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

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 (*rule classical*)
  assume ¬ *stuck e*
  from *this* **obtain** *e'* **where** *red*: *e* ⟶ *e'* **by** *auto*
  from *val-e red* **have** *False* **by** (*case-tac e*) *auto*
  from *this* **show** *?thesis* **..**
qed

lemma *subst-fv-aux*: **assumes** *fvv*: *FV v* = {} **shows** *FV* (*subst x v e*) ⊆ *FV e* − {*x*}
  using *fvv*
proof (*induction e arbitrary*: *x v rule*: *exp.induct*)
  case (*EVar x*)
  then show *?case* **by** *auto*

next
case (*ENat x*)
then show *?case* **by** *auto*
next
case (*ELam y e*)
then show *?case* **by** (*cases x = y*) *auto*
qed (*simp*,*blast*)+

lemma *subst-fv*: **assumes** *fv-e*: $FV\ e \subseteq \{x\}$ **and** *fv-v*: $FV\ v = \{\}$
shows $FV\ (subst\ x\ v\ e) = \{\}$
using *fv-e fv-v subst-fv-aux* **by** *blast*

lemma *red-pres-fv*: **fixes** *e*::*exp* **assumes** *red*: $e \longrightarrow e'$ **and** *fv*: $FV\ e = \{\}$ **shows** $FV\ e' = \{\}$
using *red fv*
proof (*induction rule*: *reduce.induct*)
case (*beta v x e*)
then show *?case* **using** *subst-fv* **by** *auto*
qed *fastforce*+

lemma *reduction-pres-fv*: **fixes** *e*::*exp* **assumes** *r*: $e \longrightarrow* e'$ **and** *fv*: $FV\ e = \{\}$ **shows** $FV\ e' = \{\}$
using *r fv*
proof (*induction*)
case (*ms-nil e*)
then show *?case* **by** *blast*
next
case (*ms-cons e1 e2 e3*)
then show *?case* **using** *red-pres-fv* **by** *auto*
qed

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

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

lemma *lookup-bs-env*: ⟦ *bs-env* ϱ ϱ'; *lookup* ϱ *x* = *Some w* ⟧ ⟹
∃ *v*. *lookup* ϱ' *x* = *Some v* ∧ *bs-val w v*
by (*induction* ϱ *arbitrary*: ϱ' *x w*) *auto*

lemma *app-red-cong1*: *e1* ⟶* *e1'* ⟹ *EApp e1 e2* ⟶* *EApp e1' e2*
by (*induction rule*: *multi-step.induct*) *blast+*

lemma *app-red-cong2*: *e2* ⟶* *e2'* ⟹ *EApp e1 e2* ⟶* *EApp e1 e2'*
by (*induction rule*: *multi-step.induct*) *blast+*

lemma *prim-red-cong1*: *e1* ⟶* *e1'* ⟹ *EPrim f e1 e2* ⟶* *EPrim f e1' e2*
by (*induction rule*: *multi-step.induct*) *blast+*

lemma *prim-red-cong2*: *e2* ⟶* *e2'* ⟹ *EPrim f e1 e2* ⟶* *EPrim f e1 e2'*
by (*induction rule*: *multi-step.induct*) *blast+*

lemma *if-red-cong1*: *e1* ⟶* *e1'* ⟹ *EIf e1 e2 e3* ⟶* *EIf e1' e2 e3*
by (*induction rule*: *multi-step.induct*) *blast+*

lemma *multi-step-trans*: ⟦ *e1* ⟶* *e2*; *e2* ⟶* *e3* ⟧ ⟹ *e1* ⟶* *e3*
proof (*induction arbitrary*: *e3 rule*: *multi-step.induct*)
case (*ms-cons e1 e2 e3 e3'*)
then have *e2* ⟶* *e3'* **by** *auto*
with *ms-cons*(*1*) **show** *?case* **by** *blast*
qed *blast*

lemma *subst-id-fv*: $x \notin FV\ e \Longrightarrow subst\ x\ v\ e = e$
by (*induction e arbitrary*: *x v*) *auto*

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

definition *closed-env* :: *env* ⇒ *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* ⇒ *env* ⇒ *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*]: $sdom\ ((x,EVar\ x)\#\varrho) = sdom\ \varrho - \{x\}$
unfolding *sdom-def* **by** *auto*

lemma *sdom-cons-v*[*simp*]: $FV\ v = \{\} \Longrightarrow sdom\ ((x,v)\#\varrho) = insert\ x\ (sdom\ \varrho)$
unfolding *sdom-def* **by** *auto*

lemma *lookup-some-in-dom*: $[\![\ lookup\ \varrho\ x = Some\ v;\ v \neq EVar\ x\ ]\!] \Longrightarrow x \in sdom\ \varrho$
proof (*induction* ϱ)
case (*Cons b* ϱ)
show *?case*
proof (*cases b*)
case (*Pair y v'*)
with *Cons* **show** *?thesis* **unfolding** *sdom-def* **by** *auto*
qed
qed *auto*

lemma *lookup-none-notin-dom*: $lookup\ \varrho\ x = None \Longrightarrow x \notin sdom\ \varrho$
proof (*induction* ϱ)
case (*Cons b* ϱ)
show *?case*
proof (*cases b*)
case (*Pair y v*)
with *Cons* **show** *?thesis* **unfolding** *sdom-def* **by** *auto*
qed
qed (*auto simp*: *sdom-def*)

lemma *psubst-change*: $equiv\text{-}env\ \varrho\ \varrho' \Longrightarrow psubst\ \varrho\ e = psubst\ \varrho'\ e$
proof (*induction e arbitrary*: $\varrho\ \varrho'$)
case (*EVar x*)
show *?case*
proof (*cases lookup* ϱ *x*)
case *None* **from** *None* **have** *lx*: *lookup* ϱ *x* = *None* **by** *simp*
show *?thesis*
proof (*cases lookup* ϱ' *x*)
case *None*
with *EVar lx* **show** *?thesis* **by** *auto*
next
case (*Some v*)
from *EVar lx Some* **have** $x \notin sdom\ \varrho'$ **unfolding** *equiv-env-def* **by** *auto*
with *lx Some* **show** *?thesis* **unfolding** *sdom-def* **by** *simp*
qed
next
case (*Some v*) **from** *Some* **have** *lx*: *lookup* ϱ *x* = *Some v* **by** *simp*
show *?thesis*
proof (*cases lookup* ϱ' *x*)
case *None*
from *EVar lx None* **have** $x \notin sdom\ \varrho$ **unfolding** *equiv-env-def* **by** *auto*
with *None Some* **show** *?thesis* **unfolding** *sdom-def* **by** *simp*

```
    next
      case (Some v')
      from EVar Some lx show ?thesis by (simp add: equiv-env-def sdom-def) force
    qed
  qed
next
  case (ELam x' e)
  from ELam(2) have equiv-env ((x',EVar x')#ϱ) ((x',EVar x')#ϱ') by (simp add: equiv-env-def)
  with ELam show ?case by (simp add: equiv-env-def)
qed fastforce+

lemma subst-psubst: ⟦ closed-env ϱ; FV v = {} ⟧ ⟹
    subst x v (psubst ((x, EVar x) # ϱ) e) = psubst ((x, v) # ϱ) e
proof (induction e arbitrary: x v ϱ)
  case (EVar x x' v ϱ)
  show ?case
  proof (cases x = x')
    case True
    then show ?thesis by force
  next
    case False from False have xxp: x ≠ x' by simp
    show ?thesis
    proof (cases lookup ϱ x)
      case None
      then show ?thesis by auto
    next
      case (Some v')
      show ?thesis
      proof (cases v' = EVar x)
        case True
        with Some show ?thesis by auto
      next
        case False
        from False Some have xdom: x ∈ sdom ϱ using lookup-some-in-dom by simp
        from this EVar Some have FV v' = {} using closed-env-def by blast
        from this Some show ?thesis using subst-id-fv by auto
      qed
    qed
  qed
next
  case (ELam x' e)
  show ?case
  proof (cases x = x')
    case True
    then show ?thesis apply simp apply (rule psubst-change)
      using equiv-env-def sdom-def by auto
  next
    case False
    then show ?thesis apply simp
    proof −
      assume x-xp: x ≠ x'
      let ?r = (x',EVar x') # ϱ
      from ELam have IHprem: closed-env ((x', EVar x') # ϱ) using closed-env-def by auto
      have psubst ((x',EVar x')#(x, EVar x)#ϱ) e = psubst ((x,EVar x)#(x',EVar x') # ϱ) e
        apply (rule psubst-change) using x-xp equiv-env-def by auto
      from this have subst x v (psubst ((x', EVar x') # (x, EVar x) # ϱ) e)
                   = subst x v (psubst ((x,EVar x)#(x',EVar x') # ϱ) e) by simp
      also with ELam IHprem have ... = psubst ((x,v)#(x',EVar x')#ϱ) e
        using ELam(1)[of (x',EVar x')#ϱ v x] by simp
      also have ... = psubst ((x',EVar x')#(x,v)#ϱ) e
        apply (rule psubst-change) using x-xp equiv-env-def sdom-def by auto
```

```
    finally show subst x v (psubst ((x', EVar x') # (x, EVar x) # ϱ) e)
                = psubst ((x', EVar x') # (x,v) # ϱ) e .
  qed
 qed
qed fastforce+

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

lemma bs-env-dom: bs-env ϱ ϱ' ⟹ set (map fst ϱ) = sdom ϱ'
proof (induction ϱ arbitrary: ϱ')
 case Nil
 then show ?case by (force simp: sdom-def)
next
 case (Cons b ϱ)
 then show ?case
 proof (cases b)
  case (Pair x v')
  with Cons show ?thesis by (cases v') force+
 qed
qed

lemma closed-env-cons[intro!]: FV v = {} ⟹ closed-env ϱ'' ⟹ closed-env ((a, v) # ϱ'')
 by (simp add: closed-env-def sdom-def)

lemma bs-env-closed: bs-env ϱ ϱ' ⟹ closed-env ϱ'
proof (induction ϱ arbitrary: ϱ')
 case Nil
 then show ?case by (force simp: closed-env-def)
next
 case (Cons b ϱ)
 from Cons obtain x v v' ϱ'' where b: b = (x,v) and rp: ϱ' = (x,v')#ϱ''
  and vvp: bs-val v v' and r-rpp: bs-env ϱ ϱ'' by (cases b) blast
 from vvp have is-val v' by blast
 from this have fv-vp: FV v' = {} by auto
 from Cons r-rpp have closed-env ϱ'' by blast
 from this rp fv-vp show ?case by blast
qed

lemma psubst-fv: closed-env ϱ ⟹ FV (psubst ϱ e) = FV e − sdom ϱ
proof (induction e arbitrary: ϱ)
 case (EVar x)
 then show ?case
  apply (simp add: closed-env-def)
  apply (cases x ∈ sdom ϱ)
   apply (erule-tac x=x in allE)
   apply (erule impE) apply blast apply (simp add: sdom-def) apply clarify
   apply force
  apply (simp add: sdom-def)
  apply (cases lookup ϱ x)
   apply force
  apply force
  done
next
 case (ELam x e)
 from ELam have closed-env ((x, EVar x) # ϱ) by (simp add: closed-env-def sdom-def)
 from this ELam show ?case by auto
qed fastforce+

lemma big-small-step:
 assumes ev: ϱ ⊢ e ⇓ 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
```

using *ev r-rp fv-e*
proof (*induction arbitrary*: ϱ' *rule*: *eval.induct*)
case (*eval-nat* ϱ *n* ϱ')
then show *?case* **by** (*rule-tac x=ENat n* **in** *exI*) *auto*
next
case (*eval-var* ϱ *x w* ϱ')
from *eval-var* **obtain** *v* **where** *lx*: *lookup* ϱ' *x* = *Some v* **and**
vv: *is-val v* **and** *w-v*: *bs-val w v* **using** *lookup-bs-env* **by** *blast*
from *lx vv w-v* **show** *?case* **by** (*rule-tac x=v* **in** *exI*) *auto*
next
case (*eval-lam* ϱ *x e* ϱ')
from *eval-lam*(*1*) **have** *dom-eq*: *set* (*map fst* ϱ) = *sdom* ϱ' **using** *bs-env-dom* **by** *blast*
from *eval-lam*(*1*) **have** *closed-env* ((*x*,*EVar x*)#ϱ') **using** *bs-env-closed closed-env-def* **by** *auto*
from *this psubst-fv* **have** *FV* (*psubst* ((*x*,*EVar x*)#ϱ') *e*) = *FV e* − *sdom* ((*x*,*EVar x*)#ϱ') **by** *blast*
from *this eval-lam*(*2*) *dom-eq*
have *fv-lam*: *FV* (*ELam x* (*psubst* ((*x*,*EVar x*)#ϱ') *e*)) = {} **by** *auto*
from *fv-lam eval-lam* **have** *1*: *bs-val* (*BClos x e* ϱ) (*ELam x* (*psubst* ((*x*, *EVar x*) # ϱ') *e*)) **by** *auto*
from *this eval-lam fv-lam* **show** *?case*
by (*rule-tac x=ELam x* (*psubst* ((*x*,*EVar x*)#ϱ') *e*) **in** *exI*) *auto*
next
case (*eval-app* ϱ *e1 x e* ϱ' *e2 arg v* ϱ'')
from *eval-app*(*8*) **have** *FV e1* ⊆ *set* (*map fst* ϱ) **by** *auto*
from *this eval-app*(*7*) *eval-app*(*4*)[*of* ϱ''] **obtain** *v1* **where** *e1-v1*: *psubst* ϱ'' *e1* ⟶∗ *v1* **and**
vv1: *is-val v1* **and** *clos-v1*: *bs-val* (*BClos x e* ϱ') *v1* **by** (*simp*, *blast*)
from *eval-app*(*8*) **have** *FV e2* ⊆ *set* (*map fst* ϱ) **by** *auto*
from *this eval-app*(*5*) *eval-app*(*7*) **obtain** *v2* **where** *e2-v2*: *psubst* ϱ'' *e2* ⟶∗ *v2* **and**
vv2: *is-val v2* **and** *arg-v2*: *bs-val arg v2* **by** *blast*
from *vv2* **have** *fv-v2*: *FV v2* = {} **by** *auto*
from *clos-v1* **obtain** $\varrho 2$ **where** *rpp-r2*: *bs-env* ϱ' $\varrho 2$ **and** *fv-v1*: *FV v1* = {} **and**
v1-lam: *v1* = *ELam x* (*psubst* ((*x*,*EVar x*)#$\varrho 2$) *e*) **by** *auto*
let *?r* = ((*x*,*v2*) # $\varrho 2$)
from *rpp-r2* **have** *cr2*: *closed-env* $\varrho 2$ **using** *bs-env-closed* **by** *auto*
from *this* **have** *closed-env* ((*x*,*EVar x*)#$\varrho 2$) **using** *closed-env-def sdom-def* **by** *auto*
from *this* **have** *fve*: *FV* (*psubst* ((*x*,*EVar x*)#$\varrho 2$) *e*) = *FV e* − *sdom* ((*x*,*EVar x*)#$\varrho 2$)
using *psubst-fv*[*of* (*x*,*EVar x*)#$\varrho 2$] **by** *blast*
let *?r2* = ((*x*, *arg*) # ϱ')
from *rpp-r2 arg-v2 vv2* **have** *rr*: *bs-env ?r2 ?r* **by** *auto*
from *rr bs-env-dom* **have** *dr2-dr*: *set* (*map fst ?r2*) = *sdom ?r* **by** *blast*
from *fve dr2-dr fv-v1 v1-lam fv-v2* **have** *FV e* ⊆ *set* (*map fst* ((*x*, *arg*) # ϱ')) **by** *auto*
from *this rr eval-app*(*6*) **obtain** *v3* **where** *e-v3*: *psubst ?r e* ⟶∗ *v3* **and**
vv3: *isval v3* **and** *v-v3*: *bs-val v v3* **by** (*simp*, *blast*)
from *e1-v1* **have** *1*: *EApp* (*psubst* ϱ'' *e1*) (*psubst* ϱ'' *e2*) ⟶∗ *EApp v1* (*psubst* ϱ'' *e2*)
by (*rule app-red-cong1*)
from *e2-v2* **have** *2*: *EApp v1* (*psubst* ϱ'' *e2*) ⟶∗ *EApp v1 v2*
by (*rule app-red-cong2*)
from *vv2 fv-v2* **have** *vv2b*: *is-val v2* **by** *auto*
let *?body* = *psubst* ((*x*,*EVar x*)#$\varrho 2$) *e*
from *v1-lam vv2b* **have** *3*: *EApp* (*ELam x ?body*) *v2* ⟶
subst x v2 (*psubst* ((*x*,*EVar x*)#$\varrho 2$) *e*) **using** *beta*[*of v2 x ?body*] **by** *simp*
have *4*: *subst x v2* (*psubst* ((*x*,*EVar x*)#$\varrho 2$) *e*) = *psubst ?r e*
apply (*rule subst-psubst*) **using** *fv-v2 cr2* **by** *auto*
have *4*: *subst x v2* (*psubst* ((*x*,*EVar x*)#$\varrho 2$) *e*) = *psubst ?r e*
apply (*rule subst-psubst*) **using** *fv-v2 cr2* **by** *auto*
from *1 2* **have** *5*: *psubst* ϱ'' (*EApp e1 e2*) ⟶∗ *EApp v1 v2* **apply** *simp*
by (*rule multi-step-trans*) *auto*
from *5 3 4 v1-lam* **have** *6*: *psubst* ϱ'' (*EApp e1 e2*) ⟶∗ *psubst ?r e*
apply *simp* **apply** (*rule multi-step-trans*) **apply** *assumption* **apply** *blast* **done**
from *6 e-v3* **have** *7*: *psubst* ϱ'' (*EApp e1 e2*) ⟶∗ *v3* **by** (*rule multi-step-trans*)
from *7 vv3 v-v3* **show** *?case* **by** *blast*
next
case (*eval-prim* ϱ *e1 n1 e2 n2 n3 f* ϱ')

```
  from eval-prim(7) have FV e1 ⊆ set (map fst ϱ) by auto
  from this eval-prim obtain v1 where e1-v1: psubst ϱ' e1 ⟶* v1 and
    n1-v1: bs-val (BNat n1) v1 by blast
  from n1-v1 have v1: v1 = ENat n1 by blast

  from eval-prim(7) have FV e2 ⊆ set (map fst ϱ) by auto
  from this eval-prim obtain v2 where e2-v2: psubst ϱ' e2 ⟶* v2 and
    n2-v2: bs-val (BNat n2) v2 by blast
  from n2-v2 have v2: v2 = ENat n2 by blast

  from e1-v1 have 1: EPrim f (psubst ϱ' e1) (psubst ϱ' e2) ⟶* EPrim f v1 (psubst ϱ' e2)
    by (rule prim-red-cong1)
  from e2-v2 have 2: EPrim f v1 (psubst ϱ' e2) ⟶* EPrim f v1 v2
    by (rule prim-red-cong2)
  from v1 v2 have 3: EPrim f v1 v2 ⟶ ENat (f n1 n2) by auto
  from 1 2 have 5: psubst ϱ' (EPrim f e1 e2) ⟶* EPrim f v1 v2 apply simp
    apply (rule multi-step-trans) apply auto done
  from 5 3  have 6: psubst ϱ' (EPrim f e1 e2) ⟶* ENat (f n1 n2) apply simp
    apply (rule multi-step-trans) apply assumption apply blast done
  from this eval-prim(3) show ?case apply (rule-tac x=ENat (f n1 n2) in exI) by auto
next
  case (eval-if0 ϱ e1 e3 v3 e2 ϱ')
  from eval-if0(6) have FV e1 ⊆ set (map fst ϱ) by auto
  from this eval-if0 obtain v1 where e1-v1: psubst ϱ' e1 ⟶* v1 and
    n1-v1: bs-val (BNat 0) v1 by blast
  from n1-v1 have v1: v1 = ENat 0 by blast
  from eval-if0(6) have FV e3 ⊆ set (map fst ϱ) by auto
  from this eval-if0 obtain v3' where e3-v3: psubst ϱ' e3 ⟶* v3' and
    v3-v3: bs-val v3 v3' by blast

  from e1-v1 have 1: EIf (psubst ϱ' e1) (psubst ϱ' e2) (psubst ϱ' e3)
    ⟶* EIf v1 (psubst ϱ' e2) (psubst ϱ' e3) by (rule if-red-cong1)
  from v1 have 3: EIf v1 (psubst ϱ' e2) (psubst ϱ' e3) ⟶ (psubst ϱ' e3) by auto
  from 1 3 have 5: psubst ϱ' (EIf e1 e2 e3) ⟶* psubst ϱ' e3 apply simp
    apply (rule multi-step-trans) apply assumption apply blast done
  from 5 e3-v3 have 6: psubst ϱ' (EIf e1 e2 e3) ⟶* v3'
    apply (rule multi-step-trans) done
  from 6 v3-v3 show ?case by blast
next
  case (eval-if1 ϱ e1 n e2 v2 e3 ϱ')
  from eval-if1 have FV e1 ⊆ set (map fst ϱ) by auto
  from this eval-if1 obtain v1 where e1-v1: psubst ϱ' e1 ⟶* v1 and
    n1-v1: bs-val (BNat n) v1 and nz: n ≠ 0 apply auto apply blast done
  from n1-v1 have v1: v1 = ENat n by blast
  from eval-if1 have FV e2 ⊆ set (map fst ϱ) by auto
  from this eval-if1 obtain v2' where e2-v2: psubst ϱ' e2 ⟶* v2' and
    v2-v2: bs-val v2 v2' by blast
  from e1-v1 have 1: EIf (psubst ϱ' e1) (psubst ϱ' e2) (psubst ϱ' e3)
    ⟶* EIf v1 (psubst ϱ' e2) (psubst ϱ' e3) by (rule if-red-cong1)
  from v1 nz have 3: EIf v1 (psubst ϱ' e2) (psubst ϱ' e3) ⟶ (psubst ϱ' e2) by auto
  from 1 3 have 5: psubst ϱ' (EIf e1 e2 e3) ⟶* psubst ϱ' e2 apply simp
    apply (rule multi-step-trans) apply assumption apply blast done
  from 5 e2-v2 have 6: psubst ϱ' (EIf e1 e2 e3) ⟶* v2'
    by (rule multi-step-trans)
  from 6 v2-v2 show ?case by blast
qed

lemma psubst-id: FV e ∩ sdom ϱ = {} ⟹ psubst ϱ e = e
proof (induction e arbitrary: ϱ)
  case (EVar x)
  then show ?case by (cases lookup ϱ x) (auto simp: sdom-def)
```

```
next
  case (ENat x ϱ)
  from ENat have sdom ((x,EVar x)#ϱ) = sdom ϱ − {x} by simp
  with ENat show ?case by auto
next
  case (ELam x e)
  from ELam have FV e ∩ sdom ((x,EVar x)#ϱ) = {} by auto
  with ELam show ?case by auto
qed fastforce+


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 −
  have 1: bs-env [] [] by blast
  from fv-e have 2: FV e ⊆ set (map fst []) by simp
  from e-v 1 2 big-small-step obtain v′ where 3: psubst [] e ⟶∗ v′ and 4: is-val v′ and
    5: bs-val v v′ by blast
  have psubst [] e = e using psubst-id sdom-def apply auto done
  from this 3 4 5 show ?thesis apply (rule-tac x=v′ in exI) apply simp
    apply (case-tac v)
     apply simp apply clarify apply simp
       apply (rename-tac n) apply (rule-tac x=ONat n in exI) apply force
     apply (rule-tac x=OFun in exI) apply force done
qed
```

