Interpreter_Optimizations

Martin Desharnais

February 23, 2021

Abstract

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

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

This formalization was described in [1].

Contents

theory *Env*
imports *Main*
begin

1 Environment

locale *env* =
fixes
empty :: *'env* **and**
get :: *'env* ⇒ *'key* ⇒ *'val option* **and**
add :: *'env* ⇒ *'key* ⇒ *'val* ⇒ *'env* **and**
to-list :: *'env* ⇒ (*'key* × *'val*) *list*
assumes
get-empty: *get empty x* = *None* **and**
get-add-eq: *get* (*add e x v*) *x* = *Some v* **and**
get-add-neq: *x* ≠ *y* ⟹ *get* (*add e x v*) *y* = *get e y* **and**
to-list-correct: *map-of* (*to-list e*) = *get e*

begin

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

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

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

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

lemma *from-list-correct*: *map-of xs k* = *get* (*from-list xs*) *k*
⟨*proof*⟩

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

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

end

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

1.1 List-based implementation of environment

context
begin

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

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

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

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

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

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

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

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

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

end

global-interpretation *env-list*:
env Env-list.empty Env-list.get Env-list.add Env-list.to-list
defines
singleton = *env-list.singleton* **and**
add-list = *env-list.add-list* **and**
from-list = *env-list.from-list*

⟨*proof*⟩

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

end
theory *List-util*
  imports *Main*
begin

inductive *same-length* :: *'a list* ⇒ *'b list* ⇒ *bool* **where**
  same-length-Nil: *same-length* [] [] |
  same-length-Cons: *same-length xs ys* ⟹ *same-length* (*x* # *xs*) (*y* # *ys*)

code-pred *same-length* ⟨*proof*⟩

lemma *same-length-iff-eq-lengths*: *same-length xs ys* ⟷ *length xs* = *length ys*
⟨*proof*⟩

lemma *same-length-Cons*:
  same-length (*x* # *xs*) *ys* ⟹ ∃ *y ys'*. *ys* = *y* # *ys'*
  same-length xs (*y* # *ys*) ⟹ ∃ *x xs'*. *xs* = *x* # *xs'*
⟨*proof*⟩

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

code-pred *for-all2* ⟨*proof*⟩

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

lemma *for-all2-refl*: (∀ *x*. *r x x*) ⟹ *for-all2 r xs xs*
  ⟨*proof*⟩

lemma *for-all2-same-length*: *for-all2 r xs ys* ⟹ *same-length xs ys*
  ⟨*proof*⟩

lemma *for-all2-ConsD*: *for-all2 r* (*x* # *xs*) (*y* # *ys*) ⟹ *r x y* ∧ *for-all2 r xs ys*
  ⟨*proof*⟩

2 nth_opt

fun *nth-opt* **where**
  nth-opt (*x* # -) *0* = *Some x* |
  nth-opt (- # *xs*) (*Suc n*) = *nth-opt xs n* |

nth-opt - - = None

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

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

3 Generic lemmas

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

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

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

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

lemma *replicate-eq-map*:
assumes *list-all g* (*take n xs*) **and** *n* $\leq$ *length xs* **and** $\forall$ *y*. *g y* $\longrightarrow$ *f y = x*
shows *replicate n x = map f* (*take n xs*)
⟨*proof*⟩

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

context begin

⟨*ML*⟩

fun *pure* **where**
pure x = Some x

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

end

lemma *identity*: *pure id <*> x = x*
⟨*proof*⟩

lemma *homomorphism*: *pure f <*> pure x = pure (f x)*
⟨*proof*⟩

lemma *interchange*: *f <*> pure x = pure (λg. g x) <*> f*
⟨*proof*⟩

lemma *composition*: *pure (∘) <*> u <*> v <*> w = u <*> (v <*> w)*
⟨*proof*⟩

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

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

4 Monadic bind

context begin

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

end

adhoc-overloading
bind Result.bind

context begin

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

qualified lemma *bind-eq-Error-conv*:
x ⤜ f = Error z ⟷ *x = Error z* ∨ (∃ *y*. *x = Ok y* ∧ *f y = Error z*)

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

lemma *if-then-else-Ok*[*simp*]:
(*if a then b else Error c*) = *Ok d* ⟷ *a* ∧ *b* = *Ok d*
(*if a then Error c else b*) = *Ok d* ⟷ ¬ *a* ∧ *b* = *Ok d*
⟨*proof*⟩ **lemma** *if-then-else-Error*[*simp*]:
(*if a then Ok b else c*) = *Error d* ⟷ ¬ *a* ∧ *c* = *Error d*
(*if a then c else Ok b*) = *Error d* ⟷ *a* ∧ *c* = *Error d*
⟨*proof*⟩ **lemma** *map-eq-Ok-conv*: *map-result f g x* = *Ok y* ⟷ (∃ *x'*. *x* = *Ok x'* ∧ *y* = *g x'*)
⟨*proof*⟩ **lemma** *map-eq-Error-conv*: *map-result f g x* = *Error y* ⟷ (∃ *x'*. *x* = *Error x'* ∧ *y* = *f x'*)
⟨*proof*⟩ **lemmas** *map-eq-OkD*[*dest*!] = *iffD1*[*OF map-eq-Ok-conv*]
qualified lemmas *map-eq-ErrorD*[*dest*!] = *iffD1*[*OF map-eq-Error-conv*]

end

5 Conversion functions

context begin

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

qualified lemma *of-option-injective*[*simp*]: (*of-option e x* = *of-option e y*) = (*x* = *y*)
⟨*proof*⟩ **lemma** *of-option-eq-Ok*[*simp*]: (*of-option x y* = *Ok z*) = (*y* = *Some z*)
⟨*proof*⟩ **fun** *to-option* **where**
to-option (*Error* -) = *None* |
to-option (*Ok x*) = *Some x*

qualified lemma *to-option-Some*[*simp*]: (*to-option r* = *Some x*) = (*r* = *Ok x*)
⟨*proof*⟩ **fun** *those* :: ('*err*, '*ok*) *result list* ⇒ ('*err*, '*ok list*) *result* **where**
those [] = *Ok* [] |
those (*x* # *xs*) = *do* {
y ← *x*;
ys ← *those xs*;
Ok (*y* # *ys*)
}

