

Declarative Semantics for Functional Languages

Jeremy G. Siek

September 13, 2023

Abstract

We present a semantics for an applied call-by-value lambda-calculus that is compositional, extensional, and elementary. We present four different views of the semantics: 1) as a relational (big-step) semantics that is not operational but instead declarative, 2) as a denotational semantics that does not use domain theory, 3) as a non-deterministic interpreter, and 4) as a variant of the intersection type systems of the Torino group. We prove that the semantics is correct by showing that it is sound and complete with respect to operational semantics on programs and that is sound with respect to contextual equivalence. We have not yet investigated whether it is fully abstract. We demonstrate that this approach to semantics is useful with three case studies. First, we use the semantics to prove correctness of a compiler optimization that inlines function application. Second, we adapt the semantics to the polymorphic lambda-calculus extended with general recursion and prove semantic type soundness. Third, we adapt the semantics to the call-by-value lambda-calculus with mutable references. The paper that accompanies these Isabelle theories is available on arXiv at the following URL:

<https://arxiv.org/abs/1707.03762>

Contents

1	Syntax of the lambda calculus	2
2	Small-step semantics of CBV lambda calculus	2
3	Big-step semantics of CBV lambda calculus	4
3.1	Big-step semantics is sound wrt. small-step semantics	4
3.2	Big-step semantics is deterministic	6
4	Declarative semantics as a relational semantics	8
5	Declarative semantics as a denotational semantics	8
6	Relational and denotational views are equivalent	8
7	Subsumption and change of environment	9
8	Declarative semantics as a non-deterministic interpreter	10
8.1	Non-determinism monad	10
8.2	Non-deterministic interpreter	11
9	Declarative semantics as a type system	11
10	Declarative semantics with tables as lists	12
10.1	Definition of values for declarative semantics	12
10.2	Properties about values	13
10.3	Declarative semantics as a denotational semantics	14
10.4	Subsumption and change of environment	15
11	Equivalence of denotational and type system views	16

12 Soundness of the declarative semantics wrt. operational	18
12.1 Substitution preserves denotation	18
12.2 Reduction preserves denotation	18
12.3 Progress	19
12.4 Logical relation between values and big-step values	19
12.5 Denotational semantics sound wrt. big-step	20
13 Completeness of the declarative semantics wrt. operational	20
13.1 Reverse substitution preserves denotation	20
13.2 Reverse reduction preserves denotation	21
13.3 Completeness	21
14 Soundness wrt. contextual equivalence	21
14.1 Denotational semantics is a congruence	21
14.2 Auxiliary lemmas	22
14.3 Soundness wrt. contextual equivalence	22
14.4 Denotational equalities regarding reduction	22
15 Correctness of an optimizer	23
16 Semantics and type soundness for System F	23
16.1 Syntax and values	23
16.2 Set monad	24
16.3 Denotational semantics	24
16.4 Types: substitution and semantics	25
16.5 Type system	25
16.6 Well-typed Programs don't go wrong	26
17 Semantics of mutable references	27
17.1 Denotations (values)	27
17.2 Non-deterministic state monad	28
17.3 Denotational semantics	29

1 Syntax of the lambda calculus

```
theory Lambda
imports Main
begin

type-synonym name = nat

datatype exp = EVar name | ENat nat | ELam name exp | EApp exp exp
  | EPrim nat  $\Rightarrow$  nat  $\Rightarrow$  nat exp exp | EIf exp exp exp

fun lookup :: ('a  $\times$  'b) list  $\Rightarrow$  'a  $\Rightarrow$  'b option where
  lookup [] x = None |
  lookup ((y,v)#ls) x = (if (x = y) then Some v else lookup ls x)

fun FV :: exp  $\Rightarrow$  nat set where
  FV (EVar x) = {x} |
  FV (ENat n) = {} |
  FV (ELam x e) = FV e - {x} |
  FV (EApp e1 e2) = FV e1  $\cup$  FV e2 |
  FV (EPrim f e1 e2) = FV e1  $\cup$  FV e2 |
  FV (EIf e1 e2 e3) = FV e1  $\cup$  FV e2  $\cup$  FV e3

fun BV :: exp  $\Rightarrow$  nat set where
  BV (EVar x) = {x} |
  BV (ENat n) = {} |
  BV (ELam x e) = BV e  $\cup$  {x} |
  BV (EApp e1 e2) = BV e1  $\cup$  BV e2 |
  BV (EPrim f e1 e2) = BV e1  $\cup$  BV e2 |
  BV (EIf e1 e2 e3) = BV e1  $\cup$  BV e2  $\cup$  BV e3

end
```

2 Small-step semantics of CBV lambda calculus

```
theory SmallStepLam
imports Lambda
begin
```

The following substitution function is not capture avoiding, so it has a precondition that v is closed. With hindsight, we should have used DeBruijn indices instead because we also use substitution in the optimizing compiler.

```
fun subst :: name  $\Rightarrow$  exp  $\Rightarrow$  exp  $\Rightarrow$  exp where
  subst x v (EVar y) = (if x = y then v else EVar y) |
  subst x v (ENat n) = ENat n |
  subst x v (ELam y e) = (if x = y then ELam y e else ELam y (subst x v e)) |
  subst x v (EApp e1 e2) = EApp (subst x v e1) (subst x v e2) |
  subst x v (EPrim f e1 e2) = EPrim f (subst x v e1) (subst x v e2) |
  subst x v (EIf e1 e2 e3) = EIf (subst x v e1) (subst x v e2) (subst x v e3)
```

```
inductive isval :: exp  $\Rightarrow$  bool where
  valnat[intro!]: isval (ENat n) |
  vallam[intro!]: isval (ELam x e)
```

```
inductive-cases
  isval-var-inv[elim!]: isval (EVar x) and
  isval-app-inv[elim!]: isval (EApp e1 e2) and
  isval-prim-inv[elim!]: isval (EPrim f e1 e2) and
  isval-if-inv[elim!]: isval (EIf e1 e2 e3)
```

definition *is-val* :: *exp* \Rightarrow *bool* **where**

is-val *v* \equiv *isval* *v* \wedge *FV* *v* = {}

declare *is-val-def*[*simp*]

inductive *reduce* :: *exp* \Rightarrow *exp* \Rightarrow *bool* (**infix** \longrightarrow 55) **where**

beta[*intro!*]: $\llbracket \text{is-val } v \rrbracket \Longrightarrow \text{EApp } (\text{ELam } x \ e) \ v \longrightarrow (\text{subst } x \ v \ e) \mid$
app-left[*intro!*]: $\llbracket e1 \longrightarrow e1' \rrbracket \Longrightarrow \text{EApp } e1 \ e2 \longrightarrow \text{EApp } e1' \ e2 \mid$
app-right[*intro!*]: $\llbracket e2 \longrightarrow e2' \rrbracket \Longrightarrow \text{EApp } e1 \ e2 \longrightarrow \text{EApp } e1 \ e2' \mid$
delta[*intro!*]: $\text{EPrim } f \ (\text{ENat } n1) \ (\text{ENat } n2) \longrightarrow \text{ENat } (f \ n1 \ n2) \mid$
prim-left[*intro!*]: $\llbracket e1 \longrightarrow e1' \rrbracket \Longrightarrow \text{EPrim } f \ e1 \ e2 \longrightarrow \text{EPrim } f \ e1' \ e2 \mid$
prim-right[*intro!*]: $\llbracket e2 \longrightarrow e2' \rrbracket \Longrightarrow \text{EPrim } f \ e1 \ e2 \longrightarrow \text{EPrim } f \ e1 \ e2' \mid$
if-zero[*intro!*]: $\text{EIf } (\text{ENat } 0) \ \text{thn } \ \text{els} \longrightarrow \ \text{els} \mid$
if-nz[*intro!*]: $n \neq 0 \Longrightarrow \text{EIf } (\text{ENat } n) \ \text{thn } \ \text{els} \longrightarrow \ \text{thn} \mid$
if-cond[*intro!*]: $\llbracket \text{cond} \longrightarrow \text{cond}' \rrbracket \Longrightarrow$
 $\text{EIf } \text{cond} \ \text{thn } \ \text{els} \longrightarrow \text{EIf } \text{cond}' \ \text{thn } \ \text{els}$

inductive-cases

red-var-inv[*elim!*]: $\text{EVar } x \longrightarrow e \ \text{and}$
red-int-inv[*elim!*]: $\text{ENat } n \longrightarrow e \ \text{and}$
red-lam-inv[*elim!*]: $\text{ELam } x \ e \longrightarrow e' \ \text{and}$
red-app-inv[*elim!*]: $\text{EApp } e1 \ e2 \longrightarrow e'$

inductive *multi-step* :: *exp* \Rightarrow *exp* \Rightarrow *bool* (**infix** \longrightarrow^* 55) **where**

ms-nil[*intro!*]: $e \longrightarrow^* e \mid$
ms-cons[*intro!*]: $\llbracket e1 \longrightarrow e2; e2 \longrightarrow^* e3 \rrbracket \Longrightarrow e1 \longrightarrow^* e3$

definition *diverge* :: *exp* \Rightarrow *bool* **where**

diverge *e* \equiv $(\forall e'. e \longrightarrow^* e' \longrightarrow (\exists e''. e' \longrightarrow e''))$

definition *stuck* :: *exp* \Rightarrow *bool* **where**

stuck *e* \equiv $\neg (\exists e'. e \longrightarrow e')$

declare *stuck-def*[*simp*]

definition *goes-wrong* :: *exp* \Rightarrow *bool* **where**

goes-wrong *e* \equiv $\exists e'. e \longrightarrow^* e' \wedge \text{stuck } e' \wedge \neg \text{isval } e'$

declare *goes-wrong-def*[*simp*]

datatype *obs* = *ONat* *nat* \mid *OFun* \mid *OBad*

fun *observe* :: *exp* \Rightarrow *obs* \Rightarrow *bool* **where**

observe (*ENat* *n*) (*ONat* *n'*) = $(n = n')$ \mid
observe (*ELam* *x* *e*) *OFun* = *True* \mid
observe *e* *ob* = *False*

definition *run* :: *exp* \Rightarrow *obs* \Rightarrow *bool* (**infix** \Downarrow 52) **where**

run *e* *ob* \equiv $((\exists v. e \longrightarrow^* v \wedge \text{observe } v \ \text{ob})$
 $\vee ((\text{diverge } e \vee \text{goes-wrong } e) \wedge \text{ob} = \text{OBad}))$

lemma *val-stuck*: **fixes** *e*::*exp* **assumes** *val-e*: *isval* *e* **shows** *stuck* *e*

<proof>

lemma *subst-fv-aux*: **assumes** *fvv*: *FV* *v* = {} **shows** *FV* (*subst* *x* *v* *e*) \subseteq *FV* *e* - {*x*}

<proof>

lemma *subst-fv*: **assumes** *fv-e*: *FV* *e* \subseteq {*x*} **and** *fv-v*: *FV* *v* = {}

shows *FV* (*subst* *x* *v* *e*) = {}

<proof>

lemma *red-pres-fv*: **fixes** *e*::*exp* **assumes** *red*: *e* \longrightarrow *e'* **and** *fv*: *FV* *e* = {} **shows** *FV* *e'* = {}

<proof>

lemma *reduction-pres-fv*: **fixes** $e::exp$ **assumes** $r: e \longrightarrow^* e'$ **and** $fv: FV\ e = \{\}$ **shows** $FV\ e' = \{\}$
<proof>

end

3 Big-step semantics of CBV lambda calculus

theory *BigStepLam*

imports *Lambda SmallStepLam*

begin

datatype *bval*

= *BNat nat*

| *BClos name exp (name × bval) list*

type-synonym *benv* = *(name × bval) list*

inductive *eval* :: *benv* \Rightarrow *exp* \Rightarrow *bval* \Rightarrow *bool* ($- \vdash - \Downarrow - [50,50,50] 51$) **where**

eval-nat[*intro!*]: $\varrho \vdash ENat\ n \Downarrow BNat\ n \mid$

eval-var[*intro!*]: $lookup\ \varrho\ x = Some\ v \Longrightarrow \varrho \vdash EVar\ x \Downarrow v \mid$

eval-lam[*intro!*]: $\varrho \vdash ELam\ x\ e \Downarrow BClos\ x\ e\ \varrho \mid$

eval-app[*intro!*]: $\llbracket \varrho \vdash e1 \Downarrow BClos\ x\ e\ \varrho'; \varrho \vdash e2 \Downarrow arg;$

$(x,arg)\#\varrho' \vdash e \Downarrow v \rrbracket \Longrightarrow$

$\varrho \vdash EApp\ e1\ e2 \Downarrow v \mid$

eval-prim[*intro!*]: $\llbracket \varrho \vdash e1 \Downarrow BNat\ n1; \varrho \vdash e2 \Downarrow BNat\ n2 ; n3 = f\ n1\ n2 \rrbracket \Longrightarrow$

$\varrho \vdash EPrim\ f\ e1\ e2 \Downarrow BNat\ n3 \mid$

eval-if0[*intro!*]: $\llbracket \varrho \vdash e1 \Downarrow BNat\ 0; \varrho \vdash e3 \Downarrow v3 \rrbracket \Longrightarrow$

$\varrho \vdash EIf\ e1\ e2\ e3 \Downarrow v3 \mid$

eval-if1[*intro!*]: $\llbracket \varrho \vdash e1 \Downarrow BNat\ n; n \neq 0; \varrho \vdash e2 \Downarrow v2 \rrbracket \Longrightarrow$

