
    using Fundef by simp
qed

lemma rel-fundefs-rewrite:
  assumes
    rel-F1-F2: rel-fundefs (Fstd-get F1) (Finca-get F2) and
    F2-get-f: Finca-get F2 f = Some fd2 and
    F2-add-f: Finca-add F2 f (rewrite-fundef-body fd2 pc instr) = F2' and
    norm-eq: norm-instr (body fd2 ! pc) = norm-instr instr
  shows rel-fundefs (Fstd-get F1) (Finca-get F2')
  unfolding rel-fundefs-def
proof
  fix x
  show rel-option (rel-fundef norm-eq) (Fstd-get F1 x) (Finca-get F2' x)
  proof (cases x = f)
    case True
   then have F2'-get-x: Finca-get F2' x = Some (rewrite-fundef-body fd2 pc instr)
      using F2-add-f by auto
    obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1 fd2
      using rel-fundefs-Some2[OF rel-F1-F2 F2-get-f] by auto
    thus ?thesis
      unfolding F2'-get-x option-rel-Some2
      using True rel-fundef-rewrite-body[OF - norm-eq]
      by auto
  next
    case False
    then have Finca-get F2' x = Finca-get F2 x
      using F2-add-f by auto
    then show ?thesis
      using rel-F1-F2 rel-fundefs-def
      by fastforce
  qed
qed
```

33 Simulation relation

```
inductive match (infix ∼ 55) where
```

rel-fundefs (*Fstd-get F1*) (*Finca-get F2*) $\Longrightarrow$ (*State F1 H st*) $\sim$ (*State F2 H st*)

34 Backward simulation

lemma *backward-lockstep-simulation*:
assumes *Sinca.step s2 s2'* **and** $s1 \sim s2$
shows $\exists$ *s1'*. *Sstd.step s1 s1'* $\wedge$ $s1' \sim s2'$
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**
rel-F1-F2: *rel-fundefs* (*Fstd-get F1*) (*Finca-get F2*)
by (*auto elim*: *match.cases*)
from *assms*(*1*) **show** *?thesis*
unfolding *s1-def s2-def*
proof (*induction State F2 H st s2' rule*: *Sinca.step.induct*)
case (*step-push f fd2 pc d* Σ *st'*)
then obtain *fd1* **where** *Fstd-get F1 f* $=$ *Some fd1* **and** *rel-fundef norm-eq fd1 fd2*
using *rel-fundefs-Some2*[*OF rel-F1-F2*] **by** *blast*
show *?case* (**is** $\exists x.$ *?STEP x* $\wedge$ *?MATCH x*)
proof −
let *?s1'* $=$ *State F1 H* (*Frame f* (*Suc pc*) (*d* # Σ) # *st'*)
have *?STEP ?s1'*
using *step-push* ‹*Fstd-get F1 f* $=$ *Some fd1*› ‹*rel-fundef norm-eq fd1 fd2*›
by (*auto intro*!: *Sstd.step-push simp*: *rel-fundef-body-nth*)
moreover have *?MATCH ?s1'*
using *rel-F1-F2* **by** (*auto intro*: *match.intros*)
ultimately show *?thesis* **by** *blast*
qed
next
case (*step-pop f fd2 pc d* Σ *st'*)
then obtain *fd1* **where** *Fstd-get F1 f* $=$ *Some fd1* **and** *rel-fundef norm-eq fd1 fd2*
using *rel-fundefs-Some2*[*OF rel-F1-F2*] **by** *blast*
show *?case* (**is** $\exists x.$ *?STEP x* $\wedge$ *?MATCH x*)
proof −
let *?s1'* $=$ *State F1 H* (*Frame f* (*Suc pc*) Σ # *st'*)
have *?STEP ?s1'*
using *step-pop* ‹*Fstd-get F1 f* $=$ *Some fd1*› ‹*rel-fundef norm-eq fd1 fd2*›
by (*auto intro*!: *Sstd.step-pop simp*: *rel-fundef-body-nth*)
moreover have *?MATCH ?s1'*
using *rel-F1-F2* **by** (*auto intro*: *match.intros*)
ultimately show *?thesis* **by** *blast*
qed
next
case (*step-load f fd2 pc x y d* Σ *st'*)
then obtain *fd1* **where** *Fstd-get F1 f* $=$ *Some fd1* **and** *rel-fundef norm-eq fd1 fd2*