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

end

end
theory *Global*
  imports *Main Result*
begin

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

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

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

fun *traverse* :: ('*a* ⇒ '*b option*) ⇒ '*a list* ⇒ '*b list option* **where**
  traverse f [] = *Some* [] |
  traverse f (*x* # *xs*) = *do* {
    x' ← *f x*;
    xs' ← *traverse f xs*;
    Some (*x'* # *xs'*)
  }

lemma *traverse-length*: *traverse f xs* = *Some ys* ⟹ *length ys* = *length xs*
⟨*proof*⟩

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

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

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

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

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

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

definition *rewrite* :: '*instr list* ⇒ *nat* ⇒ '*instr* ⇒ '*instr list* **where**

rewrite p pc i = list-update p pc i

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

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

lemma *body-rewrite-fundef-body*[*simp*]: *body (rewrite-fundef-body fd n x) = rewrite (body fd) n x*
⟨*proof*⟩

lemma *arity-rewrite-fundef-body*[*simp*]: *arity (rewrite-fundef-body fd n x) = arity fd*
⟨*proof*⟩

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

end
theory *Op*
imports *Main*
begin

6 n-ary operations

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

end
theory *OpInl*
imports *Op*
begin

7 n-ary operations

locale *nary-operations-inl* =
nary-operations $\mathfrak{Op}$ $\mathfrak{Arity}$
for

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

begin

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

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

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

end

end
theory *Dynamic*
imports *Main*
begin

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

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

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

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

end

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

8 Inline caching

8.1 Static representation

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

8.2 Dynamic representation

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

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

8.3 Semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *step-deterministic*:
s1 → s2 ⟹ s1 → s3 ⟹ s2 = s3
⟨*proof*⟩

lemma *final-finished*: *final s ⟹ finished step s*
⟨*proof*⟩

sublocale *inca-sem*: *semantics step final*
⟨*proof*⟩

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

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

end

end
theory *Unboxed*

imports *Global Dynamic*
begin

datatype *type* = *Ubx1* | *Ubx2*

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

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

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

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

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

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

fun *unbox* :: *type* ⇒ '*dyn* ⇒ ('*dyn*, '*ubx1*, '*ubx2*) *unboxed option* **where**
unbox Ubx1 = *map-option OpUbx1* ∘ *unbox-ubx1* |
unbox Ubx2 = *map-option OpUbx2* ∘ *unbox-ubx2*

fun *cast-and-box* **where**
cast-and-box Ubx1 = *map-option box-ubx1* ∘ *cast-Ubx1* |
cast-and-box Ubx2 = *map-option box-ubx2* ∘ *cast-Ubx2*

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

norm-unboxed (*OpUbx2 x*) = *box-ubx2 x*

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

fun *box-frame* **where**
box-frame f (*Frame g pc* Σ) = *Frame g pc* (*if f* = *g then map box-operand* Σ *else* Σ)

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

end

end
theory *OpUbx*
imports *OpInl Unboxed*
begin

9 n-ary operations

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

𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭-𝔄𝔯𝔦𝔱𝔶:

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

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

begin

end

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

10 Unboxed caching

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

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

    — Memory heap
```

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

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

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

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

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

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

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

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

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

lemma *arity-generalize-fundef*[*simp*]: *arity* (*generalize-fundef fd2*) = *arity fd2*
⟨*proof*⟩

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

10.1 Semantics

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

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

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

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

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

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

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

pc) (*OpDyn d* # Σ) # *st*)) |

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

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

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

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

step-op-inl-hit:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IOpInl opinl* ⟹
𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 *opinl*) = *ar* ⟹ *ar* ≤ *length* Σ ⟹
traverse cast-Dyn (*take ar* Σ) = *Some* Σ′ ⟹
𝔍𝔰𝔍𝔫𝔩 *opinl* Σ′ ⟹ 𝔍𝔫𝔩𝔒𝔭 *opinl* Σ′ = *x* ⟹
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*OpDyn x* # *drop ar* Σ) # *st*) |

step-op-inl-miss:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IOpInl opinl* ⟹
𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 *opinl*) = *ar* ⟹ *ar* ≤ *length* Σ ⟹
traverse cast-Dyn (*take ar* Σ) = *Some* Σ′ ⟹
¬ 𝔍𝔰𝔍𝔫𝔩 *opinl* Σ′ ⟹ 𝔍𝔫𝔩𝔒𝔭 *opinl* Σ′ = *x* ⟹

F-add F f (*rewrite-fundef-body fd pc* (*IOp* (𝔇𝔢𝔍𝔫𝔩 *opinl*))) = *F'* ⟹
State F H (*Frame f pc* Σ # *st*) → *State F' H* (*Frame f* (*Suc pc*) (*OpDyn x* # *drop ar* Σ) # *st*) |

step-op-ubx:
F-get F f = *Some fd* ⟹ *pc* < *length* (*body fd*) ⟹ *body fd* ! *pc* = *IOpUbx opubx* ⟹
𝔇𝔢𝔍𝔫𝔩 (𝔅𝔬𝔵 *opubx*) = *op* ⟹ 𝔄𝔯𝔦𝔱𝔶 *op* = *ar* ⟹ *ar* ≤ *length* Σ ⟹
𝔘𝔟𝔵𝔒𝔭 *opubx* (*take ar* Σ) = *Some x* ⟹
State F H (*Frame f pc* Σ # *st*) → *State F H* (*Frame f* (*Suc pc*) (*x* # *drop ar* Σ) # *st*) |

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

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

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

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

theorem *step-deterministic*: *s1* → *s2* ⟹ *s1* → *s3* ⟹ *s2* = *s3*
⟨*proof*⟩

lemma *final-finished*: *final s* ⟹ *finished step s*
⟨*proof*⟩

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

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

load (*Prog F H main*) (*State F H* [*Frame main 0* []])

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

end

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

11 Type of operations

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

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

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

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

12 Strongest postcondition of instructions

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

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

lemma *sp-instr-no-jump*: *sp-instr F instr* Σ = *Ok type* ⟹ ¬ *is-jump instr*
⟨*proof*⟩

lemma *map-constant*[*simp*]: ∀ *x* ∈ *set xs*. *x* = *y* ⟹ *map* (λ-. *y*) *xs* = *xs*
⟨*proof*⟩

lemma *sp-generalize-instr*:
assumes *sp-instr F x* Σ = *Ok* Σ′
shows *sp-instr F* (*generalize-instr x*) (*map* (λ-. *None*) Σ) = *Ok* (*map* (λ-. *None*) Σ′)
⟨*proof*⟩

lemma *map-typeof-box*: *map typeof* (*map box-operand* Σ) = *replicate* (*length* Σ) *None*
⟨*proof*⟩

13 Strongest postcondition of function definitions

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

lemma *sp-no-jump*: *sp F xs* Σ = *Ok type* ⟹ ∀ *x* ∈ *set xs*. ¬ *is-jump x*

⟨*proof*⟩

lemma *sp-append*: *sp F* (*xs* @ *ys*) Σ = *sp F xs* Σ ⤜ *sp F ys*
⟨*proof*⟩

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

lemma *sp-generalize*:
assumes *sp F xs* Σ = *Ok* Σ′
shows *sp F* (*map generalize-instr xs*) (*map* (λ-. *None*) Σ) = *Ok* (*map* (λ-. *None*) Σ′)
⟨*proof*⟩

lemma *sp-generalize2*:
assumes *sp F xs* (*replicate n None*) = *Ok* Σ′
shows *sp F* (*map generalize-instr xs*) (*replicate n None*) = *Ok* (*map* (λ-. *None*) Σ′)
⟨*proof*⟩

lemma *comp-K*[*simp*]: (λ-. *x*) ∘ *f* = (λ-. *x*)
⟨*proof*⟩

lemma *is-ok-sp-generalize*:
assumes *is-ok* (*sp F xs* (*map* (λ-. *None*) Σ))
shows *is-ok* (*sp F* (*map generalize-instr xs*) (*map* (λ-. *None*) Σ))
⟨*proof*⟩

lemma *is-ok-sp-generalize2*:
assumes *is-ok* (*sp F xs* (*replicate n None*))