$\varrho \vdash EIf\ e1\ e2\ e3 \Downarrow v2$

inductive-cases

eval-nat-inv[*elim!*]: $\varrho \vdash ENat\ n \Downarrow v$ **and**

eval-var-inv[*elim!*]: $\varrho \vdash EVar\ x \Downarrow v$ **and**

eval-lam-inv[*elim!*]: $\varrho \vdash ELam\ x\ e \Downarrow v$ **and**

eval-app-inv[*elim!*]: $\varrho \vdash EApp\ e1\ e2 \Downarrow v$ **and**

eval-prim-inv[*elim!*]: $\varrho \vdash EPrim\ f\ e1\ e2 \Downarrow v$ **and**

eval-if-inv[*elim!*]: $\varrho \vdash EIf\ e1\ e2\ e3 \Downarrow v$

3.1 Big-step semantics is sound wrt. small-step semantics

type-synonym *env* = *(name × exp) list*

fun *psubst* :: *env* \Rightarrow *exp* \Rightarrow *exp* **where**

psubst ϱ (*ENat* n) = *ENat* n \mid

psubst ϱ (*EVar* x) =

(*case lookup* ϱ x of

None \Rightarrow *EVar* x

\mid *Some* $v \Rightarrow v$) \mid

psubst ϱ (*ELam* $x\ e$) = *ELam* x (*psubst* $((x,EVar\ x)\#\varrho)$ e) \mid

psubst ϱ (*EApp* $e1\ e2$) = *EApp* (*psubst* ϱ $e1$) (*psubst* ϱ $e2$) \mid

psubst ϱ (*EPrim* $f\ e1\ e2$) = *EPrim* f (*psubst* ϱ $e1$) (*psubst* ϱ $e2$) \mid

psubst ϱ (*EIf* $e1\ e2\ e3$) = *EIf* (*psubst* ϱ $e1$) (*psubst* ϱ $e2$) (*psubst* ϱ $e3$)

inductive *bs-val* :: *bval* \Rightarrow *exp* \Rightarrow *bool* **and**

bs-env :: *benv* \Rightarrow *env* \Rightarrow *bool* **where**

bs-nat[*intro!*]: *bs-val* (*BNat* n) (*ENat* n) \mid

bs-clos[*intro!*]: $\llbracket bs-env\ \varrho\ \varrho'; FV\ (ELam\ x\ (psubst\ ((x,EVar\ x)\#\varrho')\ e)) = \{\} \rrbracket \Longrightarrow$

bs-val (*BClos* $x\ e\ \varrho$) (*ELam* x (*psubst* $((x,EVar\ x)\#\varrho')$ e)) \mid

bs-nil[*intro!*]: $bs\text{-env } [] \mid$
bs-cons[*intro!*]: $\llbracket bs\text{-val } w \ v; \ bs\text{-env } \varrho \ \varrho' \rrbracket \implies bs\text{-env } ((x,w)\#\varrho) \ ((x,v)\#\varrho')$

inductive-cases *bs-env-inv1*[*elim!*]: $bs\text{-env } ((x, w) \# \varrho) \ \varrho'$ **and**

bs-clos-inv[*elim!*]: $bs\text{-val } (BClos \ x \ e \ \varrho'') \ v1$ **and**

bs-nat-inv[*elim!*]: $bs\text{-val } (BNat \ n) \ v$

lemma *bs-val-is-val*[*intro!*]: $bs\text{-val } w \ v \implies is\text{-val } v$
 ⟨*proof*⟩

lemma *lookup-bs-env*: $\llbracket bs\text{-env } \varrho \ \varrho'; \ lookup \ \varrho \ x = Some \ w \rrbracket \implies$
 $\exists v. \ lookup \ \varrho' \ x = Some \ v \wedge bs\text{-val } w \ v$
 ⟨*proof*⟩

lemma *app-red-cong1*: $e1 \longrightarrow* e1' \implies EApp \ e1 \ e2 \longrightarrow* EApp \ e1' \ e2$
 ⟨*proof*⟩

lemma *app-red-cong2*: $e2 \longrightarrow* e2' \implies EApp \ e1 \ e2 \longrightarrow* EApp \ e1 \ e2'$
 ⟨*proof*⟩

lemma *prim-red-cong1*: $e1 \longrightarrow* e1' \implies EPrim \ f \ e1 \ e2 \longrightarrow* EPrim \ f \ e1' \ e2$
 ⟨*proof*⟩

lemma *prim-red-cong2*: $e2 \longrightarrow* e2' \implies EPrim \ f \ e1 \ e2 \longrightarrow* EPrim \ f \ e1 \ e2'$
 ⟨*proof*⟩

lemma *if-red-cong1*: $e1 \longrightarrow* e1' \implies EIf \ e1 \ e2 \ e3 \longrightarrow* EIf \ e1' \ e2 \ e3$
 ⟨*proof*⟩

lemma *multi-step-trans*: $\llbracket e1 \longrightarrow* e2; \ e2 \longrightarrow* e3 \rrbracket \implies e1 \longrightarrow* e3$
 ⟨*proof*⟩

lemma *subst-id-fv*: $x \notin FV \ e \implies subst \ x \ v \ e = e$
 ⟨*proof*⟩

definition *sdom* :: $env \Rightarrow name \ set$ **where**
 $sdom \ \varrho \equiv \{x. \exists v. \ lookup \ \varrho \ x = Some \ v \wedge v \neq EVar \ x \}$

definition *closed-env* :: $env \Rightarrow bool$ **where**
 $closed\text{-env} \ \varrho \equiv (\forall x \ v. \ x \in sdom \ \varrho \longrightarrow lookup \ \varrho \ x = Some \ v \longrightarrow FV \ v = \{\})$

definition *equiv-env* :: $env \Rightarrow env \Rightarrow bool$ **where**
 $equiv\text{-env} \ \varrho \ \varrho' \equiv (sdom \ \varrho = sdom \ \varrho' \wedge (\forall x. \ x \in sdom \ \varrho \longrightarrow lookup \ \varrho \ x = lookup \ \varrho' \ x))$

lemma *sdom-cons-xx*[*simp*]: $sdom \ ((x, EVar \ x)\#\varrho) = sdom \ \varrho - \{x\}$
 ⟨*proof*⟩

lemma *sdom-cons-v*[*simp*]: $FV \ v = \{\} \implies sdom \ ((x,v)\#\varrho) = insert \ x \ (sdom \ \varrho)$
 ⟨*proof*⟩

lemma *lookup-some-in-dom*: $\llbracket lookup \ \varrho \ x = Some \ v; \ v \neq EVar \ x \rrbracket \implies x \in sdom \ \varrho$
 ⟨*proof*⟩

lemma *lookup-none-notin-dom*: $lookup \ \varrho \ x = None \implies x \notin sdom \ \varrho$
 ⟨*proof*⟩

lemma *psubst-change*: $equiv\text{-env} \ \varrho \ \varrho' \implies psubst \ \varrho \ e = psubst \ \varrho' \ e$
 ⟨*proof*⟩

lemma *subst-psubst*: $\llbracket closed\text{-env} \ \varrho; \ FV \ v = \{\} \rrbracket \implies$
 $subst \ x \ v \ (psubst \ ((x, EVar \ x) \# \varrho) \ e) = psubst \ ((x, v) \# \varrho) \ e$

<proof>

inductive-cases *bsenv-nil[elim!]*: $bs\text{-env } [] \ \varrho'$

lemma *bs-env-dom*: $bs\text{-env } \varrho \ \varrho' \implies set \ (map \ fst \ \varrho) = sdom \ \varrho'$
<proof>

lemma *closed-env-cons[intro!]*: $FV \ v = \{\} \implies closed\text{-env } \varrho'' \implies closed\text{-env } ((a, v) \# \varrho'')$
<proof>

lemma *bs-env-closed*: $bs\text{-env } \varrho \ \varrho' \implies closed\text{-env } \varrho'$
<proof>

lemma *psubst-fv*: $closed\text{-env } \varrho \implies FV \ (psubst \ \varrho \ e) = FV \ e - sdom \ \varrho$
<proof>

lemma *big-small-step*:

assumes *ev*: $\varrho \vdash e \Downarrow w$ **and** *r-rp*: $bs\text{-env } \varrho \ \varrho'$ **and** *fv-e*: $FV \ e \subseteq set \ (map \ fst \ \varrho)$
shows $\exists v. psubst \ \varrho' \ e \longrightarrow* v \wedge is\text{-val } v \wedge bs\text{-val } w \ v$
<proof>

lemma *psubst-id*: $FV \ e \cap sdom \ \varrho = \{\} \implies psubst \ \varrho \ e = e$
<proof>

fun *bs-observe* :: $bval \Rightarrow obs \Rightarrow bool$ **where**
bs-observe (BNat n) (ONat n') = $(n = n')$ |
bs-observe (BClos $x \ e \ \varrho$) OFun = True |
bs-observe $e \ ob$ = False

theorem *sound-wrt-small-step*:

assumes *e-v*: $[] \vdash e \Downarrow v$ **and** *fv-e*: $FV \ e = \{\}$
shows $\exists v' \ ob. e \longrightarrow* v' \wedge isval \ v' \wedge observe \ v' \ ob$
 $\wedge bs\text{-observe } v \ ob$

<proof>

3.2 Big-step semantics is deterministic

theorem *big-step-fun*:

assumes *ev*: $\varrho \vdash e \Downarrow v$ **and** *evp*: $\varrho \vdash e \Downarrow v'$ **shows** $v = v'$
<proof>

end

theory *ValuesFSet*

imports *Main Lambda HOL-Library.FSet*

begin

datatype *val* = VNat nat | VFun (val × val) fset

type-synonym *func* = (val × val) fset

inductive *val-le* :: $val \Rightarrow val \Rightarrow bool$ (**infix** \sqsubseteq 52) **where**
vnat-le[intro!]: $(VNat \ n) \sqsubseteq (VNat \ n)$ |
vfun-le[intro!]: $fset \ t1 \subseteq fset \ t2 \implies (VFun \ t1) \sqsubseteq (VFun \ t2)$

type-synonym *env* = ((name × val) list)

definition *env-le* :: $env \Rightarrow env \Rightarrow bool$ (**infix** \sqsubseteq 52) **where**
 $\varrho \sqsubseteq \varrho' \equiv \forall x \ v. lookup \ \varrho \ x = Some \ v \longrightarrow (\exists v'. lookup \ \varrho' \ x = Some \ v' \wedge v \sqsubseteq v')$

definition *env-eq* :: $env \Rightarrow env \Rightarrow bool$ (**infix** \approx 50) **where**

$\varrho \approx \varrho' \equiv (\forall x. \text{lookup } \varrho \ x = \text{lookup } \varrho' \ x)$

fun *vadd* :: (val × nat) × (val × nat) ⇒ nat ⇒ nat **where**
vadd ((-,v),(-,u)) r = v + u + r

primrec *vsize* :: val ⇒ nat **where**
vsize (VNat n) = 1 |
vsize (VFun t) = 1 + *ffold* *vadd* 0
(fimage (map-prod (λ v. (v,vsize v)) (λ v. (v,vsize v))) t)

abbreviation *vprod-size* :: val × val ⇒ (val × nat) × (val × nat) **where**
vprod-size ≡ map-prod (λ v. (v,vsize v)) (λ v. (v,vsize v))

abbreviation *fsize* :: func ⇒ nat **where**
fsize t ≡ 1 + *ffold* *vadd* 0 (fimage *vprod-size* t)

interpretation *vadd-vprod*: comp-fun-commute *vadd* ∘ *vprod-size*
⟨proof⟩

lemma *vprod-size-inj*: inj-on *vprod-size* (fset A)
⟨proof⟩

lemma *fsize-def2*: *fsize* t = 1 + *ffold* (*vadd* ∘ *vprod-size*) 0 t
⟨proof⟩

lemma *fsize-finsert-in[simp]*:
assumes *v12-t*: (v1,v2) |∈| t **shows** *fsize* (finsert (v1,v2) t) = *fsize* t
⟨proof⟩

lemma *fsize-finsert-notin[simp]*:
assumes *v12-t*: (v1,v2) |∉| t
shows *fsize* (finsert (v1,v2) t) = *vsize* v1 + *vsize* v2 + *fsize* t
⟨proof⟩

end
theory *ValuesFSetProps*
imports *ValuesFSet*
begin

inductive-cases
vfun-le-inv[elim!]: VFun t1 ⊆ VFun t2 **and**
le-fun-nat-inv[elim!]: VFun t2 ⊆ VNat x1 **and**
le-any-nat-inv[elim!]: v ⊆ VNat n **and**
le-nat-any-inv[elim!]: VNat n ⊆ v **and**
le-fun-any-inv[elim!]: VFun t ⊆ v **and**
le-any-fun-inv[elim!]: v ⊆ VFun t

proposition *val-le-refl[simp]*: fixes *v*::val **shows** v ⊆ v ⟨proof⟩

proposition *val-le-trans[trans]*: fixes *v2*::val **shows** [v1 ⊆ v2; v2 ⊆ v3] ⇒ v1 ⊆ v3
⟨proof⟩

lemma *fsubset[intro!]*: fset A ⊆ fset B ⇒ A |⊆| B
⟨proof⟩

proposition *val-le-antisymm*: fixes *v1*::val **shows** [v1 ⊆ v2; v2 ⊆ v1] ⇒ v1 = v2
⟨proof⟩

lemma *le-nat-any[simp]*: VNat n ⊆ v ⇒ v = VNat n
⟨proof⟩

lemma *le-any-nat[simp]*: $v \sqsubseteq \text{VNat } n \implies v = \text{VNat } n$
 ⟨*proof*⟩

lemma *le-nat-nat[simp]*: $\text{VNat } n \sqsubseteq \text{VNat } n' \implies n = n'$
 ⟨*proof*⟩

end

4 Declarative semantics as a relational semantics

theory *RelationalSemFSet*
imports *Lambda ValuesFSet*
begin

inductive *rel-sem* :: $\text{env} \Rightarrow \text{exp} \Rightarrow \text{val} \Rightarrow \text{bool} \ (- \vdash - \Rightarrow - \ [52,52,52] \ 51)$ **where**
rnat[intro!]: $\varrho \vdash \text{ENat } n \Rightarrow \text{VNat } n \mid$
rprim[intro!]: $\llbracket \varrho \vdash e1 \Rightarrow \text{VNat } n1; \varrho \vdash e2 \Rightarrow \text{VNat } n2 \rrbracket \implies \varrho \vdash \text{EPrim } f \ e1 \ e2 \Rightarrow \text{VNat } (f \ n1 \ n2) \mid$
rvar[intro!]: $\llbracket \text{lookup } \varrho \ x = \text{Some } v'; v \sqsubseteq v' \rrbracket \implies \varrho \vdash \text{EVar } x \Rightarrow v \mid$
rlam[intro!]: $\llbracket \forall v \ v'. (v, v') \in \text{fset } t \longrightarrow (x, v) \# \varrho \vdash e \Rightarrow v' \rrbracket$
 $\implies \varrho \vdash \text{ELam } x \ e \Rightarrow \text{VFun } t \mid$
rapp[intro!]: $\llbracket \varrho \vdash e1 \Rightarrow \text{VFun } t; \varrho \vdash e2 \Rightarrow v2; (v3, v3') \in \text{fset } t; v3 \sqsubseteq v2; v \sqsubseteq v3' \rrbracket$
 $\implies \varrho \vdash \text{EApp } e1 \ e2 \Rightarrow v \mid$
rifnz[intro!]: $\llbracket \varrho \vdash e1 \Rightarrow \text{VNat } n; n \neq 0; \varrho \vdash e2 \Rightarrow v \rrbracket \implies \varrho \vdash \text{EIf } e1 \ e2 \ e3 \Rightarrow v \mid$
rifz[intro!]: $\llbracket \varrho \vdash e1 \Rightarrow \text{VNat } n; n = 0; \varrho \vdash e3 \Rightarrow v \rrbracket \implies \varrho \vdash \text{EIf } e1 \ e2 \ e3 \Rightarrow v$

end

theory *DeclSemAsDenotFSet*
imports *Lambda ValuesFSet*
begin

5 Declarative semantics as a denotational semantics

fun *E* :: $\text{exp} \Rightarrow \text{env} \Rightarrow \text{val set}$ **where**
Enat: $E (\text{ENat } n) \ \varrho = \{ v. v = \text{VNat } n \} \mid$
Evar: $E (\text{EVar } x) \ \varrho = \{ v. \exists v'. \text{lookup } \varrho \ x = \text{Some } v' \wedge v \sqsubseteq v' \} \mid$
Elam: $E (\text{ELam } x \ e) \ \varrho = \{ v. \exists f. v = \text{VFun } f \wedge (\forall v1 \ v2. (v1, v2) \in \text{fset } f \longrightarrow v2 \in E \ e \ ((x, v1) \# \varrho)) \} \mid$
Eapp: $E (\text{EApp } e1 \ e2) \ \varrho = \{ v3. \exists f \ v2 \ v2' \ v3'. \text{VFun } f \in E \ e1 \ \varrho \wedge v2 \in E \ e2 \ \varrho \wedge (v2', v3') \in \text{fset } f \wedge v2' \sqsubseteq v2 \wedge v3 \sqsubseteq v3' \} \mid$
Eprim: $E (\text{EPrim } f \ e1 \ e2) \ \varrho = \{ v. \exists n1 \ n2. \text{VNat } n1 \in E \ e1 \ \varrho \wedge \text{VNat } n2 \in E \ e2 \ \varrho \wedge v = \text{VNat } (f \ n1 \ n2) \} \mid$
Eif: $E (\text{EIf } e1 \ e2 \ e3) \ \varrho = \{ v. \exists n. \text{VNat } n \in E \ e1 \ \varrho \wedge (n = 0 \longrightarrow v \in E \ e3 \ \varrho) \wedge (n \neq 0 \longrightarrow v \in E \ e2 \ \varrho) \}$

end

6 Relational and denotational views are equivalent

theory *EquivRelationalDenotFSet*
imports *RelationalSemFSet DeclSemAsDenotFSet*
begin

lemma *denot-implies-rel*: $(v \in E \ e \ \varrho) \implies (\varrho \vdash e \Rightarrow v)$
 ⟨*proof*⟩

lemma *rel-implies-denot*: $\varrho \vdash e \Rightarrow v \implies v \in E \ e \ \varrho$
 ⟨*proof*⟩

theorem *equivalence-relational-denotational*: $(v \in E \ e \ \varrho) = (\varrho \vdash e \Rightarrow v)$