3.2 Big-step semantics is deterministic

```
theorem big-step-fun:
  assumes ev: ϱ ⊢ e ⇓ v and evp: ϱ ⊢ e ⇓ v′ shows v = v′
  using ev evp
proof (induction arbitrary: v′)
  case (eval-app ϱ e1 x e ϱ′ e2 arg v)
  from eval-app(7) obtain x′ e′ ϱ′′ arg′ where e1-cl: ϱ ⊢ e1 ⇓ BClos x′ e′ ϱ′′ and
    e2-argp: ϱ ⊢ e2 ⇓ arg′ and e-vp: (x′, arg′) # ϱ′′ ⊢ e′ ⇓ v′ by blast
  from eval-app(4) e1-cl have 1: BClos x e ϱ′ = BClos x′ e′ ϱ′′ by simp
  from eval-app(5) e2-argp have 2: arg = arg′ by simp
  from eval-app(6) e-vp 1 2 show ?case by simp
next
  case (eval-if0 ϱ e1 e3 v3 e2)
  from eval-if0(5)
  show ?case
  proof (rule eval-if-inv)
    assume ϱ ⊢ e3 ⇓ v′ with eval-if0(4) show ?thesis by simp
  next
    fix n assume ϱ ⊢ e1 ⇓ BNat n and nz: n > 0
    with eval-if0(3) have False by auto thus ?thesis ..
  qed
next
  case (eval-if1 ϱ e1 n e2 v2 e3)
  then show ?case by blast
qed fastforce+

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* ⟹ (*VFun t1*) ⊑ (*VFun t2*)

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

definition *env-le* :: *env* ⇒ *env* ⇒ *bool* (**infix** ⊑ *52*) **where**
ϱ ⊑ ϱ′ ≡ ∀ *x v*. *lookup* ϱ *x* = *Some v* ⟶ (∃ *v*′. *lookup* ϱ′ *x* = *Some v*′ ∧ *v* ⊑ *v*′)

definition *env-eq* :: *env* ⇒ *env* ⇒ *bool* (**infix** ≈ *50*) **where**
ϱ ≈ ϱ′ ≡ (∀ *x*. *lookup* ϱ *x* = *lookup* ϱ′ *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*
unfolding *comp-fun-commute-def* **by** *auto*

lemma *vprod-size-inj*: *inj-on vprod-size* (*fset A*)
unfolding *inj-on-def* **by** *auto*

lemma *fsize-def2*: *fsize t* = *1* + *ffold* (*vadd* ∘ *vprod-size*) *0 t*
using *vprod-size-inj*[*of t*] *ffold-fimage*[*of vprod-size t vadd 0*] **by** *simp*

lemma *fsize-finsert-in*[*simp*]:
assumes *v12-t*: (*v1*,*v2*) |∈| *t* **shows** *fsize* (*finsert* (*v1*,*v2*) *t*) = *fsize t*
proof −
from *v12-t* **have** *finsert* (*v1*,*v2*) *t* = *t* **by** *auto*
from *this* **show** *?thesis* **by** *simp*
qed

lemma *fsize-finsert-notin*[*simp*]:
assumes *v12-t*: (*v1*,*v2*) |∉| *t*
shows *fsize* (*finsert* (*v1*,*v2*) *t*) = *vsize v1* + *vsize v2* + *fsize t*
proof −
let *?f* = *vadd* ∘ *vprod-size*
have *fsize* (*finsert* (*v1*,*v2*) *t*) = *1* + *ffold ?f 0* (*finsert* (*v1*,*v2*) *t*)
using *fsize-def2*[*of finsert* (*v1*,*v2*) *t*] **by** *simp*
also from *v12-t* **have** ... = *1* + *?f* (*v1*,*v2*) (*ffold ?f 0 t*) **by** *simp*
finally have *fsize* (*finsert* (*v1*,*v2*) *t*) = *1* + *?f* (*v1*,*v2*) (*ffold ?f 0 t*) .
from *this* **show** *?thesis* **using** *fsize-def2*[*of t*] **by** *simp*
qed

end
theory *ValuesFSetProps*
imports *ValuesFSet*
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*

proposition *val-le-refl*[*simp*]: **fixes** *v*::*val* **shows** *v* $\sqsubseteq$ *v* **by** (*induction v*) *auto*

proposition *val-le-trans*[*trans*]: **fixes** *v2*::*val* **shows** ⟦ *v1* $\sqsubseteq$ *v2*; *v2* $\sqsubseteq$ *v3* ⟧ $\Longrightarrow$ *v1* $\sqsubseteq$ *v3*
by (*induction v2 arbitrary*: *v1 v3*) *blast*+

lemma *fsubset*[*intro*!]: *fset A* $\subseteq$ *fset B* $\Longrightarrow$ *A* |$\subseteq$| *B*
proof (*rule fsubsetI*)
fix *x* **assume** *ab*: *fset A* $\subseteq$ *fset B* **and** *xa*: *x* |$\in$| *A*
from *xa* **have** *x* $\in$ *fset A* **by** *simp*
from *this ab* **have** *x* $\in$ *fset B* **by** *blast*
from *this* **show** *x* |$\in$| *B* **by** *simp*
qed

proposition *val-le-antisymm*: **fixes** *v1*::*val* **shows** ⟦ *v1* $\sqsubseteq$ *v2*; *v2* $\sqsubseteq$ *v1* ⟧ $\Longrightarrow$ *v1* = *v2*
by (*induction v1 arbitrary*: *v2*) *auto*

lemma *le-nat-any*[*simp*]: *VNat n* $\sqsubseteq$ *v* $\Longrightarrow$ *v* = *VNat n*
by (*cases v*) *auto*

lemma *le-any-nat*[*simp*]: *v* $\sqsubseteq$ *VNat n* $\Longrightarrow$ *v* = *VNat n*
by (*cases v*) *auto*

lemma *le-nat-nat*[*simp*]: *VNat n* $\sqsubseteq$ *VNat n'* $\Longrightarrow$ *n* = *n'*
by *auto*

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*!]: *ϱ* ⊢ *ENat n* ⇒ *VNat n* |
rprim[*intro*!]: ⟦ *ϱ* ⊢ *e1* ⇒ *VNat n1*; *ϱ* ⊢ *e2* ⇒ *VNat n2* ⟧ $\Longrightarrow$ *ϱ* ⊢ *EPrim f e1 e2* ⇒ *VNat* (*f n1 n2*) |
rvar[*intro*!]: ⟦ *lookup ϱ x* = *Some v'*; *v* $\sqsubseteq$ *v'* ⟧ $\Longrightarrow$ *ϱ* ⊢ *EVar x* ⇒ *v* |
rlam[*intro*!]: ⟦ ∀ *v v'*. (*v*,*v'*) ∈ *fset t* ⟶ (*x*,*v*)#*ϱ* ⊢ *e* ⇒ *v'* ⟧
$\Longrightarrow$ *ϱ* ⊢ *ELam x e* ⇒ *VFun t* |
rapp[*intro*!]: ⟦ *ϱ* ⊢ *e1* ⇒ *VFun t*; *ϱ* ⊢ *e2* ⇒ *v2*; (*v3*,*v3'*) ∈ *fset t*; *v3* $\sqsubseteq$ *v2*; *v* $\sqsubseteq$ *v3'* ⟧
$\Longrightarrow$ *ϱ* ⊢ *EApp e1 e2* ⇒ *v* |
rifnz[*intro*!]: ⟦ *ϱ* ⊢ *e1* ⇒ *VNat n*; *n* ≠ *0*; *ϱ* ⊢ *e2* ⇒ *v* ⟧ $\Longrightarrow$ *ϱ* ⊢ *EIf e1 e2 e3* ⇒ *v* |
rifz[*intro*!]: ⟦ *ϱ* ⊢ *e1* ⇒ *VNat n*; *n* = *0*; *ϱ* ⊢ *e3* ⇒ *v* ⟧ $\Longrightarrow$ *ϱ* ⊢ *EIf e1 e2 e3* ⇒ *v*

end
theory *DeclSemAsDenotFSet*

imports *Lambda ValuesFSet*
begin

5 Declarative semantics as a denotational semantics

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

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 (*induction e arbitrary*: $v\ \varrho$)
case (*EIf e1 e2 e3*)
then show *?case*
apply *simp* **apply** *clarify* **apply** (*rename-tac n*) **apply** (*case-tac n*) **apply** *force* **apply** *simp*
apply (*rule rifnz*) **apply** *force*+ **done**
qed *auto*

lemma *rel-implies-denot*: $\varrho \vdash e \Rightarrow v \Longrightarrow v \in E\ e\ \varrho$
by (*induction* ϱ *e v rule*: *rel-sem.induct*) *auto*

theorem *equivalence-relational-denotational*: $(v \in E\ e\ \varrho) = (\varrho \vdash e \Rightarrow v)$
using *denot-implies-rel rel-implies-denot* **by** *blast*

end

7 Subsumption and change of environment

theory *ChangeEnv*
imports *Main Lambda DeclSemAsDenotFSet ValuesFSetProps*
begin

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

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

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

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* **by** *auto*

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*
by *auto*

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*
by *auto*

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

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*
by *auto*

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

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

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*
by *auto*

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

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*
by *auto*

definition *xenv-le* :: *name set* ⇒ *env* ⇒ *env* ⇒ *bool* (- ⊢ - ⊑ - [*51*,*51*,*51*] *52*) **where**
X ⊢ *ϱ* ⊑ *ϱ′* ≡ ∀ *x v*. *x* ∈ *X* ∧ *lookup ϱ x* = *Some v* ⟶ (∃ *v′*. *lookup ϱ′ x* = *Some v′* ∧ *v* ⊑ *v′*)
declare *xenv-le-def*[*simp*]

proposition *change-env-le*: **fixes** *v*::*val* **and** *ϱ*::*env*
assumes *de*: *v* ∈ *E e ϱ* **and** *vp-v*: *v′* ⊑ *v* **and** *rr*: *FV e* ⊢ *ϱ* ⊑ *ϱ′*
shows *v′* ∈ *E e ϱ′*
using *de rr vp-v*
proof (*induction e arbitrary*: *v v′ ϱ ϱ′ rule*: *exp.induct*)
case (*EVar x v v′ ϱ ϱ′*)
from *EVar* **obtain** *v2* **where** *lx*: *lookup ϱ x* = *Some v2* **and** *v-v2*: *v* ⊑ *v2* **by** *auto*
from *lx EVar* **obtain** *v3* **where**
lx2: *lookup ϱ′ x* = *Some v3* **and** *v2-v3*: *v2* ⊑ *v3* **by** *force*
from *v-v2 v2-v3* **have** *v-v3*: *v* ⊑ *v3* **by** (*rule val-le-trans*)
from *EVar v-v3* **have** *vp-v3*: *v′* ⊑ *v3* **using** *val-le-trans* **by** *blast*
from *lx2 vp-v3* **show** *?case* **by** (*rule e-var-intro*)
next
case (*ENat n*) **then show** *?case* **by** *simp*
next
case (*ELam x e*)
from *ELam*(*2*) **obtain** *f* **where** *v*: *v* = *VFun f* **and**

```
    body: ∀ v1 v2. (v1,v2) ∈ fset f ⟶ v2 ∈ E e ((x,v1)#ϱ) by auto
  from v ELam(4) obtain f' where vp: v' = VFun f' and fp-f: fset f' ⊆ fset f
    by (case-tac v') auto
  from vp show ?case
  proof (simp,clarify)
    fix v1 v2 assume v12: (v1,v2)∈ fset f'
    from v12 fp-f have v34: (v1,v2) ∈ fset f by blast
    from v34 body have v4-E: v2 ∈ E e ((x,v1)#ϱ) by blast
    from ELam(3) have rr2: FV e ⊢ ((x,v1)#ϱ) ⊑ ((x,v1)#ϱ') by auto
    from ELam(1) v4-E rr2 show v2 ∈ E e ((x,v1)#ϱ') by auto
  qed
next
  case (EApp e1 e2)
  from EApp(3) obtain f and v2::val and v2' v3' where
    f-e1: VFun f ∈ E e1 ϱ and v2-e2: v2 ∈ E e2 ϱ and
    v23p-f: (v2',v3') ∈ fset f and v2p-v2: v2' ⊑ v2 and v-v3: v ⊑ v3' by blast
  from EApp(4) have 1: FV e1 ⊢ ϱ ⊑ ϱ' by auto
  have f-f: VFun f ⊑ VFun f by auto
  from EApp(1) f-e1 1 f-f have f-e1b: VFun f ∈ E e1 ϱ' by blast
  from EApp(4) have 2: FV e2 ⊢ ϱ ⊑ ϱ' by auto
  from EApp(2) v2-e2 2 have v2-e2b: v2 ∈ E e2 ϱ' by auto
  from EApp(5) v-v3 have vp-v3p: v' ⊑ v3' by (rule val-le-trans)
  from f-e1b v2-e2b v23p-f v2p-v2 vp-v3p
  show ?case by auto
next
  case (EPrim f e1 e2)
  from EPrim(3) obtain n1 n2 where n1-e1: VNat n1 ∈ E e1 ϱ and
    n2-e2: VNat n2 ∈ E e2 ϱ and v: v = VNat (f n1 n2) by blast
  from EPrim(4) have 1: FV e1 ⊢ ϱ ⊑ ϱ' by auto
  from EPrim(1) n1-e1 1 have n1-e1b: VNat n1 ∈ E e1 ϱ' by blast
  from EPrim(4) have 2: FV e2 ⊢ ϱ ⊑ ϱ' by auto
  from EPrim(2) n2-e2 2 have n2-e2b: VNat n2 ∈ E e2 ϱ' by blast
  from v EPrim(5) have vp: v' = VNat (f n1 n2) by auto
  from n1-e1b n2-e2b vp show ?case by auto
next
  case (EIf e1 e2 e3)
  then show ?case apply simp apply clarify apply (rule-tac x=n in exI) apply (rule conjI)
     apply force apply force done
qed

— Subsumption is admissible
proposition e-sub: ⟦ v ∈ E e ϱ; v' ⊑ v ⟧ ⟹ v' ∈ E e ϱ
  apply (subgoal-tac FV e ⊢ ϱ ⊑ ϱ) using change-env-le apply blast apply auto done

lemma env-le-ext: fixes ϱ::env assumes rr: ϱ ⊑ ϱ' shows ((x,v)#ϱ) ⊑ ((x,v)#ϱ')
  using rr apply (simp add: env-le-def) done

lemma change-env: fixes ϱ::env assumes de: v ∈ E e ϱ and rr: FV e ⊢ ϱ ⊑ ϱ' shows v ∈ E e ϱ'
proof −
  have vv: v ⊑ v by auto
  from de rr vv show ?thesis using change-env-le by blast
qed

lemma raise-env: fixes ϱ::env assumes de: v ∈ E e ϱ and rr: ϱ ⊑ ϱ' shows v ∈ E e ϱ'
  using de rr change-env env-le-def by auto

lemma env-eq-refl[simp]: fixes ϱ::env shows ϱ ≈ ϱ by (simp add: env-eq-def)

