Declarative Semantics for Functional Languages

Jeremy G. Siek

May 14, 2024

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

theory *Lambda*
imports *Main*
begin

type-synonym *name* = *nat*

datatype *exp* = *EVar name* | *ENat nat* | *ELam name exp* | *EApp exp exp*
  | *EPrim nat* ⇒ *nat* ⇒ *nat exp exp* | *EIf exp exp exp*

fun *lookup* :: ('*a* × '*b*) *list* ⇒ '*a* ⇒ '*b option* **where**
  lookup [] *x* = *None* |
  lookup ((*y*,*v*)#*ls*) *x* = (*if* (*x* = *y*) *then Some v else lookup ls x*)

fun *FV* :: *exp* ⇒ *nat set* **where**
  FV (*EVar x*) = {*x*} |
  FV (*ENat n*) = {} |
  FV (*ELam x e*) = *FV e* − {*x*} |
  FV (*EApp e1 e2*) = *FV e1* ∪ *FV e2* |
  FV (*EPrim f e1 e2*) = *FV e1* ∪ *FV e2* |
  FV (*EIf e1 e2 e3*) = *FV e1* ∪ *FV e2* ∪ *FV e3*

fun *BV* :: *exp* ⇒ *nat set* **where**
  BV (*EVar x*) = {} |
  BV (*ENat n*) = {} |
  BV (*ELam x e*) = *BV e* ∪ {*x*} |
  BV (*EApp e1 e2*) = *BV e1* ∪ *BV e2* |
  BV (*EPrim f e1 e2*) = *BV e1* ∪ *BV e2* |
  BV (*EIf e1 e2 e3*) = *BV e1* ∪ *BV e2* ∪ *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* ⇒ *exp* ⇒ *exp* ⇒ *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* ⇒ *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* ⇒ *bool* **where**
  is-val v ≡ *isval v* ∧ *FV v* = {}
declare *is-val-def*[*simp*]

inductive *reduce* :: *exp* ⇒ *exp* ⇒ *bool* (**infix** ⟶ *55*) **where**
  beta[*intro*!]: ⟦ *is-val v* ⟧ ⟹ *EApp* (*ELam x e*) *v* ⟶ (*subst x v e*) |
  app-left[*intro*!]: ⟦ *e1* ⟶ *e1'* ⟧ ⟹ *EApp e1 e2* ⟶ *EApp e1' e2* |
  app-right[*intro*!]: ⟦ *e2* ⟶ *e2'* ⟧ ⟹ *EApp e1 e2* ⟶ *EApp e1 e2'* |
  delta[*intro*!]: *EPrim f* (*ENat n1*) (*ENat n2*) ⟶ *ENat* (*f n1 n2*) |
  prim-left[*intro*!]: ⟦ *e1* ⟶ *e1'* ⟧ ⟹ *EPrim f e1 e2* ⟶ *EPrim f e1' e2* |
  prim-right[*intro*!]: ⟦ *e2* ⟶ *e2'* ⟧ ⟹ *EPrim f e1 e2* ⟶ *EPrim f e1 e2'* |
  if-zero[*intro*!]: *EIf* (*ENat 0*) *thn els* ⟶ *els* |
  if-nz[*intro*!]: *n* ≠ *0* ⟹ *EIf* (*ENat n*) *thn els* ⟶ *thn* |
  if-cond[*intro*!]: ⟦ *cond* ⟶ *cond'* ⟧ ⟹
    EIf cond thn els ⟶ *EIf cond' thn els*

inductive-cases
  red-var-inv[*elim*!]: *EVar x* ⟶ *e* **and**
  red-int-inv[*elim*!]: *ENat n* ⟶ *e* **and**
  red-lam-inv[*elim*!]: *ELam x e* ⟶ *e'* **and**
  red-app-inv[*elim*!]: *EApp e1 e2* ⟶ *e'*

inductive *multi-step* :: *exp* ⇒ *exp* ⇒ *bool* (**infix** ⟶∗ *55*) **where**
  ms-nil[*intro*!]: *e* ⟶∗ *e* |
  ms-cons[*intro*!]: ⟦ *e1* ⟶ *e2*; *e2* ⟶∗ *e3* ⟧ ⟹ *e1* ⟶∗ *e3*

definition *diverge* :: *exp* ⇒ *bool* **where**
  diverge e ≡ (∀ *e'*. *e* ⟶∗ *e'* ⟶ (∃ *e''*. *e'* ⟶ *e''*))

definition *stuck* :: *exp* ⇒ *bool* **where**
  stuck e ≡ ¬ (∃ *e'*. *e* ⟶ *e'*)
declare *stuck-def*[*simp*]

definition *goes-wrong* :: *exp* ⇒ *bool* **where**
  goes-wrong e ≡ ∃ *e'*. *e* ⟶∗ *e'* ∧ *stuck e'* ∧ ¬ *isval e'*
declare *goes-wrong-def*[*simp*]

datatype *obs* = *ONat nat* | *OFun* | *OBad*

fun *observe* :: *exp* ⇒ *obs* ⇒ *bool* **where**
  observe (*ENat n*) (*ONat n'*) = (*n* = *n'*) |
  observe (*ELam x e*) *OFun* = *True* |
  observe e ob = *False*

definition *run* :: *exp* ⇒ *obs* ⇒ *bool* (**infix** ⇓ *52*) **where**
  run e ob ≡ ((∃ *v*. *e* ⟶∗ *v* ∧ *observe v ob*)
        ∨ ((*diverge e* ∨ *goes-wrong e*) ∧ *ob* = *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*) ⊆ *FV e* − {*x*}
  ⟨*proof*⟩

lemma *subst-fv*: **assumes** *fv-e*: *FV e* ⊆ {*x*} **and** *fv-v*: *FV v* = {}
  shows *FV* (*subst x v e*) = {}
  ⟨*proof*⟩

lemma *red-pres-fv*: **fixes** *e*::*exp* **assumes** *red*: *e* ⟶ *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* ⇒ *exp* ⇒ *bval* ⇒ *bool* (- ⊢ - ⇓ - [*50*,*50*,*50*] *51*) **where**
eval-nat[*intro*!]: $\varrho \vdash ENat\ n \Downarrow BNat\ n$ |
eval-var[*intro*!]: $lookup\ \varrho\ x = Some\ v \Longrightarrow \varrho \vdash EVar\ x \Downarrow v$ |
eval-lam[*intro*!]: $\varrho \vdash ELam\ x\ e \Downarrow BClos\ x\ e\ \varrho$ |
eval-app[*intro*!]: ⟦ $\varrho \vdash e1 \Downarrow BClos\ x\ e\ \varrho'$; $\varrho \vdash e2 \Downarrow arg$;
$(x,arg)\#\varrho' \vdash e \Downarrow v$ ⟧ ⟹
$\varrho \vdash EApp\ e1\ e2 \Downarrow v$ |
eval-prim[*intro*!]: ⟦ $\varrho \vdash e1 \Downarrow BNat\ n1$; $\varrho \vdash e2 \Downarrow BNat\ n2$; $n3 = f\ n1\ n2$⟧ ⟹
$\varrho \vdash EPrim\ f\ e1\ e2 \Downarrow BNat\ n3$ |
eval-if0[*intro*!]: ⟦ $\varrho \vdash e1 \Downarrow BNat\ 0$; $\varrho \vdash e3 \Downarrow v3$ ⟧ ⟹
$\varrho \vdash EIf\ e1\ e2\ e3 \Downarrow v3$ |
eval-if1[*intro*!]: ⟦ $\varrho \vdash e1 \Downarrow BNat\ n$; $n \neq 0$; $\varrho \vdash e2 \Downarrow v2$ ⟧ ⟹
$\varrho \vdash EIf\ e1\ e2\ e3 \Downarrow v2$

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* ⇒ *exp* ⇒ *exp* **where**
$psubst\ \varrho\ (ENat\ n) = ENat\ n$ |
$psubst\ \varrho\ (EVar\ x) =$
(*case* $lookup\ \varrho\ x$ *of*
None ⇒ $EVar\ x$
| *Some* $v \Rightarrow v$) |
$psubst\ \varrho\ (ELam\ x\ e) = ELam\ x\ (psubst\ ((x,EVar\ x)\#\varrho)\ e)$ |
$psubst\ \varrho\ (EApp\ e1\ e2) = EApp\ (psubst\ \varrho\ e1)\ (psubst\ \varrho\ e2)$ |
$psubst\ \varrho\ (EPrim\ f\ e1\ e2) = EPrim\ f\ (psubst\ \varrho\ e1)\ (psubst\ \varrho\ e2)$ |
$psubst\ \varrho\ (EIf\ e1\ e2\ e3) = EIf\ (psubst\ \varrho\ e1)\ (psubst\ \varrho\ e2)\ (psubst\ \varrho\ e3)$

inductive *bs-val* :: *bval* ⇒ *exp* ⇒ *bool* **and**
bs-env :: *benv* ⇒ *env* ⇒ *bool* **where**
bs-nat[*intro*!]: $bs\text{-}val\ (BNat\ n)\ (ENat\ n)$ |
bs-clos[*intro*!]: ⟦ $bs\text{-}env\ \varrho\ \varrho'$; $FV\ (ELam\ x\ (psubst\ ((x,EVar\ x)\#\varrho')\ e)) = \{\}$ ⟧ ⟹
$bs\text{-}val\ (BClos\ x\ e\ \varrho)\ (ELam\ x\ (psubst\ ((x,EVar\ x)\#\varrho')\ e))$ |

bs-nil[*intro*!]: *bs-env* [] [] |
bs-cons[*intro*!]: ⟦ *bs-val w v*; *bs-env ϱ ϱ′* ⟧ ⟹ *bs-env* ((*x*,*w*)#*ϱ*) ((*x*,*v*)#*ϱ′*)