<proof>

end

7 Subsumption and change of environment

theory *ChangeEnv*

imports *Main Lambda DeclSemAsDenotFSet ValuesFSetProps*

begin

lemma *e-prim-intro*[*intro*]: $\llbracket \text{VNat } n1 \in E \ e1 \ \varrho; \text{VNat } n2 \in E \ e2 \ \varrho; v = \text{VNat } (f \ n1 \ n2) \rrbracket$
 $\implies v \in E \ (\text{EPrim } f \ e1 \ e2) \ \varrho$ *<proof>*

lemma *e-prim-elim*[*elim*]: $\llbracket v \in E \ (\text{EPrim } f \ e1 \ e2) \ \varrho;$
 $\bigwedge n1 \ n2. \llbracket \text{VNat } n1 \in E \ e1 \ \varrho; \text{VNat } n2 \in E \ e2 \ \varrho; v = \text{VNat } (f \ n1 \ n2) \rrbracket \implies P \rrbracket \implies P$
<proof>

lemma *e-app-elim*[*elim*]: $\llbracket v3 \in E \ (\text{EApp } e1 \ e2) \ \varrho;$
 $\bigwedge f \ v2 \ v2' \ v3'. \llbracket \text{VFun } f \in E \ e1 \ \varrho; v2 \in E \ e2 \ \varrho; (v2', v3') \in \text{fset } f; v2' \sqsubseteq v2; v3 \sqsubseteq v3' \rrbracket \implies P$
 $\rrbracket \implies P$
<proof>

lemma *e-app-intro*[*intro*]: $\llbracket \text{VFun } f \in E \ e1 \ \varrho; v2 \in E \ e2 \ \varrho; (v2', v3') \in \text{fset } f; v2' \sqsubseteq v2; v3 \sqsubseteq v3' \rrbracket$
 $\implies v3 \in E \ (\text{EApp } e1 \ e2) \ \varrho$ *<proof>*

lemma *e-lam-intro*[*intro*]: $\llbracket v = \text{VFun } f;$
 $\forall v1 \ v2. (v1, v2) \in \text{fset } f \longrightarrow v2 \in E \ e \ ((x, v1) \# \varrho) \rrbracket$
 $\implies v \in E \ (\text{ELam } x \ e) \ \varrho$
<proof>

lemma *e-lam-intro2*[*intro*]:
 $\llbracket \text{VFun } f \in E \ (\text{ELam } x \ e) \ \varrho; v2 \in E \ e \ ((x, v1) \# \varrho) \rrbracket$
 $\implies \text{VFun } (\text{finsert } (v1, v2) \ f) \in E \ (\text{ELam } x \ e) \ \varrho$
<proof>

lemma *e-lam-intro3*[*intro*]: $\text{VFun } \{\|\} \in E \ (\text{ELam } x \ e) \ \varrho$
<proof>

lemma *e-if-intro*[*intro*]: $\llbracket \text{VNat } n \in E \ e1 \ \varrho; n = 0 \longrightarrow v \in E \ e3 \ \varrho; n \neq 0 \longrightarrow v \in E \ e2 \ \varrho \rrbracket$
 $\implies v \in E \ (\text{EIf } e1 \ e2 \ e3) \ \varrho$
<proof>

lemma *e-var-intro*[*elim*]: $\llbracket \text{lookup } \varrho \ x = \text{Some } v'; v \sqsubseteq v' \rrbracket \implies v \in E \ (\text{EVar } x) \ \varrho$
<proof>

lemma *e-var-elim*[*elim*]: $\llbracket v \in E \ (\text{EVar } x) \ \varrho;$
 $\bigwedge v'. \llbracket \text{lookup } \varrho \ x = \text{Some } v'; v \sqsubseteq v' \rrbracket \implies P \rrbracket \implies P$
<proof>

lemma *e-lam-elim*[*elim*]: $\llbracket v \in E \ (\text{ELam } x \ e) \ \varrho;$
 $\bigwedge f. \llbracket v = \text{VFun } f; \forall v1 \ v2. (v1, v2) \in \text{fset } f \longrightarrow v2 \in E \ e \ ((x, v1) \# \varrho) \rrbracket$
 $\implies P \rrbracket \implies P$
<proof>

lemma *e-lam-elim2*[*elim*]: $\llbracket \text{VFun } (\text{finsert } (v1, v2) \ f) \in E \ (\text{ELam } x \ e) \ \varrho;$
 $\llbracket v2 \in E \ e \ ((x, v1) \# \varrho) \rrbracket \implies P \rrbracket \implies P$
<proof>

lemma *e-if-elim*[*elim*]: $\llbracket v \in E \ (\text{EIf } e1 \ e2 \ e3) \ \varrho;$
 $\bigwedge n. \llbracket \text{VNat } n \in E \ e1 \ \varrho; n = 0 \longrightarrow v \in E \ e3 \ \varrho; n \neq 0 \longrightarrow v \in E \ e2 \ \varrho \rrbracket \implies P \rrbracket \implies P$

<proof>

definition *xenv-le* :: name set \Rightarrow env \Rightarrow env \Rightarrow bool (- \vdash - \sqsubseteq - [51,51,51] 52) **where**
X \vdash $\rho \sqsubseteq \rho' \equiv \forall x v. x \in X \wedge \text{lookup } \rho x = \text{Some } v \longrightarrow (\exists v'. \text{lookup } \rho' x = \text{Some } v' \wedge v \sqsubseteq v')$
declare *xenv-le-def*[simp]

proposition *change-env-le*: **fixes** *v::val* **and** *g::env*
assumes *de*: $v \in E e \rho$ **and** *vp-v*: $v' \sqsubseteq v$ **and** *rr*: $FV e \vdash \rho \sqsubseteq \rho'$
shows $v' \in E e \rho'$
<proof>

proposition *e-sub*: $\llbracket v \in E e \rho; v' \sqsubseteq v \rrbracket \Longrightarrow v' \in E e \rho$
<proof>

lemma *env-le-ext*: **fixes** *g::env* **assumes** *rr*: $\rho \sqsubseteq \rho'$ **shows** $((x,v)\#\rho) \sqsubseteq ((x,v)\#\rho')$
<proof>

lemma *change-env*: **fixes** *g::env* **assumes** *de*: $v \in E e \rho$ **and** *rr*: $FV e \vdash \rho \sqsubseteq \rho'$ **shows** $v \in E e \rho'$
<proof>

lemma *raise-env*: **fixes** *g::env* **assumes** *de*: $v \in E e \rho$ **and** *rr*: $\rho \sqsubseteq \rho'$ **shows** $v \in E e \rho'$
<proof>

lemma *env-eq-refl*[simp]: **fixes** *g::env* **shows** $\rho \approx \rho$ *<proof>*

lemma *env-eq-ext*: **fixes** *g::env* **assumes** *rr*: $\rho \approx \rho'$ **shows** $((x,v)\#\rho) \approx ((x,v)\#\rho')$
<proof>

lemma *eq-implies-le*: **fixes** *g::env* **shows** $\rho \approx \rho' \Longrightarrow \rho \sqsubseteq \rho'$
<proof>

lemma *env-swap*: **fixes** *g::env* **assumes** *rr*: $\rho \approx \rho'$ **and** *ve*: $v \in E e \rho$ **shows** $v \in E e \rho'$
<proof>

lemma *env-strengthen*: $\llbracket v \in E e \rho; \forall x. x \in FV e \longrightarrow \text{lookup } \rho' x = \text{lookup } \rho x \rrbracket \Longrightarrow v \in E e \rho'$
<proof>

end

8 Declarative semantics as a non-deterministic interpreter

theory *DeclSemAsNDInterpFSet*
imports *Lambda ValuesFSet*
begin

8.1 Non-determinism monad

type-synonym 'a M = 'a set

definition *set-bind* :: 'a M \Rightarrow ('a \Rightarrow 'b M) \Rightarrow 'b M **where**
set-bind m f $\equiv \{ v. \exists v'. v' \in m \wedge v \in f v' \}$
declare *set-bind-def*[simp]

syntax *-set-bind* :: [pttrns, 'a M, 'b] \Rightarrow 'c ((- \leftarrow -; /-) 0)
translations P \leftarrow E; F \Leftarrow CONST *set-bind* E ($\lambda P. F$)

definition *return* :: 'a \Rightarrow 'a M **where**
return v $\equiv \{v\}$
declare *return-def*[simp]

definition *zero* :: 'a M **where**

$zero \equiv \{\}$
declare *zero-def*[simp]

no-notation *binomial* (**infixl** *choose* 65)

definition *choose* :: 'a set \Rightarrow 'a M **where**
choose S \equiv S
declare *choose-def*[simp]

definition *down* :: val \Rightarrow val M **where**
down v \equiv (v' \leftarrow UNIV; if v' \sqsubseteq v then return v' else zero)
declare *down-def*[simp]

definition *mapM* :: 'a fset \Rightarrow ('a \Rightarrow 'b M) \Rightarrow ('b fset) M **where**
mapM as f \equiv ffold ($\lambda a. \lambda r. (b \leftarrow f a; bs \leftarrow r; return (finsert b bs))$) (return ({})) as

8.2 Non-deterministic interpreter

abbreviation *apply-fun* :: val M \Rightarrow val M \Rightarrow val M **where**

apply-fun V1 V2 \equiv (v1 \leftarrow V1; v2 \leftarrow V2;
 case v1 of VFun f \Rightarrow
 (v2',v3') \leftarrow choose (fset f);
 if v2' \sqsubseteq v2 then return v3' else zero
 | - \Rightarrow zero)

fun *E* :: exp \Rightarrow env \Rightarrow val set **where**

Enat2: $E (ENat n) \varrho = return (VNat n) |$
Evar2: $E (EVar x) \varrho = (case lookup \varrho x of None \Rightarrow zero | Some v \Rightarrow down v) |$
Elam2: $E (ELam x e) \varrho = (vs \leftarrow choose UNIV;$
 $t \leftarrow mapM vs (\lambda v. (v' \leftarrow E e ((x,v)\#\varrho); return (v, v')));$
 $return (VFun t) |$
Eapp2: $E (EApp e1 e2) \varrho = apply-fun (E e1 \varrho) (E e2 \varrho) |$
Eprim2: $E (EPrim f e1 e2) \varrho = (v1 \leftarrow E e1 \varrho; v2 \leftarrow E e2 \varrho;$
 case (v1,v2) of
 (VNat n1, VNat n2) $\Rightarrow return (VNat (f n1 n2))$
 | (VNat n1, VFun t2) $\Rightarrow zero$
 | (VFun t1, v2) $\Rightarrow zero) |$
Eif2[*eta-contract* = *false*]: $E (EIf e1 e2 e3) \varrho = (v1 \leftarrow E e1 \varrho;$
 case v1 of
 (VNat n) $\Rightarrow if n \neq 0 then E e2 \varrho else E e3 \varrho$
 | (VFun t) $\Rightarrow zero)$

end

9 Declarative semantics as a type system

theory *InterTypeSystem*

imports *Lambda*

begin

datatype *ty* = *TNat nat* | *TFun funty*

and *funty* = *TArrow ty ty* (**infix** \rightarrow 55) | *TInt funty funty* (**infix** \sqcap 56) | *TTop* (\top)

inductive *subtype* :: ty \Rightarrow ty \Rightarrow bool (**infix** $<$: 52)

and *fsubtype* :: funty \Rightarrow funty \Rightarrow bool (**infix** $<::$: 52) **where**

sub-refl: $A <: A |$

sub-funty[*intro!*]: $f1 <:: f2 \Longrightarrow TFun f1 <: TFun f2 |$

sub-fun[*intro!*]: $[[T1 <: T1'; T1' <: T1; T2 <: T2'; T2' <: T2]] \Longrightarrow (T1 \rightarrow T2) <:: (T1' \rightarrow T2') |$

sub-inter-l1[*intro!*]: $T1 \sqcap T2 <:: T1 |$

sub-inter-l2[*intro!*]: $T1 \sqcap T2 <:: T2 |$

$sub\text{-}inter\text{-}r[intro!]: \llbracket T3 <:: T1; T3 <:: T2 \rrbracket \Longrightarrow T3 <:: T1 \sqcap T2 \mid$
 $sub\text{-}fun\text{-}top[intro!]: T1 \rightarrow T2 <:: \top \mid$
 $sub\text{-}top\text{-}top[intro!]: \top <:: \top \mid$
 $fsub\text{-}refl[intro!]: T <:: T \mid$
 $sub\text{-}trans[trans]: \llbracket T1 <:: T2; T2 <:: T3 \rrbracket \Longrightarrow T1 <:: T3$

definition $ty\text{-}eq :: ty \Rightarrow ty \Rightarrow bool$ (**infix** ≈ 50) **where**

$A \approx B \equiv A <: B \wedge B <: A$

definition $fty\text{-}eq :: funty \Rightarrow funty \Rightarrow bool$ (**infix** $\simeq 50$) **where**

$F1 \simeq F2 \equiv F1 <:: F2 \wedge F2 <:: F1$

type-synonym $tyenv = (name \times ty)$ list

inductive $wt :: tyenv \Rightarrow exp \Rightarrow ty \Rightarrow bool$ ($- \vdash - : - [51,51,51]$ 51) **where**

$wt\text{-}var[intro!]: lookup \Gamma x = Some T \Longrightarrow \Gamma \vdash EVar x : T \mid$
 $wt\text{-}nat[intro!]: \Gamma \vdash ENat n : TNat n \mid$
 $wt\text{-}lam[intro!]: \llbracket (x,A)\#\Gamma \vdash e : B \rrbracket \Longrightarrow \Gamma \vdash ELam x e : TFunc (A \rightarrow B) \mid$
 $wt\text{-}app[intro!]: \llbracket \Gamma \vdash e1 : TFunc (A \rightarrow B); \Gamma \vdash e2 : A \rrbracket \Longrightarrow \Gamma \vdash EApp e1 e2 : B \mid$
 $wt\text{-}top[intro!]: \Gamma \vdash ELam x e : TFunc \top \mid$
 $wt\text{-}inter[intro!]: \llbracket \Gamma \vdash ELam x e : TFunc A; \Gamma \vdash ELam x e : TFunc B \rrbracket$
 $\Longrightarrow \Gamma \vdash ELam x e : TFunc (A \sqcap B) \mid$
 $wt\text{-}sub[intro!]: \llbracket \Gamma \vdash e : A; A <: B \rrbracket \Longrightarrow \Gamma \vdash e : B \mid$
 $wt\text{-}prim[intro!]: \llbracket \Gamma \vdash e1 : TNat n1; \Gamma \vdash e2 : TNat n2 \rrbracket$
 $\Longrightarrow \Gamma \vdash EPrim f e1 e2 : TNat (f n1 n2) \mid$
 $wt\text{-}ifz[intro!]: \llbracket \Gamma \vdash e1 : TNat 0; \Gamma \vdash e3 : B \rrbracket$
 $\Longrightarrow \Gamma \vdash EIf e1 e2 e3 : B \mid$
 $wt\text{-}ifnz[intro!]: \llbracket \Gamma \vdash e1 : TNat n; n \neq 0; \Gamma \vdash e2 : B \rrbracket$
 $\Longrightarrow \Gamma \vdash EIf e1 e2 e3 : B$

end

10 Declarative semantics with tables as lists

The semantics that represents function tables as lists is largely obsolete, being replaced by the finite set representation. However, the proof of equivalence to the intersection type system still uses the version based on lists.