lemma env-eq-ext: fixes ϱ::env assumes rr: ϱ ≈ ϱ' shows ((x,v)#ϱ) ≈ ((x,v)#ϱ')
  using rr by (simp add: env-eq-def)
```

lemma *eq-implies-le*: **fixes** ϱ::*env* **shows** $\varrho \approx \varrho' \Longrightarrow \varrho \sqsubseteq \varrho'$
by (*simp add*: *env-le-def env-eq-def*)

lemma *env-swap*: **fixes** ϱ::*env* **assumes** *rr*: $\varrho \approx \varrho'$ **and** *ve*: $v \in E\ e\ \varrho$ **shows** $v \in E\ e\ \varrho'$
using *rr ve* **apply** (*subgoal-tac* $\varrho \sqsubseteq \varrho'$) **prefer** *2* **apply** (*rule eq-implies-le*) **apply** *blast*
apply (*rule raise-env*) **apply** *auto* **done**

lemma *env-strengthen*: $[\![\ v \in E\ e\ \varrho; \forall\ x.\ x \in FV\ e \longrightarrow lookup\ \varrho'\ x = lookup\ \varrho\ x\ ]\!] \Longrightarrow v \in E\ e\ \varrho'$
using *change-env* **by** *auto*

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* (**infixl** *choose 65*)

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: $E\ (ENat\ n)\ \varrho = return\ (VNat\ n)$ |
Evar2: $E\ (EVar\ x)\ \varrho = (case\ lookup\ \varrho\ x\ of\ None \Rightarrow zero \mid Some\ v \Rightarrow down\ v)$ |
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: $E\ (EApp\ e1\ e2)\ \varrho = apply\text{-}fun\ (E\ e1\ \varrho)\ (E\ e2\ \varrho)$ |
Eprim2: $E\ (EPrim\ f\ e1\ e2)\ \varrho = (v_1 \leftarrow E\ e1\ \varrho;\ v_2 \leftarrow E\ e2\ \varrho;$
$case\ (v_1,v_2)\ of$
$(VNat\ n_1,\ VNat\ n_2) \Rightarrow return\ (VNat\ (f\ n_1\ n_2))$
$\mid (VNat\ n_1,\ VFun\ t_2) \Rightarrow zero$
$\mid (VFun\ t_1,\ v_2) \Rightarrow zero)$ |
Eif2[*eta-contract* = *false*]: $E\ (EIf\ e1\ e2\ e3)\ \varrho = (v_1 \leftarrow E\ e1\ \varrho;$
$case\ v_1\ of$
$(VNat\ n) \Rightarrow if\ n \neq 0\ then\ E\ e2\ \varrho\ else\ E\ e3\ \varrho$
$\mid (VFun\ t) \Rightarrow zero)$

end

9 Declarative semantics as a type system

theory *InterTypeSystem*
imports *Lambda*
begin

datatype $ty = TNat\ nat \mid TFun\ funty$
and $funty = TArrow\ ty\ ty$ (**infix** $\rightarrow$ *55*) $\mid TInt\ funty\ funty$ (**infix** $\sqcap$ *56*) $\mid TTop$ ($\top$)

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-inter-r[*intro!*]: $[\![\ T3 <:: T1;\ T3 <:: T2\ ]\!] \Longrightarrow T3 <:: T1 \sqcap T2$ |
sub-fun-top[*intro!*]: $T1 \rightarrow T2 <:: \top$ |
sub-top-top[*intro!*]: $\top <:: \top$ |
fsub-refl[*intro!*]: $T <:: T$ |
sub-trans[*trans*]: $[\![\ T1 <:: T2;\ T2 <:: T3\ ]\!] \Longrightarrow T1 <:: T3$

definition *ty-eq* :: $ty \Rightarrow ty \Rightarrow bool$ (**infix** $\approx$ *50*) **where**
$A \approx B \equiv A <: B \wedge B <: A$
definition *fty-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-var[*intro!*]: $lookup\ \Gamma\ x = Some\ T \Longrightarrow \Gamma \vdash EVar\ x : T$ |
wt-nat[*intro!*]: $\Gamma \vdash ENat\ n : TNat\ n$ |
wt-lam[*intro!*]: $[\![\ (x,A)\#\Gamma \vdash e : B\ ]\!] \Longrightarrow \Gamma \vdash ELam\ x\ e : TFun\ (A \rightarrow B)$ |
wt-app[*intro!*]: $[\![\ \Gamma \vdash e1 : TFun\ (A \rightarrow B);\ \Gamma \vdash e2 : A\ ]\!] \Longrightarrow \Gamma \vdash EApp\ e1\ e2 : B$ |
wt-top[*intro!*]: $\Gamma \vdash ELam\ x\ e : TFun\ \top$ |
wt-inter[*intro!*]: $[\![\ \Gamma \vdash ELam\ x\ e : TFun\ A;\ \Gamma \vdash ELam\ x\ e : TFun\ B\ ]\!]$
$\Longrightarrow \Gamma \vdash ELam\ x\ e : TFun\ (A \sqcap B)$ |
wt-sub[*intro!*]: $[\![\ \Gamma \vdash e : A;\ A <: B\ ]\!] \Longrightarrow \Gamma \vdash e : B$ |
wt-prim[*intro!*]: $[\![\ \Gamma \vdash e1 : TNat\ n1;\ \Gamma \vdash e2 : TNat\ n2\ ]\!]$
$\Longrightarrow \Gamma \vdash EPrim\ f\ e1\ e2 : TNat\ (f\ n1\ n2)$ |
wt-ifz[*intro!*]: $[\![\ \Gamma \vdash e1 : TNat\ 0;\ \Gamma \vdash e3 : B\ ]\!]$
$\Longrightarrow \Gamma \vdash EIf\ e1\ e2\ e3 : B$ |

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**
ϱ ⊑ ϱ′ ≡ ∀ *x v*. *lookup* ϱ *x* = *Some v* ⟶ (∃ *v*′. *lookup* ϱ′ *x* = *Some v*′ ∧ *v* ⊑ *v*′)

definition *env-eq* :: *env* ⇒ *env* ⇒ *bool* (**infix** ≈ *50*) **where**
ϱ ≈ ϱ′ ≡ (∀ *x*. *lookup* ϱ *x* = *lookup* ϱ′ *x*)

end

10.2 Properties about values

theory *ValueProps*
imports *Values*
begin

inductive-cases *fun-le-inv*[*elim*]: *t1* ≲ *t2* **and**
vfun-le-inv[*elim*!]: *VFun t1* ⊑ *VFun t2* **and**
le-fun-nat-inv[*elim*!]: *VFun t2* ⊑ *VNat x1* **and**
le-fun-cons-inv[*elim*!]: (*v1*, *v2*) # *t1* ≲ *t2* **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*

lemma *fun-le-cons*: (*a* # *t1*) ≲ *t2* ⟹ *t1* ≲ *t2*
by (*case-tac a*) *auto*

function *val-size* :: *val* ⇒ *nat* **and** *fun-size* :: *func* ⇒ *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
  by pat-completeness auto
termination val-size by size-change

lemma val-size-mem: (a, b) ∈ set t ⟹ val-size a + val-size b < fun-size t
  by (induction t) auto
lemma val-size-mem-l: (a, b) ∈ set t ⟹ val-size a < fun-size t
  by (induction t) auto
lemma val-size-mem-r: (a, b) ∈ set t ⟹ val-size b < fun-size t
  by (induction t) auto

lemma val-fun-le-refl: ∀ v t. n = val-size v + fun-size t ⟶ v ⊑ v ∧ t ≲ t
proof (induction n rule: nat-less-induct)
  case (1 n)
  show ?case apply clarify apply (rule conjI)
  proof −
    fix v::val and t::func assume n: n = val-size v + fun-size t
    show v ⊑ v
    proof (cases v)
      case (VNat x1)
      then show ?thesis by auto
    next
      case (VFun t')
      let ?m = val-size (VNat 0) + fun-size t'
      from 1 n VFun have t' ≲ t'
        apply (erule-tac x=?m in allE) apply (erule impE)
         apply force apply (erule-tac x=VNat 0 in allE) apply (erule-tac x=t' in allE)
        apply simp done
      from this VFun show ?thesis by force
    qed
  next
    fix v::val and t::func assume n: n = val-size v + fun-size t
    show t ≲ t
      apply (rule fun-le) apply clarify
    proof −
      fix v1 v2 assume v12: (v1,v2) ∈ set t
      from 1 v12 have v11: v1 ⊑ v1
        apply (erule-tac x=val-size v1 + fun-size [] in allE)
        apply (erule impE) using n apply simp apply (frule val-size-mem) apply force
        apply (erule-tac x=v1 in allE) apply (erule-tac x=[] in allE) apply force done
      from 1 v12 have v22: v2 ⊑ v2
        apply (erule-tac x=val-size v2 + fun-size [] in allE)
        apply (erule impE) using n apply simp apply (frule val-size-mem) apply force
        apply (erule-tac x=v2 in allE) apply (erule-tac x=[] in allE) apply force done
      from v12 v11 v22
      show ∃ v3 v4. (v3,v4) ∈ set t ∧ v1 ⊑ v3 ∧ v3 ⊑ v1 ∧ v2 ⊑ v4 ∧ v4 ⊑ v2 by blast
      qed
  qed
qed

proposition val-le-refl[simp]: fixes v::val shows v ⊑ v using val-fun-le-refl by auto

lemma fun-le-refl[simp]: fixes t::func shows t ≲ t using val-fun-le-refl by auto

definition val-eq :: val ⇒ val ⇒ bool (infix ∼ 52) where
  val-eq v1 v2 ≡ (v1 ⊑ v2 ∧ v2 ⊑ v1)

definition fun-eq :: func ⇒ func ⇒ bool (infix ∼ 52) where
```

```
  fun-eq t1 t2 ≡ (t1 ≲ t2 ∧ t2 ≲ t1)

lemma vfun-eq[intro!]: t ∼ t' ⟹ VFun t ∼ VFun t'
  apply (simp add: val-eq-def fun-eq-def)
  apply (rule conjI) apply (erule conjE) apply (rule vfun-le) apply assumption
  apply (erule conjE) apply (rule vfun-le) apply assumption
  done

lemma val-eq-refl[simp]: fixes v::val shows v ∼ v
  by (simp add: val-eq-def)

lemma val-eq-symm: fixes v1::val and v2::val shows v1 ∼ v2 ⟹ v2 ∼ v1
  unfolding val-eq-def by blast

lemma val-le-fun-le-trans:
   ∀ v2 t2. n = val-size v2 + fun-size t2 ⟶
    (∀ v1 v3. v1 ⊑ v2 ⟶ v2 ⊑ v3 ⟶ v1 ⊑ v3)
    ∧ (∀ t1 t3. t1 ≲ t2 ⟶ t2 ≲ t3 ⟶ t1 ≲ t3)
proof (induction n rule: nat-less-induct)
  case (1 n)
  show ?case apply clarify
  proof
    fix v2 t2 assume n: n = val-size v2 + fun-size t2
    show ∀ v1 v3. v1 ⊑ v2 ⟶ v2 ⊑ v3 ⟶ v1 ⊑ v3 apply clarify
    proof −
      fix v1 v3 assume v12: v1 ⊑ v2 and v23: v2 ⊑ v3
      show v1 ⊑ v3
      proof (cases v2)
        case (VNat n)
        from VNat v12 have v1: v1 = VNat n by auto
        from VNat v23 have v3: v3 = VNat n by auto
        from v1 v3 show ?thesis by auto
      next
        case (VFun t2')
        from v12 VFun obtain t1 where t12: t1 ≲ t2' and v1: v1 = VFun t1 by auto
        from v23 VFun obtain t3 where t23: t2' ≲ t3 and v3: v3 = VFun t3 by auto
        let ?m = val-size (VNat 0) + fun-size t2'
        from 1 n VFun have IH: ∀ t1 t3. t1 ≲ t2' ⟶ t2' ≲ t3 ⟶ t1 ≲ t3
          apply simp apply (erule-tac x=?m in allE) apply (erule impE) apply force
          apply (erule-tac x=VNat 0 in allE)apply (erule-tac x=t2' in allE)
          apply auto done
        from t12 t23 IH have t1 ≲ t3 by auto
        from this v1 v3 show ?thesis apply auto done
      qed
    qed
  next
    fix v5 t2 assume n: n = val-size v5 + fun-size t2
    show ∀ t1 t3. t1 ≲ t2 ⟶ t2 ≲ t3 ⟶ t1 ≲ t3 apply clarify
    proof −
      fix t1 t3 v1 v2 assume t12: t1 ≲ t2 and t23: t2 ≲ t3 and v12: (v1,v2) ∈ set t1
      from v12 t12 obtain v1' v2' where v12p: (v1',v2') ∈ set t2 and
          v1-v1p: v1 ⊑ v1' and v11p: v1' ⊑ v1 and v22p: v2 ⊑ v2' and v2p-v2: v2' ⊑ v2 by blast
      from v12p t23 obtain v1'' v2'' where v12pp: (v1'',v2'') ∈ set t3 and
         v1p-v1pp: v1' ⊑ v1'' and v11pp: v1'' ⊑ v1' and
         v22pp: v2' ⊑ v2'' and v2pp-v2p: v2'' ⊑ v2' by blast

      from v12p have sv1p: val-size v1' < fun-size t2 using val-size-mem-l by blast
      from v12 1 v11p v11pp n sv1p have v1pp-v1: v1'' ⊑ v1
        apply (erule-tac x=val-size v1' + fun-size [] in allE)
        apply (erule impE) apply force apply (erule-tac x=v1' in allE)
        apply (erule-tac x=[] in allE) apply (erule impE) apply force
```

```
    apply (erule conjE) apply blast done

  from v12p have sv2p: val-size v2' < fun-size t2 using val-size-mem-r by blast
  from v12 1 v22p v22pp n sv2p have v2-v2pp: v2 ⊑ v2''
    apply (erule-tac x=val-size v2' + fun-size [] in allE)
    apply (erule impE) apply force apply (erule-tac x=v2' in allE)
    apply (erule-tac x=[] in allE) apply (erule impE) apply force
    apply (erule conjE) apply blast done

  from v12 1 v1-v1p v1p-v1pp n sv1p have v1-v1pp: v1 ⊑ v1''
    apply (erule-tac x=val-size v1' + fun-size [] in allE)
    apply (erule impE) apply force apply (erule-tac x=v1' in allE)
    apply (erule-tac x=[] in allE) apply (erule impE) apply force
    apply (erule conjE) apply blast done

  from v12 1 v2pp-v2p v2p-v2 n sv2p have v2pp-v2: v2'' ⊑ v2
    apply (erule-tac x=val-size v2' + fun-size [] in allE)
    apply (erule impE) apply force apply (erule-tac x=v2' in allE)
    apply (erule-tac x=[] in allE) apply (erule impE) apply force
    apply (erule conjE) apply blast done

  from v12pp v1pp-v1 v2-v2pp v1-v1pp v2pp-v2
  show ∃ v3 v4. (v3, v4) ∈ set t3 ∧ v1 ⊑ v3 ∧ v3 ⊑ v1 ∧ v2 ⊑ v4 ∧ v4 ⊑ v2 by blast
 qed
qed
qed

proposition val-le-trans: fixes v2::val shows ⟦ v1 ⊑ v2; v2 ⊑ v3 ⟧ ⟹ v1 ⊑ v3
 using val-le-fun-le-trans by blast

lemma fun-le-trans: ⟦ t1 ≲ t2; t2 ≲ t3 ⟧ ⟹ t1 ≲ t3
 using val-le-fun-le-trans by blast

lemma val-eq-trans: fixes v1::val and v2::val and v3::val
 assumes v12: v1 ∼ v2 and v23: v2 ∼ v3 shows v1 ∼ v3
 using v12 v23 apply (simp only: val-eq-def) using val-le-trans apply blast done

lemma fun-eq-refl[simp]: fixes t::func shows t ∼ t
 by (simp add: fun-eq-def)

lemma fun-eq-trans: fixes t1::func and t2::func and t3::func
 assumes t12: t1 ∼ t2 and t23: t2 ∼ t3 shows t1 ∼ t3
 using t12 t23 unfolding fun-eq-def apply clarify apply (rule conjI)
  apply (rule fun-le-trans) apply assumption apply assumption
 apply (rule fun-le-trans) apply assumption apply assumption
 done

lemma append-fun-le:
  ⟦ t1' ≲ t1; t2' ≲ t2 ⟧ ⟹ t1' @ t2' ≲ t1 @ t2
 apply (rule fun-le) apply clarify apply simp apply (erule fun-le-inv)+ apply blast done

lemma append-fun-equiv:
  ⟦ t1' ∼ t1; t2' ∼ t2 ⟧ ⟹ t1' @ t2' ∼ t1 @ t2
 apply (simp add: val-eq-def fun-eq-def) using append-fun-le apply blast done

lemma append-leq-symm: t2 @ t1 ≲ t1 @ t2
 apply (rule fun-le) apply force done

lemma append-eq-symm: t2 @ t1 ∼ t1 @ t2
 unfolding fun-eq-def val-eq-def apply (rule conjI)
 apply (rule append-leq-symm) apply (rule append-leq-symm) done
```

lemma *le-nat-any*[*simp*]: $VNat\ n \sqsubseteq v \Longrightarrow v = VNat\ n$
by (*cases v*) *auto*

lemma *le-any-nat*[*simp*]: $v \sqsubseteq VNat\ n \Longrightarrow v = VNat\ n$
by (*cases v*) *auto*

lemma *le-nat-nat*[*simp*]: $VNat\ n \sqsubseteq VNat\ n' \Longrightarrow n = n'$
by *auto*

end

10.3 Declarative semantics as a denotational semantics

theory *DeclSemAsDenot*
imports *Lambda Values*
begin

fun $E :: exp \Rightarrow env \Rightarrow val\ set$ **where**
Enat: $E\ (ENat\ n)\ \varrho = \{\ v.\ v = VNat\ n\ \} \mid$
Evar: $E\ (EVar\ x)\ \varrho = \{\ v.\ \exists\ v'.\ lookup\ \varrho\ x = Some\ v' \wedge v \sqsubseteq v'\ \} \mid$
Elam: $E\ (ELam\ x\ e)\ \varrho = \{\ v.\ \exists\ f.\ v = VFun\ f \wedge (\forall\ v1\ v2.\ (v1,\ v2) \in set\ f \longrightarrow v2 \in E\ e\ ((x,v1)\#\varrho))\ \} \mid$
Eapp: $E\ (EApp\ e1\ e2)\ \varrho = \{\ v3.\ \exists\ f\ v2\ v2'\ v3'.\ VFun\ f \in E\ e1\ \varrho \wedge v2 \in E\ e2\ \varrho \wedge (v2',\ v3') \in set\ f \wedge v2' \sqsubseteq v2 \wedge v3 \sqsubseteq v3'\ \} \mid$
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*]: $[\![\ VNat\ n1 \in E\ e1\ \varrho;\ VNat\ n2 \in E\ e2\ \varrho;\ v = VNat\ (f\ n1\ n2)\ ]\!] \Longrightarrow v \in E\ (EPrim\ f\ e1\ e2)\ \varrho$ **by** *auto*

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

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

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

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

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

by *auto*

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

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

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

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

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

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

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

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 $v' \in E\ e\ \varrho'$
using *de rr vp-v*
proof (*induction e arbitrary*: *v v' ϱ ϱ' rule*: *exp.induct*)
case (*EVar x v v' ϱ ϱ'*)
from *EVar* **obtain** *v2* **where** *lx*: $lookup\ \varrho\ x = Some\ v2$ **and** *v-v2*: $v \sqsubseteq v2$ **by** *auto*
from *lx EVar* **obtain** *v3* **where**
lx2: $lookup\ \varrho'\ x = Some\ v3$ **and** *v2-v3*: $v2 \sqsubseteq v3$ **by** *force*
from *v-v2 v2-v3* **have** *v-v3*: $v \sqsubseteq v3$ **by** (*rule val-le-trans*)
from *EVar v-v3* **have** *vp-v3*: $v' \sqsubseteq v3$ **using** *val-le-trans* **by** *blast*
from *lx2 vp-v3* **show** *?case* **by** (*rule e-var-intro*)
next
case (*ENat n*) **then show** *?case* **by** *simp*
next
case (*ELam x e*)
from *ELam*(*2*) **obtain** *f* **where** *v*: $v = VFun\ f$ **and**
body: $\forall\ v1\ v2.\ (v1,v2) \in set\ f \longrightarrow v2 \in E\ e\ ((x,v1)\#\varrho)$ **by** *auto*
from *v ELam*(*4*) **obtain** f' **where** *vp*: $v' = VFun\ f'$ **and** *fp-f*: $f' \lesssim f$ **by** (*case-tac v'*) *auto*
from *vp* **show** *?case*
proof (*simp*, *clarify*)
fix *v1 v2* **assume** *v12*: $(v1,v2) \in set\ f'$
from *v12 fp-f* **obtain** *v3 v4* **where** *v34*: $(v3,v4) \in set\ f$ **and**
v31: $v3 \sqsubseteq v1$ **and** *v24*: $v2 \sqsubseteq v4$ **by** *blast*
from *v34 body* **have** *v4-E*: $v4 \in E\ e\ ((x,v3)\#\varrho)$ **by** *blast*
from *ELam*(*3*) *v31* **have** *rr2*: $FV\ e \vdash ((x,v3)\#\varrho) \sqsubseteq ((x,v1)\#\varrho')$ **by** *auto*
from *ELam*(*1*) *v24 v4-E rr2* **show** $v2 \in E\ e\ ((x,v1)\#\varrho')$ **by** *blast*
qed

next
case (*EApp e1 e2*)
from *EApp(3)* **obtain** *f* **and** *v2*::*val* **and** *v2′ v3′* **where** *f-e1*: *VFun f* ∈ *E e1 ϱ* **and**
v2-e2: *v2* ∈ *E e2 ϱ* **and** *v23p-f*: (*v2′*,*v3′*) ∈ *set f* **and** *v2p-v2*: *v2′* ⊑ *v2* **and**
v-v3: *v* ⊑ *v3′* **by** *blast*
from *EApp(4)* **have** *1*: *FV e1* ⊢ *ϱ* ⊑ *ϱ′* **by** *auto*
have *f-f*: *VFun f* ⊑ *VFun f* **by** *auto*
from *EApp(1) f-e1 1 f-f* **have** *f-e1b*: *VFun f* ∈ *E e1 ϱ′* **by** *blast*
from *EApp(4)* **have** *2*: *FV e2* ⊢ *ϱ* ⊑ *ϱ′* **by** *auto*
from *EApp(2) v2-e2 2* **have** *v2-e2b*: *v2* ∈ *E e2 ϱ′* **by** *auto*
from *EApp(5) v-v3* **have** *vp-v3p*: *v′* ⊑ *v3′* **by** (*rule val-le-trans*)
from *f-e1b v2-e2b v23p-f v2p-v2 vp-v3p*
show *?case* **by** *auto*
next
case (*EPrim f e1 e2*)
from *EPrim(3)* **obtain** *n1 n2* **where** *n1-e1*: *VNat n1* ∈ *E e1 ϱ* **and** *n2-e2*: *VNat n2* ∈ *E e2 ϱ* **and**
v: *v* = *VNat* (*f n1 n2*) **by** *blast*
from *EPrim(4)* **have** *1*: *FV e1* ⊢ *ϱ* ⊑ *ϱ′* **by** *auto*
from *EPrim(1) n1-e1 1* **have** *n1-e1b*: *VNat n1* ∈ *E e1 ϱ′* **by** *blast*
from *EPrim(4)* **have** *2*: *FV e2* ⊢ *ϱ* ⊑ *ϱ′* **by** *auto*
from *EPrim(2) n2-e2 2* **have** *n2-e2b*: *VNat n2* ∈ *E e2 ϱ′* **by** *blast*
from *v EPrim(5)* **have** *vp*: *v′* = *VNat* (*f n1 n2*) **by** *auto*
from *n1-e1b n2-e2b vp* **show** *?case* **by** *auto*
next
case (*EIf e1 e2 e3*)
then show *?case*
apply *simp* **apply** *clarify*
apply (*rename-tac n*) **apply** (*rule-tac x*=*n* **in** *exI*) **apply** (*rule conjI*)
apply *force*
apply *force* **done**
qed