inductive-cases *bs-env-inv1*[*elim*!]: *bs-env* ((*x*, *w*) # *ϱ*) *ϱ′* **and**
bs-clos-inv[*elim*!]: *bs-val* (*BClos x e ϱ″*) *v1* **and**
bs-nat-inv[*elim*!]: *bs-val* (*BNat n*) *v*

lemma *bs-val-is-val*[*intro*!]: *bs-val w v* ⟹ *is-val v*
⟨*proof*⟩

lemma *lookup-bs-env*: ⟦ *bs-env ϱ ϱ′*; *lookup ϱ x* = *Some w* ⟧ ⟹
∃ *v*. *lookup ϱ′ x* = *Some v* ∧ *bs-val w v*
⟨*proof*⟩

lemma *app-red-cong1*: *e1* ⟶∗ *e1′* ⟹ *EApp e1 e2* ⟶∗ *EApp e1′ e2*
⟨*proof*⟩

lemma *app-red-cong2*: *e2* ⟶∗ *e2′* ⟹ *EApp e1 e2* ⟶∗ *EApp e1 e2′*
⟨*proof*⟩

lemma *prim-red-cong1*: *e1* ⟶∗ *e1′* ⟹ *EPrim f e1 e2* ⟶∗ *EPrim f e1′ e2*
⟨*proof*⟩

lemma *prim-red-cong2*: *e2* ⟶∗ *e2′* ⟹ *EPrim f e1 e2* ⟶∗ *EPrim f e1 e2′*
⟨*proof*⟩

lemma *if-red-cong1*: *e1* ⟶∗ *e1′* ⟹ *EIf e1 e2 e3* ⟶∗ *EIf e1′ e2 e3*
⟨*proof*⟩

lemma *multi-step-trans*: ⟦ *e1* ⟶∗ *e2*; *e2* ⟶∗ *e3* ⟧ ⟹ *e1* ⟶∗ *e3*
⟨*proof*⟩

lemma *subst-id-fv*: *x* ∉ *FV e* ⟹ *subst x v e* = *e*
⟨*proof*⟩

definition *sdom* :: *env* ⇒ *name set* **where**
sdom ϱ ≡ {*x*. ∃ *v*. *lookup ϱ x* = *Some v* ∧ *v* ≠ *EVar x* }

definition *closed-env* :: *env* ⇒ *bool* **where**
closed-env ϱ ≡ (∀ *x v*. *x* ∈ *sdom ϱ* ⟶ *lookup ϱ x* = *Some v* ⟶ *FV v* = {})

definition *equiv-env* :: *env* ⇒ *env* ⇒ *bool* **where**
equiv-env ϱ ϱ′ ≡ (*sdom ϱ* = *sdom ϱ′* ∧ (∀ *x*. *x* ∈ *sdom ϱ* ⟶ *lookup ϱ x* = *lookup ϱ′ x*))

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

lemma *sdom-cons-v*[*simp*]: *FV v* = {} ⟹ *sdom* ((*x*,*v*)#*ϱ*) = *insert x* (*sdom ϱ*)
⟨*proof*⟩

lemma *lookup-some-in-dom*: ⟦ *lookup ϱ x* = *Some v*; *v* ≠ *EVar x* ⟧ ⟹ *x* ∈ *sdom ϱ*
⟨*proof*⟩

lemma *lookup-none-notin-dom*: *lookup ϱ x* = *None* ⟹ *x* ∉ *sdom ϱ*
⟨*proof*⟩

lemma *psubst-change*: *equiv-env ϱ ϱ′* ⟹ *psubst ϱ e* = *psubst ϱ′ e*
⟨*proof*⟩

lemma *subst-psubst*: ⟦ *closed-env ϱ*; *FV v* = {} ⟧ ⟹
subst x v (*psubst* ((*x*, *EVar x*) # *ϱ*) *e*) = *psubst* ((*x*, *v*) # *ϱ*) *e*

⟨*proof*⟩

inductive-cases *bsenv-nil*[*elim*!]: *bs-env* [] ϱ'

lemma *bs-env-dom*: *bs-env* ϱ ϱ' $\Longrightarrow$ *set* (*map fst* ϱ) = *sdom* ϱ'
⟨*proof*⟩

lemma *closed-env-cons*[*intro*!]: *FV v* = {} $\Longrightarrow$ *closed-env* ϱ'' $\Longrightarrow$ *closed-env* ((*a*, *v*) # ϱ'')
⟨*proof*⟩

lemma *bs-env-closed*: *bs-env* ϱ ϱ' $\Longrightarrow$ *closed-env* ϱ'
⟨*proof*⟩

lemma *psubst-fv*: *closed-env* ϱ $\Longrightarrow$ *FV* (*psubst* ϱ *e*) = *FV e* − *sdom* ϱ
⟨*proof*⟩

lemma *big-small-step*:
assumes *ev*: $\varrho \vdash e \Downarrow w$ **and** *r-rp*: *bs-env* ϱ ϱ' **and** *fv-e*: *FV e* ⊆ *set* (*map fst* ϱ)
shows ∃ *v*. *psubst* ϱ' *e* ⟶∗ *v* ∧ *is-val v* ∧ *bs-val w v*
⟨*proof*⟩

lemma *psubst-id*: *FV e* ∩ *sdom* ϱ = {} $\Longrightarrow$ *psubst* ϱ *e* = *e*
⟨*proof*⟩

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

theorem *sound-wrt-small-step*:
assumes *e-v*: [] ⊢ *e* ⇓ *v* **and** *fv-e*: *FV e* = {}
shows ∃ *v'* *ob*. *e* ⟶∗ *v'* ∧ *isval v'* ∧ *observe v' ob*
∧ *bs-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* ⇒ *val* ⇒ *bool* (**infix** ⊑ *52*) **where**
vnat-le[*intro*!]: (*VNat n*) ⊑ (*VNat n*) |
vfun-le[*intro*!]: *fset t1* ⊆ *fset t2* $\Longrightarrow$ (*VFun t1*) ⊑ (*VFun t2*)

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

definition *env-le* :: *env* ⇒ *env* ⇒ *bool* (**infix** ⊑ *52*) **where**
$\varrho \sqsubseteq \varrho' \equiv \forall$ *x v*. *lookup* ϱ *x* = *Some v* ⟶ (∃ *v'*. *lookup* ϱ' *x* = *Some v'* ∧ *v* ⊑ *v'*)

definition *env-eq* :: *env* ⇒ *env* ⇒ *bool* (**infix** ≈ *50*) **where**

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

fun *vadd* :: $(val \times nat) \times (val \times nat) \Rightarrow nat \Rightarrow nat$ **where**
vadd ((-,*v*),(-,*u*)) *r* = *v* + *u* + *r*

primrec *vsize* :: $val \Rightarrow nat$ **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* :: $val \times val \Rightarrow (val \times nat) \times (val \times nat)$ **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*]: $v \sqsubseteq VNat\ n \Longrightarrow v = VNat\ n$
⟨*proof*⟩

lemma *le-nat-nat*[*simp*]: $VNat\ n \sqsubseteq VNat\ n' \Longrightarrow n = n'$
⟨*proof*⟩

end

4 Declarative semantics as a relational semantics

theory *RelationalSemFSet*
imports *Lambda ValuesFSet*
begin

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

end
theory *DeclSemAsDenotFSet*
imports *Lambda ValuesFSet*
begin

5 Declarative semantics as a denotational semantics

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

end

6 Relational and denotational views are equivalent

theory *EquivRelationalDenotFSet*
imports *RelationalSemFSet DeclSemAsDenotFSet*
begin

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

lemma *rel-implies-denot*: $\varrho \vdash e \Rightarrow v \Longrightarrow v \in E\ e\ \varrho$
⟨*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*]: ⟦ *VNat n1* ∈ *E e1 ϱ*; *VNat n2* ∈ *E e2 ϱ*; *v* = *VNat* (*f n1 n2*) ⟧
⟹ *v* ∈ *E* (*EPrim f e1 e2*) *ϱ* ⟨*proof*⟩

lemma *e-prim-elim*[*elim*]: ⟦ *v* ∈ *E* (*EPrim f e1 e2*) *ϱ*;
⋀ *n1 n2*. ⟦ *VNat n1* ∈ *E e1 ϱ*; *VNat n2* ∈ *E e2 ϱ*; *v* = *VNat* (*f n1 n2*) ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *e-app-elim*[*elim*]: ⟦ *v3* ∈ *E* (*EApp e1 e2*) *ϱ*;
⋀ *f v2 v2′ v3′*. ⟦ *VFun f* ∈ *E e1 ϱ*; *v2* ∈ *E e2 ϱ*; (*v2′*,*v3′*) ∈ *fset f*; *v2′* ⊑ *v2*; *v3* ⊑ *v3′* ⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

lemma *e-app-intro*[*intro*]: ⟦ *VFun f* ∈ *E e1 ϱ*; *v2* ∈ *E e2 ϱ*; (*v2′*,*v3′*) ∈ *fset f*; *v2′* ⊑ *v2*; *v3* ⊑ *v3′*⟧
⟹ *v3* ∈ *E* (*EApp e1 e2*) *ϱ* ⟨*proof*⟩

lemma *e-lam-intro*[*intro*]: ⟦ *v* = *VFun f*;
∀ *v1 v2*. (*v1*,*v2*) ∈ *fset f* ⟶ *v2* ∈ *E e* ((*x*,*v1*)#*ϱ*) ⟧
⟹ *v* ∈ *E* (*ELam x e*) *ϱ*
⟨*proof*⟩

