Interpreter_Optimizations

Martin Desharnais

June 17, 2024

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 HOL−Library.Library*
begin

1 Generic lemmas

lemma *map-of-list-allI*:
assumes $\bigwedge k\ v.\ f\ k = Some\ v \Longrightarrow P\ (k, v)$ **and**
$\bigwedge k\ v.\ map\text{-}of\ kvs\ k = Some\ v \Longrightarrow f\ k = Some\ v$ **and**
distinct (*map fst kvs*)
shows *list-all P kvs*
⟨*proof*⟩

2 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* **and**
to-list-distinct: *distinct* (*map fst* (*to-list e*))

begin

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

definition *singleton* **where**
singleton ≡ *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*: *get* (*from-list xs*) = *map-of xs*
⟨*proof*⟩

lemma *from-list-Nil*[*simp*]: *from-list* [] = *empty*
⟨*proof*⟩

lemma *get-from-list-to-list*: *get* (*from-list* (*to-list e*)) = *get e*
⟨*proof*⟩

lemma *to-list-list-allI*:
assumes ⋀*k v*. *get e k* = *Some v* ⟹ *P* (*k*, *v*)
shows *list-all P* (*to-list e*)
⟨*proof*⟩

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

lemma *get-map-entry-eq*[*simp*]: *get* (*map-entry e k f*) *k* = *map-option f* (*get e k*)
⟨*proof*⟩

lemma *get-map-entry-neq*[*simp*]: *x* ≠ *y* ⟹ *get* (*map-entry e x f*) *y* = *get e y*

⟨*proof*⟩

lemma *dom-map-entry*[*simp*]: *dom* (*get* (*map-entry e k f*)) = *dom* (*get e*)
⟨*proof*⟩

lemma *get-map-entry-conv*:
get (*map-entry e x f*) *y* = *map-option* (λ*v*. *if x* = *y then f v else v*) (*get e y*)
⟨*proof*⟩

lemma *map-option-comp-map-entry*:
assumes ∀ *x* ∈ *ran* (*get e*). *f* (*g x*) = *f x*
shows *map-option f* ∘ *get* (*map-entry e k g*) = *map-option f* ∘ *get e*
⟨*proof*⟩

lemma *map-option-comp-get-add*:
assumes *k* ∈ *dom* (*get e*) **and** ∀ *x* ∈ *ran* (*get e*). *f v* = *f x*
shows *map-option f* ∘ *get* (*add e k v*) = *map-option f* ∘ *get e*
⟨*proof*⟩

end

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

2.1 Generic lemmas

lemma *map-of-filter*:
x ≠ *y* ⟹ *map-of* (*filter* (λ*z*. *fst z* ≠ *y*) *zs*) *x* = *map-of zs x*
⟨*proof*⟩

2.2 List-based implementation of environment

context
begin

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

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

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

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

term *filter*

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

lemma *get-empty*: *get empty x* = *None*
  ⟨*proof*⟩

lemma *get-add-eq*: *get* (*add e x v*) *x* = *Some v*
  ⟨*proof*⟩

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

lemma *to-list-correct*: *map-of* (*to-list e*) = *get e*
  ⟨*proof*⟩

lemma *set-to-list*: *set* (*to-list e*) ⊆ *set e*
  ⟨*proof*⟩

lemma *to-list-distinct*: *distinct* (*map fst* (*to-list e*))
⟨*proof*⟩

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*
  ⟨*proof*⟩

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* ⟨*proof*⟩

lemma *same-length-iff-eq-lengths*: *same-length xs ys* $\longleftrightarrow$ *length xs* = *length ys*
⟨*proof*⟩

lemma *same-length-Cons*:
same-length (*x* # *xs*) *ys* $\Longrightarrow$ $\exists$ *y ys'*. *ys* = *y* # *ys'*
same-length xs (*y* # *ys*) $\Longrightarrow$ $\exists$ *x xs'*. *xs* = *x* # *xs'*
⟨*proof*⟩

3 nth_opt

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

lemma *nth-opt-eq-Some-conv*: *nth-opt xs n* = *Some x* $\longleftrightarrow$ *n* < *length xs* $\wedge$ *xs* ! *n* = *x*
⟨*proof*⟩

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

4 Generic lemmas

lemma *list-rel-imp-pred1*:
assumes
list-all2 R xs ys **and**
$\bigwedge$*x y*. (*x*, *y*) ∈ *set* (*zip xs ys*) $\Longrightarrow$ *R x y* $\Longrightarrow$ *P x*
shows *list-all P xs*
⟨*proof*⟩

lemma *list-rel-imp-pred2*:
assumes
list-all2 R xs ys **and**
$\bigwedge$*x y*. (*x*, *y*) ∈ *set* (*zip xs ys*) $\Longrightarrow$ *R x y* $\Longrightarrow$ *P y*
shows *list-all P ys*
⟨*proof*⟩

lemma *eq-append-conv-conj*: (*zs* = *xs* @ *ys*) = (*xs* = *take* (*length xs*) *zs* $\wedge$ *ys* = *drop* (*length xs*) *zs*)
⟨*proof*⟩

lemma *list-all-list-updateI*: *list-all P xs* $\Longrightarrow$ *P x* $\Longrightarrow$ *list-all P* (*list-update xs n x*)
⟨*proof*⟩

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

lemma *map-list-update-id*:
f (*xs* ! *pc*) = *f instr* ⟹ *map f* (*xs*[*pc* := *instr*]) = *map f xs*
⟨*proof*⟩

lemma *list-all-eq-const-imp-replicate*:
assumes *list-all* (λ*x*. *x* = *y*) *xs*
shows *xs* = *replicate* (*length xs*) *y*
⟨*proof*⟩

lemma *list-all-eq-const-imp-replicate′*:
assumes *list-all* ((=) *y*) *xs*
shows *xs* = *replicate* (*length xs*) *y*
⟨*proof*⟩

lemma *list-all-eq-const-replicate-lhs*[*intro*]:
list-all (λ*x*. *y* = *x*) (*replicate n y*)
⟨*proof*⟩

lemma *list-all-eq-const-replicate-rhs*[*intro*]:
list-all (λ*x*. *x* = *y*) (*replicate n y*)
⟨*proof*⟩

lemma *list-all-eq-const-replicate*[*simp*]: *list-all* ((=) *c*) (*replicate n c*)
⟨*proof*⟩

lemma *replicate-eq-map*:
assumes *n* = *length xs* **and** ⋀*y*. *y* ∈ *set xs* ⟹ *f y* = *x*
shows *replicate n x* = *map f xs*
⟨*proof*⟩

lemma *replicate-eq-impl-Ball-eq*:
shows *replicate n c* = *xs* ⟹ (∀ *x* ∈ *set xs*. *x* = *c*)
⟨*proof*⟩

lemma *rel-option-map-of*:
assumes *list-all2* (*rel-prod* (=) *R*) *xs ys*
shows *rel-option R* (*map-of xs l*) (*map-of ys l*)
⟨*proof*⟩

lemma *list-all2-rel-prod-nth*:
assumes *list-all2* (*rel-prod R1 R2*) *xs ys* **and** *n* < *length xs*
shows *R1* (*fst* (*xs* ! *n*)) (*fst* (*ys* ! *n*)) ∧ *R2* (*snd* (*xs* ! *n*)) (*snd* (*ys* ! *n*))
⟨*proof*⟩

lemma *list-all2-rel-prod-fst-hd*:
assumes *list-all2* (*rel-prod R1 R2*) *xs ys* **and** *xs* ≠ [] ∨ *ys* ≠ []
shows *R1* (*fst* (*hd xs*)) (*fst* (*hd ys*)) ∧ *R2* (*snd* (*hd xs*)) (*snd* (*hd ys*))
⟨*proof*⟩

lemma *list-all2-rel-prod-fst-last*:
assumes *list-all2* (*rel-prod R1 R2*) *xs ys* **and** $xs \neq [] \vee ys \neq []$
shows *R1* (*fst* (*last xs*)) (*fst* (*last ys*)) $\wedge$ *R2* (*snd* (*last xs*)) (*snd* (*last ys*))
⟨*proof*⟩

lemma *list-all-nthD*[*intro*]: *list-all P xs* $\Longrightarrow$ $n <$ *length xs* $\Longrightarrow$ *P* (*xs* ! *n*)
⟨*proof*⟩

lemma *list-all P xs* $\Longrightarrow$ $\forall x \in$ *set xs*. *P x*
⟨*proof*⟩

lemma *list-all-map-of-SomeD*:
assumes *list-all P kvs* **and** *map-of kvs k* = *Some v*
shows *P* (*k*, *v*)
⟨*proof*⟩

lemma *list-all-not-nthD*:*list-all P xs* $\Longrightarrow$ $\neg$ *P* (*xs* ! *n*) $\Longrightarrow$ *length xs* $\leq$ *n*
⟨*proof*⟩

lemma *list-all-butlast-not-nthD*: *list-all P* (*butlast xs*) $\Longrightarrow$ $\neg$ *P* (*xs* ! *n*) $\Longrightarrow$ *length xs* $\leq$ *Suc n*
⟨*proof*⟩

lemma *list-all-replicateI*[*intro*]: *P x* $\Longrightarrow$ *list-all P* (*replicate n x*)
⟨*proof*⟩

lemma *map-eq-append-replicate-conv*:
assumes *map f xs* = *replicate n x* @ *ys*
shows *map f* (*take n xs*) = *replicate n x*
⟨*proof*⟩

lemma *map-eq-replicate-imp-list-all-const*:
map f xs = *replicate n x* $\Longrightarrow$ *n* = *length xs* $\Longrightarrow$ *list-all* (λy. *f y* = *x*) *xs*
⟨*proof*⟩

lemma *map-eq-replicateI*: *length xs* = *n* $\Longrightarrow$ ($\bigwedge x$. *x* $\in$ *set xs* $\Longrightarrow$ *f x* = *c*) $\Longrightarrow$ *map f xs* = *replicate n c*
⟨*proof*⟩

lemma *list-all-dropI*[*intro*]: *list-all P xs* $\Longrightarrow$ *list-all P* (*drop n xs*)
⟨*proof*⟩

5 Non-empty list

type-synonym *'a nlist* = *'a* × *'a list*

end

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

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

6 Monadic bind

context begin

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

end

adhoc-overloading
bind Result.bind

context begin

qualified lemma *bind-Ok*[*simp*]: *x* ⤜ *Ok* = *x*
⟨*proof*⟩ **lemma** *bind-eq-Ok-conv*: (*x* ⤜ *f* = *Ok z*) = (∃ *y*. *x* = *Ok y* ∧ *f y* = *Ok z*)
⟨*proof*⟩ **lemmas** *bind-eq-OkD*[*dest*!] = *bind-eq-Ok-conv*[*THEN iffD1*]
qualified lemmas *bind-eq-OkE* = *bind-eq-OkD*[*THEN exE*]
qualified lemmas *bind-eq-OkI*[*intro*] = *conjI*[*THEN exI*[*THEN bind-eq-Ok-conv*[*THEN iffD2*]]]

qualified lemma *bind-eq-Error-conv*:
x ⤜ *f* = *Error z* ⟷ *x* = *Error z* ∨ (∃ *y*. *x* = *Ok y* ∧ *f y* = *Error z*)
⟨*proof*⟩ **lemmas** *bind-eq-ErrorD*[*dest*!] = *bind-eq-Error-conv*[*THEN iffD1*]
qualified lemmas *bind-eq-ErrorE* = *bind-eq-ErrorD*[*THEN disjE*]
qualified lemmas *bind-eq-ErrorI* =
disjI1[*THEN bind-eq-Error-conv*[*THEN iffD2*]]
conjI[*THEN exI*[*THEN disjI2*[*THEN bind-eq-Error-conv*[*THEN iffD2*]]]]

lemma *if-then-else-Ok*[*simp*]:
(*if a then b else Error c*) = *Ok d* ⟷ *a* ∧ *b* = *Ok d*
(*if a then Error c else b*) = *Ok d* ⟷ ¬ *a* ∧ *b* = *Ok d*
⟨*proof*⟩ **lemma** *if-then-else-Error*[*simp*]:
(*if a then Ok b else c*) = *Error d* ⟷ ¬ *a* ∧ *c* = *Error d*
(*if a then c else Ok b*) = *Error d* ⟷ *a* ∧ *c* = *Error d*
⟨*proof*⟩ **lemma** *map-eq-Ok-conv*: *map-result f g x* = *Ok y* ⟷ (∃ *x*′. *x* = *Ok x*′ ∧ *y* = *g x*′)

⟨*proof*⟩ **lemma** *map-eq-Error-conv*: *map-result f g x* = *Error y* ⟷ (∃ *x'*. *x* = *Error x'* ∧ *y* = *f x'*)
⟨*proof*⟩ **lemmas** *map-eq-OkD*[*dest*!] = *iffD1*[*OF map-eq-Ok-conv*]
qualified lemmas *map-eq-ErrorD*[*dest*!] = *iffD1*[*OF map-eq-Error-conv*]

end

7 Conversion functions

context begin

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

qualified lemma *of-option-injective*[*simp*]: (*of-option e x* = *of-option e y*) = (*x* = *y*)
⟨*proof*⟩ **lemma** *of-option-eq-Ok*[*simp*]: (*of-option x y* = *Ok z*) = (*y* = *Some z*)
⟨*proof*⟩ **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*)
⟨*proof*⟩ **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'*
⟨*proof*⟩

end

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

fun *ap-option* (**infixl** ⋄ *60*) **where**
(*Some f*) ⋄ (*Some x*) = *Some* (*f x*) |
- ⋄ - = *None*

lemma *ap-option-eq-Some-conv*: *f* ⋄ *x* = *Some y* ⟷ (∃ *f' x'*. *f* = *Some f'* ∧ *x* = *Some x'* ∧ *y* = *f' x'*)
⟨*proof*⟩

definition *ap-map-prod* **where**
ap-map-prod f g p ≡ Some Pair ⋄ f (fst p) ⋄ g (snd p)

lemma *ap-map-prod-eq-Some-conv*:
ap-map-prod f g p = Some p′ ⟷ (∃ x y. p = (x, y) ∧ (∃ x′ y′. p′ = (x′, y′) ∧ f x = Some x′ ∧ g y = Some y′))
⟨*proof*⟩

fun *ap-map-list* :: (′*a* ⇒ ′*b option*) ⇒ ′*a list* ⇒ ′*b list option* **where**
ap-map-list - [] = Some [] |
ap-map-list f (x # xs) = Some (#) ⋄ f x ⋄ ap-map-list f xs

lemma *length-ap-map-list*: *ap-map-list f xs = Some ys ⟹ length ys = length xs*
⟨*proof*⟩

lemma *ap-map-list-imp-rel-option-map-of*:
assumes *ap-map-list f xs = Some ys* **and**
⋀x y. (x, y) ∈ set (zip xs ys) ⟹ f x = Some y ⟹ fst x = fst y
shows *rel-option* (λ*x y. f* (*k*, *x*) = *Some* (*k*, *y*)) (*map-of xs k*) (*map-of ys k*)
⟨*proof*⟩

lemma *ap-map-list-ap-map-prod-imp-rel-option-map-of*:
assumes *ap-map-list* (*ap-map-prod Some f*) *xs* = *Some ys*
shows *rel-option* (λ*x y. f x* = *Some y*) (*map-of xs k*) (*map-of ys k*)
⟨*proof*⟩

lemma *ex-ap-map-list-eq-SomeI*:
assumes *list-all* (λ*x*. ∃ *y*. *f x* = *Some y*) *xs*
shows ∃ *ys*. *ap-map-list f xs* = *Some ys*
⟨*proof*⟩

lemma *ap-map-list-iff-list-all2*:
ap-map-list f xs = Some ys ⟷ list-all2 (λ*x y. f x* = *Some y*) *xs ys*
⟨*proof*⟩

lemma *ap-map-list-map-conv*:
assumes
ap-map-list f xs = Some ys **and**
⋀x y. x ∈ set xs ⟹ f x = Some y ⟹ y = g x
shows *ys* = *map g xs*
⟨*proof*⟩

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

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

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

lemma *very-weak-map-of-SomeI*: $k \in fst\ `\ set\ kvs \Longrightarrow \exists\, v.\ map\text{-}of\ kvs\ k = Some\ v$
⟨*proof*⟩

lemma *map-of-fst-hd-neq-Nil*[*simp*]:
assumes $xs \neq []$
shows $map\text{-}of\ xs\ (fst\ (hd\ xs)) = Some\ (snd\ (hd\ xs))$
⟨*proof*⟩

