

Declarative Semantics for Functional Languages

Jeremy G. Siek

December 7, 2022

Abstract

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

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

Contents

1	Syntax of the lambda calculus	2
2	Small-step semantics of CBV lambda calculus	2
3	Big-step semantics of CBV lambda calculus	4
3.1	Big-step semantics is sound wrt. small-step semantics	5
3.2	Big-step semantics is deterministic	11
4	Declarative semantics as a relational semantics	13
5	Declarative semantics as a denotational semantics	14
6	Relational and denotational views are equivalent	14
7	Subsumption and change of environment	14
8	Declarative semantics as a non-deterministic interpreter	17
8.1	Non-determinism monad	17
8.2	Non-deterministic interpreter	17
9	Declarative semantics as a type system	18
10	Declarative semantics with tables as lists	19
10.1	Definition of values for declarative semantics	19
10.2	Properties about values	19
10.3	Declarative semantics as a denotational semantics	23
10.4	Subsumption and change of environment	23
11	Equivalence of denotational and type system views	26

12 Soundness of the declarative semantics wrt. operational	32
12.1 Substitution preserves denotation	32
12.2 Reduction preserves denotation	34
12.3 Progress	34
12.4 Logical relation between values and big-step values	35
12.5 Denotational semantics sound wrt. big-step	37
13 Completeness of the declarative semantics wrt. operational	39
13.1 Reverse substitution preserves denotation	39
13.2 Reverse reduction preserves denotation	43
13.3 Completeness	44
14 Soundness wrt. contextual equivalence	44
14.1 Denotational semantics is a congruence	44
14.2 Auxiliary lemmas	45
14.3 Soundness wrt. contextual equivalence	46
14.4 Denotational equalities regarding reduction	47
15 Correctness of an optimizer	48
16 Semantics and type soundness for System F	49
16.1 Syntax and values	49
16.2 Set monad	49
16.3 Denotational semantics	49
16.4 Types: substitution and semantics	50
16.5 Type system	50
16.6 Well-typed Programs don't go wrong	51
17 Semantics of mutable references	58
17.1 Denotations (values)	58
17.2 Non-deterministic state monad	58
17.3 Denotational semantics	59

1 Syntax of the lambda calculus

```
theory Lambda
imports Main
begin

type-synonym name = nat

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

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

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

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

end
```

2 Small-step semantics of CBV lambda calculus

```
theory SmallStepLam
imports Lambda
begin
```

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

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

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

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

definition *is-val* :: *exp* ⇒ *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  $\times$  bval) list

type-synonym benv = (name  $\times$  bval) list

inductive eval :: benv  $\Rightarrow$  exp  $\Rightarrow$  bval  $\Rightarrow$  bool (-  $\vdash$  -  $\Downarrow$  - [50,50,50] 51) where
  eval-nat[intro!]:  $\varrho \vdash \text{ENat } n \Downarrow \text{BNat } n \mid$ 
  eval-var[intro!]:  $\text{lookup } \varrho \ x = \text{Some } v \Longrightarrow \varrho \vdash \text{EVar } x \Downarrow v \mid$ 
  eval-lam[intro!]:  $\varrho \vdash \text{ELam } x \ e \Downarrow \text{BClos } x \ e \ \varrho \mid$ 
  eval-app[intro!]:  $\llbracket \varrho \vdash e1 \Downarrow \text{BClos } x \ e \ \varrho'; \varrho \vdash e2 \Downarrow \text{arg};$ 
     $(x, \text{arg}) \# \varrho' \vdash e \Downarrow v \rrbracket \Longrightarrow$ 
     $\varrho \vdash \text{EApp } e1 \ e2 \Downarrow v \mid$ 
  eval-prim[intro!]:  $\llbracket \varrho \vdash e1 \Downarrow \text{BNat } n1; \varrho \vdash e2 \Downarrow \text{BNat } n2 ; n3 = f \ n1 \ n2 \rrbracket \Longrightarrow$ 
     $\varrho \vdash \text{EPrim } f \ e1 \ e2 \Downarrow \text{BNat } n3 \mid$ 
  eval-if0[intro!]:  $\llbracket \varrho \vdash e1 \Downarrow \text{BNat } 0; \varrho \vdash e3 \Downarrow v3 \rrbracket \Longrightarrow$ 
     $\varrho \vdash \text{EIf } e1 \ e2 \ e3 \Downarrow v3 \mid$ 
  eval-if1[intro!]:  $\llbracket \varrho \vdash e1 \Downarrow \text{BNat } n; n \neq 0; \varrho \vdash e2 \Downarrow v2 \rrbracket \Longrightarrow$ 
     $\varrho \vdash \text{EIf } e1 \ e2 \ e3 \Downarrow v2$ 

inductive-cases
  eval-nat-inv[elim!]:  $\varrho \vdash \text{ENat } n \Downarrow v$  and
  eval-var-inv[elim!]:  $\varrho \vdash \text{EVar } x \Downarrow v$  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 $env = (name \times exp)\ list$

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

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

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

$bs-env :: benv \Rightarrow env \Rightarrow bool$ **where**
 $bs-nat[intro!]: bs-val\ (BNat\ n)\ (ENat\ n)$ |
 $bs-clos[intro!]: \llbracket bs-env\ \varrho\ \varrho'; FV\ (ELam\ x\ (psubst\ ((x, EVar\ x)\#\varrho')\ e)) = \{\} \rrbracket \Longrightarrow$
 $bs-val\ (BClos\ x\ e\ \varrho)\ (ELam\ x\ (psubst\ ((x, EVar\ x)\#\varrho')\ e))$ |
 $bs-nil[intro!]: bs-env\ []\ []$ |
 $bs-cons[intro!]: \llbracket bs-val\ w\ v; bs-env\ \varrho\ \varrho' \rrbracket \Longrightarrow bs-env\ ((x, w)\#\varrho)\ ((x, v)\#\varrho')$

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

$bs-clos-inv[elim!]: bs-val\ (BClos\ x\ e\ \varrho')\ v1$ **and**
 $bs-nat-inv[elim!]: bs-val\ (BNat\ n)\ v$

lemma $bs-val-is-val[intro!]: bs-val\ w\ v \Longrightarrow is-val\ v$

by (cases w) auto

lemma $lookup-bs-env: \llbracket bs-env\ \varrho\ \varrho'; lookup\ \varrho\ x = Some\ w \rrbracket \Longrightarrow$

$\exists v. lookup\ \varrho'\ x = Some\ v \wedge bs-val\ w\ v$

by (induction ϱ arbitrary: $\varrho'\ x\ w$) auto

lemma $app-red-cong1: e1 \longrightarrow* e1' \Longrightarrow EApp\ e1\ e2 \longrightarrow* EApp\ e1'\ e2$

by (induction rule: multi-step.induct) blast+

lemma $app-red-cong2: e2 \longrightarrow* e2' \Longrightarrow EApp\ e1\ e2 \longrightarrow* EApp\ e1\ e2'$

by (induction rule: multi-step.induct) blast+

lemma $prim-red-cong1: e1 \longrightarrow* e1' \Longrightarrow EPrim\ f\ e1\ e2 \longrightarrow* EPrim\ f\ e1'\ e2$

by (induction rule: multi-step.induct) blast+

lemma $prim-red-cong2: e2 \longrightarrow* e2' \Longrightarrow EPrim\ f\ e1\ e2 \longrightarrow* EPrim\ f\ e1\ e2'$

by (induction rule: multi-step.induct) blast+

lemma $if-red-cong1: e1 \longrightarrow* e1' \Longrightarrow EIf\ e1\ e2\ e3 \longrightarrow* EIf\ e1'\ e2\ e3$

by (induction rule: multi-step.induct) blast+

lemma $multi-step-trans: \llbracket e1 \longrightarrow* e2; e2 \longrightarrow* e3 \rrbracket \Longrightarrow e1 \longrightarrow* e3$

proof (induction arbitrary: $e3$ rule: multi-step.induct)

case ($ms-cons\ e1\ e2\ e3\ e3'$)

then have $e2 \longrightarrow* e3'$ **by** auto

with $ms-cons(1)$ **show** ?case **by** blast

qed blast

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

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

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

definition *equiv-env* :: *env* \Rightarrow *env* \Rightarrow *bool* **where**
equiv-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]*: $FV\ v = \{\} \implies sdom\ ((x, v)\#\varrho) = insert\ x\ (sdom\ \varrho)$
unfolding *sdom-def* **by** *auto*

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

proof (*induction* ϱ)
case (*Cons* $b\ \varrho$)
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*: $lookup\ \varrho\ x = None \implies x \notin sdom\ \varrho$

proof (*induction* ϱ)
case (*Cons* $b\ \varrho$)
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*: $equiv\ env\ \varrho\ \varrho' \implies psubst\ \varrho\ e = psubst\ \varrho'\ e$

proof (*induction e arbitrary:* $\varrho\ \varrho'$)
case (*EVar* x)
show *?case*
proof (*cases* $lookup\ \varrho\ x$)
case *None* **from** *None* **have** $lx: lookup\ \varrho\ x = None$ **by** *simp*
show *?thesis*
proof (*cases* $lookup\ \varrho'\ x$)
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** $lx: lookup\ \varrho\ x = Some\ v$ **by** *simp*
show *?thesis*
proof (*cases* $lookup\ \varrho'\ x$)
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')# $\varrho$ ) ((x',EVar x')# $\varrho'$ ) by (simp add: equiv-env-def)
with ELam show ?case by (simp add: equiv-env-def)
qed fastforce+

lemma subst-psubst:  $\llbracket \text{closed-env } \varrho; \text{FV } v = \{\} \rrbracket \implies$ 
  subst x v (psubst ((x, EVar x) #  $\varrho$ ) e) = psubst ((x, v) #  $\varrho$ ) e
proof (induction e arbitrary: x v  $\varrho$ )
case (EVar x x' v  $\varrho$ )
show ?case
proof (cases x = x')
case True
then show ?thesis by force
next
case False from False have xxp:  $x \neq x'$  by simp
show ?thesis
proof (cases lookup  $\varrho$  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:  $x \in \text{sdom } \varrho$  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- $x$ p:  $x \neq x'$ 
let ?r = (x',EVar x') #  $\varrho$ 
from ELam have IHprem: closed-env ((x', EVar x') #  $\varrho$ ) using closed-env-def by auto
have psubst ((x',EVar x')#(x, EVar x)# $\varrho$ ) e = psubst ((x,EVar x)#(x',EVar x') #  $\varrho$ ) e
  apply (rule psubst-change) using x- $x$ p equiv-env-def by auto
from this have subst x v (psubst ((x', EVar x') # (x, EVar x) #  $\varrho$ ) e)
  = subst x v (psubst ((x,EVar x)#(x',EVar x') #  $\varrho$ ) e) by simp
also with ELam IHprem have ... = psubst ((x,v)#(x',EVar x')# $\varrho$ ) e
  using ELam(1)[of (x',EVar x')# $\varrho$  v x] by simp
also have ... = psubst ((x',EVar x')#(x,v)# $\varrho$ ) e
  apply (rule psubst-change) using x- $x$ p equiv-env-def sdom-def by auto

```


finally show $\text{subst } x \ v \ (\text{psubst } ((x', \text{EVar } x') \# (x, \text{EVar } x) \# \varrho) \ e)$
 $= \text{psubst } ((x', \text{EVar } x') \# (x, v) \# \varrho) \ e \ .$

qed
qed
qed fastforce+

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

lemma bs-env-dom : $\text{bs-env } \varrho \ \varrho' \implies \text{set } (\text{map } \text{fst } \varrho) = \text{sdom } \varrho'$
proof ($\text{induction } \varrho \ \text{arbitrary: } \varrho'$)
case Nil
then show $?case$ **by** ($\text{force simp: sdom-def}$)
next
case ($\text{Cons } b \ \varrho$)
then show $?case$
proof ($\text{cases } b$)
case ($\text{Pair } x \ v'$)
with Cons **show** $?thesis$ **by** ($\text{cases } v'$) force+
qed
qed

lemma $\text{closed-env-cons}[\text{intro!}]$: $FV \ v = \{\} \implies \text{closed-env } \varrho'' \implies \text{closed-env } ((a, v) \# \varrho'')$
by ($\text{simp add: closed-env-def sdom-def}$)

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

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

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