— Subsumption is admissible
proposition *e-sub*: ⟦ *v* ∈ *E e ϱ*; *v′* ⊑ *v* ⟧ ⟹ *v′* ∈ *E e ϱ*
apply (*subgoal-tac FV e* ⊢ *ϱ* ⊑ *ϱ*) **using** *change-env-le* **apply** *blast* **apply** *auto* **done**

lemma *env-le-ext*: **fixes** *ϱ*::*env* **assumes** *rr*: *ϱ* ⊑ *ϱ′* **shows** ((*x*,*v*)#*ϱ*) ⊑ ((*x*,*v*)#*ϱ′*)
using *rr* **by** (*simp add*: *env-le-def*)

lemma *change-env*: **fixes** *ϱ*::*env* **assumes** *de*: *v* ∈ *E e ϱ* **and** *rr*: *FV e* ⊢ *ϱ* ⊑ *ϱ′* **shows** *v* ∈ *E e ϱ′*
proof −
have *vv*: *v* ⊑ *v* **by** *auto*
from *de rr vv* **show** *?thesis* **using** *change-env-le* **by** *blast*
qed

lemma *raise-env*: **fixes** *ϱ*::*env* **assumes** *de*: *v* ∈ *E e ϱ* **and** *rr*: *ϱ* ⊑ *ϱ′* **shows** *v* ∈ *E e ϱ′*
using *de rr change-env env-le-def* **by** *auto*

lemma *env-eq-refl*[*simp*]: **fixes** *ϱ*::*env* **shows** *ϱ* ≈ *ϱ* **by** (*simp add*: *env-eq-def*)

lemma *env-eq-ext*: **fixes** *ϱ*::*env* **assumes** *rr*: *ϱ* ≈ *ϱ′* **shows** ((*x*,*v*)#*ϱ*) ≈ ((*x*,*v*)#*ϱ′*)
using *rr* **by** (*simp add*: *env-eq-def*)

lemma *eq-implies-le*: **fixes** *ϱ*::*env* **shows** *ϱ* ≈ *ϱ′* ⟹ *ϱ* ⊑ *ϱ′*
by (*simp add*: *env-le-def env-eq-def*)

lemma *env-swap*: **fixes** *ϱ*::*env* **assumes** *rr*: *ϱ* ≈ *ϱ′* **and** *ve*: *v* ∈ *E e ϱ* **shows** *v* ∈ *E e ϱ′*
using *rr ve* **apply** (*subgoal-tac ϱ* ⊑ *ϱ′*) **prefer** *2* **apply** (*rule eq-implies-le*) **apply** *blast*
apply (*rule raise-env*) **apply** *auto* **done**

lemma *env-strengthen*: ⟦ *v* ∈ *E e ϱ*; ∀ *x*. *x* ∈ *FV e* ⟶ *lookup ϱ′ x* = *lookup ϱ x* ⟧ ⟹ *v* ∈ *E e ϱ′*

using *change-env* **by** *auto*

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**
$V\ (TNat\ n) = VNat\ n$ |
$V\ (TFun\ f) = VFun\ (Vf\ f)$ |
$Vf\ (A \to B) = [(V\ A,\ V\ B)]$ |
$Vf\ (A \sqcap B) = Vf\ A\ @\ Vf\ B$ |
$Vf\ \top = []$

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

function $T :: val \Rightarrow ty$ **and** $Tf :: (val \times val)\ list \Rightarrow funty$ **where**
$T\ (VNat\ n) = TNat\ n$ |
$T\ (VFun\ t) = TFun\ (Tf\ t)$ |
$Tf\ [] = \top$ |
$Tf\ ((v1,v2)\#t) = (T\ v1 \to T\ v2) \sqcap Tf\ t$
by *pat-completeness auto*
termination T **by** *size-change*

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

lemma *sub-inter-left1*: $A <:: C \Longrightarrow A \sqcap B <:: C$
apply (*subgoal-tac* $A \sqcap B <:: A$)
apply (*rule sub-trans*) **apply** *assumption* **apply** *assumption*
apply *blast*
done

lemma *sub-inter-left2*: $B <:: C \Longrightarrow A \sqcap B <:: C$
apply (*subgoal-tac* $A \sqcap B <:: B$)
apply (*rule sub-trans*) **apply** *assumption* **apply** *assumption*
apply *blast*
done

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

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

proposition *vt-id*: **shows** $V\ (T\ v) = v$ **and** $Vf\ (Tf\ t) = t$
by (*induction rule*: *T-Tf.induct*) *force+*

lemma *lookup-tenv*:
$lookup\ \varrho\ x = Some\ v \Longrightarrow lookup\ (Tenv\ \varrho)\ x = Some\ (T\ v)$
by (*induction* ϱ *arbitrary*: $x\ v$) *force+*

proposition *table-mem-sub*:
$(v,\ v') \in set\ t \Longrightarrow Tf\ t <:: (T\ v) \to (T\ v')$
proof (*induction* t *arbitrary*: $v\ v'$)
case *Nil*

```
  then show ?case by auto
next
  case (Cons p t)
  show ?case
  proof (cases p)
    case (Pair v1 v2)
    with Cons show ?thesis
      apply simp
      apply (erule disjE)
       apply force
      apply (subgoal-tac Tf t <:: T v → T v') prefer 2 apply blast
      apply (rule sub-inter-left2) apply assumption done
  qed
qed

lemma Tf-top: Tf t <:: ⊤
proof (induction t)
  case Nil
  then show ?case by auto
next
  case (Cons p t)
  with sub-inter-left2 show ?case by (cases p) auto
qed

lemma le-sub-flip-aux:
  ∀ v v' t t'. n = val-size v + val-size v' + fun-size t + fun-size t' ⟶
   (v ⊑ v' ⟶ T v' <: T v) ∧ (t ≲ t' ⟶ Tf t' <:: Tf t)
proof (induction n rule: nat-less-induct)
  case (1 n)
  show ?case apply clarify apply (rule conjI) apply clarify prefer 2 apply clarify
     prefer 2
  proof −
    fix v::val and v' t t' assume n: n = val-size v + val-size v' + fun-size t + fun-size t'
      and v-vp: v ⊑ v'
    show T v' <: T v
    proof (cases v)
      case (VNat n1)
      from VNat v-vp have vp: v' = VNat n1 by auto
      from VNat vp show ?thesis apply simp using sub-refl by blast
    next
      case (VFun t1)
      from VFun v-vp obtain t2 where vp: v' = VFun t2 and t1-t2: t1 ≲ t2 by auto
      let ?m = val-size (VNat 0) + val-size (VNat 0) + fun-size t1 + fun-size t2
      from 1 t1-t2 n VFun vp have t2-t1: Tf t2 <:: Tf t1
        apply simp
        apply (erule-tac x=?m in allE)
        apply (erule impE) apply force apply simp apply (erule-tac x=VNat 0 in allE)
        apply (erule-tac x=VNat 0 in allE)
        apply (erule-tac x=t1 in allE)
        apply (erule-tac x=t2 in allE) apply auto done
      from t2-t1 VFun vp show ?thesis by auto
    qed
  next
    fix v v' t t' assume n: n = val-size v + val-size v' + fun-size t + fun-size t'
      and t-tp: t ≲ t'
    show Tf t' <:: Tf t
    proof (cases t)
      case Nil
      from Nil have Tf t = ⊤ by simp
      then show ?thesis using Tf-top by auto
    next
```

```
  case (Cons a t1)
  show ?thesis
  proof (cases a)
    case (Pair v1 v2)
    from Cons Pair show ?thesis apply simp
    proof
      from Cons Pair have v12: (v1,v2) ∈ set t by auto
      from t-tp v12 obtain v3 v4 where v34: (v3,v4) ∈ set t' and
        v13: v1 ⊑ v3 and v31: v3 ⊑ v1 and v24: v2 ⊑ v4 and v42: v4 ⊑ v2 by blast
      have Tv3-Tv1: T v3 ≈ T v1
      proof −
        let ?m = val-size v1 + val-size v3
        from v12 have sv1: val-size v1 < fun-size t using val-size-mem-l by auto
        from v34 have sv3: val-size v3 < fun-size t' using val-size-mem-l by auto
        from 1 v13 n sv1 sv3 have Tv31: T v3 <: T v1
          apply (erule-tac x=?m in allE) apply (erule impE) apply force
          apply (erule-tac x=v1 in allE) apply (erule-tac x=v3 in allE)
          apply (erule-tac x=[] in allE)apply (erule-tac x=[] in allE)
          apply (erule impE) defer apply blast apply simp
          done
        from 1 v31 n sv1 sv3 have Tv13: T v1 <: T v3
          apply (erule-tac x=?m in allE) apply (erule impE) apply force
          apply (erule-tac x=v3 in allE) apply (erule-tac x=v1 in allE)
          apply (erule-tac x=[] in allE) apply (erule-tac x=[] in allE) apply auto done
        from Tv13 Tv31 show ?thesis unfolding ty-eq-def by blast
      qed
      have Tv4-Tv2: T v4 ≈ T v2
      proof −
        let ?m = val-size v2 + val-size v4
        from v12 have sv2: val-size v2 < fun-size t using val-size-mem-r by auto
        from v34 have sv4: val-size v4 < fun-size t' using val-size-mem-r by auto
        from 1 v42 n sv2 sv4 have Tv2-v4: T v2 <: T v4
          apply (erule-tac x=?m in allE) apply (erule impE) apply force
          apply (erule-tac x=v4 in allE) apply (erule-tac x=v2 in allE)
          apply (erule-tac x=[] in allE) apply (erule-tac x=[] in allE)
          apply (erule impE) defer apply blast apply simp
          done
        from 1 v24 n sv2 sv4 have Tv4-v2: T v4 <: T v2
          apply (erule-tac x=?m in allE) apply (erule impE) apply force
          apply (erule-tac x=v2 in allE) apply (erule-tac x=v4 in allE)
          apply (erule-tac x=[] in allE) apply (erule-tac x=[] in allE)
          apply (erule impE) defer apply blast apply simp
          done
        from Tv2-v4 Tv4-v2 show ?thesis unfolding ty-eq-def by blast
      qed
      from Tv3-Tv1 Tv4-Tv2 have T34-T12: T v3 → T v4 <:: T v1 → T v2
        unfolding ty-eq-def by blast
      from v34 have tp-T34: Tf t' <:: T v3 → T v4 using table-mem-sub by blast
      from tp-T34 T34-T12 show Tf t' <:: T v1 → T v2 by (rule sub-trans)
    next
      let ?m = fun-size t' + fun-size t1
      from Cons Pair t-tp have t1-tp: t1 ≲ t' by auto
      from 1 n t1-tp Cons Pair
      show Tf t' <:: Tf t1
        apply (erule-tac x=?m in allE) apply (erule impE) apply force
        apply (erule-tac x=VNat 0 in allE) apply (erule-tac x=VNat 0 in allE)
        apply (erule-tac x=t1 in allE) apply (erule-tac x=t' in allE)
        apply auto done
      qed
  qed
qed
```

```
  qed
qed

proposition le-sub-flip: v ⊑ v' ⟹ T v' <: T v using le-sub-flip-aux by blast

lemma le-sub-fun-flip: t ≲ t' ⟹ Tf t' <:: Tf t using le-sub-flip-aux by blast

lemma Tf-append: Tf (t1 @ t2) <:: Tf t1 ⊓ Tf t2
proof (induction t1)
  case Nil
  then show ?case
    apply simp apply (rule sub-inter-r) using Tf-top apply blast
    apply (rule fsub-refl) done
next
  case (Cons a t1)
  then show ?case
    apply (case-tac a) apply simp apply (rule sub-inter-r) apply (rule sub-inter-r)
      apply (rule sub-inter-left1) apply (rule fsub-refl) apply (rule sub-inter-left2)
     apply (subgoal-tac Tf t1 ⊓ Tf t2 <:: Tf t1) prefer 2 apply (rule sub-inter-left1)
      apply (rule fsub-refl) apply (rule sub-trans) apply assumption apply assumption
    apply (rule sub-inter-left2) apply (subgoal-tac Tf t1 ⊓ Tf t2 <:: Tf t2)
     prefer 2 apply (rule sub-inter-left2)
     apply (rule fsub-refl) apply (rule sub-trans) apply assumption apply assumption done
qed

lemma append-Tf: Tf t1 ⊓ Tf t2 <:: Tf (t1 @ t2)
proof (induction t1)
  case Nil
  then show ?case apply simp apply (rule sub-inter-left2) apply (rule fsub-refl) done
next
  case (Cons p t1)
  then show ?case
    apply (cases p) apply simp apply (rule sub-inter-r)
     apply (rule sub-inter-left1) apply (rule sub-inter-left1) apply (rule fsub-refl)
    apply (rename-tac v1 v2)
    apply (subgoal-tac ((T v1 → T v2) ⊓ Tf t1) ⊓ Tf t2 <:: Tf t1 ⊓ Tf t2)
     prefer 2 apply (rule sub-inter-r) apply (rule sub-inter-left1)
      apply (rule sub-inter-left2) apply (rule fsub-refl)
     apply (rule sub-inter-left2) apply (rule fsub-refl)
    apply (rule sub-trans) apply assumption apply assumption done
qed

proposition tv-id: shows T (V A) ≈ A and Tf (Vf F) ≃ F
proof (induction rule: V-Vf.induct)
  case (1 n)
  then show ?case apply (simp add: ty-eq-def) apply (rule sub-refl) done
next
  case (2 f)
  then show ?case
    apply (simp add: ty-eq-def) apply (rule conjI) apply (rule sub-funty)
    using le-sub-flip fty-eq-def apply blast
    apply (rule sub-funty) using le-sub-flip fty-eq-def apply blast
    done
next
  case (3 A B)
  then show ?case
    apply (simp add: fty-eq-def) apply(rule conjI) apply (rule sub-inter-left1)
    using ty-eq-def apply blast
    apply (rule sub-inter-r) using ty-eq-def apply blast
    apply blast done
next
```

```
  case (4 A B)
  then show ?case
    using fty-eq-def apply simp apply (rule conjI) apply (rule sub-inter-r)
     apply (subgoal-tac Tf (Vf A @ Vf B) <:: Tf (Vf A) ⊓ Tf (Vf B))
      prefer 2 using Tf-append apply simp
     apply (subgoal-tac Tf (Vf A) ⊓ Tf (Vf B) <:: A)
      apply (rule sub-trans) apply assumption apply assumption
     apply (rule sub-inter-left1) apply blast
    apply (subgoal-tac Tf (Vf A @ Vf B) <:: Tf (Vf A) ⊓ Tf (Vf B))
     prefer 2 using Tf-append apply simp
    apply (subgoal-tac Tf (Vf A) ⊓ Tf (Vf B) <:: B)
     apply (rule sub-trans) apply assumption apply assumption
    apply (rule sub-inter-left2) apply blast
   apply (subgoal-tac Tf (Vf A) ⊓ Tf (Vf B) <:: Tf (Vf A @ Vf B))
    prefer 2 apply (rule append-Tf)
   apply (subgoal-tac A ⊓ B <:: Tf (Vf A) ⊓ Tf (Vf B))
    apply (rule sub-trans) apply blast
    apply blast apply (rule sub-inter-r) apply (rule sub-inter-left1) apply blast
   apply (rule sub-inter-left2) apply blast done
next
  case 5
  then show ?case  using fty-eq-def by auto
qed

lemma denot-lam-implies-ts:
  assumes et: ∀ v ϱ. v ∈ E e ϱ ⟶ Tenv ϱ ⊢ e : T v and
      fe: ∀ v1 v2. (v1, v2) ∈ set f ⟶ v2 ∈ E e ((x, v1) # ϱ)
  shows Tenv ϱ ⊢ ELam x e : TFun (Tf f)
  using et fe
proof (induction f)
  case Nil
  then show ?case by auto
next
  case (Cons a f)
  then show ?case
  proof (cases a)
    case (Pair v v')
    {
      assume 1: Tenv ϱ ⊢ ELam x e : TFun (Tf f) and
        2: ∀ v ϱ. v ∈ E e ϱ ⟶ Tenv ϱ ⊢ e : T v and
        3: ∀ v1 v2.
          (v1 = v ∧ v2 = v' ⟶ v' ∈ E e ((x, v) # ϱ)) ∧
          ((v1, v2) ∈ set f ⟶ v2 ∈ E e ((x, v1) # ϱ))
      from 3 have 4: v' ∈ E e ((x,v)#ϱ) by simp
      from 2 4 have 5: Tenv ((x,v)#ϱ) ⊢ e : T v'
        apply (erule-tac x=v' in allE) apply (erule-tac x=(x,v)#ϱ in allE) apply simp done
      from 5 have (x, T v) # Tenv ϱ ⊢ e : T v' by simp
    }
    from Cons Pair this show ?thesis
      apply simp apply (rule wt-inter) apply (rule wt-lam) apply blast apply blast done
  qed
qed

theorem denot-implies-ts:
  assumes ve: v ∈ E e ϱ shows Tenv ϱ ⊢ e : T v
  using ve
proof (induction e arbitrary: v ϱ)
  case (EVar x)
  then show ?case
    apply simp apply (erule exE) apply (erule conjE)
    apply (subgoal-tac lookup (Tenv ϱ) x = Some (T v'))
```

```
    prefer 2 apply (rule lookup-tenv) apply assumption
   apply (rule wt-sub) apply blast
   apply (rule le-sub-flip) apply assumption
   done
next
 case (ENat x)
 then show ?case by auto
next
 case (ELam x e)
 then show ?case
   apply simp apply clarify apply simp
   apply (rule denot-lam-implies-ts)
    apply blast apply blast done
next
 case (EApp e1 e2)
 then show ?case
   apply simp apply clarify
   apply (subgoal-tac Tenv ϱ ⊢ e1 : T (VFun f))
    prefer 2 apply assumption apply (subgoal-tac Tf f <:: T v2' → T v3')
    prefer 2 apply (rule table-mem-sub) apply assumption
   apply (subgoal-tac Tenv ϱ ⊢ EApp e1 e2 : T v3') apply (rule wt-sub)
     apply assumption
    apply (rule le-sub-flip) apply assumption
   apply (subgoal-tac Tenv ϱ ⊢ e1 : TFun (T v2' → T v3')) prefer 2
    apply (rule wt-sub) apply assumption apply simp apply (rule sub-funty) apply assumption
   apply (rule wt-app) apply assumption
   apply (subgoal-tac Tenv ϱ ⊢ e2 : T v2) prefer 2 apply assumption
   apply (rule wt-sub) apply assumption apply (rule le-sub-flip) apply assumption  done
next
 case (EPrim f e1 e2)
 then show ?case
   apply simp apply clarify
   apply (subgoal-tac Tenv ϱ ⊢ e1 : T (VNat n1)) prefer 2 apply assumption
   apply (subgoal-tac Tenv ϱ ⊢ e2 : T (VNat n2)) prefer 2 apply assumption
   apply force done
next
 case (EIf e1 e2 e3)
 then show ?case
   apply simp apply clarify
   apply (subgoal-tac Tenv ϱ ⊢ e1 : T (VNat n)) prefer 2 apply assumption
   apply (case-tac n) apply simp
    apply (subgoal-tac Tenv ϱ ⊢ e3 : T v) prefer 2 apply assumption
    apply blast
   apply simp
   apply (subgoal-tac Tenv ϱ ⊢ e2 : T v) prefer 2 apply assumption
   apply (rule wt-ifnz) apply assumption apply simp apply assumption done
qed

lemma venv-lookup: assumes lx: lookup Γ x = Some A shows lookup (Venv Γ) x = Some (V A)
 using lx
proof (induction Γ arbitrary: A)
 case Nil
 then show ?case by auto
next
 case (Cons b Γ)
 obtain x' B where b = (x',B) by (cases b) auto
 with Cons show ?case by (cases x = x') auto
qed

lemma append-fun-equiv: ⟦ t1' ~ t1; t2' ~ t2 ⟧ ⟹ t1' @ t2' ~ t1 @ t2
 apply (simp add: val-eq-def fun-eq-def)
```

using *append-fun-le* **apply** *blast*
done

lemma *append-eq-symm*: *t2 @ t1 ∼ t1 @ t2*
unfolding *fun-eq-def val-eq-def* **apply** (*rule conjI*)
apply (*rule append-leq-symm*)
apply (*rule append-leq-symm*)
done

lemma *sub-le-flip*: (*A <: B ⟶ V B ⊑ V A*) ∧ (*f1 <:: f2 ⟶* (*Vf f2*) ≲ (*Vf f1*))
proof (*induction rule*: *subtype-fsubtype.induct*)
case (*sub-trans T1 T2 T3*)
then show *?case* **using** *fun-le-trans* **by** *blast*
qed *force+*

theorem *ts-implies-denot*:
assumes *wte*: Γ ⊢ *e* : *A* **shows** *V A* ∈ *E e* (*Venv* Γ)
using *wte*
proof (*induction* Γ *e A rule*: *wt.induct*)
case (*wt-var* Γ *x T*)
then show *?case*
apply *simp*
apply (*subgoal-tac lookup* (*Venv* Γ) *x* = *Some* (*V T*))
prefer *2* **apply** (*rule venv-lookup*)
apply *assumption*
apply (*rule-tac x*=*V T* **in** *exI*)
apply *force*
done
next
case (*wt-sub* Γ *e A B*)
then show *?case*
apply (*subgoal-tac V B ⊑ V A*)
prefer *2* **using** *sub-le-flip* **apply** *blast*
apply (*rule e-sub*)
apply *auto*
done
qed *fastforce+*