10.1 Definition of values for declarative semantics

theory *Values*

imports *Main Lambda*

begin

datatype $val = VNat nat \mid VFunc (val \times val)$ list

type-synonym $func = (val \times val)$ list

inductive $val\text{-}le :: val \Rightarrow val \Rightarrow bool$ (**infix** $\sqsubseteq 52$)

and $fun\text{-}le :: func \Rightarrow func \Rightarrow bool$ (**infix** $\lesssim 52$) **where**

$vmat\text{-}le[intro!]: (VNat n) \sqsubseteq (VNat n) \mid$
 $vfun\text{-}le[intro!]: t1 \lesssim t2 \Longrightarrow (VFunc t1) \sqsubseteq (VFunc t2) \mid$
 $fun\text{-}le[intro!]: (\forall v1 v2. (v1,v2) \in set t1 \longrightarrow$
 $(\exists v3 v4. (v3,v4) \in set t2$
 $\wedge v1 \sqsubseteq v3 \wedge v3 \sqsubseteq v1 \wedge v2 \sqsubseteq v4 \wedge v4 \sqsubseteq v2))$
 $\Longrightarrow t1 \lesssim t2$

type-synonym $env = ((name \times val)$ list)

definition $env\text{-}le :: env \Rightarrow env \Rightarrow bool$ (**infix** $\sqsubseteq 52$) **where**

$\varrho \sqsubseteq \varrho' \equiv \forall x v. \text{lookup } \varrho x = \text{Some } v \longrightarrow (\exists v'. \text{lookup } \varrho' x = \text{Some } v' \wedge v \sqsubseteq v')$

definition *env-eq* :: *env* \Rightarrow *env* \Rightarrow *bool* (**infix** ≈ 50) **where**
 $\varrho \approx \varrho' \equiv (\forall x. \text{lookup } \varrho x = \text{lookup } \varrho' x)$

end

10.2 Properties about values

theory *ValueProps*
imports *Values*
begin

inductive-cases *fun-le-inv[elim]*: $t1 \lesssim t2$ **and**
vfun-le-inv[elim!]: $VFun\ t1 \sqsubseteq VFun\ t2$ **and**
le-fun-nat-inv[elim!]: $VFun\ t2 \sqsubseteq VNat\ x1$ **and**
le-fun-cons-inv[elim!]: $(v1, v2) \# t1 \lesssim t2$ **and**
le-any-nat-inv[elim!]: $v \sqsubseteq VNat\ n$ **and**
le-nat-any-inv[elim!]: $VNat\ n \sqsubseteq v$ **and**
le-fun-any-inv[elim!]: $VFun\ t \sqsubseteq v$ **and**
le-any-fun-inv[elim!]: $v \sqsubseteq VFun\ t$

lemma *fun-le-cons*: $(a \# t1) \lesssim t2 \implies t1 \lesssim t2$
 $\langle \text{proof} \rangle$

function *val-size* :: *val* \Rightarrow *nat* **and** *fun-size* :: *func* \Rightarrow *nat* **where**
val-size ($VNat\ n$) = 0 |
val-size ($VFun\ t$) = 1 + *fun-size* t |
fun-size [] = 0 |
fun-size $((v1, v2) \# t)$ = 1 + *val-size* $v1$ + *val-size* $v2$ + *fun-size* t
 $\langle \text{proof} \rangle$

termination *val-size* $\langle \text{proof} \rangle$

lemma *val-size-mem*: $(a, b) \in \text{set } t \implies \text{val-size } a + \text{val-size } b < \text{fun-size } t$
 $\langle \text{proof} \rangle$

lemma *val-size-mem-l*: $(a, b) \in \text{set } t \implies \text{val-size } a < \text{fun-size } t$
 $\langle \text{proof} \rangle$

lemma *val-size-mem-r*: $(a, b) \in \text{set } t \implies \text{val-size } b < \text{fun-size } t$
 $\langle \text{proof} \rangle$

lemma *val-fun-le-refl*: $\forall v t. n = \text{val-size } v + \text{fun-size } t \longrightarrow v \sqsubseteq v \wedge t \lesssim t$
 $\langle \text{proof} \rangle$

proposition *val-le-refl[simp]*: **fixes** $v::\text{val}$ **shows** $v \sqsubseteq v$ $\langle \text{proof} \rangle$

lemma *fun-le-refl[simp]*: **fixes** $t::\text{func}$ **shows** $t \lesssim t$ $\langle \text{proof} \rangle$

definition *val-eq* :: *val* \Rightarrow *val* \Rightarrow *bool* (**infix** ~ 52) **where**
 $\text{val-eq } v1\ v2 \equiv (v1 \sqsubseteq v2 \wedge v2 \sqsubseteq v1)$

definition *fun-eq* :: *func* \Rightarrow *func* \Rightarrow *bool* (**infix** ~ 52) **where**
 $\text{fun-eq } t1\ t2 \equiv (t1 \lesssim t2 \wedge t2 \lesssim t1)$

lemma *vfun-eq[intro!]*: $t \sim t' \implies VFun\ t \sim VFun\ t'$
 $\langle \text{proof} \rangle$

lemma *val-eq-refl[simp]*: **fixes** $v::\text{val}$ **shows** $v \sim v$
 $\langle \text{proof} \rangle$

lemma *val-eq-symm*: **fixes** $v1::\text{val}$ **and** $v2::\text{val}$ **shows** $v1 \sim v2 \implies v2 \sim v1$
 $\langle \text{proof} \rangle$

lemma *val-le-fun-le-trans*:

$\forall v2\ t2.\ n = \text{val-size } v2 + \text{fun-size } t2 \longrightarrow$
 $(\forall v1\ v3.\ v1 \sqsubseteq v2 \longrightarrow v2 \sqsubseteq v3 \longrightarrow v1 \sqsubseteq v3)$
 $\wedge (\forall t1\ t3.\ t1 \lesssim t2 \longrightarrow t2 \lesssim t3 \longrightarrow t1 \lesssim t3)$
 $\langle \text{proof} \rangle$

proposition *val-le-trans*: **fixes** $v2::\text{val}$ **shows** $\llbracket v1 \sqsubseteq v2; v2 \sqsubseteq v3 \rrbracket \Longrightarrow v1 \sqsubseteq v3$
 $\langle \text{proof} \rangle$

lemma *fun-le-trans*: $\llbracket t1 \lesssim t2; t2 \lesssim t3 \rrbracket \Longrightarrow t1 \lesssim t3$
 $\langle \text{proof} \rangle$

lemma *val-eq-trans*: **fixes** $v1::\text{val}$ **and** $v2::\text{val}$ **and** $v3::\text{val}$
assumes $v12: v1 \sim v2$ **and** $v23: v2 \sim v3$ **shows** $v1 \sim v3$
 $\langle \text{proof} \rangle$

lemma *fun-eq-refl*[*simp*]: **fixes** $t::\text{func}$ **shows** $t \sim t$
 $\langle \text{proof} \rangle$

lemma *fun-eq-trans*: **fixes** $t1::\text{func}$ **and** $t2::\text{func}$ **and** $t3::\text{func}$
assumes $t12: t1 \sim t2$ **and** $t23: t2 \sim t3$ **shows** $t1 \sim t3$
 $\langle \text{proof} \rangle$

lemma *append-fun-le*:
 $\llbracket t1' \lesssim t1; t2' \lesssim t2 \rrbracket \Longrightarrow t1' @ t2' \lesssim t1 @ t2$
 $\langle \text{proof} \rangle$

lemma *append-fun-equiv*:
 $\llbracket t1' \sim t1; t2' \sim t2 \rrbracket \Longrightarrow t1' @ t2' \sim t1 @ t2$
 $\langle \text{proof} \rangle$

lemma *append-leq-symm*: $t2 @ t1 \lesssim t1 @ t2$
 $\langle \text{proof} \rangle$

lemma *append-eq-symm*: $t2 @ t1 \sim t1 @ t2$
 $\langle \text{proof} \rangle$

lemma *le-nat-any*[*simp*]: $VNat\ n \sqsubseteq v \Longrightarrow v = VNat\ n$
 $\langle \text{proof} \rangle$

lemma *le-any-nat*[*simp*]: $v \sqsubseteq VNat\ n \Longrightarrow v = VNat\ n$
 $\langle \text{proof} \rangle$

lemma *le-nat-nat*[*simp*]: $VNat\ n \sqsubseteq VNat\ n' \Longrightarrow n = n'$
 $\langle \text{proof} \rangle$

end

10.3 Declarative semantics as a denotational semantics

theory *DeclSemAsDenot*

imports *Lambda Values*

begin

fun $E :: \text{exp} \Rightarrow \text{env} \Rightarrow \text{val set}$ **where**

$Enat: E\ (ENat\ n)\ \varrho = \{ v.\ v = VNat\ n \} \mid$

$Evar: E\ (EVar\ x)\ \varrho = \{ v.\ \exists v'. \text{lookup } \varrho\ x = \text{Some } v' \wedge v \sqsubseteq v' \} \mid$

$Elam: E\ (ELam\ x\ e)\ \varrho = \{ v.\ \exists f.\ v = VFun\ f \wedge (\forall v1\ v2.\ (v1, v2) \in \text{set } f$
 $\longrightarrow v2 \in E\ e\ ((x, v1)\#\varrho)) \} \mid$

$Eapp: E\ (EApp\ e1\ e2)\ \varrho = \{ v3.\ \exists f\ v2\ v2'\ v3'. \}$

$$\begin{aligned}
& \text{VFun } f \in E \ e1 \ \varrho \wedge v2 \in E \ e2 \ \varrho \wedge (v2', v3') \in \text{set } f \wedge v2' \sqsubseteq v2 \wedge v3 \sqsubseteq v3' \} | \\
\text{Eprim: } & E \ (E\text{Prim } f \ e1 \ e2) \ \varrho = \{ v. \exists \ n1 \ n2. \text{VNat } n1 \in E \ e1 \ \varrho \\
& \wedge \text{VNat } n2 \in E \ e2 \ \varrho \wedge v = \text{VNat } (f \ n1 \ n2) \} | \\
\text{Eif: } & E \ (E\text{If } e1 \ e2 \ e3) \ \varrho = \{ v. \exists \ n. \text{VNat } n \in E \ e1 \ \varrho \\
& \wedge (n = 0 \longrightarrow v \in E \ e3 \ \varrho) \wedge (n \neq 0 \longrightarrow v \in E \ e2 \ \varrho) \}
\end{aligned}$$

end

10.4 Subsumption and change of environment

theory *DenotLam5*

imports *Main Lambda DeclSemAsDenot ValueProps*

begin

lemma *e-prim-intro*[intro]: $\llbracket \text{VNat } n1 \in E \ e1 \ \varrho; \text{VNat } n2 \in E \ e2 \ \varrho; v = \text{VNat } (f \ n1 \ n2) \rrbracket$
 $\implies v \in E \ (E\text{Prim } f \ e1 \ e2) \ \varrho$ *<proof>*

lemma *e-prim-elim*[elim]: $\llbracket v \in E \ (E\text{Prim } f \ e1 \ e2) \ \varrho;$
 $\wedge \ n1 \ n2. \llbracket \text{VNat } n1 \in E \ e1 \ \varrho; \text{VNat } n2 \in E \ e2 \ \varrho; v = \text{VNat } (f \ n1 \ n2) \rrbracket \implies P \rrbracket \implies P$
<proof>

lemma *e-app-elim*[elim]: $\llbracket v3 \in E \ (E\text{App } e1 \ e2) \ \varrho;$
 $\wedge \ f \ v2 \ v2' \ v3'. \llbracket \text{VFun } f \in E \ e1 \ \varrho; v2 \in E \ e2 \ \varrho; (v2', v3') \in \text{set } f; v2' \sqsubseteq v2; v3 \sqsubseteq v3' \rrbracket \implies P$
 $\rrbracket \implies P$
<proof>

lemma *e-app-intro*[intro]: $\llbracket \text{VFun } f \in E \ e1 \ \varrho; v2 \in E \ e2 \ \varrho; (v2', v3') \in \text{set } f; v2' \sqsubseteq v2; v3 \sqsubseteq v3' \rrbracket$
 $\implies v3 \in E \ (E\text{App } e1 \ e2) \ \varrho$ *<proof>*

lemma *e-lam-intro*[intro]: $\llbracket v = \text{VFun } f;$
 $\forall \ v1 \ v2. (v1, v2) \in \text{set } f \longrightarrow v2 \in E \ e \ ((x, v1) \# \varrho) \rrbracket$
 $\implies v \in E \ (E\text{Lam } x \ e) \ \varrho$
<proof>

lemma *e-lam-intro2*[intro]:
 $\llbracket \text{VFun } f \in E \ (E\text{Lam } x \ e) \ \varrho; v2 \in E \ e \ ((x, v1) \# \varrho) \rrbracket$
 $\implies \text{VFun } ((v1, v2) \# f) \in E \ (E\text{Lam } x \ e) \ \varrho$
<proof>

lemma *e-lam-intro3*[intro]: $\llbracket \text{VFun } _ \in E \ (E\text{Lam } x \ e) \ \varrho$
<proof>

lemma *e-if-intro*[intro]: $\llbracket \text{VNat } n \in E \ e1 \ \varrho; n = 0 \longrightarrow v \in E \ e3 \ \varrho; n \neq 0 \longrightarrow v \in E \ e2 \ \varrho \rrbracket$
 $\implies v \in E \ (E\text{If } e1 \ e2 \ e3) \ \varrho$
<proof>