lemma *e-lam-intro2*[*intro*]:
⟦ *VFun f* ∈ *E* (*ELam x e*) *ϱ*; *v2* ∈ *E e* ((*x*,*v1*)#*ϱ*) ⟧
⟹ *VFun* (*finsert* (*v1*,*v2*) *f*) ∈ *E* (*ELam x e*) *ϱ*
⟨*proof*⟩

lemma *e-lam-intro3*[*intro*]: *VFun* {||} ∈ *E* (*ELam x e*) *ϱ*
⟨*proof*⟩

lemma *e-if-intro*[*intro*]: ⟦ *VNat n* ∈ *E e1 ϱ*; *n* = *0* ⟶ *v* ∈ *E e3 ϱ*; *n* ≠ *0* ⟶ *v* ∈ *E e2 ϱ* ⟧
⟹ *v* ∈ *E* (*EIf e1 e2 e3*) *ϱ*
⟨*proof*⟩

lemma *e-var-intro*[*elim*]: ⟦ *lookup ϱ x* = *Some v′*; *v* ⊑ *v′* ⟧ ⟹ *v* ∈ *E* (*EVar x*) *ϱ*
⟨*proof*⟩

lemma *e-var-elim*[*elim*]: ⟦ *v* ∈ *E* (*EVar x*) *ϱ*;
⋀ *v′*. ⟦ *lookup ϱ x* = *Some v′*; *v* ⊑ *v′* ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *e-lam-elim*[*elim*]: ⟦ *v* ∈ *E* (*ELam x e*) *ϱ*;
⋀ *f*. ⟦ *v* = *VFun f*; ∀ *v1 v2*. (*v1*,*v2*) ∈ *fset f* ⟶ *v2* ∈ *E e* ((*x*,*v1*)#*ϱ*) ⟧
⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *e-lam-elim2*[*elim*]: ⟦ *VFun* (*finsert* (*v1*,*v2*) *f*) ∈ *E* (*ELam x e*) *ϱ*;
⟦ *v2* ∈ *E e* ((*x*,*v1*)#*ϱ*) ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *e-if-elim*[*elim*]: ⟦ *v* ∈ *E* (*EIf e1 e2 e3*) *ϱ*;
⋀ *n*. ⟦ *VNat n* ∈ *E e1 ϱ*; *n* = *0* ⟶ *v* ∈ *E e3 ϱ*; *n* ≠ *0* ⟶ *v* ∈ *E e2 ϱ* ⟧ ⟹ *P* ⟧ ⟹ *P*

⟨*proof*⟩

definition *xenv-le* :: *name set* ⇒ *env* ⇒ *env* ⇒ *bool* (- ⊢ - ⊑ - [*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*: $[\![\ v \in E\ e\ \varrho;\ v' \sqsubseteq v\ ]\!] \Longrightarrow v' \in E\ e\ \varrho$
  ⟨*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*: $[\![\ v \in E\ e\ \varrho;\ \forall\ x.\ x \in FV\ e \longrightarrow lookup\ \varrho'\ x = lookup\ \varrho\ x\ ]\!] \Longrightarrow v \in E\ e\ \varrho'$
  ⟨*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* ⇒ (*'a* ⇒ *'b M*) ⇒ *'b M* **where**
  set-bind m f ≡ { *v*. ∃ *v'*. *v'* ∈ *m* ∧ *v* ∈ *f v'* }
declare *set-bind-def*[*simp*]

syntax *-set-bind* :: [*pttrns*,*'a M*,*'b*] ⇒ *'c* ((- ← -;//-) *0*)
translations *P* ← *E*; *F* ⇌ *CONST set-bind E* (λ*P*. *F*)

definition *return* :: *'a* ⇒ *'a M* **where**
  return v ≡ {*v*}
declare *return-def*[*simp*]

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

zero ≡ {}
declare *zero-def*[*simp*]

no-notation *binomial* (**infix** *choose* 64)

definition *choose* :: 'a *set* ⇒ 'a *M* **where**
choose S ≡ *S*
declare *choose-def*[*simp*]

definition *down* :: *val* ⇒ *val M* **where**
down v ≡ (*v'* ← *UNIV*; *if v'* ⊑ *v then return v' else zero*)
declare *down-def*[*simp*]

definition *mapM* :: 'a *fset* ⇒ ('a ⇒ 'b *M*) ⇒ ('b *fset*) *M* **where**
mapM as f ≡ *ffold* (λ*a*. λ*r*. (*b* ← *f a*; *bs* ← *r*; *return* (*finsert b bs*))) (*return* ({||})) *as*

8.2 Non-deterministic interpreter

abbreviation *apply-fun* :: *val M* ⇒ *val M* ⇒ *val M* **where**
apply-fun V1 V2 ≡ (*v1* ← *V1*; *v2* ← *V2*;
case v1 of VFun f ⇒
(*v2'*,*v3'*) ← *choose* (*fset f*);
if v2' ⊑ *v2 then return v3' else zero*
| - ⇒ *zero*)

fun *E* :: *exp* ⇒ *env* ⇒ *val set* **where**
Enat2: *E* (*ENat n*) ϱ = *return* (*VNat n*) |
Evar2: *E* (*EVar x*) ϱ = (*case lookup* ϱ *x of None* ⇒ *zero* | *Some v* ⇒ *down v*) |
Elam2: *E* (*ELam x e*) ϱ = (*vs* ← *choose UNIV*;
t ← *mapM vs* (λ *v*. (*v'* ← *E e* ((*x*,*v*)#ϱ); *return* (*v*, *v'*)));
return (*VFun t*)) |
Eapp2: *E* (*EApp e1 e2*) ϱ = *apply-fun* (*E e1* ϱ) (*E e2* ϱ) |
Eprim2: *E* (*EPrim f e1 e2*) ϱ = (*v*₁ ← *E e1* ϱ; *v*₂ ← *E e2* ϱ;
case (*v*₁,*v*₂) *of*
(*VNat n*₁, *VNat n*₂) ⇒ *return* (*VNat* (*f n*₁ *n*₂))
| (*VNat n*₁, *VFun t*₂) ⇒ *zero*
| (*VFun t*₁, *v*₂) ⇒ *zero*) |
Eif2[*eta-contract* = *false*]: *E* (*EIf e1 e2 e3*) ϱ = (*v*₁ ← *E e1* ϱ;
*case v*₁ *of*
(*VNat n*) ⇒ *if n* ≠ *0 then E e2* ϱ *else E e3* ϱ
| (*VFun t*) ⇒ *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** → 55) | *TInt funty funty* (**infix** ⊓ 56) | *TTop* (⊤)

inductive *subtype* :: *ty* ⇒ *ty* ⇒ *bool* (**infix** <: 52)
and *fsubtype* :: *funty* ⇒ *funty* ⇒ *bool* (**infix** <:: 52) **where**
sub-refl: *A* <: *A* |
sub-funty[*intro*!]: *f1* <:: *f2* ⟹ *TFun f1* <: *TFun f2* |
sub-fun[*intro*!]: ⟦ *T1* <: *T1'*; *T1'* <: *T1*; *T2* <: *T2'*; *T2'* <: *T2* ⟧ ⟹ (*T1*→*T2*) <:: (*T1'*→*T2'*) |
sub-inter-l1[*intro*!]: *T1* ⊓ *T2* <:: *T1* |
sub-inter-l2[*intro*!]: *T1* ⊓ *T2* <:: *T2* |

sub-inter-r[*intro*!]: ⟦ *T3* <:: *T1*; *T3* <:: *T2* ⟧ ⟹ *T3* <:: *T1* ⊓ *T2* |
sub-fun-top[*intro*!]: *T1* → *T2* <:: ⊤ |
sub-top-top[*intro*!]: ⊤ <:: ⊤ |
fsub-refl[*intro*!]: *T* <:: *T* |
sub-trans[*trans*]: ⟦ *T1* <:: *T2*; *T2* <:: *T3* ⟧ ⟹ *T1* <:: *T3*

definition *ty-eq* :: *ty* ⇒ *ty* ⇒ *bool* (**infix** ≈ *50*) **where**
A ≈ *B* ≡ *A* <: *B* ∧ *B* <: *A*
definition *fty-eq* :: *funty* ⇒ *funty* ⇒ *bool* (**infix** ≃ *50*) **where**
F1 ≃ *F2* ≡ *F1* <:: *F2* ∧ *F2* <:: *F1*

type-synonym *tyenv* = (*name* × *ty*) *list*

inductive *wt* :: *tyenv* ⇒ *exp* ⇒ *ty* ⇒ *bool* (- ⊢ - : - [*51*,*51*,*51*] *51*) **where**
wt-var[*intro*!]: *lookup* Γ *x* = *Some T* ⟹ Γ ⊢ *EVar x* : *T* |
wt-nat[*intro*!]: Γ ⊢ *ENat n* : *TNat n* |
wt-lam[*intro*!]: ⟦ (*x*,*A*)#Γ ⊢ *e* : *B* ⟧ ⟹ Γ ⊢ *ELam x e* : *TFun* (*A* → *B*) |
wt-app[*intro*!]: ⟦ Γ ⊢ *e1* : *TFun* (*A* → *B*); Γ ⊢ *e2* : *A* ⟧ ⟹ Γ ⊢ *EApp e1 e2* : *B* |
wt-top[*intro*!]: Γ ⊢ *ELam x e* : *TFun* ⊤ |
wt-inter[*intro*!]: ⟦ Γ ⊢ *ELam x e* : *TFun A*; Γ ⊢ *ELam x e* : *TFun B* ⟧
⟹ Γ ⊢ *ELam x e* : *TFun* (*A* ⊓ *B*) |
wt-sub[*intro*!]: ⟦ Γ ⊢ *e* : *A*; *A* <: *B* ⟧ ⟹ Γ ⊢ *e* : *B* |
wt-prim[*intro*!]: ⟦ Γ ⊢ *e1* : *TNat n1*; Γ ⊢ *e2* : *TNat n2* ⟧
⟹ Γ ⊢ *EPrim f e1 e2* : *TNat* (*f n1 n2*) |
wt-ifz[*intro*!]: ⟦ Γ ⊢ *e1* : *TNat 0*; Γ ⊢ *e3* : *B* ⟧
⟹ Γ ⊢ *EIf e1 e2 e3* : *B* |
wt-ifnz[*intro*!]: ⟦ Γ ⊢ *e1* : *TNat n*; *n* ≠ *0*; Γ ⊢ *e2* : *B* ⟧
⟹ Γ ⊢ *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* | *VFun* (*val* × *val*) *list*