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*)
by *auto*

lemma *subst-prim*: *subst x v* (*EPrim f e1 e2*) = *EPrim f* (*subst x v e1*) (*subst x v e2*)
by *auto*

lemma *subst-lam-eq*: *subst x v* (*ELam x e*) = *ELam x e* **by** *auto*

lemma *subst-lam-neq*: *y* ≠ *x* ⟹ *subst x v* (*ELam y e*) = *ELam y* (*subst x v e*) **by** *simp*

lemma *subst-if*: *subst x v* (*EIf e1 e2 e3*) = *EIf* (*subst x v e1*) (*subst x v e2*) (*subst x v e3*)
by *auto*

lemma *substitution*:
fixes Γ::*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 $B \in E\ (subst\ x\ v\ e)\ \Gamma$
using *wte wtv gp v*
proof (*induction arbitrary*: *v A B* Γ *x rule*: *E.induct*)
case (*1 n* ϱ)
then show *?case* **by** *auto*
next
case (*2 x* ϱ *v A B* Γ *x'*)
then show *?case*
apply (*simp only*: *env-eq-def*)
apply (*cases* $x = x'$)
apply *simp* **apply** *clarify*
apply (*rule env-strengthen*)
apply (*rule e-sub*)
apply *auto*
done
next
case (*3 x e* ϱ *v A B* Γ *x'*)
then show *?case*
apply (*case-tac* $x' = x$) **apply** (*simp only*: *subst-lam-eq*)
apply (*rule env-strengthen*) **apply** *assumption* **apply** (*simp add*: *env-eq-def*)
apply (*simp only*: *subst-lam-neq*) **apply** (*erule e-lam-elim*)
apply (*rule e-lam-intro*)
apply *assumption* **apply** *clarify* **apply** (*erule-tac x=v1* **in** *allE*) **apply** (*erule-tac x=v2* **in** *allE*)
apply *clarify*
apply (*subgoal-tac* $(x,v1)\#\varrho \approx (x',A)\#(x,v1)\#\Gamma$)
prefer *2* **apply** (*simp add*: *env-eq-def*)
apply *blast*
done
next
case (*4 e1 e2* ϱ)
then show *?case*
apply (*simp only*: *subst-app*)
apply (*erule e-app-elim*)
apply (*rule e-app-intro*)
apply *auto*
done
next
case (*5 f e1 e2* ϱ)
then show *?case*
apply (*simp only*: *subst-prim*) **apply** (*erule e-prim-elim*) **apply** *simp*
apply (*rule-tac x=n1* **in** *exI*) **apply** (*rule conjI*)
apply *force*
apply (*rule-tac x=n2* **in** *exI*)
apply *auto*
done
next
case (*6 e1 e2 e3* ϱ)
then show *?case*
apply (*simp only*: *subst-if*) **apply** (*erule e-if-elim*) **apply** (*rename-tac n*)
apply *simp*
apply (*case-tac* $n = 0$) **apply** (*rule-tac x=0* **in** *exI*)
apply *force*
apply (*rule-tac x=n* **in** *exI*) **apply** *simp* **done**
qed

12.2 Reduction preserves denotation

lemma *subject-reduction*: **fixes** *e*::*exp* **assumes** *v*: $v \in E\ e\ \varrho$ **and** *r*: $e \longrightarrow e'$ **shows** $v \in E\ e'\ \varrho$
using *r v*
proof (*induction arbitrary*: *v* ϱ *rule*: *reduce.induct*)
case (*beta v x e v'* ϱ)
then show *?case* **apply** (*simp only*: *is-val-def*)
apply (*erule e-app-elim*) **apply** (*erule e-lam-elim*) **apply** *clarify*
apply (*rename-tac f v2 v2' v3' f'*)
apply (*erule-tac x=v2'* **in** *allE*) **apply** (*erule-tac x=v3'* **in** *allE*) **apply** *clarify*
apply (*subgoal-tac v3'* $\in E$ (*subst x v e*) ϱ) **prefer** *2* **apply** (*rule substitution*)
apply (*subgoal-tac v3'* $\in E\ e$ ((x,*v2*)#ϱ)) **prefer** *2* **apply** (*rule raise-env*)
apply *assumption* **apply** (*simp add*: *env-le-def*) **prefer** *2* **apply** (*rule env-strengthen*)
apply *assumption* **apply** *force* **prefer** *2* **apply** (*subgoal-tac* (x,*v2*)#$\varrho \approx$ (x,*v2*)#ϱ) **prefer** *2*
apply (*simp add*: *env-eq-def*) **apply** *assumption* **apply** *assumption*
apply *simp*
apply *simp*
apply (*rule e-sub*)
apply *assumption*
apply (*rule val-le-trans*)
apply *blast*
apply *force*
done
qed *force+*

theorem *preservation*: **assumes** *v*: $v \in E\ e\ \varrho$ **and** *rr*: $e \longrightarrow* e'$ **shows** $v \in E\ e'\ \varrho$
using *rr v subject-reduction* **by** (*induction arbitrary*: ϱ *v*) *auto*

lemma *canonical-nat*: **assumes** *v*: $VNat\ n \in E\ v\ \varrho$ **and** *vv*: *isval v* **shows** $v = ENat\ n$
using *v vv* **by** (*cases v*) *auto*

lemma *canonical-fun*: **assumes** *v*: $VFun\ f \in E\ v\ \varrho$ **and** *vv*: *isval v* **shows** $\exists\ x\ e.\ v = ELam\ x\ e$
using *v vv* **by** (*cases v*) *auto*

12.3 Progress

theorem *progress*: **assumes** *v*: $v \in E\ e\ \varrho$ **and** *r*: $\varrho = []$ **and** *fve*: $FV\ e = \{\}$
shows *is-val e* $\vee$ ($\exists\ e'.\ e \longrightarrow e'$)
using *v r fve*
proof (*induction arbitrary*: *v rule*: *E.induct*)
case (*4 e1 e2* ϱ)
show *?case*
apply (*rule e-app-elim*) **using** *4*(*3*) **apply** *assumption*
apply (*cases is-val e1*)
apply (*cases is-val e2*)
apply (*frule canonical-fun*) **apply** *force* **apply** (*erule exE*)+ **apply** *simp* **apply** (*rule disjI2*)
apply (*rename-tac x e*)
apply (*rule-tac x=subst x e2 e* **in** *exI*)
apply (*rule beta*) **apply** *simp*
using *4* **apply** *simp*
apply *blast*
using *4* **apply** *simp*
apply *blast* **done**
next
case (*5 f e1 e2* ϱ)
show *?case*
apply (*rule e-prim-elim*) **using** *5*(*3*) **apply** *assumption*
using *5* **apply** (*case-tac isval e1*)
apply (*case-tac isval e2*)
apply (*subgoal-tac e1 = ENat n1*) **prefer** *2* **using** *canonical-nat* **apply** *blast*
apply (*subgoal-tac e2 = ENat n2*) **prefer** *2* **using** *canonical-nat* **apply** *blast*

```
      apply force
     apply force
    apply force done
next
  case (6 e1 e2 e3 ϱ)
  show ?case
    apply (rule e-if-elim)
    using 6(4) apply assumption
    apply (cases isval e1)
     apply (rename-tac n)
     apply (subgoal-tac e1 = ENat n) prefer 2 apply (rule canonical-nat) apply blast apply blast
     apply (rule disjI2) apply (case-tac n = 0) apply force apply force
    apply (rule disjI2)
    using 6 apply (subgoal-tac ∃ e1'. e1 ⟶ e1') prefer 2 apply force
    apply clarify apply (rename-tac e1')
    apply (rule-tac x=EIf e1' e2 e3 in exI)
    apply (rule if-cond) apply assumption
    done
qed auto
```

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 (λv. (v,good v)) (λv. (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
  using l EE
proof (induction ϱ' arbitrary: x A ϱ)
  case Nil
  show ?case apply (rule genv-any-nil-inv) using Nil by auto
next
  case (Cons a ϱ')
  show ?case
    apply (rule genv-any-cons-inv)
     using Cons apply force
    apply (rename-tac x')apply clarify
    using Cons apply (case-tac x = x')
     apply force
    apply force
    done
qed

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

lemma good-prod-inj: inj-on good-prod (fset A)
```

unfolding *inj-on-def* **apply** *auto* **done**

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 −
interpret *ge*: *comp-fun-commute* (*good-entry x e ϱ*) ∘ *good-prod*
unfolding *comp-fun-commute-def* **by** *auto*
show *good-fun f x e ϱ*
= *ffold* ((*good-entry x e ϱ*) ∘ *good-prod*) *True f*
using *good-prod-inj*[*of f*] *good-fun-def*
ffold-fimage[*of good-prod f good-entry x e ϱ True*] **by** *auto*
qed

lemma *gfun-elim*: *w* ∈ *good* (*VFun f*) ⟹ ∃ *x e ϱ*. *w* = *BClos x e ϱ* ∧ *good-fun f x e ϱ*
using *good-fun-def* **by** *auto*

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 (*induction f arbitrary*: *x e ϱ*)
case *empty*
interpret *ge*: *comp-fun-commute* (*good-entry x e ϱ*)
unfolding *comp-fun-commute-def* **by** *auto*
from *empty* **show** *?case* **using** *good-fun-def2*
by (*simp add*: *comp-fun-commute.ffold-empty ge.comp-comp-fun-commute*)
next
case (*insert p f*)
interpret *ge*: *comp-fun-commute* (*good-entry x e ϱ*) ∘ *good-prod*
unfolding *comp-fun-commute-def* **by** *auto*
have *good-fun* (*finsert p f*) *x e ϱ*
= *ffold* ((*good-entry x e ϱ*) ∘ *good-prod*) *True* (*finsert p f*) **by** (*simp add*: *good-fun-def2*)
also from *insert*(*1*) **have** *...* = ((*good-entry x e ϱ*) ∘ *good-prod*) *p*
(*ffold* ((*good-entry x e ϱ*) ∘ *good-prod*) *True f*) **by** *simp*
finally have *1*: *good-fun* (*finsert p f*) *x e ϱ*
= ((*good-entry x e ϱ*) ∘ *good-prod*) *p* (*ffold* ((*good-entry x e ϱ*) ∘ *good-prod*) *True f*) **.**
show *?case*
proof
assume *2*: *good-fun* (*finsert p f*) *x e ϱ*
show ∀ *v1 v2*. (*v1*, *v2*) ∈ *fset* (*finsert p f*) ⟶
(∀ *v*∈*good v1*. ∃ *v′*. (*x*, *v*) # *ϱ* ⊢ *e* ⇓ *v′* ∧ *v′* ∈ *good v2*)
proof *clarify*
fix *v1 v2 v* **assume** *3*: (*v1*, *v2*) ∈ *fset* (*finsert p f*) **and** *4*: *v* ∈ *good v1*
from *3* **have** (*v1*,*v2*) = *p* ∨ (*v1*,*v2*) ∈ *fset f* **by** *auto*
from *this* **show** ∃ *v′*. (*x*, *v*) # *ϱ* ⊢ *e* ⇓ *v′* ∧ *v′* ∈ *good v2*
proof
assume *v12-p*: (*v1*,*v2*) = *p*
from *1 v12-p*[*THEN sym*] *2 4* **show** *?thesis* **by** *simp*
next
assume *v12-f*: (*v1*,*v2*) ∈ *fset f*
from *1 2* **have** *5*: *good-fun f x e ϱ* **apply** *simp*
apply (*cases* (*good-prod p*)) **by** (*auto simp*: *good-fun-def2*)
from *v12-f 5 4 insert*(*2*)[*of x e ϱ*] **show** *?thesis* **by** *auto*
qed
qed
next
assume *2*: ∀ *v1 v2*. (*v1*, *v2*) ∈ *fset* (*finsert p f*) ⟶
(∀ *v*∈*good v1*. ∃ *v′*. (*x*, *v*) # *ϱ* ⊢ *e* ⇓ *v′* ∧ *v′* ∈ *good v2*)
have *3*: *good-entry x e ϱ* (*good-prod p*) *True*
apply (*cases p*) **apply** *simp* **apply** *clarify*

```
    proof −
      fix v1 v2 v
      assume p: p = (v1,v2) and v-v1: v ∈ good v1
      from p have (v1,v2) ∈ fset (finsert p f) by simp
      from this 2 v-v1 show ∃ v'. (x, v) # ϱ ⊢ e ⇓ v' ∧ v' ∈ good v2 by blast
    qed
    from insert(2) 2 have 4: good-fun f x e ϱ by auto
    have (good-entry x e ϱ ∘ good-prod) p
     (ffold (good-entry x e ϱ ∘ good-prod) True f)
      apply simp apply (cases good-prod p)
      apply (rename-tac a b c)
      apply (case-tac a) apply simp
      apply (rule conjI) prefer 2 using 4 good-fun-def2 apply force
      using 3 apply force done
    from this 1 show good-fun (finsert p f) x e ϱ
      unfolding good-fun-def by simp
  qed
qed

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
  using gfun-mem-iff by blast

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 using gfun-mem-iff[of f x e ϱ] by simp

lemma sub-good: fixes v::val assumes wv: w ∈ good v and vp-v: v' ⊑ v shows w ∈ good v'
proof (cases v)
  case (VNat n)
  from this wv vp-v show ?thesis by auto
next
  case (VFun t1)
  from vp-v VFun obtain t2 where b: v' = VFun t2 and t2-t1: fset t2 ⊆ fset t1 by auto
  from wv VFun obtain x e ϱ where w: w = BClos x e ϱ by auto
  from w wv VFun have gt1: good-fun t1 x e ϱ by (simp add: good-fun-def)
  have gt2: good-fun t2 x e ϱ apply (rule gfun-intro) apply clarify
  proof −
    fix v1 v2 w1
    assume v12: (v1,v2) ∈ fset t2 and w1-v1: w1 ∈ good v1
    from v12 t2-t1 have v12-t1: (v1,v2) ∈ fset t1 by blast
    from gt1 v12-t1 w1-v1 show ∃ v'. (x, w1) # ϱ ⊢ e ⇓ v' ∧ v' ∈ good v2
      by (simp add: gfun-mem)
  qed
  from gt2 b w show ?thesis by (simp add: good-fun-def)
qed
```

12.5 Denotational semantics sound wrt. big-step

```
lemma denot-terminates: assumes vp-e: v' ∈ E e ϱ' and ge: good-env ϱ ϱ'
  shows ∃ v. ϱ ⊢ e ⇓ v ∧ v ∈ good v'
  using vp-e ge
proof (induction arbitrary: v' ϱ rule: E.induct)
  case (1 n ϱ) — ENat
  then show ?case by auto
next — EVar
  case (2 x ϱ v' ϱ')
  from 2 obtain v1 where lx-vpp: lookup ϱ x = Some v1 and vp-v1: v' ⊑ v1 by auto
  from lx-vpp 2(2) obtain v2 where lx: lookup ϱ' x = Some v2 and v2-v1: v2 ∈ good v1
    using lookup-good[of ϱ x v1 ϱ'] by blast
  from lx have x-v2: ϱ' ⊢ EVar x ⇓ v2 by auto
  from v2-v1 vp-v1 have v2-vp: v2 ∈ good v' using sub-good by blast
```

```
  from x-v2 v2-vp show ?case by blast
next — ELam
  case (3 x e ϱ v' ϱ')
  have 1: ϱ' ⊢ ELam x e ⇓ BClos x e ϱ' by auto
  have 2: BClos x e ϱ' ∈ good v'
  proof −
    from 3(2) obtain t where vp: v' = VFun t and
      body: ∀ v1 v2. (v1, v2) ∈ fset t ⟶ v2 ∈ E e ((x, v1) # ϱ) by blast
    have gt: good-fun t x e ϱ' apply (rule gfun-intro) apply clarify
    proof −
      fix v1 v2 w1 assume v12-t: (v1,v2) ∈ fset t and w1-v1: w1 ∈ good v1
      from v12-t body have v2-Ee: v2 ∈ E e ((x, v1) # ϱ) by blast
      from 3(3) w1-v1 have ge: good-env ((x,w1)#ϱ') ((x,v1)#ϱ) by auto
      from v12-t v2-Ee ge 3(1)[of v1 v2 t v2]
      show ∃ v'. (x, w1) # ϱ' ⊢ e ⇓ v' ∧ v' ∈ good v2 by blast
    qed
    from vp gt show ?thesis unfolding good-fun-def by simp
  qed
  from 1 2 show ?case by blast
next — EApp
  case (4 e1 e2 ϱ v' ϱ')
  from 4(3) show ?case
  proof
    fix t v2 and v2'::val and v3' assume t-Ee1: VFun t ∈ E e1 ϱ and v2-Ee2: v2 ∈ E e2 ϱ and
      v23-t: (v2',v3') ∈ fset t and v2p-v2: v2' ⊑ v2 and vp-v3p: v' ⊑ v3'
    from 4(1) t-Ee1 4(4) obtain w1 where e1-w1: ϱ' ⊢ e1 ⇓ w1 and
      w1-t: w1 ∈ good (VFun t) by blast
    from 4(2) v2-Ee2 4(4) obtain w2 where e2-w2: ϱ' ⊢ e2 ⇓ w2 and w2-v2: w2 ∈ good v2 by blast
    from w1-t obtain x e ϱ1 where w1: w1 = BClos x e ϱ1 and gt: good-fun t x e ϱ1
      by (auto simp: good-fun-def)
    from w2-v2 v2p-v2 have w2-v2p: w2 ∈ good v2' by (rule sub-good)
    from v23-t gt w2-v2p obtain w3 where e-w3: (x,w2)#ϱ1 ⊢ e ⇓ w3 and w3-v3p: w3 ∈ good v3'
      using gfun-mem[of v2' v3' t x e ϱ1] by blast
    from w3-v3p vp-v3p have w3-vp: w3 ∈ good v' by (rule sub-good)
    from e1-w1 e2-w2 w1 e-w3 w3-vp show ∃ v. ϱ' ⊢ EApp e1 e2 ⇓ v ∧ v ∈ good v' by blast
  qed
next — EPrim
  case (5 f e1 e2 ϱ v' ϱ')
  from 5(3) show ?case
  proof
    fix n1 n2 assume n1-e1: VNat n1 ∈ E e1 ϱ and n2-e2: VNat n2 ∈ E e2 ϱ and
      vp: v' = VNat (f n1 n2)
    from 5(1)[of VNat n1 ϱ'] n1-e1 5(4) have e1-w1: ϱ' ⊢ e1 ⇓ BNat n1 by auto
    from 5(2)[of VNat n2 ϱ'] n2-e2 5(4) have e2-w2: ϱ' ⊢ e2 ⇓ BNat n2 by auto
    from e1-w1 e2-w2 have 1: ϱ' ⊢ EPrim f e1 e2 ⇓ BNat (f n1 n2) by blast
    from vp have 2: BNat (f n1 n2) ∈ good v' by auto
    from 1 2 show ∃ v. ϱ' ⊢ EPrim f e1 e2 ⇓ v ∧ v ∈ good v' by auto
  qed
next — EIf
  case (6 e1 e2 e3 ϱ v' ϱ')
  from 6(4) show ?case
  proof
    fix n assume n-e1: VNat n ∈ E e1 ϱ and els: n = 0 ⟶ v' ∈ E e3 ϱ and
      thn: n ≠ 0 ⟶ v' ∈ E e2 ϱ
    from 6(1)[of VNat n ϱ'] n-e1 6(5) have e1-w1: ϱ' ⊢ e1 ⇓ BNat n by auto
    show ∃ v. ϱ' ⊢ EIf e1 e2 e3 ⇓ v ∧ v ∈ good v'
    proof (cases n = 0)
      case True
      from 6(2)[of n v' ϱ'] True els 6(5) obtain w3 where
        e3-w3: ϱ' ⊢ e3 ⇓ w3 and w3-vp: w3 ∈ good v' by blast
      from e1-w1 True e3-w3 w3-vp show ?thesis by blast
```

```
    next
      case False
      from 6(3)[of n v' ϱ'] False thn 6(5) obtain w2 where
        e2-w2: ϱ' ⊢ e2 ⇓ w2 and w2-vp: w2 ∈ good v' by blast
      from e1-w1 False e2-w2 have ϱ' ⊢ EIf e1 e2 e3 ⇓ w2
        using eval-if1[of ϱ' e1 n e2 w2 e3] by simp
      from this w2-vp show ?thesis by (rule-tac x=w2 in exI) simp
    qed
  qed
qed

theorem sound-wrt-op-sem:
  assumes E-e-n: E e [] = E (ENat n) [] and fv-e: FV e = {} shows e ⇓ ONat n
proof −
  have VNat n ∈ E (ENat n) [] by simp
  with E-e-n have 1: VNat n ∈ E e [] by simp
  have 2: good-env [] [] by auto
  from 1 2 obtain v where e-v: [] ⊢ e ⇓ v and v-n: v ∈ good (VNat n) using denot-terminates by blast
  from v-n have v: v = BNat n by auto
  from e-v fv-e obtain v' ob where e-vp: e ⟶* v' and
    vp-ob: observe v' ob and v-ob: bs-observe v ob using sound-wrt-small-step by blast
  from e-vp vp-ob v-ob v show ?thesis unfolding run-def by (case-tac ob) auto
qed

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) ⊔ (VNat n') = (if n = n' then Some (VNat n) else None) |
  (VFun f) ⊔ (VFun f') = Some (VFun (f |∪| f')) |
  v ⊔ v' = None

lemma combine-values:
  assumes vv: isval v and v1v: v1 ∈ E v ϱ and v2v: v2 ∈ E v ϱ
  shows ∃ v3. v3 ∈ E v ϱ ∧ (v1 ⊔ v2 = Some v3)
  using vv v1v v2v by (induction v arbitrary: v1 v2 ϱ) auto

lemma le-union1: fixes v1::val assumes v12: v1 ⊔ v2 = Some v12 shows v1 ⊑ v12
proof (cases v1)
  case (VNat n1) hence v1: v1=VNat n1 by simp
  show ?thesis
  proof (cases v2)
    case (VNat n2) with v1 v12 show ?thesis by (cases n1=n2) auto
  next
    case (VFun x2) with v1 v12 show ?thesis by auto
  qed
next
  case (VFun t2) from VFun have v1: v1=VFun t2 by simp
  show ?thesis
  proof (cases v2)
    case (VNat n1) with v1 v12 show ?thesis by auto
  next
    case (VFun n2) with v1 v12 show ?thesis by auto
```