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

lemma *option-case-cancel*[*simp*]: (*case opt of None* ⇒ *x* | *Some* - ⇒ *x*) = *x*
⟨*proof*⟩

lemma *map-le-map-merge-Some-const*:
$f \subseteq_m map\text{-}merge\ (\lambda x\ y.\ Some\ x)\ f\ g$ **and**
$g \subseteq_m map\text{-}merge\ (\lambda x\ y.\ Some\ y)\ f\ g$
⟨*proof*⟩

8 pred_map

definition *pred-map* **where**
$pred\text{-}map\ P\ m \equiv (\forall\, x\ y.\ m\ x = Some\ y \longrightarrow P\ y)$

lemma *pred-map-get*:
assumes *pred-map P m* **and** *m x* = *Some y*
shows *P y*
⟨*proof*⟩

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

lemma *list-all2-rel-prod-updateI*:
assumes *list-all2* (*rel-prod* (=) *R*) *xs ys* **and** *R xval yval*
shows *list-all2* (*rel-prod* (=) *R*) (*AList.update k xval xs*) (*AList.update k yval ys*)
⟨*proof*⟩

lemma *length-map-entry*[*simp*]: *length* (*AList.map-entry k f al*) = *length al*
⟨*proof*⟩

lemma *map-entry-id0*[*simp*]: *AList.map-entry k id = id*
⟨*proof*⟩

lemma *map-entry-id*: *AList.map-entry k id xs = xs*
⟨*proof*⟩

lemma *map-entry-map-of-Some-conv*:
map-of xs k = Some v ⟹ AList.map-entry k f xs = AList.update k (f v) xs
⟨*proof*⟩

lemma *map-entry-map-of-None-conv*:
map-of xs k = None ⟹ AList.map-entry k f xs = xs
⟨*proof*⟩

lemma *list-all2-rel-prod-map-entry*:
assumes
list-all2 (rel-prod (=) R) xs ys **and**
⋀*xval yval. map-of xs k = Some xval ⟹ map-of ys k = Some yval ⟹ R (f xval) (g yval)*
shows *list-all2 (rel-prod (=) R) (AList.map-entry k f xs) (AList.map-entry k g ys)*
⟨*proof*⟩

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

lemma *list-all-updateI*:
assumes *list-all P xs* **and** *P (k, v)*
shows *list-all P (AList.update k v xs)*
⟨*proof*⟩

lemma *set-update*: *set (AList.update k v xs) ⊆ {(k, v)} ∪ set xs*
⟨*proof*⟩

end
theory *Global*
imports *HOL−Library.Library Result Env List-util Option-Extra Map-Extra AList-Extra*
begin

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

declare *K-record-comp*[*simp*]

lemma *if-then-Some-else-None-eq*[*simp*]:
(if a then Some b else None) = Some c ⟷ a ∧ b = c

(*if a then Some b else None*) = *None* ⟷ ¬ *a*
⟨*proof*⟩

lemma *if-then-else-distributive*: (*if a then f b else f c*) = *f* (*if a then b else c*)
⟨*proof*⟩

9 Rest

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

lemma *list-all2-assoc-map-rel-get*:
assumes *list-all2* (=) (*map fst xs*) (*map fst ys*) **and** *list-all2 R* (*map snd xs*) (*map snd ys*)
shows *rel-option R* (*map-of xs k*) (*map-of ys k*)
⟨*proof*⟩

9.1 Function definition

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

lemma *rel-fundef-arities*: *rel-fundef r1 r2 gd1 gd2* ⟹ *arity gd1* = *arity gd2*
⟨*proof*⟩

lemma *rel-fundef-return*: *rel-fundef R1 R2 gd1 gd2* ⟹ *return gd1* = *return gd2*
⟨*proof*⟩

lemma *rel-fundef-locals*: *rel-fundef R1 R2 gd1 gd2* ⟹ *fundef-locals gd1* = *fundef-locals gd2*
⟨*proof*⟩

lemma *rel-fundef-body-length*[*simp*]:
rel-fundef r1 r2 fd1 fd2 ⟹ *length* (*body fd1*) = *length* (*body fd2*)
⟨*proof*⟩

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

lemma *rel-fundef-funtype*[*simp*]: *rel-fundef R1 R2 fd1 fd2* ⟹ *funtype fd1* = *funtype fd2*
⟨*proof*⟩

lemma *rel-fundef-rel-fst-hd-bodies*:
assumes *rel-fundef R1 R2 fd1 fd2* **and** *body fd1* ≠ [] ∨ *body fd2* ≠ []

shows *R1* (*fst* (*hd* (*body fd1*))) (*fst* (*hd* (*body fd2*)))
⟨*proof*⟩

lemma *map-option-comp-conv*:
assumes
⋀*x*. *rel-option R* (*f x*) (*g x*)
⋀*fd1 fd2*. *fd1* ∈ *ran f* ⟹ *fd2* ∈ *ran g* ⟹ *R fd1 fd2* ⟹ *h fd1* = *i fd2*
shows *map-option h* ∘ *f* = *map-option i* ∘ *g*
⟨*proof*⟩

lemma *map-option-arity-comp-conv*:
assumes (⋀*x*. *rel-option* (*rel-fundef R S*) (*f x*) (*g x*))
shows *map-option arity* ∘ *f* = *map-option arity* ∘ *g*
⟨*proof*⟩

definition *wf-fundef* **where**
wf-fundef fd ⟷ *body fd* ≠ []

lemma *wf-fundef-non-empty-bodyD*[*dest*,*intro*]: *wf-fundef fd* ⟹ *body fd* ≠ []
⟨*proof*⟩

definition *wf-fundefs* **where**
wf-fundefs F ⟷ (∀ *f fd*. *F f* = *Some fd* ⟶ *wf-fundef fd*)

lemma *wf-fundefsI*:
assumes ⋀*f fd*. *F f* = *Some fd* ⟹ *wf-fundef fd*
shows *wf-fundefs F*
⟨*proof*⟩

lemma *wf-fundefsI'*:
assumes ⋀*f*. *pred-option wf-fundef* (*F f*)
shows *wf-fundefs F*
⟨*proof*⟩

lemma *wf-fundefs-imp-wf-fundef*:
assumes *wf-fundefs F* **and** *F f* = *Some fd*
shows *wf-fundef fd*
⟨*proof*⟩

hide-fact *wf-fundefs-def*

9.2 Program

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

definition *wf-prog* **where**
wf-prog Get p ⟷ *wf-fundefs* (*Get* (*prog-fundefs p*))

9.3 Stack frame

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

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

lemma *instr-atD*:
assumes *instr-at fd l pc* = *Some instr*
shows ∃ *instrs*. *map-of* (*body fd*) *l* = *Some instrs* ∧ *pc* < *length instrs* ∧ *instrs* ! *pc* = *instr*
⟨*proof*⟩

lemma *rel-fundef-imp-rel-option-instr-at*:
assumes *rel-fundef* (=) *R fd1 fd2*
shows *rel-option R* (*instr-at fd1 l pc*) (*instr-at fd2 l pc*)
⟨*proof*⟩

definition *next-instr* **where**
next-instr F f label pc ≡ *do* {
fd ← *F f*;
instr-at fd label pc
}

lemma *next-instr-eq-Some-conv*:
next-instr F f l pc = *Some instr* ⟷ (∃ *fd*. *F f* = *Some fd* ∧ *instr-at fd l pc* = *Some instr*)
⟨*proof*⟩

lemma *next-instrD*:
assumes *next-instr F f l pc* = *Some instr*
shows ∃ *fd*. *F f* = *Some fd* ∧ *instr-at fd l pc* = *Some instr*
⟨*proof*⟩

lemma *next-instr-pc-lt-length-instrsI*:
assumes
next-instr F f l pc = *Some instr* **and**
F f = *Some fd* **and**
map-of (*body fd*) *l* = *Some instrs*
shows *pc* < *length instrs*
⟨*proof*⟩

lemma *next-instr-get-map-ofD*:
assumes
next-instr F f l pc = Some instr **and**
F f = Some fd **and**
map-of (body fd) l = Some instrs
shows *pc < length instrs* **and** *instrs ! pc = instr*
⟨*proof*⟩

lemma *next-instr-length-instrs*:
assumes
F f = Some fd **and**
map-of (body fd) label = Some instrs
shows *next-instr F f label (length instrs) = None*
⟨*proof*⟩

lemma *next-instr-take-Suc-conv*:
assumes *next-instr F f l pc = Some instr* **and**
F f = Some fd **and**
map-of (body fd) l = Some instrs
shows *take (Suc pc) instrs = take pc instrs @ [instr]*
⟨*proof*⟩

9.4 Dynamic state

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

definition *state-ok* **where**
state-ok Get s ⟷ *wf-fundefs (Get (state-fundefs s))*

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

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

inductive *load* **for** *F-get Uninitialized* **where**
F-get F main = Some fd ⟹ *arity fd = 0* ⟹ *body fd* ≠ [] ⟹
load F-get Uninitialized (Prog F H main) (State F H [allocate-frame main fd []
Uninitialized])

lemma *length-neq-imp-not-list-all2*:
assumes *length xs* ≠ *length ys*
shows ¬ *list-all2 R xs ys*
⟨*proof*⟩

lemma *list-all2-rel-prod-conv*:
list-all2 (*rel-prod R S*) *xs ys* ⟷
list-all2 R (*map fst xs*) (*map fst ys*) ∧ *list-all2 S* (*map snd xs*) (*map snd ys*)
⟨*proof*⟩

definition *rewrite-fundef-body* ::
('*label*, '*instr*) *fundef* ⇒ '*label* ⇒ *nat* ⇒ '*instr* ⇒ ('*label*, '*instr*) *fundef* **where**
rewrite-fundef-body fd l n instr =
(*case fd of Fundef bblocks ar ret locals* ⇒
Fundef (*AList.map-entry l* (λ*instrs*. *instrs*[*n* := *instr*]) *bblocks*) *ar ret locals*)

lemma *rewrite-fundef-body-cases*[*case-names invalid-label valid-label*]:
assumes
⋀*bs ar ret locals*. *fd* = *Fundef bs ar ret locals* ⟹ *map-of bs l* = *None* ⟹ *P*
⋀*bs ar ret locals instrs*. *fd* = *Fundef bs ar ret locals* ⟹ *map-of bs l* = *Some instrs* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *arity-rewrite-fundef-body*[*simp*]: *arity* (*rewrite-fundef-body fd l pc instr*) = *arity fd*
⟨*proof*⟩

lemma *return-rewrite-fundef-body*[*simp*]: *return* (*rewrite-fundef-body fd l pc instr*) = *return fd*
⟨*proof*⟩

lemma *funtype-rewrite-fundef-body*[*simp*]: *funtype* (*rewrite-fundef-body fd l pc instr'*) = *funtype fd*
⟨*proof*⟩

lemma *length-body-rewrite-fundef-body*[*simp*]:
length (*body* (*rewrite-fundef-body fd l pc instr*)) = *length* (*body fd*)
⟨*proof*⟩

lemma *prod-in-set-fst-image-conv*: (*x*, *y*) ∈ *set xys* ⟹ *x* ∈ *fst* ‘ *set xys*
⟨*proof*⟩

lemma *map-of-body-rewrite-fundef-body-conv-neq*[*simp*]:
assumes *l* ≠ *l'*
shows *map-of* (*body* (*rewrite-fundef-body fd l pc instr*)) *l'* = *map-of* (*body fd*) *l'*
⟨*proof*⟩

lemma *map-of-body-rewrite-fundef-body-conv-eq*[*simp*]:
map-of (*body* (*rewrite-fundef-body fd l pc instr*)) *l* =
map-option (λ*xs*. *xs*[*pc* := *instr*]) (*map-of* (*body fd*) *l*)
⟨*proof*⟩

lemma *instr-at-rewrite-fundef-body-conv*:
instr-at (*rewrite-fundef-body fd l′ pc′ instr′*) *l pc* =
map-option (λ*instr*. *if l* = *l′* ∧ *pc* = *pc′ then instr′ else instr*) (*instr-at fd l pc*)
⟨*proof*⟩

lemma *fundef-locals-rewrite-fundef-body*[*simp*]:
fundef-locals (*rewrite-fundef-body fd l pc instr*) = *fundef-locals fd*
⟨*proof*⟩

lemma *rel-fundef-rewrite-body1*:
assumes
rel-fd1-fd2: *rel-fundef* (=) *R fd1 fd2* **and**
instr-at-l-pc: *instr-at fd1 l pc* = *Some instr* **and**
R-iff: ⋀*x*. *R instr x* ⟷ *R instr′ x*
shows *rel-fundef* (=) *R* (*rewrite-fundef-body fd1 l pc instr′*) *fd2*
⟨*proof*⟩

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

lemma *rewrite-fundef-body-triv*:
instr-at fd l pc = *Some instr* ⟹ *rewrite-fundef-body fd l pc instr* = *fd*
⟨*proof*⟩

lemma *rel-fundef-rewrite-body2*:
assumes
rel-fd1-fd2: *rel-fundef* (=) *R fd1 fd2* **and**
instr-at-l-pc: *instr-at fd2 l pc* = *Some instr* **and**
R-iff: ⋀*x*. *R x instr* ⟷ *R x instr′*
shows *rel-fundef* (=) *R fd1* (*rewrite-fundef-body fd2 l pc instr′*)
⟨*proof*⟩

lemma *rel-fundef-rewrite-body2′*:
assumes
rel-fd1-fd2: *rel-fundef* (=) *R fd1 fd2* **and**
instr-at1: *instr-at fd1 l pc* = *Some instr1* **and**
R-instr1-instr2: *R instr1 instr2′*
shows *rel-fundef* (=) *R fd1* (*rewrite-fundef-body fd2 l pc instr2′*)
⟨*proof*⟩

thm *rel-fundef-rewrite-body*

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

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

⟨*proof*⟩

end
theory *Op*
imports *Main*
begin

10 n-ary operations

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

end
theory *OpInl*
imports *Op*
begin

11 n-ary operations

locale *nary-operations-inl* =
nary-operations $\mathfrak{Op}$ $\mathfrak{Arity}$
for
$\mathfrak{Op}$ :: *'op* ⇒ *'a list* ⇒ *'a* **and** $\mathfrak{Arity}$ +
fixes
$\mathfrak{Inl}\mathfrak{Op}$ :: *'opinl* ⇒ *'a list* ⇒ *'a* **and**
$\mathfrak{Inl}$ :: *'op* ⇒ *'a list* ⇒ *'opinl option* **and**
$\mathfrak{IsInl}$ :: *'opinl* ⇒ *'a list* ⇒ *bool* **and**
$\mathfrak{DeInl}$ :: *'opinl* ⇒ *'op*
assumes
$\mathfrak{Inl}$-*invertible*: $\mathfrak{Inl}$ *op xs* = *Some opinl* ⟹ $\mathfrak{DeInl}$ *opinl* = *op* **and**
$\mathfrak{Inl}\mathfrak{Op}$-*correct*: *length xs* = $\mathfrak{Arity}$ ($\mathfrak{DeInl}$ *opinl*) ⟹ $\mathfrak{Inl}\mathfrak{Op}$ *opinl xs* = $\mathfrak{Op}$ ($\mathfrak{DeInl}$ *opinl*) *xs* **and**
$\mathfrak{Inl}$-$\mathfrak{IsInl}$: $\mathfrak{Inl}$ *op xs* = *Some opinl* ⟹ $\mathfrak{IsInl}$ *opinl xs*

begin

lemma $\mathfrak{Inl}$-*inj-on*: *inj-on* $\mathfrak{Inl}$ { *op* | *op args*. $\mathfrak{Inl}$ *op args* ≠ *None* }
⟨*proof*⟩

abbreviation $\mathfrak{Inl}$-*dom* **where**

$\mathfrak{Inl}$-*dom* $\equiv$ {*op* | *op args*. $\mathfrak{Inl}$ *op args* $\neq$ *None* }

lemma *bij-betw* $\mathfrak{Inl}$ $\mathfrak{Inl}$-*dom* { $\mathfrak{Inl}$ *op* | *op*. *op* $\in$ $\mathfrak{Inl}$-*dom*}
⟨*proof*⟩

end

end
theory *Dynamic*
imports *Main*
begin

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

lemma *false-not-true*: *is-false d* ⟹ ¬ *is-true d*
⟨*proof*⟩

lemma *true-not-false*: *is-true d* ⟹ ¬ *is-false d*
⟨*proof*⟩

lemma *is-true-and-is-false-implies-False*: *is-true d* ⟹ *is-false d* ⟹ *False*
⟨*proof*⟩

end

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

12 Inline caching

12.1 Static representation

datatype (*'dyn*, *'var*, *'fun*, *'label*, *'op*, *'opinl*) *instr* =
IPush 'dyn |
IPop |
IGet nat |
ISet nat |
ILoad 'var |
IStore 'var |