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

inductive *val-le* :: *val* ⇒ *val* ⇒ *bool* (**infix** ⊑ *52*)
and *fun-le* :: *func* ⇒ *func* ⇒ *bool* (**infix** ≲ *52*) **where**
vnat-le[*intro*!]: (*VNat n*) ⊑ (*VNat n*) |
vfun-le[*intro*!]: *t1* ≲ *t2* ⟹ (*VFun t1*) ⊑ (*VFun t2*) |
fun-le[*intro*!]: (∀ *v1 v2*. (*v1*,*v2*) ∈ *set t1* ⟶
(∃ *v3 v4*. (*v3*,*v4*) ∈ *set t2*
∧ *v1* ⊑ *v3* ∧ *v3* ⊑ *v1* ∧ *v2* ⊑ *v4* ∧ *v4* ⊑ *v2*))
⟹ *t1* ≲ *t2*

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

definition *env-le* :: *env* ⇒ *env* ⇒ *bool* (**infix** ⊑ *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* ⇒ *env* ⇒ *bool* (**infix** ≈ *50*) **where**
$\varrho \approx \varrho' \equiv (\forall\ x.\ lookup\ \varrho\ x = lookup\ \varrho'\ x)$

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*: $(a \# t1) \lesssim t2 \Longrightarrow t1 \lesssim t2$
⟨*proof*⟩

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

lemma *val-size-mem*: $(a, b) \in set\ t \Longrightarrow val\text{-}size\ a + val\text{-}size\ b < fun\text{-}size\ t$
⟨*proof*⟩
lemma *val-size-mem-l*: $(a, b) \in set\ t \Longrightarrow val\text{-}size\ a < fun\text{-}size\ t$
⟨*proof*⟩
lemma *val-size-mem-r*: $(a, b) \in set\ t \Longrightarrow val\text{-}size\ b < fun\text{-}size\ t$
⟨*proof*⟩

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

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

lemma *fun-le-refl*[*simp*]: **fixes** *t*::*func* **shows** $t \lesssim t$ ⟨*proof*⟩

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

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

lemma *vfun-eq*[*intro*!]: $t \sim t' \Longrightarrow VFun\ t \sim VFun\ t'$
⟨*proof*⟩

lemma *val-eq-refl*[*simp*]: **fixes** *v*::*val* **shows** $v \sim v$
⟨*proof*⟩

lemma *val-eq-symm*: **fixes** *v1*::*val* **and** *v2*::*val* **shows** $v1 \sim v2 \Longrightarrow v2 \sim v1$
⟨*proof*⟩

lemma *val-le-fun-le-trans*:
$\forall\ v2\ t2.\ n = \mathit{val\text{-}size}\ v2 + \mathit{fun\text{-}size}\ t2 \longrightarrow$
$(\forall\ v1\ v3.\ v1 \sqsubseteq v2 \longrightarrow v2 \sqsubseteq v3 \longrightarrow v1 \sqsubseteq v3)$
$\wedge\ (\forall\ t1\ t3.\ t1 \lesssim t2 \longrightarrow t2 \lesssim t3 \longrightarrow t1 \lesssim t3)$
⟨*proof*⟩

proposition *val-le-trans*: **fixes** *v2*::*val* **shows** $[\![\ v1 \sqsubseteq v2;\ v2 \sqsubseteq v3\ ]\!] \Longrightarrow v1 \sqsubseteq v3$
⟨*proof*⟩

lemma *fun-le-trans*: $[\![\ t1 \lesssim t2;\ t2 \lesssim t3\ ]\!] \Longrightarrow t1 \lesssim t3$
⟨*proof*⟩

lemma *val-eq-trans*: **fixes** *v1*::*val* **and** *v2*::*val* **and** *v3*::*val*
assumes *v12*: $v1 \sim v2$ **and** *v23*: $v2 \sim v3$ **shows** $v1 \sim v3$
⟨*proof*⟩

lemma *fun-eq-refl*[*simp*]: **fixes** *t*::*func* **shows** $t \sim t$
⟨*proof*⟩

lemma *fun-eq-trans*: **fixes** *t1*::*func* **and** *t2*::*func* **and** *t3*::*func*
assumes *t12*: $t1 \sim t2$ **and** *t23*: $t2 \sim t3$ **shows** $t1 \sim t3$
⟨*proof*⟩

lemma *append-fun-le*:
$[\![\ t1' \lesssim t1;\ t2' \lesssim t2\ ]\!] \Longrightarrow t1'\ @\ t2' \lesssim t1\ @\ t2$
⟨*proof*⟩

lemma *append-fun-equiv*:
$[\![\ t1' \sim t1;\ t2' \sim t2\ ]\!] \Longrightarrow t1'\ @\ t2' \sim t1\ @\ t2$
⟨*proof*⟩

lemma *append-leq-symm*: $t2\ @\ t1 \lesssim t1\ @\ t2$
⟨*proof*⟩

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

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

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

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

end

10.3 Declarative semantics as a denotational semantics

theory *DeclSemAsDenot*
imports *Lambda Values*
begin

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

$VFun\ f \in E\ e1\ \varrho \wedge v2 \in E\ e2\ \varrho \wedge (v2', v3') \in set\ f \wedge v2' \sqsubseteq v2 \wedge v3 \sqsubseteq v3'\ \} \mid$
Eprim: $E\ (EPrim\ f\ e1\ e2)\ \varrho = \{\ v.\ \exists\ n1\ n2.\ VNat\ n1 \in E\ e1\ \varrho$
$\wedge\ VNat\ n2 \in E\ e2\ \varrho \wedge v = VNat\ (f\ n1\ n2)\ \} \mid$
Eif: $E\ (EIf\ e1\ e2\ e3)\ \varrho = \{\ v.\ \exists\ n.\ VNat\ n \in E\ e1\ \varrho$
$\wedge\ (n = 0 \longrightarrow v \in E\ e3\ \varrho) \wedge (n \neq 0 \longrightarrow v \in E\ e2\ \varrho)\ \}$

end

10.4 Subsumption and change of environment

theory *DenotLam5*
imports *Main Lambda DeclSemAsDenot ValueProps*
begin

lemma *e-prim-intro*[*intro*]: $[\![\ VNat\ n1 \in E\ e1\ \varrho;\ VNat\ n2 \in E\ e2\ \varrho;\ v = VNat\ (f\ n1\ n2)\ ]\!]$
$\Longrightarrow v \in E\ (EPrim\ f\ e1\ e2)\ \varrho$ ⟨*proof*⟩

lemma *e-prim-elim*[*elim*]: $[\![\ v \in E\ (EPrim\ f\ e1\ e2)\ \varrho;$
$\bigwedge n1\ n2.\ [\![\ VNat\ n1 \in E\ e1\ \varrho;\ VNat\ n2 \in E\ e2\ \varrho;\ v = VNat\ (f\ n1\ n2)\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *e-app-elim*[*elim*]: $[\![\ v3 \in E\ (EApp\ e1\ e2)\ \varrho;$
$\bigwedge f\ v2\ v2'\ v3'.\ [\![\ VFun\ f \in E\ e1\ \varrho;\ v2 \in E\ e2\ \varrho;\ (v2',v3') \in set\ f;\ v2' \sqsubseteq v2;\ v3 \sqsubseteq v3'\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *e-app-intro*[*intro*]: $[\![\ VFun\ f \in E\ e1\ \varrho;\ v2 \in E\ e2\ \varrho;\ (v2',v3') \in set\ f;\ v2' \sqsubseteq v2;\ v3 \sqsubseteq v3'\ ]\!]$
$\Longrightarrow v3 \in E\ (EApp\ e1\ e2)\ \varrho$ ⟨*proof*⟩

lemma *e-lam-intro*[*intro*]: $[\![\ v = VFun\ f;$
$\forall\ v1\ v2.\ (v1,v2) \in set\ f \longrightarrow v2 \in E\ e\ ((x,v1)\#\varrho)\ ]\!]$
$\Longrightarrow v \in E\ (ELam\ x\ e)\ \varrho$
⟨*proof*⟩

lemma *e-lam-intro2*[*intro*]:
$[\![\ VFun\ f \in E\ (ELam\ x\ e)\ \varrho;\ v2 \in E\ e\ ((x,v1)\#\varrho)\ ]\!]$
$\Longrightarrow VFun\ ((v1,v2)\#f) \in E\ (ELam\ x\ e)\ \varrho$
⟨*proof*⟩

lemma *e-lam-intro3*[*intro*]: $VFun\ [] \in E\ (ELam\ x\ e)\ \varrho$
⟨*proof*⟩

lemma *e-if-intro*[*intro*]: $[\![\ VNat\ n \in E\ e1\ \varrho;\ n = 0 \longrightarrow v \in E\ e3\ \varrho;\ n \neq 0 \longrightarrow v \in E\ e2\ \varrho\ ]\!]$
$\Longrightarrow v \in E\ (EIf\ e1\ e2\ e3)\ \varrho$
⟨*proof*⟩