shows *is-ok* (*sp F* (*map generalize-instr xs*) (*replicate n None*))
⟨*proof*⟩

lemma *sp-instr-op*:
assumes $\mathfrak{DeInl}$ *opinl* = *op*
shows *sp-instr F* (*IOp op*) Σ = *sp-instr F* (*IOpInl opinl*) Σ
⟨*proof*⟩

lemma *sp-list-update*:
assumes
n < *length xs* **and**
∀ Σ. *sp-instr F* (*xs* ! *n*) Σ = *sp-instr F x* Σ
shows *sp F* (*xs*[*n* := *x*]) = *sp F xs*
⟨*proof*⟩

lemma *sp-list-update-eq-Ok*:
assumes
n < *length xs* **and**
∀ Σ. *sp-instr F* (*xs* ! *n*) Σ = *sp-instr F x* Σ **and**

sp F xs ys = Ok zs
shows *sp F (xs[n := x]) ys = Ok zs*
⟨*proof*⟩

lemma *is-ok-sp-list-update*:
assumes
is-ok (sp F xs types) **and**
pc < length xs **and**
∀ Σ. *sp-instr F (xs ! pc) Σ = sp-instr F instr Σ*
shows *is-ok (sp F (xs[pc := instr]) types)*
⟨*proof*⟩

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

end

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

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

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

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

lemma *typeof-unboxed-eq-const*:
fixes *x*
shows
typeof x = None ⟷ (∃ d. x = OpDyn d)
typeof x = Some Ubx1 ⟷ (∃ n. x = OpUbx1 n)
typeof x = Some Ubx2 ⟷ (∃ b. x = OpUbx2 b)
⟨*proof*⟩

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

lemma *cast-inversions*:
cast-Dyn x = Some d ⟹ x = OpDyn d
cast-Ubx1 x = Some n ⟹ x = OpUbx1 n
cast-Ubx2 x = Some b ⟹ x = OpUbx2 b
⟨*proof*⟩

lemma *traverse-cast-Dyn-replicate*:

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

context *unboxedval* **begin**

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

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

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

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

lemma *typeof-and-norm-unboxed-imp-cast-and-box*:
assumes *typeof x′ = Some τ norm-unboxed x′ = x*
shows *cast-and-box τ x′ = Some x*
⟨*proof*⟩

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

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

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

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

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

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

end

end

theory *Inca-to-Ubx-simulation*
imports *List-util Option-applicative Result*
VeriComp.Simulation
Inca Ubx Ubx-type-inference Unboxed-lemmas
begin

14 Locale imports

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

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

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

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

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

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

15 Strongest postcondition

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

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

lemma *eq-sp-to-eq-sp-fundef*:
assumes *Subx.sp F1* = (*Subx.sp F2* :: (*'dyn*, *'id*, -, -, -, -, *'num*, *'bool*) *Ubx.instr list* ⇒ -)
shows *sp-fundef F1* = (*sp-fundef F2* :: - ⇒ (*'dyn*, *'id*, -, -, -, -, *'num*, *'bool*) *Ubx.instr list* ⇒ -)
⟨*proof*⟩

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

16 Normalization

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

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

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

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

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

⟨*proof*⟩

lemma *rel-fundef-rewrite-body*:
assumes
rel-fundef norm-eq fd1 fd2 **and**
norm-instr (*body fd2* ! *pc*) = *norm-instr instr*
shows *rel-fundef norm-eq fd1* (*rewrite-fundef-body fd2 pc instr*)
⟨*proof*⟩

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

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

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

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

lemma *rel-fundefs-Some1*:
assumes *rel-fundefs f g* **and** *f x* = *Some y*
shows ∃ *z*. *g x* = *Some z* ∧ *rel-fundef norm-eq y z*
⟨*proof*⟩

lemma *rel-fundefs-Some2*:
assumes *rel-fundefs f g* **and** *g x* = *Some y*
shows ∃ *z*. *f x* = *Some z* ∧ *rel-fundef norm-eq z y*
⟨*proof*⟩

lemma *rel-fundefs-rel-option*:
assumes *rel-fundefs f g* **and** ⋀*x y*. *rel-fundef norm-eq x y* ⟹ *h x y*
shows *rel-option h* (*f z*) (*g z*)
⟨*proof*⟩

17 Equivalence of call stacks

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

lemma *norm-stack-Nil*[*simp*]: *norm-stack* [] = []
⟨*proof*⟩

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

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

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

lemma *norm-box-stack*[*simp*]: *norm-stack* (*map Subx.box-operand* Σ) = *norm-stack* Σ
⟨*proof*⟩

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

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

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

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

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

lemma *all-same-arities-commutative*: *all-same-arities F1 F2* = *all-same-arities F2 F1*
⟨*proof*⟩

lemma *sp-instr-same-arities*:
all-same-arities F1 F2 ⟹ *Subx.sp-instr F1 x ys* = *Subx.sp-instr F2 x ys*
⟨*proof*⟩

lemma *sp-same-arities*:
assumes *all-same-arities F1 F2*
shows *Subx.sp F1 = Subx.sp F2*
⟨*proof*⟩

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

lemma *all-same-arities-add*:
assumes *Fubx-get F f = Some fd1* **and** *arity fd1 = arity fd2*
shows *all-same-arities* (*Fubx-get F*) (*Fubx-get* (*Fubx-add F f fd2*))
⟨*proof*⟩

lemma *all-same-arities-generalize-fundef*:
assumes *Fubx-get F f = Some fd*
shows *all-same-arities* (*Fubx-get F*) (*Fubx-get* (*Fubx-add F f* (*Subx.generalize-fundef fd*)))
⟨*proof*⟩

lemma *rel-stacktraces-box*:
assumes
rel-stacktraces F1 xs ys opt **and**
F2 f = map-option Subx.generalize-fundef (*F1 f*) **and**
⋀*g*. *f* ≠ *g* ⟹ *F2 g = F1 g* **and**
all-same-arities F1 F2
shows *rel-stacktraces F2 xs* (*Subx.box-stack f ys*) *opt*
⟨*proof*⟩

lemma *rel-stacktraces-generalize*:
assumes
rel-stacktraces (*Fubx-get F*) *st1 st2* (*Some f*) **and**
Fubx-get F f = Some fd **and**
sp-prefix: *sp-fundef* (*Fubx-get F*) *fd* (*take pc* (*body fd*)) = *Ok* (*None # map typeof* Σ*2*) **and**
norm-stacks: Σ*1 = norm-stack* Σ*2* **and**
pc-in-range: *pc < length* (*body fd*) **and**
sp-instr: *Subx.sp-instr* (*Fubx-get F*) (*Subx.generalize-instr* (*body fd* ! *pc*))
(*None # map* (λ-. *None*) Σ*2*) = *Ok* (*None # map* (*typeof* ∘ *Subx.box-operand*) Σ*2*)
shows *rel-stacktraces* (*Fubx-get* (*Fubx-add F f* (*Subx.generalize-fundef fd*)))
(*Frame f* (*Suc pc*) (*d # Σ1*) *# st1*)
(*Frame f* (*Suc pc*) (*OpDyn d # map Subx.box-operand* Σ*2*) *# Subx.box-stack f st2*) *None*
⟨*proof*⟩

lemma *rel-fundefs-rewrite*:
assumes
rel-F1-F2: *rel-fundefs* (*Finca-get F1*) (*Fubx-get F2*) **and**
F2-get-f: *Fubx-get F2 f = Some fd2* **and**
F2-add-f: *Fubx-add F2 f* (*rewrite-fundef-body fd2 pc instr*) = *F2′* **and**

eq-norm: *norm-instr* (*body fd2* ! *pc*) = *norm-instr instr*
shows *rel-fundefs* (*Finca-get F1*) (*Fubx-get F2'*)
⟨*proof*⟩

lemma *rel-fundef-rewrite-both*:
assumes *rel-fundef norm-eq fd1 fd2* **and** *norm-instr y* = *x*
shows *rel-fundef norm-eq* (*rewrite-fundef-body fd1 pc x*) (*rewrite-fundef-body fd2 pc y*)
⟨*proof*⟩

lemma *rel-fundefs-rewrite-both*:
assumes
rel-init: *rel-fundefs* (*Finca-get F1*) (*Fubx-get F2*) **and**
rel-new: *rel-fundef norm-eq fd1 fd2*
shows *rel-fundefs* (*Finca-get* (*Finca-add F1 f fd1*)) (*Fubx-get* (*Fubx-add F2 f fd2*))
⟨*proof*⟩

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

lemma *rel-stacktraces-rewrite-fundef*:
assumes
rel-stacktraces (*Fubx-get F2*) *xs ys opt* **and**
Fubx-get F2 f = *Some fd* **and**
pc < *length* (*body fd*) **and**
∀ Σ. *Subx.sp-instr* (*Fubx-get F2*) (*body fd* ! *pc*) Σ = *Subx.sp-instr* (*Fubx-get F2*) *instr* Σ **and**
¬ *is-fun-call* (*body fd* ! *pc*)
shows *rel-stacktraces*
(*Fubx-get* (*Fubx-add F2 f* (*rewrite-fundef-body fd pc instr*))) *xs ys opt*
⟨*proof*⟩

18 Matching relation

lemma *sp-fundefs-get*:
assumes *sp-fundefs F* **and** *F f* = *Some fd*
shows *sp-fundef F fd* (*body fd*) = *Ok* [*None*]
⟨*proof*⟩

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

lemma *sp-fundefs-add*:
assumes
sp-fundefs (*Fubx-get F*) **and**
sp-fundef (*Fubx-get F*) *fd* (*body fd*) = *Ok* [*None*] **and**
all-same-arities (*Fubx-get F*) (*Fubx-get* (*Fubx-add F f fd*))
shows *sp-fundefs* (*Fubx-get* (*Fubx-add F f fd*))
⟨*proof*⟩

inductive *match* (**infix** ∼ *55*) **where**
rel-fundefs (*Finca-get F1*) (*Fubx-get F2*) ⟹
sp-fundefs (*Fubx-get F2*) ⟹
rel-stacktraces (*Fubx-get F2*) *st1 st2 None* ⟹
match (*State F1 H st1*) (*State F2 H st2*)

19 Backward simulation

lemma *traverse-cast-Dyn-to-norm*: *traverse cast-Dyn xs* = *Some ys* ⟹ *norm-stack xs* = *ys*
⟨*proof*⟩

lemma *traverse-cast-Dyn-to-all-Dyn*:
traverse cast-Dyn xs = *Some ys* ⟹ *list-all* (λ*x*. *typeof x* = *None*) *xs*
⟨*proof*⟩

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

lemma *match-final-backward*:
s1 ∼ *s2* ⟹ *Subx.final s2* ⟹ *Sinca.final s1*
⟨*proof*⟩

sublocale *inca-to-ubx-simulation*:
backward-simulation Sinca.step Subx.step Sinca.final Subx.final λ- -. *False* λ-. *match*
⟨*proof*⟩

20 Forward simulation

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

lemma *forward-lockstep-simulation*:
assumes *Sinca.step s1 s1′* **and** *s1* ∼ *s2*
shows ∃ *s2′*. *Subx.step s2 s2′* ∧ *s1′* ∼ *s2′*

⟨*proof*⟩

lemma *match-final-forward*:
s1 ∼ *s2* ⟹ *Sinca.final s1* ⟹ *Subx.final s2*
⟨*proof*⟩

sublocale *inca-ubx-forward-simulation*:
forward-simulation Sinca.step Subx.step Sinca.final Subx.final λ- -. *False* λ-. *match*
⟨*proof*⟩

21 Bisimulation

sublocale *inca-ubx-bisimulation*:
bisimulation Sinca.step Subx.step Sinca.final Subx.final λ- -. *False* λ-. *match*
⟨*proof*⟩

end

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

22 Generic program rewriting

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

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

lemma *rewrite-prog-map-f*:
assumes ⋀*x Σ1 n x' Σ2*. *rewrite-instr n x Σ1* = *Ok* (*x'*, *Σ2*) ⟹ *f x'* = *x*
shows *rewrite-prog n xs Σ1* = *Ok* (*ys*, *Σ2*) ⟹ *map f ys* = *xs*
⟨*proof*⟩

end

fun *gen-pop-push* **where**
gen-pop-push instr (*domain*, *codomain*) *Σ* = (