```
  qed
qed

lemma le-union2: v1 ⊔ v2 = Some v12 ⟹ v2 ⊑ v12
  apply (cases v1)
   apply (cases v2)
    apply auto
   apply (rename-tac x1 x1')
   apply (case-tac x1 = x1')
    apply auto
  apply (cases v2)
   apply auto
  done

lemma le-union-left: ⟦ v1 ⊔ v2 = Some v12; v1 ⊑ v3; v2 ⊑ v3 ⟧ ⟹ v12 ⊑ v3
  apply (cases v1) apply (cases v2) apply force+ done

lemma e-val: isval v ⟹ ∃ v'. v' ∈ E v ϱ
  by (cases v) auto

lemma reverse-subst-lam:
  assumes fl: VFun f ∈ E (ELam x e) ϱ
    and vv: is-val v and ls: ELam x e = ELam x (subst y v e') and xy: x ≠ y
    and IH: ∀ v1 v2. v2 ∈ E (subst y v e') ((x,v1)#ϱ)
        ⟶ (∃ ϱ' v'. v' ∈ E v [] ∧ v2 ∈ E e' ϱ' ∧ ϱ' ≈ (y,v')#(x,v1)#ϱ)
  shows ∃ ϱ' v''. v'' ∈ E v [] ∧ VFun f ∈ E (ELam x e') ϱ' ∧ ϱ' ≈ ((y,v'')#ϱ)
  using fl vv ls IH xy
proof (induction f arbitrary: x e e' ϱ v y)
  case empty
  from empty(2) is-val-def obtain v' where vp-v: v' ∈ E v [] using e-val[of v []] by blast
  let ?R = (y,v')#ϱ
  have 1: VFun {||} ∈ E (ELam x e') ?R by simp
  have 2: ?R ≈ (y, v') # ϱ by auto
  from vp-v 1 2 show ?case by blast
next
  case (insert a f x e e' ϱ v y)
  from insert(3) have 1: VFun f ∈ E (ELam x e) ϱ by auto
  obtain v1 v2 where a: a = (v1,v2) by (cases a) simp
  from insert 1 have ∃ ϱ' v''. v'' ∈ E v [] ∧ VFun f ∈ E (ELam x e') ϱ' ∧ ϱ' ≈ ((y,v'')#ϱ)
    by metis
  from this obtain ϱ'' v'' where vpp-v: v'' ∈ E v [] and f-l: VFun f ∈ E (ELam x e') ϱ''
    and rpp-r: ϱ'' ≈ ((y,v'')#ϱ) by blast
  from insert(3) a have v2-e: v2 ∈ E e ((x,v1)#ϱ) using e-lam-elim2 by blast
  from insert v2-e have ∃ϱ'' v'. v' ∈ E v [] ∧ v2 ∈ E e' ϱ'' ∧ ϱ'' ≈ (y, v')#(x, v1)#ϱ by auto
  from this obtain ϱ3 v' where vp-v: v' ∈ E v [] and v2-ep: v2 ∈ E e' ϱ3
    and r3: ϱ3 ≈ (y,v') # (x,v1) # ϱ by blast
  from insert(4) have isval v by auto
  from this vp-v vpp-v obtain v3 where v3-v: v3 ∈ E v [] and vp-vpp: v' ⊔ v'' = Some v3
    using combine-values by blast
  have 4: VFun (finsert a f) ∈ E (ELam x e') ((y, v3) # ϱ)
  proof −
    from vp-vpp have v3-vpp: v'' ⊑ v3 using le-union2 by simp
    from rpp-r v3-vpp have ϱ'' ⊑ (y,v3)#ϱ by (simp add: env-eq-def env-le-def)
    from f-l this have 2: VFun f ∈ E (ELam x e') ((y, v3) # ϱ) by (rule raise-env)
    from vp-vpp have vp-v3: v' ⊑ v3 using le-union1 by simp
    from vp-v3 r3 insert have ϱ3 ⊑ (x,v1)#(y,v3)#ϱ by (simp add: env-eq-def env-le-def)
    from v2-ep this have 3: v2 ∈ E e' ((x,v1)#(y,v3)#ϱ) by (rule raise-env)
    from 2 3 a show ?thesis using e-lam-intro2 by blast
  qed
  have 5: (y, v3) # ϱ ≈ (y, v3) # ϱ by auto
  from v3-v 4 5 show ?case by blast
```

qed

lemma *lookup-ext-none*: ⟦ *lookup* ϱ *y* = *None*; *x* ≠ *y* ⟧ ⟹ *lookup* ((*x*,*v*)#ϱ) *y* = *None*
by *auto*

— For reverse subst lemma, the variable case shows up over and over, so we prove it as a lemma
lemma *rev-subst-var*:
assumes *ev*: *e* = *EVar y* ∧ *v* = *e*′ **and** *vv*: *is-val v* **and** *vp-E*: *v*′ ∈ *E e*′ ϱ
shows ∃ ϱ′ *v*′′. *v*′′ ∈ *E v* [] ∧ *v*′ ∈ *E e* ϱ′ ∧ ϱ′ ≈ ((*y*,*v*′′)#ϱ)
proof −
from *vv* **have** *lx*: ∀ *x*. *x* ∈ *FV v* ⟶ *lookup* [] *x* = *lookup* ϱ *x* **by** *auto*
from *ev vp-E lx env-strengthen*[*of v*′ *v* ϱ []] **have** *n-Ev*: *v*′ ∈ *E v* [] **by** *blast*
have *ly*: *lookup* ((*y*,*v*′)#ϱ) *y* = *Some v*′ **by** *simp*
from *env-eq-def* **have** *rr*: ((*y*,*v*′)#ϱ) ≈ ((*y*,*v*′)#ϱ) **by** *simp*
from *ev ly* **have** *n-Ee*: *v*′ ∈ *E e* ((*y*,*v*′)#ϱ) **by** *simp*
from *n-Ev rr n-Ee* **show** *?thesis* **by** *blast*
qed

lemma *reverse-subst-pres-denot*:
assumes *vep*: *v*′ ∈ *E e*′ ϱ **and** *vv*: *is-val v* **and** *ep*: *e*′ = *subst y v e*
shows ∃ ϱ′ *v*′′. *v*′′ ∈ *E v* [] ∧ *v*′ ∈ *E e* ϱ′ ∧ ϱ′ ≈ ((*y*,*v*′′)#ϱ)
using *vep vv ep*
proof (*induction arbitrary*: *v*′ *y v e rule*: *E.induct*)
case (*1 n* ϱ) — e' = ENat n
from *1*(*1*) **have** *vp*: *v*′ = *VNat n* **by** *auto*
from *1*(*3*) **have** *e* = *ENat n* ∨ (*e* = *EVar y* ∧ *v* = *ENat n*) **by** (*cases e*, *auto*)
then show *?case*
proof
assume *e*: *e* = *ENat n*
from *1*(*2*) *e-val is-val-def* **obtain** *v*′′ **where** *vpp-E*: *v*′′ ∈ *E v* [] **by** *force*
from *env-eq-def* **have** *rr*: ((*y*,*v*′′)#ϱ) ≈ ((*y*,*v*′′)#ϱ) **by** *simp*
from *vp e* **have** *vp-E*: *v*′ ∈ *E e* ((*y*,*v*′′)#ϱ) **by** *simp*
from *vpp-E vp-E rr* **show** *?thesis* **by** *blast*
next
assume *ev*: *e* = *EVar y* ∧ *v* = *ENat n*
from *ev 1*(*2*) *1*(*1*) *rev-subst-var* **show** *?thesis* **by** *blast*
qed
next
case (*2 x* ϱ) — e' = EVar x
from *2* **have** *e*: *e* = *EVar x* **by** (*cases e*, *auto*)
from *2 e* **have** *xy*: *x* ≠ *y* **by** *force*
from *2*(*2*) *e-val is-val-def* **obtain** *v*′′ **where** *vpp-E*: *v*′′ ∈ *E v* [] **by** *force*
from *env-eq-def* **have** *rr*: ((*y*,*v*′′)#ϱ) ≈ ((*y*,*v*′′)#ϱ) **by** *simp*
from *2*(*1*) **obtain** *vx* **where** *lx*: *lookup* ϱ *x* = *Some vx* **and** *vp-vx*: *v*′ ⊑ *vx* **by** *auto*
from *e lx vp-vx xy* **have** *vp-E*: *v*′ ∈ *E e* ((*y*,*v*′′)#ϱ) **by** *simp*
from *vpp-E rr vp-E* **show** *?case* **by** *blast*
next
case (*3 x eb* ϱ)
{ assume *ev*: *e* = *EVar y* ∧ *v* = *ELam x eb*
from *ev 3*(*3*) *3*(*2*) *rev-subst-var* **have** *?case* **by** *metis*
} also { assume *ex*: *e* = *ELam x eb* ∧ *x* = *y*
from *3*(*3*) *e-val is-val-def* **obtain** *v*′′ **where** *vpp-E*: *v*′′ ∈ *E v* [] **by** *force*
from *env-eq-def* **have** *rr*: ((*y*,*v*′′)#ϱ) ≈ ((*y*,*v*′′)#ϱ) **by** *simp*
from *ex* **have** *lz*: ∀ *z*. *z* ∈ *FV* (*ELam x eb*) ⟶ *lookup* ((*y*,*v*′′)#ϱ) *z* = *lookup* ϱ *z* **by** *auto*
from *ex 3*(*2*) *lz env-strengthen*[*of v*′ *ELam x eb* ϱ (*y*,*v*′′)#ϱ]
have *vp-E*: *v*′ ∈ *E e* ((*y*,*v*′′)#ϱ) **by** *blast*
from *vpp-E vp-E rr* **have** *?case* **by** *blast*
} moreover { assume *exb*: ∃ *e*′. *e* = *ELam x e*′ ∧ *x* ≠ *y* ∧ *eb* = *subst y v e*′
from *exb* **obtain** *e*′′ **where** *e*: *e* = *ELam x e*′′ **and** *xy*: *x* ≠ *y*
and *eb*: *eb* = *subst y v e*′′ **by** *blast*
from *3*(*2*) **obtain** *f* **where** *vp*: *v*′ = *VFun f* **by** *auto*

```
    from 3(2) vp have f-E: VFun f ∈ E (ELam x eb) ϱ by simp
    from 3(4) e xy have ls: ELam x eb = ELam x (subst y v e'') by simp
    from 3(3) eb have IH: ∀ v1 v2. v2 ∈ E (subst y v e'') ((x,v1)#ϱ)
        ⟶ (∃ ϱ' v'. v' ∈ E v [] ∧ v2 ∈ E e'' ϱ' ∧ ϱ' ≈ (y,v')#(x,v1)#ϱ)
      apply clarify apply (subgoal-tac (v1,v2) ∈ fset {|(v1,v2)|}) prefer 2 apply simp
      apply (rule 3(1)) apply assumption apply simp+ done
    from f-E 3(3) ls xy IH e vp have ?case apply clarify apply (rule reverse-subst-lam)
       apply blast+ done
  } moreover from 3(4) have (e = EVar y ∧ v = ELam x eb)
      ∨ (e = ELam x eb ∧ x = y)
      ∨ (∃ e'. e = ELam x e' ∧ x ≠ y ∧ eb = subst y v e') by (cases e) auto
  ultimately show ?case by blast
next
  case (4 e1 e2 ϱ) — e' = EApp e1 e2
  from 4(4) 4(5) obtain e1' e2' where
    e: e = EApp e1' e2' and e1:e1 = subst y v e1' and e2: e2 = subst y v e2'
    apply (cases e) apply (rename-tac x) apply auto apply (case-tac y = x) apply auto
    apply (rename-tac x1 x2) apply (case-tac y = x1) apply auto done
  from 4(3) obtain f v2 and v2'::val and v3' where
    f-E: VFun f ∈ E e1 ϱ and v2-E: v2 ∈ E e2 ϱ and v23: (v2',v3') ∈ fset f
    and v2p-v2: v2' ⊑ v2 and vp-v3: v' ⊑ v3' by blast
  from 4(1) f-E 4(4) e1 obtain ϱ1 w1 where v1-Ev: w1 ∈ E v [] and f-E1: VFun f ∈ E e1' ϱ1
    and r1: ϱ1 ≈ (y,w1)#ϱ by blast
  from 4(2) v2-E 4(4) e2 obtain ϱ2 w2 where v2-Ev: w2 ∈ E v [] and v2-E2: v2 ∈ E e2' ϱ2
    and r2: ϱ2 ≈ (y,w2)#ϱ by blast
  from 4(4) v1-Ev v2-Ev combine-values obtain w3 where
    w3-Ev: w3 ∈ E v [] and w123: w1 ⊔ w2 = Some w3 by (simp only: is-val-def) blast
  from w123 le-union1 have w13: w1 ⊑ w3 by blast
  from w123 le-union2 have w23: w2 ⊑ w3 by blast
  from w13 have r13: ((y,w1)#ϱ) ⊑ ((y,w3)#ϱ) by (simp add: env-le-def)
  from w23 have r23: ((y,w2)#ϱ) ⊑ ((y,w3)#ϱ) by (simp add: env-le-def)
  from r1 f-E1 have f-E1b: VFun f ∈ E e1' ((y,w1)#ϱ) by (rule env-swap)
  from f-E1b r13 have f-E1c: VFun f ∈ E e1' ((y,w3)#ϱ) by (rule raise-env)
  from r2 v2-E2 have v2-E2b: v2 ∈ E e2' ((y,w2)#ϱ) by (rule env-swap)
  from v2-E2b r23 have v2-E2c: v2 ∈ E e2' ((y,w3)#ϱ) by (rule raise-env)
  from f-E1c v2-E2c v23 v2p-v2 vp-v3 have vp-E2: v' ∈ E (EApp e1' e2') ((y,w3)#ϱ) by blast
  have rr3: ((y,w3)#ϱ) ≈ ((y,w3)#ϱ) by (simp add: env-eq-def)
  from w3-Ev vp-E2 rr3 e show ?case by blast
next
  case (5 f e1 e2 ϱ) — e' = EPrim f e1 e2, very similar to case for EApp
  from 5(4) 5(5) obtain e1' e2' where
    e: e = EPrim f e1' e2' and e1:e1 = subst y v e1' and e2: e2 = subst y v e2'
    apply (cases e) apply auto apply (rename-tac x) apply (case-tac y = x) apply auto
    apply (rename-tac x1 x2) apply (case-tac y = x1) apply auto done
  from 5(3) obtain n1 n2 where
    n1-E: VNat n1 ∈ E e1 ϱ and n2-E: VNat n2 ∈ E e2 ϱ and vp: v' = VNat (f n1 n2) by blast
  from 5(1) n1-E 5(4) e1 obtain ϱ1 w1 where v1-Ev: w1 ∈ E v [] and n1-E1: VNat n1 ∈ E e1' ϱ1
    and r1: ϱ1 ≈ (y,w1)#ϱ by blast
  from 5(2) n2-E 5(4) e2 obtain ϱ2 w2 where v2-Ev: w2 ∈ E v [] and n2-E2: VNat n2 ∈ E e2' ϱ2
    and r2: ϱ2 ≈ (y,w2)#ϱ by blast
  from 5(4) v1-Ev v2-Ev combine-values obtain w3 where
    w3-Ev: w3 ∈ E v [] and w123: w1 ⊔ w2 = Some w3 by (simp only: is-val-def) blast
  from w123 le-union1 have w13: w1 ⊑ w3 by blast
  from w123 le-union2 have w23: w2 ⊑ w3 by blast
  from w13 have r13: ((y,w1)#ϱ) ⊑ ((y,w3)#ϱ) by (simp add: env-le-def)
  from w23 have r23: ((y,w2)#ϱ) ⊑ ((y,w3)#ϱ) by (simp add: env-le-def)
  from r1 n1-E1 have n1-E1b: VNat n1 ∈ E e1' ((y,w1)#ϱ) by (rule env-swap)
  from n1-E1b r13 have n1-E1c: VNat n1 ∈ E e1' ((y,w3)#ϱ) by (rule raise-env)
  from r2 n2-E2 have n2-E2b: VNat n2 ∈ E e2' ((y,w2)#ϱ) by (rule env-swap)
  from n2-E2b r23 have v2-E2c: VNat n2 ∈ E e2' ((y,w3)#ϱ) by (rule raise-env)
  from n1-E1c v2-E2c vp have vp-E2: v' ∈ E (EPrim f e1' e2') ((y,w3)#ϱ) by blast
```

```
  have rr3: ((y,w3)#ϱ) ≈ ((y,w3)#ϱ) by (simp add: env-eq-def)
  from w3-Ev vp-E2 rr3 e show ?case by blast
next
  case (6 e1 e2 e3 ϱ) — e' = EIf e1 e2 e3
  from 6(5) 6(6) obtain e1' e2' e3' where
    e: e = EIf e1' e2' e3' and e1:e1 = subst y v e1' and e2: e2 = subst y v e2'
    and e3: e3 = subst y v e3'
    apply (cases e) apply auto apply (case-tac y=x1) apply auto apply (case-tac y=x31) by auto
  from 6(4) e-if-elim obtain n where n-E: VNat n ∈ E e1 ϱ and
    els: n = 0 ⟶ v' ∈ E e3 ϱ and thn: n ≠ 0 ⟶ v' ∈ E e2 ϱ by blast
  from 6 n-E e1 obtain ϱ1 w1 where w1-Ev: w1 ∈ E v [] and n-E2: VNat n ∈ E e1' ϱ1
    and r1: ϱ1 ≈ (y,w1)#ϱ by blast
  show ?case
  proof (cases n = 0)
    case True with els have vp-E2: v' ∈ E e3 ϱ by simp
    from 6 vp-E2 e3 obtain ϱ2 w2 where w2-Ev: w2 ∈ E v [] and vp-E2: v' ∈ E e3' ϱ2
    and r2: ϱ2 ≈ (y,w2)#ϱ by blast
    from 6(5) w1-Ev w2-Ev combine-values obtain w3 where
      w3-Ev: w3 ∈ E v [] and w123: w1 ⊔ w2 = Some w3 by (simp only: is-val-def) blast
    from w123 le-union1 have w13: w1 ⊑ w3 by blast
    from w123 le-union2 have w23: w2 ⊑ w3 by blast
    from w13 have r13: ((y,w1)#ϱ) ⊑ ((y,w3)#ϱ) by (simp add: env-le-def)
    from w23 have r23: ((y,w2)#ϱ) ⊑ ((y,w3)#ϱ) by (simp add: env-le-def)
    from r1 n-E2 have n-E1b: VNat n ∈ E e1' ((y,w1)#ϱ) by (rule env-swap)
    from n-E1b r13 have n-E1c: VNat n ∈ E e1' ((y,w3)#ϱ) by (rule raise-env)
    from r2 vp-E2 have vp-E2b: v' ∈ E e3' ((y,w2)#ϱ) by (rule env-swap)
    from vp-E2b r23 have vp-E2c: v' ∈ E e3' ((y,w3)#ϱ) by (rule raise-env)
    have rr3: ((y,w3)#ϱ) ≈ ((y,w3)#ϱ) by (simp add: env-eq-def)
    from True n-E1c vp-E2c e have vp-E3: v' ∈ E e ((y,w3)#ϱ) by auto
    from w3-Ev rr3 vp-E3 show ?thesis by blast
  next
    case False with thn have vp-E2: v' ∈ E e2 ϱ by simp
    from 6 vp-E2 e2 obtain ϱ2 w2 where w2-Ev: w2 ∈ E v [] and vp-E2: v' ∈ E e2' ϱ2
    and r2: ϱ2 ≈ (y,w2)#ϱ by blast
    from 6(5) w1-Ev w2-Ev combine-values obtain w3 where
      w3-Ev: w3 ∈ E v [] and w123: w1 ⊔ w2 = Some w3 by (simp only: is-val-def) blast
    from w123 le-union1 have w13: w1 ⊑ w3 by blast
    from w123 le-union2 have w23: w2 ⊑ w3 by blast
    from w13 have r13: ((y,w1)#ϱ) ⊑ ((y,w3)#ϱ) by (simp add: env-le-def)
    from w23 have r23: ((y,w2)#ϱ) ⊑ ((y,w3)#ϱ) by (simp add: env-le-def)
    from r1 n-E2 have n-E1b: VNat n ∈ E e1' ((y,w1)#ϱ) by (rule env-swap)
    from n-E1b r13 have n-E1c: VNat n ∈ E e1' ((y,w3)#ϱ) by (rule raise-env)
    from r2 vp-E2 have vp-E2b: v' ∈ E e2' ((y,w2)#ϱ) by (rule env-swap)
    from vp-E2b r23 have vp-E2c: v' ∈ E e2' ((y,w3)#ϱ) by (rule raise-env)
    have rr3: ((y,w3)#ϱ) ≈ ((y,w3)#ϱ) by (simp add: env-eq-def)
    from False n-E1c vp-E2c e have vp-E3: v' ∈ E e ((y,w3)#ϱ) by auto
    from w3-Ev rr3 vp-E3 show ?thesis by blast
  qed
qed
```

13.2 Reverse reduction preserves denotation

