

Interpreter_Optimizations

Martin Desharnais

May 26, 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

1	Generic lemmas	3
2	Environment	3
2.1	Generic lemmas	5
2.2	List-based implementation of environment	5
3	nth_opt	7
4	Generic lemmas	7
5	Non-empty list	9
6	Monadic bind	10
7	Conversion functions	11
8	pred_map	13

9 Rest	15
9.1 Function definition	15
9.2 Program	16
9.3 Stack frame	17
9.4 Dynamic state	18
10 n-ary operations	21
11 n-ary operations	21
12 Inline caching	22
12.1 Static representation	22
12.2 Semantics	23
13 n-ary operations	27
14 Unboxed caching	28
14.1 Static representation	28
14.2 Semantics	31
15 Locale imports	43
16 Normalization	44
17 Equivalence of call stacks	46
18 Simulation relation	48
19 Backward simulation	48
20 Forward simulation	49
21 Bisimulation	49
22 Strongest postcondition	50
23 Range validations	51
24 Basic block validation	51
25 Function definition validation	51
26 Program definition validation	51
27 Generic program rewriting	52
28 Lifting	53

29 Optimization	55
30 Compilation of function definition	59
31 Compilation of function environment	61
32 Compilation of program	62
32.1 Completeness of compilation	62
33 Dynamic values	64
34 Normal operations	64
35 Inlined operations	65
36 Unboxed operations	66
36.1 Typing	67
37 Generic definitions	70
38 Simulation relation	73
39 Backward simulation	73
40 Forward simulation	73
41 Bisimulation	74
41.1 Compilation of function definitions	74
41.2 Compilation of function environments	75
41.3 Compilation of programs	75
41.4 Completeness of compilation	76
theory Env	
imports <i>Main HOL-Library.Library</i>	
begin	

1 Generic lemmas

lemma *map-of-list-allI*:

assumes $\bigwedge k v. f k = \text{Some } v \implies P (k, v)$ **and**
 $\bigwedge k v. \text{map-of } kvs k = \text{Some } v \implies f k = \text{Some } v$ **and**
 distinct (map fst kvs)
 shows *list-all P kvs*
 <proof>

2 Environment

locale *env* =

```

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

begin

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

definition singleton where
  singleton  $\equiv$  add empty

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

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

lemma from-list-correct: get (from-list xs) = map-of xs
  <proof>

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-all:
  assumes  $\bigwedge k v. \text{get } e \ k = \text{Some } v \implies P \ (k, v)$ 
  shows list-all P (to-list e)
  <proof>

definition map-entry where
  map-entry e k f  $\equiv$  case get e k of None  $\Rightarrow$  e | Some v  $\Rightarrow$  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  $\neq$  y  $\implies$  get (map-entry e x f) y = get e y

```

<proof>

lemma *dom-map-entry[simp]*: $\text{dom } (\text{get } (\text{map-entry } e \ k \ f)) = \text{dom } (\text{get } e)$
<proof>

lemma *get-map-entry-conv*:
 $\text{get } (\text{map-entry } e \ x \ f) \ y = \text{map-option } (\lambda v. \text{if } x = y \text{ then } f \ v \ \text{else } \ v) (\text{get } e \ y)$
<proof>

lemma *map-option-comp-map-entry*:
assumes $\forall x \in \text{ran } (\text{get } e). f \ (g \ x) = f \ x$
shows $\text{map-option } f \circ \text{get } (\text{map-entry } e \ k \ g) = \text{map-option } f \circ \text{get } e$
<proof>

lemma *map-option-comp-get-add*:
assumes $k \in \text{dom } (\text{get } e)$ **and** $\forall x \in \text{ran } (\text{get } e). f \ v = f \ x$
shows $\text{map-option } f \circ \text{get } (\text{add } e \ k \ v) = \text{map-option } f \circ \text{get } e$
<proof>

end

end

theory *Env-list*

imports *Env HOL-Library.Library*

begin

2.1 Generic lemmas

lemma *map-of-filter*:
 $x \neq y \implies \text{map-of } (\text{filter } (\lambda z. \text{fst } z \neq y) \ zs) \ x = \text{map-of } \ zs \ x$
<proof>

2.2 List-based implementation of environment

context

begin

qualified type-synonym $(\text{'key}, \text{'val}) \ t = (\text{'key} \times \text{'val}) \ \text{list}$

qualified definition *empty* :: $(\text{'key}, \text{'val}) \ t$ **where**
 $\text{empty} \equiv []$

qualified definition *get* :: $(\text{'key}, \text{'val}) \ t \Rightarrow \text{'key} \Rightarrow \text{'val} \ \text{option}$ **where**
 $\text{get} \equiv \text{map-of}$

qualified definition *add* :: $(\text{'key}, \text{'val}) \ t \Rightarrow \text{'key} \Rightarrow \text{'val} \Rightarrow (\text{'key}, \text{'val}) \ t$ **where**
 $\text{add } e \ k \ v \equiv \text{AList.update } k \ v \ e$

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-neg*: 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*: $\text{same-length } xs \ ys \longleftrightarrow \text{length } xs = \text{length } ys$
 ⟨proof⟩

lemma *same-length-Cons*:

$\text{same-length } (x \# xs) \ ys \implies \exists y \ ys'. \ ys = y \# ys'$
 $\text{same-length } xs \ (y \# ys) \implies \exists x \ xs'. \ xs = x \# xs'$
 ⟨proof⟩

3 nth_opt

fun *nth-opt* **where**

$\text{nth-opt } (x \# -) \ 0 = \text{Some } x \mid$
 $\text{nth-opt } (- \# xs) \ (\text{Suc } n) = \text{nth-opt } xs \ n \mid$
 $\text{nth-opt } - \ - = \text{None}$

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

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

4 Generic lemmas

lemma *list-rel-imp-pred1*:

assumes
 $\text{list-all2 } R \ xs \ ys$ **and**
 $\bigwedge x \ y. (x, y) \in \text{set } (\text{zip } xs \ ys) \implies R \ x \ y \implies P \ x$
shows $\text{list-all } P \ xs$
 ⟨proof⟩

lemma *list-rel-imp-pred2*:

assumes
 $\text{list-all2 } R \ xs \ ys$ **and**
 $\bigwedge x \ y. (x, y) \in \text{set } (\text{zip } xs \ ys) \implies R \ x \ y \implies P \ y$
shows $\text{list-all } P \ ys$
 ⟨proof⟩

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

lemma *list-all-list-updateI*: $\text{list-all } P \ xs \implies P \ x \implies \text{list-all } P \ (\text{list-update } xs \ n \ x)$
 ⟨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*:
 $f (xs ! pc) = f instr \implies \text{map } f (xs[pc := instr]) = \text{map } f xs$
 $\langle \text{proof} \rangle$

lemma *list-all-eq-const-imp-replicate*:
assumes *list-all* $(\lambda x. x = y) xs$
shows $xs = \text{replicate } (\text{length } xs) y$
 $\langle \text{proof} \rangle$

lemma *list-all-eq-const-imp-replicate'*:
assumes *list-all* $((=) y) xs$
shows $xs = \text{replicate } (\text{length } xs) y$
 $\langle \text{proof} \rangle$

lemma *list-all-eq-const-replicate-lhs[intro]*:
 $\text{list-all } (\lambda x. y = x) (\text{replicate } n y)$
 $\langle \text{proof} \rangle$

lemma *list-all-eq-const-replicate-rhs[intro]*:
 $\text{list-all } (\lambda x. x = y) (\text{replicate } n y)$
 $\langle \text{proof} \rangle$

lemma *list-all-eq-const-replicate[simp]*: $\text{list-all } ((=) c) (\text{replicate } n c)$
 $\langle \text{proof} \rangle$

lemma *replicate-eq-map*:
assumes $n = \text{length } xs$ **and** $\bigwedge y. y \in \text{set } xs \implies f y = x$
shows $\text{replicate } n x = \text{map } f xs$
 $\langle \text{proof} \rangle$

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

lemma *rel-option-map-of*:
assumes *list-all2* $(\text{rel-prod } (=) R) xs ys$
shows $\text{rel-option } R (\text{map-of } xs l) (\text{map-of } ys l)$
 $\langle \text{proof} \rangle$

lemma *list-all2-rel-prod-nth*:
assumes *list-all2* $(\text{rel-prod } R1 R2) xs ys$ **and** $n < \text{length } xs$
shows $R1 (\text{fst } (xs ! n)) (\text{fst } (ys ! n)) \wedge R2 (\text{snd } (xs ! n)) (\text{snd } (ys ! n))$
 $\langle \text{proof} \rangle$

lemma *list-all2-rel-prod-fst-hd*:
assumes *list-all2* $(\text{rel-prod } R1 R2) xs ys$ **and** $xs \neq [] \vee ys \neq []$
shows $R1 (\text{fst } (\text{hd } xs)) (\text{fst } (\text{hd } ys)) \wedge R2 (\text{snd } (\text{hd } xs)) (\text{snd } (\text{hd } ys))$
 $\langle \text{proof} \rangle$

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]*: $list\text{-}all\ P\ xs \implies n < length\ xs \implies P\ (xs\ !\ n)$
<proof>

lemma *list-all* $P\ xs \implies \forall x \in set\ xs. P\ x$
<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*: $list\text{-}all\ P\ xs \implies \neg P\ (xs\ !\ n) \implies length\ xs \leq n$
<proof>

lemma *list-all-butlast-not-nthD*: $list\text{-}all\ P\ (butlast\ xs) \implies \neg P\ (xs\ !\ n) \implies length\ xs \leq Suc\ n$
<proof>

lemma *list-all-replicateI[intro]*: $P\ x \implies list\text{-}all\ P\ (replicate\ n\ x)$
<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*:
 $map\ f\ xs = replicate\ n\ x \implies n = length\ xs \implies list\text{-}all\ (\lambda y. f\ y = x)\ xs$
<proof>

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

lemma *list-all-dropI[intro]*: $list\text{-}all\ P\ xs \implies list\text{-}all\ P\ (drop\ n\ xs)$
<proof>

5 Non-empty list

type-synonym $'a\ nlist = 'a \times 'a\ list$

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  $\Rightarrow$  ('b  $\Rightarrow$  ('a, 'c) result)  $\Rightarrow$  ('a, 'c) result where
  bind (Error x) - = Error x |
  bind (Ok x) f = f x

```

```

end

```

```

adhoc-overloading

```

```

  bind Result.bind

```

```

context begin

```

```

qualified lemma bind-Ok[simp]:  $x \gg= Ok = x$ 
  (proof) lemma bind-eq-Ok-conv:  $(x \gg= f = Ok z) = (\exists y. x = Ok y \wedge f y = Ok z)$ 
  (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:
   $x \gg= f = Error z \longleftrightarrow x = Error z \vee (\exists y. x = Ok y \wedge f y = Error z)$ 
  (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  $\longleftrightarrow a \wedge b = Ok d$ 
  (if a then Error c else b) = Ok d  $\longleftrightarrow \neg a \wedge b = Ok d$ 
  (proof) lemma if-then-else-Error[simp]:
  (if a then Ok b else c) = Error d  $\longleftrightarrow \neg a \wedge c = Error d$ 
  (if a then c else Ok b) = Error d  $\longleftrightarrow a \wedge c = Error d$ 
  (proof) lemma map-eq-Ok-conv:  $map\text{-}result\ f\ g\ x = Ok\ y \longleftrightarrow (\exists x'. x = Ok\ x' \wedge y = g\ x')$ 

```

```

  <proof> lemma map-eq-Error-conv: map-result f g x = Error y  $\longleftrightarrow$  ( $\exists x'. x =$ 
  Error x'  $\wedge y = f x'$ )
  <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  $\Rightarrow$  ('err, 'ok list) result where
```

```

  those [] = Ok [] |
  those (x # xs) = do {
    y  $\leftarrow$  x;
    ys  $\leftarrow$  those xs;
    Ok (y # ys)
  }

```

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

```
<proof>
```

```
end
```

```
end
```

```
theory Option-Extra
```

```
  imports Main
```

```
begin
```

```
fun ap-option (infixl  $\diamond$  60) where
```

```

  (Some f)  $\diamond$  (Some x) = Some (f x) |
  -  $\diamond$  - = None

```

```
lemma ap-option-eq-Some-conv: f  $\diamond$  x = Some y  $\longleftrightarrow$  ( $\exists f' x'. f = Some f' \wedge x =$ 
  Some x'  $\wedge y = f' x'$ )
```