```

using ev r-rp fv-e
proof (induction arbitrary:  $\varrho'$  rule: eval.induct)
  case (eval-nat  $\varrho$  n  $\varrho'$ )
  then show ?case by (rule-tac x=ENat n in exI) auto
next
  case (eval-var  $\varrho$  x w  $\varrho'$ )
  from eval-var obtain v where lx: lookup  $\varrho'$  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  $\varrho$  x e  $\varrho'$ )
  from eval-lam(1) have dom-eq: set (map fst  $\varrho$ ) = sdom  $\varrho'$  using bs-env-dom by blast
  from eval-lam(1) have closed-env ((x,EVar x)# $\varrho'$ ) using bs-env-closed closed-env-def by auto
  from this psubst-fv have FV (psubst ((x,EVar x)# $\varrho'$ ) e) = FV e - sdom ((x,EVar x)# $\varrho'$ ) by blast
  from this eval-lam(2) dom-eq
  have fv-lam: FV (ELam x (psubst ((x,EVar x)# $\varrho'$ ) e)) = {} by auto
  from fv-lam eval-lam have 1: bs-val (BClos x e  $\varrho$ ) (ELam x (psubst ((x,EVar x)# $\varrho'$ ) e)) by auto
  from this eval-lam fv-lam show ?case
    by (rule-tac x=ELam x (psubst ((x,EVar x)# $\varrho'$ ) e) in exI) auto
next
  case (eval-app  $\varrho$  e1 x e  $\varrho'$  e2 arg v  $\varrho''$ )
  from eval-app(8) have FV e1  $\subseteq$  set (map fst  $\varrho$ ) by auto
  from this eval-app(7) eval-app(4)[of  $\varrho''$ ] obtain v1 where e1-v1: psubst  $\varrho''$  e1  $\longrightarrow^*$  v1 and
    vv1: is-val v1 and clos-v1: bs-val (BClos x e  $\varrho'$ ) v1 by (simp, blast)
  from eval-app(8) have FV e2  $\subseteq$  set (map fst  $\varrho$ ) by auto
  from this eval-app(5) eval-app(7) obtain v2 where e2-v2: psubst  $\varrho''$  e2  $\longrightarrow^*$  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  $\varrho2$  where rpp-r2: bs-env  $\varrho'$   $\varrho2$  and fv-v1: FV v1 = {} and
    v1-lam: v1 = ELam x (psubst ((x,EVar x)# $\varrho2$ ) e) by auto
  let ?r = ((x,v2) #  $\varrho2$ )
  from rpp-r2 have cr2: closed-env  $\varrho2$  using bs-env-closed by auto
  from this have closed-env ((x,EVar x)# $\varrho2$ ) using closed-env-def sdom-def by auto
  from this have fve: FV (psubst ((x,EVar x)# $\varrho2$ ) e) = FV e - sdom ((x,EVar x)# $\varrho2$ )
    using psubst-fv[of (x,EVar x)# $\varrho2$ ] by blast
  let ?r2 = ((x, arg) #  $\varrho'$ )
  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  $\subseteq$  set (map fst ((x, arg) #  $\varrho'$ )) by auto
  from this rr eval-app(6) obtain v3 where e-v3: psubst ?r e  $\longrightarrow^*$  v3 and
    vv3: isval v3 and v-v3: bs-val v v3 by (simp, blast)
  from e1-v1 have 1: EApp (psubst  $\varrho''$  e1) (psubst  $\varrho''$  e2)  $\longrightarrow^*$  EApp v1 (psubst  $\varrho''$  e2)
    by (rule app-red-cong1)
  from e2-v2 have 2: EApp v1 (psubst  $\varrho''$  e2)  $\longrightarrow^*$  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)# $\varrho2$ ) e
  from v1-lam vv2b have 3: EApp (ELam x ?body) v2  $\longrightarrow$ 
    subst x v2 (psubst ((x,EVar x)# $\varrho2$ ) e) using beta[of v2 x ?body] by simp
  have 4: subst x v2 (psubst ((x,EVar x)# $\varrho2$ ) e) = psubst ?r e
    apply (rule subst-psubst) using fv-v2 cr2 by auto
  have 4: subst x v2 (psubst ((x,EVar x)# $\varrho2$ ) e) = psubst ?r e
    apply (rule subst-psubst) using fv-v2 cr2 by auto
  from 1 2 have 5: psubst  $\varrho''$  (EApp e1 e2)  $\longrightarrow^*$  EApp v1 v2 apply simp
    by (rule multi-step-trans) auto
  from 5 3 4 v1-lam have 6: psubst  $\varrho''$  (EApp e1 e2)  $\longrightarrow^*$  psubst ?r e
    apply simp apply (rule multi-step-trans) apply assumption apply blast done
  from 6 e-v3 have 7: psubst  $\varrho''$  (EApp e1 e2)  $\longrightarrow^*$  v3 by (rule multi-step-trans)
  from 7 vv3 v-v3 show ?case by blast
next
  case (eval-prim  $\varrho$  e1 n1 e2 n2 n3 f  $\varrho'$ )

```

from *eval-prim*(7) **have** $FV\ e1 \subseteq \text{set}(\text{map}\ \text{fst}\ \varrho)$ **by** *auto*
from *this eval-prim* **obtain** $v1$ **where** $e1-v1: \text{psubst}\ \varrho'\ e1 \longrightarrow^* v1$ **and**
 $n1-v1: \text{bs-val}\ (BNat\ n1)\ v1$ **by** *blast*
from $n1-v1$ **have** $v1: v1 = ENat\ n1$ **by** *blast*

from *eval-prim*(7) **have** $FV\ e2 \subseteq \text{set}(\text{map}\ \text{fst}\ \varrho)$ **by** *auto*
from *this eval-prim* **obtain** $v2$ **where** $e2-v2: \text{psubst}\ \varrho'\ e2 \longrightarrow^* v2$ **and**
 $n2-v2: \text{bs-val}\ (BNat\ n2)\ v2$ **by** *blast*
from $n2-v2$ **have** $v2: v2 = ENat\ n2$ **by** *blast*

from $e1-v1$ **have** $1: EPrim\ f\ (\text{psubst}\ \varrho'\ e1)\ (\text{psubst}\ \varrho'\ e2) \longrightarrow^* EPrim\ f\ v1\ (\text{psubst}\ \varrho'\ e2)$
by (*rule prim-red-cong1*)
from $e2-v2$ **have** $2: EPrim\ f\ v1\ (\text{psubst}\ \varrho'\ e2) \longrightarrow^* EPrim\ f\ v1\ v2$
by (*rule prim-red-cong2*)
from $v1\ v2$ **have** $3: EPrim\ f\ v1\ v2 \longrightarrow ENat\ (f\ n1\ n2)$ **by** *auto*
from $1\ 2$ **have** $5: \text{psubst}\ \varrho'\ (EPrim\ f\ e1\ e2) \longrightarrow^* EPrim\ f\ v1\ v2$ **apply** *simp*
apply (*rule multi-step-trans*) **apply** *auto* **done**
from $5\ 3$ **have** $6: \text{psubst}\ \varrho'\ (EPrim\ f\ e1\ e2) \longrightarrow^* ENat\ (f\ n1\ n2)$ **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** $ex1$) **by** *auto*

next

case (*eval-if0* $\varrho\ e1\ e3\ v3\ e2\ \varrho'$)
from *eval-if0*(6) **have** $FV\ e1 \subseteq \text{set}(\text{map}\ \text{fst}\ \varrho)$ **by** *auto*
from *this eval-if0* **obtain** $v1$ **where** $e1-v1: \text{psubst}\ \varrho'\ e1 \longrightarrow^* v1$ **and**
 $n1-v1: \text{bs-val}\ (BNat\ 0)\ v1$ **by** *blast*
from $n1-v1$ **have** $v1: v1 = ENat\ 0$ **by** *blast*
from *eval-if0*(6) **have** $FV\ e3 \subseteq \text{set}(\text{map}\ \text{fst}\ \varrho)$ **by** *auto*
from *this eval-if0* **obtain** $v3'$ **where** $e3-v3: \text{psubst}\ \varrho'\ e3 \longrightarrow^* v3'$ **and**
 $v3-v3: \text{bs-val}\ v3\ v3'$ **by** *blast*

from $e1-v1$ **have** $1: EIf\ (\text{psubst}\ \varrho'\ e1)\ (\text{psubst}\ \varrho'\ e2)\ (\text{psubst}\ \varrho'\ e3) \longrightarrow^* EIf\ v1\ (\text{psubst}\ \varrho'\ e2)\ (\text{psubst}\ \varrho'\ e3)$ **by** (*rule if-red-cong1*)
from $v1$ **have** $3: EIf\ v1\ (\text{psubst}\ \varrho'\ e2)\ (\text{psubst}\ \varrho'\ e3) \longrightarrow (\text{psubst}\ \varrho'\ e3)$ **by** *auto*
from $1\ 3$ **have** $5: \text{psubst}\ \varrho'\ (EIf\ e1\ e2\ e3) \longrightarrow^* \text{psubst}\ \varrho'\ e3$ **apply** *simp*
apply (*rule multi-step-trans*) **apply** *assumption* **apply** *blast* **done**
from $5\ e3-v3$ **have** $6: \text{psubst}\ \varrho'\ (EIf\ e1\ e2\ e3) \longrightarrow^* v3'$
apply (*rule multi-step-trans*) **done**
from $6\ v3-v3$ **show** *?case* **by** *blast*

next

case (*eval-if1* $\varrho\ e1\ n\ e2\ v2\ e3\ \varrho'$)
from *eval-if1* **have** $FV\ e1 \subseteq \text{set}(\text{map}\ \text{fst}\ \varrho)$ **by** *auto*
from *this eval-if1* **obtain** $v1$ **where** $e1-v1: \text{psubst}\ \varrho'\ e1 \longrightarrow^* v1$ **and**
 $n1-v1: \text{bs-val}\ (BNat\ n)\ v1$ **and** $nz: n \neq 0$ **apply** *auto* **apply** *blast* **done**
from $n1-v1$ **have** $v1: v1 = ENat\ n$ **by** *blast*
from *eval-if1* **have** $FV\ e2 \subseteq \text{set}(\text{map}\ \text{fst}\ \varrho)$ **by** *auto*
from *this eval-if1* **obtain** $v2'$ **where** $e2-v2: \text{psubst}\ \varrho'\ e2 \longrightarrow^* v2'$ **and**
 $v2-v2: \text{bs-val}\ v2\ v2'$ **by** *blast*
from $e1-v1$ **have** $1: EIf\ (\text{psubst}\ \varrho'\ e1)\ (\text{psubst}\ \varrho'\ e2)\ (\text{psubst}\ \varrho'\ e3) \longrightarrow^* EIf\ v1\ (\text{psubst}\ \varrho'\ e2)\ (\text{psubst}\ \varrho'\ e3)$ **by** (*rule if-red-cong1*)
from $v1\ nz$ **have** $3: EIf\ v1\ (\text{psubst}\ \varrho'\ e2)\ (\text{psubst}\ \varrho'\ e3) \longrightarrow (\text{psubst}\ \varrho'\ e2)$ **by** *auto*
from $1\ 3$ **have** $5: \text{psubst}\ \varrho'\ (EIf\ e1\ e2\ e3) \longrightarrow^* \text{psubst}\ \varrho'\ e2$ **apply** *simp*
apply (*rule multi-step-trans*) **apply** *assumption* **apply** *blast* **done**
from $5\ e2-v2$ **have** $6: \text{psubst}\ \varrho'\ (EIf\ e1\ e2\ e3) \longrightarrow^* v2'$
by (*rule multi-step-trans*)
from $6\ v2-v2$ **show** *?case* **by** *blast*

qed

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

proof (*induction* e *arbitrary*: ϱ)

case (*EVar* x)

then **show** *?case* **by** (*cases* *lookup* $\varrho\ x$) (*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' 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 ]  $\implies$  v1  $\sqsubseteq$  v3
  by (induction v2 arbitrary: v1 v3) blast+

lemma fsubset[intro!]: fset A  $\subseteq$  fset B  $\implies$  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 using fmember-iff-member-fset[of x A] by simp
  from this ab have x  $\in$  fset B by blast
  from this show x | $\in$ | B using fmember-iff-member-fset[of x B] by simp
qed

proposition val-le-antisymm: fixes v1::val shows [ v1  $\sqsubseteq$  v2; v2  $\sqsubseteq$  v1 ]  $\implies$  v1 = v2
  by (induction v1 arbitrary: v2) auto

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

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

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

end

```

4 Declarative semantics as a relational semantics

```

theory RelationalSemFSet
  imports Lambda ValuesFSet
begin

inductive rel-sem :: env  $\Rightarrow$  exp  $\Rightarrow$  val  $\Rightarrow$  bool (-  $\vdash$  -  $\Rightarrow$  - [52,52,52] 51) where
  rnat[intro!]:  $\rho \vdash$  ENat n  $\Rightarrow$  VNat n |
  rprim[intro!]: [  $\rho \vdash$  e1  $\Rightarrow$  VNat n1;  $\rho \vdash$  e2  $\Rightarrow$  VNat n2 ]  $\implies$   $\rho \vdash$  EPrim f e1 e2  $\Rightarrow$  VNat (f n1 n2) |
  rvar[intro!]: [ lookup  $\rho$  x = Some v'; v  $\sqsubseteq$  v' ]  $\implies$   $\rho \vdash$  EVar x  $\Rightarrow$  v |
  rlam[intro!]: [  $\forall v v'. (v, v') \in$  fset t  $\longrightarrow$  (x,v)# $\rho \vdash$  e  $\Rightarrow$  v' ]
     $\implies$   $\rho \vdash$  ELam x e  $\Rightarrow$  VFun t |
  rapp[intro!]: [  $\rho \vdash$  e1  $\Rightarrow$  VFun t;  $\rho \vdash$  e2  $\Rightarrow$  v2; (v3,v3')  $\in$  fset t; v3  $\sqsubseteq$  v2; v  $\sqsubseteq$  v3' ]
     $\implies$   $\rho \vdash$  EApp e1 e2  $\Rightarrow$  v |
  rifnz[intro!]: [  $\rho \vdash$  e1  $\Rightarrow$  VNat n; n  $\neq$  0;  $\rho \vdash$  e2  $\Rightarrow$  v ]  $\implies$   $\rho \vdash$  EIf e1 e2 e3  $\Rightarrow$  v |
  rifz[intro!]: [  $\rho \vdash$  e1  $\Rightarrow$  VNat n; n = 0;  $\rho \vdash$  e3  $\Rightarrow$  v ]  $\implies$   $\rho \vdash$  EIf e1 e2 e3  $\Rightarrow$  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 } |
  Evar: E (EVar x)  $\varrho$  = { v.  $\exists$  v'. lookup  $\varrho$  x = Some v'  $\wedge$  v  $\sqsubseteq$  v' } |
  Elam: E (ELam x e)  $\varrho$  = { v.  $\exists$  f. v = VFun f  $\wedge$  ( $\forall$  v1 v2. (v1, v2)  $\in$  fset f
     $\longrightarrow$  v2  $\in$  E e ((x,v1)# $\varrho$ )) } |
  Eapp: E (EApp e1 e2)  $\varrho$  = { v3.  $\exists$  f v2 v2' v3'.
    VFun f  $\in$  E e1  $\varrho$   $\wedge$  v2  $\in$  E e2  $\varrho$   $\wedge$  (v2', v3')  $\in$  fset f  $\wedge$  v2'  $\sqsubseteq$  v2  $\wedge$  v3  $\sqsubseteq$  v3' } |
  Eprim: E (EPrim f e1 e2)  $\varrho$  = { v.  $\exists$  n1 n2. VNat n1  $\in$  E e1  $\varrho$ 
     $\wedge$  VNat n2  $\in$  E e2  $\varrho$   $\wedge$  v = VNat (f n1 n2) } |
  Eif: E (EIf e1 e2 e3)  $\varrho$  = { v.  $\exists$  n. VNat n  $\in$  E e1  $\varrho$ 
     $\wedge$  (n = 0  $\longrightarrow$  v  $\in$  E e3  $\varrho$ )  $\wedge$  (n  $\neq$  0  $\longrightarrow$  v  $\in$  E e2  $\varrho$ ) }

```

```

end

```

6 Relational and denotational views are equivalent

```

theory EquivRelationalDenotFSet
imports RelationalSemFSet DeclSemAsDenotFSet
begin

```

```

lemma denot-implies-rel: (v  $\in$  E e  $\varrho$ )  $\Longrightarrow$  ( $\varrho \vdash e \Rightarrow v$ )
proof (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  $\varrho$  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]:  $\llbracket \text{VNat } n1 \in E e1 \varrho; \text{VNat } n2 \in E e2 \varrho; v = \text{VNat } (f n1 n2) \rrbracket$ 
 $\Longrightarrow v \in E (EPrim f e1 e2) \varrho$  by auto

```

```

lemma e-prim-elim[elim]:  $\llbracket v \in E (EPrim f e1 e2) \varrho;$ 
 $\wedge n1 n2. \llbracket \text{VNat } n1 \in E e1 \varrho; \text{VNat } n2 \in E e2 \varrho; v = \text{VNat } (f n1 n2) \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ 
by auto

```

```

lemma e-app-elim[elim]:  $\llbracket v3 \in E (EApp e1 e2) \varrho;$ 
 $\wedge f v2 v2' v3'. \llbracket \text{VFun } f \in E e1 \varrho; v2 \in E e2 \varrho; (v2', v3') \in \text{fset } f; v2' \sqsubseteq v2; v3 \sqsubseteq v3' \rrbracket \Longrightarrow P$ 
 $\rrbracket \Longrightarrow P$ 
by auto

```

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

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

lemma *e-lam-intro2*[*intro*]:
 $\llbracket VFun\ f \in E\ (ELam\ x\ e)\ \varrho; v2 \in E\ e\ ((x,v1)\#\varrho) \rrbracket$
 $\implies VFun\ (finsert\ (v1,v2)\ f) \in E\ (ELam\ x\ e)\ \varrho$
by *auto*