```
    let ar = length domain in
    if ar ≤ length Σ ∧ take ar Σ = domain then
      Ok (instr, codomain # drop ar Σ)
    else
      Error ()
  )

context inca-to-ubx-simulation begin

lemma sp-rewrite-prog:
  assumes
    rewrite-prog f n p1 Σ1 = Ok (p2, Σ2) and
    ⋀x Σ1 n x' Σ2. f n x Σ1 = Ok (x', Σ2) ⟹ Subx.sp-instr F x' Σ1 = Ok Σ2
  shows Subx.sp F p2 Σ1 = Ok Σ2
  ⟨proof⟩
```

23 Lifting

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

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

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

lemma sp-lift-instr:
  assumes
    lift-instr instr Σ1 = Ok (instr', Σ2) and
    ⋀x. rel-option (λar fd. arity fd = ar) (get-arity x) (F x)
  shows Subx.sp-instr F instr' Σ1 = Ok Σ2
  ⟨proof⟩
```

lemma *sp-lift*:
assumes
lift n p1 Σ1 = Ok (p2, Σ2) **and**
⋀*x. rel-option* (λ*ar fd. arity fd = ar*) (*get-arity x*) (*F x*)
shows *Subx.sp F p2 Σ1 = Ok Σ2*
⟨*proof*⟩

lemma *norm-lift-instr*: *lift-instr x Σ1 = Ok* (*x′*, *Σ2*) ⟹ *norm-instr x′ = x*
⟨*proof*⟩

lemma *norm-lift*:
assumes *lift n xs Σ1 = Ok* (*ys*, *Σ2*)
shows *map norm-instr ys = xs*
⟨*proof*⟩

24 Optimization

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

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

fun *optim-instr* **where**
optim-instr - (*IPush d*) *Σ* =
try-unbox Ubx1 d Σ unbox-ubx1 IPushUbx1 <|>
try-unbox Ubx2 d Σ unbox-ubx2 IPushUbx2 <|>
Ok (*IPush d*, *None # Σ*)
|
optim-instr - (*IPushUbx1 n*) *Σ* = *Ok* (*IPushUbx1 n*, *Some Ubx1 # Σ*) |
optim-instr - (*IPushUbx2 b*) *Σ* = *Ok* (*IPushUbx2 b*, *Some Ubx2 # Σ*) |
optim-instr - *IPop* (- # *Σ*) = *Ok* (*IPop*, *Σ*) |
optim-instr n (*ILoad x*) (*None # Σ*) = (
case load-oracle n of
Some τ ⇒ *Ok* (*ILoadUbx τ x*, *Some τ # Σ*) |
- ⇒ *Ok* (*ILoad x*, *None # Σ*)
) |
optim-instr - (*ILoadUbx τ x*) (*None # Σ*) = *Ok* (*ILoadUbx τ x*, *Some τ # Σ*) |
optim-instr - (*IStore x*) (*None # None # Σ*) = *Ok* (*IStore x*, *Σ*) |
optim-instr - (*IStore x*) (*None # Some τ # Σ*) = *Ok* (*IStoreUbx τ x*, *Σ*) |
optim-instr - (*IStoreUbx* τ_1 *x*) (*None # Some* τ_2 *# Σ*) = (*if* $\tau_1 = \tau_2$ *then Ok* (*IStoreUbx* τ_1 *x*, *Σ*) *else Error* ()) |
optim-instr - (*IOp op*) *Σ* = *gen-pop-push* (*IOp op*) (*replicate* (𝔄𝔯𝔦𝔱𝔶 *op*) *None*, *None*) *Σ* |
optim-instr - (*IOpInl opinl*) *Σ* = (

```
    let ar = 𝔄𝔯𝔦𝔱𝔶 (𝔇𝔢𝔍𝔫𝔩 opinl) in
    if ar ≤ length Σ then
      case 𝔘𝔟𝔵 opinl (take ar Σ) of
        None ⇒ gen-pop-push (IOpInl opinl) (replicate ar None, None) Σ |
        Some opubx ⇒ Ok (IOpUbx opubx, snd (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭 opubx) # drop ar Σ)
    else
      Error ()
  ) |
  optim-instr - (IOpUbx opubx) Σ = gen-pop-push (IOpUbx opubx) (𝔗𝔶𝔭𝔢𝔒𝔣𝔒𝔭
opubx) Σ |
  optim-instr - (ICall f) Σ = do {
    ar ← Result.of-option () (get-arity f);
    gen-pop-push (ICall f) (replicate ar None, None) Σ
  } |
  optim-instr - - - = Error ()

definition optim where
  optim ≡ rewrite-prog optim-instr

lemma
  assumes
    optim 0 [IPush d1, IPush d2, IStore y] [] = Ok (xs, ys)
    unbox-ubx1 d1 = Some x
    unbox-ubx1 d2 = None unbox-ubx2 d2 = None
  shows xs = [IPushUbx1 x, IPush d2, IStoreUbx Ubx1 y] ∧ ys = []
  ⟨proof⟩

lemma norm-optim-instr: optim-instr n x Σ1 = Ok (x', Σ2) ⟹ norm-instr x' =
norm-instr x
    for x Σ1 n x' Σ2
⟨proof⟩

lemma norm-optim:
  assumes optim n xs Σ1 = Ok (ys, Σ2)
  shows map norm-instr ys = map norm-instr xs
  ⟨proof⟩

lemma sp-optim-instr:
  assumes
    optim-instr n instr Σ1 = Ok (instr', Σ2) and
    ⋀x. rel-option (λar fd. arity fd = ar) (get-arity x) (F x)
  shows Subx.sp-instr F instr' Σ1 = Ok Σ2
  ⟨proof⟩

lemma sp-optim:
  assumes
    optim n p1 Σ1 = Ok (p2, Σ2) and
    ⋀x. rel-option (λar fd. arity fd = ar) (get-arity x) (F x)
  shows Subx.sp F p2 Σ1 = Ok Σ2
```