IOp 'op |
IOpInl 'opinl |
is-jump: *ICJump 'label 'label* |
ICall 'fun |
is-return: *IReturn*

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

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

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

— n-ary operations
$\mathfrak{Op}$ :: *'op* ⇒ *'dyn list* ⇒ *'dyn* **and** $\mathfrak{Arity}$ **and**
$\mathfrak{InlOp}$ **and** $\mathfrak{Inl}$ **and** $\mathfrak{IsInl}$ **and** $\mathfrak{DeInl}$ :: *'opinl* ⇒ *'op*
begin

12.2 Semantics

inductive *step* (**infix** → *55*) **where**
step-push:
next-instr (*F-get F*) *f l pc* = *Some* (*IPush d*) ⟹
State F H (*Frame f l pc R* Σ # *st*) → *State F H* (*Frame f l* (*Suc pc*) *R* (*d* # Σ) # *st*) |

step-pop:
next-instr (*F-get F*) *f l pc* = *Some IPop* ⟹
State F H (*Frame f l pc R* (*d* # Σ) # *st*) → *State F H* (*Frame f l* (*Suc pc*) *R* Σ # *st*) |

step-get:
next-instr (*F-get F*) *f l pc* = *Some* (*IGet n*) ⟹
n < *length R* ⟹ *d* = *R* ! *n* ⟹
State F H (*Frame f l pc R* Σ # *st*) → *State F H* (*Frame f l* (*Suc pc*) *R* (*d* # Σ) # *st*) |

step-set:
next-instr (*F-get F*) *f l pc* = *Some* (*ISet n*) $\Longrightarrow$
$n <$ *length R* $\Longrightarrow$ $R' = R[n := d] \Longrightarrow$
State F H (*Frame f l pc R* (*d* # Σ) # *st*) $\rightarrow$ *State F H* (*Frame f l* (*Suc pc*) R' Σ # *st*) |

step-load:
next-instr (*F-get F*) *f l pc* = *Some* (*ILoad x*) $\Longrightarrow$
heap-get H (*x*, *y*) = *Some d* $\Longrightarrow$
State F H (*Frame f l pc R* (*y* # Σ) # *st*) $\rightarrow$ *State F H* (*Frame f l* (*Suc pc*) *R* (*d* # Σ) # *st*) |

step-store:
next-instr (*F-get F*) *f l pc* = *Some* (*IStore x*) $\Longrightarrow$
heap-add H (*x*, *y*) *d* = $H' \Longrightarrow$
State F H (*Frame f l pc R* (*y* # *d* # Σ) # *st*) $\rightarrow$ *State F* H' (*Frame f l* (*Suc pc*) *R* Σ # *st*) |

step-op:
next-instr (*F-get F*) *f l pc* = *Some* (*IOp op*) $\Longrightarrow$
$\mathfrak{Arity}$ *op* = *ar* $\Longrightarrow$ *ar* $\leq$ *length* Σ $\Longrightarrow$ $\mathfrak{Inl}$ *op* (*take ar* Σ) = *None* $\Longrightarrow$
$\mathfrak{Op}$ *op* (*take ar* Σ) = *x* $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State F H* (*Frame f l* (*Suc pc*) *R* (*x* # *drop ar* Σ) # *st*) |

step-op-inl:
next-instr (*F-get F*) *f l pc* = *Some* (*IOp op*) $\Longrightarrow$
$\mathfrak{Arity}$ *op* = *ar* $\Longrightarrow$ *ar* $\leq$ *length* Σ $\Longrightarrow$ $\mathfrak{Inl}$ *op* (*take ar* Σ) = *Some opinl* $\Longrightarrow$
$\mathfrak{InlOp}$ *opinl* (*take ar* Σ) = *x* $\Longrightarrow$
F' = *Fenv.map-entry F f* (λ*fd. rewrite-fundef-body fd l pc* (*IOpInl opinl*)) $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State* F' *H* (*Frame f l* (*Suc pc*) *R* (*x* # *drop ar* Σ) # *st*) |

step-op-inl-hit:
next-instr (*F-get F*) *f l pc* = *Some* (*IOpInl opinl*) $\Longrightarrow$
$\mathfrak{Arity}$ ($\mathfrak{DeInl}$ *opinl*) = *ar* $\Longrightarrow$ *ar* $\leq$ *length* Σ $\Longrightarrow$ $\mathfrak{IsInl}$ *opinl* (*take ar* Σ) $\Longrightarrow$
$\mathfrak{InlOp}$ *opinl* (*take ar* Σ) = *x* $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State F H* (*Frame f l* (*Suc pc*) *R* (*x* # *drop ar* Σ) # *st*) |

step-op-inl-miss:
next-instr (*F-get F*) *f l pc* = *Some* (*IOpInl opinl*) $\Longrightarrow$
$\mathfrak{Arity}$ ($\mathfrak{DeInl}$ *opinl*) = *ar* $\Longrightarrow$ *ar* $\leq$ *length* Σ $\Longrightarrow$ $\neg$ $\mathfrak{IsInl}$ *opinl* (*take ar* Σ) $\Longrightarrow$
$\mathfrak{InlOp}$ *opinl* (*take ar* Σ) = *x* $\Longrightarrow$
F' = *Fenv.map-entry F f* (λ*fd. rewrite-fundef-body fd l pc* (*IOp* ($\mathfrak{DeInl}$ *opinl*))) $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State* F' *H* (*Frame f l* (*Suc pc*) *R* (*x* # *drop ar* Σ) # *st*) |

step-cjump:
next-instr (*F-get F*) *f l pc* = *Some* (*ICJump* l_t l_f) $\Longrightarrow$
is-true d $\wedge$ l' = l_t $\vee$ *is-false d* $\wedge$ l' = l_f $\Longrightarrow$
State F H (*Frame f l pc R* (*d* # Σ) # *st*) $\rightarrow$ *State F H* (*Frame f l'* 0 *R* Σ # *st*) |

step-call:
next-instr (*F-get F*) *f l pc* = *Some* (*ICall g*) $\Longrightarrow$
F-get F g = *Some gd* $\Longrightarrow$ *arity gd* $\leq$ *length* Σ $\Longrightarrow$
$frame_g$ = *allocate-frame g gd* (*take* (*arity gd*) Σ) *uninitialized* $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State F H* ($frame_g$ # *Frame f l pc R* Σ # *st*) |

step-return:
next-instr (*F-get F*) *g* l_g pc_g = *Some IReturn* $\Longrightarrow$
F-get F g = *Some gd* $\Longrightarrow$ *arity gd* $\leq$ *length* Σ_f $\Longrightarrow$
length Σ_g = *return gd* $\Longrightarrow$
$frame_f{}'$ = *Frame f* l_f (*Suc* pc_f) R_f (Σ_g @ *drop* (*arity gd*) Σ_f) $\Longrightarrow$
State F H (*Frame g* l_g pc_g R_g Σ_g # *Frame f* l_f pc_f R_f Σ_f # *st*) $\rightarrow$ *State F H* ($frame_f{}'$ # *st*)

lemma *step-deterministic*:
assumes *s1* $\rightarrow$ *s2* **and** *s1* $\rightarrow$ *s3*
shows *s2* = *s3*
⟨*proof*⟩

lemma *step-right-unique*: *right-unique step*
⟨*proof*⟩

lemma *final-finished*:
assumes *final F-get IReturn s*
shows *finished step s*
⟨*proof*⟩