```
<proof>
```

definition *ap-map-prod* **where**

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

lemma *ap-map-prod-eq-Some-conv*:

$ap\text{-}map\text{-}prod\ f\ g\ p = Some\ p' \longleftrightarrow (\exists x\ y. p = (x, y) \wedge (\exists x'\ y'. p' = (x', y') \wedge f\ x = Some\ x' \wedge g\ y = Some\ y'))$

<proof>

fun *ap-map-list* :: ('a \Rightarrow 'b option) \Rightarrow 'a list \Rightarrow 'b list option **where**

$ap\text{-}map\text{-}list\ -\ [] = Some\ []\ |$

$ap\text{-}map\text{-}list\ f\ (x\ \#\ xs) = Some\ (\#) \diamond f\ x \diamond ap\text{-}map\text{-}list\ f\ xs$

lemma *length-ap-map-list*: $ap\text{-}map\text{-}list\ f\ xs = Some\ ys \Longrightarrow length\ ys = length\ xs$

<proof>

lemma *ap-map-list-imp-rel-option-map-of*:

assumes $ap\text{-}map\text{-}list\ f\ xs = Some\ ys$ **and**

$\bigwedge x\ y. (x, y) \in set\ (zip\ xs\ ys) \Longrightarrow f\ x = Some\ y \Longrightarrow fst\ x = fst\ y$

shows $rel\text{-}option\ (\lambda x\ y. f\ (k, x) = Some\ (k, y))\ (map\text{-}of\ xs\ k)\ (map\text{-}of\ ys\ k)$

<proof>

lemma *ap-map-list-ap-map-prod-imp-rel-option-map-of*:

assumes $ap\text{-}map\text{-}list\ (ap\text{-}map\text{-}prod\ Some\ f)\ xs = Some\ ys$

shows $rel\text{-}option\ (\lambda x\ y. f\ x = Some\ y)\ (map\text{-}of\ xs\ k)\ (map\text{-}of\ ys\ k)$

<proof>

lemma *ex-ap-map-list-eq-SomeI*:

assumes $list\text{-}all\ (\lambda x. \exists y. f\ x = Some\ y)\ xs$

shows $\exists ys. ap\text{-}map\text{-}list\ f\ xs = Some\ ys$

<proof>

lemma *ap-map-list-iff-list-all2*:

$ap\text{-}map\text{-}list\ f\ xs = Some\ ys \longleftrightarrow list\text{-}all2\ (\lambda x\ y. f\ x = Some\ y)\ xs\ ys$

<proof>

lemma *ap-map-list-map-conv*:

assumes

$ap\text{-}map\text{-}list\ f\ xs = Some\ ys$ **and**

$\bigwedge x\ y. x \in set\ xs \Longrightarrow f\ x = Some\ y \Longrightarrow y = g\ x$

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*: $k \in \text{fst } \text{'set } kvs \implies \exists v. \text{map-of } kvs \ k = \text{Some } v$
<proof>

lemma *map-of-fst-hd-neq-Nil*[simp]:

assumes $xs \neq []$

shows $\text{map-of } xs \ (\text{fst } (\text{hd } xs)) = \text{Some } (\text{snd } (\text{hd } xs))$

<proof>

definition *map-merge* **where**

$\text{map-merge } f \ m1 \ m2 \ x =$
 (case (m1 x, m2 x) of
 (None, None) \Rightarrow None
 | (None, Some z) \Rightarrow Some z
 | (Some y, None) \Rightarrow Some y
 | (Some y, Some z) \Rightarrow f y z)

lemma *option-case-cancel*[simp]: (case opt of None \Rightarrow x | Some - \Rightarrow x) = x
<proof>

lemma *map-le-map-merge-Some-const*:

$f \subseteq_m \text{map-merge } (\lambda x \ y. \text{Some } x) \ f \ g$ **and**

$g \subseteq_m \text{map-merge } (\lambda x \ y. \text{Some } y) \ f \ g$

<proof>

8 pred_map

definition *pred-map* **where**

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

lemma *pred-map-get*:

assumes $\text{pred-map } P \ m$ **and** $m \ x = \text{Some } y$

shows $P \ y$

<proof>

end

theory *AList-Extra*

imports *HOL-Library.ALlist List-util*

begin

lemma *list-all2-rel-prod-updateI*:

assumes $\text{list-all2 } (\text{rel-prod } (=) \ R) \ xs \ ys$ **and** $R \ xval \ yval$

shows $\text{list-all2 } (\text{rel-prod } (=) \ R) \ (\text{AList.update } k \ xval \ xs) \ (\text{AList.update } k \ yval \ ys)$

<proof>

lemma *length-map-entry*[simp]: $\text{length } (\text{AList.map-entry } k \ f \ al) = \text{length } al$

<proof>

lemma *map-entry-id0*[simp]: $AList.map\text{-}entry\ k\ id = id$
(proof)

lemma *map-entry-id*: $AList.map\text{-}entry\ k\ id\ xs = xs$
(proof)

lemma *map-entry-map-of-Some-conv*:
 $map\text{-}of\ xs\ k = Some\ v \implies AList.map\text{-}entry\ k\ f\ xs = AList.update\ k\ (f\ v)\ xs$
(proof)

lemma *map-entry-map-of-None-conv*:
 $map\text{-}of\ xs\ k = None \implies AList.map\text{-}entry\ k\ f\ xs = xs$
(proof)

lemma *list-all2-rel-prod-map-entry*:
assumes
 $list\text{-}all2\ (rel\text{-}prod\ (=)\ R)\ xs\ ys$ **and**
 $\bigwedge xval\ yval. map\text{-}of\ xs\ k = Some\ xval \implies map\text{-}of\ ys\ k = Some\ yval \implies R\ (f\ xval)\ (g\ yval)$
shows $list\text{-}all2\ (rel\text{-}prod\ (=)\ R)\ (AList.map\text{-}entry\ k\ f\ xs)\ (AList.map\text{-}entry\ k\ g\ ys)$
(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 $list\text{-}all\ P\ xs$ **and** $P\ (k,\ v)$
shows $list\text{-}all\ P\ (AList.update\ k\ v\ xs)$
(proof)

lemma *set-update*: $set\ (AList.update\ k\ v\ xs) \subseteq \{(k,\ v)\} \cup set\ xs$
(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* $\longleftrightarrow a \wedge b = c$

(if a then Some b else None) = None \longleftrightarrow \neg 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 \vee (\exists 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 \times 'instr list) list) (arity: nat) (return: nat) (fundef-locals:
nat)

lemma rel-fundef-arithies: rel-fundef r1 r2 gd1 gd2 \implies arity gd1 = arity gd2
<proof>

lemma rel-fundef-return: rel-fundef R1 R2 gd1 gd2 \implies return gd1 = return gd2
<proof>

lemma rel-fundef-locals: rel-fundef R1 R2 gd1 gd2 \implies fundef-locals gd1 = fundef-locals gd2
<proof>

lemma rel-fundef-body-length[simp]:
rel-fundef r1 r2 fd1 fd2 \implies length (body fd1) = length (body fd2)
<proof>

definition funtype where
funtype fd \equiv (arity fd, return fd)

lemma rel-fundef-funtype[simp]: rel-fundef R1 R2 fd1 fd2 \implies funtype fd1 = funtype fd2
<proof>

lemma rel-fundef-rel-fst-hd-bodies:
assumes rel-fundef R1 R2 fd1 fd2 and body fd1 \neq [] \vee body fd2 \neq []

shows $R1$ (fst (hd ($body$ $fd1$))) (fst (hd ($body$ $fd2$)))
 $\langle proof \rangle$

lemma *map-option-comp-conv*:

assumes

$\bigwedge x. rel-option\ R\ (f\ x)\ (g\ x)$

$\bigwedge fd1\ fd2. fd1 \in ran\ f \implies fd2 \in ran\ g \implies R\ fd1\ fd2 \implies h\ fd1 = i\ fd2$

shows $map-option\ h \circ f = map-option\ i \circ g$
 $\langle proof \rangle$

lemma *map-option-arity-comp-conv*:

assumes ($\bigwedge x. rel-option\ (rel-fundef\ R\ S)\ (f\ x)\ (g\ x)$)

shows $map-option\ arity \circ f = map-option\ arity \circ g$
 $\langle proof \rangle$

definition *wf-fundef* **where**

$wf-fundef\ fd \longleftrightarrow body\ fd \neq []$

lemma *wf-fundef-non-empty-bodyD*[*dest,intro*]: $wf-fundef\ fd \implies body\ fd \neq []$
 $\langle proof \rangle$

definition *wf-fundefs* **where**

$wf-fundefs\ F \longleftrightarrow (\forall f\ fd. F\ f = Some\ fd \longrightarrow wf-fundef\ fd)$

lemma *wf-fundefsI*:

assumes $\bigwedge f\ fd. F\ f = Some\ fd \implies wf-fundef\ fd$

shows $wf-fundefs\ F$
 $\langle proof \rangle$

lemma *wf-fundefsI'*:

assumes $\bigwedge f. pred-option\ wf-fundef\ (F\ f)$

shows $wf-fundefs\ F$
 $\langle proof \rangle$

lemma *wf-fundefs-imp-wf-fundef*:

assumes $wf-fundefs\ F$ **and** $F\ f = Some\ fd$

shows $wf-fundef\ fd$
 $\langle proof \rangle$

hide-fact *wf-fundefs-def*

9.2 Program

datatype (*'fenv, 'henv, 'fun*) *prog* =

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

definition *wf-prog* **where**

$wf-prog\ Get\ p \longleftrightarrow wf-fundefs\ (Get\ (prog-fundefs\ p))$

9.3 Stack frame

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

definition *instr-at* where

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

lemma *instr-atD*:

assumes *instr-at* fd l pc = Some instr
shows \exists instrs. map-of (body fd) l = Some instrs \wedge pc < length instrs \wedge instrs ! pc = instr
 <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 \equiv do {
 fd \leftarrow F f;
instr-at fd label pc
 }

lemma *next-instr-eq-Some-conv*:

next-instr F f l pc = Some instr \longleftrightarrow (\exists fd. F f = Some fd \wedge *instr-at* fd l pc = Some instr)
 <proof>

lemma *next-instrD*:

assumes *next-instr* F f l pc = Some instr
shows \exists fd. F f = Some fd \wedge *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 \longleftrightarrow wf-fundefs (Get (state-fundefs s))

inductive *final* **for** *F-get Return* **where**

finalI: next-instr (F-get F) f l pc = Some Return \implies

final F-get Return (State F H [Frame f l pc R Σ])

definition *allocate-frame* **where**

allocate-frame f fd args uninitialized \equiv

Frame f (fst (hd (body fd))) 0 (args @ replicate (fundef-locals fd) uninitialized)

\square

inductive *load* **for** *F-get Uninitialized* **where**

F-get F main = Some fd \implies arity fd = 0 \implies body fd \neq [] \implies

load F-get Uninitialized (Prog F H main) (State F H [allocate-frame main fd [] Uninitialized])

lemma *length-neq-imp-not-list-all2*:

assumes *length xs \neq length ys*

shows \neg *list-all2 R xs ys*

<proof>

lemma *list-all2-rel-prod-conv*:

$list\text{-}all2\ (rel\text{-}prod\ R\ S)\ xs\ ys \iff$
 $list\text{-}all2\ R\ (map\ fst\ xs)\ (map\ fst\ ys) \wedge list\text{-}all2\ S\ (map\ snd\ xs)\ (map\ snd\ ys)$
(*proof*)

definition *rewrite-fundef-body* ::

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

lemma *rewrite-fundef-body-cases*[*case-names invalid-label valid-label*]:

assumes
 $\wedge bs\ ar\ ret\ locals.\ fd = Fundef\ bs\ ar\ ret\ locals \implies map\text{-}of\ bs\ l = None \implies P$
 $\wedge bs\ ar\ ret\ locals\ instrs.\ fd = Fundef\ bs\ ar\ ret\ locals \implies map\text{-}of\ bs\ l = Some$
 $instrs \implies P$
shows P
(*proof*)

lemma *arity-rewrite-fundef-body*[*simp*]: $arity\ (rewrite\text{-}fundef\text{-}body\ fd\ l\ pc\ instr) =$
 $arity\ fd$
(*proof*)

lemma *return-rewrite-fundef-body*[*simp*]: $return\ (rewrite\text{-}fundef\text{-}body\ fd\ l\ pc\ instr)$
 $= return\ fd$
(*proof*)

lemma *funtype-rewrite-fundef-body*[*simp*]: $funtype\ (rewrite\text{-}fundef\text{-}body\ fd\ l\ pc\ instr')$
 $= funtype\ fd$
(*proof*)

lemma *length-body-rewrite-fundef-body*[*simp*]:
 $length\ (body\ (rewrite\text{-}fundef\text{-}body\ fd\ l\ pc\ instr)) = length\ (body\ fd)$
(*proof*)

lemma *prod-in-set-fst-image-conv*: $(x, y) \in set\ xys \implies x \in fst\ 'set\ xys$
(*proof*)

lemma *map-of-body-rewrite-fundef-body-conv-neq*[*simp*]:

assumes $l \neq l'$
shows $map\text{-}of\ (body\ (rewrite\text{-}fundef\text{-}body\ fd\ l\ pc\ instr))\ l' = map\text{-}of\ (body\ fd)\ l'$
(*proof*)

lemma *map-of-body-rewrite-fundef-body-conv-eq*[*simp*]:

$map\text{-}of\ (body\ (rewrite\text{-}fundef\text{-}body\ fd\ l\ pc\ instr))\ l =$
 $map\text{-}option\ (\lambda xs.\ xs[pc := instr])\ (map\text{-}of\ (body\ fd)\ l)$
(*proof*)

lemma *instr-at-rewrite-fundef-body-conv*:
 $instr\text{-}at\ (rewrite\text{-}fundef\text{-}body\ fd\ l'\ pc'\ instr')\ l\ pc =$
 $map\text{-}option\ (\lambda instr.\ if\ l = l' \wedge pc = pc'\ then\ instr'\ else\ instr)\ (instr\text{-}at\ fd\ l\ pc)$
 $\langle proof \rangle$

lemma *fundef-locals-rewrite-fundef-body[simp]*:
 $fundef\text{-}locals\ (rewrite\text{-}fundef\text{-}body\ fd\ l\ pc\ instr) = fundef\text{-}locals\ fd$
 $\langle proof \rangle$

lemma *rel-fundef-rewrite-body1*:
assumes
 $rel\text{-}fd1\text{-}fd2: rel\text{-}fundef\ (=)\ R\ fd1\ fd2$ **and**
 $instr\text{-}at\text{-}l\text{-}pc: instr\text{-}at\ fd1\ l\ pc = Some\ instr$ **and**
 $R\text{-}iff: \bigwedge x.\ R\ instr\ x \longleftrightarrow R\ instr'\ x$
shows $rel\text{-}fundef\ (=)\ R\ (rewrite\text{-}fundef\text{-}body\ fd1\ l\ pc\ instr')\ fd2$
 $\langle proof \rangle$

lemma *rel-fundef-rewrite-body*:
assumes $rel\text{-}fd1\text{-}fd2: rel\text{-}fundef\ (=)\ R\ fd1\ fd2$ **and** $R\text{-}i1\text{-}i2: R\ i1\ i2$
shows $rel\text{-}fundef\ (=)\ R\ (rewrite\text{-}fundef\text{-}body\ fd1\ l\ pc\ i1)\ (rewrite\text{-}fundef\text{-}body\ fd2\ l\ pc\ i2)$
 $\langle proof \rangle$

lemma *rewrite-fundef-body-triv*:
 $instr\text{-}at\ fd\ l\ pc = Some\ instr \implies rewrite\text{-}fundef\text{-}body\ fd\ l\ pc\ instr = fd$
 $\langle proof \rangle$

lemma *rel-fundef-rewrite-body2*:
assumes
 $rel\text{-}fd1\text{-}fd2: rel\text{-}fundef\ (=)\ R\ fd1\ fd2$ **and**
 $instr\text{-}at\text{-}l\text{-}pc: instr\text{-}at\ fd2\ l\ pc = Some\ instr$ **and**
 $R\text{-}iff: \bigwedge x.\ R\ x\ instr \longleftrightarrow R\ x\ instr'$
shows $rel\text{-}fundef\ (=)\ R\ fd1\ (rewrite\text{-}fundef\text{-}body\ fd2\ l\ pc\ instr')$
 $\langle proof \rangle$

lemma *rel-fundef-rewrite-body2'*:
assumes
 $rel\text{-}fd1\text{-}fd2: rel\text{-}fundef\ (=)\ R\ fd1\ fd2$ **and**
 $instr\text{-}at1: instr\text{-}at\ fd1\ l\ pc = Some\ instr1$ **and**
 $R\text{-}instr1\text{-}instr2: R\ instr1\ instr2'$
shows $rel\text{-}fundef\ (=)\ R\ fd1\ (rewrite\text{-}fundef\text{-}body\ fd2\ l\ pc\ instr2')$
 $\langle proof \rangle$

thm *rel-fundef-rewrite-body*

lemma *if-eq-const-conv*: $(if\ x\ then\ y\ else\ z) = w \longleftrightarrow x \wedge y = w \vee \neg x \wedge z = w$
 $\langle proof \rangle$

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