lemma *e-var-intro*[elim]: $\llbracket \text{lookup } \varrho \ x = \text{Some } v'; v \sqsubseteq v' \rrbracket \implies v \in E \ (E\text{Var } x) \ \varrho$
<proof>

lemma *e-var-elim*[elim]: $\llbracket v \in E \ (E\text{Var } x) \ \varrho;$
 $\wedge \ v'. \llbracket \text{lookup } \varrho \ x = \text{Some } v'; v \sqsubseteq v' \rrbracket \implies P \rrbracket \implies P$
<proof>

lemma *e-lam-elim*[elim]: $\llbracket v \in E \ (E\text{Lam } x \ e) \ \varrho;$
 $\wedge \ f. \llbracket v = \text{VFun } f; \forall \ v1 \ v2. (v1, v2) \in \text{set } f \longrightarrow v2 \in E \ e \ ((x, v1) \# \varrho) \rrbracket$
 $\implies P \rrbracket \implies P$
<proof>

lemma *e-lam-elim2*[elim]: $\llbracket \text{VFun } ((v1, v2) \# f) \in E \ (E\text{Lam } x \ e) \ \varrho;$
 $\llbracket v2 \in E \ e \ ((x, v1) \# \varrho) \rrbracket \implies P \rrbracket \implies P$
<proof>

lemma *e-if-elim*[*elim*]: $\llbracket v \in E (EIf\ e1\ e2\ e3)\ \varrho; \bigwedge n. \llbracket VNat\ n \in E\ e1\ \varrho; n = 0 \longrightarrow v \in E\ e3\ \varrho; n \neq 0 \longrightarrow v \in E\ e2\ \varrho \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
 <proof>

definition *xenv-le* :: *name set* \Rightarrow *env* \Rightarrow *env* \Rightarrow *bool* ($- \vdash - \sqsubseteq -$ [51,51,51] 52) **where**
 $X \vdash \varrho \sqsubseteq \varrho' \equiv \forall x v. x \in X \wedge lookup\ \varrho\ x = Some\ v \longrightarrow (\exists v'. lookup\ \varrho'\ x = Some\ v' \wedge v \sqsubseteq v')$
declare *xenv-le-def*[*simp*]

proposition *change-env-le*: **fixes** *v::val* **and** *ϱ::env*
assumes *de*: $v \in E\ e\ \varrho$ **and** *vp-v*: $v' \sqsubseteq v$ **and** *rr*: $FV\ e \vdash \varrho \sqsubseteq \varrho'$
shows $v' \in E\ e\ \varrho'$
 <proof>

proposition *e-sub*: $\llbracket v \in E\ e\ \varrho; v' \sqsubseteq v \rrbracket \Longrightarrow v' \in E\ e\ \varrho$
 <proof>

lemma *env-le-ext*: **fixes** *ϱ::env* **assumes** *rr*: $\varrho \sqsubseteq \varrho'$ **shows** $((x,v)\#\varrho) \sqsubseteq ((x,v)\#\varrho')$
 <proof>

lemma *change-env*: **fixes** *ϱ::env* **assumes** *de*: $v \in E\ e\ \varrho$ **and** *rr*: $FV\ e \vdash \varrho \sqsubseteq \varrho'$ **shows** $v \in E\ e\ \varrho'$
 <proof>

lemma *raise-env*: **fixes** *ϱ::env* **assumes** *de*: $v \in E\ e\ \varrho$ **and** *rr*: $\varrho \sqsubseteq \varrho'$ **shows** $v \in E\ e\ \varrho'$
 <proof>

lemma *env-eq-refl*[*simp*]: **fixes** *ϱ::env* **shows** $\varrho \approx \varrho$ <proof>

lemma *env-eq-ext*: **fixes** *ϱ::env* **assumes** *rr*: $\varrho \approx \varrho'$ **shows** $((x,v)\#\varrho) \approx ((x,v)\#\varrho')$
 <proof>

lemma *eq-implies-le*: **fixes** *ϱ::env* **shows** $\varrho \approx \varrho' \Longrightarrow \varrho \sqsubseteq \varrho'$
 <proof>

lemma *env-swap*: **fixes** *ϱ::env* **assumes** *rr*: $\varrho \approx \varrho'$ **and** *ve*: $v \in E\ e\ \varrho$ **shows** $v \in E\ e\ \varrho'$
 <proof>

lemma *env-strengthen*: $\llbracket v \in E\ e\ \varrho; \forall x. x \in FV\ e \longrightarrow lookup\ \varrho'\ x = lookup\ \varrho\ x \rrbracket \Longrightarrow v \in E\ e\ \varrho'$
 <proof>

end

11 Equivalence of denotational and type system views

theory *EquivDenotInterTypes*

imports *InterTypeSystem DeclSemAsDenot DenotLam5*

begin

fun *V* :: *ty* \Rightarrow *val* **and** *Vf* :: *funty* \Rightarrow (*val* \times *val*) *list* **where**
 $V\ (TNat\ n) = VNat\ n \mid$
 $V\ (TFun\ f) = VFun\ (Vf\ f) \mid$
 $Vf\ (A \rightarrow B) = [(V\ A, V\ B)] \mid$
 $Vf\ (A \sqcap B) = Vf\ A\ @\ Vf\ B \mid$
 $Vf\ \top = []$

fun *Venv* :: *tyenv* \Rightarrow *env* **where**
 $Venv\ [] = [] \mid$
 $Venv\ ((x,A)\#\Gamma) = (x, V\ A)\#Venv\ \Gamma$

function *T* :: *val* \Rightarrow *ty* **and** *Tf* :: (*val* \times *val*) *list* \Rightarrow *funty* **where**
 $T\ (VNat\ n) = TNat\ n \mid$

$T (VFun\ t) = TFun\ (Tf\ t) \mid$
 $Tf\ [] = \top \mid$
 $Tf\ ((v1, v2)\#t) = (T\ v1 \rightarrow T\ v2) \sqcap Tf\ t$
 $\langle proof \rangle$
termination $T\ \langle proof \rangle$

fun $Tenv :: env \Rightarrow tyenv$ **where**
 $Tenv\ [] = [] \mid$
 $Tenv\ ((x, v)\#\rho) = (x, T\ v)\#Tenv\ \rho$

lemma *sub-inter-left1*: $A <:: C \implies A \sqcap B <:: C$
 $\langle proof \rangle$

lemma *sub-inter-left2*: $B <:: C \implies A \sqcap B <:: C$
 $\langle proof \rangle$

lemma *vf-nil[simp]*: $Vf\ (Tf\ []) = [] \langle proof \rangle$

lemma *vf-cons[simp]*: $Vf\ (Tf\ ((v, v')\#t)) = (V\ (T\ v), V\ (T\ v'))\#(Vf\ (Tf\ t)) \langle proof \rangle$

proposition *vt-id*: **shows** $V\ (T\ v) = v$ **and** $Vf\ (Tf\ t) = t$
 $\langle proof \rangle$

lemma *lookup-tenv*:
 $lookup\ \rho\ x = Some\ v \implies lookup\ (Tenv\ \rho)\ x = Some\ (T\ v)$
 $\langle proof \rangle$

proposition *table-mem-sub*:
 $(v, v') \in set\ t \implies Tf\ t <:: (T\ v) \rightarrow (T\ v')$
 $\langle proof \rangle$

lemma *Tf-top*: $Tf\ t <:: \top$
 $\langle proof \rangle$

lemma *le-sub-flip-aux*:
 $\forall v\ v'\ t\ t'. n = val\text{-size}\ v + val\text{-size}\ v' + fun\text{-size}\ t + fun\text{-size}\ t' \longrightarrow$
 $(v \sqsubseteq v' \longrightarrow T\ v' <: T\ v) \wedge (t \lesssim t' \longrightarrow Tf\ t' <:: Tf\ t)$
 $\langle proof \rangle$

proposition *le-sub-flip*: $v \sqsubseteq v' \implies T\ v' <: T\ v \langle proof \rangle$

lemma *le-sub-fun-flip*: $t \lesssim t' \implies Tf\ t' <:: Tf\ t \langle proof \rangle$

lemma *Tf-append*: $Tf\ (t1\ @\ t2) <:: Tf\ t1 \sqcap Tf\ t2$
 $\langle proof \rangle$

lemma *append-Tf*: $Tf\ t1 \sqcap Tf\ t2 <:: Tf\ (t1\ @\ t2)$
 $\langle proof \rangle$

proposition *tv-id*: **shows** $T\ (V\ A) \approx A$ **and** $Tf\ (Vf\ F) \simeq F$
 $\langle proof \rangle$

lemma *denot-lam-implies-ts*:
assumes $et: \forall v\ \rho. v \in E\ e\ \rho \longrightarrow Tenv\ \rho \vdash e : T\ v$ **and**
 $fe: \forall v1\ v2. (v1, v2) \in set\ f \longrightarrow v2 \in E\ e\ ((x, v1)\#\rho)$
shows $Tenv\ \rho \vdash ELam\ x\ e : TFun\ (Tf\ f)$
 $\langle proof \rangle$

theorem *denot-implies-ts*:
assumes $ve: v \in E\ e\ \rho$ **shows** $Tenv\ \rho \vdash e : T\ v$
 $\langle proof \rangle$

lemma *venv-lookup*: **assumes** lx : $lookup\ \Gamma\ x = Some\ A$ **shows** $lookup\ (Venv\ \Gamma)\ x = Some\ (V\ A)$
 ⟨*proof*⟩

lemma *append-fun-equiv*: $\llbracket t1' \sim t1; t2' \sim t2 \rrbracket \implies t1' @ t2' \sim t1 @ t2$
 ⟨*proof*⟩

lemma *append-eq-symm*: $t2 @ t1 \sim t1 @ t2$
 ⟨*proof*⟩

lemma *sub-le-flip*: $(A <: B \longrightarrow V\ B \sqsubseteq V\ A) \wedge (f1 <:: f2 \longrightarrow (Vf\ f2) \lesssim (Vf\ f1))$
 ⟨*proof*⟩

theorem *ts-implies-denot*:
assumes wte : $\Gamma \vdash e : A$ **shows** $V\ A \in E\ e\ (Venv\ \Gamma)$
 ⟨*proof*⟩

end

12 Soundness of the declarative semantics wrt. operational

theory *DenotSoundFSet*
imports *SmallStepLam BigStepLam ChangeEnv*
begin

12.1 Substitution preserves denotation

lemma *subst-app*: $subst\ x\ v\ (EApp\ e1\ e2) = EApp\ (subst\ x\ v\ e1)\ (subst\ x\ v\ e2)$
 ⟨*proof*⟩

lemma *subst-prim*: $subst\ x\ v\ (EPrim\ f\ e1\ e2) = EPrim\ f\ (subst\ x\ v\ e1)\ (subst\ x\ v\ e2)$
 ⟨*proof*⟩

lemma *subst-lam-eq*: $subst\ x\ v\ (ELam\ x\ e) = ELam\ x\ e$ ⟨*proof*⟩

lemma *subst-lam-neq*: $y \neq x \implies subst\ x\ v\ (ELam\ y\ e) = ELam\ y\ (subst\ x\ v\ e)$ ⟨*proof*⟩

lemma *subst-if*: $subst\ x\ v\ (EIf\ e1\ e2\ e3) = EIf\ (subst\ x\ v\ e1)\ (subst\ x\ v\ e2)\ (subst\ x\ v\ e3)$
 ⟨*proof*⟩

lemma *substitution*:
fixes $\Gamma::env$ **and** $A::val$
assumes wte : $B \in E\ e\ \Gamma'$ **and** wtv : $A \in E\ v\ []$
and gp : $\Gamma' \approx (x,A)\#\Gamma$ **and** v : *is-val* v
shows $B \in E\ (subst\ x\ v\ e)\ \Gamma$
 ⟨*proof*⟩

12.2 Reduction preserves denotation

lemma *subject-reduction*: **fixes** $e::exp$ **assumes** v : $v \in E\ e\ \rho$ **and** r : $e \longrightarrow e'$ **shows** $v \in E\ e'\ \rho$
 ⟨*proof*⟩

theorem *preservation*: **assumes** v : $v \in E\ e\ \rho$ **and** rr : $e \longrightarrow^* e'$ **shows** $v \in E\ e'\ \rho$
 ⟨*proof*⟩

lemma *canonical-nat*: **assumes** v : $VNat\ n \in E\ v\ \rho$ **and** vv : *isval* v **shows** $v = ENat\ n$
 ⟨*proof*⟩

lemma *canonical-fun*: **assumes** v : $VFun\ f \in E\ v\ \rho$ **and** vv : *isval* v **shows** $\exists\ x\ e. v = ELam\ x\ e$
 ⟨*proof*⟩

12.3 Progress

theorem progress: **assumes** $v \in E \ e \ \varrho$ **and** $r: \varrho = []$ **and** $fve: FV \ e = \{\}$
shows $is\text{-}val \ e \vee (\exists \ e'. \ e \longrightarrow e')$
 $\langle proof \rangle$

12.4 Logical relation between values and big-step values

fun good-entry $:: name \Rightarrow exp \Rightarrow benv \Rightarrow (val \times bval \ set) \times (val \times bval \ set) \Rightarrow bool \Rightarrow bool$ **where**
 $good\text{-}entry \ x \ e \ \varrho \ ((v1, g1), (v2, g2)) \ r = ((\forall \ v \in g1. \exists \ v'. (x, v)\#_{\varrho} \vdash e \Downarrow v' \wedge v' \in g2) \wedge r)$

primrec good $:: val \Rightarrow bval \ set$ **where**

$Gnat: good \ (VNat \ n) = \{ BNat \ n \} \mid$
 $Gfun: good \ (VFun \ f) = \{ vc. \exists \ x \ e \ \varrho. vc = BClos \ x \ e \ \varrho$
 $\wedge (\text{ffold} \ (good\text{-}entry \ x \ e \ \varrho) \ True \ (fimage \ (map\text{-}prod \ (\lambda v. (v, good \ v)) \ (\lambda v. (v, good \ v)))) \ f) \}$

inductive good-env $:: benv \Rightarrow env \Rightarrow bool$ **where**

$genv\text{-}nil[intr0!]: good\text{-}env \ [] \ [] \mid$
 $genv\text{-}cons[intr0!]: [\![\ v \in good \ v'; \ good\text{-}env \ \varrho \ \varrho' \]\!] \Longrightarrow good\text{-}env \ ((x, v)\#_{\varrho}) \ ((x, v')\#_{\varrho'})$

inductive-cases

$genv\text{-}any\text{-}nil\text{-}inv: good\text{-}env \ \varrho \ []$ **and**
 $genv\text{-}any\text{-}cons\text{-}inv: good\text{-}env \ \varrho \ (b\#_{\varrho'})$

lemma lookup-good:

assumes $l: lookup \ \varrho' \ x = Some \ A$ **and** $EE: good\text{-}env \ \varrho \ \varrho'$
shows $\exists \ v. lookup \ \varrho \ x = Some \ v \wedge v \in good \ A$
 $\langle proof \rangle$

abbreviation good-prod $:: val \times val \Rightarrow (val \times bval \ set) \times (val \times bval \ set)$ **where**
 $good\text{-}prod \equiv map\text{-}prod \ (\lambda v. (v, good \ v)) \ (\lambda v. (v, good \ v))$

lemma good-prod-inj: $inj\text{-}on \ good\text{-}prod \ (fset \ A)$
 $\langle proof \rangle$

definition good-fun $:: func \Rightarrow name \Rightarrow exp \Rightarrow benv \Rightarrow bool$ **where**