sublocale *semantics step final F-get IReturn*
⟨*proof*⟩

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

sublocale *language step final F-get IReturn load*
⟨*proof*⟩

end

end
theory *Unboxed*
imports *Global Dynamic*

```
begin

datatype type = Ubx1 | Ubx2

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

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

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

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

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

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

fun unbox :: type ⇒ '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 l pc R* Σ) = *Frame g l pc R* (*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

13 n-ary operations

locale *nary-operations-ubx* =
nary-operations-inl 𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩 +
unboxedval unitialized 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**
unitialized :: *'dyn* **and** *is-true* **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**

$\mathfrak{Ubx}$-*opubx-type*:
$\mathfrak{Ubx}$ *opinl ts* = *Some opubx* $\Longrightarrow$ *fst* ($\mathfrak{TypeOfOp}$ *opubx*) = *ts* **and**

$\mathfrak{TypeOfOp}$-*correct*:
$\mathfrak{TypeOfOp}$ *opubx* = (*map typeof xs*, τ) $\Longrightarrow$
$\exists\, y.$ $\mathfrak{UbxOp}$ *opubx xs* = *Some y* $\wedge$ *typeof y* = τ **and**
$\mathfrak{TypeOfOp}$-*complete*:
$\mathfrak{UbxOp}$ *opubx xs* = *Some y* $\Longrightarrow$ $\mathfrak{TypeOfOp}$ *opubx* = (*map typeof xs*, *typeof y*)

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

14 Unboxed caching

14.1 Static representation

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

locale *ubx* =
Fenv: *env F-empty F-get F-add F-to-list* +
Henv: *env heap-empty heap-get heap-add heap-to-list* +
nary-operations-ubx
$\mathfrak{Op}$ $\mathfrak{Arity}$
$\mathfrak{InlOp}$ $\mathfrak{Inl}$ $\mathfrak{IsInl}$ $\mathfrak{DeInl}$
uninitialized is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
$\mathfrak{UbxOp}$ $\mathfrak{Ubx}$ $\mathfrak{Box}$ $\mathfrak{TypeOfOp}$
for
— Functions environment
F-empty **and**
F-get :: '*fenv* $\Rightarrow$ '*fun* $\Rightarrow$ ('*label*, ('*dyn*, '*var*, '*fun*, '*label*, '*op*, '*opinl*, '*opubx*, '*num*, '*bool*) *instr*) *fundef option* **and**
F-add **and** *F-to-list* **and**

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

heap-add **and** *heap-to-list* **and**

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

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

— n-ary operations
𝔒𝔭 :: *'op* ⇒ *'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

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

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

lemma *instr-sel-generalize-instr*[*simp*]:
is-jump (*generalize-instr instr*) ⟷ *is-jump instr*
is-fun-call (*generalize-instr instr*) ⟷ *is-fun-call instr*
is-return (*generalize-instr instr*) ⟷ *is-return instr*
⟨*proof*⟩

lemma *instr-sel-generalize-instr-comp*[*simp*]:
is-jump ∘ *generalize-instr* = *is-jump* **and**
is-fun-call ∘ *generalize-instr* = *is-fun-call* **and**
is-return ∘ *generalize-instr* = *is-return*
⟨*proof*⟩

fun *generalize-fundef* **where**
generalize-fundef (*Fundef xs ar ret locals*) =
Fundef (*map-ran* (λ-. *map generalize-instr*) *xs*) *ar ret locals*

lemma *generalize-instr-idempotent*[*simp*]:
generalize-instr (*generalize-instr x*) = *generalize-instr x*
⟨*proof*⟩

lemma *generalize-instr-idempotent-comp*[*simp*]:
generalize-instr ∘ *generalize-instr* = *generalize-instr*
⟨*proof*⟩

lemma *length-body-generalize-fundef*[*simp*]: *length* (*body* (*generalize-fundef fd*)) = *length* (*body fd*)
⟨*proof*⟩

lemma *arity-generalize-fundef*[*simp*]: *arity* (*generalize-fundef fd*) = *arity fd*
⟨*proof*⟩

lemma *return-generalize-fundef*[*simp*]: *return* (*generalize-fundef fd*) = *return fd*
⟨*proof*⟩

lemma *fundef-locals-generalize*[*simp*]: *fundef-locals* (*generalize-fundef fd*) = *fundef-locals fd*
⟨*proof*⟩

lemma *funtype-generalize-fundef*[*simp*]: *funtype* (*generalize-fundef fd*) = *funtype fd*
⟨*proof*⟩

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

lemma *image-fst-set-body-generalize-fundef*[*simp*]:
fst ‘ *set* (*body* (*generalize-fundef fd*)) = *fst* ‘ *set* (*body fd*)
⟨*proof*⟩

lemma *map-of-generalize-fundef-conv*:
map-of (*body* (*generalize-fundef fd*)) *l* = *map-option* (*map generalize-instr*) (*map-of* (*body fd*) *l*)
⟨*proof*⟩

lemma *map-option-arity-get-map-entry-generalize-fundef*[*simp*]:
map-option arity ∘ *F-get* (*Fenv.map-entry F2 f generalize-fundef*) =
map-option arity ∘ *F-get F2*
⟨*proof*⟩

lemma *instr-at-generalize-fundef-conv*:
instr-at (*generalize-fundef fd*) *l* = *map-option generalize-instr o instr-at fd l*
⟨*proof*⟩

14.2 Semantics

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

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

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

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

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

step-get-ubx-hit:
next-instr (*F-get F*) *f l pc* = *Some* (*IGetUbx* τ *n*) $\Longrightarrow$
n < *length R* $\Longrightarrow$ *cast-Dyn* (*R* ! *n*) = *Some d* $\Longrightarrow$ *unbox* τ *d* = *Some blob* $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State F H* (*Frame f l* (*Suc pc*) *R* (*blob* # Σ) # *st*) |

step-get-ubx-miss:
next-instr (*F-get F*) *f l pc* = *Some* (*IGetUbx* τ *n*) $\Longrightarrow$
n < *length R* $\Longrightarrow$ *cast-Dyn* (*R* ! *n*) = *Some d* $\Longrightarrow$ *unbox* τ *d* = *None* $\Longrightarrow$
F' = *Fenv.map-entry F f generalize-fundef* $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State F' H* (*box-stack f* (*Frame f l* (*Suc pc*) *R* (*OpDyn d* # Σ) # *st*)) |

step-set:
next-instr (*F-get F*) *f l pc* = *Some* (*ISet n*) $\Longrightarrow$
n < *length R* $\Longrightarrow$ *cast-Dyn blob* = *Some d* $\Longrightarrow$ *R'* = *R*[*n* := *OpDyn d*] $\Longrightarrow$
State F H (*Frame f l pc R* (*blob* # Σ) # *st*) $\rightarrow$ *State F H* (*Frame f l* (*Suc pc*) *R'* Σ # *st*) |

step-set-ubx:
next-instr (*F-get F*) *f l pc* = *Some* (*ISetUbx* τ *n*) $\Longrightarrow$

$n < \mathit{length}\ R \Longrightarrow \mathit{cast\text{-}and\text{-}box}\ \tau\ \mathit{blob} = \mathit{Some}\ d \Longrightarrow R' = R[n := \mathit{OpDyn}\ d] \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ \mathit{pc}\ R\ (\mathit{blob} \#\ \Sigma) \#\ st) \rightarrow \mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ \mathit{pc})\ R'\ \Sigma \#\ st)$ |

step-load:
$\mathit{next\text{-}instr}\ (\mathit{F\text{-}get}\ F)\ f\ l\ \mathit{pc} = \mathit{Some}\ (\mathit{ILoad}\ x) \Longrightarrow$
$\mathit{cast\text{-}Dyn}\ i = \mathit{Some}\ i' \Longrightarrow \mathit{heap\text{-}get}\ H\ (x,\ i') = \mathit{Some}\ d \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ \mathit{pc}\ R\ (i \#\ \Sigma) \#\ st) \rightarrow \mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ \mathit{pc})\ R\ (\mathit{OpDyn}\ d \#\ \Sigma) \#\ st)$ |

step-load-ubx-hit:
$\mathit{next\text{-}instr}\ (\mathit{F\text{-}get}\ F)\ f\ l\ \mathit{pc} = \mathit{Some}\ (\mathit{ILoadUbx}\ \tau\ x) \Longrightarrow$
$\mathit{cast\text{-}Dyn}\ i = \mathit{Some}\ i' \Longrightarrow \mathit{heap\text{-}get}\ H\ (x,\ i') = \mathit{Some}\ d \Longrightarrow \mathit{unbox}\ \tau\ d = \mathit{Some}\ \mathit{blob} \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ \mathit{pc}\ R\ (i \#\ \Sigma) \#\ st) \rightarrow \mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ \mathit{pc})\ R\ (\mathit{blob} \#\ \Sigma) \#\ st)$ |

step-load-ubx-miss:
$\mathit{next\text{-}instr}\ (\mathit{F\text{-}get}\ F)\ f\ l\ \mathit{pc} = \mathit{Some}\ (\mathit{ILoadUbx}\ \tau\ x) \Longrightarrow$
$\mathit{cast\text{-}Dyn}\ i = \mathit{Some}\ i' \Longrightarrow \mathit{heap\text{-}get}\ H\ (x,\ i') = \mathit{Some}\ d \Longrightarrow \mathit{unbox}\ \tau\ d = \mathit{None} \Longrightarrow$
$F' = \mathit{Fenv.map\text{-}entry}\ F\ f\ \mathit{generalize\text{-}fundef} \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ \mathit{pc}\ R\ (i \#\ \Sigma) \#\ st) \rightarrow \mathit{State}\ F'\ H\ (\mathit{box\text{-}stack}\ f\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ \mathit{pc})\ R\ (\mathit{OpDyn}\ d \#\ \Sigma) \#\ st))$ |

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

step-store-ubx:
$\mathit{next\text{-}instr}\ (\mathit{F\text{-}get}\ F)\ f\ l\ \mathit{pc} = \mathit{Some}\ (\mathit{IStoreUbx}\ \tau\ x) \Longrightarrow$
$\mathit{cast\text{-}Dyn}\ i = \mathit{Some}\ i' \Longrightarrow \mathit{cast\text{-}and\text{-}box}\ \tau\ \mathit{blob} = \mathit{Some}\ d \Longrightarrow \mathit{heap\text{-}add}\ H\ (x,\ i')\ d = H' \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ \mathit{pc}\ R\ (i \#\ \mathit{blob} \#\ \Sigma) \#\ st) \rightarrow \mathit{State}\ F\ H'\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ \mathit{pc})\ R\ \Sigma \#\ st)$ |

step-op:
$\mathit{next\text{-}instr}\ (\mathit{F\text{-}get}\ F)\ f\ l\ \mathit{pc} = \mathit{Some}\ (\mathit{IOp}\ \mathit{op}) \Longrightarrow$
$\mathfrak{Arity}\ \mathit{op} = \mathit{ar} \Longrightarrow \mathit{ar} \leq \mathit{length}\ \Sigma \Longrightarrow$
$\mathit{ap\text{-}map\text{-}list}\ \mathit{cast\text{-}Dyn}\ (\mathit{take}\ \mathit{ar}\ \Sigma) = \mathit{Some}\ \Sigma' \Longrightarrow$
$\mathfrak{Inl}\ \mathit{op}\ \Sigma' = \mathit{None} \Longrightarrow \mathfrak{Op}\ \mathit{op}\ \Sigma' = x \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ \mathit{pc}\ R\ \Sigma \#\ st) \rightarrow \mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ \mathit{pc})\ R\ (\mathit{OpDyn}\ x \#\ \mathit{drop}\ \mathit{ar}\ \Sigma) \#\ st)$ |

step-op-inl:

next-instr (*F-get F*) *f l pc* = *Some* (*IOp op*) $\Longrightarrow$
$\mathfrak{Arity}$ *op* = *ar* $\Longrightarrow$ *ar* $\leq$ *length* Σ $\Longrightarrow$
ap-map-list cast-Dyn (*take ar* Σ) = *Some* Σ' $\Longrightarrow$
$\mathfrak{Inl}$ *op* Σ' = *Some opinl* $\Longrightarrow$ $\mathfrak{InlOp}$ *opinl* Σ' = *x* $\Longrightarrow$
F' = *Fenv.map-entry F f* (λ*fd. rewrite-fundef-body fd l pc* (*IOpInl opinl*)) $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State F' H* (*Frame f l* (*Suc pc*) *R* (*OpDyn x* # *drop ar* Σ) # *st*) |

step-op-inl-hit:
next-instr (*F-get F*) *f l pc* = *Some* (*IOpInl opinl*) $\Longrightarrow$
$\mathfrak{Arity}$ ($\mathfrak{DeInl}$ *opinl*) = *ar* $\Longrightarrow$ *ar* $\leq$ *length* Σ $\Longrightarrow$
ap-map-list cast-Dyn (*take ar* Σ) = *Some* Σ' $\Longrightarrow$
$\mathfrak{IsInl}$ *opinl* Σ' $\Longrightarrow$ $\mathfrak{InlOp}$ *opinl* Σ' = *x* $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State F H* (*Frame f l* (*Suc pc*) *R* (*OpDyn x* # *drop ar* Σ) # *st*) |

step-op-inl-miss:
next-instr (*F-get F*) *f l pc* = *Some* (*IOpInl opinl*) $\Longrightarrow$
$\mathfrak{Arity}$ ($\mathfrak{DeInl}$ *opinl*) = *ar* $\Longrightarrow$ *ar* $\leq$ *length* Σ $\Longrightarrow$
ap-map-list cast-Dyn (*take ar* Σ) = *Some* Σ' $\Longrightarrow$
$\neg$ $\mathfrak{IsInl}$ *opinl* Σ' $\Longrightarrow$ $\mathfrak{InlOp}$ *opinl* Σ' = *x* $\Longrightarrow$
F' = *Fenv.map-entry F f* (λ*fd. rewrite-fundef-body fd l pc* (*IOp* ($\mathfrak{DeInl}$ *opinl*))) $\Longrightarrow$
State F H (*Frame f l pc R* Σ # *st*) $\rightarrow$ *State F' H* (*Frame f l* (*Suc pc*) *R* (*OpDyn x* # *drop ar* Σ) # *st*) |

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

step-cjump:
next-instr (*F-get F*) *f l pc* = *Some* (*ICJump* l_t l_f) $\Longrightarrow$
cast-Dyn y = *Some d* $\Longrightarrow$ *is-true d* $\wedge$ *l'* = l_t $\vee$ *is-false d* $\wedge$ *l'* = l_f $\Longrightarrow$
State F H (*Frame f l pc R* (*y* # Σ) # *st*) $\rightarrow$ *State F H* (*Frame f l' 0 R* Σ # *st*) |

step-call:
next-instr (*F-get F*) *f l pc* = *Some* (*ICall g*) $\Longrightarrow$
F-get F g = *Some gd* $\Longrightarrow$ *arity gd* $\leq$ *length* Σ $\Longrightarrow$
list-all is-dyn-operand (*take* (*arity gd*) Σ) $\Longrightarrow$
$frame_g$ = *allocate-frame g gd* (*take* (*arity gd*) Σ) (*OpDyn uninitialized*) $\Longrightarrow$
State F H (*Frame f l pc* R_f Σ # *st*) $\rightarrow$ *State F H* ($frame_g$ # *Frame f l pc* R_f Σ # *st*) |

step-return:
next-instr (*F-get F*) *g* l_g pc_g = *Some IReturn* $\Longrightarrow$

F-get F g = Some gd ⟹ arity gd ≤ length Σ_f ⟹
length Σ_g = return gd ⟹ list-all is-dyn-operand Σ_g ⟹
frame_f' = Frame f l_f (Suc pc_f) R_f (Σ_g @ drop (arity gd) Σ_f) ⟹
State F H (Frame g l_g pc_g R_g Σ_g # Frame f l_f pc_f R_f Σ_f # st) → State F H (frame_f' # st)

lemma *step-deterministic*:
assumes *s1 → s2* **and** *s1 → s3*
shows *s2 = s3*
⟨*proof*⟩

lemma *step-right-unique*: *right-unique step*
⟨*proof*⟩

lemma *final-finished*:
assumes *final F-get IReturn s*
shows *finished step s*
⟨*proof*⟩

sublocale *ubx-sem*: *semantics step final F-get IReturn*
⟨*proof*⟩

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

sublocale *inca-lang*: *language step final F-get IReturn load*
⟨*proof*⟩

end

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

lemma *f-g-eq-f-imp-f-comp-g-eq-f*[*simp*]: *(⋀x. f (g x) = f x) ⟹ (f ∘ g) = f*
⟨*proof*⟩

context *ubx* **begin**

inductive *sp-instr* **for** *F ret* **where**
Push:
sp-instr F ret (IPush d) Σ (None # Σ) |
PushUbx1:
sp-instr F ret (IPushUbx1 u) Σ (Some Ubx1 # Σ) |
PushUbx2:
sp-instr F ret (IPushUbx2 u) Σ (Some Ubx2 # Σ) |
Pop:
sp-instr F ret IPop (τ # Σ) Σ |

Get:
sp-instr F ret (*IGet n*) Σ (*None* # Σ) |
GetUbx:
sp-instr F ret (*IGetUbx τ n*) Σ (*Some τ* # Σ) |
Set:
sp-instr F ret (*ISet n*) (*None* # Σ) Σ |
SetUbx:
sp-instr F ret (*ISetUbx τ n*) (*Some τ* # Σ) Σ |
Load:
sp-instr F ret (*ILoad x*) (*None* # Σ) (*None* # Σ) |
LoadUbx:
sp-instr F ret (*ILoadUbx τ x*) (*None* # Σ) (*Some τ* # Σ) |
Store:
sp-instr F ret (*IStore x*) (*None* # *None* # Σ) Σ |
StoreUbx:
sp-instr F ret (*IStoreUbx τ x*) (*None* # *Some τ* # Σ) Σ |
Op:
Σ*i* = (*replicate* (𝔄𝔯𝔦𝔱𝔶 *op*) *None* @ Σ) ⟹
sp-instr F ret (*IOp op*) Σ*i* (*None* # Σ) |
OpInl:
Σ*i* = (*replicate* (𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 *opinl*)) *None* @ Σ) ⟹
sp-instr F ret (*IOpInl opinl*) Σ*i* (*None* # Σ) |
OpUbx:
Σ*i* = *fst* (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 *opubx*) @ Σ ⟹ *result* = *snd* (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 *opubx*) ⟹
sp-instr F ret (*IOpUbx opubx*) Σ*i* (*result* # Σ) |
CJump:
sp-instr F ret (*ICJump* l_t l_f) [*None*] [] |
Call:
F f = *Some* (*ar*, *r*) ⟹ Σ*i* = *replicate ar None* @ Σ ⟹ Σ*o* = *replicate r None* @ Σ ⟹
sp-instr F ret (*ICall f*) Σ*i* Σ*o* |
Return: Σ*i* = *replicate ret None* ⟹
sp-instr F ret IReturn Σ*i* []

inductive *sp-instrs* **for** *F ret* **where**
Nil:
sp-instrs F ret [] Σ Σ |
Cons:
sp-instr F ret instr Σ*i* Σ ⟹ *sp-instrs F ret instrs* Σ Σ*o* ⟹
sp-instrs F ret (*instr* # *instrs*) Σ*i* Σ*o*

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

lemma *sp-instrs-ConsD*:
assumes *sp-instrs F ret* (*instr* # *instrs*) Σ*i* Σ*o*
shows ∃Σ. *sp-instr F ret instr* Σ*i* Σ ∧ *sp-instrs F ret instrs* Σ Σ*o*
⟨*proof*⟩

lemma *sp-instr-deterministic*:

assumes
sp-instr F ret instr Σi Σo **and**
sp-instr F ret instr Σi $\Sigma o'$
shows $\Sigma o = \Sigma o'$
⟨*proof*⟩

lemma *sp-instr-right-unique*: *right-unique* (λ(*instr*, Σi) Σo. *sp-instr F ret instr* Σi Σo)
⟨*proof*⟩

lemma *sp-instrs-deterministic*:
assumes
sp-instrs F ret instr Σi Σo **and**
sp-instrs F ret instr Σi $\Sigma o'$
shows $\Sigma o = \Sigma o'$
⟨*proof*⟩

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

lemma *fun-call-in-range-generalize-instr*[*simp*]:
fun-call-in-range F (*generalize-instr instr*) $\longleftrightarrow$ *fun-call-in-range F instr*
⟨*proof*⟩

lemma *sp-instr-complete*:
assumes *fun-call-in-range F instr*
shows $\exists \Sigma i\ \Sigma o$. *sp-instr F ret instr* Σi Σo
⟨*proof*⟩

lemma *sp-instr-Op2I*:
assumes $\mathfrak{Arity}$ *op* = *2*
shows *sp-instr F ret* (*IOp op*) (*None # None #* Σ) (*None #* Σ)
⟨*proof*⟩

lemma
assumes
F-def: *F* = *Map.empty* **and**
arity-op: $\mathfrak{Arity}$ *op* = *2*
shows *sp-instrs F 1* [*IPush x*, *IPush y*, *IOp op*, *IReturn*] [] []
⟨*proof*⟩

lemma *sp-intrs-NilD*[*dest*]: *sp-instrs F ret* [] Σi $\Sigma o \Longrightarrow \Sigma i = \Sigma o$
⟨*proof*⟩

lemma *sp-instrs-list-update*:
assumes
sp-instrs F ret instrs Σi Σo **and**
sp-instr F ret (*instrs*!*n*) = *sp-instr F ret instr*

shows *sp-instrs F ret* (*instrs*[*n* := *instr*]) Σ*i* Σ*o*
⟨*proof*⟩

lemma *sp-instrs-appendD*:
assumes *sp-instrs F ret* (*instrs1* @ *instrs2*) Σ*i* Σ*o*
shows ∃Σ. *sp-instrs F ret instrs1* Σ*i* Σ ∧ *sp-instrs F ret instrs2* Σ Σ*o*
⟨*proof*⟩

lemma *sp-instrs-appendD′*:
assumes *sp-instrs F ret* (*instrs1* @ *instrs2*) Σ*i* Σ*o* **and** *sp-instrs F ret instrs1* Σ*i* Σ
shows *sp-instrs F ret instrs2* Σ Σ*o*
⟨*proof*⟩

lemma *sp-instrs-appendI*[*intro*]:
assumes *sp-instrs F ret instrs1* Σ*i* Σ **and** *sp-instrs F ret instrs2* Σ Σ*o*
shows *sp-instrs F ret* (*instrs1* @ *instrs2*) Σ*i* Σ*o*
⟨*proof*⟩

lemma *sp-instrs-singleton-conv*[*simp*]:
sp-instrs F ret [*instr*] Σ*i* Σ*o* ⟷ *sp-instr F ret instr* Σ*i* Σ*o*
⟨*proof*⟩

lemma *sp-instrs-singletonI*:
assumes *sp-instr F ret instr* Σ*i* Σ*o*
shows *sp-instrs F ret* [*instr*] Σ*i* Σ*o*
⟨*proof*⟩

fun *local-var-in-range* **where**
local-var-in-range n (*IGet k*) ⟷ *k* < *n* |
local-var-in-range n (*IGetUbx τ k*) ⟷ *k* < *n* |
local-var-in-range n (*ISet k*) ⟷ *k* < *n* |
local-var-in-range n (*ISetUbx τ k*) ⟷ *k* < *n* |
local-var-in-range - - ⟷ *True*

lemma *local-var-in-range-generalize-instr*[*simp*]:
local-var-in-range n (*generalize-instr instr*) ⟷ *local-var-in-range n instr*
⟨*proof*⟩

lemma *local-var-in-range-comp-generalize-instr*[*simp*]:
local-var-in-range n ∘ *generalize-instr* = *local-var-in-range n*
⟨*proof*⟩

fun *jump-in-range* **where**
jump-in-range L (*ICJump* l_t l_f) ⟷ {l_t, l_f} ⊆ *L* |
jump-in-range L - ⟷ *True*

inductive *wf-basic-block* **for** *F L ret n* **where**
instrs ≠ [] ⟹

list-all (*local-var-in-range n*) *instrs* ⟹
list-all (*fun-call-in-range F*) *instrs* ⟹
list-all (*jump-in-range L*) *instrs* ⟹
list-all (λ*i*. ¬ *is-jump i* ∧ ¬ *is-return i*) (*butlast instrs*) ⟹
sp-instrs F ret instrs [] [] ⟹
wf-basic-block F L ret n (*label*, *instrs*)

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

definition *wf-fundef* **where**
wf-fundef F fd ⟷
body fd ≠ [] ∧
list-all
(*wf-basic-block F* (*fst* ' *set* (*body fd*)) (*return fd*) (*arity fd* + *fundef-locals fd*))
(*body fd*)

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

definition *wf-fundefs* **where**
wf-fundefs F ⟷ (∀ *f*. *pred-option* (*wf-fundef* (*map-option funtype* ∘ *F*)) (*F f*))

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

lemma *wf-fundefs-getD*:
shows *wf-fundefs F* ⟹ *F f* = *Some fd* ⟹ *wf-fundef* (*map-option funtype* ∘ *F*) *fd*
⟨*proof*⟩

definition *wf-prog* **where**
wf-prog p ⟷ *wf-fundefs* (*F-get* (*prog-fundefs p*))

definition *wf-state* **where**
wf-state s ⟷ *wf-fundefs* (*F-get* (*state-fundefs s*))

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

lemma *sp-instr-generalize0*:
assumes *sp-instr F ret instr* Σ*i* Σ*o* **and**
Σ*i*′ = *map* (λ-. *None*) Σ*i* **and** Σ*o*′ = *map* (λ-. *None*) Σ*o*
shows *sp-instr F ret* (*generalize-instr instr*) Σ*i*′ Σ*o*′
⟨*proof*⟩

lemma *sp-instrs-generalize0*:

assumes *sp-instrs F ret instrs* Σi Σo **and**
$\Sigma i' = map\ (\lambda\text{-}.\ None)\ \Sigma i$ **and** $\Sigma o' = map\ (\lambda\text{-}.\ None)\ \Sigma o$
shows *sp-instrs F ret* (*map generalize-instr instrs*) $\Sigma i'$ $\Sigma o'$
⟨*proof*⟩

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

lemma *jump-in-range-generalize-instr*[*simp*]:
jump-in-range L (*generalize-instr instr*) ⟷ *jump-in-range L instr*
⟨*proof*⟩

lemma *wf-basic-block-map-generalize-instr*:
assumes *wf-basic-block F L ret n* (*label*, *instrs*)
shows *wf-basic-block F L ret n* (*label*, *map generalize-instr instrs*)
⟨*proof*⟩

lemma *list-all-wf-basic-block-generalize-fundef*:
assumes *list-all* (*wf-basic-block F L ret n*) (*body fd*)
shows *list-all* (*wf-basic-block F L ret n*) (*body* (*generalize-fundef fd*))
⟨*proof*⟩

lemma *wf-fundefs-map-entry*:
assumes *wf-F*: *wf-fundefs* (*F-get F*) **and**
same-funtype: ⋀*fd*. *fd* ∈ *ran* (*F-get F*) ⟹ *funtype* (*f fd*) = *funtype fd* **and**
same-arity: ⋀*fd*. *F-get F x* = *Some fd* ⟹ *arity* (*f fd*) = *arity fd* **and**
same-return: ⋀*fd*. *F-get F x* = *Some fd* ⟹ *return* (*f fd*) = *return fd* **and**
same-body-length: ⋀*fd*. *F-get F x* = *Some fd* ⟹ *length* (*body* (*f fd*)) = *length* (*body fd*) **and**
same-locals: ⋀*fd*. *F-get F x* = *Some fd* ⟹ *fundef-locals* (*f fd*) = *fundef-locals fd* **and**
same-labels: ⋀*fd*. *F-get F x* = *Some fd* ⟹ *fst* ‘ *set* (*body* (*f fd*)) = *fst* ‘ *set* (*body fd*) **and**
list-all-wf-basic-block-f: ⋀*fd*.
F-get F x = *Some fd* ⟹
list-all (*wf-basic-block* (*map-option funtype* ∘ *F-get F*) (*fst* ‘ *set* (*body fd*)) (*return fd*)
(*arity fd* + *fundef-locals fd*)) (*body fd*) ⟹
list-all (*wf-basic-block* (*map-option funtype* ∘ *F-get F*) (*fst* ‘ *set* (*body fd*)) (*return fd*)
(*arity fd* + *fundef-locals fd*)) (*body* (*f fd*))
shows *wf-fundefs* (*F-get* (*Fenv.map-entry F x f*))
⟨*proof*⟩

lemma *wf-fundefs-generalize*:
assumes *wf-F*: *wf-fundefs* (*F-get F*)
shows *wf-fundefs* (*F-get* (*Fenv.map-entry F f generalize-fundef*))

⟨*proof*⟩

lemma *list-all-wf-basic-block-rewrite-fundef-body*:
assumes
list-all (*wf-basic-block F L ret n*) (*body fd*) **and**
instr-at fd l pc = *Some instr* **and**
sp-instr-eq: *sp-instr F ret instr* = *sp-instr F ret instr′* **and**
local-var-in-range-iff: *local-var-in-range n instr′* ⟷ *local-var-in-range n instr* **and**
fun-call-in-range-iff: *fun-call-in-range F instr′* ⟷ *fun-call-in-range F instr* **and**
jump-in-range-iff: *jump-in-range L instr′* ⟷ *jump-in-range L instr* **and**
is-jump-iff: *is-jump instr′* ⟷ *is-jump instr* **and**
is-return-iff: *is-return instr′* ⟷ *is-return instr*
shows *list-all* (*wf-basic-block F L ret n*) (*body* (*rewrite-fundef-body fd l pc instr′*))
⟨*proof*⟩

lemma *wf-fundefs-rewrite-body*:
assumes *wf-fundefs* (*F-get F*) **and**
next-instr (*F-get F*) *f l pc* = *Some instr* **and**