```

    <proof>

end
theory Op
  imports Main
begin

```

10 n-ary operations

```

locale nary-operations =
  fixes
     $\mathcal{O}p :: 'op \Rightarrow 'a \text{ list} \Rightarrow 'a$  and
     $\mathcal{A}rity :: 'op \Rightarrow \text{nat}$ 
  assumes
     $\mathcal{O}p\text{-}\mathcal{A}rity\text{-domain: length } xs = \mathcal{A}rity \text{ } op \Longrightarrow \exists y. \mathcal{O}p \text{ } op \text{ } xs = y$ 

```

```

end
theory OpInl
  imports Op
begin

```

11 n-ary operations

```

locale nary-operations-inl =
  nary-operations  $\mathcal{O}p$   $\mathcal{A}rity$ 
  for
     $\mathcal{O}p :: 'op \Rightarrow 'a \text{ list} \Rightarrow 'a$  and  $\mathcal{A}rity +$ 
  fixes
     $\mathcal{I}nl\mathcal{O}p :: 'opinl \Rightarrow 'a \text{ list} \Rightarrow 'a$  and
     $\mathcal{I}nl :: 'op \Rightarrow 'a \text{ list} \Rightarrow 'opinl \text{ option}$  and
     $\mathcal{I}s\mathcal{I}nl :: 'opinl \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$  and
     $\mathcal{D}e\mathcal{I}nl :: 'opinl \Rightarrow 'op$ 
  assumes
     $\mathcal{I}nl\text{-invertible: } \mathcal{I}nl \text{ } op \text{ } xs = \text{Some } opinl \Longrightarrow \mathcal{D}e\mathcal{I}nl \text{ } opinl = op$  and
     $\mathcal{I}nl\mathcal{O}p\text{-correct: length } xs = \mathcal{A}rity (\mathcal{D}e\mathcal{I}nl \text{ } opinl) \Longrightarrow \mathcal{I}nl\mathcal{O}p \text{ } opinl \text{ } xs = \mathcal{O}p (\mathcal{D}e\mathcal{I}nl$ 
     $opinl) \text{ } xs$  and
     $\mathcal{I}nl\text{-}\mathcal{I}s\mathcal{I}nl: \mathcal{I}nl \text{ } op \text{ } xs = \text{Some } opinl \Longrightarrow \mathcal{I}s\mathcal{I}nl \text{ } opinl \text{ } xs$ 

```

```

begin

```

```

lemma  $\mathcal{I}nl\text{-inj-on: inj-on } \mathcal{I}nl \{ op \mid op \text{ args. } \mathcal{I}nl \text{ } op \text{ } args \neq \text{None} \}$ 
  <proof>

```

```

abbreviation  $\mathcal{I}nl\text{-dom}$  where

```

```

     $\mathcal{I}nl\text{-}dom \equiv \{ op \mid op \text{ args. } \mathcal{I}nl \text{ op args} \neq None \}$ 

lemma bij-betw  $\mathcal{I}nl \mathcal{I}nl\text{-}dom \{ \mathcal{I}nl \text{ op} \mid op. \text{ op} \in \mathcal{I}nl\text{-}dom \}$ 
  <proof>

end

end
theory Dynamic
  imports Main
begin

locale dynval =
  fixes
    uninitialized :: 'dyn and
    is-true :: 'dyn  $\Rightarrow$  bool and
    is-false :: 'dyn  $\Rightarrow$  bool
  assumes
    not-true-and-false:  $\neg (is\text{-}true \ d \wedge is\text{-}false \ d)$ 
begin

lemma false-not-true: is-false d  $\Longrightarrow$   $\neg is\text{-}true \ d$ 
  <proof>

lemma true-not-false: is-true d  $\Longrightarrow$   $\neg is\text{-}false \ d$ 
  <proof>

lemma is-true-and-is-false-implies-False: is-true d  $\Longrightarrow is\text{-}false \ d \Longrightarrow False$ 
  <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 $\mathcal{O}p$ $\mathcal{A}rity$ $\mathcal{I}nl\mathcal{O}p$ $\mathcal{I}nl$ $\mathcal{I}s\mathcal{I}nl$ $\mathcal{D}e\mathcal{I}nl$
for

— Functions environment

F-empty **and**

F-get :: 'fenv \Rightarrow 'fun \Rightarrow ('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 \Rightarrow 'var \times 'dyn \Rightarrow 'dyn option **and**

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

— Dynamic values

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

— n-ary operations

$\mathcal{O}p$:: 'op \Rightarrow 'dyn list \Rightarrow 'dyn **and** $\mathcal{A}rity$ **and**

$\mathcal{I}nl\mathcal{O}p$ **and** $\mathcal{I}nl$ **and** $\mathcal{I}s\mathcal{I}nl$ **and** $\mathcal{D}e\mathcal{I}nl$:: 'opinl \Rightarrow 'op

begin

12.2 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* (*d* # Σ) # *st*) |

step-pop:

next-instr (*F-get* *F*) *f l pc* = *Some* *IPop* \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-get:

next-instr (*F-get* *F*) *f l pc* = *Some* (*IGet* *n*) \Longrightarrow

n < *length R* \Longrightarrow *d* = *R* ! *n* \Longrightarrow

State F H (*Frame f l pc R* Σ # *st*) \rightarrow *State F H* (*Frame f l* (*Suc pc*) *R* (*d* # Σ) # *st*) |

step-set:
 $next_instr (F.get F) f l pc = Some (ISet n) \implies$
 $n < length R \implies R' = R[n := d] \implies$
 $State F H (Frame f l pc R (d \# \Sigma) \# st) \rightarrow State F H (Frame f l (Suc pc) R' \Sigma \# st) |$

step-load:
 $next_instr (F.get F) f l pc = Some (ILoad x) \implies$
 $heap_get H (x, y) = Some d \implies$
 $State F H (Frame f l pc R (y \# \Sigma) \# st) \rightarrow State F H (Frame f l (Suc pc) R (d \# \Sigma) \# st) |$

step-store:
 $next_instr (F.get F) f l pc = Some (IStore x) \implies$
 $heap_add H (x, y) d = H' \implies$
 $State F H (Frame f l pc R (y \# d \# \Sigma) \# st) \rightarrow State F H' (Frame f l (Suc pc) R \Sigma \# st) |$

step-op:
 $next_instr (F.get F) f l pc = Some (IOp op) \implies$
 $\mathcal{A}rity op = ar \implies ar \leq length \Sigma \implies \mathcal{I}nl op (take ar \Sigma) = None \implies$
 $\mathcal{O}p op (take ar \Sigma) = x \implies$
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F H (Frame f l (Suc pc) R (x \# drop ar \Sigma) \# st) |$

step-op-inl:
 $next_instr (F.get F) f l pc = Some (IOp op) \implies$
 $\mathcal{A}rity op = ar \implies ar \leq length \Sigma \implies \mathcal{I}nl op (take ar \Sigma) = Some opinl \implies$
 $\mathcal{I}nl\mathcal{O}p opinl (take ar \Sigma) = x \implies$
 $F' = Fenv.map_entry F f (\lambda fd. rewrite_fundef_body fd l pc (IOpInl opinl)) \implies$
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F' H (Frame f l (Suc pc) R (x \# drop ar \Sigma) \# st) |$

step-op-inl-hit:
 $next_instr (F.get F) f l pc = Some (IOpInl opinl) \implies$
 $\mathcal{A}rity (\mathcal{D}e\mathcal{I}nl opinl) = ar \implies ar \leq length \Sigma \implies \mathcal{I}s\mathcal{I}nl opinl (take ar \Sigma) \implies$
 $\mathcal{I}nl\mathcal{O}p opinl (take ar \Sigma) = x \implies$
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F H (Frame f l (Suc pc) R (x \# drop ar \Sigma) \# st) |$

step-op-inl-miss:
 $next_instr (F.get F) f l pc = Some (IOpInl opinl) \implies$
 $\mathcal{A}rity (\mathcal{D}e\mathcal{I}nl opinl) = ar \implies ar \leq length \Sigma \implies \neg \mathcal{I}s\mathcal{I}nl opinl (take ar \Sigma) \implies$
 $\mathcal{I}nl\mathcal{O}p opinl (take ar \Sigma) = x \implies$
 $F' = Fenv.map_entry F f (\lambda fd. rewrite_fundef_body fd l pc (IOp (\mathcal{D}e\mathcal{I}nl opinl))) \implies$
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F' H (Frame f l (Suc pc) R (x \# drop ar \Sigma) \# st) |$

step-cjump:
 $next_instr (F.get F) f l pc = Some (ICJump l_t l_f) \implies$
 $is_true d \wedge l' = l_t \vee is_false d \wedge l' = l_f \implies$
 $State F H (Frame f l pc R (d \# \Sigma) \# st) \rightarrow State F H (Frame f l' 0 R \Sigma \# st) |$

step-call:
 $next_instr (F.get F) f l pc = Some (ICall g) \implies$
 $F.get F g = Some gd \implies arity\ gd \leq length\ \Sigma \implies$
 $frame_g = allocate_frame\ g\ gd\ (take\ (arity\ gd)\ \Sigma)\ uninitialized \implies$
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F H (frame_g \# Frame f l pc R \Sigma \# st) |$

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

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

lemma *step-right-unique: right-unique step*
 $\langle proof \rangle$

lemma *final-finished:*
assumes *final* $F.get IReturn s$
shows *finished step* s
 $\langle proof \rangle$

sublocale *semantics step final F-get IReturn*
 $\langle proof \rangle$

definition *load where*
 $load \equiv Global.load\ F.get\ uninitialized$

sublocale *language step final F-get IReturn load*
 $\langle proof \rangle$

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  $\Sigma$ ) = Frame g l pc R (if f = g then map box-operand
 $\Sigma$  else  $\Sigma$ )

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

end

end
theory OpUbx
  imports OpInl Unboxed
begin

```

13 n-ary operations

```

locale nary-operations-ubx =
  nary-operations-inl Op Arity InlOp Inl IsInl DeInl +
  unboxedval uninitialized is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2
for
  Op :: 'op  $\Rightarrow$  'dyn list  $\Rightarrow$  'dyn and Arity and
  InlOp and Inl and IsInl and DeInl :: 'opinl  $\Rightarrow$  'op and
  uninitialized :: 'dyn and is-true and is-false and
  box-ubx1 :: 'ubx1  $\Rightarrow$  'dyn and unbox-ubx1 and
  box-ubx2 :: 'ubx2  $\Rightarrow$  'dyn and unbox-ubx2 +
fixes
  UbxOp :: 'opubx  $\Rightarrow$  ('dyn, 'ubx1, 'ubx2) unboxed list  $\Rightarrow$  ('dyn, 'ubx1, 'ubx2)
unboxed option and
  Ubx :: 'opinl  $\Rightarrow$  type option list  $\Rightarrow$  'opubx option and
  Box :: 'opubx  $\Rightarrow$  'opinl and
  TypeOfOp :: 'opubx  $\Rightarrow$  type option list  $\times$  type option
assumes
  Ubx-invertible:
    Ubx opinl ts = Some opubx  $\implies$  Box opubx = opinl and
  UbxOp-correct:
    UbxOp opubx  $\Sigma$  = Some z  $\implies$  InlOp (Box opubx) (map norm-unboxed  $\Sigma$ ) =
norm-unboxed z and
  UbxOp-to-Inl:
    UbxOp opubx  $\Sigma$  = Some z  $\implies$  Inl (DeInl (Box opubx)) (map norm-unboxed
 $\Sigma$ ) = Some (Box opubx) and