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

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

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

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

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

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

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

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

proposition *change-env-le*: **fixes** *v::val* **and** *ϱ::env*

assumes *de*: $v \in E\ e\ \varrho$ **and** *vp-v*: $v' \sqsubseteq v$ **and** *rr*: $FV\ e \vdash \varrho \sqsubseteq \varrho'$

shows $v' \in E\ e\ \varrho'$

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** $v = VFun\ f$ **and**

body: $\forall v1\ v2. (v1, v2) \in \text{fset } f \longrightarrow v2 \in E\ e\ ((x, v1)\#\varrho)$ **by** *auto*
from $v\ \text{ELam}(4)$ **obtain** f' **where** $vp: v' = \text{VFun } f'$ **and** $fp\text{-}f: \text{fset } f' \sqsubseteq \text{fset } f$
by (*case-tac v'*) *auto*
from vp **show** $?case$
proof (*simp, clarify*)
fix $v1\ v2$ **assume** $v12: (v1, v2) \in \text{fset } f'$
from $v12\ fp\text{-}f$ **have** $v34: (v1, v2) \in \text{fset } f$ **by** *blast*
from $v34\ \text{body}$ **have** $v4\text{-}E: v2 \in E\ e\ ((x, v1)\#\varrho)$ **by** *blast*
from $\text{ELam}(3)$ **have** $rr2: FV\ e \vdash ((x, v1)\#\varrho) \sqsubseteq ((x, v1)\#\varrho')$ **by** *auto*
from $\text{ELam}(1)\ v4\text{-}E\ rr2$ **show** $v2 \in E\ e\ ((x, v1)\#\varrho')$ **by** *auto*
qed
next
case ($\text{EApp } e1\ e2$)
from $\text{EApp}(3)$ **obtain** f **and** $v2::val$ **and** $v2'\ v3'$ **where**
 $f\text{-}e1: \text{VFun } f \in E\ e1\ \varrho$ **and** $v2\text{-}e2: v2 \in E\ e2\ \varrho$ **and**
 $v23p\text{-}f: (v2', v3') \in \text{fset } f$ **and** $v2p\text{-}v2: v2' \sqsubseteq v2$ **and** $v\text{-}v3: v \sqsubseteq v3'$ **by** *blast*
from $\text{EApp}(4)$ **have** $1: FV\ e1 \vdash \varrho \sqsubseteq \varrho'$ **by** *auto*
have $f\text{-}f: \text{VFun } f \sqsubseteq \text{VFun } f$ **by** *auto*
from $\text{EApp}(1)\ f\text{-}e1\ 1\ f\text{-}f$ **have** $f\text{-}e1b: \text{VFun } f \in E\ e1\ \varrho'$ **by** *blast*
from $\text{EApp}(4)$ **have** $2: FV\ e2 \vdash \varrho \sqsubseteq \varrho'$ **by** *auto*
from $\text{EApp}(2)\ v2\text{-}e2\ 2$ **have** $v2\text{-}e2b: v2 \in E\ e2\ \varrho'$ **by** *auto*
from $\text{EApp}(5)\ v\text{-}v3$ **have** $vp\text{-}v3p: v' \sqsubseteq v3'$ **by** (*rule val-le-trans*)
from $f\text{-}e1b\ v2\text{-}e2b\ v23p\text{-}f\ v2p\text{-}v2\ vp\text{-}v3p$
show $?case$ **by** *auto*
next
case ($\text{EPrim } f\ e1\ e2$)
from $\text{EPrim}(3)$ **obtain** $n1\ n2$ **where** $n1\text{-}e1: \text{VNat } n1 \in E\ e1\ \varrho$ **and**
 $n2\text{-}e2: \text{VNat } n2 \in E\ e2\ \varrho$ **and** $v: v = \text{VNat } (f\ n1\ n2)$ **by** *blast*
from $\text{EPrim}(4)$ **have** $1: FV\ e1 \vdash \varrho \sqsubseteq \varrho'$ **by** *auto*
from $\text{EPrim}(1)\ n1\text{-}e1\ 1$ **have** $n1\text{-}e1b: \text{VNat } n1 \in E\ e1\ \varrho'$ **by** *blast*
from $\text{EPrim}(4)$ **have** $2: FV\ e2 \vdash \varrho \sqsubseteq \varrho'$ **by** *auto*
from $\text{EPrim}(2)\ n2\text{-}e2\ 2$ **have** $n2\text{-}e2b: \text{VNat } n2 \in E\ e2\ \varrho'$ **by** *blast*
from $v\ \text{EPrim}(5)$ **have** $vp: v' = \text{VNat } (f\ n1\ n2)$ **by** *auto*
from $n1\text{-}e1b\ n2\text{-}e2b\ vp$ **show** $?case$ **by** *auto*
next
case ($\text{EIf } e1\ e2\ e3$)
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*: $\llbracket v \in E\ e\ \varrho; v' \sqsubseteq v \rrbracket \Longrightarrow v' \in E\ e\ \varrho$

apply (*subgoal-tac FV e \vdash \varrho \sqsubseteq \varrho*) **using** *change-env-le* **apply** *blast* **apply** *auto* **done**

lemma *env-le-ext*: **fixes** $\varrho::env$ **assumes** $rr: \varrho \sqsubseteq \varrho'$ **shows** $((x, v)\#\varrho) \sqsubseteq ((x, v)\#\varrho')$
using rr **apply** (*simp add: env-le-def*) **done**

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

proof —

have $vv: v \sqsubseteq v$ **by** *auto*

from $de\ rr\ vv$ **show** $?thesis$ **using** *change-env-le* **by** *blast*

qed

lemma *raise-env*: **fixes** $\varrho::env$ **assumes** $de: v \in E\ e\ \varrho$ **and** $rr: \varrho \sqsubseteq \varrho'$ **shows** $v \in E\ e\ \varrho'$
using $de\ rr$ *change-env env-le-def* **by** *auto*

lemma *env-eq-refl[*simp*]*: **fixes** $\varrho::env$ **shows** $\varrho \approx \varrho$ **by** (*simp add: env-eq-def*)

lemma *env-eq-ext*: **fixes** $\varrho::env$ **assumes** $rr: \varrho \approx \varrho'$ **shows** $((x, v)\#\varrho) \approx ((x, v)\#\varrho')$
using rr **by** (*simp add: env-eq-def*)

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

lemma *env-swap*: **fixes** $\varrho::env$ **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*: $\llbracket v \in E e \varrho; \forall x. x \in FV e \implies lookup \varrho' x = lookup \varrho x \rrbracket \implies v \in E e \varrho'$
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* :: $'a M \Rightarrow ('a \Rightarrow 'b M) \Rightarrow 'b M$ **where**
 $set\ bind\ m\ f \equiv \{ v. \exists v'. v' \in m \wedge v \in f\ v' \}$
declare *set-bind-def*[*simp*]

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

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

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

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

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

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

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

8.2 Non-deterministic interpreter

abbreviation *apply-fun* :: $val M \Rightarrow val M \Rightarrow val M$ **where**
 $apply\ fun\ V1\ V2 \equiv (v1 \leftarrow V1; v2 \leftarrow V2;$
 $\text{case } v1 \text{ of } VFun\ f \Rightarrow$
 $(v2', v3') \leftarrow choose\ (fset\ f);$
 $\text{if } v2' \sqsubseteq v2 \text{ then } return\ v3' \text{ else } zero$
 $| - \Rightarrow zero)$

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

Enat2: $E (ENat\ n)\ \varrho = \text{return } (VNat\ n) \mid$
Evar2: $E (EVar\ x)\ \varrho = (\text{case lookup } \varrho\ x\ \text{of None} \Rightarrow \text{zero} \mid \text{Some } v \Rightarrow \text{down } v) \mid$
Elam2: $E (ELam\ x\ e)\ \varrho = (vs \leftarrow \text{choose UNIV};$
 $t \leftarrow \text{mapM } vs\ (\lambda v. (v' \leftarrow E\ e\ ((x,v)\#\varrho); \text{return } (v, v')));$
 $\text{return } (VFun\ t)) \mid$
Eapp2: $E (EApp\ e1\ e2)\ \varrho = \text{apply-fun } (E\ e1\ \varrho)\ (E\ e2\ \varrho) \mid$
Eprim2: $E (EPrim\ f\ e1\ e2)\ \varrho = (v_1 \leftarrow E\ e1\ \varrho; v_2 \leftarrow E\ e2\ \varrho;$
 $\text{case } (v_1, v_2)\ \text{of}$
 $(VNat\ n_1, VNat\ n_2) \Rightarrow \text{return } (VNat\ (f\ n_1\ n_2))$
 $\mid (VNat\ n_1, VFun\ t_2) \Rightarrow \text{zero}$
 $\mid (VFun\ t_1, v_2) \Rightarrow \text{zero}) \mid$
Eif2[*eta-contract = false*]: $E (EIf\ e1\ e2\ e3)\ \varrho = (v_1 \leftarrow E\ e1\ \varrho;$
 $\text{case } v_1\ \text{of}$
 $(VNat\ n) \Rightarrow \text{if } n \neq 0\ \text{then } E\ e2\ \varrho\ \text{else } E\ e3\ \varrho$
 $\mid (VFun\ t) \Rightarrow \text{zero})$

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 *Top* (\top)

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

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

sub-refl: $A <: A \mid$

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

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

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

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

sub-inter-r[*intro!*]: $\llbracket T3 <:: T1; T3 <:: T2 \rrbracket \Longrightarrow T3 <:: T1 \sqcap T2 \mid$

sub-fun-top[*intro!*]: $T1 \rightarrow T2 <:: \top \mid$

sub-top-top[*intro!*]: $\top <:: \top \mid$

fsub-refl[*intro!*]: $T <:: T \mid$

sub-trans[*trans*]: $\llbracket T1 <:: T2; T2 <:: T3 \rrbracket \Longrightarrow T1 <:: T3$

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!*]: $\text{lookup } \Gamma\ x = \text{Some } T \Longrightarrow \Gamma \vdash EVar\ x : T \mid$

wt-nat[*intro!*]: $\Gamma \vdash ENat\ n : TNat\ n \mid$

wt-lam[*intro!*]: $\llbracket (x, A)\#\Gamma \vdash e : B \rrbracket \Longrightarrow \Gamma \vdash ELam\ x\ e : TFun\ (A \rightarrow B) \mid$

wt-app[*intro!*]: $\llbracket \Gamma \vdash e1 : TFun\ (A \rightarrow B); \Gamma \vdash e2 : A \rrbracket \Longrightarrow \Gamma \vdash EApp\ e1\ e2 : B \mid$

wt-top[*intro!*]: $\Gamma \vdash ELam\ x\ e : TFun\ \top \mid$

wt-inter[*intro!*]: $\llbracket \Gamma \vdash ELam\ x\ e : TFun\ A; \Gamma \vdash ELam\ x\ e : TFun\ B \rrbracket$

$\Longrightarrow \Gamma \vdash ELam\ x\ e : TFun\ (A \sqcap B) \mid$

wt-sub[*intro!*]: $\llbracket \Gamma \vdash e : A; A <: B \rrbracket \Longrightarrow \Gamma \vdash e : B \mid$

wt-prim[*intro!*]: $\llbracket \Gamma \vdash e1 : TNat\ n1; \Gamma \vdash e2 : TNat\ n2 \rrbracket$

$\Longrightarrow \Gamma \vdash EPrim\ f\ e1\ e2 : TNat\ (f\ n1\ n2) \mid$

wt-ifz[*intro!*]: $\llbracket \Gamma \vdash e1 : TNat\ 0; \Gamma \vdash e3 : B \rrbracket$

$\Longrightarrow \Gamma \vdash EIf\ e1\ e2\ e3 : B \mid$

wt-ifnz[intro!]: $\llbracket \Gamma \vdash e1 : \text{TNat } n; n \neq 0; \Gamma \vdash e2 : B \rrbracket$
 $\implies \Gamma \vdash \text{EIf } e1 \ e2 \ e3 : B$

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** \sqsubseteq 52)
and *fun-le* :: *func* ⇒ *func* ⇒ *bool* (**infix** \lesssim 52) **where**
vnat-le[intro!]: (*VNat* *n*) \sqsubseteq (*VNat* *n*) |
vfun-le[intro!]: *t1* \lesssim *t2* \implies (*VFun* *t1*) \sqsubseteq (*VFun* *t2*) |
fun-le[intro!]: ($\forall v1 \ v2. (v1, v2) \in \text{set } t1 \longrightarrow$
 $(\exists v3 \ v4. (v3, v4) \in \text{set } t2$
 $\wedge v1 \sqsubseteq v3 \wedge v3 \sqsubseteq v1 \wedge v2 \sqsubseteq v4 \wedge v4 \sqsubseteq v2))$
 $\implies t1 \lesssim t2$

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

definition *env-le* :: *env* ⇒ *env* ⇒ *bool* (**infix** \sqsubseteq 52) **where**
 $\rho \sqsubseteq \rho' \equiv \forall x \ v. \text{lookup } \rho \ x = \text{Some } v \longrightarrow (\exists v'. \text{lookup } \rho' \ x = \text{Some } v' \wedge v \sqsubseteq v')$