```
      using rel-fundefs-Some2[OF rel-F1-F2] by blast
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F1 H (Frame f (Suc pc) (d # Σ) # st')
      have ?STEP ?s1'
        using step-load ‹Fstd-get F1 f = Some fd1› ‹rel-fundef norm-eq fd1 fd2›
        by (auto intro!: Sstd.step-load simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1'
        using rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by blast
    qed
  next
    case (step-store f fd2 pc x y d H' Σ st')
    then obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1
fd2
      using rel-fundefs-Some2[OF rel-F1-F2] by blast
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F1 H' (Frame f (Suc pc) Σ # st')
      have ?STEP ?s1'
        using step-store ‹Fstd-get F1 f = Some fd1› ‹rel-fundef norm-eq fd1 fd2›
        by (auto intro!: Sstd.step-store simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1'
        using rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by blast
    qed
  next
    case (step-op f fd2 pc op ar Σ x st')
    then obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1
fd2
      using rel-fundefs-Some2[OF rel-F1-F2] by blast
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F1 H (Frame f (Suc pc) (x # drop ar Σ) # st')
      have ?STEP ?s1'
        using step-op ‹Fstd-get F1 f = Some fd1› ‹rel-fundef norm-eq fd1 fd2›
        by (auto intro!: Sstd.step-op simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1'
        using rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by blast
    qed
  next
    case (step-op-inl f fd2 pc op ar Σ opinl x F2' st')
    then obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1
fd2
      using rel-fundefs-Some2[OF rel-F1-F2] by blast
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F1 H (Frame f (Suc pc) (x # drop ar Σ) # st')
```

```
    have ?STEP ?s1'
      using step-op-inl ‹Fstd-get F1 f = Some fd1› ‹rel-fundef norm-eq fd1 fd2›
      using  Sinca.𝔍𝔫𝔩𝔒𝔭-correct Sinca.𝔍𝔫𝔩-invertible
      by (auto intro!: Sstd.step-op simp: rel-fundef-body-nth)
    moreover have ?MATCH ?s1'
      using step-op-inl Sinca.𝔍𝔫𝔩-invertible
      by (auto intro!: match.intros rel-fundefs-rewrite[OF rel-F1-F2])
    ultimately show ?thesis by blast
  qed
 next
  case (step-op-inl-hit f fd2 pc opinl ar Σ x st')
  then obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1
fd2
    using rel-fundefs-Some2[OF rel-F1-F2] by blast
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
  proof −
    let ?s1' = State F1 H (Frame f (Suc pc) (x # drop ar Σ) # st')
    have ?STEP ?s1'
     using step-op-inl-hit ‹Fstd-get F1 f = Some fd1› ‹rel-fundef norm-eq fd1 fd2›
Sinca.𝔍𝔫𝔩𝔒𝔭-correct
      by (auto intro!: Sstd.step-op simp: rel-fundef-body-nth)
    moreover have ?MATCH ?s1'
      using rel-F1-F2 by (auto intro: match.intros)
    ultimately show ?thesis by blast
  qed
 next
  case (step-op-inl-miss f fd2 pc opinl ar Σ x F' st')
  then obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1
fd2
    using rel-fundefs-Some2[OF rel-F1-F2] by blast
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
  proof −
    let ?s1' = State F1 H (Frame f (Suc pc) (x # drop ar Σ) # st')
    have ?STEP ?s1'
      using step-op-inl-miss ‹Fstd-get F1 f = Some fd1› ‹rel-fundef norm-eq fd1
fd2› Sinca.𝔍𝔫𝔩𝔒𝔭-correct
      by (auto intro!: Sstd.step-op simp: rel-fundef-body-nth)
    moreover have ?MATCH ?s1'
      using step-op-inl-miss rel-fundefs-rewrite[OF rel-F1-F2]
      by (auto intro: match.intros)
    ultimately show ?thesis by blast
  qed
 next
  case (step-cjump-true f fd2 pc n d Σ st')
  then obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1
fd2
    using rel-fundefs-Some2[OF rel-F1-F2] by blast
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
  proof −
```