  TypeOfOp-Arity:
    Arity (DeInl (Box opubx)) = length (fst (TypeOfOp opubx)) and

```

Ubx-opubx-type:

$\forall \text{opinl } ts = \text{Some } \text{opubx} \implies \text{fst } (\text{TypeDfDp } \text{opubx}) = ts$ **and**

TypeDfDp-correct:

$\text{TypeDfDp } \text{opubx} = (\text{map } \text{typeof } xs, \tau) \implies$

$\exists y. \forall \text{Dp } \text{opubx } xs = \text{Some } y \wedge \text{typeof } y = \tau$ **and**

TypeDfDp-complete:

$\forall \text{Dp } \text{opubx } xs = \text{Some } y \implies \text{TypeDfDp } \text{opubx} = (\text{map } \text{typeof } xs, \text{typeof } y)$

end

theory *Ubx*

imports *Global OpUbx Env*

VeriComp.Language

begin

14 Unboxed caching

14.1 Static representation

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

locale *ubx* =

Fenv: env F-empty F-get F-add F-to-list +

Henv: env heap-empty heap-get heap-add heap-to-list +

nary-operations-ubx

Dp Arity

InlDp Inl IsInl DeInl

uninitialized is-true is-false box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2

UbxDp Ubx Box TypeDfDp

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**
InlOp and Inl and IsInl and DeInl :: '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]:
 $generalize-instr \circ generalize-instr = generalize-instr$
 ⟨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]:
 $fst \text{ ' set } (body (generalize-fundef fd)) = fst \text{ ' set } (body fd)$
 ⟨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]:
 $map-option arity \circ F-get (Fenv.map-entry F2 f generalize-fundef) = map-option arity \circ F-get F2$
 ⟨proof⟩

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

14.2 Semantics

inductive step (infix \rightarrow 55) where

step-push:

$$\begin{aligned} & \text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } (IPush \ d) \implies \\ & \text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } \ pc) \ R \ (\text{OpDyn} \\ & \ d \ \# \ \Sigma) \ \# \ st) \mid \end{aligned}$$

step-push-ubx1:

$$\begin{aligned} & \text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } (IPushUbx1 \ n) \implies \\ & \text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } \ pc) \ R \ (\text{OpUbx1} \\ & \ n \ \# \ \Sigma) \ \# \ st) \mid \end{aligned}$$

step-push-ubx2:

$$\begin{aligned} & \text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } (IPushUbx2 \ b) \implies \\ & \text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } \ pc) \ R \ (\text{OpUbx2} \\ & \ b \ \# \ \Sigma) \ \# \ st) \mid \end{aligned}$$

step-pop:

$$\begin{aligned} & \text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } IPop \implies \\ & \text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ (x \ \# \ \Sigma) \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } \ pc) \ R \\ & \ \Sigma \ \# \ st) \mid \end{aligned}$$

step-get:

$$\begin{aligned} & \text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } (IGet \ n) \implies \\ & \ n < \text{length } R \implies \text{cast-Dyn } (R \ ! \ n) = \text{Some } d \implies \\ & \text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } \ pc) \ R \ (\text{OpDyn} \\ & \ d \ \# \ \Sigma) \ \# \ st) \mid \end{aligned}$$

step-get-ubx-hit:

$$\begin{aligned} & \text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } (IGetUbx \ \tau \ n) \implies \\ & \ n < \text{length } R \implies \text{cast-Dyn } (R \ ! \ n) = \text{Some } d \implies \text{unbox } \tau \ d = \text{Some } \text{blob} \implies \\ & \text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } \ pc) \ R \ (\text{blob} \\ & \ \# \ \Sigma) \ \# \ st) \mid \end{aligned}$$

step-get-ubx-miss:

$$\begin{aligned} & \text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } (IGetUbx \ \tau \ n) \implies \\ & \ n < \text{length } R \implies \text{cast-Dyn } (R \ ! \ n) = \text{Some } d \implies \text{unbox } \tau \ d = \text{None} \implies \\ & \ F' = \text{Fenv.map-entry } F \ f \ \text{generalize-fundef} \implies \\ & \text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ \Sigma \ \# \ st) \rightarrow \text{State } F' \ H \ (\text{box-stack } f \ (\text{Frame } f \ l \ (\text{Suc} \\ & \ \text{pc}) \ R \ (\text{OpDyn } d \ \# \ \Sigma) \ \# \ st)) \mid \end{aligned}$$

step-set:

$$\begin{aligned} & \text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } (ISet \ n) \implies \\ & \ n < \text{length } R \implies \text{cast-Dyn } \text{blob} = \text{Some } d \implies R' = R[n := \text{OpDyn } d] \implies \\ & \text{State } F \ H \ (\text{Frame } f \ l \ pc \ R \ (\text{blob} \ \# \ \Sigma) \ \# \ st) \rightarrow \text{State } F \ H \ (\text{Frame } f \ l \ (\text{Suc } \ pc) \\ & \ R' \ \Sigma \ \# \ st) \mid \end{aligned}$$

step-set-ubx:

$$\text{next-instr } (F\text{-get } F) \text{ f l pc} = \text{Some } (ISetUbx \ \tau \ n) \implies$$

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

step-load:

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

step-load-ubx-hit:

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

step-load-ubx-miss:

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

step-store:

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

step-store-ubx:

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

step-op:

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

step-op-inl:

$next-instr (F-get F) f l pc = Some (IOp op) \implies$
 $\mathcal{A}rity op = ar \implies ar \leq length \Sigma \implies$
 $ap-map-list cast-Dyn (take ar \Sigma) = Some \Sigma' \implies$
 $\mathcal{I}nl op \Sigma' = Some opinl \implies \mathcal{I}nlOp opinl \Sigma' = x \implies$
 $F' = Fenv.map-entry F f (\lambda fd. rewrite-fundef-body fd l pc (IOpInl opinl)) \implies$
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F' H (Frame f l (Suc pc) R (OpDyn$
 $x \# drop ar \Sigma) \# st) |$

step-op-inl-hit:

$next-instr (F-get F) f l pc = Some (IOpInl opinl) \implies$
 $\mathcal{A}rity (\mathcal{D}e\mathcal{I}nl opinl) = ar \implies ar \leq length \Sigma \implies$
 $ap-map-list cast-Dyn (take ar \Sigma) = Some \Sigma' \implies$
 $\mathcal{I}s\mathcal{I}nl opinl \Sigma' \implies \mathcal{I}nlOp opinl \Sigma' = x \implies$
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F H (Frame f l (Suc pc) R (OpDyn$
 $x \# drop ar \Sigma) \# st) |$

step-op-inl-miss:

$next-instr (F-get F) f l pc = Some (IOpInl opinl) \implies$
 $\mathcal{A}rity (\mathcal{D}e\mathcal{I}nl opinl) = ar \implies ar \leq length \Sigma \implies$
 $ap-map-list cast-Dyn (take ar \Sigma) = Some \Sigma' \implies$
 $\neg \mathcal{I}s\mathcal{I}nl opinl \Sigma' \implies \mathcal{I}nlOp opinl \Sigma' = x \implies$
 $F' = Fenv.map-entry F f (\lambda fd. rewrite-fundef-body fd l pc (IOp (\mathcal{D}e\mathcal{I}nl opinl)))$
 \implies
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F' H (Frame f l (Suc pc) R (OpDyn$
 $x \# drop ar \Sigma) \# st) |$

step-op-ubx:

$next-instr (F-get F) f l pc = Some (IOpUbx opubx) \implies$
 $\mathcal{D}e\mathcal{I}nl (\mathcal{B}ox opubx) = op \implies \mathcal{A}rity op = ar \implies ar \leq length \Sigma \implies$
 $\mathcal{U}brOp opubx (take ar \Sigma) = Some x \implies$
 $State F H (Frame f l pc R \Sigma \# st) \rightarrow State F H (Frame f l (Suc pc) R (x \#$
 $drop ar \Sigma) \# st) |$

step-cjump:

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

step-call:

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

step-return:

$next-instr (F-get F) g l_g pc_g = Some IReturn \implies$

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

lemma *step-deterministic*:
assumes $s1 \rightarrow s2$ **and** $s1 \rightarrow s3$
shows $s2 = s3$
 $\langle \text{proof} \rangle$

lemma *step-right-unique: right-unique step*
 $\langle \text{proof} \rangle$

lemma *final-finished*:
assumes *final F-get IReturn s*
shows *finished step s*
 $\langle \text{proof} \rangle$

sublocale *ubx-sem: semantics step final F-get IReturn*
 $\langle \text{proof} \rangle$

definition *load where*
 $\text{load} \equiv \text{Global.load } F\text{-get } (\text{OpDyn uninitialized})$

sublocale *inca-lang: language step final F-get IReturn load*
 $\langle \text{proof} \rangle$

end

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

lemma *f-g-eq-f-imp-f-comp-g-eq-f[simp]*: $(\bigwedge x. f\ (g\ x) = f\ x) \implies (f \circ g) = f$
 $\langle \text{proof} \rangle$

context *ubx begin*

inductive *sp-instr for F ret where*

Push:
 $\text{sp-instr } F\ \text{ret } (\text{IPush } d)\ \Sigma\ (\text{None } \# \Sigma) \mid$
PushUbx1:
 $\text{sp-instr } F\ \text{ret } (\text{IPushUbx1 } u)\ \Sigma\ (\text{Some } \text{Ubx1 } \# \Sigma) \mid$
PushUbx2:
 $\text{sp-instr } F\ \text{ret } (\text{IPushUbx2 } u)\ \Sigma\ (\text{Some } \text{Ubx2 } \# \Sigma) \mid$
Pop:
 $\text{sp-instr } F\ \text{ret } \text{IPop } (\tau \# \Sigma)\ \Sigma \mid$

Get:
 $sp\text{-instr } F \text{ ret } (I\text{Get } n) \Sigma (None \# \Sigma) \mid$
GetUbx:
 $sp\text{-instr } F \text{ ret } (I\text{GetUbx } \tau \ n) \Sigma (Some \ \tau \ \# \ \Sigma) \mid$
Set:
 $sp\text{-instr } F \text{ ret } (I\text{Set } n) (None \# \Sigma) \Sigma \mid$
SetUbx:
 $sp\text{-instr } F \text{ ret } (I\text{SetUbx } \tau \ n) (Some \ \tau \ \# \ \Sigma) \Sigma \mid$
Load:
 $sp\text{-instr } F \text{ ret } (I\text{Load } x) (None \# \Sigma) (None \# \Sigma) \mid$
LoadUbx:
 $sp\text{-instr } F \text{ ret } (I\text{LoadUbx } \tau \ x) (None \# \Sigma) (Some \ \tau \ \# \ \Sigma) \mid$
Store:
 $sp\text{-instr } F \text{ ret } (I\text{Store } x) (None \# \ None \# \Sigma) \Sigma \mid$
StoreUbx:
 $sp\text{-instr } F \text{ ret } (I\text{StoreUbx } \tau \ x) (None \# \ Some \ \tau \ \# \ \Sigma) \Sigma \mid$
Op:
 $\Sigma i = (\text{replicate } (\mathcal{A}rity \ op) \ None \ @ \ \Sigma) \implies$
 $sp\text{-instr } F \text{ ret } (I\text{Op } op) \Sigma i (None \# \Sigma) \mid$
OpInl:
 $\Sigma i = (\text{replicate } (\mathcal{A}rity \ (\mathcal{D}e\mathcal{I}nl \ op\text{inl})) \ None \ @ \ \Sigma) \implies$
 $sp\text{-instr } F \text{ ret } (I\text{OpInl } op\text{inl}) \Sigma i (None \# \Sigma) \mid$
OpUbx:
 $\Sigma i = \text{fst } (\mathcal{T}ype\mathcal{D}f\mathcal{D}p \ opubx) \ @ \ \Sigma \implies \text{result} = \text{snd } (\mathcal{T}ype\mathcal{D}f\mathcal{D}p \ opubx) \implies$
 $sp\text{-instr } F \text{ ret } (I\text{OpUbx } opubx) \Sigma i (\text{result} \# \Sigma) \mid$
CJump:
 $sp\text{-instr } F \text{ ret } (I\text{CJump } l_t \ l_f) [None] \ [] \mid$
Call:
 $F \ f = \text{Some } (ar, r) \implies \Sigma i = \text{replicate } ar \ None \ @ \ \Sigma \implies \Sigma o = \text{replicate } r \ None$
 $@ \ \Sigma \implies$
 $sp\text{-instr } F \text{ ret } (I\text{Call } f) \Sigma i \Sigma o \mid$
Return: $\Sigma i = \text{replicate } ret \ None \implies$
 $sp\text{-instr } F \text{ ret } I\text{Return } \Sigma i \ []$

inductive $sp\text{-instrs}$ for $F \text{ ret}$ where

Nil:
 $sp\text{-instrs } F \text{ ret } [] \Sigma \Sigma \mid$
Cons:
 $sp\text{-instr } F \text{ ret } instr \Sigma i \Sigma \implies sp\text{-instrs } F \text{ ret } instrs \Sigma \Sigma o \implies$
 $sp\text{-instrs } F \text{ ret } (instr \# instrs) \Sigma i \Sigma o$

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

lemma $sp\text{-instrs-ConsD}$:

assumes $sp\text{-instrs } F \text{ ret } (instr \# instrs) \Sigma i \Sigma o$
shows $\exists \Sigma. sp\text{-instr } F \text{ ret } instr \Sigma i \Sigma \wedge sp\text{-instrs } F \text{ ret } instrs \Sigma \Sigma o$
<proof>

lemma $sp\text{-instr-deterministic}$:

assumes
sp-instr F *ret instr* $\Sigma i \Sigma o$ **and**
sp-instr F *ret instr* $\Sigma i \Sigma o'$
shows $\Sigma o = \Sigma o'$
 \langle *proof* \rangle

lemma *sp-instr-right-unique*: *right-unique* $(\lambda(\text{instr}, \Sigma i) \Sigma o. \text{sp-instr } F \text{ ret instr } \Sigma i \Sigma o)$
 \langle *proof* \rangle

lemma *sp-instrs-deterministic*:
assumes
sp-instrs F *ret instr* $\Sigma i \Sigma o$ **and**
sp-instrs F *ret instr* $\Sigma i \Sigma o'$
shows $\Sigma o = \Sigma o'$
 \langle *proof* \rangle

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

lemma *fun-call-in-range-generalize-instr*[*simp*]:
fun-call-in-range F (*generalize-instr* *instr*) $\longleftrightarrow \text{fun-call-in-range } F \text{ instr}$
 \langle *proof* \rangle

lemma *sp-instr-complete*:
assumes *fun-call-in-range* F *instr*
shows $\exists \Sigma i \Sigma o. \text{sp-instr } F \text{ ret instr } \Sigma i \Sigma o$
 \langle *proof* \rangle

lemma *sp-instr-Op2I*:
assumes $\mathfrak{A}(\text{arity}) \text{ op} = 2$
shows *sp-instr* F *ret* (*I**Op* *op*) (*None* # *None* # Σ) (*None* # Σ)
 \langle *proof* \rangle

lemma
assumes
F-def: $F = \text{Map.empty}$ **and**
arity-op: $\mathfrak{A}(\text{arity}) \text{ op} = 2$
shows *sp-instrs* F 1 [*I**Push* x , *I**Push* y , *I**Op* *op*, *I**Return*] [] []
 \langle *proof* \rangle

lemma *sp-instrs-NilD*[*dest*]: *sp-instrs* F *ret* [] $\Sigma i \Sigma o \implies \Sigma i = \Sigma o$
 \langle *proof* \rangle

lemma *sp-instrs-list-update*:
assumes
sp-instrs F *ret instrs* $\Sigma i \Sigma o$ **and**
sp-instr F *ret* (*instrs*! n) = *sp-instr* F *ret instr*

shows $sp\text{-instrs } F \text{ ret } (instrs[n := instr]) \Sigma i \Sigma o$
 $\langle proof \rangle$

lemma $sp\text{-instrs-appendD}$:

assumes $sp\text{-instrs } F \text{ ret } (instrs1 @ instrs2) \Sigma i \Sigma o$
shows $\exists \Sigma. sp\text{-instrs } F \text{ ret } instrs1 \Sigma i \Sigma \wedge sp\text{-instrs } F \text{ ret } instrs2 \Sigma \Sigma o$
 $\langle proof \rangle$

lemma $sp\text{-instrs-appendD}'$:

assumes $sp\text{-instrs } F \text{ ret } (instrs1 @ instrs2) \Sigma i \Sigma o$ **and** $sp\text{-instrs } F \text{ ret } instrs1 \Sigma i \Sigma$
shows $sp\text{-instrs } F \text{ ret } instrs2 \Sigma \Sigma o$
 $\langle proof \rangle$

lemma $sp\text{-instrs-appendI[intro]}$:

assumes $sp\text{-instrs } F \text{ ret } instrs1 \Sigma i \Sigma$ **and** $sp\text{-instrs } F \text{ ret } instrs2 \Sigma \Sigma o$
shows $sp\text{-instrs } F \text{ ret } (instrs1 @ instrs2) \Sigma i \Sigma o$
 $\langle proof \rangle$

lemma $sp\text{-instrs-singleton-conv[simp]}$:

$sp\text{-instrs } F \text{ ret } [instr] \Sigma i \Sigma o \longleftrightarrow sp\text{-instr } F \text{ ret } instr \Sigma i \Sigma o$
 $\langle proof \rangle$

lemma $sp\text{-instrs-singletonI}$:

assumes $sp\text{-instr } F \text{ ret } instr \Sigma i \Sigma o$
shows $sp\text{-instrs } F \text{ ret } [instr] \Sigma i \Sigma o$
 $\langle proof \rangle$

fun $local\text{-var-in-range}$ **where**

$local\text{-var-in-range } n (IGet k) \longleftrightarrow k < n \mid$
 $local\text{-var-in-range } n (IGetUbx \tau k) \longleftrightarrow k < n \mid$
 $local\text{-var-in-range } n (ISet k) \longleftrightarrow k < n \mid$
 $local\text{-var-in-range } n (ISetUbx \tau k) \longleftrightarrow k < n \mid$
 $local\text{-var-in-range } - - \longleftrightarrow True$

lemma $local\text{-var-in-range-generalize-instr[simp]}$:

$local\text{-var-in-range } n (generalize\text{-instr } instr) \longleftrightarrow local\text{-var-in-range } n instr$
 $\langle proof \rangle$

lemma $local\text{-var-in-range-comp-generalize-instr[simp]}$:

$local\text{-var-in-range } n \circ generalize\text{-instr} = local\text{-var-in-range } n$
 $\langle proof \rangle$

fun $jump\text{-in-range}$ **where**

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

inductive $wf\text{-basic-block}$ **for** $F L \text{ ret } n$ **where**

$instrs \neq [] \implies$

list-all (*local-var-in-range* *n*) *instrs* \implies
list-all (*fun-call-in-range* *F*) *instrs* \implies
list-all (*jump-in-range* *L*) *instrs* \implies
list-all ($\lambda i. \neg \text{is-jump } i \wedge \neg \text{is-return } i$) (*butlast* *instrs*) \implies
sp-instrs *F* *ret* *instrs* [] [] \implies
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* \longleftrightarrow
body *fd* \neq [] \wedge
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* \longleftrightarrow ($\forall f. \text{pred-option } (\text{wf-fundef } (\text{map-option funtype } \circ F)) (F f)$)

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

lemma *wf-fundefs-getD*:

shows *wf-fundefs* *F* \implies *F* *f* = *Some* *fd* \implies *wf-fundef* (*map-option funtype* \circ *F*)
fd
<proof>

definition *wf-prog* **where**

wf-prog *p* \longleftrightarrow *wf-fundefs* (*F*.*get* (*prog-fundefs* *p*))

definition *wf-state* **where**

wf-state *s* \longleftrightarrow *wf-fundefs* (*F*.*get* (*state-fundefs* *s*))

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

lemma *sp-instr-generalize0*:

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

lemma *sp-instrs-generalize0*:

assumes *sp-instrs* *F* *ret* *instrs* Σi Σo **and**
 $\Sigma i' = \text{map } (\lambda \cdot \text{None}) \Sigma i$ **and** $\Sigma o' = \text{map } (\lambda \cdot \text{None}) \Sigma o$
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*) \longleftrightarrow *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: $\bigwedge \text{fd}. \text{fd} \in \text{ran } (F\text{-get } F) \implies \text{funtype } (f \text{ fd}) = \text{funtype } \text{fd}$ **and**
same-arity: $\bigwedge \text{fd}. F\text{-get } F \ x = \text{Some } \text{fd} \implies \text{arity } (f \text{ fd}) = \text{arity } \text{fd}$ **and**
same-return: $\bigwedge \text{fd}. F\text{-get } F \ x = \text{Some } \text{fd} \implies \text{return } (f \text{ fd}) = \text{return } \text{fd}$ **and**
same-body-length: $\bigwedge \text{fd}. F\text{-get } F \ x = \text{Some } \text{fd} \implies \text{length } (\text{body } (f \text{ fd})) = \text{length}$
(*body* *fd*) **and**
same-locals: $\bigwedge \text{fd}. F\text{-get } F \ x = \text{Some } \text{fd} \implies \text{fundef-locals } (f \text{ fd}) = \text{fundef-locals}$
fd **and**
same-labels: $\bigwedge \text{fd}. F\text{-get } F \ x = \text{Some } \text{fd} \implies \text{fst ' set } (\text{body } (f \text{ fd})) = \text{fst ' set}$
(*body* *fd*) **and**
list-all-wf-basic-block-f: $\bigwedge \text{fd}.$
 $F\text{-get } F \ x = \text{Some } \text{fd} \implies$
list-all (*wf-basic-block* (*map-option funtype* \circ *F*-*get* *F*) (*fst ' set* (*body* *fd*))
(*return* *fd*)
(*arity* *fd* + *fundef-locals* *fd*)) (*body* *fd*) \implies
list-all (*wf-basic-block* (*map-option funtype* \circ *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* *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' \longleftrightarrow local-var-in-range n instr

and

fun-call-in-range-iff: fun-call-in-range F instr' \longleftrightarrow fun-call-in-range F instr

and

jump-in-range-iff: jump-in-range L instr' \longleftrightarrow jump-in-range L instr and

is-jump-iff: is-jump instr' \longleftrightarrow is-jump instr and

is-return-iff: is-return instr' \longleftrightarrow is-return instr

shows *list-all (wf-basic-block F L ret n) (body (rewrite-fundef-body fd l pc instr'))*

<proof>

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: \bigwedge ret.

sp-instr (map-option funtype \circ F-get F) ret instr' =

sp-instr (map-option funtype \circ F-get F) ret instr and

local-var-in-range-iff: \bigwedge n. local-var-in-range n instr' \longleftrightarrow local-var-in-range n

instr and

fun-call-in-range-iff:

fun-call-in-range (map-option funtype \circ F-get F) instr' \longleftrightarrow

fun-call-in-range (map-option funtype \circ F-get F) instr and

jump-in-range-iff: \bigwedge L. jump-in-range L instr' \longleftrightarrow jump-in-range L instr and

is-jump-iff: is-jump instr' \longleftrightarrow is-jump instr and

is-return-iff: is-return instr' \longleftrightarrow is-return instr

shows *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 = \mathfrak{DcInl} 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  $\implies$  typeof u = None
  <proof>

lemma typeof-bind-OpDyn[simp]: typeof  $\circ$  OpDyn = ( $\lambda$ -. None)
  <proof>

lemma is-dyn-operand-eq-typeof: is-dyn-operand = ( $\lambda$ x. typeof x = None)
  <proof>

lemma is-dyn-operand-eq-typeof-Dyn[simp]: is-dyn-operand x  $\longleftrightarrow$  typeof x = None
  <proof>

lemma typeof-unboxed-eq-const:
  fixes x
  shows
    typeof x = None  $\longleftrightarrow$  ( $\exists$  d. x = OpDyn d)
    typeof x = Some Ubx1  $\longleftrightarrow$  ( $\exists$  n. x = OpUbx1 n)
    typeof x = Some Ubx2  $\longleftrightarrow$  ( $\exists$  b. x = OpUbx2 b)
  <proof>

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

lemma cast-inversions:
  cast-Dyn x = Some d  $\implies$  x = OpDyn d
  cast-Ubx1 x = Some n  $\implies$  x = OpUbx1 n
  cast-Ubx2 x = Some b  $\implies$  x = OpUbx2 b
  <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  $\tau$  d = Some blob  $\implies$  typeof blob = Some  $\tau$ 
  <proof>

lemma cast-and-box-imp-typeof[simp]: cast-and-box  $\tau$  blob = Some d  $\implies$  typeof
blob = Some  $\tau$ 
  <proof>

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