```
lemma reverse-step-pres-denot:
  fixes e::exp assumes e-ep: e ⟶ e' and v-ep: v ∈ E e' ϱ
  shows v ∈ E e ϱ
  using e-ep v-ep
proof (induction arbitrary: v ϱ rule: reduce.induct)
  case (beta v x e v' ϱ)
  from beta obtain ϱ' v'' where 1: v'' ∈ E v [] and 2: v' ∈ E e ϱ' and 3: ϱ' ≈ (x, v'') # ϱ
    using reverse-subst-pres-denot[of v' subst x v e ϱ v x e] by blast
  from beta 1 2 3 show ?case
```

apply *simp* **apply** (*rule-tac x*={|(*v''*,*v'*)|} **in** *exI*) **apply** (*rule conjI*)
apply *clarify* **apply** *simp* **apply** (*rule env-swap*) **apply** *blast* **apply** *blast*
apply (*rule-tac x*=*v''* **in** *exI*) **apply** (*rule conjI*) **apply** (*rule env-strengthen*)
apply *assumption* **apply** *force* **apply** *force* **done**
qed *auto*

lemma *reverse-multi-step-pres-denot*:
fixes *e*::*exp* **assumes** *e-ep*: $e \longrightarrow* e'$ **and** *v-ep*: $v \in E\ e'\ \varrho$ **shows** $v \in E\ e\ \varrho$
using *e-ep v-ep reverse-step-pres-denot*
by (*induction arbitrary*: *v* ϱ *rule*: *multi-step.induct*) *auto*

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 −
from *vv* **have** $\exists\ v'.\ v' \in E\ v\ []$ **using** *e-val* **by** *auto*
from *this* **obtain** v' **where** *vp-v*: $v' \in E\ v\ []$ **by** *blast*
from *vp-v vv* **have** *vp-v2*: $v' \in E\ v\ \varrho$ **using** *env-strengthen* **by** *force*
from *ev vp-v2 reverse-multi-step-pres-denot*[*of e v v'* ϱ]
have $v' \in E\ e\ \varrho$ **by** *simp*
from *this vp-v* **show** *?thesis* **by** *blast*
qed

theorem *reduce-pres-denot*: **fixes** *e*::*exp* **assumes** *r*: $e \longrightarrow e'$ **shows** $E\ e = E\ e'$
apply (*rule ext*) **apply** (*rule equalityI*) **apply** (*rule subsetI*)
apply (*rule subject-reduction*) **apply** *assumption* **using** *r* **apply** *assumption*
apply (*rule subsetI*)
using *r* **apply** (*rule reverse-step-pres-denot*) **apply** *assumption*
done

theorem *multi-reduce-pres-denot*: **fixes** *e*::*exp* **assumes** *r*: $e \longrightarrow* e'$ **shows** $E\ e = E\ e'$
using *r reduce-pres-denot* **by** *induction auto*

theorem *complete-wrt-op-sem*:
assumes *e-n*: $e \Downarrow ONat\ n$ **shows** $E\ e\ [] = E\ (ENat\ n)\ []$
proof −
from *e-n* **have** *1*: $e \longrightarrow* ENat\ n$
unfolding *run-def* **apply** *simp* **apply** (*erule exE*)
apply (*rename-tac v*) **apply** (*case-tac v*) **apply** *auto* **done**
from *1* **show** *?thesis* **using** *multi-reduce-pres-denot* **by** *simp*
qed

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*]: $E\ e\ = E\ e' \Longrightarrow E\ (ELam\ x\ e) = E\ (ELam\ x\ e')$
by (*rule ext*) *simp*

lemma *e-app-cong*[*cong*]: $[\![\ E\ e1 = E\ e1';\ E\ e2 = E\ e2'\ ]\!] \Longrightarrow E\ (EApp\ e1\ e2) = E\ (EApp\ e1'\ e2')$
by (*rule ext*) *simp*

lemma *e-prim-cong*[*cong*]: ⟦ *E e1 = E e1′*; *E e2 = E e2′* ⟧ ⟹ *E*(*EPrim f e1 e2*) = *E*(*EPrim f e1′ e2′*)
by (*rule ext*) *simp*

lemma *e-if-cong*[*cong*]: ⟦ *E e1 = E e1′*; *E e2 = E e2′*; *E e3 = E e3′* ⟧
⟹ *E* (*EIf e1 e2 e3*) = *E* (*EIf e1′ e2′ e3′*)
by (*rule ext*) *simp*

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′* ⟹ *E* (*plug C e*) = *E* (*plug C e′*)
proof (*induction C arbitrary*: *e e′*)
case (*CIf1 C e2 e3*)
have *E* (*EIf* (*plug C e*) *e2 e3*) = *E* (*EIf* (*plug C e′*) *e2 e3*)
apply (*rule e-if-cong*) **using** *CIf1* **apply** *blast*+ **done**
then show *?case* **by** *simp*
next
case (*CIf2 e1 C e3*)
have *E* (*EIf e1* (*plug C e*) *e3*) = *E* (*EIf e1* (*plug C e′*) *e3*)
apply (*rule e-if-cong*) **using** *CIf2* **apply** *blast*+ **done**
then show *?case* **by** *simp*
next
case (*CIf3 e1 e2 C*)
have *E* (*EIf e1 e2* (*plug C e*)) = *E* (*EIf e1 e2* (*plug C e′*))
apply (*rule e-if-cong*) **using** *CIf3* **apply** *blast*+ **done**
then show *?case* **by** *simp*
qed *force*+

14.2 Auxiliary lemmas

lemma *diverge-denot-empty*: **assumes** *d*: *diverge e* **and** *fve*: *FV e* = {} **shows** *E e* [] = {}
proof (*rule classical*)
assume *E e* [] ≠ {}
from *this* **obtain** *A* **where** *wte*: *A* ∈ *E e* [] **by** *auto*
have *ge*: *good-env* [] [] **by** *blast*
from *wte ge* **obtain** *v* **where** *e-v*: [] ⊢ *e* ⇓ *v* **and** *gv*: *v* ∈ *good A*
using *denot-terminates* **by** *blast*
from *e-v fve* **obtain** *v′* **where** *e-vp*: *e* ⟶∗ *v′* **and** *val-vp*: *isval v′*
using *sound-wrt-small-step* **by** *blast*
from *d e-vp* **have** ∃ *e′*. *v′* ⟶ *e′* **by** (*simp add*: *diverge-def*)
with *val-vp* **have** *False* **using** *val-stuck* **by** *force*
from *this* **show** *?thesis* **..**
qed

lemma *goes-wrong-denot-empty*:
assumes *gw*: *goes-wrong e* **and** *fv-e*: *FV e* = {} **shows** *E e* [] = {}
proof (*rule classical*)
assume *E e* [] ≠ {}
from *this* **obtain** *A* **where** *wte*: *A* ∈ *E e* [] **by** *auto*

have *ge*: *good-env* [] [] **by** *blast*
from *gw* **obtain** e' **where** *e-ep*: $e \longrightarrow* e'$ **and** *s-ep*: *stuck* e' **and** *nv-ep*: $\neg$ *isval* e'
by *auto*
from *wte e-ep* **have** *wtep*: $A \in E\ e'$ [] **using** *preservation* **by** *blast*
from *fv-e e-ep* **have** *fv-ep*: $FV\ e' = \{\}$ **using** *reduction-pres-fv* **by** *auto*
from *wtep fv-ep* **have** *is-val* $e' \vee (\exists\ e''.\ e' \longrightarrow e'')$ **using** *progress*[*of A* e' []] **by** *simp*
from *this s-ep nv-ep* **have** *False* **by** *simp*
from *this* **show** *?thesis* **..**
qed

lemma *denot-empty-diverge*: **assumes** *E-e*: $E\ e$ [] $= \{\}$ **and** *fv-e*: $FV\ e = \{\}$
shows *diverge* $e \vee$ *goes-wrong* e
proof (*rule classical*)
assume *nd-gw*: $\neg$ (*diverge* $e \vee$ *goes-wrong* e)
from *this* **have** *nd*: $\neg$ *diverge* e **by** *blast*
from *nd-gw* **have** *gw*: $\neg$ *goes-wrong* e **by** *blast*
from *nd* **obtain** *v*::*exp* **where** *e-v*: $e \longrightarrow* v$ **and** *stuck*: $\neg\ (\exists\ e'.\ v \longrightarrow e')$
by (*simp only*: *diverge-def*) *blast*
from *gw e-v stuck* **have** *val-v*: *isval v* **by** (*simp only*: *goes-wrong-def stuck-def*) *blast*
from *fv-e e-v* **have** *fv-v*: $FV\ v = \{\}$ **using** *reduction-pres-fv* **by** *auto*
from *val-v fv-v* **have** *val-v2*: *is-val v* **by** *simp*
from *e-v val-v2* **obtain** *A* **where** *wte*: $A \in E\ e$ [] **and** *wtv*: $A \in E\ v$ []
using *completeness*[*of e v*] **by** *blast*
from *this E-e* **have** *False* **by** *auto*
from *this* **show** *?thesis* **..**
qed

lemma *val-ty-observe*:
⟦ $A \in E\ v$ []; $A \in E\ v'$ [];
observe v ob; *isval v'*; *isval v* ⟧ $\Longrightarrow$ *observe* v' *ob*
apply (*cases v*) **apply** *auto* **apply** (*cases* v') **apply** *auto*
apply (*cases* v') **apply** *auto*
apply (*cases ob*) **apply** *auto*
done

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
assume *run-Ce1*: *run* (*plug C e1*) *ob*
from *e12* **have** *pe12*: *E* (*plug C e1*) = *E* (*plug C e2*) **by** (*rule congruence*)
from *run-Ce1* **have** ((∃ *v*. (*plug C e1*) $\longrightarrow*$ *v* ∧ *observe v ob*)
∨ ((*diverge* (*plug C e1*) ∨ *goes-wrong* (*plug C e1*)) ∧ *ob* = *OBad*))
by (*simp only*: *run-def*)
from *this* **show** *run* (*plug C e2*) *ob*
proof
assume ∃ *v*. *plug C e1* $\longrightarrow*$ *v* ∧ *observe v ob*
from *this* **obtain** *v* **where** *r-v*: *plug C e1* $\longrightarrow*$ *v*
and *ob-v*: *observe v ob* **by** *blast*
from *r-v fv-e1* **have** *fv-v*: $FV\ v = \{\}$ **by** (*rule reduction-pres-fv*)
from *ob-v fv-v* **have** *val-v*: *is-val v* **by** (*cases v*) *auto*
from *r-v val-v* **obtain** *A* **where** *ce1a*: $A \in E\ (plug\ C\ e1)$ []
and *wt-v-ap*: $A \in E\ v$ [] **using** *completeness*[*of plug C e1 v*] **by** *auto*
from *ce1a pe12* **have** *ce2a*: $A \in E\ (plug\ C\ e2)$ [] **by** *force*
have *ge*: *good-env* [] [] **by** *blast*
from *ce2a ge* **obtain** v' **where** *Ce2-vp*: [] ⊢ *plug C e2* ⇓ v' **and** *vpa*: $v' \in good\ A$
using *denot-terminates* **by** *blast*
from *Ce2-vp fv-e2* **obtain** v'' ob' **where** *Ce2-vpp*: *plug C e2* $\longrightarrow*$ v'' **and** *vvpp*: *isval* v''

```
    and ovpp: observe v'' ob' and vp-ob: bs-observe v' ob'
    using sound-wrt-small-step[of plug C e2 v'] by blast
  from ovpp have vpp-ob: observe v'' ob
  proof −
    from ce2a Ce2-vpp have vpp-app: A ∈ E v'' [] using preservation by blast
    from vpp-app wt-v-ap ob-v vvpp val-v
    show ?thesis apply simp apply (rule val-ty-observe) prefer 3 apply assumption apply auto done
  qed
  from Ce2-vpp vpp-ob show ?thesis by (simp add: run-def) blast
 next
  assume d-e1: (diverge (plug C e1) ∨ goes-wrong (plug C e1)) ∧ ob = OBad
  from d-e1 fv-e1 have E-Ce1: E (plug C e1) [] = {}
    using diverge-denot-empty goes-wrong-denot-empty by blast
  from E-Ce1 pe12 have E-Ce2: E (plug C e2) [] = {} by simp
  from E-Ce2 fv-e2 have diverge (plug C e2) ∨ goes-wrong (plug C e2)
    using denot-empty-diverge by blast
  from this d-e1 show ?thesis  by (simp add: run-def)
 qed
qed

definition ctx-equiv :: exp ⇒ exp ⇒ bool (infix ≃ 51) where
e ≃ e' ≡ ∀ C ob. FV (plug C e) = {} ∧ FV (plug C e') = {} ⟶
   run (plug C e) ob = run (plug C e') ob

theorem denot-sound-wrt-ctx-equiv: assumes e12: E e1 = E e2 shows e1 ≃ e2
 using e12
 apply (simp only: ctx-equiv-def) apply clarify apply (rule iffI)
  apply (rule soundness-wrt-ctx-equiv-aux) apply assumption+
 apply (rule soundness-wrt-ctx-equiv-aux) apply auto
 done

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))
   using e1 e2 by auto

theorem eval-ifz[simp]: assumes e1: E e1  = E(ENat 0) shows E(EIf e1 e2 e3) = E(e3)
 using e1 by auto

theorem eval-ifnz[simp]: assumes e1: E(e1) = E(ENat n) and nz: n ≠ 0
 shows E(EIf e1 e2 e3) = E(e2)
 using e1 nz by auto

theorem eval-app-lam: assumes vv: is-val v
 shows E(EApp (ELam x e) v) = E (subst x v e)
 apply (rule ext) apply (rule equalityI) apply (rule subsetI)
  apply (subgoal-tac EApp (ELam x e) v ⟶ subst x v e) prefer 2 apply (rule beta)
 using vv apply assumption apply (rule subject-reduction) apply assumption apply assumption
 apply (rule subsetI)
 apply (subgoal-tac EApp (ELam x e) v ⟶ subst x v e) prefer 2 apply (rule beta)
 using vv apply assumption apply (rule reverse-step-pres-denot) apply assumption
 apply assumption
 done
```

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* = {}
by (*case-tac e*) *auto*

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 (*induction e k rule*: *opt.induct*)
case (*5 e1 e2 k*)
then show *?case*
apply (*cases opt e1* (*Suc k*))
apply *force*
apply *force*
prefer *2* **apply** *force*
prefer *2* **apply** *force*
prefer *2* **apply** *force*
apply (*rename-tac x e*)
apply (*cases is-value* (*opt e2* (*Suc k*)))
apply (*subgoal-tac E* (*EApp* (*ELam x e*) (*opt e2* (*Suc k*)))
= *E* (*subst x* (*opt e2* (*Suc k*)) *e*))
prefer *2* **apply** (*rule eval-app-lam*)
apply (*simp del*: *E.simps*)
apply (*subgoal-tac E*(*EApp e1 e2*) = *E*(*EApp* (*ELam x e*) (*opt e2* (*Suc k*))))
prefer *2*
apply (*rule e-app-cong*)
apply *force*+ **done**
next
case (*6 f e1 e2 k*)

then show *?case* **apply** *auto* **apply** (*cases opt e1 k*) **apply** *auto*
apply (*cases opt e2 k*) **apply** *auto* **done**
next
case (*7 e1 e2 e3 k*)
then show *?case* **by** (*cases opt e1 k*) *auto*
qed *auto*

theorem *opt-correct*: $e \simeq opt\ e\ k$
using *opt-correct-aux denot-sound-wrt-ctx-equiv* **by** *blast*

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* × *val*) *fset* | *Abs val option* | *Wrong*

fun *val-le* :: *val* ⇒ *val* ⇒ *bool* (**infix** $\sqsubseteq$ *52*) **where**
(*VNat n*) $\sqsubseteq$ (*VNat n′*) = (*n* = *n′*) |
(*Fun f*) $\sqsubseteq$ (*Fun f′*) = (*fset f* ⊆ *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* ⇒ (*′a* ⇒ *′b set*) ⇒ *′b set* **where**
set-bind m f ≡ { *v*. ∃ *v′*. *v′* ∈ *m* ∧ *v* ∈ *f v′* }
declare *set-bind-def*[*simp*]

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

definition *errset-bind* :: *val set* ⇒ (*val* ⇒ *val set*) ⇒ *val set* **where**
errset-bind m f ≡ { *v*. ∃ *v′*. *v′* ∈ *m* ∧ *v′* ≠ *Wrong* ∧ *v* ∈ *f v′* } ∪ {*v*. *v* = *Wrong* ∧ *Wrong* ∈ *m* }
declare *errset-bind-def*[*simp*]

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

definition *return* :: *val* ⇒ *val set* **where**
return v ≡ {*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* ⇒ *val set*) ⇒ *env* ⇒ *val* ⇒ *bool* **where**
iterate-none[*intro*!]: *iterate Ee ϱ* (*Fun* {||}) |
iterate-again[*intro*!]: ⟦ *iterate Ee ϱ f*; *f′* ∈ *Ee* (*f*#*ϱ*) ⟧ ⟹ *iterate Ee ϱ f′*

abbreviation *apply-fun* :: *val set* ⇒ *val set* ⇒ *val set* **where**
apply-fun V1 V2 ≡ (*v1* := *V1*; *v2* := *V2*;
case v1 of Fun f ⇒
(*v2′*,*v3′*) ← *fset f*;
if v2′ ⊑ *v2 then return v3′ else* {}
| - ⇒ *return Wrong*)

fun *E* :: *exp* ⇒ *env* ⇒ *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*. ∃ *f*. *v* = *Fun f* ∧ (∀ *v1 v2′*. (*v1*,*v2′*) ∈ *fset f* ⟶
(∃ *v2*. *v2* ∈ *E e* (*v1*#*ϱ*) ∧ *v2′* ⊑ *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*. (∃ *v′*. *v* = *Abs* (*Some v′*) ∧ *v′* ∈ *E e ϱ*)
∨ (*v* = *Abs None* ∧ *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 (*induction rule*: *wfenv.induct*)
case *wfnil*
then show *?case* **by** (*force simp*: *good-ctx-def*)
next
case (*wfvbind* ϱ η Γ *v* τ)
then show *?case*
apply (*simp add*: *good-ctx-def push-ty-def*) **apply** (*cases* Γ) **apply** *simp*
apply *clarify* **apply** (*rename-tac n*) **apply** (*case-tac n*) **apply** *force* **apply** *force* **done**
next
case (*wftbind* ϱ η Γ *V*)
then show *?case*
apply (*simp add*: *good-ctx-def push-tyvar-def*) **apply** (*cases* Γ) **apply** *simp*
apply *clarify* **apply** (*rename-tac n*) **apply** (*case-tac n*) **apply** *auto* **done**
qed

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 (*induction* ϱ*1 arbitrary*: ϱ*2 n*)
case *Nil*
then show *?case* **by** *auto*
next

```
  case (Cons a ϱ1)
  then show ?case by (cases n) auto
qed

lemma nth-append2[simp]: n ≥ length ϱ1 ⟹ (ϱ1@ϱ2)!n = ϱ2!(n − length ϱ1)
proof (induction ϱ1 arbitrary: ϱ2 n)
  case Nil
  then show ?case by auto
next
  case (Cons a ϱ1)
  then show ?case by (cases n) auto
qed

lemma shift-append-preserves-T-aux:
  shows T τ (ϱ1@ϱ3) = T (shift (length ϱ2) (length ϱ1) τ) (ϱ1@ϱ2@ϱ3)
proof (induction τ arbitrary: ϱ1 ϱ2 ϱ3)
  case (Forall τ)
  then show ?case
    apply simp
    apply (rule equalityI) apply (rule subsetI) apply (simp only: mem-Collect-eq)
     apply (erule disjE) apply (erule exE) apply (erule conjE) apply (rule disjI1)
      apply (rename-tac x v')
      apply (rule-tac x=v' in exI) apply simp apply clarify
      apply (rename-tac V)
      apply (erule-tac x=V in allE)
      apply (subgoal-tac T τ ((V#ϱ1) @ ϱ3) =
       T (shift (length ϱ2) (length (V#ϱ1)) τ) ((V#ϱ1) @ ϱ2 @ ϱ3))
       prefer 2 apply blast apply force
     apply (rule disjI2) apply force
    apply (rule subsetI) apply (simp only: mem-Collect-eq)
    apply (erule disjE) apply (erule exE) apply (erule conjE) apply (rule disjI1)
     apply (rename-tac x v')
     apply (rule-tac x=v' in exI) apply simp apply clarify
     apply (rename-tac V)
     apply (erule-tac x=V in allE)
     apply (subgoal-tac T τ ((V#ϱ1) @ ϱ3) =
       T (shift (length ϱ2) (length (V#ϱ1)) τ) ((V#ϱ1) @ ϱ2 @ ϱ3))
      prefer 2 apply blast apply force
    apply (rule disjI2) apply force done
qed force+

lemma shift-append-preserves-T: shows T τ ϱ3 = T (shift (length ϱ2) 0 τ) (ϱ2@ϱ3)
    using shift-append-preserves-T-aux[of τ [] ϱ3 ϱ2] by auto

lemma drop-shift-preserves-T:
  assumes k: k ≤ length ϱ shows T τ (drop k ϱ) = T (shift k 0 τ) ϱ
proof −
  let ?r2 = take k ϱ and ?r3 = drop k ϱ
  have 1: T τ (?r3) = T (shift (length ?r2) 0 τ) (?r2@?r3)
    using shift-append-preserves-T-aux[of τ [] ?r3 ?r2] by simp
  have 2: ?r2@?r3 = ϱ by simp
  from k have 3: length ?r2 = k by simp
  from 1 2 3 show ?thesis by simp
qed

lemma shift-cons-preserves-T: shows T τ ϱ = T (shift (Suc 0) 0 τ) (b#ϱ)
  using drop-shift-preserves-T[of Suc 0 b#ϱ τ] by simp