sp-instr-eq: ⋀*ret*.
sp-instr (*map-option funtype* ∘ *F-get F*) *ret instr′* =
sp-instr (*map-option funtype* ∘ *F-get F*) *ret instr* **and**
local-var-in-range-iff: ⋀*n*. *local-var-in-range n instr′* ⟷ *local-var-in-range n instr* **and**
fun-call-in-range-iff:
fun-call-in-range (*map-option funtype* ∘ *F-get F*) *instr′* ⟷
fun-call-in-range (*map-option funtype* ∘ *F-get F*) *instr* **and**
jump-in-range-iff: ⋀*L*. *jump-in-range L instr′* ⟷ *jump-in-range L instr* **and**
is-jump-iff: *is-jump instr′* ⟷ *is-jump instr* **and**
is-return-iff: *is-return instr′* ⟷ *is-return instr*
shows *wf-fundefs* (*F-get* (*Fenv.map-entry F f* (λ*fd*. *rewrite-fundef-body fd l pc instr′*)))
⟨*proof*⟩

lemma *sp-instr-Op-OpInl-conv*:
assumes *op* = 𝔇𝔢𝔍𝔫𝔩 *opinl*
shows *sp-instr F ret* (*IOp op*) = *sp-instr F ret* (*IOpInl opinl*)
⟨*proof*⟩

lemma *wf-state-step-preservation*:
assumes *wf-state s* **and** *step s s′*
shows *wf-state s′*
⟨*proof*⟩

end

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

lemma *cast-Dyn-eq-Some-imp-typeof*: *cast-Dyn u = Some d ⟹ typeof u = None*
⟨*proof*⟩

lemma *typeof-bind-OpDyn*[*simp*]: *typeof ∘ OpDyn = (λ-. None)*
⟨*proof*⟩

lemma *is-dyn-operand-eq-typeof*: *is-dyn-operand = (λx. typeof x = None)*
⟨*proof*⟩

lemma *is-dyn-operand-eq-typeof-Dyn*[*simp*]: *is-dyn-operand x ⟷ typeof x = None*
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *ap-map-list-cast-Dyn-replicate*:
assumes *ap-map-list cast-Dyn xs = Some ys*
shows *map typeof xs = replicate (length xs) None*
⟨*proof*⟩

context *unboxedval* **begin**

lemma *unbox-typeof*[*simp*]: *unbox τ d = Some blob ⟹ typeof blob = Some τ*
⟨*proof*⟩

lemma *cast-and-box-imp-typeof*[*simp*]: *cast-and-box τ blob = Some d ⟹ typeof blob = Some τ*
⟨*proof*⟩

lemma *norm-unbox-inverse*[*simp*]: *unbox τ d = Some blob ⟹ norm-unboxed blob*

$= d$
⟨*proof*⟩

lemma *norm-cast-and-box-inverse*[*simp*]:
cast-and-box τ *blob* = *Some* $d \Longrightarrow$ *norm-unboxed blob* = d
⟨*proof*⟩

lemma *typeof-and-norm-unboxed-imp-cast-Dyn*:
assumes *typeof* x' = *None norm-unboxed* x' = x
shows *cast-Dyn* x' = *Some* x
⟨*proof*⟩

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
⟨*proof*⟩

lemma *norm-unboxed-bind-OpDyn*[*simp*]: *norm-unboxed* ∘ *OpDyn* = *id*
⟨*proof*⟩

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

lemma *typeof-box-operand*[*simp*]: *typeof* (*box-operand u*) = *None*
⟨*proof*⟩

lemma *typeof-box-operand-comp*[*simp*]: *typeof* ∘ *box-operand* = (λ-. *None*)
⟨*proof*⟩

lemma *is-dyn-box-operand*: *is-dyn-operand* (*box-operand x*)
⟨*proof*⟩

lemma *is-dyn-operand-comp-box-operand*[*simp*]: *is-dyn-operand* ∘ *box-operand* = (λ-. *True*)
⟨*proof*⟩

lemma *norm-box-operand*[*simp*]: *norm-unboxed* (*box-operand x*) = *norm-unboxed x*
⟨*proof*⟩

end

end
theory *Inca-to-Ubx-simulation*
imports *List-util Result*
VeriComp.Simulation
Inca Ubx Ubx-Verification Unboxed-lemmas
begin

lemma *take-:Suc n = length xs ⟹ take n xs = butlast xs*
⟨*proof*⟩

lemma *append-take-singleton-conv:Suc n = length xs ⟹ xs = take n xs @ [xs ! n]*
⟨*proof*⟩

15 Locale imports

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

Fubx-empty **and**
Fubx-get :: *'fenv-ubx* ⇒ *'fun* ⇒ (*'label*, (*'dyn*, *'var*, *'fun*, *'label*, *'op*, *'opinl*, *'opubx*, *'ubx1*, *'ubx2*) *Ubx.instr*) *fundef option* **and**
Fubx-add **and** *Fubx-to-list* **and**

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

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

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

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

16 Normalization

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

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

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

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

lemma *rel-fundefsI*:
assumes ⋀*x*. *rel-option* (*rel-fundef* (=) *norm-eq*) (*F1 x*) (*F2 x*)
shows *rel-fundefs F1 F2*
⟨*proof*⟩

lemma *rel-fundefsD*:
assumes *rel-fundefs F1 F2*
shows *rel-option* (*rel-fundef* (=) *norm-eq*) (*F1 x*) (*F2 x*)
⟨*proof*⟩

lemma *rel-fundefs-next-instr*:
assumes *rel-F1-F2*: *rel-fundefs F1 F2*
shows *rel-option norm-eq* (*next-instr F1 f l pc*) (*next-instr F2 f l pc*)
⟨*proof*⟩

lemma *rel-fundefs-next-instr1*:
assumes *rel-F1-F2*: *rel-fundefs F1 F2* **and** *next-instr1*: *next-instr F1 f l pc* =

Some instr1
shows $\exists$ *instr2*. *next-instr F2 f l pc* = *Some instr2* $\wedge$ *norm-eq instr1 instr2*
⟨*proof*⟩

lemma *rel-fundefs-next-instr2*:
assumes *rel-F1-F2*: *rel-fundefs F1 F2* **and** *next-instr2*: *next-instr F2 f l pc* = *Some instr2*
shows $\exists$ *instr1*. *next-instr F1 f l pc* = *Some instr1* $\wedge$ *norm-eq instr1 instr2*
⟨*proof*⟩

lemma *rel-fundefs-empty*: *rel-fundefs* (λ-. *None*) (λ-. *None*)
⟨*proof*⟩

lemma *rel-fundefs-None1*:
assumes *rel-fundefs f g* **and** *f x* = *None*
shows *g x* = *None*
⟨*proof*⟩

lemma *rel-fundefs-None2*:
assumes *rel-fundefs f g* **and** *g x* = *None*
shows *f x* = *None*
⟨*proof*⟩

lemma *rel-fundefs-Some1*:
assumes *rel-fundefs f g* **and** *f x* = *Some y*
shows $\exists$ *z*. *g x* = *Some z* $\wedge$ *rel-fundef* (=) *norm-eq y z*
⟨*proof*⟩

lemma *rel-fundefs-Some2*:
assumes *rel-fundefs f g* **and** *g x* = *Some y*
shows $\exists$ *z*. *f x* = *Some z* $\wedge$ *rel-fundef* (=) *norm-eq z y*
⟨*proof*⟩

lemma *rel-fundefs-rel-option*:
assumes *rel-fundefs f g* **and** $\bigwedge$*x y*. *rel-fundef* (=) *norm-eq x y* $\Longrightarrow$ *h x y*
shows *rel-option h* (*f z*) (*g z*)
⟨*proof*⟩

lemma *rel-fundef-generalizeI*:
assumes *rel-fundef* (=) *norm-eq fd1 fd2*
shows *rel-fundef* (=) *norm-eq fd1* (*Subx.generalize-fundef fd2*)
⟨*proof*⟩

lemma *rel-fundefs-generalizeI*:
assumes *rel-fundefs* (*Finca-get F1*) (*Fubx-get F2*)
shows *rel-fundefs* (*Finca-get F1*) (*Fubx-get* (*Subx.Fenv.map-entry F2 f Subx.generalize-fundef*))
⟨*proof*⟩

lemma *rel-fundefs-rewriteI*:

assumes
rel-F1-F2: *rel-fundefs* (*Finca-get F1*) (*Fubx-get F2*) **and**
norm-eq instr1′ instr2′
shows *rel-fundefs*
(*Finca-get* (*Sinca.Fenv.map-entry F1 f* (λ*fd*. *rewrite-fundef-body fd l pc instr1′*)))
(*Fubx-get* (*Subx.Fenv.map-entry F2 f* (λ*fd*. *rewrite-fundef-body fd l pc instr2′*)))
(**is** *rel-fundefs* (*Finca-get ?F1′*) (*Fubx-get ?F2′*))
⟨*proof*⟩

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* [] = []
⟨*proof*⟩

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

lemma *norm-stack-append*: *norm-stack* (*xs* @ *ys*) = *norm-stack xs* @ *norm-stack ys*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *length-norm-stack*[*simp*]: *length* (*norm-stack xs*) = *length xs*
⟨*proof*⟩

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

lemma *is-valid-funcall-map-entry-generalize-fundefI*:
assumes *is-valid-fun-call* (*Fubx-get F2*) *g l pc* Σ *z*
shows *is-valid-fun-call* (*Fubx-get* (*Subx.Fenv.map-entry F2 f Subx.generalize-fundef*)) *g l pc* Σ *z*
⟨*proof*⟩

lemma *is-valid-fun-call-map-box-operandI*:
assumes *is-valid-fun-call* (*Fubx-get F2*) *g l pc* Σ *z*

shows *is-valid-fun-call* (*Fubx-get F2*) *g l pc* (*map Subx.box-operand* Σ) *z*
⟨*proof*⟩

lemma *inst-at-rewrite-fundef-body-disj*:
instr-at (*rewrite-fundef-body fd l pc instr*) *l pc* = *Some instr* ∨
instr-at (*rewrite-fundef-body fd l pc instr*) *l pc* = *None*
⟨*proof*⟩

lemma *is-valid-fun-call-map-entry-conv*:
assumes *next-instr* (*Fubx-get F2*) *f l pc* = *Some instr* ¬ *is-fun-call instr* ¬ *is-fun-call instr′*
shows
is-valid-fun-call (*Fubx-get* (*Subx.Fenv.map-entry F2 f* (λ*fd. rewrite-fundef-body fd l pc instr′*)))=
is-valid-fun-call (*Fubx-get F2*)
⟨*proof*⟩

lemma *is-valid-fun-call-map-entry-neq-f-neq-l*:
assumes *f* ≠ *g l* ≠ *l′*
shows
is-valid-fun-call (*Fubx-get* (*Subx.Fenv.map-entry F2 f* (λ*fd. rewrite-fundef-body fd l pc instr′*))) *g l′* =
is-valid-fun-call (*Fubx-get F2*) *g l′*
⟨*proof*⟩

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

rel-stacktraces-Cons:
rel-stacktraces F (*Some f*) *st1 st2* ⟹
Σ*1* = *map Subx.norm-unboxed* Σ*2* ⟹
R1 = *map Subx.norm-unboxed R2* ⟹
list-all is-dyn-operand R2 ⟹
F f = *Some fd2* ⟹ *map-of* (*body fd2*) *l* = *Some instrs* ⟹
Subx.sp-instrs (*map-option funtype* ∘ *F*) (*return fd2*) (*take pc instrs*) [] (*map typeof* Σ*2*) ⟹
pred-option (*is-valid-fun-call F f l pc* Σ*2*) *opt* ⟹
rel-stacktraces F opt (*Frame f l pc R1* Σ*1* # *st1*) (*Frame f l pc R2* Σ*2* # *st2*)

lemma *rel-stacktraces-map-entry-gneralize-fundefI*[*intro*]:
assumes *rel-stacktraces* (*Fubx-get F2*) *opt st1 st2*
shows *rel-stacktraces* (*Fubx-get* (*Subx.Fenv.map-entry F2 f Subx.generalize-fundef*))
opt st1 (*Subx.box-stack f st2*)
⟨*proof*⟩

lemma *rel-stacktraces-map-entry-rewrite-fundef-body*:
assumes
rel-stacktraces (*Fubx-get F2*) *opt st1 st2* **and**

next-instr (*Fubx-get F2*) *f l pc* = *Some instr* **and**
⋀*ret*. *Subx.sp-instr* (*map-option funtype* ∘ *Fubx-get F2*) *ret instr* =
Subx.sp-instr (*map-option funtype* ∘ *Fubx-get F2*) *ret instr′* **and**
¬ *is-fun-call instr* ¬ *is-fun-call instr′*
shows *rel-stacktraces*
(*Fubx-get* (*Subx.Fenv.map-entry F2 f* (λ*fd*. *rewrite-fundef-body fd l pc instr′*)))
opt st1 st2
⟨*proof*⟩

18 Simulation relation

inductive *match* (**infix** ∼ *55*) **where**
matchI: *Subx.wf-state* (*State F2 H st2*) ⟹
rel-fundefs (*Finca-get F1*) (*Fubx-get F2*) ⟹
rel-stacktraces (*Fubx-get F2*) *None st1 st2* ⟹
match (*State F1 H st1*) (*State F2 H st2*)

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

19 Backward simulation

lemma *map-eq-append-map-drop*:
map f xs = *ys* @ *map f* (*drop n xs*) ⟷ *map f* (*take n xs*) = *ys*
⟨*proof*⟩

lemma *ap-map-list-cast-Dyn-to-map-norm*:
assumes *ap-map-list cast-Dyn xs* = *Some ys*
shows *ys* = *map Subx.norm-unboxed xs*
⟨*proof*⟩

lemma *ap-map-list-cast-Dyn-to-all-Dyn*:
assumes *ap-map-list cast-Dyn xs* = *Some ys*
shows *list-all* (λ*x*. *typeof x* = *None*) *xs*
⟨*proof*⟩

lemma *ap-map-list-cast-Dyn-map-typeof-replicate-conv*:
assumes *ap-map-list cast-Dyn xs* = *Some ys* **and** *n* = *length xs*
shows *map typeof xs* = *replicate n None*
⟨*proof*⟩

lemma *cast-Dyn-eq-Some-conv-norm-unboxed*[*simp*]: *cast-Dyn i* = *Some i′* ⟹ *Subx.norm-unboxed i* = *i′*
⟨*proof*⟩

lemma *cast-Dyn-eq-Some-conv-typeof*[*simp*]: *cast-Dyn i* = *Some i′* ⟹ *typeof i* = *None*
⟨*proof*⟩

lemma *backward-lockstep-simulation*:
assumes *match s1 s2* **and** *Subx.step s2 s2'*
shows ∃ *s1'*. *Sinca.step s1 s1'* ∧ *match s1' s2'*
⟨*proof*⟩

lemma *match-final-backward*:
assumes *match s1 s2* **and** *final-s2*: *final Fubx-get Ubx.IReturn s2*
shows *final Finca-get Inca.IReturn s1*
⟨*proof*⟩

sublocale *inca-to-ubx-simulation*:
backward-simulation Sinca.step Subx.step
final Finca-get Inca.IReturn
final Fubx-get Ubx.IReturn
λ- -. *False* λ-. *match*
⟨*proof*⟩

20 Forward simulation

lemma *ap-map-list-cast-Dyn-eq-norm-stack*:
assumes *list-all* (λ*x*. *x* = *None*) (*map typeof xs*)
shows *ap-map-list cast-Dyn xs* = *Some* (*map Subx.norm-unboxed xs*)
⟨*proof*⟩

lemma *forward-lockstep-simulation*:
assumes *match s1 s2* **and** *Sinca.step s1 s1'*
shows ∃ *s2'*. *Subx.step s2 s2'* ∧ *match s1' s2'*
⟨*proof*⟩

lemma *match-final-forward*:
assumes *match s1 s2* **and** *final-s1*: *final Finca-get Inca.IReturn s1*
shows *final Fubx-get Ubx.IReturn s2*
⟨*proof*⟩

sublocale *inca-ubx-forward-simulation*:
forward-simulation Sinca.step Subx.step
final Finca-get Inca.IReturn
final Fubx-get Ubx.IReturn
λ- -. *False* λ-. *match*
⟨*proof*⟩

21 Bisimulation

sublocale *inca-ubx-bisimulation*:
bisimulation Sinca.step Subx.step final Finca-get Inca.IReturn final Fubx-get Ubx.IReturn
λ- -. *False* λ-. *match*
⟨*proof*⟩

end

end
theory *Inca-Verification*
imports *Inca*
begin

context *inca* **begin**

22 Strongest postcondition

inductive *sp-instr* **for** *F ret* **where**
Push:
sp-instr F ret (*IPush d*) Σ (*Suc* Σ) |
Pop:
sp-instr F ret IPop (*Suc* Σ) Σ |
Get:
sp-instr F ret (*IGet n*) Σ (*Suc* Σ) |
Set:
sp-instr F ret (*ISet n*) (*Suc* Σ) Σ |
Load:
sp-instr F ret (*ILoad x*) (*Suc* Σ) (*Suc* Σ) |
Store:
sp-instr F ret (*IStore x*) (*Suc* (*Suc* Σ)) Σ |
Op:
$\Sigma i = \mathfrak{Arity}\ op + \Sigma \Longrightarrow$
sp-instr F ret (*IOp op*) Σ*i* (*Suc* Σ) |
OpInl:
$\Sigma i = \mathfrak{Arity}\ (\mathfrak{DeInl}\ opinl) + \Sigma \Longrightarrow$
sp-instr F ret (*IOpInl opinl*) Σ*i* (*Suc* Σ) |
CJump:
sp-instr F ret (*ICJump* l_t l_f) *1 0* |
Call:
$F\ f = Some\ (ar, r) \Longrightarrow \Sigma i = ar + \Sigma \Longrightarrow \Sigma o = r + \Sigma \Longrightarrow$
sp-instr F ret (*ICall f*) Σ*i* Σ*o* |
Return: $\Sigma i = ret \Longrightarrow$
sp-instr F ret IReturn Σ*i 0*

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

inductive *sp-instrs* **for** *F ret* **where**
Nil:
sp-instrs F ret [] Σ Σ |
Cons:
sp-instr F ret instr Σ*i* Σ $\Longrightarrow$ *sp-instrs F ret instrs* Σ Σ*o* $\Longrightarrow$
sp-instrs F ret (*instr # instrs*) Σ*i* Σ*o*

23 Range validations

fun *local-var-in-range* **where**
local-var-in-range n (*IGet k*) ⟷ *k* < *n* |
local-var-in-range n (*ISet k*) ⟷ *k* < *n* |
local-var-in-range - - ⟷ *True*

fun *jump-in-range* **where**
jump-in-range L (*ICJump* l_t l_f) ⟷ {l_t, l_f} ⊆ *L* |
jump-in-range L - ⟷ *True*

fun *fun-call-in-range* **where**
fun-call-in-range F (*ICall f*) ⟷ *f* ∈ *dom F* |
fun-call-in-range F instr ⟷ *True*

24 Basic block validation

definition *wf-basic-block* **where**
wf-basic-block F L ret n bblock ⟷
(*let* (*label*, *instrs*) = *bblock in*
list-all (*local-var-in-range n*) *instrs* ∧
list-all (*jump-in-range L*) *instrs* ∧
list-all (*fun-call-in-range F*) *instrs* ∧
list-all (λ*i*. ¬ *Inca.is-jump i* ∧ ¬ *Inca.is-return i*) (*butlast instrs*) ∧
instrs ≠ [] ∧
sp-instrs F ret instrs 0 0)

25 Function definition validation

definition *wf-fundef* **where**
wf-fundef F fd ⟷
body fd ≠ [] ∧
list-all
(*wf-basic-block F* (*fst* ‘ *set* (*body fd*)) (*return fd*) (*arity fd* + *fundef-locals fd*))
(*body fd*)

26 Program definition validation

definition *wf-prog* **where**
wf-prog p ⟷
(*let F* = *F-get* (*prog-fundefs p*) *in*
pred-map (*wf-fundef* (*map-option funtype* ∘ *F*)) *F*)

end

end
theory *Inca-to-Ubx-compiler*
imports *Inca-to-Ubx-simulation Result*

Inca-Verification
VeriComp.Compiler
HOL−Library.Monad-Syntax
begin

27 Generic program rewriting

primrec *monadic-fold-map* **where**
monadic-fold-map f acc [] = *Some* (*acc*, []) |
monadic-fold-map f acc (*x* # *xs*) = *do* {
(*acc'*, *x'*) ← *f acc x*;
(*acc''*, *xs'*) ← *monadic-fold-map f acc' xs*;
Some (*acc''*, *x'* # *xs'*)
}

lemma *monadic-fold-map-length*:
monadic-fold-map f acc xs = *Some* (*acc'*, *xs'*) ⟹ *length xs* = *length xs'*
⟨*proof*⟩

lemma *monadic-fold-map-ConsD*[*dest*]:
assumes *monadic-fold-map f a* (*x* # *xs*) = *Some* (*c*, *ys*)
shows ∃ *y ys'* *b*. *ys* = *y* # *ys'* ∧ *f a x* = *Some* (*b*, *y*) ∧ *monadic-fold-map f b xs* = *Some* (*c*, *ys'*)
⟨*proof*⟩

lemma *monadic-fold-map-eq-Some-conv*:
monadic-fold-map f a (*x* # *xs*) = *Some* (*c*, *ys*) ⟷
(∃ *y ys'* *b*. *f a x* = *Some* (*b*, *y*) ∧ *monadic-fold-map f b xs* = *Some* (*c*, *ys'*) ∧ *ys* = *y* # *ys'*)
⟨*proof*⟩

lemma *monadic-fold-map-eq-Some-conv'*:
monadic-fold-map f a (*x* # *xs*) = *Some p* ⟷
(∃ *y ys'* *b*. *f a x* = *Some* (*b*, *y*) ∧ *monadic-fold-map f b xs* = *Some* (*fst p*, *ys'*) ∧ *snd p* = *y* # *ys'*)
⟨*proof*⟩

lemma *monadic-fold-map-list-all2*:
assumes *monadic-fold-map f acc xs* = *Some* (*acc'*, *ys*) **and**
⋀*acc acc' x y*. *f acc x* = *Some* (*acc'*, *y*) ⟹ *P x y*
shows *list-all2 P xs ys*
⟨*proof*⟩

lemma *monadic-fold-map-list-all*:
assumes *monadic-fold-map f acc xs* = *Some* (*acc'*, *ys*) **and**
⋀*acc acc' x y*. *f acc x* = *Some* (*acc'*, *y*) ⟹ *P y*
shows *list-all P ys*
⟨*proof*⟩

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*
Some (*instr*, *codomain* @ *drop ar* Σ)
else
None
)

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

28 Lifting

fun *lift-instr* :: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ - ⇒
((-, -, -, -, -, -, *'opubx*, *'ubx1*, *'ubx2*) *Ubx.instr* × -) *option* **where**
lift-instr F L ret N (*Inca.IPush d*) Σ = *Some* (*IPush d*, *None* # Σ) |
lift-instr F L ret N Inca.IPop (- # Σ) = *Some* (*IPop*, Σ) |
lift-instr F L ret N (*Inca.IGet n*) Σ = (*if n* < *N then Some* (*IGet n*, *None* # Σ) *else None*) |
lift-instr F L ret N (*Inca.ISet n*) (*None* # Σ) = (*if n* < *N then Some* (*ISet n*, Σ) *else None*) |
lift-instr F L ret N (*Inca.ILoad x*) (*None* # Σ) = *Some* (*ILoad x*, *None* # Σ) |
lift-instr F L ret N (*Inca.IStore x*) (*None* # *None* # Σ) = *Some* (*IStore x*, Σ) |
lift-instr F L ret N (*Inca.IOp op*) Σ =
gen-pop-push (*IOp op*) (*replicate* (𝔄𝔯𝔦𝔱𝔶 *op*) *None*, [*None*]) Σ |
lift-instr F L ret N (*Inca.IOpInl opinl*) Σ =
gen-pop-push (*IOpInl opinl*) (*replicate* (𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 *opinl*)) *None*, [*None*]) Σ |
lift-instr F L ret N (*Inca.ICJump* l_t l_f) [*None*] =
(*if List.member L* l_t ∧ *List.member L* l_f *then Some* (*ICJump* l_t l_f, []) *else None*) |
lift-instr F L ret N (*Inca.ICall f*) Σ = *do* {
(*ar*, *ret*) ← *F f*;
gen-pop-push (*ICall f*) (*replicate ar None*, *replicate ret None*) Σ
} |
lift-instr F L ret N Inca.IReturn Σ =
(*if* Σ = *replicate ret None then Some* (*IReturn*, []) *else None*) |
lift-instr - - - - - - = *None*

definition *lift-instrs* **where**
lift-instrs F L ret N ≡
monadic-fold-map (λΣ *instr*. *map-option prod.swap* (*lift-instr F L ret N instr* Σ))

lemma *lift-instrs-length*:
assumes *lift-instrs F L ret N* Σ*i xs* = *Some* (Σ*o*, *ys*)
shows *length xs* = *length ys*
⟨*proof*⟩

lemma *lift-instrs-not-Nil*: *lift-instrs F L ret N* Σ*i xs* = *Some* (Σ*o*, *ys*) ⟹ *xs* ≠ []

$\longleftrightarrow$ *ys* $\neq$ []
⟨*proof*⟩

lemma *lift-instrs-NilD*[*dest*]:
assumes *lift-instrs F L ret N* Σ*i* [] = *Some* (Σ*o*, *ys*)
shows Σ*o* = Σ*i* ∧ *ys* = []
⟨*proof*⟩

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

lemma *lift-instr-is-jump*:
assumes *lift-instr F L ret N x* Σ*i* = *Some* (*y*, Σ*o*)
shows *Inca.is-jump x* $\longleftrightarrow$ *Ubx.is-jump y*
⟨*proof*⟩

lemma *lift-instr-is-return*:
assumes *lift-instr F L ret N x* Σ*i* = *Some* (*y*, Σ*o*)
shows *Inca.is-return x* $\longleftrightarrow$ *Ubx.is-return y*
⟨*proof*⟩

lemma *lift-instrs-all-not-jump-not-return*:
assumes *lift-instrs F L ret N* Σ*i xs* = *Some* (Σ*o*, *ys*)
shows
list-all (λ*i*. ¬ *Inca.is-jump i* ∧ ¬ *Inca.is-return i*) *xs* $\longleftrightarrow$
list-all (λ*i*. ¬ *Ubx.is-jump i* ∧ ¬ *Ubx.is-return i*) *ys*
⟨*proof*⟩

lemma *lift-instrs-all-butlast-not-jump-not-return*:
assumes *lift-instrs F L ret N* Σ*i xs* = *Some* (Σ*o*, *ys*)
shows
list-all (λ*i*. ¬ *Inca.is-jump i* ∧ ¬ *Inca.is-return i*) (*butlast xs*) $\longleftrightarrow$
list-all (λ*i*. ¬ *Ubx.is-jump i* ∧ ¬ *Ubx.is-return i*) (*butlast ys*)
⟨*proof*⟩

lemma *lift-instr-sp*:
assumes *lift-instr F L ret N x* Σ*i* = *Some* (*y*, Σ*o*)
shows *Subx.sp-instr F ret y* Σ*i* Σ*o*
⟨*proof*⟩

lemma *lift-instrs-sp*:
assumes *lift-instrs F L ret N* Σ*i xs* = *Some* (Σ*o*, *ys*)
shows *Subx.sp-instrs F ret ys* Σ*i* Σ*o*
⟨*proof*⟩

lemma *lift-instr-fun-call-in-range*:
assumes *lift-instr F L ret N x* Σ*i* = *Some* (*y*, Σ*o*)
shows *Subx.fun-call-in-range F y*
⟨*proof*⟩

lemma *lift-instrs-all-fun-call-in-range*:
assumes *lift-instrs F L ret N Σi xs = Some (Σo, ys)*
shows *list-all (Subx.fun-call-in-range F) ys*
⟨*proof*⟩

lemma *lift-instr-local-var-in-range*:
assumes *lift-instr F L ret N x Σi = Some (y, Σo)*
shows *Subx.local-var-in-range N y*
⟨*proof*⟩

lemma *lift-instrs-all-local-var-in-range*:
assumes *lift-instrs F L ret N Σi xs = Some (Σo, ys)*
shows *list-all (Subx.local-var-in-range N) ys*
⟨*proof*⟩

lemma *lift-instr-jump-in-range*:
assumes *lift-instr F L ret N x Σi = Some (y, Σo)*
shows *Subx.jump-in-range (set L) y*
⟨*proof*⟩

lemma *lift-instrs-all-jump-in-range*:
assumes *lift-instrs F L ret N Σi xs = Some (Σo, ys)*
shows *list-all (Subx.jump-in-range (set L)) ys*
⟨*proof*⟩

lemma *lift-instr-norm*:
lift-instr F L ret N instr1 Σ1 = Some (instr2, Σ2) ⟹ norm-eq instr1 instr2
⟨*proof*⟩

lemma *lift-instrs-all-norm*:
assumes *lift-instrs F L ret N Σ1 instrs1 = Some (Σ2, instrs2)*
shows *list-all2 norm-eq instrs1 instrs2*
⟨*proof*⟩

29 Optimization

context
fixes *load-oracle* :: *nat ⇒ type option*
begin

definition *orelse* :: *'a option ⇒ 'a option ⇒ 'a option* (**infixr** *orelse 55*) **where**
x orelse y = (case x of Some x' ⇒ Some x' | None ⇒ y)

lemma *None-orelse*[*simp*]: *None orelse y = y*
⟨*proof*⟩

lemma *orelse-None*[*simp*]: *x orelse None = x*
⟨*proof*⟩

lemma *Some-orelse*[*simp*]: *Some x orelse y = Some x*
⟨*proof*⟩

lemma *orelse-eq-Some-conv*:
x orelse y = Some z ⟷ (*x = Some z* ∨ *x = None* ∧ *y = Some z*)
⟨*proof*⟩

lemma *orelse-eq-SomeE*:
assumes
x orelse y = Some z **and**
x = Some z ⟹ *P* **and**
x = None ⟹ *y = Some z* ⟹ *P*
shows *P*
⟨*proof*⟩

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

lemma *drop-prefix-append-prefix*[*simp*]: *drop-prefix xs* (*xs* @ *ys*) = *Some ys*
⟨*proof*⟩

lemma *drop-prefix-eq-Some-conv*: *drop-prefix xs ys = Some zs* ⟷ *ys = xs* @ *zs*
⟨*proof*⟩

fun *optim-instr* **where**
optim-instr - - - (*IPush d*) Σ =
Some Pair ⋄ (*Some IPushUbx1* ⋄ (*unbox-ubx1 d*)) ⋄ *Some* (*Some Ubx1* # Σ)
orelse
Some Pair ⋄ (*Some IPushUbx2* ⋄ (*unbox-ubx2 d*)) ⋄ *Some* (*Some Ubx2* # Σ)
orelse
Some (*IPush d*, *None* # Σ)
|
optim-instr - - - (*IPushUbx1 n*) Σ = *Some* (*IPushUbx1 n*, *Some Ubx1* # Σ) |
optim-instr - - - (*IPushUbx2 b*) Σ = *Some* (*IPushUbx2 b*, *Some Ubx2* # Σ) |
optim-instr - - - IPop (- # Σ) = *Some* (*IPop*, Σ) |
optim-instr - - pc (*IGet n*) Σ =
map-option (λτ. (*IGetUbx* τ *n*, *Some* τ # Σ)) (*load-oracle pc*) *orelse*
Some (*IGet n*, *None* # Σ) |
optim-instr - - pc (*IGetUbx* τ *n*) Σ = *Some* (*IGetUbx* τ *n*, *Some* τ # Σ) |
optim-instr - - - (*ISet n*) (*None* # Σ) = *Some* (*ISet n*, Σ) |
optim-instr - - - (*ISet n*) (*Some* τ # Σ) = *Some* (*ISetUbx* τ *n*, Σ) |
optim-instr - - - (*ISetUbx - n*) (*None* # Σ) = *Some* (*ISet n*, Σ) |
optim-instr - - - (*ISetUbx - n*) (*Some* τ # Σ) = *Some* (*ISetUbx* τ *n*, Σ) |
optim-instr - - pc (*ILoad x*) (*None* # Σ) =
map-option (λτ. (*ILoadUbx* τ *x*, *Some* τ # Σ)) (*load-oracle pc*) *orelse*
Some (*ILoad x*, *None* # Σ) |