```

= d
⟨proof⟩

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

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

lemma *typeof-and-norm-unboxed-imp-cast-and-box*:
assumes *typeof* $x' = \text{Some } \tau$ norm-unboxed $x' = x$
shows *cast-and-box* τ $x' = \text{Some } x$
⟨proof⟩

lemma *norm-unboxed-bind-OpDyn*[simp]: norm-unboxed \circ 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* \circ 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* \circ 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 \implies take n xs = butlast xs*
 <proof>

lemma *append-take-singleton-conv:Suc n = length xs \implies 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

$\mathcal{D}p$ Arity InlOp Inl IsInl DeInl +

Subx: ubx

Fubx-empty Fubx-get Fubx-add Fubx-to-list
heap-empty heap-get heap-add heap-to-list
uninitialized is-true is-false

box-ubx1 unbox-ubx1 box-ubx2 unbox-ubx2

$\mathcal{D}p$ Arity InlOp Inl IsInl DeInl UbrOp Ubr Box TypeOfOp

for

— Functions environments

Finca-empty and

Finca-get :: 'fenv-inca \implies 'fun \implies ('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 \implies 'fun \implies ('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 \implies 'var \times 'dyn \implies 'dyn option and heap-add and heap-to-list and

— Dynamic values

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

— Unboxed values

box-ubx1 and unbox-ubx1 and

box-ubx2 and unbox-ubx2 and

— n-ary operations

$\mathcal{D}p$ and Arity and InlOp and Inl and IsInl and DeInl and UbrOp and Ubr 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 $\exists \text{instr2}. \text{next-instr } F2 \text{ f l pc} = \text{Some instr2} \wedge \text{norm-eq instr1 instr2}$
<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 $\exists \text{instr1}. \text{next-instr } F1 \text{ f l pc} = \text{Some instr1} \wedge \text{norm-eq instr1 instr2}$
<proof>

lemma *rel-fundefs-empty: rel-fundefs ($\lambda\cdot. \text{None}$) ($\lambda\cdot. \text{None}$)*
<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 $\exists z. g x = \text{Some } z \wedge \text{rel-fundef } (=) \text{norm-eq } y z$
<proof>

lemma *rel-fundefs-Some2:*

assumes *rel-fundefs f g* **and** *g x = Some y*
shows $\exists z. f x = \text{Some } z \wedge \text{rel-fundef } (=) \text{norm-eq } z y$
<proof>

lemma *rel-fundefs-rel-option:*

assumes *rel-fundefs f g* **and** $\bigwedge x y. \text{rel-fundef } (=) \text{norm-eq } x y \implies h x y$
shows *rel-option h (f z) (g z)*
<proof>

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 (Finc-get F1) (Fubx-get F2)*
shows *rel-fundefs (Finc-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* ($\lambda fd. \text{rewrite-fundef-body } fd \ l \ pc \ instr1'$)))
(*Fubx-get* (*Subx.Fenv.map-entry* *F2 f* ($\lambda fd. \text{rewrite-fundef-body } fd \ l \ pc \ instr2'$)))
(**is** *rel-fundefs* (*Finca-get* *?F1'*) (*Fubx-get* *?F2'*))
⟨*proof*⟩

17 Equivalence of call stacks

definition *norm-stack* :: (*'dyn*, *'ubx1*, *'ubx2*) *unboxed list* \Rightarrow *'dyn list* **where**
norm-stack $\Sigma \equiv List.map \ Subx.norm-unboxed \ \Sigma$

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* \equiv *next-instr* *F f l pc* = *Some* (*ICall* *g*) \wedge
 $(\exists gd. F \ g = \text{Some } gd \wedge \text{arity } gd \leq \text{length } \Sigma \wedge \text{list-all is-dyn-operand } (\text{take}$
 $(\text{arity } gd) \ \Sigma))$

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* \vee
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* \neg *is-fun-call instr* \neg
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* \neq *g* *l* \neq *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* \implies
 $\Sigma 1 = \text{map } \text{Subx.norm-unboxed } \Sigma 2 \implies$
 $R 1 = \text{map } \text{Subx.norm-unboxed } R 2 \implies$
 $\text{list-all is-dyn-operand } R 2 \implies$
 $F f = \text{Some } fd 2 \implies \text{map-of (body } fd 2) l = \text{Some instrs} \implies$
 $\text{Subx.sp-instrs (map-option funtype } \circ F) (\text{return } fd 2) (\text{take } pc \text{ instrs}) [] (\text{map}$
 $\text{typeof } \Sigma 2) \implies$
 $\text{pred-option (is-valid-fun-call } F f l pc \Sigma 2) opt \implies$
rel-stacktraces *F opt* (*Frame* *f l pc R1* $\Sigma 1$ $\#$ *st1*) (*Frame* *f l pc R2* $\Sigma 2$ $\#$ *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**
 $\bigwedge ret. Subx.sp_instr (map_option\ funtype \circ Fubx_get\ F2) ret\ instr =$
 $Subx.sp_instr (map_option\ funtype \circ Fubx_get\ F2) ret\ instr'$ **and**
 $\neg is_fun_call\ instr \neg is_fun_call\ instr'$
shows $rel_stacktraces$
 $(Fubx_get (Subx.Fenv.map_entry\ F2\ f (\lambda fd. rewrite_fundef_body\ fd\ l\ pc\ instr')))$
 $opt\ st1\ st2$
 $\langle proof \rangle$

18 Simulation relation

inductive $match$ (**infix** ~ 55) **where**
 $matchI: Subx.wf_state (State\ F2\ H\ st2) \implies$
 $rel_fundefs (Finca_get\ F1) (Fubx_get\ F2) \implies$
 $rel_stacktraces (Fubx_get\ F2) None\ st1\ st2 \implies$
 $match (State\ F1\ H\ st1) (State\ F2\ H\ st2)$

lemmas $matchI[consumes\ 0, case_names\ wf_state\ rel_fundefs\ rel_stacktraces] =$
 $match.intros(1)$

19 Backward simulation

lemma $map_eq_append_map_drop$:
 $map\ f\ xs = ys @ map\ f (drop\ n\ xs) \longleftrightarrow map\ f (take\ n\ xs) = ys$
 $\langle proof \rangle$

lemma $ap_map_list_cast_Dyn_to_map_norm$:
assumes $ap_map_list\ cast_Dyn\ xs = Some\ ys$
shows $ys = map\ Subx.norm_unboxed\ xs$
 $\langle proof \rangle$

lemma $ap_map_list_cast_Dyn_to_all_Dyn$:
assumes $ap_map_list\ cast_Dyn\ xs = Some\ ys$
shows $list_all (\lambda x. typeof\ x = None)\ xs$
 $\langle proof \rangle$

lemma $ap_map_list_cast_Dyn_map_typeof_replicate_conv$:
assumes $ap_map_list\ cast_Dyn\ xs = Some\ ys$ **and** $n = length\ xs$
shows $map\ typeof\ xs = replicate\ n\ None$
 $\langle proof \rangle$

lemma $cast_Dyn_eq_Some_conv_norm_unboxed[simp]$: $cast_Dyn\ i = Some\ i' \implies$
 $Subx.norm_unboxed\ i = i'$
 $\langle proof \rangle$

lemma $cast_Dyn_eq_Some_conv_typeof[simp]$: $cast_Dyn\ i = Some\ i' \implies typeof\ i =$
 $None$
 $\langle proof \rangle$

lemma *backward-lockstep-simulation*:
assumes *match s1 s2 and Subx.step s2 s2'*
shows $\exists s1'. \text{Sinca.step } s1 \ s1' \wedge \text{match } s1' \ s2'$
<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
 $\lambda-. \text{False } \lambda-. \text{match}$
<proof>

20 Forward simulation

lemma *ap-map-list-cast-Dyn-eq-norm-stack*:
assumes *list-all ($\lambda x. x = \text{None}$) (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 $\exists s2'. \text{Subx.step } s2 \ s2' \wedge \text{match } s1' \ s2'$
<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
 $\lambda-. \text{False } \lambda-. \text{match}$
<proof>

21 Bisimulation

sublocale *inca-ubx-bisimulation*:
bisimulation Sinca.step Subx.step final Finca-get Inca.IReturn final Fubx-get Ubx.IReturn
 $\lambda-. \text{False } \lambda-. \text{match}$
<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)  $\Sigma$  (Suc  $\Sigma$ ) |
Pop:
  sp-instr F ret IPop (Suc  $\Sigma$ )  $\Sigma$  |
Get:
  sp-instr F ret (IGet n)  $\Sigma$  (Suc  $\Sigma$ ) |
Set:
  sp-instr F ret (ISet n) (Suc  $\Sigma$ )  $\Sigma$  |
Load:
  sp-instr F ret (ILoad x) (Suc  $\Sigma$ ) (Suc  $\Sigma$ ) |
Store:
  sp-instr F ret (IStore x) (Suc (Suc  $\Sigma$ ))  $\Sigma$  |
Op:
   $\Sigma i = \mathfrak{Arity} \text{ op} + \Sigma \implies$ 
  sp-instr F ret (IOp op)  $\Sigma i$  (Suc  $\Sigma$ ) |
OpInl:
   $\Sigma i = \mathfrak{Arity} (\mathfrak{OpInl} \text{ opinl}) + \Sigma \implies$ 
  sp-instr F ret (IOpInl opinl)  $\Sigma i$  (Suc  $\Sigma$ ) |
CJump:
  sp-instr F ret (ICJump  $l_t$   $l_f$ ) 1 0 |
Call:
   $F f = \text{Some} (ar, r) \implies \Sigma i = ar + \Sigma \implies \Sigma o = r + \Sigma \implies$ 
  sp-instr F ret (ICall f)  $\Sigma i$   $\Sigma o$  |
Return:  $\Sigma i = \text{ret} \implies$ 
  sp-instr F ret IReturn  $\Sigma i$  0

```

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

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

```

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

```

23 Range validations

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

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

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

24 Basic block validation

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

25 Function definition validation

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

26 Program definition validation

definition *wf-prog* **where**
 wf-prog p \longleftrightarrow
 (*let* $F = F\text{-get}$ (*prog-fundefs* p) *in*
 pred-map (*wf-fundef* (*map-option* *funtype* $\circ F$)) F)

end

end

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

begin

27 Generic program rewriting

primrec *monadic-fold-map* **where**

$$\begin{aligned} \text{monadic-fold-map } f \text{ acc } [] &= \text{Some } (\text{acc}, []) \mid \\ \text{monadic-fold-map } f \text{ acc } (x \# xs) &= \text{do } \{ \\ & \quad (\text{acc}', x') \leftarrow f \text{ acc } x; \\ & \quad (\text{acc}'', xs') \leftarrow \text{monadic-fold-map } f \text{ acc}' xs; \\ & \quad \text{Some } (\text{acc}'', x' \# xs') \\ & \} \end{aligned}$$

lemma *monadic-fold-map-length*:

$$\text{monadic-fold-map } f \text{ acc } xs = \text{Some } (\text{acc}', xs') \implies \text{length } xs = \text{length } xs'$$

<proof>

lemma *monadic-fold-map-ConsD[dest]*:

assumes *monadic-fold-map* $f \ a \ (x \# xs) = \text{Some } (c, ys)$
shows $\exists y \ ys' \ b. \ ys = y \# ys' \wedge f \ a \ x = \text{Some } (b, y) \wedge \text{monadic-fold-map } f \ b \ xs = \text{Some } (c, ys')$
 $= \text{Some } (c, ys')$
<proof>

lemma *monadic-fold-map-eq-Some-conv*:

$$\begin{aligned} \text{monadic-fold-map } f \ a \ (x \# xs) = \text{Some } (c, ys) &\longleftrightarrow \\ (\exists y \ ys' \ b. \ f \ a \ x = \text{Some } (b, y) \wedge \text{monadic-fold-map } f \ b \ xs &= \text{Some } (c, ys') \wedge ys \\ = y \# ys') & \end{aligned}$$

<proof>

lemma *monadic-fold-map-eq-Some-conv'*:

$$\begin{aligned} \text{monadic-fold-map } f \ a \ (x \# xs) = \text{Some } p &\longleftrightarrow \\ (\exists y \ ys' \ b. \ f \ a \ x = \text{Some } (b, y) \wedge \text{monadic-fold-map } f \ b \ xs &= \text{Some } (\text{fst } p, ys') \\ \wedge \text{snd } p = y \# ys') & \end{aligned}$$

<proof>

lemma *monadic-fold-map-list-all2*:

assumes *monadic-fold-map* $f \ \text{acc} \ xs = \text{Some } (\text{acc}', ys)$ **and**
 $\bigwedge \text{acc} \ \text{acc}' \ x \ y. \ f \ \text{acc} \ x = \text{Some } (\text{acc}', y) \implies P \ x \ y$
shows *list-all2* $P \ xs \ ys$
<proof>

lemma *monadic-fold-map-list-all*:

assumes *monadic-fold-map* $f \ \text{acc} \ xs = \text{Some } (\text{acc}', ys)$ **and**
 $\bigwedge \text{acc} \ \text{acc}' \ x \ y. \ f \ \text{acc} \ x = \text{Some } (\text{acc}', y) \implies P \ y$
shows *list-all* $P \ ys$
<proof>

```

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

```

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

28 Lifting

```

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

```

definition lift-instrs **where**

```

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

```

lemma lift-instrs-length:

```

assumes lift-instrs F L ret N  $\Sigma$  i xs = Some ( $\Sigma$ o, ys)
shows length xs = length ys
<proof>

```

lemma lift-instrs-not-Nil: lift-instrs F L ret N Σ i xs = Some (Σ o, ys) \implies xs \neq []

$\longleftrightarrow ys \neq []$
<proof>

lemma *lift-instrs-NilD[dest]*:
assumes *lift-instrs F L ret N $\Sigma i [] = Some (\Sigma o, ys)$*
shows $\Sigma o = \Sigma i \wedge ys = []$
<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 $\Sigma i = Some (y, \Sigma o)$*
shows *Inca.is-jump x \longleftrightarrow Ubx.is-jump y*
<proof>

lemma *lift-instr-is-return*:
assumes *lift-instr F L ret N x $\Sigma i = Some (y, \Sigma o)$*
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 $\Sigma i xs = Some (\Sigma o, ys)$*
shows
list-all ($\lambda i. \neg$ Inca.is-jump i \wedge \neg Inca.is-return i) xs \longleftrightarrow
list-all ($\lambda i. \neg$ Ubx.is-jump i \wedge \neg Ubx.is-return i) ys
<proof>

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

lemma *lift-instr-sp*:
assumes *lift-instr F L ret N x $\Sigma i = Some (y, \Sigma o)$*
shows *Subx.sp-instr F ret y $\Sigma i \Sigma o$*
<proof>

lemma *lift-instrs-sp*:
assumes *lift-instrs F L ret N $\Sigma i xs = Some (\Sigma o, ys)$*
shows *Subx.sp-instrs F ret ys $\Sigma i \Sigma o$*
<proof>

lemma *lift-instr-fun-call-in-range*:
assumes *lift-instr F L ret N x $\Sigma i = Some (y, \Sigma o)$*
shows *Subx.fun-call-in-range F y*
<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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

lemma *lift-instr-norm*:
lift-instr F L ret N instr1 $\Sigma 1$ = Some (instr2, $\Sigma 2$) \implies norm-eq instr1 instr2
 \langle *proof* \rangle

lemma *lift-instrs-all-norm*:
assumes *lift-instrs F L ret N $\Sigma 1$ instrs1 = Some ($\Sigma 2$, instrs2)*
shows *list-all2 norm-eq instrs1 instrs2*
 \langle *proof* \rangle

29 Optimization

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

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

lemma *None-orelse[simp]*: *None orelse y = y*
 \langle *proof* \rangle

lemma *orelse-None[simp]*: *x orelse None = x*
 \langle *proof* \rangle

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

lemma *orelse-eq-Some-conv*:
x orelse y = Some z \longleftrightarrow *(x = Some z \vee x = None \wedge y = Some z)*
 ⟨proof⟩

lemma *orelse-eq-SomeE*:
assumes
x orelse y = Some z **and**
x = Some z \implies P **and**
x = None \implies y = Some z \implies 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* \longleftrightarrow *ys = xs @ zs*
 ⟨proof⟩

fun *optim-instr where*
optim-instr - - - (IPush d) Σ =
Some Pair \diamond (Some IPushUbx1 \diamond (unbox-ubx1 d)) \diamond Some (Some Ubx1 # Σ)
orelse
Some Pair \diamond (Some IPushUbx2 \diamond (unbox-ubx2 d)) \diamond 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 ($\lambda\tau$. (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 ($\lambda\tau$. (ILoadUbx τ x, Some τ # Σ)) (load-oracle pc) orelse
Some (ILoad x, None # Σ) |

$optim-instr \ - \ - \ (IloadUbx \ \tau \ x) \ (None \ \# \ \Sigma) = Some \ (IloadUbx \ \tau \ x, \ Some \ \tau \ \# \ \Sigma) \ |$
 $optim-instr \ - \ - \ (IStore \ x) \ (None \ \# \ None \ \# \ \Sigma) = Some \ (IStore \ x, \ \Sigma) \ |$
 $optim-instr \ - \ - \ (IStore \ x) \ (None \ \# \ Some \ \tau \ \# \ \Sigma) = Some \ (IStoreUbx \ \tau \ x, \ \Sigma) \ |$
 $optim-instr \ - \ - \ (IStoreUbx \ - \ x) \ (None \ \# \ None \ \# \ \Sigma) = Some \ (IStore \ x, \ \Sigma) \ |$
 $optim-instr \ - \ - \ (IStoreUbx \ - \ x) \ (None \ \# \ Some \ \tau \ \# \ \Sigma) = Some \ (IStoreUbx \ \tau \ x, \ \Sigma) \ |$
 $optim-instr \ - \ - \ (IOp \ op) \ \Sigma =$
 $\quad map-option \ (\lambda \Sigma o. \ (IOp \ op, \ None \ \# \ \Sigma o)) \ (drop-prefix \ (replicate \ (\mathfrak{A}rity \ op) \ None) \ \Sigma) \ |$
 $optim-instr \ - \ - \ (IOpInl \ opinl) \ \Sigma = ($
 $\quad let \ ar = \mathfrak{A}rity \ (\mathfrak{D}e\mathfrak{I}nl \ opinl) \ in$
 $\quad if \ ar \leq length \ \Sigma \ then$
 $\quad \quad case \ \mathfrak{U}bx \ opinl \ (take \ ar \ \Sigma) \ of$
 $\quad \quad \quad None \Rightarrow map-option \ (\lambda \Sigma o. \ (IOpInl \ opinl, \ None \ \# \ \Sigma o)) \ (drop-prefix \ (replicate \ ar \ None) \ \Sigma) \ |$
 $\quad \quad \quad Some \ opubx \Rightarrow map-option \ (\lambda \Sigma o. \ (IOpUbx \ opubx, \ snd \ (\mathfrak{T}hpe\mathfrak{D}f\mathfrak{D}p \ opubx) \ \# \ \Sigma o))$
 $\quad \quad \quad \quad (drop-prefix \ (fst \ (\mathfrak{T}hpe\mathfrak{D}f\mathfrak{D}p \ opubx)) \ \Sigma)$
 $\quad \quad \quad else$
 $\quad \quad \quad \quad None$
 $\quad) \ |$
 $optim-instr \ - \ - \ (IOpUbx \ opubx) \ \Sigma =$
 $\quad (let \ p = \mathfrak{T}hpe\mathfrak{D}f\mathfrak{D}p \ opubx \ in$
 $\quad \quad map-option \ (\lambda \Sigma o. \ (IOpUbx \ opubx, \ snd \ p \ \# \ \Sigma o)) \ (drop-prefix \ (fst \ p) \ \Sigma)) \ |$
 $optim-instr \ - \ - \ (ICJump \ l_t \ l_f) \ [None] = Some \ (ICJump \ l_t \ l_f, \ []) \ |$
 $optim-instr \ F \ - \ (ICall \ f) \ \Sigma = do \ {$
 $\quad (ar, \ ret) \leftarrow F \ f;$
 $\quad \Sigma o \leftarrow drop-prefix \ (replicate \ ar \ None) \ \Sigma;$
 $\quad Some \ (ICall \ f, \ replicate \ ret \ None \ @ \ \Sigma o)$
 $\quad } \ |$
 $optim-instr \ - \ ret \ - \ IReturn \ \Sigma = (if \ \Sigma = replicate \ ret \ None \ then \ Some \ (IReturn,$
 $\quad []) \ else \ None) \ |$
 $optim-instr \ - \ - \ - \ - \ = \ None$

definition *optim-instrs where*

$optim-instrs \ F \ ret \equiv \lambda pc \ \Sigma i \ instrs.$
 $\quad map-option \ (\lambda(-, \ \Sigma o), \ instrs'). \ (\Sigma o, \ instrs')$
 $\quad \quad (monadic-fold-map \ (\lambda(pc, \ \Sigma) \ instr.$
 $\quad \quad \quad map-option \ (\lambda(instr', \ \Sigma o). \ ((Suc \ pc, \ \Sigma o), \ instr')) \ (optim-instr \ F \ ret \ pc \ instr$
 $\quad \quad \quad \Sigma))$
 $\quad \quad (pc, \ \Sigma i) \ instrs)$

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

$optim-instrs \ F \ ret \ pc \ \Sigma i \ (instr \ \# \ instrs) = Some \ (\Sigma o, \ ys) \longleftrightarrow (\exists y \ ys' \ \Sigma.$
 $\quad ys = y \ \# \ ys' \wedge$
 $\quad \quad optim-instr \ F \ ret \ pc \ instr \ \Sigma i = Some \ (y, \ \Sigma) \wedge$
 $\quad \quad optim-instrs \ F \ ret \ (Suc \ pc) \ \Sigma \ instrs = Some \ (\Sigma o, \ ys'))$
 $\langle proof \rangle$

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) \implies xs*

$\neq [] \longleftrightarrow$ ys $\neq []$

<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' \wedge*

optim-instr F ret pc x Σi = Some (y, Σ) \wedge

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 ($\lambda i 1$ 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 \longleftrightarrow is-jump y*

<proof>

lemma *optim-instr-is-return*:

assumes *optim-instr F ret pc x Σi = Some (y, Σo)*

shows *is-return x \longleftrightarrow 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 (λi . \neg is-jump i \wedge \neg is-return i) (butlast xs) \longleftrightarrow

list-all (λi . \neg is-jump i \wedge \neg is-return i) (butlast ys)

<proof>

lemma *optim-instr-jump-in-range*:

assumes *optim-instr* F *ret pc* x $\Sigma i = \text{Some } (y, \Sigma o)$

shows $\text{Subx.jump-in-range } L \ x \longleftrightarrow \text{Subx.jump-in-range } L \ y$

<proof>

lemma *optim-instrs-all-jump-in-range*:

assumes *optim-instrs* F *ret pc* Σi $xs = \text{Some } (\Sigma o, ys)$

shows $\text{list-all } (\text{Subx.jump-in-range } L) \ xs \longleftrightarrow \text{list-all } (\text{Subx.jump-in-range } L) \ ys$

<proof>

lemma *optim-instr-fun-call-in-range*:

assumes *optim-instr* F *ret pc* x $\Sigma i = \text{Some } (y, \Sigma o)$

shows $\text{Subx.fun-call-in-range } F \ x \longleftrightarrow \text{Subx.fun-call-in-range } F \ y$

<proof>

lemma *optim-instrs-all-fun-call-in-range*:

assumes *optim-instrs* F *ret pc* Σi $xs = \text{Some } (\Sigma o, ys)$

shows $\text{list-all } (\text{Subx.fun-call-in-range } F) \ xs \longleftrightarrow \text{list-all } (\text{Subx.fun-call-in-range } F) \ ys$

<proof>

lemma *optim-instr-local-var-in-range*:

assumes *optim-instr* F *ret pc* x $\Sigma i = \text{Some } (y, \Sigma o)$

shows $\text{Subx.local-var-in-range } N \ x \longleftrightarrow \text{Subx.local-var-in-range } N \ y$

<proof>

lemma *optim-instrs-all-local-var-in-range*:

assumes *optim-instrs* F *ret pc* Σi $xs = \text{Some } (\Sigma o, ys)$

shows $\text{list-all } (\text{Subx.local-var-in-range } N) \ xs \longleftrightarrow \text{list-all } (\text{Subx.local-var-in-range } N) \ ys$

<proof>

lemma *optim-instr-sp*:

assumes *optim-instr* F *ret pc* x $\Sigma i = \text{Some } (y, \Sigma o)$

shows $\text{Subx.sp-instr } F \ \text{ret } y \ \Sigma i \ \Sigma o$

<proof>

lemma *optim-instrs-sp*:

assumes *optim-instrs* F *ret pc* Σi $xs = \text{Some } (\Sigma o, ys)$

shows $\text{Subx.sp-instrs } F \ \text{ret } ys \ \Sigma i \ \Sigma o$

<proof>

30 Compilation of function definition

definition *compile-basic-block where*

compile-basic-block $F \ L \ \text{ret } N \equiv$

ap-map-prod $\text{Some } (\lambda i1. \text{do } \{$

- \leftarrow if $i1 \neq []$ then $\text{Some } ()$ else None ;

- \leftarrow if $\text{list-all } (\lambda i. \neg \text{Inca.is-jump } i \wedge \neg \text{Inca.is-return } i)$ (butlast $i1$) then