$good\text{-}fun \ f \ x \ e \ \varrho \equiv (\text{ffold} \ (good\text{-}entry \ x \ e \ \varrho) \ True \ (fimage \ good\text{-}prod \ f))$

lemma good-fun-def2:

$good\text{-}fun \ f \ x \ e \ \varrho = \text{ffold} \ (good\text{-}entry \ x \ e \ \varrho \circ good\text{-}prod) \ True \ f$
 $\langle proof \rangle$

lemma gfun-elim: $w \in good \ (VFun \ f) \Longrightarrow \exists \ x \ e \ \varrho. w = BClos \ x \ e \ \varrho \wedge good\text{-}fun \ f \ x \ e \ \varrho$
 $\langle proof \rangle$

lemma gfun-mem-iff: $good\text{-}fun \ f \ x \ e \ \varrho = (\forall \ v1 \ v2. (v1, v2) \in fset \ f \longrightarrow$
 $(\forall \ v \in good \ v1. \exists \ v'. (x, v)\#_{\varrho} \vdash e \Downarrow v' \wedge v' \in good \ v2))$
 $\langle proof \rangle$

lemma gfun-mem: $[(v1, v2) \in fset \ f; good\text{-}fun \ f \ x \ e \ \varrho]$
 $\Longrightarrow \forall \ v \in good \ v1. \exists \ v'. (x, v)\#_{\varrho} \vdash e \Downarrow v' \wedge v' \in good \ v2$
 $\langle proof \rangle$

lemma gfun-intro: $(\forall \ v1 \ v2. (v1, v2) \in fset \ f \longrightarrow (\forall \ v \in good \ v1. \exists \ v'. (x, v)\#_{\varrho} \vdash e \Downarrow v' \wedge v' \in good \ v2))$
 $\Longrightarrow good\text{-}fun \ f \ x \ e \ \varrho$ $\langle proof \rangle$

lemma sub-good: **fixes** $v::val$ **assumes** $wv: w \in good \ v$ **and** $vp\text{-}v: v' \sqsubseteq v$ **shows** $w \in good \ v'$
 $\langle proof \rangle$

12.5 Denotational semantics sound wrt. big-step

lemma *denot-terminates*: **assumes** *vp-e*: $v' \in E\ e\ \varrho'$ **and** *ge*: *good-env* $\varrho\ \varrho'$
shows $\exists v. \varrho \vdash e \Downarrow v \wedge v \in \text{good } v'$
<proof>

theorem *sound-wrt-op-sem*:

assumes *E-e-n*: $E\ e\ [] = E\ (ENat\ n)\ []$ **and** *fv-e*: $FV\ e = \{\}$ **shows** $e \Downarrow ONat\ n$
<proof>

end

13 Completeness of the declarative semantics wrt. operational

theory *DenotCompleteFSet*

imports *ChangeEnv SmallStepLam DenotSoundFSet*
begin

13.1 Reverse substitution preserves denotation

fun *join* :: *val* \Rightarrow *val* \Rightarrow *val option* (**infix** \sqcup 60) **where**
 $(VNat\ n) \sqcup (VNat\ n') = (\text{if } n = n' \text{ then } Some\ (VNat\ n) \text{ else } None) \mid$
 $(VFun\ f) \sqcup (VFun\ f') = Some\ (VFun\ (f \mid \cup \mid f')) \mid$
 $v \sqcup v' = None$

lemma *combine-values*:

assumes *vv*: *isval* v **and** *v1v*: $v1 \in E\ v\ \varrho$ **and** *v2v*: $v2 \in E\ v\ \varrho$
shows $\exists v3. v3 \in E\ v\ \varrho \wedge (v1 \sqcup v2 = Some\ v3)$
<proof>

lemma *le-union1*: **fixes** *v1::val* **assumes** *v12*: $v1 \sqcup v2 = Some\ v12$ **shows** $v1 \sqsubseteq v12$
<proof>

lemma *le-union2*: $v1 \sqcup v2 = Some\ v12 \Longrightarrow v2 \sqsubseteq v12$
<proof>

lemma *le-union-left*: $\llbracket v1 \sqcup v2 = Some\ v12; v1 \sqsubseteq v3; v2 \sqsubseteq v3 \rrbracket \Longrightarrow v12 \sqsubseteq v3$
<proof>

lemma *e-val*: *isval* $v \Longrightarrow \exists v'. v' \in E\ v\ \varrho$
<proof>

lemma *reverse-subst-lam*:

assumes *fl*: $VFun\ f \in E\ (ELam\ x\ e)\ \varrho$
and *vv*: *is-val* v **and** *ls*: $ELam\ x\ e = ELam\ x\ (subst\ y\ v\ e')$ **and** *xy*: $x \neq y$
and *IH*: $\forall v1\ v2. v2 \in E\ (subst\ y\ v\ e')\ ((x, v1) \# \varrho)$
 $\longrightarrow (\exists \varrho'\ v'. v' \in E\ v\ [] \wedge v2 \in E\ e'\ \varrho' \wedge \varrho' \approx (y, v') \# ((x, v1) \# \varrho))$
shows $\exists \varrho'\ v''. v'' \in E\ v\ [] \wedge VFun\ f \in E\ (ELam\ x\ e')\ \varrho' \wedge \varrho' \approx ((y, v'') \# \varrho)$
<proof>

lemma *lookup-ext-none*: $\llbracket lookup\ \varrho\ y = None; x \neq y \rrbracket \Longrightarrow lookup\ ((x, v) \# \varrho)\ y = None$
<proof>

lemma *rev-subst-var*:

assumes *ev*: $e = EVar\ y\ v \wedge v = e'$ **and** *vv*: *is-val* v **and** *vp-E*: $v' \in E\ e'\ \varrho$
shows $\exists \varrho'\ v''. v'' \in E\ v\ [] \wedge v' \in E\ e\ \varrho' \wedge \varrho' \approx ((y, v'') \# \varrho)$
<proof>

lemma *reverse-subst-pres-denot*:

assumes *vep*: $v' \in E\ e'\ \varrho$ **and** *vv*: *is-val* v **and** *ep*: $e' = subst\ y\ v\ e$
shows $\exists \varrho'\ v''. v'' \in E\ v\ [] \wedge v' \in E\ e\ \varrho' \wedge \varrho' \approx ((y, v'') \# \varrho)$
<proof>

13.2 Reverse reduction preserves denotation

lemma *reverse-step-pres-denot*:

fixes $e::exp$ **assumes** $e-ep: e \longrightarrow e'$ **and** $v-ep: v \in E e' \varrho$
shows $v \in E e \varrho$
 $\langle proof \rangle$

lemma *reverse-multi-step-pres-denot*:

fixes $e::exp$ **assumes** $e-ep: e \longrightarrow^* e'$ **and** $v-ep: v \in E e' \varrho$ **shows** $v \in E e \varrho$
 $\langle proof \rangle$

13.3 Completeness

theorem *completeness*:

assumes $ev: e \longrightarrow^* v$ **and** $vv: is-val\ v$
shows $\exists v'. v' \in E e \varrho \wedge v' \in E v \square$
 $\langle proof \rangle$

theorem *reduce-pres-denot*: **fixes** $e::exp$ **assumes** $r: e \longrightarrow e'$ **shows** $E e = E e'$
 $\langle proof \rangle$

theorem *multi-reduce-pres-denot*: **fixes** $e::exp$ **assumes** $r: e \longrightarrow^* e'$ **shows** $E e = E e'$
 $\langle proof \rangle$

theorem *complete-wrt-op-sem*:

assumes $e-n: e \Downarrow ONat\ n$ **shows** $E e \square = E (ENat\ n) \square$
 $\langle proof \rangle$

end

14 Soundness wrt. contextual equivalence

14.1 Denotational semantics is a congruence

theory *DenotCongruenceFSet*

imports *ChangeEnv DenotSoundFSet DenotCompleteFSet*

begin

lemma *e-lam-cong*[*cong*]: $E e = E e' \Longrightarrow E (ELam\ x\ e) = E (ELam\ x\ e')$
 $\langle proof \rangle$

lemma *e-app-cong*[*cong*]: $\llbracket E e1 = E e1'; E e2 = E e2' \rrbracket \Longrightarrow E (EApp\ e1\ e2) = E (EApp\ e1'\ e2')$
 $\langle proof \rangle$

lemma *e-prim-cong*[*cong*]: $\llbracket E e1 = E e1'; E e2 = E e2' \rrbracket \Longrightarrow E (EPrim\ f\ e1\ e2) = E (EPrim\ f\ e1'\ e2')$
 $\langle proof \rangle$

lemma *e-if-cong*[*cong*]: $\llbracket E e1 = E e1'; E e2 = E e2'; E e3 = E e3' \rrbracket$
 $\Longrightarrow E (EIf\ e1\ e2\ e3) = E (EIf\ e1'\ e2'\ e3')$
 $\langle proof \rangle$

datatype $ctx = CHole \mid CLam\ name\ ctx \mid CAppL\ ctx\ exp \mid CAppR\ exp\ ctx$
 $\mid CPrimL\ nat \Rightarrow nat \Rightarrow nat\ ctx\ exp \mid CPrimR\ nat \Rightarrow nat \Rightarrow nat\ exp\ ctx$
 $\mid CIf1\ ctx\ exp\ exp \mid CIf2\ exp\ ctx\ exp \mid CIf3\ exp\ exp\ ctx$

fun *plug* :: $ctx \Rightarrow exp \Rightarrow exp$ **where**

plug $CHole\ e = e \mid$
plug $(CLam\ x\ C)\ e = ELam\ x\ (plug\ C\ e) \mid$
plug $(CAppL\ C\ e2)\ e = EApp\ (plug\ C\ e)\ e2 \mid$
plug $(CAppR\ e1\ C)\ e = EApp\ e1\ (plug\ C\ e) \mid$
plug $(CPrimL\ f\ C\ e2)\ e = EPrim\ f\ (plug\ C\ e)\ e2 \mid$

$plug (CPrimR f e1 C) e = EPrim f e1 (plug C e) \mid$
 $plug (CIf1 C e2 e3) e = EIf (plug C e) e2 e3 \mid$
 $plug (CIf2 e1 C e3) e = EIf e1 (plug C e) e3 \mid$
 $plug (CIf3 e1 e2 C) e = EIf e1 e2 (plug C e)$

lemma congruence: $E e = E e' \implies E (plug C e) = E (plug C e')$
 <proof>

14.2 Auxiliary lemmas

lemma diverge-denot-empty: **assumes** d : *diverge e* **and** fv : $FV e = \{\}$ **shows** $E e [] = \{\}$
 <proof>

lemma goes-wrong-denot-empty:
assumes gw : *goes-wrong e* **and** fv : $FV e = \{\}$ **shows** $E e [] = \{\}$
 <proof>

lemma denot-empty-diverge: **assumes** E : $E e [] = \{\}$ **and** fv : $FV e = \{\}$
shows *diverge e* \vee *goes-wrong e*
 <proof>

lemma val-ty-observe:
 $\llbracket A \in E v []; A \in E v' [] \rrbracket$;
 $observe v ob; isval v'; isval v \rrbracket \implies observe v' ob$
 <proof>

14.3 Soundness wrt. contextual equivalence

lemma soundness-wrt-ctx-equiv-aux[*rule-format*]:
assumes $e12$: $E e1 = E e2$
and fv : $FV (plug C e1) = \{\}$ **and** fv : $FV (plug C e2) = \{\}$
shows $run (plug C e1) ob \longrightarrow run (plug C e2) ob$
 <proof>

definition *ctx-equiv* :: $exp \Rightarrow exp \Rightarrow bool$ (**infix** \simeq 51) **where**
 $e \simeq e' \equiv \forall C ob. FV (plug C e) = \{\} \wedge FV (plug C e') = \{\} \longrightarrow$
 $run (plug C e) ob = run (plug C e') ob$

theorem denot-sound-wrt-ctx-equiv: **assumes** $e12$: $E e1 = E e2$ **shows** $e1 \simeq e2$
 <proof>

end

14.4 Denotational equalities regarding reduction

theory *DenotEqualitiesFSet*
imports *DenotCongruenceFSet*
begin

theorem *eval-prim*[*simp*]: **assumes** $e1$: $E e1 = E (ENat n1)$ **and** $e2$: $E e2 = E (ENat n2)$
shows $E(EPrim f e1 e2) = E(ENat (f n1 n2))$
 <proof>

theorem *eval-ifz*[*simp*]: **assumes** $e1$: $E e1 = E(ENat 0)$ **shows** $E(EIf e1 e2 e3) = E(e3)$
 <proof>

theorem *eval-ifnz*[*simp*]: **assumes** $e1$: $E(e1) = E(ENat n)$ **and** nz : $n \neq 0$
shows $E(EIf e1 e2 e3) = E(e2)$
 <proof>

theorem *eval-app-lam:* **assumes** vv : *is-val v*

shows $E(\text{EApp } (\text{ELam } x \ e) \ v) = E(\text{subst } x \ v \ e)$
 ⟨proof⟩

end

15 Correctness of an optimizer

theory *Optimizer*

imports *Lambda DenotEqualitiesFSet*

begin

fun *is-value* :: *exp* \Rightarrow *bool* **where**

is-value (*ENat* *n*) = *True* |
is-value (*ELam* *x e*) = (*FV e* = {}) |
is-value - = *False*

lemma *is-value-is-val[simp]*: *is-value e* \Longrightarrow *isval e* \wedge *FV e* = {}
 ⟨proof⟩

fun *opt* :: *exp* \Rightarrow *nat* \Rightarrow *exp* **where**

opt (*EVar* *x*) *k* = *EVar* *x* |
opt (*ENat* *n*) *k* = *ENat* *n* |
opt (*ELam* *x e*) *k* = *ELam* *x* (*opt e k*) |
opt (*EApp* *e1 e2*) *0* = *EApp* (*opt e1 0*) (*opt e2 0*) |
opt (*EApp* *e1 e2*) (*Suc k*) =
 (let *e1'* = *opt e1* (*Suc k*) in let *e2'* = *opt e2* (*Suc k*) in
 (case *e1'* of
 ELam *x e* \Rightarrow if *is-value e2'* then *opt* (*subst x e2'* *e*) *k*
 else *EApp* *e1' e2'*
 | - \Rightarrow *EApp* *e1' e2'*)) |
opt (*EPrim* *f e1 e2*) *k* =
 (let *e1'* = *opt e1 k* in let *e2'* = *opt e2 k* in
 (case (*e1'*, *e2'*) of
 (*ENat* *n1*, *ENat* *n2*) \Rightarrow *ENat* (*f n1 n2*)
 | - \Rightarrow *EPrim* *f e1' e2'*)) |
opt (*EIf* *e1 e2 e3*) *k* =
 (let *e1'* = *opt e1 k* in let *e2'* = *opt e2 k* in let *e3'* = *opt e3 k* in
 (case *e1'* of
 ENat *n* \Rightarrow if *n* = 0 then *e3'* else *e2'*
 | - \Rightarrow *EIf* *e1' e2' e3'*))

lemma *opt-correct-aux*: $E \ e = E \ (\text{opt } e \ k)$
 ⟨proof⟩

theorem *opt-correct*: $e \simeq \text{opt } e \ k$
 ⟨proof⟩

end

16 Semantics and type soundness for System F

theory *SystemF*

imports *Main HOL-Library.FSet*

begin

16.1 Syntax and values

type-synonym *name* = *nat*

datatype $ty = TVar\ nat \mid TNat \mid Fun\ ty\ ty\ (\text{infix } \rightarrow 60) \mid Forall\ ty$

datatype $exp = EVar\ name \mid ENat\ nat \mid ELam\ ty\ exp \mid EApp\ exp\ exp$
 $\mid EAbs\ exp \mid EInst\ exp\ ty \mid EFix\ ty\ exp$

datatype $val = VNat\ nat \mid Fun\ (val \times val)\ fset \mid Abs\ val\ option \mid Wrong$

fun $val-le :: val \Rightarrow val \Rightarrow bool\ (\text{infix } \sqsubseteq 52)\ \text{where}$