```
    let ?s1' = State F1 H (Frame f n Σ # st')
    have ?STEP ?s1'
      using step-cjump-true ‹Fstd-get F1 f = Some fd1› ‹rel-fundef norm-eq fd1
fd2›
      by (auto intro!: Sstd.step-cjump-true simp: rel-fundef-body-nth)
    moreover have ?MATCH ?s1'
      using rel-F1-F2 by (auto intro: match.intros)
    ultimately show ?thesis by blast
  qed
 next
  case (step-cjump-false f fd2 pc n d Σ st')
  then obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1
fd2
    using rel-fundefs-Some2[OF rel-F1-F2] by blast
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
  proof −
    let ?s1' = State F1 H (Frame f (Suc pc) Σ # st')
    have ?STEP ?s1'
      using step-cjump-false ‹Fstd-get F1 f = Some fd1› ‹rel-fundef norm-eq fd1
fd2›
      by (auto intro!: Sstd.step-cjump-false simp: rel-fundef-body-nth)
    moreover have ?MATCH ?s1'
      using rel-F1-F2 by (auto intro: match.intros)
    ultimately show ?thesis by blast
  qed
 next
  case (step-fun-call f fd2 pc g gd2 Σ frame_f frame_g st')
  obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fd1-fd2: rel-fundef norm-eq
fd1 fd2
    using step-fun-call rel-fundefs-Some2[OF rel-F1-F2] by blast
   obtain gd1 where Fstd-get F1 g = Some gd1 and rel-gd1-gd2: rel-fundef
norm-eq gd1 gd2
    using step-fun-call rel-fundefs-Some2[OF rel-F1-F2] by blast
  have pc-in-range: pc < length (body fd1)
    using ‹pc < length (body fd2)› rel-fundef-body-length[OF rel-fd1-fd2]
    by simp
  have arity-gd1-gd2: arity gd2 = arity gd1
    using rel-fundef-arities[OF rel-gd1-gd2, symmetric] .
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
  proof −
    let ?Σg = take (arity gd1) Σ
    let ?s1' = State F1 H (Frame g 0 ?Σg # Frame f pc Σ # st')
    have ?STEP ?s1'
      using step-fun-call pc-in-range
      using ‹Fstd-get F1 f = Some fd1› ‹Fstd-get F1 g = Some gd1›
      using rel-fundef-body-nth[OF rel-fd1-fd2 pc-in-range] arity-gd1-gd2
      by (auto intro: Sstd.step-fun-call)
    moreover have ?MATCH ?s1'
      using step-fun-call rel-F1-F2 arity-gd1-gd2 by (auto intro: match.intros)
```