lemma *e-var-intro*[*elim*]: $[\![\ lookup\ \varrho\ x = Some\ v';\ v \sqsubseteq v'\ ]\!] \Longrightarrow v \in E\ (EVar\ x)\ \varrho$
⟨*proof*⟩

lemma *e-var-elim*[*elim*]: $[\![\ v \in E\ (EVar\ x)\ \varrho;$
$\bigwedge v'.\ [\![\ lookup\ \varrho\ x = Some\ v';\ v \sqsubseteq v'\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *e-lam-elim*[*elim*]: $[\![\ v \in E\ (ELam\ x\ e)\ \varrho;$
$\bigwedge f.\ [\![\ v = VFun\ f;\ \forall\ v1\ v2.\ (v1,v2) \in set\ f \longrightarrow v2 \in E\ e\ ((x,v1)\#\varrho)\ ]\!]$
$\Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *e-lam-elim2*[*elim*]: $[\![\ VFun\ ((v1,v2)\#f) \in E\ (ELam\ x\ e)\ \varrho;$
$[\![\ v2 \in E\ e\ ((x,v1)\#\varrho)\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *e-if-elim*[*elim*]: ⟦ $v \in E$ (*EIf e1 e2 e3*) ϱ;
$\bigwedge n.$ ⟦ *VNat* $n \in E$ *e1* ϱ; $n = 0 \longrightarrow v \in E$ *e3* ϱ; $n \neq 0 \longrightarrow v \in E$ *e2* ϱ ⟧ $\Longrightarrow P$ ⟧ $\Longrightarrow P$
⟨*proof*⟩

definition *xenv-le* :: *name set* ⇒ *env* ⇒ *env* ⇒ *bool* (- ⊢ - ⊑ - [*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*: ⟦ $v \in E\ e\ \varrho$; $v' \sqsubseteq v$ ⟧ $\Longrightarrow v' \in E\ e\ \varrho$
⟨*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*: ⟦ $v \in E\ e\ \varrho$; $\forall\ x.\ x \in FV\ e \longrightarrow$ *lookup* $\varrho'\ x =$ *lookup* $\varrho\ x$ ⟧ $\Longrightarrow v \in E\ e\ \varrho'$
⟨*proof*⟩

end

11 Equivalence of denotational and type system views

theory *EquivDenotInterTypes*
imports *InterTypeSystem DeclSemAsDenot DenotLam5*
begin

fun *V* :: *ty* ⇒ *val* **and** *Vf* :: *funty* ⇒ (*val* × *val*) *list* **where**
V (*TNat n*) = *VNat n* |
V (*TFun f*) = *VFun* (*Vf f*) |
Vf ($A \to B$) = [(*V A*, *V B*)] |
Vf ($A \sqcap B$) = *Vf A* @ *Vf B* |
Vf $\top$ = []

fun *Venv* :: *tyenv* ⇒ *env* **where**
Venv [] = [] |
Venv ((*x*,*A*)#Γ) = (*x*, *V A*)# *Venv* Γ

function *T* :: *val* ⇒ *ty* **and** *Tf* :: (*val* × *val*) *list* ⇒ *funty* **where**
T (*VNat n*) = *TNat n* |

$T\ (VFun\ t) = TFun\ (Tf\ t)$ |
$Tf\ [] = \top$ |
$Tf\ ((v1,v2)\#t) = (T\ v1 \to T\ v2) \sqcap Tf\ t$
⟨*proof*⟩
termination T ⟨*proof*⟩

fun *Tenv* :: *env* ⇒ *tyenv* **where**
$Tenv\ [] = []$ |
$Tenv\ ((x,v)\#\varrho) = (x, T\ v)\#Tenv\ \varrho$

lemma *sub-inter-left1*: $A <:: C \Longrightarrow A \sqcap B <:: C$
⟨*proof*⟩

lemma *sub-inter-left2*: $B <:: C \Longrightarrow A \sqcap B <:: C$
⟨*proof*⟩

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

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

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

lemma *lookup-tenv*:
$lookup\ \varrho\ x = Some\ v \Longrightarrow lookup\ (Tenv\ \varrho)\ x = Some\ (T\ v)$
⟨*proof*⟩

proposition *table-mem-sub*:
$(v,\ v') \in set\ t \Longrightarrow Tf\ t <:: (T\ v) \to (T\ v')$
⟨*proof*⟩

lemma *Tf-top*: $Tf\ t <:: \top$
⟨*proof*⟩

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

proposition *le-sub-flip*: $v \sqsubseteq v' \Longrightarrow T\ v' <:\ T\ v$ ⟨*proof*⟩

lemma *le-sub-fun-flip*: $t \lesssim t' \Longrightarrow Tf\ t' <:: Tf\ t$ ⟨*proof*⟩

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

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

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

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

theorem *denot-implies-ts*:
assumes *ve*: $v \in E\ e\ \varrho$ **shows** $Tenv\ \varrho \vdash e : T\ v$
⟨*proof*⟩

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

lemma *append-fun-equiv*: ⟦ *t1'* ∼ *t1*; *t2'* ∼ *t2* ⟧ ⟹ *t1'* @ *t2'* ∼ *t1* @ *t2*
⟨*proof*⟩

lemma *append-eq-symm*: *t2* @ *t1* ∼ *t1* @ *t2*
⟨*proof*⟩

lemma *sub-le-flip*: (*A* <: *B* ⟶ *V B* ⊑ *V A*) ∧ (*f1* <:: *f2* ⟶ (*Vf f2*) ≲ (*Vf f1*))
⟨*proof*⟩

theorem *ts-implies-denot*:
assumes *wte*: Γ ⊢ *e* : *A* **shows** *V A* ∈ *E e* (*Venv* Γ)
⟨*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* ≠ *x* ⟹ *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 Γ::*env* **and** *A*::*val*
assumes *wte*: *B* ∈ *E e* Γ' **and** *wtv*: *A* ∈ *E v* []
and *gp*: Γ' ≈ (*x*,*A*)#Γ **and** *v*: *is-val v*
shows *B* ∈ *E* (*subst x v e*) Γ
⟨*proof*⟩

12.2 Reduction preserves denotation

lemma *subject-reduction*: **fixes** *e*::*exp* **assumes** *v*: *v* ∈ *E e* ϱ **and** *r*: *e* ⟶ *e'* **shows** *v* ∈ *E e'* ϱ
⟨*proof*⟩

theorem *preservation*: **assumes** *v*: *v* ∈ *E e* ϱ **and** *rr*: *e* ⟶∗ *e'* **shows** *v* ∈ *E e'* ϱ
⟨*proof*⟩

lemma *canonical-nat*: **assumes** *v*: *VNat n* ∈ *E v* ϱ **and** *vv*: *isval v* **shows** *v* = *ENat n*
⟨*proof*⟩

lemma *canonical-fun*: **assumes** *v*: *VFun f* ∈ *E v* ϱ **and** *vv*: *isval v* **shows** ∃ *x e*. *v* = *ELam x e*
⟨*proof*⟩

12.3 Progress

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

12.4 Logical relation between values and big-step values

fun *good-entry* :: *name* ⇒ *exp* ⇒ *benv* ⇒ (*val* × *bval set*) × (*val* × *bval set*) ⇒ *bool* ⇒ *bool* **where**
good-entry x e ϱ ((*v1*,*g1*),(*v2*,*g2*)) *r* = ((∀ *v* ∈ *g1*. ∃ *v'*. (*x*,*v*)#*ϱ* ⊢ *e* ⇓ *v'* ∧ *v'* ∈ *g2*) ∧ *r*)

primrec *good* :: *val* ⇒ *bval set* **where**
Gnat: *good* (*VNat n*) = { *BNat n* } |
Gfun: *good* (*VFun f*) = { *vc*. ∃ *x e ϱ*. *vc* = *BClos x e ϱ*
∧ (*ffold* (*good-entry x e ϱ*) *True* (*fimage* (*map-prod* (λ*v*. (*v*,*good v*)) (λ*v*. (*v*,*good v*))) *f*)) }

inductive *good-env* :: *benv* ⇒ *env* ⇒ *bool* **where**
genv-nil[*intro*!]: *good-env* [] [] |
genv-cons[*intro*!]: ⟦ *v* ∈ *good v'*; *good-env ϱ ϱ'* ⟧ ⟹ *good-env* ((*x*,*v*)#*ϱ*) ((*x*,*v'*)#*ϱ'*)

inductive-cases
genv-any-nil-inv: *good-env ϱ* [] **and**
genv-any-cons-inv: *good-env ϱ* (*b*#*ϱ'*)

lemma *lookup-good*:
assumes *l*: *lookup ϱ' x* = *Some A* **and** *EE*: *good-env ϱ ϱ'*
shows ∃ *v*. *lookup ϱ x* = *Some v* ∧ *v* ∈ *good A*
⟨*proof*⟩

abbreviation *good-prod* :: *val* × *val* ⇒ (*val* × *bval set*) × (*val* × *bval set*) **where**
good-prod ≡ *map-prod* (λ*v*. (*v*,*good v*)) (λ*v*. (*v*,*good v*))

lemma *good-prod-inj*: *inj-on good-prod* (*fset A*)
⟨*proof*⟩

definition *good-fun* :: *func* ⇒ *name* ⇒ *exp* ⇒ *benv* ⇒ *bool* **where**
good-fun f x e ϱ ≡ (*ffold* (*good-entry x e ϱ*) *True* (*fimage good-prod f*))

lemma *good-fun-def2*:
good-fun f x e ϱ = *ffold* (*good-entry x e ϱ* ∘ *good-prod*) *True f*
⟨*proof*⟩

lemma *gfun-elim*: *w* ∈ *good* (*VFun f*) ⟹ ∃ *x e ϱ*. *w* = *BClos x e ϱ* ∧ *good-fun f x e ϱ*
⟨*proof*⟩

lemma *gfun-mem-iff*: *good-fun f x e ϱ* = (∀ *v1 v2*. (*v1*,*v2*) ∈ *fset f* ⟶
(∀ *v* ∈ *good v1*. ∃ *v'*. (*x*,*v*)#*ϱ* ⊢ *e* ⇓ *v'* ∧ *v'* ∈ *good v2*))
⟨*proof*⟩

lemma *gfun-mem*: ⟦ (*v1*,*v2*) ∈ *fset f*; *good-fun f x e ϱ* ⟧
⟹ ∀ *v* ∈ *good v1*. ∃ *v'*. (*x*,*v*)#*ϱ* ⊢ *e* ⇓ *v'* ∧ *v'* ∈ *good v2*
⟨*proof*⟩

lemma *gfun-intro*: (∀ *v1 v2*.(*v1*,*v2*)∈*fset f*⟶(∀ *v*∈*good v1*.∃ *v'*.(*x*,*v*)#*ϱ* ⊢ *e* ⇓ *v'*∧*v'*∈*good v2*))
⟹ *good-fun f x e ϱ* ⟨*proof*⟩

lemma *sub-good*: **fixes** *v*::*val* **assumes** *wv*: *w* ∈ *good v* **and** *vp-v*: *v'* ⊑ *v* **shows** *w* ∈ *good v'*
⟨*proof*⟩