```
  optim-instr - - - (ILoadUbx τ x) (None # Σ) = Some (ILoadUbx τ x, Some τ # Σ) |
  optim-instr - - - (IStore x) (None # None # Σ) = Some (IStore x, Σ) |
  optim-instr - - - (IStore x) (None # Some τ # Σ) = Some (IStoreUbx τ x, Σ) |
  optim-instr - - - (IStoreUbx - x) (None # None # Σ) = Some (IStore x, Σ) |
  optim-instr - - - (IStoreUbx - x) (None # Some τ # Σ) = Some (IStoreUbx τ x, Σ) |
  optim-instr - - - (IOp op) Σ =
     map-option (λΣo. (IOp op, None # Σo)) (drop-prefix (replicate (𝔄𝔯𝔦𝔱𝔶 op) 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 ⇒ map-option (λΣo. (IOpInl opinl, None # Σo)) (drop-prefix (replicate ar None) Σ) |
        Some opubx ⇒ map-option (λΣo. (IOpUbx opubx, snd (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx) # Σo))
          (drop-prefix (fst (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx)) Σ)
    else
      None
  ) |
  optim-instr - - - (IOpUbx opubx) Σ =
    (let p = 𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx in
     map-option (λΣo. (IOpUbx opubx, snd p # Σo)) (drop-prefix (fst p) Σ)) |
  optim-instr - - - (ICJump l_t l_f) [None] = Some (ICJump l_t l_f, []) |
  optim-instr F - - (ICall f) Σ = do {
    (ar, ret) ← F f;
    Σo ← drop-prefix (replicate ar None) Σ;
    Some (ICall f, replicate ret None @ Σo)
  } |
  optim-instr - ret - IReturn Σ = (if Σ = replicate ret None then Some (IReturn, []) else None) |
  optim-instr - - - - - = None