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

end

10.2 Properties about values

theory *ValueProps*

imports *Values*

begin

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

lemma *fun-le-cons*: (*a* $\#$ *t1*) \lesssim *t2* \implies *t1* \lesssim *t2*
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

```

$fun\text{-}eq\ t1\ t2 \equiv (t1 \lesssim t2 \wedge t2 \lesssim t1)$

lemma *vfundef*[intro!]: $t \sim t' \implies VFun\ t \sim VFun\ t'$
apply (*simp* *add*: *val-eq-def* *fun-eq-def*)
apply (*rule* *conjI*) **apply** (*erule* *conjE*) **apply** (*rule* *vfundef*) **apply** *assumption*
apply (*erule* *conjE*) **apply** (*rule* *vfundef*) **apply** *assumption*
done

lemma *val-eq-refl*[*simp*]: **fixes** $v::val$ **shows** $v \sim v$
by (*simp* *add*: *val-eq-def*)

lemma *val-eq-symm*: **fixes** $v1::val$ **and** $v2::val$ **shows** $v1 \sim v2 \implies v2 \sim v1$
unfolding *val-eq-def* **by** *blast*

lemma *val-le-fundef-trans*:

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

proof (*induction* n *rule*: *nat-less-induct*)

case ($1\ n$)

show *?case* **apply** *clarify*

proof

fix $v2\ t2$ **assume** $n: n = val\text{-}size\ v2 + fun\text{-}size\ t2$

show $\forall v1\ v3. v1 \sqsubseteq v2 \longrightarrow v2 \sqsubseteq v3 \longrightarrow v1 \sqsubseteq v3$ **apply** *clarify*

proof $-$

fix $v1\ v3$ **assume** $v12: v1 \sqsubseteq v2$ **and** $v23: v2 \sqsubseteq v3$

show $v1 \sqsubseteq v3$

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** $t12: t1 \lesssim t2'$ **and** $v1: v1 = VFun\ t1$ **by** *auto*

from $v23\ VFun$ **obtain** $t3$ **where** $t23: t2' \lesssim t3$ **and** $v3: v3 = VFun\ t3$ **by** *auto*

let $?m = val\text{-}size\ (VNat\ 0) + fun\text{-}size\ t2'$

from $1\ n\ VFun$ **have** $IH: \forall t1\ t3. t1 \lesssim t2' \longrightarrow t2' \lesssim t3 \longrightarrow t1 \lesssim t3$

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** $t1 \lesssim t3$ **by** *auto*

from *this* $v1\ v3$ **show** *?thesis* **apply** *auto* **done**

qed

qed

next

fix $v5\ t2$ **assume** $n: n = val\text{-}size\ v5 + fun\text{-}size\ t2$

show $\forall t1\ t3. t1 \lesssim t2 \longrightarrow t2 \lesssim t3 \longrightarrow t1 \lesssim t3$ **apply** *clarify*

proof $-$

fix $t1\ t3\ v1\ v2$ **assume** $t12: t1 \lesssim t2$ **and** $t23: t2 \lesssim t3$ **and** $v12: (v1, v2) \in set\ t1$

from $v12\ t12$ **obtain** $v1'\ v2'$ **where** $v12p: (v1', v2') \in set\ t2$ **and**

$v1\text{-}v1p: v1 \sqsubseteq v1'$ **and** $v11p: v1' \sqsubseteq v1$ **and** $v22p: v2 \sqsubseteq v2'$ **and** $v2p\text{-}v2: v2' \sqsubseteq v2$ **by** *blast*

from $v12p\ t23$ **obtain** $v1''\ v2''$ **where** $v12pp: (v1'', v2'') \in set\ t3$ **and**

$v1p\text{-}v1pp: v1' \sqsubseteq v1''$ **and** $v11pp: v1'' \sqsubseteq v1'$ **and**

$v22pp: v2' \sqsubseteq v2''$ **and** $v2pp\text{-}v2p: v2'' \sqsubseteq v2'$ **by** *blast*

from $v12p$ **have** $sv1p: val\text{-}size\ v1' < fun\text{-}size\ t2$ **using** *val-size-mem-l* **by** *blast*

from $v12\ 1\ v11p\ v11pp\ n\ sv1p$ **have** $v1pp\text{-}v1: v1'' \sqsubseteq v1$

apply (*erule-tac* $x=val\text{-}size\ v1' + fun\text{-}size\ []$ **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*: $v2 \sqsubseteq v2''$
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*: $v1 \sqsubseteq v1''$
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*: $v2'' \sqsubseteq v2$
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 $\exists v3 v4. (v3, v4) \in \text{set } t3 \wedge v1 \sqsubseteq v3 \wedge v3 \sqsubseteq v1 \wedge v2 \sqsubseteq v4 \wedge v4 \sqsubseteq v2$ **by blast**
qed
qed
qed

proposition *val-le-trans*: **fixes** *v2::val* **shows** $[v1 \sqsubseteq v2; v2 \sqsubseteq v3] \implies v1 \sqsubseteq v3$
using *val-le-fun-le-trans* **by blast**

lemma *fun-le-trans*: $[t1 \lesssim t2; t2 \lesssim t3] \implies t1 \lesssim t3$
using *val-le-fun-le-trans* **by blast**

lemma *val-eq-trans*: **fixes** *v1::val* **and** *v2::val* **and** *v3::val*
assumes *v12*: $v1 \sim v2$ **and** *v23*: $v2 \sim v3$ **shows** $v1 \sim v3$
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** $t \sim t$
by (*simp add: fun-eq-def*)

lemma *fun-eq-trans*: **fixes** *t1::func* **and** *t2::func* **and** *t3::func*
assumes *t12*: $t1 \sim t2$ **and** *t23*: $t2 \sim t3$ **shows** $t1 \sim t3$
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*:
 $[t1' \lesssim t1; t2' \lesssim t2] \implies t1' @ t2' \lesssim t1 @ t2$
apply (*rule fun-le*) **apply** *clarify* **apply** *simp* **apply** (*erule fun-le-inv*) **apply blast done**

lemma *append-fun-equiv*:
 $[t1' \sim t1; t2' \sim t2] \implies t1' @ t2' \sim t1 @ t2$
apply (*simp add: val-eq-def fun-eq-def*) **using** *append-fun-le* **apply blast done**

lemma *append-leq-symm*: $t2 @ t1 \lesssim t1 @ t2$
apply (*rule fun-le*) **apply force done**

lemma *append-eq-symm*: $t2 @ t1 \sim t1 @ t2$
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]*: $VNat\ n \sqsubseteq v \implies v = VNat\ n$
by (*cases v*) *auto*

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

lemma *le-nat-nat[simp]*: $VNat\ n \sqsubseteq VNat\ n' \implies n = n'$
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]*: $\llbracket VNat\ n1 \in E\ e1\ \varrho; VNat\ n2 \in E\ e2\ \varrho; v = VNat\ (f\ n1\ n2) \rrbracket$
 $\implies v \in E\ (EPrim\ f\ e1\ e2)\ \varrho$ **by** *auto*

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

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

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

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

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

by auto

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

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

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

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

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

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

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

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

proposition *change-env-le*: **fixes** $v::val$ **and** $\varrho::env$

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: $v\ v'\ \varrho\ \varrho'$ rule: *exp.induct*)

case $(EVar\ x\ v\ v'\ \varrho\ \varrho')$

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*: $\llbracket v \in E e \rho; v' \subseteq v \rrbracket \implies v' \in E e \rho$

apply (subgoal-tac FV e ⊢ ρ ⊆ ρ) using change-env-le apply blast apply auto done

lemma *env-le-ext*: fixes $\rho::env$ assumes $rr: \rho \subseteq \rho'$ shows $((x,v)\#\rho) \subseteq ((x,v)\#\rho')$
using *rr* by (simp add: env-le-def)

lemma *change-env*: fixes $\rho::env$ assumes $de: v \in E e \rho$ and $rr: FV e \vdash \rho \subseteq \rho'$ shows $v \in E e \rho'$
proof —

have *vv*: $v \subseteq v$ by auto

from *de rr vv* show ?thesis using change-env-le by blast

qed

lemma *raise-env*: fixes $\rho::env$ assumes $de: v \in E e \rho$ and $rr: \rho \subseteq \rho'$ shows $v \in E e \rho'$
using *de rr* change-env env-le-def by auto

lemma *env-eq-refl*[*simp*]: fixes $\rho::env$ shows $\rho \approx \rho$ by (simp add: env-eq-def)

lemma *env-eq-ext*: fixes $\rho::env$ assumes $rr: \rho \approx \rho'$ shows $((x,v)\#\rho) \approx ((x,v)\#\rho')$
using *rr* by (simp add: env-eq-def)

lemma *eq-implies-le*: fixes $\rho::env$ shows $\rho \approx \rho' \implies \rho \subseteq \rho'$
by (simp add: env-le-def env-eq-def)

lemma *env-swap*: fixes $\rho::env$ assumes $rr: \rho \approx \rho'$ and $ve: v \in E e \rho$ shows $v \in E e \rho'$
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*: $\llbracket v \in E e \rho; \forall x. x \in FV e \longrightarrow \text{lookup } \rho' x = \text{lookup } \rho x \rrbracket \implies v \in E e \rho'$

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**

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

fun $Venv :: tyenv \Rightarrow env$ **where**

$Venv\ [] = [] \mid$
 $Venv\ ((x,A)\#\Gamma) = (x, V\ A)\#Venv\ \Gamma$

function $T :: val \Rightarrow ty$ **and** $Tf :: (val \times val)$ *list* $\Rightarrow funty$ **where**

$T\ (VNat\ n) = TNat\ n \mid$
 $T\ (VFun\ t) = TFun\ (Tf\ t) \mid$
 $Tf\ [] = \top \mid$
 $Tf\ ((v1,v2)\#t) = (T\ v1 \rightarrow T\ v2) \sqcap Tf\ t$
by *pat-completeness auto*

termination T **by** *size-change*

fun $Tenv :: env \Rightarrow tyenv$ **where**

$Tenv\ [] = [] \mid$
 $Tenv\ ((x,v)\#\varrho) = (x, T\ v)\#Tenv\ \varrho$

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*:

$(v, v') \in set\ t \Longrightarrow Tf\ t <:: (T\ v) \rightarrow (T\ v')$

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*: $v \sqsubseteq v' \implies T v' <: T v$ **using** *le-sub-flip-aux* **by** *blast*

lemma *le-sub-fun-flip*: $t \lesssim t' \implies Tf t' <:: Tf t$ **using** *le-sub-flip-aux* **by** *blast*

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

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 \sqcap 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 \sqcap 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*: $Tf t1 \sqcap Tf t2 <:: Tf (t1 @ t2)$

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 \to T v2) \sqcap Tf t1) \sqcap Tf t2 <:: Tf t1 \sqcap 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** $T (V A) \approx A$ **and** $Tf (Vf F) \simeq F$

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)  $\sqcap$  Tf (Vf B))
  prefer 2 using Tf-append apply simp
  apply (subgoal-tac Tf (Vf A)  $\sqcap$  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)  $\sqcap$  Tf (Vf B))
  prefer 2 using Tf-append apply simp
  apply (subgoal-tac Tf (Vf A)  $\sqcap$  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)  $\sqcap$  Tf (Vf B) <:: Tf (Vf A @ Vf B))
  prefer 2 apply (rule append-Tf)
  apply (subgoal-tac A  $\sqcap$  B <:: Tf (Vf A)  $\sqcap$  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:  $\forall v \varrho. v \in E e \varrho \longrightarrow Tenv \varrho \vdash e : T v$  and
    fe:  $\forall v1 v2. (v1, v2) \in set f \longrightarrow v2 \in E e ((x, v1) \# \varrho)$ 
  shows  $Tenv \varrho \vdash ELam x e : TFun (Tf f)$ 
  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:  $Tenv \varrho \vdash ELam x e : TFun (Tf f)$  and
    2:  $\forall v \varrho. v \in E e \varrho \longrightarrow Tenv \varrho \vdash e : T v$  and
    3:  $\forall v1 v2. (v1 = v \wedge v2 = v' \longrightarrow v' \in E e ((x, v) \# \varrho)) \wedge$ 
       $((v1, v2) \in set f \longrightarrow v2 \in E e ((x, v1) \# \varrho))$ 
  from 3 have 4:  $v' \in E e ((x, v) \# \varrho)$  by simp
  from 2 4 have 5:  $Tenv ((x, v) \# \varrho) \vdash e : T v'$ 
  apply (erule-tac  $x=v'$  in allE) apply (erule-tac  $x=(x, v) \# \varrho$  in allE) apply simp done
  from 5 have  $(x, T v) \# Tenv \varrho \vdash e : T v'$  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:  $v \in E e \varrho$  shows  $Tenv \varrho \vdash e : T v$ 
  using ve
proof (induction e arbitrary: v  $\varrho$ )
  case (EVar x)
  then show ?case
  apply simp apply (erule exE) apply (erule conjE)
  apply (subgoal-tac lookup (Tenv  $\varrho$ ) 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 T 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 $x = \text{Some } A$ **shows** *lookup* (*Venv* Γ) $x = \text{Some } (V A)$
using *lx*

proof (*induction* Γ *arbitrary*: A)

case *Nil*

then show ?case by auto

next

case (*Cons* $b \Gamma$)

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*: $\llbracket t1' \sim t1; t2' \sim t2 \rrbracket \implies t1' @ t2' \sim t1 @ t2$
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  $\Gamma::env$  and  $A::val$ 
  assumes  $wte: B \in E e \Gamma'$  and  $wtv: A \in E v []$ 
    and  $gp: \Gamma' \approx (x,A)\#\Gamma$  and  $v: is-val v$ 
  shows  $B \in E (subst x v e) \Gamma$ 
  using  $wte wtv gp v$ 
proof (induction arbitrary:  $v A B \Gamma x$  rule:  $E.induct$ )
  case (1 n  $\varrho$ )
  then show ?case by auto
next
  case (2 x  $\varrho v A B \Gamma x'$ )
  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  $\varrho v A B \Gamma x'$ )
  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  $\varrho$ )
  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  $\varrho$ )
  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  $\varrho$ )
  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: $v \varrho$ rule: *reduce.induct*)

case (beta $v x e v' \varrho$)

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 $v3' \in E (subst x v e) \varrho$) prefer 2 apply (rule *substitution*)

apply (subgoal-tac $v3' \in E e ((x,v2)\#\varrho)$) 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 $(x,v2)\#\varrho \approx (x,v2)\#\varrho$) 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: ϱv) 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 $e1 e2 \varrho$)

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 $f e1 e2 \varrho$)

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

primrec *good* :: $val \Rightarrow bval\ set$ **where**
Gnat: $good\ (VNat\ n) = \{ BNat\ n \}$ |
Gfun: $good\ (VFun\ f) = \{ vc. \exists x\ e\ \rho. vc = BClos\ x\ e\ \rho \wedge (\text{ffold}\ (good\text{-entry}\ x\ e\ \rho)\ True\ (fimage\ (map\text{-prod}\ (\lambda v. (v,good\ v))\ (\lambda v. (v,good\ v))))\ f) \}$

inductive *good-env* :: $benv \Rightarrow env \Rightarrow bool$ **where**
genv-nil[intro!]: $good\text{-env}\ []\ []\ |$
genv-cons[intro!]: $[[v \in good\ v'; good\text{-env}\ \rho\ \rho'] \Longrightarrow good\text{-env}\ ((x,v)\#_{\rho})\ ((x,v')\#_{\rho'})]$

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

lemma *lookup-good*:
assumes l : $lookup\ \rho'\ x = Some\ A$ **and** EE : $good\text{-env}\ \rho\ \rho'$
shows $\exists v. lookup\ \rho\ x = Some\ v \wedge v \in good\ A$
using $l\ EE$
proof (*induction* ρ' *arbitrary*: $x\ A\ \rho$)
case *Nil*
show ?case **apply** (rule *genv-any-nil-inv*) **using** *Nil* **by** *auto*
next
case (*Cons* $a\ \rho'$)
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* :: $val \times val \Rightarrow (val \times bval\ set) \times (val \times bval\ set)$ **where**
good-prod $\equiv map\text{-prod}\ (\lambda v. (v,good\ v))\ (\lambda v. (v,good\ v))$

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

unfolding inj-on-def apply auto done

definition *good-fun* :: *func* \Rightarrow *name* \Rightarrow *exp* \Rightarrow *benv* \Rightarrow *bool* **where**
good-fun *f* *x* *e* $\varrho \equiv$ (*ffold* (*good-entry* *x* *e* ϱ) *True* (*fimage* *good-prod* *f*))

lemma *good-fun-def2*:

good-fun *f* *x* *e* $\varrho =$ *ffold* (*good-entry* *x* *e* ϱ \circ *good-prod*) *True* *f*

proof –

interpret *ge*: *comp-fun-commute* (*good-entry* *x* *e* ϱ) \circ *good-prod*

unfolding *comp-fun-commute-def* **by** *auto*

show *good-fun* *f* *x* *e* ϱ

$=$ *ffold* ((*good-entry* *x* *e* ϱ) \circ *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*: $w \in \text{good } (V\text{Fun } f) \implies \exists x e \varrho. w = \text{BClos } x e \varrho \wedge \text{good-fun } f x e \varrho$

using *good-fun-def* **by** *auto*

lemma *gfun-mem-iff*: *good-fun* *f* *x* *e* $\varrho = (\forall v1 v2. (v1, v2) \in \text{fset } f \longrightarrow$

$(\forall v \in \text{good } v1. \exists v'. (x, v) \# \varrho \vdash e \Downarrow v' \wedge v' \in \text{good } v2))$

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*

next

case (*insert* *p* *f*)

interpret *ge*: *comp-fun-commute* (*good-entry* *x* *e* ϱ) \circ *good-prod*

unfolding *comp-fun-commute-def* **by** *auto*

have *good-fun* (*insert* *p* *f*) *x* *e* ϱ

$=$ *ffold* ((*good-entry* *x* *e* ϱ) \circ *good-prod*) *True* (*insert* *p* *f*) **by** (*simp* *add*: *good-fun-def2*)

also from *insert(1)* **have** ... $=$ ((*good-entry* *x* *e* ϱ) \circ *good-prod*) *p*

(*ffold* ((*good-entry* *x* *e* ϱ) \circ *good-prod*) *True* *f*) **by** *simp*

finally have 1: *good-fun* (*insert* *p* *f*) *x* *e* ϱ

$=$ ((*good-entry* *x* *e* ϱ) \circ *good-prod*) *p* (*ffold* ((*good-entry* *x* *e* ϱ) \circ *good-prod*) *True* *f*) .

show *?case*

proof

assume 2: *good-fun* (*insert* *p* *f*) *x* *e* ϱ

show $\forall v1 v2. (v1, v2) \in \text{fset } (\text{insert } p f) \longrightarrow$

$(\forall v \in \text{good } v1. \exists v'. (x, v) \# \varrho \vdash e \Downarrow v' \wedge v' \in \text{good } v2)$

proof *clarify*

fix *v1* *v2* *v* **assume** 3: $(v1, v2) \in \text{fset } (\text{insert } p f)$ **and** 4: $v \in \text{good } v1$

from 3 **have** $(v1, v2) = p \vee (v1, v2) \in \text{fset } f$ **by** *auto*

from this **show** $\exists v'. (x, v) \# \varrho \vdash e \Downarrow v' \wedge v' \in \text{good } v2$

proof

assume *v12-p*: $(v1, v2) = p$

from 1 *v12-p*[*THEN* *sym*] 2 4 **show** *?thesis* **by** *simp*

next

assume *v12-f*: $(v1, v2) \in \text{fset } f$

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: $\forall v1 v2. (v1, v2) \in \text{fset } (\text{insert } p f) \longrightarrow$

$(\forall v \in \text{good } v1. \exists v'. (x, v) \# \varrho \vdash e \Downarrow v' \wedge v' \in \text{good } v2)$

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 $(3\ x\ e\ \varrho\ v'\ \varrho')$

have 1: $\varrho' \vdash ELam\ x\ e \Downarrow BClos\ x\ e\ \varrho'$ by auto

have 2: $BClos\ x\ e\ \varrho' \in good\ v'$

proof —

from 3(2) obtain t where $vp: v' = VFun\ t$ and

body: $\forall v1\ v2. (v1, v2) \in fset\ t \longrightarrow v2 \in E\ e\ ((x, v1) \# \varrho)$ by blast

have $gt: good\text{-}fun\ t\ x\ e\ \varrho'$ apply (rule $gfun\text{-}intro$) apply clarify

proof —

fix $v1\ v2\ w1$ assume $v12\text{-}t: (v1, v2) \in fset\ t$ and $w1\text{-}v1: w1 \in good\ v1$

from $v12\text{-}t$ body have $v2\text{-}Ee: v2 \in E\ e\ ((x, v1) \# \varrho)$ by blast

from 3(3) $w1\text{-}v1$ have $ge: good\text{-}env\ ((x, w1) \# \varrho')\ ((x, v1) \# \varrho)$ by auto

from $v12\text{-}t\ v2\text{-}Ee\ ge$ 3(1)[of $v1\ v2\ t\ v2$]

show $\exists v'. (x, w1) \# \varrho' \vdash e \Downarrow v' \wedge v' \in good\ v2$ by blast

qed

from $vp\ gt$ show ?thesis unfolding $good\text{-}fun\text{-}def$ by simp

qed

from 1 2 show ?case by blast

next — EApp

case $(4\ e1\ e2\ \varrho\ v'\ \varrho')$

from 4(3) show ?case

proof

fix $t\ v2$ and $v2': val$ and $v3'$ assume $t\text{-}Ee1: VFun\ t \in E\ e1\ \varrho$ and $v2\text{-}Ee2: v2 \in E\ e2\ \varrho$ and

$v23\text{-}t: (v2', v3') \in fset\ t$ and $v2p\text{-}v2: v2' \sqsubseteq v2$ and $vp\text{-}v3p: v' \sqsubseteq v3'$

from 4(1) $t\text{-}Ee1$ 4(4) obtain $w1$ where $e1\text{-}w1: \varrho' \vdash e1 \Downarrow w1$ and

$w1\text{-}t: w1 \in good\ (VFun\ t)$ by blast

from 4(2) $v2\text{-}Ee2$ 4(4) obtain $w2$ where $e2\text{-}w2: \varrho' \vdash e2 \Downarrow w2$ and $w2\text{-}v2: w2 \in good\ v2$ by blast

from $w1\text{-}t$ obtain $x\ e\ \varrho1$ where $w1: w1 = BClos\ x\ e\ \varrho1$ and $gt: good\text{-}fun\ t\ x\ e\ \varrho1$

by (auto simp: $good\text{-}fun\text{-}def$)

from $w2\text{-}v2\ v2p\text{-}v2$ have $w2\text{-}v2p: w2 \in good\ v2'$ by (rule $sub\text{-}good$)

from $v23\text{-}t\ gt\ w2\text{-}v2p$ obtain $w3$ where $e\text{-}w3: (x, w2) \# \varrho1 \vdash e \Downarrow w3$ and $w3\text{-}v3p: w3 \in good\ v3'$

using $gfun\text{-}mem$ [of $v2'\ v3'\ t\ x\ e\ \varrho1$] by blast

from $w3\text{-}v3p\ vp\text{-}v3p$ have $w3\text{-}vp: w3 \in good\ v'$ by (rule $sub\text{-}good$)

from $e1\text{-}w1\ e2\text{-}w2\ w1\ e\text{-}w3\ w3\text{-}vp$ show $\exists v. \varrho' \vdash EApp\ e1\ e2 \Downarrow v \wedge v \in good\ v'$ by blast

qed

next — EPrim

case $(5\ f\ e1\ e2\ \varrho\ v'\ \varrho')$

from 5(3) show ?case

proof

fix $n1\ n2$ assume $n1\text{-}e1: VNat\ n1 \in E\ e1\ \varrho$ and $n2\text{-}e2: VNat\ n2 \in E\ e2\ \varrho$ and

$vp: v' = VNat\ (f\ n1\ n2)$

from 5(1)[of $VNat\ n1\ \varrho$] $n1\text{-}e1$ 5(4) have $e1\text{-}w1: \varrho' \vdash e1 \Downarrow BNat\ n1$ by auto

from 5(2)[of $VNat\ n2\ \varrho$] $n2\text{-}e2$ 5(4) have $e2\text{-}w2: \varrho' \vdash e2 \Downarrow BNat\ n2$ by auto

from $e1\text{-}w1\ e2\text{-}w2$ have 1: $\varrho' \vdash EPrim\ f\ e1\ e2 \Downarrow BNat\ (f\ n1\ n2)$ by blast

from vp have 2: $BNat\ (f\ n1\ n2) \in good\ v'$ by auto

from 1 2 show $\exists v. \varrho' \vdash EPrim\ f\ e1\ e2 \Downarrow v \wedge v \in good\ v'$ by auto

qed

next — EIf

case $(6\ e1\ e2\ e3\ \varrho\ v'\ \varrho')$

from 6(4) show ?case

proof

fix n assume $n\text{-}e1: VNat\ n \in E\ e1\ \varrho$ and $els: n = 0 \longrightarrow v' \in E\ e3\ \varrho$ and

$thn: n \neq 0 \longrightarrow v' \in E\ e2\ \varrho$

from 6(1)[of $VNat\ n\ \varrho$] $n\text{-}e1$ 6(5) have $e1\text{-}w1: \varrho' \vdash e1 \Downarrow BNat\ n$ by auto

show $\exists v. \varrho' \vdash EIf\ e1\ e2\ e3 \Downarrow v \wedge v \in good\ v'$

proof (cases $n = 0$)

case True

from 6(2)[of $n\ v'\ \varrho$] True els 6(5) obtain $w3$ where

$e3\text{-}w3: \varrho' \vdash e3 \Downarrow w3$ and $w3\text{-}vp: w3 \in good\ v'$ by blast

from $e1\text{-}w1$ True $e3\text{-}w3\ w3\text{-}vp$ show ?thesis by blast

next