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 $\exists\ v.\ \varrho \vdash e \Downarrow v \wedge v \in good\ v'$
⟨*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* ⇒ *val* ⇒ *val option* (**infix** ⊔ *60*) **where**
$(VNat\ n) \sqcup (VNat\ n') = (if\ n = n'\ then\ Some\ (VNat\ n)\ else\ None)$ |
$(VFun\ f) \sqcup (VFun\ f') = Some\ (VFun\ (f\ |\cup|\ f'))$ |
$v \sqcup v' = None$

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*: $[\![\ v1 \sqcup v2 = Some\ v12;\ v1 \sqsubseteq v3;\ v2 \sqsubseteq v3\ ]\!] \Longrightarrow v12 \sqsubseteq v3$
⟨*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*: $[\![\ lookup\ \varrho\ y = None;\ x \neq y\ ]\!] \Longrightarrow lookup\ ((x,v)\#\varrho)\ y = None$
⟨*proof*⟩
lemma *rev-subst-var*:
assumes *ev*: $e = EVar\ y \wedge v = e'$ **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 $v \in E\ e\ \varrho$
⟨*proof*⟩

lemma *reverse-multi-step-pres-denot*:
fixes *e*::*exp* **assumes** *e-ep*: $e \longrightarrow* e'$ **and** *v-ep*: $v \in E\ e'\ \varrho$ **shows** $v \in E\ e\ \varrho$
⟨*proof*⟩

13.3 Completeness

theorem *completeness*:
assumes *ev*: $e \longrightarrow* v$**and** *vv*: *is-val v*
shows $\exists\ v'.\ v' \in E\ e\ \varrho \wedge v' \in E\ v\ []$
⟨*proof*⟩

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

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

theorem *complete-wrt-op-sem*:
assumes *e-n*: $e \Downarrow ONat\ n$ **shows** $E\ e\ [] = E\ (ENat\ n)\ []$
⟨*proof*⟩

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*]: $E\ e\ = E\ e' \Longrightarrow E\ (ELam\ x\ e) = E\ (ELam\ x\ e')$
⟨*proof*⟩

lemma *e-app-cong*[*cong*]: $[\![\ E\ e1 = E\ e1';\ E\ e2 = E\ e2'\ ]\!] \Longrightarrow E\ (EApp\ e1\ e2) = E\ (EApp\ e1'\ e2')$
⟨*proof*⟩

lemma *e-prim-cong*[*cong*]: $[\![\ E\ e1 = E\ e1';\ E\ e2 = E\ e2'\ ]\!] \Longrightarrow E(EPrim\ f\ e1\ e2) = E(EPrim\ f\ e1'\ e2')$
⟨*proof*⟩

lemma *e-if-cong*[*cong*]: $[\![\ E\ e1 = E\ e1';\ E\ e2 = E\ e2';\ E\ e3 = E\ e3'\ ]\!]$
$\Longrightarrow E\ (EIf\ e1\ e2\ e3) = E\ (EIf\ e1'\ e2'\ e3')$
⟨*proof*⟩

datatype *ctx* = *CHole* | *CLam name ctx* | *CAppL ctx exp* | *CAppR exp ctx*
| *CPrimL nat* ⇒ *nat* ⇒ *nat ctx exp* | *CPrimR nat* ⇒ *nat* ⇒ *nat exp ctx*
| *CIf1 ctx exp exp* | *CIf2 exp ctx exp* | *CIf3 exp exp ctx*

fun *plug* :: *ctx* ⇒ *exp* ⇒ *exp* **where**
plug CHole e = *e* |
plug (*CLam x C*) *e* = *ELam x* (*plug C e*) |
plug (*CAppL C e2*) *e* = *EApp* (*plug C e*) *e2* |
plug (*CAppR e1 C*) *e* = *EApp e1* (*plug C e*) |
plug (*CPrimL f C e2*) *e* = *EPrim f* (*plug C e*) *e2* |

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

lemma *congruence*: $E\ e = E\ e' \Longrightarrow E\ (plug\ C\ e) = E\ (plug\ C\ e')$
⟨*proof*⟩

14.2 Auxiliary lemmas

lemma *diverge-denot-empty*: **assumes** *d*: *diverge e* **and** *fve*: $FV\ e = \{\}$ **shows** $E\ e\ [] = \{\}$
⟨*proof*⟩

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

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

lemma *val-ty-observe*:
⟦ $A \in E\ v\ []$; $A \in E\ v'\ []$;
observe v ob; *isval v'*; *isval v* ⟧ ⟹ *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-e1*: $FV\ (plug\ C\ e1) = \{\}$ **and** *fv-e2*: $FV\ (plug\ C\ e2) = \{\}$
shows $run\ (plug\ C\ e1)\ ob \longrightarrow run\ (plug\ C\ e2)\ ob$
⟨*proof*⟩

definition *ctx-equiv* :: *exp* ⇒ *exp* ⇒ *bool* (**infix** ≃ *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 $E(EApp\ (ELam\ x\ e)\ v) = E\ (subst\ x\ v\ e)$
⟨*proof*⟩

end

15 Correctness of an optimizer

theory *Optimizer*
imports *Lambda DenotEqualitiesFSet*
begin

fun *is-value* :: *exp* ⇒ *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* ⟹ *isval e* ∧ *FV e* = {}
⟨*proof*⟩

fun *opt* :: *exp* ⇒ *nat* ⇒ *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 ⇒ *if is-value e2′ then opt* (*subst x e2′ e*) *k*
else EApp e1′ e2′
| - ⇒ *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*) ⇒ *ENat* (*f n1 n2*)
| - ⇒ *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 ⇒ *if n* = *0 then e3′ else e2′*
| - ⇒ *EIf e1′ e2′ e3′*))

lemma *opt-correct-aux*: *E e* = *E* (*opt e k*)
⟨*proof*⟩

theorem *opt-correct*: *e* ≃ *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* | *TNat* | *Fun ty ty* (**infix** $\rightarrow$ *60*) | *Forall ty*

datatype *exp* = *EVar name* | *ENat nat* | *ELam ty exp* | *EApp exp exp*
| *EAbs exp* | *EInst exp ty* | *EFix ty exp*

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

fun *val-le* :: *val* $\Rightarrow$ *val* $\Rightarrow$ *bool* (**infix** $\sqsubseteq$ *52*) **where**
(*VNat n*) $\sqsubseteq$ (*VNat n'*) = (*n* = *n'*) |
(*Fun f*) $\sqsubseteq$ (*Fun f'*) = (*fset f* $\subseteq$ *fset f'*) |
(*Abs None*) $\sqsubseteq$ (*Abs None*) = *True* |
Abs (*Some v*) $\sqsubseteq$ *Abs* (*Some v'*) = *v* $\sqsubseteq$ *v'* |
Wrong $\sqsubseteq$ *Wrong* = *True* |
(*v*::*val*) $\sqsubseteq$ *v'* = *False*

16.2 Set monad

definition *set-bind* :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'b set*) $\Rightarrow$ *'b set* **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* $\rightleftharpoons$ *CONST set-bind E* (λ*P*. *F*)

definition *errset-bind* :: *val set* $\Rightarrow$ (*val* $\Rightarrow$ *val set*) $\Rightarrow$ *val set* **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* $\rightleftharpoons$ *CONST errset-bind E* (λ*P*. *F*)

definition *return* :: *val* $\Rightarrow$ *val set* **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* **where**
iterate-none[*intro*!]: *iterate Ee* ϱ (*Fun* {||}) |
iterate-again[*intro*!]: $[\![$ *iterate Ee* ϱ *f*; *f'* $\in$ *Ee* (*f*#ϱ) $]\!]$ $\Longrightarrow$ *iterate Ee* ϱ *f'*

abbreviation *apply-fun* :: *val set* $\Rightarrow$ *val set* $\Rightarrow$ *val set* **where**
apply-fun V1 V2 $\equiv$ (*v1* := *V1*; *v2* := *V2*;
case v1 of Fun f $\Rightarrow$
(*v2'*,*v3'*) $\leftarrow$ *fset f*;
if v2' $\sqsubseteq$ *v2 then return v3' else* {}
| - $\Rightarrow$ *return Wrong*)