$(VNat\ n) \sqsubseteq (VNat\ n') = (n = n') \mid$
 $(Fun\ f) \sqsubseteq (Fun\ f') = (fset\ f \subseteq fset\ f') \mid$
 $(Abs\ None) \sqsubseteq (Abs\ None) = True \mid$
 $Abs\ (Some\ v) \sqsubseteq Abs\ (Some\ v') = v \sqsubseteq v' \mid$
 $Wrong \sqsubseteq Wrong = True \mid$
 $(v::val) \sqsubseteq v' = False$

16.2 Set monad

definition $set-bind :: 'a\ set \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'b\ set\ \text{where}$

$set-bind\ m\ f \equiv \{ v. \exists v'. v' \in m \wedge v \in f\ v' \}$

declare $set-bind-def[simp]$

syntax $-set-bind :: [pttrns, 'a\ set, 'b] \Rightarrow 'c\ ((- \leftarrow -; /-)\ 0)$

translations $P \leftarrow E; F \Rightarrow CONST\ set-bind\ E\ (\lambda P.\ F)$

definition $errset-bind :: val\ set \Rightarrow (val \Rightarrow val\ set) \Rightarrow val\ set\ \text{where}$

$errset-bind\ m\ f \equiv \{ v. \exists v'. v' \in m \wedge v' \neq Wrong \wedge v \in f\ v' \} \cup \{ v. v = Wrong \wedge Wrong \in m \}$

declare $errset-bind-def[simp]$

syntax $-errset-bind :: [pttrns, val\ set, val] \Rightarrow 'c\ ((- := -; /-)\ 0)$

translations $P := E; F \Rightarrow CONST\ errset-bind\ E\ (\lambda P.\ F)$

definition $return :: val \Rightarrow val\ set\ \text{where}$

$return\ v \equiv \{ v'. v' \sqsubseteq v \}$

declare $return-def[simp]$

16.3 Denotational semantics

type-synonym $tyenv = (val\ set)\ list$

type-synonym $env = val\ list$

inductive $iterate :: (env \Rightarrow val\ set) \Rightarrow env \Rightarrow val \Rightarrow bool\ \text{where}$

$iterate-none[intro!]:\ iterate\ Ee\ \varrho\ (Fun\ \{\|\}) \mid$

$iterate-again[intro!]:\ \llbracket\ iterate\ Ee\ \varrho\ f; f' \in Ee\ (f\#\varrho)\ \rrbracket \Longrightarrow iterate\ Ee\ \varrho\ f'$

abbreviation $apply-fun :: val\ set \Rightarrow val\ set \Rightarrow val\ set\ \text{where}$

$apply-fun\ V1\ V2 \equiv (v1 := V1; v2 := V2;$

$\text{case } v1 \text{ of } Fun\ f \Rightarrow$

$(v2', v3') \leftarrow fset\ f;$

$\text{if } v2' \sqsubseteq v2 \text{ then return } v3' \text{ else } \{\}$

$\mid - \Rightarrow \text{return } Wrong)$

fun $E :: exp \Rightarrow env \Rightarrow val\ set\ \text{where}$

$Enat: E\ (ENat\ n)\ \varrho = \text{return } (VNat\ n) \mid$

$Evar: E\ (EVar\ n)\ \varrho = \text{return } (\varrho!n) \mid$

$Elam: E\ (ELam\ \tau\ e)\ \varrho = \{ v. \exists f. v = Fun\ f \wedge (\forall v1\ v2'. (v1, v2') \in fset\ f \longrightarrow$

$(\exists v2. v2 \in E\ e\ (v1\#\varrho) \wedge v2' \sqsubseteq v2)) \}$ \mid

$Eapp: E\ (EApp\ e1\ e2)\ \varrho = \text{apply-fun } (E\ e1\ \varrho)\ (E\ e2\ \varrho) \mid$

$Efix: E\ (EFix\ \tau\ e)\ \varrho = \{ v. \text{iterate } (E\ e)\ \varrho\ v \} \mid$

$Eabs: E\ (EAbs\ e)\ \varrho = \{ v. (\exists v'. v = Abs\ (Some\ v') \wedge v' \in E\ e\ \varrho)$

$\vee (v = Abs\ None \wedge E\ e\ \varrho = \{\}) \}$ \mid

$Einst: E\ (EInst\ e\ \tau)\ \varrho =$

```

(v := E e ρ;
 case v of
   Abs None ⇒ {}
 | Abs (Some v') ⇒ return v'
 | - ⇒ return Wrong)

```

16.4 Types: substitution and semantics

```

fun shift :: nat ⇒ nat ⇒ ty ⇒ ty where
  shift k c TNat = TNat |
  shift k c (TVar n) = (if c ≤ n then TVar (n + k) else TVar n) |
  shift k c (σ → σ') = (shift k c σ) → (shift k c σ') |
  shift k c (Forall σ) = Forall (shift k (Suc c) σ)

```

```

fun subst :: nat ⇒ ty ⇒ ty ⇒ ty where
  subst k τ TNat = TNat |
  subst k τ (TVar n) = (if k = n then τ
                       else if k < n then TVar (n - 1)
                       else TVar n) |
  subst k τ (σ → σ') = (subst k τ σ) → (subst k τ σ') |
  subst k τ (Forall σ) = Forall (subst (Suc k) (shift (Suc 0) 0 τ) σ)

```

```

fun T :: ty ⇒ tyenv ⇒ val set where
  Tnat: T TNat ρ = {v. ∃ n. v = VNat n} |
  Tvar: T (TVar n) ρ = (if n < length ρ then
                        {v. ∃ v'. v' ∈ ρ!n ∧ v ⊆ v' ∧ v ≠ Wrong}
                        else {}) |
  Tfun: T (σ → τ) ρ = {v. ∃ f. v = Fun f ∧
                        (∀ v1 v2'. (v1, v2') ∈ fset f →
                          v1 ∈ T σ ρ → (∃ v2. v2 ∈ T τ ρ ∧ v2' ⊆ v2))} |
  Tall: T (Forall τ) ρ = {v. (∃ v'. v = Abs (Some v') ∧ (∀ V. v' ∈ T τ (V#ρ)))
                        ∨ v = Abs None}

```

16.5 Type system

type-synonym $tyctx = (ty \times nat) list \times nat$

definition $wf\text{-}tyvar :: tyctx \Rightarrow nat \Rightarrow bool$ **where**

$wf\text{-}tyvar \Gamma n \equiv n < snd \Gamma$

definition $push\text{-}ty :: ty \Rightarrow tyctx \Rightarrow tyctx$ **where**

$push\text{-}ty \tau \Gamma \equiv ((\tau, snd \Gamma) \# fst \Gamma, snd \Gamma)$

definition $push\text{-}tyvar :: tyctx \Rightarrow tyctx$ **where**

$push\text{-}tyvar \Gamma \equiv (fst \Gamma, Suc (snd \Gamma))$

definition $good\text{-}ctx :: tyctx \Rightarrow bool$ **where**

$good\text{-}ctx \Gamma \equiv \forall n. n < length (fst \Gamma) \longrightarrow snd ((fst \Gamma)!n) \leq snd \Gamma$

definition $lookup :: tyctx \Rightarrow nat \Rightarrow ty$ **option** **where**

$lookup \Gamma n \equiv$ (if $n < length (fst \Gamma)$ then
 let $k = snd \Gamma - snd ((fst \Gamma)!n)$ in
 Some (shift k 0 (fst ((fst \Gamma)!n)))
 else None)

inductive $well\text{-}typed :: tyctx \Rightarrow exp \Rightarrow ty \Rightarrow bool$ $(- \vdash - : - [55,55,55] 54)$ **where**

$wtnat[intro!]: \Gamma \vdash ENat n : TNat$ |
 $wtvar[intro!]: \llbracket lookup \Gamma n = Some \tau \rrbracket \Longrightarrow \Gamma \vdash EVar n : \tau$ |
 $wtapp[intro!]: \llbracket \Gamma \vdash e : \sigma \rightarrow \tau; \Gamma \vdash e' : \sigma \rrbracket \Longrightarrow \Gamma \vdash EApp e e' : \tau$ |
 $wtlam[intro!]: \llbracket push\text{-}ty \sigma \Gamma \vdash e : \tau \rrbracket \Longrightarrow \Gamma \vdash ELam \sigma e : \sigma \rightarrow \tau$ |
 $wtfix[intro!]: \llbracket push\text{-}ty (\sigma \rightarrow \tau) \Gamma \vdash e : \sigma \rightarrow \tau \rrbracket \Longrightarrow \Gamma \vdash EFix (\sigma \rightarrow \tau) e : \sigma \rightarrow \tau$ |
 $wtabs[intro!]: \llbracket push\text{-}tyvar \Gamma \vdash e : \tau \rrbracket \Longrightarrow \Gamma \vdash EAbs e : Forall \tau$ |
 $wtinst[intro!]: \llbracket \Gamma \vdash e : Forall \tau \rrbracket \Longrightarrow \Gamma \vdash EInst e \sigma : (subst 0 \sigma \tau)$

inductive *wfenv* :: *env* \Rightarrow *tyenv* \Rightarrow *tyctx* \Rightarrow *bool* (\vdash $_$, $_$: - [55,55,55] 54) **where**
wfnil[*intro!*]: $\vdash [] , [] : ([], 0) \mid$
wfbind[*intro!*]: $\llbracket \vdash \varrho, \eta : \Gamma ; v \in T \tau \eta \rrbracket \Longrightarrow \vdash (v\#\varrho), \eta : \text{push-ty } \tau \Gamma \mid$
wftbind[*intro!*]: $\llbracket \vdash \varrho, \eta : \Gamma \rrbracket \Longrightarrow \vdash \varrho, (V\#\eta) : \text{push-tyvar } \Gamma$

inductive-cases

wtnat-inv[*elim!*]: $\Gamma \vdash ENat \ n : \tau$ **and**
wtvar-inv[*elim!*]: $\Gamma \vdash EVar \ n : \tau$ **and**
wtapp-inv[*elim!*]: $\Gamma \vdash EApp \ e \ e' : \tau$ **and**
wtlam-inv[*elim!*]: $\Gamma \vdash ELam \ \sigma \ e : \tau$ **and**
wtfix-inv[*elim!*]: $\Gamma \vdash EFix \ \sigma \ e : \tau$ **and**
wtabs-inv[*elim!*]: $\Gamma \vdash EAbs \ e : \tau$ **and**
wtinst-inv[*elim!*]: $\Gamma \vdash EInst \ e \ \sigma : \tau$

lemma *wfenv-good-ctx*: $\vdash \varrho, \eta : \Gamma \Longrightarrow \text{good-ctx } \Gamma$
<proof>

16.6 Well-typed Programs don't go wrong

lemma *nth-append1*[*simp*]: $n < \text{length } \varrho1 \Longrightarrow (\varrho1 @ \varrho2)!n = \varrho1!n$
<proof>

lemma *nth-append2*[*simp*]: $n \geq \text{length } \varrho1 \Longrightarrow (\varrho1 @ \varrho2)!n = \varrho2!(n - \text{length } \varrho1)$
<proof>

lemma *shift-append-preserves-T-aux*:
shows $T \tau (\varrho1 @ \varrho3) = T (\text{shift } (\text{length } \varrho2) (\text{length } \varrho1) \tau) (\varrho1 @ \varrho2 @ \varrho3)$
<proof>

lemma *shift-append-preserves-T*: **shows** $T \tau \varrho3 = T (\text{shift } (\text{length } \varrho2) 0 \tau) (\varrho2 @ \varrho3)$
<proof>

lemma *drop-shift-preserves-T*:
assumes $k: k \leq \text{length } \varrho$ **shows** $T \tau (\text{drop } k \ \varrho) = T (\text{shift } k \ 0 \ \tau) \ \varrho$
<proof>

lemma *shift-cons-preserves-T*: **shows** $T \tau \varrho = T (\text{shift } (\text{Suc } 0) 0 \ \tau) (b\#\varrho)$
<proof>

lemma *compose-shift*: **shows** $\text{shift } (j+k) \ c \ \tau = \text{shift } j \ c (\text{shift } k \ c \ \tau)$
<proof>

lemma *shift-zero-id*[*simp*]: $\text{shift } 0 \ c \ \tau = \tau$
<proof>

lemma *lookup-wfenv*: **assumes** $r-g: \vdash \varrho, \eta : \Gamma$ **and** $ln: \text{lookup } \Gamma \ n = \text{Some } \tau$
shows $\exists v. \varrho!n = v \wedge v \in T \tau \eta$
<proof>

lemma *less-wrong*[*elim!*]: $\llbracket v \sqsubseteq \text{Wrong}; v = \text{Wrong} \Longrightarrow P \rrbracket \Longrightarrow P$
<proof>

lemma *less-nat*[*elim!*]: $\llbracket v \sqsubseteq \text{VNat } n; v = \text{VNat } n \Longrightarrow P \rrbracket \Longrightarrow P$
<proof>

lemma *less-fun*[*elim!*]: $\llbracket v \sqsubseteq \text{Fun } f; \wedge f'. \llbracket v = \text{Fun } f'; \text{fset } f' \subseteq \text{fset } f \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
<proof>

lemma *less-refl*[*simp*]: $v \sqsubseteq v$
<proof>

lemma *less-trans*: fixes $v1::val$ and $v2::val$ and $v3::val$
shows $\llbracket v1 \sqsubseteq v2; v2 \sqsubseteq v3 \rrbracket \implies v1 \sqsubseteq v3$
 $\langle proof \rangle$

lemma *T-down-closed*: assumes $vt: v \in T \tau \eta$ and $vp-v: v' \sqsubseteq v$
shows $v' \in T \tau \eta$
 $\langle proof \rangle$

lemma *wrong-not-in-T*: $Wrong \notin T \tau \eta$
 $\langle proof \rangle$

lemma *fun-app*: assumes $vmn: V \subseteq T (m \rightarrow n) \eta$ and $v2s: V' \subseteq T m \eta$
shows *apply-fun* $V V' \subseteq T n \eta$
 $\langle proof \rangle$

lemma *T-eta*: $\{v. \exists v'. v' \in T \sigma (\eta) \wedge v \sqsubseteq v' \wedge v \neq Wrong\} = T \sigma \eta$
 $\langle proof \rangle$

lemma *compositionality*: $T \tau (\eta1 @ (T \sigma (\eta1 @ \eta2))) \# \eta2 = T (\text{subst } (\text{length } \eta1) \sigma \tau) (\eta1 @ \eta2)$
 $\langle proof \rangle$

lemma *iterate-sound*:
assumes *it*: *iterate* $Ee \varrho v$
and *IH*: $\forall v. v \in T (\sigma \rightarrow \tau) \eta \longrightarrow Ee (v \# \varrho) \subseteq T (\sigma \rightarrow \tau) \eta$
shows $v \in T (\sigma \rightarrow \tau) \eta$ $\langle proof \rangle$

theorem *welltyped-dont-go-wrong*:
assumes *wte*: $\Gamma \vdash e : \tau$ and *wfr*: $\vdash \varrho, \eta : \Gamma$
shows $E e \varrho \subseteq T \tau \eta$
 $\langle proof \rangle$

end

17 Semantics of mutable references

theory *MutableRef*
imports *Main HOL-Library.FSet*
begin

datatype $ty = T\text{Nat} \mid T\text{Fun } ty \ ty \ (\text{infix } \rightarrow 60) \mid T\text{Pair } ty \ ty \mid T\text{Ref } ty$

type-synonym $\text{name} = \text{nat}$

datatype $\text{exp} = E\text{Var } \text{name} \mid E\text{Nat } \text{nat} \mid E\text{Lam } ty \ \text{exp} \mid E\text{App } \text{exp } \text{exp}$
 $\mid E\text{Prim } \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat } \text{exp } \text{exp} \mid E\text{If } \text{exp } \text{exp } \text{exp}$
 $\mid E\text{Pair } \text{exp } \text{exp} \mid E\text{Fst } \text{exp} \mid E\text{Snd } \text{exp}$
 $\mid E\text{Ref } \text{exp} \mid E\text{Read } \text{exp} \mid E\text{Write } \text{exp } \text{exp}$