```

case False
from  $6(3)[\text{of } n \ v' \ \varrho']$  False thn  $6(5)$  obtain  $w2$  where
   $e2\text{-}w2: \varrho' \vdash e2 \Downarrow w2$  and  $w2\text{-}vp: w2 \in \text{good } v'$  by blast
from  $e1\text{-}w1$  False  $e2\text{-}w2$  have  $\varrho' \vdash \text{EIf } e1 \ e2 \ e3 \Downarrow w2$ 
  using eval-if1[of  $\varrho' \ e1 \ n \ e2 \ w2 \ e3$ ] by simp
from this  $w2\text{-}vp$  show ?thesis by (rule-tac  $x=w2$  in exI) simp
qed
qed
qed

```

theorem *sound-wrt-op-sem*:

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

proof –

```

have  $VNat \ n \in E \ (ENat \ n) \ []$  by simp
with  $E\text{-}e\text{-}n$  have  $1: VNat \ n \in E \ e \ []$  by simp
have  $2: \text{good-env} \ [] \ []$  by auto
from  $1 \ 2$  obtain  $v$  where  $e\text{-}v: [] \vdash e \Downarrow v$  and  $v\text{-}n: v \in \text{good} \ (VNat \ n)$  using denot-terminates by blast
from  $v\text{-}n$  have  $v: v = BNat \ n$  by auto
from  $e\text{-}v \ fv\text{-}e$  obtain  $v' \ ob$  where  $e\text{-}vp: e \longrightarrow^* v'$  and
   $vp\text{-}ob: \text{observe } v' \ ob$  and  $v\text{-}ob: bs\text{-}observe \ v \ ob$  using sound-wrt-small-step by blast
from  $e\text{-}vp \ vp\text{-}ob \ v\text{-}ob \ v$  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 ::  $val \Rightarrow val \Rightarrow val \ option$  (infix  $\sqcup$  60) where
   $(VNat \ n) \sqcup (VNat \ n') = (\text{if } n = n' \ \text{then } Some \ (VNat \ n) \ \text{else } None) \mid$ 
   $(VFun \ f) \sqcup (VFun \ f') = Some \ (VFun \ (f \mid \cup \ f')) \mid$ 
   $v \sqcup v' = None$ 

```

lemma *combine-values*:

```

assumes  $vv: \text{isval } v$  and  $v1v: v1 \in E \ v \ \varrho$  and  $v2v: v2 \in E \ v \ \varrho$ 
shows  $\exists v3. v3 \in E \ v \ \varrho \wedge (v1 \sqcup v2 = Some \ v3)$ 
using  $vv \ v1v \ v2v$  by (induction  $v$  arbitrary: v1 v2 \varrho) auto

```

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

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*: $v1 \sqcup v2 = \text{Some } v12 \implies v2 \sqsubseteq v12$

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*: $\llbracket v1 \sqcup v2 = \text{Some } v12; v1 \sqsubseteq v3; v2 \sqsubseteq v3 \rrbracket \implies v12 \sqsubseteq v3$

apply (*cases* $v1$) **apply** (*cases* $v2$) **apply** *force+* **done**

lemma *e-val*: $\text{isval } v \implies \exists v'. v' \in E v \varrho$

apply (*case-tac* v) **apply** *auto* **apply** (*rule-tac* $x=\{\|\}$ in exI) **apply** *force* **done**

lemma *reverse-subst-lam*:

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

proof (*induction* f arbitrary: $x e e' \varrho v y$)

case *empty*
from *empty*(2) *is-val-def* **obtain** v' **where** $vp-v: v' \in E v \llbracket$ **using** *e-val*[*of* $v \llbracket$] **by** *blast*
let $?R = (y,v')\#\varrho$
have 1: $VFun \{\|\} \in E (ELam x e') ?R$ **by** *simp*
have 2: $?R \approx (y, v') \# \varrho$ **by** *auto*
from $vp-v$ 1 2 **show** $?case$ **by** *blast*

next

case (*insert* $a f x e e' \varrho v y$)
from *insert*(3) **have** 1: $VFun f \in E (ELam x e) \varrho$ **by** *auto*
obtain $v1 v2$ **where** $a = (v1,v2)$ **by** (*cases* a) *simp*
from *insert* 1 **have** $\exists \varrho' v''. v'' \in E v \llbracket \wedge VFun f \in E (ELam x e') \varrho' \wedge \varrho' \approx ((y,v'')\#\varrho)$
by *blast*
from *this* **obtain** $\varrho'' v''$ **where** $vpp-v: v'' \in E v \llbracket$ **and** $f-l: VFun f \in E (ELam x e') \varrho''$
and $rpp-r: \varrho'' \approx ((y,v'')\#\varrho)$ **by** *blast*
from *insert*(3) a **have** $v2-e: v2 \in E e ((x,v1)\#\varrho)$ **using** *e-lam-elim2* **by** *blast*
from *insert* $v2-e$ **have** $\exists \varrho'' v'. v' \in E v \llbracket \wedge v2 \in E e' \varrho'' \wedge \varrho'' \approx (y, v')\#(x, v1)\#\varrho$ **by** *auto*
from *this* **obtain** $\varrho3 v'$ **where** $vp-v: v' \in E v \llbracket$ **and** $v2-ep: v2 \in E e' \varrho3$
and $r3: \varrho3 \approx (y,v') \# (x,v1) \# \varrho$ **by** *blast*
from *insert*(4) **have** *isval* v **by** *auto*
from *this* $vp-v$ $vpp-v$ **obtain** $v3$ **where** $v3-v: v3 \in E v \llbracket$ **and** $vp-vpp: v' \sqcup v'' = \text{Some } v3$
using *combine-values* **by** *blast*
have 4: $VFun (\text{finsert } a f) \in E (ELam x e') ((y, v3) \# \varrho)$

proof –

from $vp-vpp$ **have** $v3-vpp: v'' \sqsubseteq v3$ **using** *le-union2* **by** *simp*
from $rpp-r$ $v3-vpp$ **have** $\varrho'' \sqsubseteq (y,v3)\#\varrho$ **by** (*simp* *add: env-eq-def env-le-def*)
from $f-l$ *this* **have** 2: $VFun f \in E (ELam x e') ((y, v3) \# \varrho)$ **by** (*rule* *raise-env*)
from $vp-vpp$ **have** $vp-v3: v' \sqsubseteq v3$ **using** *le-union1* **by** *simp*
from $vp-v3$ $r3$ **insert** **have** $\varrho3 \sqsubseteq (x,v1)\#(y,v3)\#\varrho$ **by** (*simp* *add: env-eq-def env-le-def*)
from $v2-ep$ *this* **have** 3: $v2 \in E e' ((x,v1)\#(y,v3)\#\varrho)$ **by** (*rule* *raise-env*)
from 2 3 a **show** $?thesis$ **using** *e-lam-intro2* **by** *blast*

qed

have 5: $(y, v3) \# \varrho \approx (y, v3) \# \varrho$ **by** *auto*

from $v3-v$ 4 5 **show** $?case$ **by** *blast*

qed

lemma *lookup-ext-none*: $\llbracket \text{lookup } \varrho \ y = \text{None}; x \neq y \rrbracket \implies \text{lookup } ((x,v)\#\varrho) \ y = \text{None}$
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*: $e = \text{EVar } y \wedge v = e'$ **and** *vv*: *is-val* v **and** *vp-E*: $v' \in E \ e' \ \varrho$
shows $\exists \varrho' \ v'' . v'' \in E \ v \ \square \wedge v' \in E \ e \ \varrho' \wedge \varrho' \approx ((y,v'')\#\varrho)$

proof —

from *vv* **have** *lx*: $\forall x . x \in FV \ v \longrightarrow \text{lookup } \square \ x = \text{lookup } \varrho \ x$ **by** *auto*
from *ev vp-E lx env-strengthen*[of $v' \ v \ \varrho \ \square$] **have** *n-Ev*: $v' \in E \ v \ \square$ **by** *blast*
have *ly*: $\text{lookup } ((y,v')\#\varrho) \ y = \text{Some } v'$ **by** *simp*
from *env-eq-def* **have** *rr*: $((y,v')\#\varrho) \approx ((y,v')\#\varrho)$ **by** *simp*
from *ev ly* **have** *n-Ee*: $v' \in E \ e \ ((y,v')\#\varrho)$ **by** *simp*
from *n-Ev rr n-Ee* **show** *?thesis* **by** *blast*

qed

lemma *reverse-subst-pres-denot*:

assumes *vep*: $v' \in E \ e' \ \varrho$ **and** *vv*: *is-val* v **and** *ep*: $e' = \text{subst } y \ v \ e$
shows $\exists \varrho' \ v'' . v'' \in E \ v \ \square \wedge v' \in E \ e \ \varrho' \wedge \varrho' \approx ((y,v'')\#\varrho)$
using *vep vv ep*

proof (*induction arbitrary*: $v' \ y \ v \ e$ *rule*: *E.induct*)

case $(1 \ n \ \varrho) \text{ — } e' = \text{ENat } n$
from $1(1)$ **have** *vp*: $v' = \text{VNat } n$ **by** *auto*
from $1(3)$ **have** $e = \text{ENat } n \vee (e = \text{EVar } y \wedge v = \text{ENat } n)$ **by** (*cases e, auto*)
then show *?case*

proof

assume $e = \text{ENat } n$
from $1(2)$ *e-val is-val-def* **obtain** v'' **where** *vpp-E*: $v'' \in E \ v \ \square$ **by** *force*
from *env-eq-def* **have** *rr*: $((y,v'')\#\varrho) \approx ((y,v'')\#\varrho)$ **by** *simp*
from *vp e* **have** *vp-E*: $v' \in E \ e \ ((y,v'')\#\varrho)$ **by** *simp*
from *vpp-E vp-E rr* **show** *?thesis* **by** *blast*

next

assume $e = \text{EVar } y \wedge v = \text{ENat } n$
from *ev* $1(2)$ $1(1)$ *rev-subst-var* **show** *?thesis* **by** *blast*

qed

next

case $(2 \ x \ \varrho) \text{ — } e' = \text{EVar } x$
from 2 **have** $e = \text{EVar } x$ **by** (*cases e, auto*)
from $2 \ e$ **have** *xy*: $x \neq y$ **by** *force*
from $2(2)$ *e-val is-val-def* **obtain** v'' **where** *vpp-E*: $v'' \in E \ v \ \square$ **by** *force*
from *env-eq-def* **have** *rr*: $((y,v'')\#\varrho) \approx ((y,v'')\#\varrho)$ **by** *simp*
from $2(1)$ **obtain** *vx* **where** *lx*: $\text{lookup } \varrho \ x = \text{Some } vx$ **and** *vp-vx*: $v' \sqsubseteq vx$ **by** *auto*
from $e \ lx \ vp-vx \ xy$ **have** *vp-E*: $v' \in E \ e \ ((y,v'')\#\varrho)$ **by** *simp*
from *vpp-E rr vp-E* **show** *?case* **by** *blast*

next

case $(3 \ x \ eb \ \varrho)$
{ assume $e = \text{EVar } y \wedge v = \text{ELam } x \ eb$
from *ev* $3(3)$ $3(2)$ *rev-subst-var* **have** *?case* **by** *blast*
} **also** **{ assume** $e = \text{ELam } x \ eb \wedge x = y$
from $3(3)$ *e-val is-val-def* **obtain** v'' **where** *vpp-E*: $v'' \in E \ v \ \square$ **by** *force*
from *env-eq-def* **have** *rr*: $((y,v'')\#\varrho) \approx ((y,v'')\#\varrho)$ **by** *simp*
from *ex* **have** *lz*: $\forall z . z \in FV \ (\text{ELam } x \ eb) \longrightarrow \text{lookup } ((y,v'')\#\varrho) \ z = \text{lookup } \varrho \ z$ **by** *auto*
from *ex* $3(2)$ *lz env-strengthen*[of $v' \ \text{ELam } x \ eb \ \varrho \ (y,v'')\#\varrho$]
have *vp-E*: $v' \in E \ e \ ((y,v'')\#\varrho)$ **by** *blast*
from *vpp-E vp-E rr* **have** *?case* **by** *blast*
} **moreover** **{ assume** *exb*: $\exists e' . e = \text{ELam } x \ e' \wedge x \neq y \wedge eb = \text{subst } y \ v \ e'$
from *exb* **obtain** e'' **where** $e = \text{ELam } x \ e''$ **and** *xy*: $x \neq y$
and *eb*: $eb = \text{subst } y \ v \ e''$ **by** *blast*
from $3(2)$ **obtain** f **where** *vp*: $v' = \text{VFun } f$ **by** *auto*
from $3(2)$ *vp* **have** *f-E*: $\text{VFun } f \in E \ (\text{ELam } x \ eb) \ \varrho$ **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** $IH: \forall\ v1\ v2. v2 \in E\ (subst\ y\ v\ e')\ ((x,v1)\#\rho)$
 $\rightarrow (\exists\ \rho'\ v'. v' \in E\ v\ [] \wedge v2 \in E\ e'\ \rho' \wedge \rho' \approx (y,v')\#\rho)$
apply *clarify* **apply** $(subgoal\text{-}tac\ (v1,v2) \in fset\ \{(v1,v2)\})$ **prefer** 2 **apply** *simp*
apply $(rule\ 3(1))$ **apply** *assumption* **apply** *simp+* **done**
from $f\text{-}E\ 3(3)$ $ls\ xy\ IH\ e\ vp$ **have** $?case$ **apply** *clarify* **apply** $(rule\ reverse\text{-}subst\text{-}lam)$
apply *blast+* **done**
} moreover from $3(4)$ **have** $(e = EVar\ y \wedge v = ELam\ x\ eb)$
 $\vee (e = ELam\ x\ eb \wedge x = y)$
 $\vee (\exists\ e'. e = ELam\ x\ e' \wedge x \neq y \wedge eb = subst\ y\ v\ e')$ **by** $(cases\ e)$ *auto*
ultimately show $?case$ **by** *blast*
next
case $(4\ e1\ e2\ \rho) \text{ --- } e' = EApp\ e1\ e2$
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** $(rename\text{-}tac\ x)$ **apply** *auto* **apply** $(case\text{-}tac\ y = x)$ **apply** *auto*
apply $(rename\text{-}tac\ x1\ x2)$ **apply** $(case\text{-}tac\ y = x1)$ **apply** *auto* **done**
from $4(3)$ **obtain** $f\ v2$ **and** $v2':val$ **and** $v3'$ **where**
 $f\text{-}E: VFun\ f \in E\ e1\ \rho$ **and** $v2\text{-}E: v2 \in E\ e2\ \rho$ **and** $v23: (v2',v3') \in fset\ f$
and $v2p\text{-}v2: v2' \sqsubseteq v2$ **and** $vp\text{-}v3: v' \sqsubseteq v3'$ **by** *blast*
from $4(1)$ $f\text{-}E\ 4(4)$ $e1$ **obtain** $\rho1\ w1$ **where** $v1\text{-}Ev: w1 \in E\ v\ []$ **and** $f\text{-}E1: VFun\ f \in E\ e1'\ \rho1$
and $r1: \rho1 \approx (y,w1)\#\rho$ **by** *blast*
from $4(2)$ $v2\text{-}E\ 4(4)$ $e2$ **obtain** $\rho2\ w2$ **where** $v2\text{-}Ev: w2 \in E\ v\ []$ **and** $v2\text{-}E2: v2 \in E\ e2'\ \rho2$
and $r2: \rho2 \approx (y,w2)\#\rho$ **by** *blast*
from $4(4)$ $v1\text{-}Ev\ v2\text{-}Ev$ *combine-values* **obtain** $w3$ **where**
 $w3\text{-}Ev: w3 \in E\ v\ []$ **and** $w123: w1 \sqcup w2 = Some\ w3$ **by** $(simp\ only: is\text{-}val\text{-}def)$ *blast*
from $w123$ *le-union1* **have** $w13: w1 \sqsubseteq w3$ **by** *blast*
from $w123$ *le-union2* **have** $w23: w2 \sqsubseteq w3$ **by** *blast*
from $w13$ **have** $r13: ((y,w1)\#\rho) \sqsubseteq ((y,w3)\#\rho)$ **by** $(simp\ add: env\text{-}le\text{-}def)$
from $w23$ **have** $r23: ((y,w2)\#\rho) \sqsubseteq ((y,w3)\#\rho)$ **by** $(simp\ add: env\text{-}le\text{-}def)$
from $r1\ f\text{-}E1$ **have** $f\text{-}E1b: VFun\ f \in E\ e1'\ ((y,w1)\#\rho)$ **by** $(rule\ env\text{-}swap)$
from $f\text{-}E1b\ r13$ **have** $f\text{-}E1c: VFun\ f \in E\ e1'\ ((y,w3)\#\rho)$ **by** $(rule\ raise\text{-}env)$
from $r2\ v2\text{-}E2$ **have** $v2\text{-}E2b: v2 \in E\ e2'\ ((y,w2)\#\rho)$ **by** $(rule\ env\text{-}swap)$
from $v2\text{-}E2b\ r23$ **have** $v2\text{-}E2c: v2 \in E\ e2'\ ((y,w3)\#\rho)$ **by** $(rule\ raise\text{-}env)$
from $f\text{-}E1c\ v2\text{-}E2c\ v23\ v2p\text{-}v2\ vp\text{-}v3$ **have** $vp\text{-}E2: v' \in E\ (EApp\ e1'\ e2')\ ((y,w3)\#\rho)$ **by** *blast*
have $rr3: ((y,w3)\#\rho) \approx ((y,w3)\#\rho)$ **by** $(simp\ add: env\text{-}eq\text{-}def)$
from $w3\text{-}Ev\ vp\text{-}E2\ rr3\ e$ **show** $?case$ **by** *blast*
next
case $(5\ f\ e1\ e2\ \rho) \text{ --- } e' = EPrim\ f\ e1\ e2$, 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** $(rename\text{-}tac\ x)$ **apply** $(case\text{-}tac\ y = x)$ **apply** *auto*
apply $(rename\text{-}tac\ x1\ x2)$ **apply** $(case\text{-}tac\ y = x1)$ **apply** *auto* **done**
from $5(3)$ **obtain** $n1\ n2$ **where**
 $n1\text{-}E: VNat\ n1 \in E\ e1\ \rho$ **and** $n2\text{-}E: VNat\ n2 \in E\ e2\ \rho$ **and** $vp: v' = VNat\ (f\ n1\ n2)$ **by** *blast*
from $5(1)$ $n1\text{-}E\ 5(4)$ $e1$ **obtain** $\rho1\ w1$ **where** $v1\text{-}Ev: w1 \in E\ v\ []$ **and** $n1\text{-}E1: VNat\ n1 \in E\ e1'\ \rho1$
and $r1: \rho1 \approx (y,w1)\#\rho$ **by** *blast*
from $5(2)$ $n2\text{-}E\ 5(4)$ $e2$ **obtain** $\rho2\ w2$ **where** $v2\text{-}Ev: w2 \in E\ v\ []$ **and** $n2\text{-}E2: VNat\ n2 \in E\ e2'\ \rho2$
and $r2: \rho2 \approx (y,w2)\#\rho$ **by** *blast*
from $5(4)$ $v1\text{-}Ev\ v2\text{-}Ev$ *combine-values* **obtain** $w3$ **where**
 $w3\text{-}Ev: w3 \in E\ v\ []$ **and** $w123: w1 \sqcup w2 = Some\ w3$ **by** $(simp\ only: is\text{-}val\text{-}def)$ *blast*
from $w123$ *le-union1* **have** $w13: w1 \sqsubseteq w3$ **by** *blast*
from $w123$ *le-union2* **have** $w23: w2 \sqsubseteq w3$ **by** *blast*
from $w13$ **have** $r13: ((y,w1)\#\rho) \sqsubseteq ((y,w3)\#\rho)$ **by** $(simp\ add: env\text{-}le\text{-}def)$
from $w23$ **have** $r23: ((y,w2)\#\rho) \sqsubseteq ((y,w3)\#\rho)$ **by** $(simp\ add: env\text{-}le\text{-}def)$
from $r1\ n1\text{-}E1$ **have** $n1\text{-}E1b: VNat\ n1 \in E\ e1'\ ((y,w1)\#\rho)$ **by** $(rule\ env\text{-}swap)$
from $n1\text{-}E1b\ r13$ **have** $n1\text{-}E1c: VNat\ n1 \in E\ e1'\ ((y,w3)\#\rho)$ **by** $(rule\ raise\text{-}env)$
from $r2\ n2\text{-}E2$ **have** $n2\text{-}E2b: VNat\ n2 \in E\ e2'\ ((y,w2)\#\rho)$ **by** $(rule\ env\text{-}swap)$
from $n2\text{-}E2b\ r23$ **have** $v2\text{-}E2c: VNat\ n2 \in E\ e2'\ ((y,w3)\#\rho)$ **by** $(rule\ raise\text{-}env)$
from $n1\text{-}E1c\ v2\text{-}E2c\ vp$ **have** $vp\text{-}E2: v' \in E\ (EPrim\ f\ e1'\ e2')\ ((y,w3)\#\rho)$ **by** *blast*
have $rr3: ((y,w3)\#\rho) \approx ((y,w3)\#\rho)$ **by** $(simp\ add: env\text{-}eq\text{-}def)$

from $w3\text{-}Ev\ vp\text{-}E2\ rr3\ e$ **show** $?case$ **by** $blast$
next
case $(6\ e1\ e2\ e3\ \varrho) \text{---} e' = \text{Elf}\ e1\ e2\ e3$
from $6(5)\ 6(6)$ **obtain** $e1'\ e2'\ e3'$ **where**
 $e: e = \text{Elf}\ e1'\ e2'\ e3'$ **and** $e1:e1 = \text{subst}\ y\ v\ e1'$ **and** $e2: e2 = \text{subst}\ y\ v\ e2'$
and $e3: e3 = \text{subst}\ y\ v\ e3'$
apply $(cases\ e)$ **apply** $auto$ **apply** $(case\text{-}tac\ y=x1)$ **apply** $auto$ **apply** $(case\text{-}tac\ y=x31)$ **by** $auto$
from $6(4)$ $e\text{-if}\text{-elim}$ **obtain** n **where** $n\text{-}E: \text{VNat}\ n \in E\ e1\ \varrho$ **and**
 $els: n = 0 \longrightarrow v' \in E\ e3\ \varrho$ **and** $thn: n \neq 0 \longrightarrow v' \in E\ e2\ \varrho$ **by** $blast$
from $6\ n\text{-}E\ e1$ **obtain** $\varrho1\ w1$ **where** $w1\text{-}Ev: w1 \in E\ v\ []$ **and** $n\text{-}E2: \text{VNat}\ n \in E\ e1'\ \varrho1$
and $r1: \varrho1 \approx (y, w1)\#\varrho$ **by** $blast$
show $?case$
proof $(cases\ n = 0)$
case $True$ **with** els **have** $vp\text{-}E2: v' \in E\ e3\ \varrho$ **by** $simp$
from $6\ vp\text{-}E2\ e3$ **obtain** $\varrho2\ w2$ **where** $w2\text{-}Ev: w2 \in E\ v\ []$ **and** $vp\text{-}E2: v' \in E\ e3'\ \varrho2$
and $r2: \varrho2 \approx (y, w2)\#\varrho$ **by** $blast$
from $6(5)\ w1\text{-}Ev\ w2\text{-}Ev\ combine\text{-}values$ **obtain** $w3$ **where**
 $w3\text{-}Ev: w3 \in E\ v\ []$ **and** $w123: w1 \sqcup w2 = \text{Some}\ w3$ **by** $(simp\ only: is\text{-}val\text{-}def)$ $blast$
from $w123\ le\text{-}union1$ **have** $w13: w1 \sqsubseteq w3$ **by** $blast$
from $w123\ le\text{-}union2$ **have** $w23: w2 \sqsubseteq w3$ **by** $blast$
from $w13$ **have** $r13: ((y, w1)\#\varrho) \sqsubseteq ((y, w3)\#\varrho)$ **by** $(simp\ add: env\text{-}le\text{-}def)$
from $w23$ **have** $r23: ((y, w2)\#\varrho) \sqsubseteq ((y, w3)\#\varrho)$ **by** $(simp\ add: env\text{-}le\text{-}def)$
from $r1\ n\text{-}E2$ **have** $n\text{-}E1b: \text{VNat}\ n \in E\ e1'\ ((y, w1)\#\varrho)$ **by** $(rule\ env\text{-}swap)$
from $n\text{-}E1b\ r13$ **have** $n\text{-}E1c: \text{VNat}\ n \in E\ e1'\ ((y, w3)\#\varrho)$ **by** $(rule\ raise\text{-}env)$
from $r2\ vp\text{-}E2$ **have** $vp\text{-}E2b: v' \in E\ e3'\ ((y, w2)\#\varrho)$ **by** $(rule\ env\text{-}swap)$
from $vp\text{-}E2b\ r23$ **have** $vp\text{-}E2c: v' \in E\ e3'\ ((y, w3)\#\varrho)$ **by** $(rule\ raise\text{-}env)$
have $rr3: ((y, w3)\#\varrho) \approx ((y, w3)\#\varrho)$ **by** $(simp\ add: env\text{-}eq\text{-}def)$
from $True\ n\text{-}E1c\ vp\text{-}E2c\ e$ **have** $vp\text{-}E3: v' \in E\ e\ ((y, w3)\#\varrho)$ **by** $auto$
from $w3\text{-}Ev\ rr3\ vp\text{-}E3$ **show** $?thesis$ **by** $blast$
next
case $False$ **with** thn **have** $vp\text{-}E2: v' \in E\ e2\ \varrho$ **by** $simp$
from $6\ vp\text{-}E2\ e2$ **obtain** $\varrho2\ w2$ **where** $w2\text{-}Ev: w2 \in E\ v\ []$ **and** $vp\text{-}E2: v' \in E\ e2'\ \varrho2$
and $r2: \varrho2 \approx (y, w2)\#\varrho$ **by** $blast$
from $6(5)\ w1\text{-}Ev\ w2\text{-}Ev\ combine\text{-}values$ **obtain** $w3$ **where**
 $w3\text{-}Ev: w3 \in E\ v\ []$ **and** $w123: w1 \sqcup w2 = \text{Some}\ w3$ **by** $(simp\ only: is\text{-}val\text{-}def)$ $blast$
from $w123\ le\text{-}union1$ **have** $w13: w1 \sqsubseteq w3$ **by** $blast$
from $w123\ le\text{-}union2$ **have** $w23: w2 \sqsubseteq w3$ **by** $blast$
from $w13$ **have** $r13: ((y, w1)\#\varrho) \sqsubseteq ((y, w3)\#\varrho)$ **by** $(simp\ add: env\text{-}le\text{-}def)$
from $w23$ **have** $r23: ((y, w2)\#\varrho) \sqsubseteq ((y, w3)\#\varrho)$ **by** $(simp\ add: env\text{-}le\text{-}def)$
from $r1\ n\text{-}E2$ **have** $n\text{-}E1b: \text{VNat}\ n \in E\ e1'\ ((y, w1)\#\varrho)$ **by** $(rule\ env\text{-}swap)$
from $n\text{-}E1b\ r13$ **have** $n\text{-}E1c: \text{VNat}\ n \in E\ e1'\ ((y, w3)\#\varrho)$ **by** $(rule\ raise\text{-}env)$
from $r2\ vp\text{-}E2$ **have** $vp\text{-}E2b: v' \in E\ e2'\ ((y, w2)\#\varrho)$ **by** $(rule\ env\text{-}swap)$
from $vp\text{-}E2b\ r23$ **have** $vp\text{-}E2c: v' \in E\ e2'\ ((y, w3)\#\varrho)$ **by** $(rule\ raise\text{-}env)$
have $rr3: ((y, w3)\#\varrho) \approx ((y, w3)\#\varrho)$ **by** $(simp\ add: env\text{-}eq\text{-}def)$
from $False\ n\text{-}E1c\ vp\text{-}E2c\ e$ **have** $vp\text{-}E3: v' \in E\ e\ ((y, w3)\#\varrho)$ **by** $auto$
from $w3\text{-}Ev\ rr3\ vp\text{-}E3$ **show** $?thesis$ **by** $blast$
qed
qed

13.2 Reverse reduction preserves denotation

lemma $reverse\text{-}step\text{-}pres\text{-}denot$:

fixes $e::exp$ **assumes** $e\text{-ep}: e \longrightarrow e'$ **and** $v\text{-ep}: v \in E\ e'\ \varrho$

shows $v \in E\ e\ \varrho$

using $e\text{-ep}\ v\text{-ep}$

proof $(induction\ arbitrary: v\ \varrho\ rule: reduce.induct)$

case $(beta\ v\ x\ e\ v'\ \varrho)$

from $beta$ **obtain** $\varrho'\ v''$ **where** $1: v'' \in E\ v\ []$ **and** $2: v' \in E\ e\ \varrho'$ **and** $3: \varrho' \approx (x, v'')\ \#\ \varrho$

using $reverse\text{-}subst\text{-}pres\text{-}denot[of\ v'\ subst\ x\ v\ e\ \varrho\ v\ x\ e]$ **by** $blast$

from $beta\ 1\ 2\ 3$ **show** $?case$

apply $simp$ **apply** $(rule\text{-}tac\ x=\{|(v'', v')|})$ **in** exI **apply** $(rule\ conjI)$