fun *E* :: *exp* $\Rightarrow$ *env* $\Rightarrow$ *val set* **where**
Enat: *E* (*ENat n*) ϱ = *return* (*VNat n*) |
Evar: *E* (*EVar n*) ϱ = *return* (ϱ!*n*) |
Elam: *E* (*ELam* τ *e*) ϱ = {*v*. $\exists$ *f*. *v* = *Fun f* $\wedge$ ($\forall$ *v1 v2'*. (*v1*,*v2'*) $\in$ *fset f* $\longrightarrow$
($\exists$ *v2*. *v2* $\in$ *E e* (*v1*#ϱ) $\wedge$ *v2'* $\sqsubseteq$ *v2*)) } |
Eapp: *E* (*EApp e1 e2*) ϱ = *apply-fun* (*E e1* ϱ) (*E e2* ϱ) |
Efix: *E* (*EFix* τ *e*) ϱ = { *v*. *iterate* (*E e*) ϱ *v* } |
Eabs: *E* (*EAbs e*) ϱ = {*v*. ($\exists$ *v'*. *v* = *Abs* (*Some v'*) $\wedge$ *v'* $\in$ *E e* ϱ)
$\vee$ (*v* = *Abs None* $\wedge$ *E e* ϱ = {}) } |
Einst: *E* (*EInst e* τ) ϱ =

```
(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 × nat) list × nat

definition wf-tyvar :: tyctx ⇒ nat ⇒ bool where
  wf-tyvar Γ n ≡ n < snd Γ
definition push-ty :: ty ⇒ tyctx ⇒ tyctx where
  push-ty τ Γ ≡ ((τ,snd Γ) # fst Γ, snd Γ)
definition push-tyvar :: tyctx ⇒ tyctx where
  push-tyvar Γ ≡ (fst Γ, Suc (snd Γ))

definition good-ctx :: tyctx ⇒ bool where
  good-ctx Γ ≡ ∀ n. n < length (fst Γ) ⟶ snd ((fst Γ) ! n) ≤ snd Γ

definition lookup :: tyctx ⇒ nat ⇒ ty option where
  lookup Γ n ≡ (if n < length (fst Γ) then
                  let k = snd Γ − snd ((fst Γ)!n) in
                  Some (shift k 0 (fst ((fst Γ)!n)))
                else None)

inductive well-typed :: tyctx ⇒ exp ⇒ ty ⇒ bool (- ⊢ - : - [55,55,55] 54) where
  wtnat[intro!]: Γ ⊢ ENat n : TNat |
  wtvar[intro!]: ⟦ lookup Γ n = Some τ ⟧ ⟹ Γ ⊢ EVar n : τ |
  wtapp[intro!]: ⟦ Γ ⊢ e : σ → τ; Γ ⊢ e' : σ ⟧ ⟹ Γ ⊢ EApp e e' : τ |
  wtlam[intro!]: ⟦ push-ty σ Γ ⊢ e : τ ⟧ ⟹ Γ ⊢ ELam σ e : σ → τ |
  wtfix[intro!]: ⟦ push-ty (σ→τ) Γ ⊢ e : σ→τ ⟧ ⟹ Γ ⊢ EFix (σ → τ) e : σ → τ |
  wtabs[intro!]: ⟦ push-tyvar Γ ⊢ e : τ ⟧ ⟹ Γ ⊢ EAbs e : Forall τ |
  wtinst[intro!]: ⟦ Γ ⊢ e : Forall τ ⟧ ⟹ Γ ⊢ EInst e σ : (subst 0 σ τ)
```

inductive *wfenv* :: *env* ⇒ *tyenv* ⇒ *tyctx* ⇒ *bool* (⊢ -,- : - [*55*,*55*,*55*] *54*) **where**
wfnil[*intro*!]: ⊢ [],[] : ([],*0*) |
wfvbind[*intro*!]: ⟦ ⊢ ϱ,η : Γ; *v* ∈ *T* τ η ⟧ ⟹ ⊢ (*v*#ϱ),η : *push-ty* τ Γ |
wftbind[*intro*!]: ⟦ ⊢ ϱ,η : Γ ⟧ ⟹ ⊢ ϱ, (*V*#η) : *push-tyvar* Γ

inductive-cases
wtnat-inv[*elim*!]: Γ ⊢ *ENat n* : τ **and**
wtvar-inv[*elim*!]: Γ ⊢ *EVar n* : τ **and**
wtapp-inv[*elim*!]: Γ ⊢ *EApp e e′* : τ **and**
wtlam-inv[*elim*!]: Γ ⊢ *ELam* σ *e* : τ **and**
wtfix-inv[*elim*!]: Γ ⊢ *EFix* σ *e* : τ **and**
wtabs-inv[*elim*!]: Γ ⊢ *EAbs e* : τ **and**
wtinst-inv[*elim*!]: Γ ⊢ *EInst e* σ : τ

lemma *wfenv-good-ctx*: ⊢ ϱ,η : Γ ⟹ *good-ctx* Γ
⟨*proof*⟩

16.6 Well-typed Programs don't go wrong

lemma *nth-append1*[*simp*]: *n* < *length* ϱ*1* ⟹ (ϱ*1*@ϱ*2*)!*n* = ϱ*1*!*n*
⟨*proof*⟩

lemma *nth-append2*[*simp*]: *n* ≥ *length* ϱ*1* ⟹ (ϱ*1*@ϱ*2*)!*n* = ϱ*2*!(*n* − *length* ϱ*1*)
⟨*proof*⟩

lemma *shift-append-preserves-T-aux*:
shows *T* τ (ϱ*1*@ϱ*3*) = *T* (*shift* (*length* ϱ*2*) (*length* ϱ*1*) τ) (ϱ*1*@ϱ*2*@ϱ*3*)
⟨*proof*⟩

lemma *shift-append-preserves-T*: **shows** *T* τ ϱ*3* = *T* (*shift* (*length* ϱ*2*) *0* τ) (ϱ*2*@ϱ*3*)
⟨*proof*⟩

lemma *drop-shift-preserves-T*:
assumes *k*: *k* ≤ *length* ϱ **shows** *T* τ (*drop k* ϱ) = *T* (*shift k 0* τ) ϱ
⟨*proof*⟩

lemma *shift-cons-preserves-T*: **shows** *T* τ ϱ = *T* (*shift* (*Suc 0*) *0* τ) (*b*#ϱ)
⟨*proof*⟩

lemma *compose-shift*: **shows** *shift* (*j*+*k*) *c* τ = *shift j c* (*shift k c* τ)
⟨*proof*⟩

lemma *shift-zero-id*[*simp*]: *shift 0 c* τ = τ
⟨*proof*⟩

lemma *lookup-wfenv*: **assumes** *r-g*: ⊢ ϱ,η : Γ **and** *ln*: *lookup* Γ *n* = *Some* τ
shows ∃ *v*. ϱ!*n* = *v* ∧ *v* ∈ *T* τ η
⟨*proof*⟩

lemma *less-wrong*[*elim*!]: ⟦ *v* ⊑ *Wrong*; *v* = *Wrong* ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *less-nat*[*elim*!]: ⟦ *v* ⊑ *VNat n*; *v* = *VNat n* ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *less-fun*[*elim*!]: ⟦ *v* ⊑ *Fun f*; ⋀ *f′*. ⟦ *v* = *Fun f′*; *fset f′* ⊆ *fset f* ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *less-refl*[*simp*]: *v* ⊑ *v*
⟨*proof*⟩

lemma *less-trans*: **fixes** *v1*::*val* **and** *v2*::*val* **and** *v3*::*val*
shows ⟦ *v1* ⊑ *v2*; *v2* ⊑ *v3* ⟧ ⟹ *v1* ⊑ *v3*
⟨*proof*⟩

lemma *T-down-closed*: **assumes** *vt*: *v* ∈ *T* τ η **and** *vp-v*: *v'* ⊑ *v*
shows *v'* ∈ *T* τ η
⟨*proof*⟩

lemma *wrong-not-in-T*: *Wrong* ∉ *T* τ η
⟨*proof*⟩

lemma *fun-app*: **assumes** *vmn*: *V* ⊆ *T* (*m* → *n*) η **and** *v2s*: *V'* ⊆ *T m* η
shows *apply-fun V V'* ⊆ *T n* η
⟨*proof*⟩

lemma *T-eta*: {*v*. ∃ *v'*. *v'* ∈ *T* σ (η) ∧ *v* ⊑ *v'* ∧ *v* ≠ *Wrong*} = *T* σ η
⟨*proof*⟩

lemma *compositionality*: *T* τ (η1 @ (*T* σ (η1@η2)) # η2) = *T* (*subst* (*length* η1) σ τ) (η1@η2)
⟨*proof*⟩

lemma *iterate-sound*:
assumes *it*: *iterate Ee* ϱ *v*
and *IH*: ∀ *v*. *v* ∈ *T* (σ→τ) η ⟶ *Ee* (*v*#ϱ) ⊆ *T* (σ→τ) η
shows *v* ∈ *T* (σ→τ) η ⟨*proof*⟩

theorem *welltyped-dont-go-wrong*:
assumes *wte*: Γ ⊢ *e* : τ **and** *wfr*: ⊢ ϱ,η : Γ
shows *E e* ϱ ⊆ *T* τ η
⟨*proof*⟩

end

17 Semantics of mutable references

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

datatype *ty* = *TNat* | *TFun ty ty* (**infix** → *60*) | *TPair ty ty* | *TRef ty*

type-synonym *name* = *nat*

datatype *exp* = *EVar name* | *ENat nat* | *ELam ty exp* | *EApp exp exp*
| *EPrim nat* ⇒ *nat* ⇒ *nat exp exp* | *EIf exp exp exp*
| *EPair exp exp* | *EFst exp* | *ESnd exp*
| *ERef exp* | *ERead exp* | *EWrite exp exp*