⟨*proof*⟩

25 Compilation of function definition

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

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

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

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

lemma *rel-compile-fundef*:
  assumes *compile-fundef fd1* = *Ok fd2*
  shows *rel-fundef norm-eq fd1 fd2*
⟨*proof*⟩

lemma *sp-compile-fundef*:
  assumes
    compile-fundef fd1 = *Ok fd2* **and**
    ⋀*x*. *rel-option* (λ*ar fd*. *arity fd* = *ar*) (*get-arity x*) (*F x*)
  shows *sp-fundef F fd2* (*body fd2*) = *Ok* [*None*]
⟨*proof*⟩

end
end

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

26 Compilation of function environment

definition *compile-env-entry* **where**

compile-env-entry $\mathcal{A}$ $\mathcal{O}$ *x* = *map-result id* (*Pair* (*fst x*)) (*compile-fundef* $\mathcal{A}$ ($\mathcal{O}$ (*fst x*)) (*snd x*))

lemma *rel-compile-env-entry*:
assumes *compile-env-entry* $\mathcal{O}$ $\mathcal{A}$ (*f*, *fd1*) = *Ok* (*f*, *fd2*)
shows *rel-fundef norm-eq fd1 fd2*
⟨*proof*⟩

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

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

lemma *list-assoc-those-compile-env-entries*:
Result.those (*map* (*compile-env-entry* $\mathcal{A}$ $\mathcal{O}$) *xs*) = *Ok ys* ⟹
rel-option (λ*fd1 fd2*. *compile-fundef* $\mathcal{A}$ ($\mathcal{O}$ *f*) *fd1* = *Ok fd2*) (*map-of xs f*) (*map-of ys f*)
⟨*proof*⟩

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

lemma *rel-those-compile-env-entries*:
Result.those (*map* (*compile-env-entry* $\mathcal{A}$ $\mathcal{O}$) *xs*) = *Ok ys* ⟹
rel-fundefs (*Finca-get* (*Sinca.Fenv.from-list xs*)) (*Fubx-get* (*Subx.Fenv.from-list ys*))
⟨*proof*⟩

lemma *rel-fundefs-compile-env*:
assumes *compile-env* $\mathcal{A}$ $\mathcal{O}$ *e* = *Ok e′*
shows *rel-fundefs* (*Finca-get e*) (*Fubx-get e′*)
⟨*proof*⟩

lemma *sp-fundefs-compile-env*:
assumes
compile-env $\mathcal{A}$ $\mathcal{O}$ *F1* = *Ok F2* **and**
$\bigwedge$*x*. *rel-option* (λ*ar fd*. *arity fd* = *ar*) ($\mathcal{A}$ *x*) (*Finca-get F1 x*)
shows *sp-fundefs* (*Fubx-get F2*)
⟨*proof*⟩

27 Compilation of program

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

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

interpretation *std-to-inca-compiler*:
  compiler Sinca.step Subx.step Sinca.final Subx.final Sinca.load Subx.load
    λ- -. *False* λ-. *match compile*
⟨*proof*⟩

end

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

28 Dynamic values

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

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

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

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

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

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

fun *unbox-bool* :: *dynamic* ⇒ *bool option* **where**
  unbox-bool (*DBool b*) = *Some b* |

unbox-bool - = None

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

29 Normal operations

datatype *op* =
OpNeg |
OpAdd |
OpMul

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

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

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

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

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

lemma *eval-arith-domain*: *length xs* = *ar op* ⟹ ∃ *y*. *eval op xs* = *y*
⟨*proof*⟩

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

30 Inlined operations

datatype *opinl* =
OpAddNum |
OpMulNum

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

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

fun *deinl* :: *opinl* ⇒ *op* **where**
deinl OpAddNum = *OpAdd* |
deinl OpMulNum = *OpMul*

lemma *inl-inj*: *inj inl*
⟨*proof*⟩

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

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

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

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

lemma *eval-AddNum-correct*:
length xs = 2 ⟹ *eval-AddNum xs* = *eval-Add xs*
⟨*proof*⟩

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

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

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

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

31 Unboxed operations

datatype *opubx* =
OpAddNumUbx

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

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

lemma *ubx-invertible*: *ubx opinl xs* = *Some opubx* ⟹ *deubx opubx* = *opinl*
⟨*proof*⟩

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

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

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

31.1 Abstract interpretation

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

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

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

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

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

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

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

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

locale *std* =
Fenv: *env F-empty F-get F-add F-to-list* +
Henv: *env heap-empty heap-get heap-add heap-to-list* +
dynval is-true is-false +
nary-operations 𝔒𝔭 𝔄𝔯𝔦𝔱𝔶
for
F-empty **and**
F-get :: *'fenv* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*) *instr fundef option* **and**
F-add **and** *F-to-list* **and**
heap-empty **and**
heap-get :: *'henv* ⇒ *'var* × *'dyn* ⇒ *'dyn option* **and**
heap-add **and** *heap-to-list* **and**
is-true :: *'dyn* ⇒ *bool* **and** *is-false* **and**

$\mathfrak{Op}$ :: *'op* ⇒ *'dyn list* ⇒ *'dyn* **and** $\mathfrak{Arity}$
begin

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

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

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

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

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

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

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

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

step-fun-call:
F-get F f = Some fd ⟹ *pc < length* (*body fd*) ⟹ *body fd* ! *pc = ICall g* ⟹
F-get F g = Some gd ⟹ *arity gd* ≤ *length* Σ ⟹