```
    ultimately show ?thesis by auto
   qed
  next
   case (step-fun-end g gd2 Σ_f pc_g frame_g Σ_g frame_f f pc_f frame_f' st')
  then obtain gd1 where 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 arity-gd1-gd2: arity gd2 = arity gd1
    using rel-fundef-arities[OF rel-gd1-gd2, symmetric] .
   show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
   proof −
    let ?s1' = State F1 H (Frame f (Suc pc_f) (Σ_g @ drop (arity gd1) Σ_f) # st')
    have ?STEP ?s1'
     using step-fun-end ‹Fstd-get F1 g = Some gd1›
     using rel-fundef-body-length[OF rel-gd1-gd2] arity-gd1-gd2
     by (auto intro: Sstd.step-fun-end)
    moreover have ?MATCH ?s1'
     using step-fun-end rel-F1-F2 arity-gd1-gd2 by (auto intro: match.intros)
    ultimately show ?thesis by auto
   qed
  qed
qed

lemma match-final-backward:
  s1 ∼ s2 ⟹ Sinca.final s2 ⟹ Sstd.final s1
proof (induction s1 s2 rule: match.induct)
  case (1 F1 F2 H st)
  then obtain f fd2 pc Σ where
   st-def: st = [Frame f pc Σ] and
   Finca-get F2 f = Some fd2 and
   pc-def: pc = length (body fd2)
   by (auto elim: Sinca.final.cases)
  then obtain fd1 where Fstd-get F1 f = Some fd1 and rel-fundef norm-eq fd1
fd2
   using 1 rel-fundefs-Some2 by fast
  then show ?case
   unfolding st-def
   using pc-def rel-fundef-body-length[OF ‹rel-fundef norm-eq fd1 fd2›]
   by (auto intro: Sstd.final.intros)
qed

sublocale std-inca-simulation:
   backward-simulation Sstd.step Sinca.step Sstd.final Sinca.final λ- -. False λ-.
match
  using match-final-backward backward-lockstep-simulation
   lockstep-to-plus-backward-simulation[of match Sinca.step Sstd.step]
  by unfold-locales auto
```

35 Forward simulation

lemma *forward-lockstep-simulation*:
assumes *Sstd.step s1 s1′* **and** *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**
rel-F1-F2: *rel-fundefs* (*Fstd-get F1*) (*Finca-get F2*)
by (*auto elim*: *match.cases*)
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 fd1 pc d* Σ *st′*)
then obtain *fd2* **where** *Finca-get F2 f* = *Some fd2* **and** *rel-fundef norm-eq fd1 fd2*
using *rel-fundefs-Some1*[*OF rel-F1-F2*] **by** *blast*
show *?case* (**is** ∃ *x*. *?STEP x* ∧ *?MATCH x*)
proof −
let *?s1′* = *State F2 H* (*Frame f* (*Suc pc*) (*d* # Σ) # *st′*)
have *?STEP ?s1′*
using *step-push* ‹*Finca-get F2 f* = *Some fd2*› ‹*rel-fundef norm-eq fd1 fd2*›
by (*auto intro*!: *Sinca.step-push elim*!: *norm-instr.elims simp*: *rel-fundef-body-nth*)
moreover have *?MATCH ?s1′*
using *rel-F1-F2* **by** (*auto intro*: *match.intros*)
ultimately show *?thesis* **by** *blast*
qed
next
case (*step-pop f fd1 pc d* Σ *st′*)
then obtain *fd2* **where** *Finca-get F2 f* = *Some fd2* **and** *rel-fundef norm-eq fd1 fd2*
using *rel-fundefs-Some1*[*OF rel-F1-F2*] **by** *blast*
show *?case* (**is** ∃ *x*. *?STEP x* ∧ *?MATCH x*)
proof −
let *?s1′* = *State F2 H* (*Frame f* (*Suc pc*) Σ # *st′*)
have *?STEP ?s1′*
using *step-pop* ‹*Finca-get F2 f* = *Some fd2*› ‹*rel-fundef norm-eq fd1 fd2*›
by (*auto intro*!: *Sinca.step-pop elim*!: *norm-instr.elims simp*: *rel-fundef-body-nth*)
moreover have *?MATCH ?s1′*
using *rel-F1-F2* **by** (*auto intro*: *match.intros*)
ultimately show *?thesis* **by** *blast*
qed
next
case (*step-load f fd1 pc x y d* Σ *st′*)
then obtain *fd2* **where** *Finca-get F2 f* = *Some fd2* **and** *rel-fundef norm-eq fd1 fd2*
using *rel-fundefs-Some1*[*OF rel-F1-F2*] **by** *blast*
show *?case* (**is** ∃ *x*. *?STEP x* ∧ *?MATCH x*)