lemma compose-shift: shows shift (j+k) c τ = shift j c (shift k c τ)
  by (induction τ arbitrary: j k c) auto
```

lemma *shift-zero-id*[*simp*]: *shift 0 c* $\tau = \tau$
by (*induction* τ *arbitrary*: *c*) *auto*

lemma *lookup-wfenv*: **assumes** *r-g*: $\vdash \varrho,\eta : \Gamma$ **and** *ln*: *lookup* Γ *n* = *Some* τ
shows $\exists$ *v*. $\varrho!n = v \wedge v \in T\ \tau\ \eta$
using *r-g ln*
proof (*induction* ϱ η Γ *arbitrary*: *n* τ *rule*: *wfenv.induct*)
case *wfnil*
then show *?case* **unfolding** *lookup-def* **by** *force*
next
case (*wfvbind* ϱ η Γ *v* τ')
from *wfvbind*(*2*) **have** *vtp*: $v \in T\ \tau'\ \eta$ **.**
show *?case*
proof (*cases n*)
case *0*
from *0 wfvbind*(*4*) **have** *t*: $\tau =$ *shift 0 0* τ' **unfolding** *lookup-def* **by** (*simp add*: *push-ty-def*)
from *0 vtp t* **show** *?thesis* **by** *simp*
next
case (*Suc n'*)
let *?G* = *push-ty* τ' Γ
from *wfvbind*(*4*) *Suc* **obtain** σ *k* **where** *gnp*: (*fst* Γ)!n' = (σ,*k*) **and** *t*: τ = *shift* (*snd* Γ − *k*) *0* σ
and *npg*: $n' <$ *length* (*fst* Γ)
unfolding *lookup-def push-ty-def* **apply** (*cases* $n' <$ *length* (*fst* Γ)) **apply** *auto*
apply (*cases fst* Γ ! n') **apply** *auto* **done**
from *gnp Suc npg t* **have** *ln*: *lookup* Γ n' = *Some* τ **unfolding** *lookup-def* **by** *auto*
from *wfvbind*(*3*) *ln* **obtain** v' **where** *rnp*: $\varrho!n' = v'$ **and** *vt*: $v' \in T\ \tau\ \eta$ **by** *blast*
from *Suc rnp vt* **show** *?thesis* **by** *simp*
qed
next
case (*wftbind* ϱ η Γ *V*)
let *?a* = *fst* Γ **and** *?b* = *snd* Γ
obtain σ *k* **where** *s*: σ = *fst* (*fst* Γ ! *n*) **and** *k*: *k* = *snd* (*fst* Γ ! *n*) **by** *auto*
from *wftbind*(*3*) *s k* **have** *t*: τ = *shift* (*Suc ?b* − *k*) *0* σ **and** *nl*: *n* < *length* (*fst* Γ)
unfolding *push-tyvar-def lookup-def* **apply** *auto*
apply (*case-tac n* < *length* (*fst* Γ), *auto*)+ **done**
let *?t* = *shift* (*?b* − *k*) *0* (*fst* (*?a* ! *n*))
from *wftbind*(*3*) *k* **have** *ln*: *lookup* Γ *n* = *Some ?t*
unfolding *push-tyvar-def lookup-def*
apply (*cases* Γ) **apply** (*rename-tac k' G*) **apply** *simp* **apply** (*case-tac n* < *length k'*) **by** *auto*
from *wftbind*(*2*) *ln* **obtain** v' **where** *rn-vp*: ϱ ! *n* = v' **and** *vp-t*: $v' \in T$ *?t* η **by** *blast*
from *vp-t* **have** $v' \in T$ (*shift* (*Suc 0*) *0 ?t*) (*V* # η) **using** *shift-cons-preserves-T* **by** *auto*
hence *vp-t2*: $v' \in T$ (*shift* (*Suc 0* + (*?b* − *k*)) *0* (*fst* (*?a*!*n*))) (*V* # η)
using *compose-shift*[*of Suc 0 ?b* − *k 0 fst* (*?a*!*n*)] **by** *simp*
from *wftbind*(*1*) **have** *good-ctx* Γ **using** *wfenv-good-ctx* **by** *blast*
from *this k nl* **have** *?b* ≥ *k* **unfolding** *good-ctx-def* **by** *auto*
from *this* **have** *Suc 0* + (*?b* − *k*) = *Suc ?b* − *k* **by** *simp*
from *this vp-t2* **have** *vp-t3*: $v' \in T$ (*shift* (*Suc ?b* − *k*) *0* (*fst* (*?a*!*n*))) (*V* # η) **by** *simp*
from *rn-vp vp-t3 t s* **show** *?case* **by** *auto*
qed

lemma *less-wrong*[*elim*!]: ⟦ *v* ⊑ *Wrong*; *v* = *Wrong* ⟹ *P* ⟧ ⟹ *P*
by (*case-tac v*) *auto*

lemma *less-nat*[*elim*!]: ⟦ *v* ⊑ *VNat n*; *v* = *VNat n* ⟹ *P* ⟧ ⟹ *P*
by (*case-tac v*) *auto*

lemma *less-fun*[*elim*!]: ⟦ *v* ⊑ *Fun f*; ⋀ *f'*. ⟦ *v* = *Fun f'*; *fset f'* ⊆ *fset f* ⟧ ⟹ *P* ⟧ ⟹ *P*
by (*case-tac v*) *auto*

lemma *less-refl*[*simp*]: *v* ⊑ *v*
proof (*induction v*)

```
    case (Abs v')
    then show ?case by (cases v') auto
qed force+

lemma less-trans: fixes v1::val and v2::val and v3::val
  shows ⟦ v1 ⊑ v2; v2 ⊑ v3 ⟧ ⟹ v1 ⊑ v3
proof (induction v2 arbitrary: v1 v3)
  case (VNat n)
  then show ?case by (cases v1) auto
next
  case (Fun t)
  then show ?case
    apply (cases v1)
      apply force
     apply simp
     apply (cases v3)
       apply auto done
next
  case (Abs v)
  then show ?case
    apply (cases v1) apply force apply force apply (case-tac v3) apply force apply force
     apply (rename-tac v' v3') apply simp apply (cases v) apply (case-tac v')
      apply force apply force
     apply (case-tac v3') apply force apply simp apply (case-tac v')
      apply force+ done
next
  case Wrong
  then show ?case by auto
qed

lemma T-down-closed: assumes vt: v ∈ T τ η and vp-v: v' ⊑ v
  shows v' ∈ T τ η
  using vt vp-v
proof (induction τ arbitrary: v v' η)
  case (TVar x v v' η)
  then show ?case
    apply simp apply (case-tac x < length η)
     apply simp apply clarify
     apply (rule-tac x=v' in exI)
     apply simp apply (rule conjI)
      apply (rule less-trans) apply blast apply blast
     apply (case-tac v')
       apply (case-tac v)
         apply force+
      apply (case-tac v)
        apply force+ done
next
  case TNat
  then show ?case by auto
next
  case (Fun τ1 τ2)
  then show ?case apply simp apply clarify apply (rule-tac x=f' in exI) apply fastforce done
next
  case (Forall τ v v' η)
  then show ?case
    apply simp apply (erule disjE) apply clarify apply (cases v') apply force apply force
     apply simp apply (rename-tac v'') apply (case-tac v'') apply simp apply simp apply clarify
     apply (erule-tac x=V in allE) apply blast
    apply force
   apply simp
   apply (case-tac v') apply auto done
```

qed

lemma *wrong-not-in-T*: $Wrong \notin T\ \tau\ \eta$
by (*induction* τ) *auto*

lemma *fun-app*: **assumes** *vmn*: $V \subseteq T\ (m \to n)\ \eta$ **and** *v2s*: $V' \subseteq T\ m\ \eta$
shows *apply-fun* $V\ V' \subseteq T\ n\ \eta$
using *vmn v2s* **apply** *simp* **apply** (*rule conjI*)
prefer *2* **apply** *force*
apply *clarify*
apply (*erule disjE*)
prefer *2* **using** *wrong-not-in-T* **apply** *blast*
apply *clarify* **apply** (*rename-tac* v'') **apply** (*case-tac* v') **apply** *auto*
apply (*rename-tac v1 v2*) **apply** (*case-tac* $v1 \sqsubseteq v''$) **apply** *auto*
apply (*subgoal-tac* $\forall\, v1\ v2'.$
$(v1, v2') \in fset\ x2 \longrightarrow v1 \in T\ m\ \eta \longrightarrow (\exists\, v2.\ v2 \in T\ n\ \eta \wedge v2' \sqsubseteq v2)$)
prefer *2* **apply** *blast*
apply (*rename-tac v1 v2*)
apply (*erule-tac* $x{=}v1$ **in** *allE*) **apply** (*erule-tac* $x{=}v2$ **in** *allE*) **apply** (*erule impE*) **apply** *simp*
apply (*erule impE*) **using** *T-down-closed* **apply** *blast*
apply *clarify* **using** *T-down-closed* **apply** *blast*
done

lemma *T-eta*: $\{v.\ \exists\, v'.\ v' \in T\ \sigma\ (\eta) \wedge v \sqsubseteq v' \wedge v \neq Wrong\} = T\ \sigma\ \eta$
apply *auto*
using *T-down-closed* **apply** *blast*
apply (*rename-tac v*)
apply (*rule-tac* $x{=}v$ **in** *exI*)
apply *simp*
using *wrong-not-in-T* **apply** *blast* **done**

lemma *compositionality*: $T\ \tau\ (\eta1\ @\ (T\ \sigma\ (\eta1 @ \eta2))\ \#\ \eta2) = T\ (subst\ (length\ \eta1)\ \sigma\ \tau)\ (\eta1 @ \eta2)$
proof (*induction* τ *arbitrary*: $\sigma\ \eta1\ \eta2$)
case (*TVar x*)
then show *?case*
apply (*case-tac length* $\eta1 = x$) **apply** *simp* **using** *T-eta* **apply** *blast*
apply (*case-tac length* $\eta1 < x$) **apply** (*subgoal-tac* $\exists\ x'.\ x = Suc\ x'$) **prefer** *2*
apply (*cases x*)
apply *force+*
done
next
case *TNat*
then show *?case* **by** *auto*
next
case (*Fun* $\tau1\ \tau2$)
then show *?case* **by** *auto*
next
case (*Forall* τ)
show $T\ (Forall\ \tau)\ (\eta1\ @\ T\ \sigma\ (\eta1\ @\ \eta2)\ \#\ \eta2) =$
$T\ (subst\ (length\ \eta1)\ \sigma\ (Forall\ \tau))\ (\eta1\ @\ \eta2)$
apply *simp*
apply (*rule equalityI*) **apply** (*rule subsetI*) **apply** (*simp only*: *mem-Collect-eq*)
apply (*erule disjE*) **prefer** *2* **apply** *force* **apply** (*erule exE*) **apply** (*erule conjE*) **apply** (*rule disjI1*)
apply (*rule-tac* $x{=}v'$ **in** *exI*) **apply** *simp* **apply** *clarify*
apply (*erule-tac* $x{=}V$ **in** *allE*)
prefer *2* **apply** (*rule subsetI*) **apply** (*simp only*: *mem-Collect-eq*)
apply (*erule disjE*) **prefer** *2* **apply** *force* **apply** (*erule exE*) **apply** (*erule conjE*) **apply** (*rule disjI1*)
apply (*rule-tac* $x{=}v'$ **in** *exI*) **apply** *simp* **apply** *clarify*
apply (*erule-tac* $x{=}V$ **in** *allE*)
defer
proof −

```
    fix x v' V
    let ?L1 = length η1 and ?R1 = V#η1 and ?s = shift (Suc 0) 0 σ
    assume 1: v' ∈ T τ (V # (η1 @ T σ (η1 @ η2) # η2))
    from 1 have a: v' ∈ T τ (?R1 @ T σ (η1@η2) # η2) by simp

    have b: T σ (η1@η2) = T ?s (V#(η1@η2)) by (rule shift-cons-preserves-T)
    from a b have c: v' ∈ T τ (?R1 @ T ?s (?R1 @ η2) # η2) by simp
    from Forall[of ?R1 ?s η2] have 2: T τ (?R1 @ T ?s (?R1 @ η2) # η2) =
                                  T (subst (length ?R1) ?s τ) (?R1 @ η2) by simp
    from c 2 show v' ∈ T (subst (Suc ?L1) ?s τ) (V # (η1 @ η2)) by simp
  next
    fix x v' V
    let ?L1 = length η1 and ?R1 = V#η1 and ?s = shift (Suc 0) 0 σ
    assume 1: v' ∈ T (subst (Suc (length η1)) (shift (Suc 0) 0 σ) τ) (V # η1 @ η2)
    from Forall[of ?R1 ?s η2] have 2: T τ (?R1 @ T ?s (?R1 @ η2) # η2) =
                                  T (subst (length ?R1) ?s τ) (?R1 @ η2) by simp
    from 1 2 have 3: v' ∈ T τ (?R1 @ T ?s (?R1 @ η2) # η2) by simp
    have b: T σ (η1@η2) = T ?s (V#(η1@η2)) by (rule shift-cons-preserves-T)
    from 3 b have a: v' ∈ T τ (?R1 @ T σ (η1@η2) # η2) by simp
    from this show v' ∈ T τ (V # η1 @ T σ (η1 @ η2) # η2) by simp
  qed
qed

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 using it IH
proof (induction rule: iterate.induct)
  case (iterate-none Ee ϱ)
  then show ?case by auto
next
  case (iterate-again Ee ϱ f f')
  from iterate-again have f-st: f ∈ T (σ→τ) η by blast
  from iterate-again f-st have Ee (f#ϱ) ⊆ T (σ→τ) η by blast
  from this iterate-again show ?case by auto
qed

theorem welltyped-dont-go-wrong:
  assumes wte: Γ ⊢ e : τ and wfr: ⊢ ϱ,η : Γ
  shows E e ϱ ⊆ T τ η
  using wte wfr
proof (induction Γ e τ arbitrary: ϱ η rule: well-typed.induct)
  case (wtnat Γ n ϱ η)
  then show ?case by auto
next
  case (wtvar Γ n τ ϱ η)
  from wtvar obtain v where lx: ϱ ! n = v and vt: v ∈ T τ ηusing lookup-wfenv by blast
  from lx vt show ?case apply auto using T-down-closed[of ϱ!n τ η] by blast
next
  case (wtapp Γ e σ τ e' ϱ η)
  from wtapp have Ee: E e ϱ ⊆ T (σ → τ) η by blast
  from wtapp have Eep: E e' ϱ ⊆ T σ η by blast
  from Ee Eep show ?case using fun-app by simp
next
  case (wtlam σ Γ e τ ϱ η)
  show ?case
    apply simp apply (rule subsetI) apply clarify apply (rule-tac x=f in exI) apply simp
    apply clarify apply (erule-tac x=v1 in allE) apply (erule-tac x=v2' in allE) apply clarify
  proof −
    fix f v1 v2' v2
    assume v1-T: v1 ∈ T σ η and v2-E: v2 ∈ E e (v1#ϱ) and v2p-v2: v2' ⊑ v2
```

```
    let ?r = v1#ϱ
    from wtlam(3) v1-T have 1: ⊢ v1#ϱ,η : push-ty σ Γ by blast
    from wtlam(2) 1 have IH: E e (v1#ϱ) ⊆ T τ η by blast
    from IH v2-E have v2-T: v2 ∈ T τ η by blast
    from v2-T have v2-Tb: v2 ∈ T τ η by simp
    from v2-Tb v2p-v2 show ∃ v2. v2 ∈ T τ η ∧ v2' ⊑ v2 by blast
  qed
next
  case (wtfix σ τ Γ e ϱ η)
  have ∀ v. iterate (E e) ϱ v ⟶ v ∈ T (σ → τ) η
  proof clarify
    fix v assume it: iterate (E e) ϱ v
    have 1: ∀ v. v ∈ T (σ → τ) η ⟶ E e (v#ϱ) ⊆ T (σ → τ) η
    proof clarify
      fix v' v'' assume 2: v' ∈ T (σ→τ) η and 3: v'' ∈ E e (v'#ϱ)
      from wtfix(3) 2 have ⊢ (v'#ϱ),η : push-ty (σ → τ) Γ by blast
      from wtfix(2) this have IH: E e (v'#ϱ) ⊆ T (σ→τ) η by blast
      from 3 IH have v'' ∈ T (σ→τ) η by blast
      from this show v'' ∈ T (σ → τ) η by simp
    qed
    from it 1 show v ∈ T (σ → τ) η using iterate-sound[of E e ϱ v σ τ] by blast
  qed
  from this show ?case by auto
next
  case (wtabs Γ e τ ϱ η)
  show ?case apply simp apply (rule subsetI) apply (simp only: mem-Collect-eq)
    apply (erule disjE) apply (erule exE) apply (erule conjE) apply (rule disjI1)
     apply (rule-tac x=v' in exI) apply simp apply clarify prefer 2 apply (rule disjI2)
     apply force
  proof −
    fix x v' V assume 2: v' ∈ E e ϱ
    from wtabs(3) have 3: ⊢ ϱ,(V#η) : push-tyvar Γ by blast
    from wtabs(2) 3 have IH: E e ϱ ⊆ T τ (V#η) by blast
    from 2 IH show v' ∈ T τ (V#η) by (case-tac ϱ) auto
  qed
next
  case (wtinst Γ e τ σ ϱ η)
  from wtinst(2) wtinst(3) have IH: E e ϱ ⊆ T (Forall τ) η by blast
  show ?case
    apply simp apply (rule conjI)
     apply (rule subsetI) apply (simp only: mem-Collect-eq) apply (erule exE)
     apply (erule conjE)+
  proof −
    fix x v' assume vp-E: v' ∈ E e ϱ and vp-w: v' ≠ Wrong and
      x: x ∈ (case v' of Abs None ⇒ {} | Abs (Some xa) ⇒ return xa
            | - ⇒ {v'. v' ⊑ Wrong})
    from IH vp-E have vp-T: v' ∈ T (Forall τ) η by blast
    from vp-T have (∃ v''. v' = Abs (Some v'') ∧ (∀ V. v'' ∈ T τ (V#η)))
                          ∨ v' = Abs None by simp
    from this show x ∈ T (subst 0 σ τ) η
    proof
      assume ∃ v''. v' = Abs (Some v'') ∧ (∀ V. v'' ∈ T τ (V#η))
      from this obtain v'' where vp: v' = Abs (Some v'') and
        vpp-T: ∀ V. v'' ∈ T τ (V#η) by blast
      from vp x have x-vpp: x ⊑ v'' by auto
      let ?V = T σ η
      from vpp-T have v'' ∈ T τ (?V#η) by blast
      from this have v'' ∈ T (subst 0 σ τ) η using compositionality[of τ [] σ] by simp
      from this x-vpp show x ∈ T (subst 0 σ τ) η using T-down-closed by blast
    next
      assume vp: v' = Abs None
```

from *vp x* **show** $x \in T\ (subst\ 0\ \sigma\ \tau)\ \eta$ **by** *simp*
qed
next
from *IH* **show** $\{v.\ v = Wrong \wedge Wrong \in E\ e\ \varrho\} \subseteq T\ (subst\ 0\ \sigma\ \tau)\ \eta$
using *wrong-not-in-T* **by** *auto*
qed
qed

end

17 Semantics of mutable references

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

datatype *ty* = *TNat* | *TFun ty ty* (**infix** $\rightarrow$ *60*) | *TPair ty ty* | *TRef ty*

type-synonym *name* = *nat*

datatype *exp* = *EVar name* | *ENat nat* | *ELam ty exp* | *EApp exp exp*
| *EPrim nat* $\Rightarrow$ *nat* $\Rightarrow$ *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* $\times$ *val*) *fset* | *VPair val val* | *VAddr nat* | *Wrong*

type-synonym *func* = (*val* $\times$ *val*) *fset*
type-synonym *store* = *func*

inductive *val-le* :: *val* $\Rightarrow$ *val* $\Rightarrow$ *bool* (**infix** $\sqsubseteq$ *52*) **where**
vnat-le[*intro*!]: (*VNat n*) $\sqsubseteq$ (*VNat n*) |
vaddr-le[*intro*!]: (*VAddr a*) $\sqsubseteq$ (*VAddr a*) |
wrong-le[*intro*!]: *Wrong* $\sqsubseteq$ *Wrong* |
vfun-le[*intro*!]: *t1* $|\subseteq|$ *t2* $\Longrightarrow$ (*VFun t1*) $\sqsubseteq$ (*VFun t2*) |
vpair-le[*intro*!]: $[\![\ v1 \sqsubseteq v1';\ v2 \sqsubseteq v2'\ ]\!] \Longrightarrow$ (*VPair v1 v2*) $\sqsubseteq$ (*VPair v1′ v2′*)

primrec *vsize* :: *val* $\Rightarrow$ *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* $\Rightarrow$ (*′a* $\times$ *store*) *set*

definition *bind* :: *′a M* $\Rightarrow$ (*′a* $\Rightarrow$ *′b M*) $\Rightarrow$ *′b M* **where**
bind m f μ1 $\equiv$ { (*v*,*μ3*). $\exists$ *v′ μ2*. (*v′*,*μ2*) $\in$ *m μ1* $\wedge$ (*v*,*μ3*) $\in$ *f v′ μ2* }
declare *bind-def*[*simp*]

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

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

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* (λ*a*.λ*r*. *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* $\sqsubseteq$ *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* **by** (*induction v*) *auto*

proposition *val-le-trans*: $[\![$ *v1* $\sqsubseteq$ *v2*; *v2* $\sqsubseteq$ *v3* $]\!] \Longrightarrow$ *v1* $\sqsubseteq$ *v3*
by (*induction v2 arbitrary*: *v1 v3*) *blast+*

proposition *val-le-antisymm*: $[\![$ *v1* $\sqsubseteq$ *v2*; *v2* $\sqsubseteq$ *v1* $]\!] \Longrightarrow$ *v1* = *v2*
by (*induction v1 arbitrary*: *v2*) *blast+*

end