```

Some () else None;
  ( $\Sigma o$ ,  $i2$ )  $\leftarrow$  lift-instrs  $F L$  ret  $N$  ( $[]$  :: type option list)  $i1$ ;
  if  $\Sigma o = []$  then
    case optim-instrs  $F$  ret  $0$  ( $[]$  :: type option list)  $i2$  of
      Some ( $\Sigma o'$ ,  $i2'$ )  $\Rightarrow$  Some (if  $\Sigma o' = []$  then  $i2'$  else  $i2$ ) |
      None  $\Rightarrow$  Some  $i2$ 
    else
      None
  })

```

lemma *compile-basic-block-rel-prod-all-norm-eq*:
assumes *compile-basic-block* $F L$ ret N $bblock1 = \text{Some } bblock2$
shows *rel-prod* (=) (*list-all2 norm-eq*) $bblock1$ $bblock2$
<proof>

lemma *list-all-iff-butlast-last*:
assumes $xs \neq []$
shows *list-all* P $xs \longleftrightarrow$ *list-all* P (*butlast* xs) \wedge P (*last* xs)
<proof>

lemma *compile-basic-block-wf*:
assumes *compile-basic-block* $F L$ ret N $x = \text{Some } y$
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 {
 - \leftarrow if $bblocks1 = []$ then None else Some ();
 $bblocks2 \leftarrow$ 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 $fd1 = \text{Some } fd2 \implies$ *arity* $fd1 =$
arity $fd2$
<proof>

lemma *compile-fundef-returns*: *compile-fundef* F $fd1 = \text{Some } fd2 \implies$ *return* $fd1$
 $=$ *return* $fd2$
<proof>

lemma *compile-fundef-locals*:
compile-fundef F $fd1 = \text{Some } fd2 \implies$ *fundef-locals* $fd1 =$ *fundef-locals* $fd2$
<proof>

lemma *if-then-None-else-Some-eq[simp]*:
(*if* a then None else Some b) = Some $c \longleftrightarrow \neg a \wedge b = c$
(*if* a then None else Some b) = None $\longleftrightarrow a$
<proof>

lemma
assumes *compile-fundef* F $fd1 = \text{Some } fd2$
shows
rel-compile-fundef: *rel-fundef* $(=)$ *norm-eq* $fd1$ $fd2$ (**is** *?REL*) **and**
wf-compile-fundef: *Subx.wf-fundef* F $fd2$ (**is** *?WF*)
 $\langle \text{proof} \rangle$

end

end

locale *inca-ubx-compiler* =
inca-to-ubx-simulation *Finca-empty* *Finca-get*
for
Finca-empty **and**
Finca-get $:: - \Rightarrow 'fun \Rightarrow - \text{option} +$
fixes
load-oracle $:: 'fun \Rightarrow \text{nat} \Rightarrow \text{type option}$
begin

31 Compilation of function environment

definition *compile-env-entry* **where**
compile-env-entry $F \equiv \lambda p. \text{ap-map-prod } \text{Some} (\text{compile-fundef} (\text{load-oracle} (\text{fst } p)) F) p$

lemma *rel-compile-env-entry*:
assumes *compile-env-entry* F $(f, fd1) = \text{Some} (f, fd2)$
shows *rel-fundef* $(=)$ *norm-eq* $fd1$ $fd2$
 $\langle \text{proof} \rangle$

definition *compile-env* **where**
compile-env $e \equiv \text{do } \{$
 $\text{let } \text{fundefs1} = \text{Finca-to-list } e;$
 $\text{fundefs2} \leftarrow \text{ap-map-list} (\text{compile-env-entry} (\text{map-option } \text{funtype} \circ \text{Finca-get } e))$
 $\text{fundefs1};$
 $\text{Some} (\text{Subx.Fenv.from-list } \text{fundefs2})$
 $\}$

lemma *rel-ap-map-list-ap-map-list-compile-env-entries*:
assumes *ap-map-list* $(\text{compile-env-entry } F) xs = \text{Some } ys$
shows *rel-fundefs* $(\text{Finca-get} (\text{Sinca.Fenv.from-list } xs)) (\text{Fubx-get} (\text{Subx.Fenv.from-list } ys))$
 $\langle \text{proof} \rangle$

lemma *rel-fundefs-compile-env*:
assumes *compile-env* $F1 = \text{Some } F2$
shows *rel-fundefs* $(\text{Finca-get } F1) (\text{Fubx-get } F2)$

<proof>

32 Compilation of program

fun *compile* **where**

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

lemma *ap-map-list-cong*:

assumes $\bigwedge x. x \in \text{set } ys \implies f x = g x$ **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 $\exists s1. \text{Sinca.load } p1 \ s1 \wedge \text{match } s1 \ s2$

<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*

$\lambda - . \text{False}$ $\lambda - . \text{match compile}$

<proof>

32.1 Completeness of compilation

lemma *lift-instr-None-preservation*:

assumes *lift-instr* F L ret N *instr* $\Sigma = \text{Some } (instr', \Sigma')$ **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 $\exists instr' \Sigma'. \text{lift-instr } F \ L \ \text{ret } N \ \text{instr } \Sigma = \text{Some } (instr', \Sigma') \wedge \text{length } \Sigma' = k$

<proof>

lemma *lift-instrs-complete*:
fixes $\Sigma :: \text{type option list}$
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 $\exists \Sigma' \text{ instrs}'.$ *lift-instrs* $F L \text{ ret } N \Sigma \text{ instrs} = \text{Some } (\Sigma', \text{instrs}') \wedge \text{length}$
 $\Sigma' = k$
<proof>

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

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

lemma *bind-eq-map-option[simp]*: $x \gg= (\lambda y. \text{Some } (f y)) = \text{map-option } f x$
<proof>

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

lemma *compile-env-entry-complete*:
assumes *wf-fd1*: *Sinca.wf-fundef* $F \text{ fd1}$
shows $\exists \text{ fd2}.$ *compile-env-entry* $F (f, \text{fd1}) = \text{Some } \text{ fd2}$
<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* $F1 = \text{Some } F2$
<proof>

theorem *compile-complete*:
assumes *wf-p1*: *Sinca.wf-prog* $p1$
shows $\exists p2.$ *compile* $p1 = \text{Some } p2$
<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  $\equiv$  (d = DBool True)

```

```

definition is-false where
  is-false d  $\equiv$  (d = DBool False)

```

```

interpretation dynval-dynamic: dynval DNil is-true is-false
<proof>

```

```

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

```

```

fun unbox-bool :: dynamic  $\Rightarrow$  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  $\Rightarrow$  nat where
  ar OpNeg = 1 |
  ar OpAdd = 2 |
  ar OpMul = 2

```

```

fun eval-Neg :: dynamic list  $\Rightarrow$  dynamic where
  eval-Neg [DBool b] = DBool ( $\neg$ b) |
  eval-Neg [-] = DNil

```

```

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

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

lemma *eval-AddNum-correct*:
 $\text{length } xs = 2 \implies \text{eval-AddNum } xs = \text{eval-Add } xs$
<proof>

lemma *eval-MulNum-correct*:
 $\text{length } xs = 2 \implies \text{eval-MulNum } xs = \text{eval-Mul } xs$
<proof>

lemma *eval-inl-correct*:
 $\text{length } xs = \text{ar } (\text{deinl } \text{opinl}) \implies \text{eval-inl } \text{opinl } xs = \text{eval } (\text{deinl } \text{opinl}) xs$
<proof>

lemma *inl-isinl*:
 $\text{inl } \text{op } xs = \text{Some } \text{opinl} \implies \text{isinl } \text{opinl } xs$
<proof>

interpretation *op-OpInl*: *nary-operations-inl eval ar eval-inl inl isinl deinl*
<proof>

36 Unboxed operations

datatype *opubx* =
OpAddNumUbx

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

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

lemma *ubx-invertible*: $\text{ubx } \text{opinl } xs = \text{Some } \text{opubx} \implies \text{deubx } \text{opubx} = \text{opinl}$
<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 \implies
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* \Rightarrow *type option list* \times *type option* **where**
typeof-opubx OpAddNumUbx = ([Some Ubx1, Some Ubx1], Some Ubx1)

lemma *ubx-imp-typeof-opubx*:
ubx opinl ts = Some opubx \implies *fst (typeof-opubx opubx) = ts*
 ⟨*proof*⟩

lemma *typeof-opubx-correct*:
typeof-opubx opubx = (map typeof xs, codomain) \implies
 $\exists y. \text{eval-ubx opubx xs} = \text{Some } y \wedge \text{typeof } y = \text{codomain}$
 ⟨*proof*⟩

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

lemma *typeof-opubx-ar*: *length (fst (typeof-opubx opubx)) = ar (deinl (deubx op-
 ubx))*
 ⟨*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 |
  IJump '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  $\Rightarrow$  'fun  $\Rightarrow$  ('label, ('dyn, 'var, 'fun, 'label, 'op) instr) fundef option
and
  F-add and F-to-list and

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

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

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

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

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

  step-get:

```

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

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

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

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

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

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

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

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

```

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  $\equiv$  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
   $\Delta$  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
   $\Delta$  Arity InlOp Inl IsInl DeInl
for
  — Functions environments
  Fstd-empty and
  Fstd-get :: 'fenv-std  $\Rightarrow$  'fun  $\Rightarrow$  ('label, ('dyn, 'var, 'fun, 'label, 'op) Std.instr)
fundef option and

```

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

Finca-empty **and**

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

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

— Memory heap

heap-empty **and**

heap-get :: 'henv \Rightarrow 'var \times 'dyn \Rightarrow 'dyn option **and**

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

— Dynamic values

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

— n-ary operations

\mathfrak{Op} :: 'op \Rightarrow 'dyn list \Rightarrow 'dyn **and** \mathfrak{Arity} **and**

\mathfrak{InlOp} **and** \mathfrak{Inl} **and** \mathfrak{IsInl} **and** \mathfrak{DeInl} :: 'opinl \Rightarrow 'op

begin

fun *norm-instr* **where**

norm-instr (*Inca.IPush* *d*) = *Std.IPush* *d* |

norm-instr *Inca.IPop* = *Std.IPop* |

norm-instr (*Inca.IGet* *n*) = *Std.IGet* *n* |

norm-instr (*Inca.ISet* *n*) = *Std.ISet* *n* |

norm-instr (*Inca.ILoad* *x*) = *Std.ILoad* *x* |

norm-instr (*Inca.IStore* *x*) = *Std.IStore* *x* |

norm-instr (*Inca.IOp* *op*) = *Std.IOp* *op* |

norm-instr (*Inca.IOpInl* *opinl*) = *Std.IOp* (\mathfrak{DeInl} *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* \equiv *norm-instr* *y* = *x*

definition *rel-fundefs* **where**

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

lemma *rel-fundefsI*:

assumes $\bigwedge x$. *rel-option* (*rel-fundef* (=) *norm-eq*) (*F1* *x*) (*F2* *x*)

shows *rel-fundefs* *F1* *F2*

<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 $\exists \text{instr2. next-instr F2 f l pc = Some instr2} \wedge \text{norm-eq instr1 instr2}$

<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 $\exists \text{instr1. next-instr F1 f l pc = Some instr1} \wedge \text{norm-eq instr1 instr2}$

<proof>

lemma *rel-fundefs-empty*: *rel-fundefs ($\lambda\cdot$. None) ($\lambda\cdot$. 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 $\exists z. g x = Some z \wedge \text{rel-fundef } (=) \text{ norm-eq } y z$

<proof>

lemma *rel-fundefs-Some2*:

assumes *rel-fundefs f g* **and** *g x = Some y*

shows $\exists z. f x = Some z \wedge \text{rel-fundef } (=) \text{ norm-eq } z y$

<proof>

lemma *rel-fundefs-rel-option*:

assumes *rel-fundefs f g* **and** $\bigwedge x y. \text{rel-fundef } (=) \text{ norm-eq } x y \implies h x y$

shows *rel-option h (f z) (g z)*

<proof>

lemma *rel-fundefs-rewriteI2*:

assumes

rel-F1-F2: *rel-fundefs (Fstd-get F1) (Finca-get F2)* **and**

next-instr1: *next-instr (Fstd-get F1) f l pc = Some instr1*

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

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 $\exists s1'. Sstd.step s1 s1' \wedge match s1' s2'$
 ⟨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 $\exists s2'. Sinca.step s2 s2' \wedge s1' \sim s2'$
 ⟨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 = *Std.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* $fd1 = \text{Some } fd2$
shows *rel-fundef* $(=)$ *norm-eq* $fd1\ fd2$
 $\langle \text{proof} \rangle$

lemma *rel-fundef-imp-fundef-ok-iff*:
assumes *rel-fundef* $(=)$ *norm-eq* $fd1\ fd2$
shows *wf-fundef* $fd1 \longleftrightarrow wf-fundef\ fd2$
 $\langle \text{proof} \rangle$

lemma *rel-fundefs-imp-wf-fundefs-iff*:
assumes *rel-f-g*: *rel-fundefs* $f\ g$
shows *wf-fundefs* $f \longleftrightarrow wf-fundefs\ g$
 $\langle \text{proof} \rangle$

lemma *compile-fundef-wf*:
assumes *compile-fundef* $fd = \text{Some } fd'$
shows *wf-fundef* fd'
 $\langle \text{proof} \rangle$

41.2 Compilation of function environments

definition *compile-env* **where**

compile-env $e \equiv$
 $\text{Some } \text{Sinca.Fenv.from-list} \diamond \text{ap-map-list } (\text{ap-map-prod } \text{Some } \text{compile-fundef})$
 $(\text{Fstd-to-list } e)$

lemma *compile-env-imp-rel-option*:
assumes *compile-env* $F1 = \text{Some } F2$
shows *rel-option* $(\lambda fd1\ fd2. \text{compile-fundef } fd1 = \text{Some } fd2)$ $(\text{Fstd-get } F1\ f)$
 $(\text{Finca-get } F2\ f)$
 $\langle \text{proof} \rangle$

lemma *Finca-get-compile*:
assumes *compile-F1*: *compile-env* $F1 = \text{Some } F2$
shows *Finca-get* $F2\ f = \text{Fstd-get } F1\ f \ggg \text{compile-fundef}$
 $\langle \text{proof} \rangle$

lemma *compile-env-rel-fundefs*:
assumes *compile-env* $F1 = \text{Some } F2$
shows *rel-fundefs* $(\text{Fstd-get } F1)$ $(\text{Finca-get } F2)$
 $\langle \text{proof} \rangle$

lemma *compile-env-imp-wf-fundefs2*:
assumes *compile-env* $F1 = \text{Some } F2$
shows *wf-fundefs* $(\text{Finca-get } F2)$
 $\langle \text{proof} \rangle$

41.3 Compilation of programs

fun *compile* **where**

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

theorem *compile-load*:

assumes

$compile-p1: compile p1 = Some p2$ **and**

$load: Sinca.load p2 s2$

shows $\exists s1. Sstd.load p1 s1 \wedge match s1 s2$

<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**

$order = \lambda-. False$ **and** $match = \lambda-. match$ **and** $compile = compile$

<proof>

41.4 Completeness of compilation

lemma *compile-fundef-complete*:

assumes $wf-fundef fd1$

shows $\exists fd2. compile-fundef fd1 = Some fd2$

<proof>

lemma *compile-env-complete*:

assumes $wf-F1: wf-fundefs (Fstd-get F1)$

shows $\exists F2. compile-env F1 = Some F2$

<proof>

theorem *compile-complete*:

assumes $wf-p1: wf-prog Fstd-get p1$

shows $\exists p2. compile p1 = Some p2$

<proof>

theorem *compile-load-forward*:

assumes

$wf-p1: wf-prog Fstd-get p1$ **and** $load-p1: Sstd.load p1 s1$

shows $\exists p2 s2. compile p1 = Some p2 \wedge Sinca.load p2 s2 \wedge match s1 s2$

<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 SIG-PLAN International Conference on Certified Programs and Proofs*, CPP 2021. Association for Computing Machinery, 2021.