```
    proof −
      let ?s1' = State F2 H (Frame f (Suc pc) (d # Σ) # st')
      have ?STEP ?s1'
        using step-load ‹Finca-get F2 f = Some fd2› ‹rel-fundef norm-eq fd1 fd2›
       by (auto intro!: Sinca.step-load elim!: norm-instr.elims simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1'
        using rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by blast
    qed
  next
    case (step-store f fd1 pc x y d H' Σ st')
     then obtain fd2 where Finca-get F2 f = Some fd2 and rel-fundef norm-eq
fd1 fd2
      using rel-fundefs-Some1[OF rel-F1-F2] by blast
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F2 H' (Frame f (Suc pc) Σ # st')
      have ?STEP ?s1'
        using step-store ‹Finca-get F2 f = Some fd2› ‹rel-fundef norm-eq fd1 fd2›
       by (auto intro!: Sinca.step-store elim!: norm-instr.elims simp: rel-fundef-body-nth)
      moreover have ?MATCH ?s1'
        using rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by blast
    qed
  next
    case (step-op f fd1 pc op ar Σ x st')
     then obtain fd2 where F2-get-f[intro]: Finca-get F2 f = Some fd2 and
rel-fundef norm-eq fd1 fd2
      using rel-fundefs-Some1[OF rel-F1-F2] by blast
    have pc-in-range[intro]: pc < length (body fd2)
      using step-op ‹rel-fundef norm-eq fd1 fd2› rel-fundef-body-length by fastforce
    have norm-body2: norm-instr (body fd2 ! pc) = Std.IOp op
      using step-op rel-fundef-body-nth[OF ‹rel-fundef norm-eq fd1 fd2›, of pc]
      by simp
    then show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof (cases body fd2 ! pc)
      case (IOp op')
      then have op' = op using norm-body2 by simp
      show ?thesis
      proof (cases ℑnl op' (take ar Σ))
        case None
        let ?s2' = State F2 H (Frame f (Suc pc) (x # drop ar Σ) # st')
        have ?STEP ?s2'
          using None IOp step-op ‹op' = op›
          by (auto intro: Sinca.step-op)
        moreover have ?MATCH ?s2'
          using rel-F1-F2 by (auto intro: match.intros)
        ultimately show ?thesis by auto
      next
```