17.1 Denotations (values)

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

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

inductive *val-le* :: *val* ⇒ *val* ⇒ *bool* (**infix** ⊑ *52*) **where**
vnat-le[*intro*!]: (*VNat n*) ⊑ (*VNat n*) |
vaddr-le[*intro*!]: (*VAddr a*) ⊑ (*VAddr a*) |
wrong-le[*intro*!]: *Wrong* ⊑ *Wrong* |

vfun-le[*intro*!]: *t1* |⊆| *t2* ⟹ (*VFun t1*) ⊑ (*VFun t2*) |
vpair-le[*intro*!]: ⟦ *v1* ⊑ *v1'*; *v2* ⊑ *v2'* ⟧ ⟹ (*VPair v1 v2*) ⊑ (*VPair v1' v2'*)

primrec *vsize* :: *val* ⇒ *nat* **where**
vsize (*VNat n*) = *1* |
vsize (*VFun t*) = *1* + *ffold* (λ((-,*v*), (-,*u*)).λ*r*. *v* + *u* + *r*) *0*
(*fimage* (*map-prod* (λ *v*. (*v*,*vsize v*)) (λ *v*. (*v*,*vsize v*))) *t*) |
vsize (*VPair v1 v2*) = *1* + *vsize v1* + *vsize v2* |
vsize (*VAddr a*) = *1* |
vsize Wrong = *1*

17.2 Non-deterministic state monad

type-synonym *'a M* = *store* ⇒ (*'a* × *store*) *set*

definition *bind* :: *'a M* ⇒ (*'a* ⇒ *'b M*) ⇒ *'b M* **where**
bind m f μ1 ≡ { (*v*,*μ3*). ∃ *v' μ2*. (*v'*,*μ2*) ∈ *m μ1* ∧ (*v*,*μ3*) ∈ *f v' μ2* }
declare *bind-def*[*simp*]

syntax -*bind* :: [*pttrns*,*'a M*,*'b*] ⇒ *'c* ((- ← -;//-) *0*)
translations *P* ← *E*; *F* ⇌ *CONST bind E* (λ*P*. *F*)

no-notation *binomial* (**infix** *choose 64*)

definition *choose* :: *'a set* ⇒ *'a M* **where**
choose S μ ≡ {(*a*,*μ1*). *a* ∈ *S* ∧ *μ1*=*μ*}
declare *choose-def*[*simp*]

definition *return* :: *'a* ⇒ *'a M* **where**
return v μ ≡ { (*v*,*μ*) }
declare *return-def*[*simp*]

definition *zero* :: *'a M* **where**
zero μ ≡ {}
declare *zero-def*[*simp*]

definition *err-bind* :: *val M* ⇒ (*val* ⇒ *val M*) ⇒ *val M* **where**
err-bind m f ≡ (*x* ← *m*; *if x* = *Wrong then return Wrong else f x*)
declare *err-bind-def*[*simp*]

syntax -*errset-bind* :: [*pttrns*,*val M*,*val*] ⇒ *'c* ((- := -;//-) *0*)
translations *P* := *E*; *F* ⇌ *CONST err-bind E* (λ*P*. *F*)

definition *down* :: *val* ⇒ *val M* **where**
down v μ1 ≡ {(*v'*,*μ*). *v'* ⊑ *v* ∧ *μ* = *μ1* }
declare *down-def*[*simp*]

definition *get-store* :: *store M* **where**
get-store μ ≡ { (*μ*,*μ*) }
declare *get-store-def*[*simp*]

definition *put-store* :: *store* ⇒ *unit M* **where**
put-store μ ≡ λ-. { ((),*μ*) }
declare *put-store-def*[*simp*]

definition *mapM* :: *'a fset* ⇒ (*'a* ⇒ *'b M*) ⇒ (*'b fset*) *M* **where**
mapM as f ≡ *ffold* (λ*a*. λ*r*. (*b* ← *f a*; *bs* ← *r*; *return* (*finsert b bs*))) (*return* {||}) *as*

definition *run* :: *store* ⇒ *val M* ⇒ (*val* × *store*) *set* **where**
run σ m ≡ *m σ*
declare *run-def*[*simp*]

```
definition sdom :: store ⇒ nat set where
  sdom μ ≡ {a. ∃ v. (VAddr a,v) ∈ fset μ }

definition max-addr :: store ⇒ nat where
  max-addr μ = ffold (λa.λr. case a of (VAddr n,-) ⇒ max n r | - ⇒ r) 0 μ
```

17.3 Denotational semantics

```
abbreviation apply-fun :: val M ⇒ val M ⇒ val M where
  apply-fun V1 V2 ≡ (v1 := V1; v2 := V2;
                     case v1 of VFun f ⇒
                       (p, p') ← choose (fset f); μ0 ← get-store;
                       (case (p,p') of (VPair v (VFun μ), VPair v' (VFun μ')) ⇒
                         if v ⊑ v2 ∧ (VFun μ) ⊑ (VFun μ0) then (- ← put-store μ'; down v')
                         else zero
                       | - ⇒ zero)
                     | - ⇒ return Wrong)

fun nvals :: nat ⇒ (val fset) M where
  nvals 0 = return {||} |
  nvals (Suc k) = (v ← choose UNIV; L ← nvals k; return (finsert v L))

definition vals :: (val fset) M where
  vals ≡ (n ← choose UNIV; nvals n)
declare vals-def[simp]

fun npairs :: nat ⇒ func M where
  npairs 0 = return {||} |
  npairs (Suc k) = (v ← choose UNIV; v' ← choose {v::val. True};
                    P ← npairs k; return (finsert (v,v') P))

definition tables :: func M where
  tables ≡ (n ← choose {k::nat. True}; npairs n)
declare tables-def[simp]

definition read :: nat ⇒ val M where
  read a ≡ (μ ← get-store; if a ∈ sdom μ then
                       ((v1,v2) ← choose (fset μ); if v1 = VAddr a then return v2 else zero)
                     else return Wrong)
declare read-def[simp]

definition update :: nat ⇒ val ⇒ val M where
  update a v ≡ (μ ← get-store;
                - ← put-store (finsert (VAddr a,v) (ffilter (λ(v,v'). v ≠ VAddr a) μ));
                return (VAddr a))
declare update-def[simp]

type-synonym env = val list

fun E :: exp ⇒ env ⇒ val M where
  Enat: E (ENat n) ϱ = return (VNat n) |
  Evar: E (EVar n) ϱ = (if n < length ϱ then down (ϱ!n) else return Wrong) |
  Elam: E (ELam A e) ϱ = (L ← vals;
                          t ← mapM L (λ v. (μ ← tables; (v',μ') ← choose (run μ (E e (v#ϱ)));
                                  return (VPair v (VFun μ),VPair v' (VFun μ'))));
                          return (VFun t)) |
  Eapp: E (EApp e1 e2) ϱ = apply-fun (E e1 ϱ) (E e2 ϱ) |
  Eprim: E (EPrim f e1 e2) ϱ = (v1 := E e1 ϱ; v2 := E e2 ϱ;
                                case (v1, v2) of (VNat n1,VNat n2) ⇒ return (VNat (f n1 n2))
                                | - ⇒ return Wrong) |
```

Eif: *E* (*EIf e1 e2 e3*) *ϱ* = (*v1* := *E e1 ϱ*; *case v1 of VNat n* ⇒ (*if n* = *0 then E e3 ϱ else E e2 ϱ*)
| - ⇒ *return Wrong*) |
Epair: *E* (*EPair e1 e2*) *ϱ* = (*v1* := *E e1 ϱ*; *v2* := *E e2 ϱ*; *return* (*VPair v1 v2*)) |
Efst: *E* (*EFst e*) *ϱ* = (*v*:=*E e ϱ*; *case v of VPair v1 v2* ⇒ *return v1* | - ⇒ *return Wrong*) |
Esnd: *E* (*ESnd e*) *ϱ* = (*v*:=*E e ϱ*; *case v of VPair v1 v2* ⇒ *return v2* | - ⇒ *return Wrong*) |
Eref: *E* (*ERef e*) *ϱ* = (*v*:=*E e ϱ*; *μ* ← *get-store*; *a* ← *choose UNIV*;
if a ∈ *sdom μ then zero*
else (- ← *put-store* (*finsert* (*VAddr a*,*v*) *μ*);
return (*VAddr a*))) |
Eread: *E* (*ERead e*) *ϱ* = (*v* := *E e ϱ*; *case v of VAddr a* ⇒ *read a* | - ⇒ *return Wrong*) |
Ewrite: *E* (*EWrite e1 e2*) *ϱ* = (*v1* := *E e1 ϱ*; *v2* := *E e2 ϱ*;
case v1 of VAddr a ⇒ *update a v2* | - ⇒ *return Wrong*)

end
theory *MutableRefProps*
imports *MutableRef*
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* **and**
le-pair-any-inv[*elim*!]: *VPair v1 v2* ⊑ *v* **and**
le-any-pair-inv[*elim*!]: *v* ⊑ *VPair v1 v2* **and**
le-addr-any-inv[*elim*!]: *VAddr a* ⊑ *v* **and**
le-any-addr-inv[*elim*!]: *v* ⊑ *VAddr a* **and**
le-wrong-any-inv[*elim*!]: *Wrong* ⊑ *v* **and**
le-any-wrong-inv[*elim*!]: *v* ⊑ *Wrong*

proposition *val-le-refl*: *v* ⊑ *v* ⟨*proof*⟩

proposition *val-le-trans*: ⟦ *v1* ⊑ *v2*; *v2* ⊑ *v3* ⟧ ⟹ *v1* ⊑ *v3*
⟨*proof*⟩

proposition *val-le-antisymm*: ⟦ *v1* ⊑ *v2*; *v2* ⊑ *v1* ⟧ ⟹ *v1* = *v2*
⟨*proof*⟩

end