```

  apply clarify apply simp apply clarify 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 →* e' and v-ep: v ∈ E e' ρ shows v ∈ E e ρ
  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 →* v and vv: is-val v
  shows ∃ v'. v' ∈ E e ρ ∧ v' ∈ E v []
proof -
  from vv have ∃ v'. v' ∈ E v [] using e-val by auto
  from this obtain v' where vp-v: v' ∈ E v [] by blast
  from vp-v vv have vp-v2: v' ∈ E v ρ using env-strengthen by force
  from ev vp-v2 reverse-multi-step-pres-denot[of e v v' ρ]
  have v' ∈ E e ρ by simp
  from this vp-v show ?thesis by blast
qed

```

```

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

```

```

theorem complete-wrt-op-sem:
  assumes e-n: e ↓ ONat n shows E e [] = E (ENat n) []
proof -
  from e-n have 1: e →* 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' ⇒ 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' ] ⇒ 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]: $\llbracket E\ e1 = E\ e1'; E\ e2 = E\ e2'; E\ e3 = E\ e3' \rrbracket$
 $\implies E\ (EIf\ e1\ e2\ e3) = E\ (EIf\ e1'\ e2'\ e3')$

by (rule ext) simp

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

fun *plug* :: *ctx* \Rightarrow *exp* \Rightarrow *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*: $E\ e = E\ e' \implies E\ (plug\ C\ e) = E\ (plug\ C\ e')$

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** *fv*: $FV\ e = \{\}$ **shows** $E\ e\ [] = \{\}$

proof (rule classical)

assume $E\ e\ [] \neq \{\}$

from this obtain *A* where *wte*: $A \in E\ e\ []$ by auto

have *ge*: *good-env* $[]\ []$ by blast

from *wte* *ge* obtain *v* where *e-v*: $[] \vdash e \Downarrow v$ **and** *gv*: $v \in good\ A$

using *denot-terminates* by blast

from *e-v* *fv* obtain *v'* where *e-vp*: $e \longrightarrow^* v'$ **and** *val-vp*: *isval* *v'*

using *sound-wrt-small-step* by blast

from *d* *e-vp* have $\exists\ e'.\ v' \longrightarrow e'$ 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*: $FV\ e = \{\}$ **shows** $E\ e\ [] = \{\}$

proof (rule classical)

assume $E\ e\ [] \neq \{\}$

from this obtain *A* where *wte*: $A \in E\ e\ []$ by auto

have *ge*: *good-env* $[]\ []$ by blast

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

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

proof (*rule classical*)

assume $nd\text{-gw}: \neg (\text{diverge } e \vee \text{goes-wrong } e)$
 from *this* have $nd: \neg \text{diverge } e$ by *blast*
 from $nd\text{-gw}$ have $gw: \neg \text{goes-wrong } e$ by *blast*
 from nd obtain $v::\text{exp}$ where $e\text{-v}: e \longrightarrow^* v$ and $\text{stuck}: \neg (\exists e'. v \longrightarrow e')$
 by (*simp only: diverge-def*) *blast*
 from gw $e\text{-v}$ stuck have $\text{val-v}: \text{isval } v$ by (*simp only: goes-wrong-def stuck-def*) *blast*
 from $fv\text{-e}$ $e\text{-v}$ have $fv\text{-v}: FV v = \{\}$ using *reduction-pres-fv by auto*
 from val-v $fv\text{-v}$ have $\text{val-v2}: \text{is-val } v$ by *simp*
 from $e\text{-v}$ 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* $E\text{-e}$ have *False* by *auto*
 from *this* show *?thesis ..*
 qed

lemma *val-ty-observe*:

$[A \in E v []; A \in E v' [];$
 $\text{observe } v \text{ ob}; \text{isval } v'; \text{isval } v] \implies \text{observe } v' \text{ ob}$
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 $fv\text{-e1}: FV (\text{plug } C e1) = \{\}$ and $fv\text{-e2}: FV (\text{plug } C e2) = \{\}$
shows $\text{run } (\text{plug } C e1) \text{ ob} \longrightarrow \text{run } (\text{plug } C e2) \text{ ob}$

proof

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

```

    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 vpp 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** \simeq 51) **where**
 $e \simeq e' \equiv \forall C \text{ ob. } FV(\text{plug } C \ e) = \{\} \wedge FV(\text{plug } C \ e') = \{\} \longrightarrow$
 $\text{run}(\text{plug } C \ e) \text{ ob} = \text{run}(\text{plug } C \ e') \text{ ob}$

theorem *denot-sound-wrt-ctx-equiv*: **assumes** $e12: E \ e1 = E \ e2$ **shows** $e1 \simeq e2$
 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** $nz: n \neq 0$
shows $E(EIf \ e1 \ e2 \ e3) = E(e2)$
 using *e1 nz* **by** *auto*

theorem *eval-app-lam*: **assumes** $vv: \text{is-val } v$
shows $E(EApp \ (ELam \ x \ e) \ v) = E(\text{subst } x \ v \ e)$
apply (*rule ext*) **apply** (*rule equalityI*) **apply** (*rule subsetI*)
apply (*subgoal-tac EApp (ELam x e) v \longrightarrow 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 \longrightarrow 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* \Rightarrow *bool* where

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

lemma *is-value-is-val[simp]*: *is-value e* \Longrightarrow *isval e* \wedge *FV e* = {}
by (*case-tac e*) *auto*

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

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

lemma *opt-correct-aux*: *E e* = *E* (*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  $\times$  val) fset | Abs val option | Wrong

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

```

16.2 Set monad

```

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

```

```

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

```

```

definition errset-bind :: val set  $\Rightarrow$  (val  $\Rightarrow$  val set)  $\Rightarrow$  val set where
  errset-bind m f  $\equiv$  { v.  $\exists$  v'. v'  $\in$  m  $\wedge$  v'  $\neq$  Wrong  $\wedge$  v  $\in$  f v' }  $\cup$  {v. v = Wrong  $\wedge$  Wrong  $\in$  m }
declare errset-bind-def[simp]

```

```

syntax -errset-bind :: [pttrns, val set, val]  $\Rightarrow$  'c ((- := -;//-) 0)
translations P := E; F  $\Rightarrow$  CONST errset-bind E ( $\lambda$ P. F)

```

```

definition return :: val  $\Rightarrow$  val set where
  return v  $\equiv$  {v'. v'  $\sqsubseteq$  v }
declare return-def[simp]

```

16.3 Denotational semantics

```

type-synonym tyenv = (val set) list
type-synonym env = val list

```

```

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

```

iterate-none[intro!]: $iterate\ Ee\ \varrho\ (Fun\ \{\{\}\})\ |$
iterate-again[intro!]: $\llbracket\ iterate\ Ee\ \varrho\ f;\ f' \in Ee\ (f\#\varrho)\ \rrbracket \implies iterate\ Ee\ \varrho\ f'$

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

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

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

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

16.4 Types: substitution and semantics

fun *shift* :: $nat \Rightarrow nat \Rightarrow ty \Rightarrow ty\ \mathbf{where}$

shift $k\ c\ TNat = TNat\ |$
shift $k\ c\ (TVar\ n) = (if\ c \leq n\ then\ TVar\ (n + k)\ else\ TVar\ n)\ |$
shift $k\ c\ (\sigma \rightarrow \sigma') = (shift\ k\ c\ \sigma) \rightarrow (shift\ k\ c\ \sigma')\ |$
shift $k\ c\ (Forall\ \sigma) = Forall\ (shift\ k\ (Suc\ c)\ \sigma)$

fun *subst* :: $nat \Rightarrow ty \Rightarrow ty \Rightarrow ty\ \mathbf{where}$

subst $k\ \tau\ TNat = TNat\ |$
subst $k\ \tau\ (TVar\ n) = (if\ k = n\ then\ \tau$
 $else\ if\ k < n\ then\ TVar\ (n - 1)$
 $else\ TVar\ n)\ |$
subst $k\ \tau\ (\sigma \rightarrow \sigma') = (subst\ k\ \tau\ \sigma) \rightarrow (subst\ k\ \tau\ \sigma')\ |$
subst $k\ \tau\ (Forall\ \sigma) = Forall\ (subst\ (Suc\ k)\ (shift\ (Suc\ 0)\ 0\ \tau)\ \sigma)$

fun $T :: ty \Rightarrow tyenv \Rightarrow val\ set\ \mathbf{where}$

Tnat: $T\ TNat\ \varrho = \{v. \exists n. v = VNat\ n\ \}\ |$
Tvar: $T\ (TVar\ n)\ \varrho = (if\ n < length\ \varrho\ then$
 $\{v. \exists v'. v' \in \varrho!n \wedge v \sqsubseteq v' \wedge v \neq Wrong\}$
 $else\ \{\})\ |$
Tfun: $T\ (\sigma \rightarrow \tau)\ \varrho = \{v. \exists f. v = Fun\ f \wedge$
 $(\forall v1\ v2'. (v1, v2') \in fset\ f \longrightarrow$
 $v1 \in T\ \sigma\ \varrho \longrightarrow (\exists v2. v2 \in T\ \tau\ \varrho \wedge v2' \sqsubseteq v2))\ \}\ |$
Tall: $T\ (Forall\ \tau)\ \varrho = \{v. (\exists v'. v = Abs\ (Some\ v') \wedge (\forall V. v' \in T\ \tau\ (V\#\varrho)))$
 $\vee v = Abs\ None\ \}$