```
      case (Some opinl)
      let ?fd2' = rewrite-fundef-body fd2 pc (IOpInl opinl)
      let ?s2' = State (Finca-add F2 f ?fd2') H (Frame f (Suc pc) (x # drop ar
Σ) # st')
      have ?STEP ?s2'
        using Some IOp step-op ‹op' = op›
        using Sinca.𝔍𝔫𝔩𝔒𝔭-correct Sinca.𝔍𝔫𝔩-invertible
        by (auto intro: Sinca.step-op-inl)
      moreover have ?MATCH ?s2'
        using Some IOp Sinca.𝔍𝔫𝔩-invertible
        by (auto intro!: match.intros rel-fundefs-rewrite[OF rel-F1-F2])
      ultimately show ?thesis by auto
    qed
  next
    case (IOpInl op')
    then have 𝔇𝔢𝔍𝔫𝔩 op' = op using norm-body2 by simp
    show ?thesis
    proof (cases 𝔍𝔰𝔍𝔫𝔩 op' (take ar Σ))
      case True
      let ?s2' = State F2 H (Frame f (Suc pc) (x # drop ar Σ) # st')
      have ?STEP ?s2'
        using True IOpInl step-op ‹𝔇𝔢𝔍𝔫𝔩 op' = op› Sinca.𝔍𝔫𝔩𝔒𝔭-correct
        by (auto intro!: Sinca.step-op-inl-hit)
      moreover have ?MATCH ?s2'
        using rel-F1-F2 by (auto intro: match.intros)
      ultimately show ?thesis by auto
    next
      case False
      let ?fd2' = rewrite-fundef-body fd2 pc (IOp (𝔇𝔢𝔍𝔫𝔩 op'))
      let ?s2' = State (Finca-add F2 f ?fd2') H (Frame f (Suc pc) (x # drop ar
Σ) # st')
      have ?STEP ?s2'
        using False IOpInl step-op ‹𝔇𝔢𝔍𝔫𝔩 op' = op› Sinca.𝔍𝔫𝔩𝔒𝔭-correct
        by (auto intro!: Sinca.step-op-inl-miss)
      moreover have ?MATCH ?s2'
        using IOpInl
        by (auto intro!: match.intros intro: rel-fundefs-rewrite[OF rel-F1-F2])
      ultimately show ?thesis by auto
    qed
  qed simp-all
next
  case (step-cjump-true f fd1 pc n d Σ st')
    then obtain fd2 where F2-get-f[intro]: Finca-get F2 f = Some fd2 and
rel-fundef norm-eq fd1 fd2
    using rel-fundefs-Some1[OF rel-F1-F2] by blast
  have pc-in-range[intro]: pc < length (body fd2)
  using ‹rel-fundef norm-eq fd1 fd2› rel-fundef-body-length step-cjump-true.hyps(2)
by fastforce
  have norm-body2: norm-instr (body fd2 ! pc) = Std.ICJump n
```

```
    using step-cjump-true rel-fundef-body-nth[OF ‹rel-fundef norm-eq fd1 fd2›, of
pc]
    by simp
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
  proof −
    let ?s1' = State F2 H (Frame f n Σ # st')
    have ?STEP ?s1'
      using step-cjump-true norm-body2
      by (auto intro: Sinca.step-cjump-true elim: norm-instr.elims)
    moreover have ?MATCH ?s1'
      using step-cjump-true rel-F1-F2 by (auto intro: match.intros)
    ultimately show ?thesis by blast
  qed
 next
  case (step-cjump-false f fd1 pc n d Σ st')
   then obtain fd2 where F2-get-f[intro]: Finca-get F2 f = Some fd2 and
rel-fundef norm-eq fd1 fd2
    using rel-fundefs-Some1[OF rel-F1-F2] by blast
  have pc-in-range[intro]: pc < length (body fd2)
    using ‹rel-fundef norm-eq fd1 fd2› rel-fundef-body-length step-cjump-false by
fastforce
  have norm-body2: norm-instr (body fd2 ! pc) = Std.ICJump n
   using step-cjump-false rel-fundef-body-nth[OF ‹rel-fundef norm-eq fd1 fd2›, of
pc]
    by simp
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
  proof −
    let ?s1' = State F2 H (Frame f (Suc pc) Σ # st')
    have ?STEP ?s1'
      using step-cjump-false norm-body2
      by (auto intro: Sinca.step-cjump-false elim: norm-instr.elims)
    moreover have ?MATCH ?s1'
      using step-cjump-false rel-F1-F2 by (auto intro: match.intros)
    ultimately show ?thesis by blast
  qed
 next
  case (step-fun-call f fd1 pc g gd1 Σ frame_f frame_g st')
   then obtain fd2 where Finca-get F2 f = Some fd2 and rel-fundef norm-eq
fd1 fd2
    using rel-fundefs-Some1[OF rel-F1-F2] by blast
   obtain gd2 where Finca-get F2 g = Some gd2 and rel-gd1-gd2: rel-fundef
norm-eq gd1 gd2
    using step-fun-call rel-fundefs-Some1[OF rel-F1-F2] by blast
  have pc-in-range: pc < length (body fd2)
     using step-fun-call ‹rel-fundef norm-eq fd1 fd2› rel-fundef-body-length by
fastforce
  have arity-gd1-gd2: arity gd1 = arity gd2
    using rel-fundef-arities[OF rel-gd1-gd2] .
  show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
```