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

lemma optim-instrs-Cons-eq-Some-conv:
  optim-instrs F ret pc Σi (instr # instrs) = Some (Σo, ys) ⟷ (∃ y ys' Σ.
    ys = y # ys' ∧
    optim-instr F ret pc instr Σi = Some (y, Σ) ∧
    optim-instrs F ret (Suc pc) Σ instrs = Some (Σo, ys'))
  ⟨proof⟩
```

lemma *optim-instrs-length*:
assumes *optim-instrs F ret pc* Σi *xs* = *Some* (Σo, *ys*)
shows *length xs* = *length ys*
⟨*proof*⟩

lemma *optim-instrs-not-Nil*: *optim-instrs F ret pc* Σi *xs* = *Some* (Σo, *ys*) $\Longrightarrow$ *xs* ≠ [] ⟷ *ys* ≠ []
⟨*proof*⟩

lemma *optim-instrs-NilD*[*dest*]:
assumes *optim-instrs F ret pc* Σi [] = *Some* (Σo, *ys*)
shows $\Sigma o = \Sigma i \wedge$ *ys* = []
⟨*proof*⟩

lemma *optim-instrs-ConsD*[*dest*]:
assumes *optim-instrs F ret pc* Σi (*x* # *xs*) = *Some* (Σo, *ys*)
shows $\exists$ *y ys'* Σ. *ys* = *y* # *ys'* ∧
optim-instr F ret pc x Σi = *Some* (*y*, Σ) ∧
optim-instrs F ret (*Suc pc*) Σ *xs* = *Some* (Σo, *ys'*)
⟨*proof*⟩

lemma *optim-instr-norm*:
assumes *optim-instr F ret pc instr1* $\Sigma 1$ = *Some* (*instr2*, $\Sigma 2$)
shows *norm-instr instr1* = *norm-instr instr2*
⟨*proof*⟩

lemma *optim-instrs-all-norm*:
assumes *optim-instrs F ret pc* $\Sigma 1$ *instrs1* = *Some* ($\Sigma 2$, *instrs2*)
shows *list-all2* (λ*i1 i2*. *norm-instr i1* = *norm-instr i2*) *instrs1 instrs2*
⟨*proof*⟩

lemma *optim-instr-is-jump*:
assumes *optim-instr F ret pc x* Σi = *Some* (*y*, Σo)
shows *is-jump x* ⟷ *is-jump y*
⟨*proof*⟩

lemma *optim-instr-is-return*:
assumes *optim-instr F ret pc x* Σi = *Some* (*y*, Σo)
shows *is-return x* ⟷ *is-return y*
⟨*proof*⟩

lemma *optim-instrs-all-butlast-not-jump-not-return*:
assumes *optim-instrs F ret pc* Σi *xs* = *Some* (Σo, *ys*)
shows
list-all (λ*i*. ¬ *is-jump i* ∧ ¬ *is-return i*) (*butlast xs*) ⟷
list-all (λ*i*. ¬ *is-jump i* ∧ ¬ *is-return i*) (*butlast ys*)
⟨*proof*⟩

lemma *optim-instr-jump-in-range*:
assumes *optim-instr F ret pc x* Σi = *Some* (y, Σo)
shows *Subx.jump-in-range L x* $\longleftrightarrow$ *Subx.jump-in-range L y*
⟨*proof*⟩

lemma *optim-instrs-all-jump-in-range*:
assumes *optim-instrs F ret pc* Σi *xs* = *Some* (Σo, *ys*)
shows *list-all* (*Subx.jump-in-range L*) *xs* $\longleftrightarrow$ *list-all* (*Subx.jump-in-range L*) *ys*
⟨*proof*⟩

lemma *optim-instr-fun-call-in-range*:
assumes *optim-instr F ret pc x* Σi = *Some* (y, Σo)
shows *Subx.fun-call-in-range F x* $\longleftrightarrow$ *Subx.fun-call-in-range F y*
⟨*proof*⟩

lemma *optim-instrs-all-fun-call-in-range*:
assumes *optim-instrs F ret pc* Σi *xs* = *Some* (Σo, *ys*)
shows *list-all* (*Subx.fun-call-in-range F*) *xs* $\longleftrightarrow$ *list-all* (*Subx.fun-call-in-range F*) *ys*
⟨*proof*⟩

lemma *optim-instr-local-var-in-range*:
assumes *optim-instr F ret pc x* Σi = *Some* (y, Σo)
shows *Subx.local-var-in-range N x* $\longleftrightarrow$ *Subx.local-var-in-range N y*
⟨*proof*⟩

lemma *optim-instrs-all-local-var-in-range*:
assumes *optim-instrs F ret pc* Σi *xs* = *Some* (Σo, *ys*)
shows *list-all* (*Subx.local-var-in-range N*) *xs* $\longleftrightarrow$ *list-all* (*Subx.local-var-in-range N*) *ys*
⟨*proof*⟩

lemma *optim-instr-sp*:
assumes *optim-instr F ret pc x* Σi = *Some* (y, Σo)
shows *Subx.sp-instr F ret y* Σi Σo
⟨*proof*⟩

lemma *optim-instrs-sp*:
assumes *optim-instrs F ret pc* Σi *xs* = *Some* (Σo, *ys*)
shows *Subx.sp-instrs F ret ys* Σi Σo
⟨*proof*⟩

30 Compilation of function definition

definition *compile-basic-block* **where**
compile-basic-block F L ret N ≡
ap-map-prod Some (λ*i1*. *do* {
- ← *if i1* ≠ [] *then Some* () *else None*;
- ← *if list-all* (λ*i*. ¬ *Inca.is-jump i* ∧ ¬ *Inca.is-return i*) (*butlast i1*) *then*

```
Some () else None;
      (Σo, i2) ← lift-instrs F L ret N ([] :: type option list) i1;
      if Σo = [] then
        case optim-instrs F ret 0 ([] :: type option list) i2 of
          Some (Σo', i2') ⇒ Some (if Σo' = [] then i2' else i2) |
          None ⇒ Some i2
      else
        None
    })

lemma compile-basic-block-rel-prod-all-norm-eq:
  assumes compile-basic-block F L ret N bblock1 = Some bblock2
  shows rel-prod (=) (list-all2 norm-eq) bblock1 bblock2
  ⟨proof⟩

lemma list-all-iff-butlast-last:
  assumes xs ≠ []
  shows list-all P xs ⟷ list-all P (butlast xs) ∧ P (last xs)
  ⟨proof⟩

lemma compile-basic-block-wf:
  assumes compile-basic-block F L ret N x = Some y
  shows Subx.wf-basic-block F (set L) ret N y
⟨proof⟩

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

lemma compile-fundef-arities: compile-fundef F fd1 = Some fd2 ⟹ arity fd1 =
arity fd2
  ⟨proof⟩

lemma compile-fundef-returns: compile-fundef F fd1 = Some fd2 ⟹ return fd1
= return fd2
  ⟨proof⟩

lemma compile-fundef-locals:
  compile-fundef F fd1 = Some fd2 ⟹ fundef-locals fd1 = fundef-locals fd2
  ⟨proof⟩