$frame_f = Frame\ f\ pc\ \Sigma \Longrightarrow frame_g = Frame\ g\ 0\ (take\ (arity\ gd)\ \Sigma) \Longrightarrow$
$State\ F\ H\ (frame_f\ \#\ st) \rightarrow State\ F\ H\ (frame_g\ \#\ frame_f\ \#\ st)\ |$

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

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

lemma *final-finished*: $final\ s \Longrightarrow finished\ step\ s$
⟨*proof*⟩

sublocale *semantics step final*
⟨*proof*⟩

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

sublocale *language step final load*
⟨*proof*⟩

end

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

32 Generic definitions

print-locale *std*

locale *std-inca-simulation* =
Sstd: *std*
Fstd-empty Fstd-get Fstd-add Fstd-to-list
heap-empty heap-get heap-add heap-to-list
is-true is-false
𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 +
Sinca: *inca*
Finca-empty Finca-get Finca-add Finca-to-list
heap-empty heap-get heap-add heap-to-list
is-true is-false
𝔒𝔭 𝔄𝔯𝔦𝔱𝔶 𝔍𝔫𝔩𝔒𝔭 𝔍𝔫𝔩 𝔍𝔰𝔍𝔫𝔩 𝔇𝔢𝔍𝔫𝔩

for
— Functions environments
Fstd-empty **and**
Fstd-get :: *'fenv-std* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*) *Std.instr fundef option*
and
Fstd-add **and** *Fstd-to-list* **and**

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

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

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

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

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

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

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

lemma *rel-fundefs-Some1*:
assumes *rel-fundefs f g* **and** *f x* = *Some y*
shows ∃ *z*. *g x* = *Some z* ∧ *rel-fundef norm-eq y z*
⟨*proof*⟩

lemma *rel-fundefs-Some2*:
assumes *rel-fundefs f g* **and** *g x* = *Some y*
shows ∃ *z*. *f x* = *Some z* ∧ *rel-fundef norm-eq z y*
⟨*proof*⟩

lemma *rel-fundef-body-nth*:
assumes *rel-fundef norm-eq fd1 fd2* **and** *pc* < *length* (*body fd1*)
shows *body fd1* ! *pc* = *norm-instr* (*body fd2* ! *pc*)
⟨*proof*⟩

lemma *rel-fundef-rewrite-body*:
assumes
rel-fundef norm-eq fd1 fd2 **and**
norm-instr (*body fd2* ! *pc*) = *norm-instr instr*
shows *rel-fundef norm-eq fd1* (*rewrite-fundef-body fd2 pc instr*)
⟨*proof*⟩

lemma *rel-fundefs-rewrite*:
assumes
rel-F1-F2: *rel-fundefs* (*Fstd-get F1*) (*Finca-get F2*) **and**
F2-get-f: *Finca-get F2 f* = *Some fd2* **and**
F2-add-f: *Finca-add F2 f* (*rewrite-fundef-body fd2 pc instr*) = *F2′* **and**
norm-eq: *norm-instr* (*body fd2* ! *pc*) = *norm-instr instr*
shows *rel-fundefs* (*Fstd-get F1*) (*Finca-get F2′*)
⟨*proof*⟩

33 Simulation relation

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

34 Backward simulation

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

lemma *match-final-backward*:
s1 ∼ *s2* ⟹ *Sinca.final s2* ⟹ *Sstd.final s1*
⟨*proof*⟩

sublocale *std-inca-simulation*:
backward-simulation Sstd.step Sinca.step Sstd.final Sinca.final λ- -. *False* λ-. *match*
⟨*proof*⟩

35 Forward simulation

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

⟨*proof*⟩

lemma *forward-match-final*:
s1 ∼ *s2* ⟹ *Sstd.final s1* ⟹ *Sinca.final s2*
⟨*proof*⟩

sublocale *std-inca-forward-simulation*:
forward-simulation Sstd.step Sinca.step Sstd.final Sinca.final λ- -. *False* λ-. *match*
⟨*proof*⟩

36 Bisimulation

sublocale *std-inca-bisimulation*:
bisimulation Sstd.step Sinca.step Sstd.final Sinca.final λ- -. *False* λ-. *match*
⟨*proof*⟩

end

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

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

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

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

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

lemma *rel-compile-fundef*: *rel-fundef norm-eq fd* (*compile-fundef fd*)
⟨*proof*⟩

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

lemma *Finca-get-compile*: *Finca-get* (*compile-env e*) *x* = *map-option compile-fundef* (*Fstd-get e x*)
⟨*proof*⟩

lemma *rel-fundefs-compile-env*: *rel-fundefs* (*Fstd-get e*) (*Finca-get* (*compile-env e*))
⟨*proof*⟩

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

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

sublocale *std-to-inca-compiler*:
compiler Sstd.step Sinca.step Sstd.final Sinca.final Sstd.load Sinca.load
λ- -. *False* λ-. *match compile*
⟨*proof*⟩

theorem *compile-complete*:
assumes *Sstd.load* p_1 s_1
shows ∃ p_2 s_2. *compile* p_1 = *Some* p_2 ∧ *Sinca.load* p_2 s_2 ∧ *match* s_1 s_2
⟨*proof*⟩

end

end

References

[1] M. Desharnais and S. Brunthaler. Towards efficient and verified virtual machines for dynamic languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021. Association for Computing Machinery, 2021.