```
    proof −
      let ?Σg = take (arity gd2) Σ
      let ?s2' = State F2 H (Frame g 0 ?Σg # Frame f pc Σ # st')
      have ?STEP ?s2'
      proof −
        have arity gd2 ≤ length Σ
          using ⟨arity gd1 ≤ length Σ⟩ arity-gd1-gd2 by simp
        moreover have body fd2 ! pc = Inca.ICall g
          using step-fun-call rel-fundef-body-nth[OF ⟨rel-fundef norm-eq fd1 fd2⟩, of
pc]
          by (auto elim: norm-instr.elims)
        ultimately show ?thesis
          using step-fun-call
          using ⟨Finca-get F2 g = Some gd2⟩ ⟨Finca-get F2 f = Some fd2⟩
          using pc-in-range ⟨body fd2 ! pc = Inca.ICall g⟩
          by (auto intro: Sinca.step-fun-call)
      qed
      moreover have ?MATCH ?s2'
        using step-fun-call rel-F1-F2 arity-gd1-gd2 by (auto intro: match.intros)
      ultimately show ?thesis by auto
    qed
  next
    case (step-fun-end g gd1 Σ_f pc_g frame_g Σ_g frame_f f pc_f frame_f' st')
   then obtain gd2 where 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 pc-at-end: pc_g = length (body gd2)
      using rel-fundef-body-length[OF rel-gd1-gd2] step-fun-end by fastforce
    have arity-gd1-gd2: arity gd1 = arity gd2
      using rel-fundef-arities[OF rel-gd1-gd2] .
    show ?case (is ∃ x. ?STEP x ∧ ?MATCH x)
    proof −
      let ?s1' = State F2 H (Frame f (Suc pc_f) (Σ_g @ drop (arity gd2) Σ_f) # st')
      have ?STEP ?s1'
      proof −
        have arity gd2 ≤ length Σ_f
          using ⟨arity gd1 ≤ length Σ_f⟩ arity-gd1-gd2 by simp
        then show ?thesis
          using step-fun-end pc-at-end ⟨Finca-get F2 g = Some gd2⟩
          by (auto intro: Sinca.step-fun-end)
      qed
      moreover have ?MATCH ?s1'
        using step-fun-end rel-F1-F2 arity-gd1-gd2 by (auto intro: match.intros)
      ultimately show ?thesis by auto
    qed
  qed
qed