lemma if-then-None-else-Some-eq[simp]:
  (if a then None else Some b) = Some c ⟷ ¬ a ∧ b = c
  (if a then None else Some b) = None ⟷ a
  ⟨proof⟩
```

lemma
assumes *compile-fundef F fd1 = Some fd2*
shows
rel-compile-fundef: *rel-fundef* (=) *norm-eq fd1 fd2* (**is** *?REL*) **and**
wf-compile-fundef: *Subx.wf-fundef F fd2* (**is** *?WF*)
⟨*proof*⟩

end

end

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

31 Compilation of function environment

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

lemma *rel-compile-env-entry*:
assumes *compile-env-entry F* (*f*, *fd1*) = *Some* (*f*, *fd2*)
shows *rel-fundef* (=) *norm-eq fd1 fd2*
⟨*proof*⟩

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

lemma *rel-ap-map-list-ap-map-list-compile-env-entries*:
assumes *ap-map-list* (*compile-env-entry F*) *xs* = *Some ys*
shows *rel-fundefs* (*Finca-get* (*Sinca.Fenv.from-list xs*)) (*Fubx-get* (*Subx.Fenv.from-list ys*))
⟨*proof*⟩

lemma *rel-fundefs-compile-env*:
assumes *compile-env F1* = *Some F2*
shows *rel-fundefs* (*Finca-get F1*) (*Fubx-get F2*)

⟨*proof*⟩

32 Compilation of program

fun *compile* **where**
compile (*Prog F1 H f*) = *Some Prog* ⋄ *compile-env F1* ⋄ *Some H* ⋄ *Some f*

lemma *ap-map-list-cong*:
assumes ⋀*x*. *x* ∈ *set ys* ⟹ *f x* = *g x* **and** *xs* = *ys*
shows *ap-map-list f xs* = *ap-map-list g ys*
⟨*proof*⟩

lemma *compile-env-wf-fundefs*:
assumes *compile-env F1* = *Some F2*
shows *Subx.wf-fundefs* (*Fubx-get F2*)
⟨*proof*⟩

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*⟩

interpretation *std-to-inca-compiler*:
compiler Sinca.step Subx.step final Finca-get Inca.IReturn final Fubx-get Ubx.IReturn
Sinca.load Subx.load
λ- -. False λ-. match compile
⟨*proof*⟩

32.1 Completeness of compilation

lemma *lift-instr-None-preservation*:
assumes *lift-instr F L ret N instr* Σ = *Some* (*instr′*, Σ′) **and** *list-all* ((=) *None*) Σ
shows *list-all* ((=) *None*) Σ′
⟨*proof*⟩

lemma *lift-instr-complete*:
assumes
Sinca.local-var-in-range N instr **and**
Sinca.jump-in-range (*set L*) *instr* **and**
Sinca.fun-call-in-range F instr **and**
Sinca.sp-instr F ret instr (*length* Σ) *k* **and**
list-all ((=) *None*) Σ
shows ∃ *instr′* Σ′. *lift-instr F L ret N instr* Σ = *Some* (*instr′*, Σ′) ∧ *length* Σ′ = *k*
⟨*proof*⟩

lemma *lift-instrs-complete*:
fixes Σ :: *type option list*
assumes
list-all (*Sinca.local-var-in-range N*) *instrs* **and**
list-all (*Sinca.jump-in-range* (*set L*)) *instrs* **and**
list-all (*Sinca.fun-call-in-range F*) *instrs* **and**
Sinca.sp-instrs F ret instrs (*length* Σ) *k* **and**
list-all ((=) *None*) Σ
shows $\exists \Sigma'$ *instrs'*. *lift-instrs F L ret N* Σ *instrs* = *Some* (Σ', *instrs'*) $\wedge$ *length* $\Sigma' = k$
⟨*proof*⟩

lemma *optim-instr-complete*:
assumes *sp*: *Subx.sp-instr F ret instr* Σ Σ'
shows $\exists \Sigma''$ *instr'*. *optim-instr* $\mathcal{O}$ *F ret pc instr* Σ = *Some* (*instr'*, Σ'') $\wedge$ *length* Σ' = *length* Σ''
⟨*proof*⟩

lemma *compile-basic-block-complete*:
assumes *wf-bblock1*: *Sinca.wf-basic-block F* (*set L*) *ret n bblock1*
shows $\exists$ *bblock2*. *compile-basic-block* $\mathcal{O}$ *F L ret n bblock1* = *Some bblock2*
⟨*proof*⟩

lemma *bind-eq-map-option*[*simp*]: *x* $\gg=$ (λ*y*. *Some* (*f y*)) = *map-option f x*
⟨*proof*⟩

lemma *compile-fundef-complete*:
assumes *wf-fd1*: *Sinca.wf-fundef F fd1*
shows $\exists$ *fd2*. *compile-fundef* $\mathcal{O}$ *F fd1* = *Some fd2*
⟨*proof*⟩

lemma *compile-env-entry-complete*:
assumes *wf-fd1*: *Sinca.wf-fundef F fd1*
shows $\exists$ *fd2*. *compile-env-entry F* (*f*, *fd1*) = *Some fd2*
⟨*proof*⟩

lemma *compile-env-complete*:
assumes *wf-F1*: *pred-map* (*Sinca.wf-fundef* (*map-option funtype* $\circ$ *Finca-get F1*)) (*Finca-get F1*)
shows $\exists$ *F2*. *compile-env F1* = *Some F2*
⟨*proof*⟩

theorem *compile-complete*:
assumes *wf-p1*: *Sinca.wf-prog p1*
shows $\exists$ *p2*. *compile p1* = *Some p2*
⟨*proof*⟩

end

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

33 Dynamic values

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

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

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

interpretation *dynval-dynamic*: *dynval DNil is-true is-false*
⟨*proof*⟩

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

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

interpretation *unboxed-dynamic*:
unboxedval DNil is-true is-false DNum unbox-num DBool unbox-bool
⟨*proof*⟩

34 Normal operations

datatype *op* =
OpNeg |
OpAdd |
OpMul

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

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

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

eval-Add [*DNum x*, *DNum y*] = *DNum* (*x* + *y*) |
eval-Add [-, -] = *DNil*

fun *eval-Mul* :: *dynamic list* ⇒ *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*
⟨*proof*⟩

interpretation *op-Op*: *nary-operations eval ar*
⟨*proof*⟩

35 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*
⟨*proof*⟩

lemma *inl-invertible*: *inl op xs* = *Some opinl* ⟹ *deinl opinl* = *op*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eval-MulNum-correct*:
length xs = *2* ⟹ *eval-MulNum xs* = *eval-Mul xs*
⟨*proof*⟩

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

lemma *inl-isinl*:
inl op xs = *Some opinl* ⟹ *isinl opinl xs*
⟨*proof*⟩

interpretation *op-OpInl*: *nary-operations-inl eval ar eval-inl inl isinl deinl*
⟨*proof*⟩

36 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*
⟨*proof*⟩

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

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

lemma *eval-ubx-correct*:
eval-ubx opubx xs = Some z ⟹
eval-inl (deubx opubx) (map unboxed-dynamic.norm-unboxed xs) = unboxed-dynamic.norm-unboxed z
⟨*proof*⟩

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*⟩

36.1 Typing

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
⟨*proof*⟩

lemma *typeof-opubx-correct*:
typeof-opubx opubx = (map typeof xs, codomain) ⟹
∃ y. eval-ubx opubx xs = Some y ∧ typeof y = codomain
⟨*proof*⟩

lemma *typeof-opubx-complete*:
eval-ubx opubx xs = Some y ⟹
typeof-opubx opubx = (map typeof xs, typeof y)
⟨*proof*⟩

lemma *typeof-opubx-ar*: *length (fst (typeof-opubx opubx)) = ar (deinl (deubx opubx))*
⟨*proof*⟩

interpretation *op-OpUbx*:
nary-operations-ubx
eval ar eval-inl inl isinl deinl
DNil is-true is-false DNum unbox-num DBool unbox-bool
eval-ubx ubx deubx typeof-opubx
⟨*proof*⟩

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

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

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

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

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

— n-ary operations
$\mathfrak{Op}$ :: *'op* ⇒ *'dyn list* ⇒ *'dyn* **and** $\mathfrak{Arity}$
begin

inductive *step* (**infix** → *55*) **where**
step-push:
next-instr (*F-get F*) *f l pc* = *Some* (*IPush d*) ⟹
State F H (*Frame f l pc R* Σ # *st*) → *State F H* (*Frame f l* (*Suc pc*) *R* (*d* # Σ) # *st*) |

step-pop:
next-instr (*F-get F*) *f l pc* = *Some IPop* ⟹
State F H (*Frame f l pc R* (*d* # Σ) # *st*) → *State F H* (*Frame f l* (*Suc pc*) *R* Σ # *st*) |

step-get:

$\mathit{next\text{-}instr}\ (\mathit{F\text{-}get}\ F)\ f\ l\ pc = \mathit{Some}\ (\mathit{IGet}\ n) \Longrightarrow$
$n < \mathit{length}\ R \Longrightarrow d = R\ !\ n \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ pc\ R\ \Sigma\ \#\ st) \rightarrow \mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ pc)\ R\ (d\ \#\ \Sigma)\ \#\ st)\ |$

step-set:
$\mathit{next\text{-}instr}\ (\mathit{F\text{-}get}\ F)\ f\ l\ pc = \mathit{Some}\ (\mathit{ISet}\ n) \Longrightarrow$
$n < \mathit{length}\ R \Longrightarrow\ R' = R[n := d] \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ pc\ R\ (d\ \#\ \Sigma)\ \#\ st) \rightarrow \mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ pc)\ R'\ \Sigma\ \#\ st)\ |$

step-load:
$\mathit{next\text{-}instr}\ (\mathit{F\text{-}get}\ F)\ f\ l\ pc = \mathit{Some}\ (\mathit{ILoad}\ x) \Longrightarrow$
$\mathit{heap\text{-}get}\ H\ (x,\ y) = \mathit{Some}\ d \Longrightarrow$
$\mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ pc\ R\ (y\ \#\ \Sigma)\ \#\ st) \rightarrow \mathit{State}\ F\ H\ (\mathit{Frame}\ f\ l\ (\mathit{Suc}\ pc)\ R\ (d\ \#\ \Sigma)\ \#\ st)\ |$

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

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

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

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

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

lemma *step-deterministic*:
assumes *s1* → *s2* **and** *s1* → *s3*
shows *s2* = *s3*
⟨*proof*⟩

lemma *step-right-unique*: *right-unique step*
⟨*proof*⟩

lemma *final-finished*:
assumes *final F-get IReturn s*
shows *finished step s*
⟨*proof*⟩

sublocale *semantics step final F-get IReturn*
⟨*proof*⟩

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

sublocale *language step final F-get IReturn load*
⟨*proof*⟩

end

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

37 Generic definitions

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

Fstd-add **and** *Fstd-to-list* **and**

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

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

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

— n-ary operations
𝔒𝔭 :: *'op* ⇒ *'dyn list* ⇒ *'dyn* **and** 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.IGet n*) = *Std.IGet n* |
norm-instr (*Inca.ISet n*) = *Std.ISet n* |
norm-instr (*Inca.ILoad x*) = *Std.ILoad x* |
norm-instr (*Inca.IStore x*) = *Std.IStore x* |
norm-instr (*Inca.IOp op*) = *Std.IOp op* |
norm-instr (*Inca.IOpInl opinl*) = *Std.IOp* (𝔇𝔢𝔍𝔫𝔩 *opinl*) |
norm-instr (*Inca.ICJump* l_t l_f) = *Std.ICJump* l_t l_f |
norm-instr (*Inca.ICall x*) = *Std.ICall x* |
norm-instr Inca.IReturn = *Std.IReturn*

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

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

lemma *rel-fundefsI*:
assumes ⋀*x*. *rel-option* (*rel-fundef* (=) *norm-eq*) (*F1 x*) (*F2 x*)
shows *rel-fundefs F1 F2*
⟨*proof*⟩

lemma *rel-fundefsD*:
assumes *rel-fundefs F1 F2*
shows *rel-option* (*rel-fundef* (=) *norm-eq*) (*F1 x*) (*F2 x*)
⟨*proof*⟩

lemma *rel-fundefs-next-instr*:
assumes *rel-F1-F2*: *rel-fundefs F1 F2*
shows *rel-option norm-eq* (*next-instr F1 f l pc*) (*next-instr F2 f l pc*)
⟨*proof*⟩

lemma *rel-fundefs-next-instr1*:
assumes *rel-F1-F2*: *rel-fundefs F1 F2* **and** *next-instr1*: *next-instr F1 f l pc* = *Some instr1*
shows ∃ *instr2*. *next-instr F2 f l pc* = *Some instr2* ∧ *norm-eq instr1 instr2*
⟨*proof*⟩

lemma *rel-fundefs-next-instr2*:
assumes *rel-F1-F2*: *rel-fundefs F1 F2* **and** *next-instr2*: *next-instr F2 f l pc* = *Some instr2*
shows ∃ *instr1*. *next-instr F1 f l pc* = *Some instr1* ∧ *norm-eq instr1 instr2*
⟨*proof*⟩

lemma *rel-fundefs-empty*: *rel-fundefs* (λ-. *None*) (λ-. *None*)
⟨*proof*⟩

lemma *rel-fundefs-None1*:
assumes *rel-fundefs f g* **and** *f x* = *None*
shows *g x* = *None*
⟨*proof*⟩

lemma *rel-fundefs-None2*:
assumes *rel-fundefs f g* **and** *g x* = *None*
shows *f x* = *None*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *rel-fundefs-rewriteI2*:
assumes
rel-F1-F2: *rel-fundefs* (*Fstd-get F1*) (*Finca-get F2*) **and**
next-instr1: *next-instr* (*Fstd-get F1*) *f l pc* = *Some instr1*

norm-eq instr1 instr2′
shows *rel-fundefs* (*Fstd-get F1*)
(*Finca-get* (*Sinca.Fenv.map-entry F2 f* (λ*fd*. *rewrite-fundef-body fd l pc instr2′*)))
(**is** *rel-fundefs* (*Fstd-get ?F1′*) (*Finca-get ?F2′*))
⟨*proof*⟩

38 Simulation relation

inductive *match* (**infix** ∼ *55*) **where**
wf-fundefs (*Fstd-get F1*) ⟹
rel-fundefs (*Fstd-get F1*) (*Finca-get F2*) ⟹
(*State F1 H st*) ∼ (*State F2 H st*)

39 Backward simulation

lemma *backward-lockstep-simulation*:
assumes *Sinca.step s2 s2′* **and** *match s1 s2*
shows ∃ *s1′*. *Sstd.step s1 s1′* ∧ *match s1′ s2′*
⟨*proof*⟩

lemma *match-final-backward*:
assumes *match s1 s2* **and** *final-s2*: *final Finca-get Inca.IReturn s2*
shows *final Fstd-get Std.IReturn s1*
⟨*proof*⟩

sublocale *std-inca-simulation*:
backward-simulation **where**
step1 = *Sstd.step* **and** *final1* = *final Fstd-get Std.IReturn* **and**
step2 = *Sinca.step* **and** *final2* = *final Finca-get Inca.IReturn* **and**
order = λ- -. *False* **and** *match* = λ-. *match*
⟨*proof*⟩

40 Forward simulation

lemma *forward-lockstep-simulation*:
assumes *Sstd.step s1 s1′* **and** *match s1 s2*
shows ∃ *s2′*. *Sinca.step s2 s2′* ∧ *s1′* ∼ *s2′*
⟨*proof*⟩

lemma *match-final-forward*:
assumes *match s1 s2* **and** *final-s1*: *final Fstd-get Std.IReturn s1*
shows *final Finca-get Inca.IReturn s2*
⟨*proof*⟩

sublocale *std-inca-forward-simulation*:
forward-simulation **where**
step1 = *Sstd.step* **and** *final1* = *final Fstd-get Std.IReturn* **and**
step2 = *Sinca.step* **and** *final2* = *final Finca-get Inca.IReturn* **and**

order = λ- -. *False* **and** *match* = λ-. *match*
⟨*proof*⟩

41 Bisimulation

sublocale *std-inca-bisimulation*:
bisimulation **where**
step1 = *Sstd.step* **and** *final1* = *final Fstd-get Std.IReturn* **and**
step2 = *Sinca.step* **and** *final2* = *final Finca-get Inca.IReturn* **and**
order = λ- -. *False* **and** *match* = λ-. *match*
⟨*proof*⟩

end

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

41.1 Compilation of function definitions

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

fun *compile-fundef* **where**
compile-fundef (*Fundef* [] - - -) = *None* |
compile-fundef (*Fundef bblocks ar ret locals*) =
Some (*Fundef* (*map-ran* (λ-. *map compile-instr*) *bblocks*) *ar ret locals*)

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

lemma *lambda-eq-eq*[*simp*]: (λ*x y*. *y* = *x*) = (=)
⟨*proof*⟩

lemma *norm-compile-instr*:
norm-instr (*compile-instr instr*) = *instr*
⟨*proof*⟩

lemma *rel-compile-fundef*:

assumes *compile-fundef fd1 = Some fd2*
shows *rel-fundef (=) norm-eq fd1 fd2*
⟨*proof*⟩

lemma *rel-fundef-imp-fundef-ok-iff*:
assumes *rel-fundef (=) norm-eq fd1 fd2*
shows *wf-fundef fd1 ⟷ wf-fundef fd2*
⟨*proof*⟩

lemma *rel-fundefs-imp-wf-fundefs-iff*:
assumes *rel-f-g*: *rel-fundefs f g*
shows *wf-fundefs f ⟷ wf-fundefs g*
⟨*proof*⟩

lemma *compile-fundef-wf*:
assumes *compile-fundef fd = Some fd'*
shows *wf-fundef fd'*
⟨*proof*⟩

41.2 Compilation of function environments

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

lemma *compile-env-imp-rel-option*:
assumes *compile-env F1 = Some F2*
shows *rel-option (λfd1 fd2. compile-fundef fd1 = Some fd2) (Fstd-get F1 f) (Finca-get F2 f)*
⟨*proof*⟩

lemma *Finca-get-compile*:
assumes *compile-F1*: *compile-env F1 = Some F2*
shows *Finca-get F2 f = Fstd-get F1 f ⤜ compile-fundef*
⟨*proof*⟩

lemma *compile-env-rel-fundefs*:
assumes *compile-env F1 = Some F2*
shows *rel-fundefs (Fstd-get F1) (Finca-get F2)*
⟨*proof*⟩

lemma *compile-env-imp-wf-fundefs2*:
assumes *compile-env F1 = Some F2*
shows *wf-fundefs (Finca-get F2)*
⟨*proof*⟩

41.3 Compilation of programs

fun *compile* **where**

compile (*Prog F1 H f*) = *Some Prog* ⋄ *compile-env F1* ⋄ *Some H* ⋄ *Some f*

theorem *compile-load*:
assumes
compile-p1: *compile p1* = *Some p2* **and**
load: *Sinca.load p2 s2*
shows ∃ *s1*. *Sstd.load p1 s1* ∧ *match s1 s2*
⟨*proof*⟩

sublocale *std-to-inca-compiler*:
compiler **where**
step1 = *Sstd.step* **and** *final1* = *final Fstd-get Std.IReturn* **and** *load1* = *Sstd.load* **and**
step2 = *Sinca.step* **and** *final2* = *final Finca-get Inca.IReturn* **and** *load2* = *Sinca.load* **and**
order = λ- -. *False* **and** *match* = λ-. *match* **and** *compile* = *compile*
⟨*proof*⟩

41.4 Completeness of compilation

lemma *compile-fundef-complete*:
assumes *wf-fundef fd1*
shows ∃ *fd2*. *compile-fundef fd1* = *Some fd2*
⟨*proof*⟩

lemma *compile-env-complete*:
assumes *wf-F1*: *wf-fundefs* (*Fstd-get F1*)
shows ∃ *F2*. *compile-env F1* = *Some F2*
⟨*proof*⟩

theorem *compile-complete*:
assumes *wf-p1*: *wf-prog Fstd-get p1*
shows ∃ *p2*. *compile p1* = *Some p2*
⟨*proof*⟩

theorem *compile-load-forward*:
assumes
wf-p1: *wf-prog Fstd-get p1* **and** *load-p1*: *Sstd.load p1 s1*
shows ∃ *p2 s2*. *compile p1* = *Some p2* ∧ *Sinca.load p2 s2* ∧ *match s1 s2*
⟨*proof*⟩

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.