17.1 Denotations (values)

datatype $\text{val} = V\text{Nat } \text{nat} \mid V\text{Fun } (\text{val} \times \text{val}) \ \text{fset} \mid V\text{Pair } \text{val } \text{val} \mid V\text{Addr } \text{nat} \mid \text{Wrong}$

type-synonym $\text{func} = (\text{val} \times \text{val}) \ \text{fset}$
type-synonym $\text{store} = \text{func}$

inductive $\text{val-le} :: \text{val} \Rightarrow \text{val} \Rightarrow \text{bool}$ (**infix** $\sqsubseteq 52$) **where**
 $\text{vnat-le}[\text{intro!}]: (V\text{Nat } n) \sqsubseteq (V\text{Nat } n) \mid$
 $\text{vaddr-le}[\text{intro!}]: (V\text{Addr } a) \sqsubseteq (V\text{Addr } a) \mid$
 $\text{wrong-le}[\text{intro!}]: \text{Wrong} \sqsubseteq \text{Wrong} \mid$

vfun-le[intro!]: $t1 \sqsubseteq t2 \implies (VFun\ t1) \sqsubseteq (VFun\ t2) \mid$
vpair-le[intro!]: $\llbracket v1 \sqsubseteq v1'; v2 \sqsubseteq v2' \rrbracket \implies (VPair\ v1\ v2) \sqsubseteq (VPair\ v1'\ v2')$

primrec *vsize* :: *val* \Rightarrow *nat* **where**

vsize (VNat *n*) = 1 \mid
vsize (VFun *t*) = 1 + $\text{ffold } (\lambda(-,v), (-,u)).\lambda r. v + u + r) 0$
(fimage (map-prod ($\lambda v. (v, \text{vsize } v)$) ($\lambda v. (v, \text{vsize } v)))$) *t*) \mid
vsize (VPair *v1 v2*) = 1 + *vsize* *v1* + *vsize* *v2* \mid
vsize (VAddr *a*) = 1 \mid
vsize Wrong = 1

17.2 Non-deterministic state monad

type-synonym *'a M* = *store* \Rightarrow (*'a* \times *store*) *set*

definition *bind* :: *'a M* \Rightarrow (*'a* \Rightarrow *'b M*) \Rightarrow *'b M* **where**

bind *m f* $\mu 1 \equiv \{ (v, \mu 3). \exists v' \mu 2. (v', \mu 2) \in m\ \mu 1 \wedge (v, \mu 3) \in f\ v' \mu 2 \}$
declare *bind-def[simp]*

syntax *-bind* :: [*pttrns, 'a M, 'b*] \Rightarrow *'c* ($(- \leftarrow -; / -) 0$)

translations *P* \leftarrow *E*; *F* \Leftarrow *CONST* *bind* *E* ($\lambda P. F$)

no-notation *binomial* (**infixl** *choose* 65)

definition *choose* :: *'a set* \Rightarrow *'a M* **where**

choose *S* $\mu \equiv \{ (a, \mu 1). a \in S \wedge \mu 1 = \mu \}$
declare *choose-def[simp]*

definition *return* :: *'a* \Rightarrow *'a M* **where**

return *v* $\mu \equiv \{ (v, \mu) \}$
declare *return-def[simp]*

definition *zero* :: *'a M* **where**

zero $\mu \equiv \{ \}$
declare *zero-def[simp]*

definition *err-bind* :: *val M* \Rightarrow (*val* \Rightarrow *val M*) \Rightarrow *val M* **where**

err-bind *m f* $\equiv (x \leftarrow m; \text{if } x = \text{Wrong} \text{ then return Wrong else } f\ x)$
declare *err-bind-def[simp]*

syntax *-errset-bind* :: [*pttrns, val M, val*] \Rightarrow *'c* ($(- := -; / -) 0$)

translations *P* := *E*; *F* \Leftarrow *CONST* *err-bind* *E* ($\lambda P. F$)

definition *down* :: *val* \Rightarrow *val M* **where**

down *v* $\mu 1 \equiv \{ (v', \mu). v' \sqsubseteq v \wedge \mu = \mu 1 \}$
declare *down-def[simp]*

definition *get-store* :: *store M* **where**

get-store $\mu \equiv \{ (\mu, \mu) \}$
declare *get-store-def[simp]*

definition *put-store* :: *store* \Rightarrow *unit M* **where**

put-store $\mu \equiv \lambda-. \{ ((), \mu) \}$
declare *put-store-def[simp]*

definition *mapM* :: *'a fset* \Rightarrow (*'a* \Rightarrow *'b M*) \Rightarrow (*'b fset*) *M* **where**

mapM *as f* $\equiv \text{ffold } (\lambda a. \lambda r. (b \leftarrow f\ a; bs \leftarrow r; \text{return } (\text{finsert } b\ bs))) (\text{return } \{\})$ *as*

definition *run* :: *store* \Rightarrow *val M* \Rightarrow (*val* \times *store*) *set* **where**

run $\sigma m \equiv m\ \sigma$
declare *run-def[simp]*

definition $sdom :: store \Rightarrow nat \text{ set}$ **where**
 $sdom \mu \equiv \{a. \exists v. (VAddr a, v) \in fset \mu \}$

definition $max\text{-}addr :: store \Rightarrow nat$ **where**
 $max\text{-}addr \mu = \text{ffold } (\lambda a. \lambda r. \text{case } a \text{ of } (VAddr n, -) \Rightarrow \max n r \mid - \Rightarrow r) 0 \mu$

17.3 Denotational semantics

abbreviation $apply\text{-}fun :: val M \Rightarrow val M \Rightarrow val M$ **where**
 $apply\text{-}fun V1 V2 \equiv (v1 := V1; v2 := V2;$
 $\text{case } v1 \text{ of } VFun f \Rightarrow$
 $(p, p') \leftarrow \text{choose } (fset f); \mu 0 \leftarrow \text{get-store};$
 $(\text{case } (p, p') \text{ of } (VPair v (VFun \mu), VPair v' (VFun \mu')) \Rightarrow$
 $\text{if } v \sqsubseteq v2 \wedge (VFun \mu) \sqsubseteq (VFun \mu 0) \text{ then } (- \leftarrow \text{put-store } \mu'; \text{down } v')$
 else zero
 $\mid - \Rightarrow \text{zero})$
 $\mid - \Rightarrow \text{return Wrong})$

fun $nvals :: nat \Rightarrow (val fset) M$ **where**
 $nvals 0 = \text{return } \{\}\}$ |
 $nvals (Suc k) = (v \leftarrow \text{choose UNIV}; L \leftarrow nvals k; \text{return } (finsert v L))$

definition $vals :: (val fset) M$ **where**
 $vals \equiv (n \leftarrow \text{choose UNIV}; nvals n)$
declare $vals\text{-}def[simp]$

fun $npairs :: nat \Rightarrow func M$ **where**
 $npairs 0 = \text{return } \{\}\}$ |
 $npairs (Suc k) = (v \leftarrow \text{choose UNIV}; v' \leftarrow \text{choose } \{v::val. True\};$
 $P \leftarrow npairs k; \text{return } (finsert (v, v') P))$

definition $tables :: func M$ **where**
 $tables \equiv (n \leftarrow \text{choose } \{k::nat. True\}; npairs n)$
declare $tables\text{-}def[simp]$

definition $read :: nat \Rightarrow val M$ **where**
 $read a \equiv (\mu \leftarrow \text{get-store}; \text{if } a \in sdom \mu \text{ then}$
 $((v1, v2) \leftarrow \text{choose } (fset \mu); \text{if } v1 = VAddr a \text{ then return } v2 \text{ else zero})$
 $\text{else return Wrong})$
declare $read\text{-}def[simp]$

definition $update :: nat \Rightarrow val \Rightarrow val M$ **where**
 $update a v \equiv (\mu \leftarrow \text{get-store};$
 $- \leftarrow \text{put-store } (finsert (VAddr a, v) (\text{ffilter } (\lambda(v, v'). v \neq VAddr a) \mu));$
 $\text{return } (VAddr a))$
declare $update\text{-}def[simp]$

type-synonym $env = val \text{ list}$

fun $E :: exp \Rightarrow env \Rightarrow val M$ **where**
 $Enat: E (ENat n) \varrho = \text{return } (VNat n) \mid$
 $Evar: E (EVar n) \varrho = (\text{if } n < \text{length } \varrho \text{ then down } (\varrho!n) \text{ else return Wrong}) \mid$
 $Elam: E (ELam A e) \varrho = (L \leftarrow \text{vals};$
 $t \leftarrow \text{mapM } L (\lambda v. (\mu \leftarrow \text{tables}; (v', \mu') \leftarrow \text{choose } (\text{run } \mu (E e (v \# \varrho))));$
 $\text{return } (VPair v (VFun \mu), VPair v' (VFun \mu')));$
 $\text{return } (VFun t)) \mid$
 $Eapp: E (EApp e1 e2) \varrho = \text{apply-fun } (E e1 \varrho) (E e2 \varrho) \mid$
 $Eprim: E (EPrim f e1 e2) \varrho = (v1 := E e1 \varrho; v2 := E e2 \varrho;$
 $\text{case } (v1, v2) \text{ of } (VNat n1, VNat n2) \Rightarrow \text{return } (VNat (f n1 n2))$
 $\mid - \Rightarrow \text{return Wrong}) \mid$

Eif: $E (EIf\ e1\ e2\ e3)\ \varrho = (v1 := E\ e1\ \varrho; \text{case } v1 \text{ of } VNat\ n \Rightarrow (\text{if } n = 0 \text{ then } E\ e3\ \varrho \text{ else } E\ e2\ \varrho) \mid - \Rightarrow \text{return } Wrong) \mid$
Epair: $E (EPair\ e1\ e2)\ \varrho = (v1 := E\ e1\ \varrho; v2 := E\ e2\ \varrho; \text{return } (VPair\ v1\ v2)) \mid$
Efst: $E (EFst\ e)\ \varrho = (v := E\ e\ \varrho; \text{case } v \text{ of } VPair\ v1\ v2 \Rightarrow \text{return } v1 \mid - \Rightarrow \text{return } Wrong) \mid$
Esnd: $E (ESnd\ e)\ \varrho = (v := E\ e\ \varrho; \text{case } v \text{ of } VPair\ v1\ v2 \Rightarrow \text{return } v2 \mid - \Rightarrow \text{return } Wrong) \mid$
Eref: $E (ERef\ e)\ \varrho = (v := E\ e\ \varrho; \mu \leftarrow \text{get-store}; a \leftarrow \text{choose } UNIV;$
 if $a \in \text{sdom } \mu$ *then* zero
 else $(- \leftarrow \text{put-store } (\text{finsert } (VAddr\ a, v)\ \mu);$
 return $(VAddr\ a)) \mid$
Eread: $E (ERead\ e)\ \varrho = (v := E\ e\ \varrho; \text{case } v \text{ of } VAddr\ a \Rightarrow \text{read } a \mid - \Rightarrow \text{return } Wrong) \mid$
Ewrite: $E (EWrite\ e1\ e2)\ \varrho = (v1 := E\ e1\ \varrho; v2 := E\ e2\ \varrho;$
 case $v1 \text{ of } VAddr\ a \Rightarrow \text{update } a\ v2 \mid - \Rightarrow \text{return } Wrong)$

end

theory *MutableRefProps*

imports *MutableRef*

begin

inductive-cases

vfun-le-inv[*elim!*]: $VFun\ t1 \sqsubseteq VFun\ t2$ **and**
le-fun-nat-inv[*elim!*]: $VFun\ t2 \sqsubseteq VNat\ x1$ **and**
le-any-nat-inv[*elim!*]: $v \sqsubseteq VNat\ n$ **and**
le-nat-any-inv[*elim!*]: $VNat\ n \sqsubseteq v$ **and**
le-fun-any-inv[*elim!*]: $VFun\ t \sqsubseteq v$ **and**
le-any-fun-inv[*elim!*]: $v \sqsubseteq VFun\ t$ **and**
le-pair-any-inv[*elim!*]: $VPair\ v1\ v2 \sqsubseteq v$ **and**
le-any-pair-inv[*elim!*]: $v \sqsubseteq VPair\ v1\ v2$ **and**
le-addr-any-inv[*elim!*]: $VAddr\ a \sqsubseteq v$ **and**
le-any-addr-inv[*elim!*]: $v \sqsubseteq VAddr\ a$ **and**
le-wrong-any-inv[*elim!*]: $Wrong \sqsubseteq v$ **and**
le-any-wrong-inv[*elim!*]: $v \sqsubseteq Wrong$

proposition *val-le-refl*: $v \sqsubseteq v$ *<proof>*

proposition *val-le-trans*: $\llbracket v1 \sqsubseteq v2; v2 \sqsubseteq v3 \rrbracket \Longrightarrow v1 \sqsubseteq v3$
<proof>

proposition *val-le-antisymm*: $\llbracket v1 \sqsubseteq v2; v2 \sqsubseteq v1 \rrbracket \Longrightarrow v1 = v2$
<proof>

end