lemma forward-match-final:
```

s1 ∼ *s2* ⟹ *Sstd.final s1* ⟹ *Sinca.final s2*
proof (*induction s1 s2 rule*: *match.induct*)
case (*1 F1 F2 H st*)
then obtain *f fd1 pc* Σ **where**
st-def: *st* = [*Frame f pc* Σ] **and**
Fstd-get F1 f = *Some fd1* **and**
pc-def: *pc* = *length* (*body fd1*)
by (*auto elim*: *Sstd.final.cases*)
then obtain *fd2* **where** *Finca-get F2 f* = *Some fd2* **and** *rel-fundef norm-eq fd1 fd2*
using *1 rel-fundefs-Some1* **by** *fast*
then show *?case*
unfolding *st-def*
using *pc-def rel-fundef-body-length*[*OF* ‹*rel-fundef norm-eq fd1 fd2*›]
by (*auto intro*: *Sinca.final.intros*)
qed

sublocale *std-inca-forward-simulation*:
forward-simulation Sstd.step Sinca.step Sstd.final Sinca.final λ- -. *False* λ-. *match*
using *forward-match-final forward-lockstep-simulation*
lockstep-to-plus-forward-simulation[*of match Sstd.step - Sinca.step*]
by *unfold-locales auto*

36 Bisimulation

sublocale *std-inca-bisimulation*:
bisimulation Sstd.step Sinca.step Sstd.final Sinca.final λ- -. *False* λ-. *match*
by *unfold-locales*

end

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

fun *compile-instr* **where**
compile-instr (*Std.IPush d*) = *Inca.IPush d* |
compile-instr Std.IPop = *Inca.IPop* |
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 n*) = *Inca.ICJump n* |
compile-instr (*Std.ICall f*) = *Inca.ICall f*

fun *compile-fundef* **where**
compile-fundef (*Fundef xs ar*) = *Fundef* (*map compile-instr xs*) *ar*

context *std-inca-simulation* **begin**

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

lemma *rel-compile-fundef*: *rel-fundef norm-eq fd* (*compile-fundef fd*)
proof (*cases fd*)
case (*Fundef xs ar*)
then show *?thesis*
by (*simp add*: *list.rel-map list.rel-eq norm-compile-instr*)
qed

definition *compile-env* **where**
compile-env e = *Sinca.Fenv.from-list* (*map* (*map-prod id compile-fundef*) (*Fstd-to-list e*))

lemma *Finca-get-compile*: *Finca-get* (*compile-env e*) *x* = *map-option compile-fundef* (*Fstd-get e x*)
using *Sinca.Fenv.from-list-correct*[*symmetric*] *Sstd.Fenv.to-list-correct*
by (*simp add*: *compile-env-def map-prod-def map-of-map*)

lemma *rel-fundefs-compile-env*: *rel-fundefs* (*Fstd-get e*) (*Finca-get* (*compile-env e*))
unfolding *rel-fundefs-def*
unfolding *Finca-get-compile*
unfolding *option.rel-map*
by (*auto intro*: *option.rel-refl rel-compile-fundef*)

fun *compile* **where**
compile (*Prog F H main*) = *Some* (*Prog* (*compile-env F*) *H main*)

theorem *compile-load*:
assumes *compile p1* = *Some p2* **and** *Sinca.load p2 s2*
shows ∃ *s1*. *Sstd.load p1 s1* ∧ *match s1 s2*
proof (*cases p1*)
case (*Prog F H main*)
then have *p2-def*: *p2* = *Prog* (*compile-env F*) *H main*
using *assms*(*1*) **by** *simp*

show *?thesis*
using *assms*(*2*) **unfolding** *p2-def*
proof (*cases - s2 rule*: *Sinca.load.cases*)
case (*1 fd'*)
let *?s1* = *State F H* [*Frame main 0* []]

from *1* **obtain** *fd* **where**
Fstd-main: *Fstd-get F main* = *Some fd* **and** *fd'* = *compile-fundef fd*
using *Finca-get-compile* **by** *auto*

```
    with 1 have arity fd = 0
      by (metis fundef.rel-sel rel-compile-fundef)
    hence Sstd.load p1 ?s1
      unfolding Prog
      by (auto intro: Sstd.load.intros[OF Fstd-main])
    moreover have ?s1 ∼ s2
      unfolding 1
      using rel-fundefs-compile-env
      by (auto intro!: match.intros)
    ultimately show ?thesis by auto
  qed
qed

sublocale std-to-inca-compiler:
  compiler Sstd.step Sinca.step Sstd.final Sinca.final Sstd.load Sinca.load
    λ- -. False λ-. match compile
using compile-load
  by unfold-locales auto

theorem compile-complete:
  assumes Sstd.load p1 s1
  shows ∃ p2 s2. compile p1 = Some p2 ∧ Sinca.load p2 s2 ∧ match s1 s2
  using assms
  by (auto elim!: Sstd.load.cases
      intro!: match.intros rel-fundefs-compile-env
    simp add: Sinca.load.simps Finca-get-compile rel-fundef-arities[OF rel-compile-fundef])

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.