16.5 Type system

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

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

wf-tyvar $\Gamma\ n \equiv n < snd\ \Gamma$

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

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

definition *push-tyvar* :: *tyctx* \Rightarrow *tyctx* **where**
push-tyvar $\Gamma \equiv (fst \Gamma, Suc (snd \Gamma))$

definition *good-ctx* :: *tyctx* \Rightarrow *bool* **where**
good-ctx $\Gamma \equiv \forall n. n < length (fst \Gamma) \longrightarrow snd ((fst \Gamma)!n) \leq snd \Gamma$

definition *lookup* :: *tyctx* \Rightarrow *nat* \Rightarrow *ty option* **where**
lookup $\Gamma n \equiv (if\ n < length\ (fst\ \Gamma)\ then$
 let $k = snd\ \Gamma - snd\ ((fst\ \Gamma)!n)\ in$
 Some (*shift* $k\ 0\ (fst\ ((fst\ \Gamma)!n))$)
 else *None*)

inductive *well-typed* :: *tyctx* \Rightarrow *exp* \Rightarrow *ty* \Rightarrow *bool* ($\vdash - \vdash -$ - [55,55,55] 54) **where**
wtnat[*intro!*]: $\Gamma \vdash ENat\ n : TNat \mid$
wtvar[*intro!*]: $\llbracket lookup\ \Gamma\ n = Some\ \tau \rrbracket \Longrightarrow \Gamma \vdash EVar\ n : \tau \mid$
wtapp[*intro!*]: $\llbracket \Gamma \vdash e : \sigma \rightarrow \tau; \Gamma \vdash e' : \sigma \rrbracket \Longrightarrow \Gamma \vdash EApp\ e\ e' : \tau \mid$
wtlam[*intro!*]: $\llbracket push-ty\ \sigma\ \Gamma \vdash e : \tau \rrbracket \Longrightarrow \Gamma \vdash ELam\ \sigma\ e : \sigma \rightarrow \tau \mid$
wtfix[*intro!*]: $\llbracket push-ty\ (\sigma \rightarrow \tau)\ \Gamma \vdash e : \sigma \rightarrow \tau \rrbracket \Longrightarrow \Gamma \vdash EFix\ (\sigma \rightarrow \tau)\ e : \sigma \rightarrow \tau \mid$
wtabs[*intro!*]: $\llbracket push-tyvar\ \Gamma \vdash e : \tau \rrbracket \Longrightarrow \Gamma \vdash EAbs\ e : Forall\ \tau \mid$
wtinst[*intro!*]: $\llbracket \Gamma \vdash e : Forall\ \tau \rrbracket \Longrightarrow \Gamma \vdash EInst\ e\ \sigma : (subst\ 0\ \sigma\ \tau)$

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

inductive-cases

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

lemma *wfenv-good-ctx*: $\vdash \varrho, \eta : \Gamma \Longrightarrow good-ctx\ \Gamma$

proof (*induction rule*: *wfenv.induct*)

case *wfnil*

then show *?case* **by** (*force simp*: *good-ctx-def*)

next

case (*wfvbind* $\varrho\ \eta\ \Gamma\ v\ \tau$)

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* $\varrho\ \eta\ \Gamma\ V$)

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*]: $n < length\ \varrho1 \Longrightarrow (\varrho1 @ \varrho2)!n = \varrho1!n$

proof (*induction* $\varrho1$ *arbitrary*: $\varrho2\ n$)

case *Nil*

then show *?case* **by** *auto*

next

case (*Cons* $a\ \varrho1$)

then show *?case* **by** (*cases* n) *auto*

qed

lemma *nth-append2[simp]*: $n \geq \text{length } \varrho_1 \implies (\varrho_1 @ \varrho_2)!n = \varrho_2!(n - \text{length } \varrho_1)$
proof (*induction* ϱ_1 *arbitrary*: ϱ_2 n)
 case *Nil*
 then show *?case* **by** *auto*
next
 case (*Cons* a ϱ_1)
 then show *?case* **by** (*cases* n) *auto*
qed

lemma *shift-append-preserves-T-aux*:
 shows $T \tau (\varrho_1 @ \varrho_3) = T (\text{shift } (\text{length } \varrho_2) (\text{length } \varrho_1) \tau) (\varrho_1 @ \varrho_2 @ \varrho_3)$
proof (*induction* τ *arbitrary*: ϱ_1 ϱ_2 ϱ_3)
 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* $T \tau ((V\#\varrho_1) @ \varrho_3) =$
 $T (\text{shift } (\text{length } \varrho_2) (\text{length } (V\#\varrho_1)) \tau) ((V\#\varrho_1) @ \varrho_2 @ \varrho_3))$
 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* $T \tau ((V\#\varrho_1) @ \varrho_3) =$
 $T (\text{shift } (\text{length } \varrho_2) (\text{length } (V\#\varrho_1)) \tau) ((V\#\varrho_1) @ \varrho_2 @ \varrho_3))$
 prefer 2 **apply** *blast* **apply** *force*
 apply (*rule* *disjI2*) **apply** *force* **done**
qed *force+*

lemma *shift-append-preserves-T*: **shows** $T \tau \varrho_3 = T (\text{shift } (\text{length } \varrho_2) 0 \tau) (\varrho_2 @ \varrho_3)$
 using *shift-append-preserves-T-aux*[*of* τ [] ϱ_3 ϱ_2] **by** *auto*

lemma *drop-shift-preserves-T*:
 assumes $k: k \leq \text{length } \varrho$ **shows** $T \tau (\text{drop } k \varrho) = T (\text{shift } k 0 \tau) \varrho$
proof –
 let $?r2 = \text{take } k \varrho$ **and** $?r3 = \text{drop } k \varrho$
 have 1: $T \tau (?r3) = T (\text{shift } (\text{length } ?r2) 0 \tau) (?r2 @ ?r3)$
 using *shift-append-preserves-T-aux*[*of* τ [] $?r3$ $?r2$] **by** *simp*
 have 2: $?r2 @ ?r3 = \varrho$ **by** *simp*
 from k **have** 3: $\text{length } ?r2 = k$ **by** *simp*
 from 1 2 3 **show** *?thesis* **by** *simp*
qed

lemma *shift-cons-preserves-T*: **shows** $T \tau \varrho = T (\text{shift } (\text{Suc } 0) 0 \tau) (b\#\varrho)$
 using *drop-shift-preserves-T*[*of* $\text{Suc } 0$ $b\#\varrho$ τ] **by** *simp*

lemma *compose-shift*: **shows** $\text{shift } (j+k) c \tau = \text{shift } j c (\text{shift } k c \tau)$
 by (*induction* τ *arbitrary*: j k c) *auto*

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

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

lemma *less-wrong[elim!]*: $\llbracket v \sqsubseteq \text{Wrong}; v = \text{Wrong} \implies P \rrbracket \implies P$
by (*case-tac v*) *auto*

lemma *less-nat[elim!]*: $\llbracket v \sqsubseteq \text{VNat } n; v = \text{VNat } n \implies P \rrbracket \implies P$
by (*case-tac v*) *auto*

lemma *less-fun[elim!]*: $\llbracket v \sqsubseteq \text{Fun } f; \bigwedge f'. \llbracket v = \text{Fun } f'; \text{fset } f' \subseteq \text{fset } f \rrbracket \implies P \rrbracket \implies P$
by (*case-tac v*) *auto*

lemma *less-refl[simp]*: $v \sqsubseteq v$
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  $\llbracket v1 \sqsubseteq v2; v2 \sqsubseteq v3 \rrbracket \implies v1 \sqsubseteq v3$ 
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
  next
  case T-down-closed: assumes vt:  $v \in T \tau \eta$  and vp-v:  $v' \sqsubseteq v$ 
  shows  $v' \in T \tau \eta$ 
  using vt vp-v
proof (induction  $\tau$  arbitrary: v v'  $\eta$ )
  case (TVar x v v'  $\eta$ )
  then show ?case
    apply simp apply (case-tac x < length  $\eta$ )
    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  $\tau1 \tau2$ )
      then show ?case apply simp apply clarify apply (rule-tac x=f' in exI) apply fastforce done
    next
      case (Forall  $\tau v v' \eta$ )
      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
  qed

```

lemma wrong-not-in-T: Wrong $\notin T \tau \eta$

by (induction τ) auto

lemma fun-app: assumes $vmn: V \subseteq T (m \rightarrow n) \eta$ and $v2s: V' \subseteq T m \eta$
 shows apply-fun $V V' \subseteq T n \eta$
 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 $\forall v1 v2'.$
 $(v1, v2') \in \text{fset } x2 \longrightarrow v1 \in T m \eta \longrightarrow (\exists v2. v2 \in T n \eta \wedge v2' \sqsubseteq v2))$)
 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: $\{v. \exists v'. v' \in T \sigma (\eta) \wedge v \sqsubseteq v' \wedge v \neq \text{Wrong}\} = T \sigma \eta$
 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: $T \tau (\eta1 @ (T \sigma (\eta1 @ \eta2)) \# \eta2) = T (\text{subst } (\text{length } \eta1) \sigma \tau) (\eta1 @ \eta2)$

proof (induction τ arbitrary: $\sigma \eta1 \eta2$)

case (TVar x)

then show ?case

apply (case-tac $\text{length } \eta1 = x$) apply simp using T-eta apply blast
 apply (case-tac $\text{length } \eta1 < x$) apply (subgoal-tac $\exists x'. x = \text{Suc } x'$) 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 $T (\text{Forall } \tau) (\eta1 @ T \sigma (\eta1 @ \eta2) \# \eta2) =$

$T (\text{subst } (\text{length } \eta1) \sigma (\text{Forall } \tau)) (\eta1 @ \eta2)$

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 = $\text{length } \eta1$ and ?R1 = $V \# \eta1$ and ?s = $\text{shift } (\text{Suc } 0) 0 \sigma$

assume 1: $v' \in T \tau (V \# (\eta1 @ T \sigma (\eta1 @ \eta2)) \# \eta2)$

from 1 have $a: v' \in T \tau (?R1 @ T \sigma (\eta1 @ \eta2) \# \eta2)$ **by simp**
have $b: T \sigma (\eta1 @ \eta2) = T ?s (V \# (\eta1 @ \eta2))$ **by (rule shift-cons-preserves-T)**
from a b have $c: v' \in T \tau (?R1 @ T ?s (?R1 @ \eta2) \# \eta2)$ **by simp**
from Forall[of ?R1 ?s \eta2] have 2: $T \tau (?R1 @ T ?s (?R1 @ \eta2) \# \eta2) =$
 $T (subst (length ?R1) ?s \tau) (?R1 @ \eta2)$ **by simp**
from c 2 show $v' \in T (subst (Suc ?L1) ?s \tau) (V \# (\eta1 @ \eta2))$ **by simp**
next
fix $x v' V$
let $?L1 = length \eta1$ **and** $?R1 = V \# \eta1$ **and** $?s = shift (Suc 0) 0 \sigma$
assume 1: $v' \in T (subst (Suc (length \eta1)) (shift (Suc 0) 0 \sigma) \tau) (V \# \eta1 @ \eta2)$
from Forall[of ?R1 ?s \eta2] have 2: $T \tau (?R1 @ T ?s (?R1 @ \eta2) \# \eta2) =$
 $T (subst (length ?R1) ?s \tau) (?R1 @ \eta2)$ **by simp**
from 1 2 have 3: $v' \in T \tau (?R1 @ T ?s (?R1 @ \eta2) \# \eta2)$ **by simp**
have $b: T \sigma (\eta1 @ \eta2) = T ?s (V \# (\eta1 @ \eta2))$ **by (rule shift-cons-preserves-T)**
from 3 b have $a: v' \in T \tau (?R1 @ T \sigma (\eta1 @ \eta2) \# \eta2)$ **by simp**
from this show $v' \in T \tau (V \# \eta1 @ T \sigma (\eta1 @ \eta2) \# \eta2)$ **by simp**
qed
qed

lemma iterate-sound:
assumes $it: iterate Ee \rho v$
and $IH: \forall v. v \in T (\sigma \rightarrow \tau) \eta \longrightarrow Ee (v \# \rho) \subseteq T (\sigma \rightarrow \tau) \eta$
shows $v \in T (\sigma \rightarrow \tau) \eta$ **using it IH**
proof (induction rule: iterate.induct)
case (iterate-none Ee \rho)
then show ?case by auto
next
case (iterate-again Ee \rho f f')
from iterate-again have f-st: $f \in T (\sigma \rightarrow \tau) \eta$ **by blast**
from iterate-again f-st have $Ee (f \# \rho) \subseteq T (\sigma \rightarrow \tau) \eta$ **by blast**
from this iterate-again show ?case by auto
qed

theorem welltyped-dont-go-wrong:
assumes $wte: \Gamma \vdash e : \tau$ **and** $wfr: \vdash \rho, \eta : \Gamma$
shows $E e \rho \subseteq T \tau \eta$
using wte wfr
proof (induction \Gamma e \tau arbitrary: \rho \eta rule: well-typed.induct)
case (wtnat \Gamma n \rho \eta)
then show ?case by auto
next
case (wtvar \Gamma n \tau \rho \eta)
from wtvar obtain v where $lx: \rho ! n = v$ **and** $vt: v \in T \tau \eta$ **using lookup-wfenv by blast**
from lx vt show ?case apply auto using T-down-closed[of \rho ! n \tau \eta] by blast
next
case (wtapp \Gamma e \sigma \tau e' \rho \eta)
from wtapp have $Ee: E e \rho \subseteq T (\sigma \rightarrow \tau) \eta$ **by blast**
from wtapp have $Eep: E e' \rho \subseteq T \sigma \eta$ **by blast**
from Ee Eep show ?case using fun-app by simp
next
case (wtlam \sigma \Gamma e \tau \rho \eta)
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 $v1-T: v1 \in T \sigma \eta$ **and** $v2-E: v2 \in E e (v1 \# \rho)$ **and** $v2p-v2: v2' \sqsubseteq v2$
let $?r = v1 \# \rho$
from wtlam(3) v1-T have 1: $\vdash v1 \# \rho, \eta : push-ty \sigma \Gamma$ **by blast**
from wtlam(2) 1 have IH: $E e (v1 \# \rho) \subseteq T \tau \eta$ **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
qed
next
assume vp: v' = Abs None
from vp x show x ∈ T (subst 0 σ τ) η by simp
qed
next

```

```

    from IH show {v. v = Wrong ∧ Wrong ∈ E e ρ} ⊆ T (subst 0 σ τ) η
    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 ⇒ nat ⇒ nat exp exp* | *EIf exp exp exp*
 | *EPair exp exp* | *EFst exp* | *ESnd exp*
 | *ERef exp* | *ERead exp* | *EWrite exp exp*

17.1 Denotations (values)

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

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

type-synonym *store* = *func*

inductive *val-le* :: *val* ⇒ *val* ⇒ *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!*]: $t1 \sqsubseteq t2 \implies (VFun\ t1) \sqsubseteq (VFun\ t2)$ |
vpair-le[*intro!*]: $\llbracket v1 \sqsubseteq v1'; v2 \sqsubseteq v2' \rrbracket \implies (VPair\ v1\ v2) \sqsubseteq (VPair\ v1'\ v2')$

primrec *vsize* :: *val* ⇒ *nat* **where**

vsize $(VNat\ n) = 1$ |
vsize $(VFun\ t) = 1 + \text{ffold } (\lambda((- , v), (- , u)). \lambda r. v + u + r) 0$
 $(\text{fimage } (\text{map-prod } (\lambda v. (v, \text{vsize } v)) (\lambda v. (v, \text{vsize } v))) t)$ |
vsize $(VPair\ v1\ v2) = 1 + \text{vsize } v1 + \text{vsize } v2$ |
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 $m\ f\ \mu1 \equiv \{ (v, \mu3). \exists v' \mu2. (v', \mu2) \in m\ \mu1 \wedge (v, \mu3) \in f\ v' \mu2 \}$

declare *bind-def*[*simp*]

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

translations $P \leftarrow E; F \equiv CONST\ \text{bind } E\ (\lambda P. F)$

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

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

choose $S\ \mu \equiv \{(a, \mu1). a \in S \wedge \mu1 = \mu\}$

declare *choose-def*[*simp*]

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

definition $zero :: 'a M$ **where**
 $zero\ \mu \equiv \{ \}$
declare $zero-def[simp]$

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

syntax $-errset-bind :: [pttrns, val\ M, val] \Rightarrow 'c ((- := -; / -) 0)$
translations $P := E; F \Rightarrow CONST\ err-bind\ E\ (\lambda P. F)$

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

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

definition $put-store :: store \Rightarrow unit\ M$ **where**
 $put-store\ \mu \equiv \lambda -. \{ ((), \mu) \}$
declare $put-store-def[simp]$

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

definition $run :: store \Rightarrow val\ M \Rightarrow (val \times store)\ set$ **where**
 $run\ \sigma\ m \equiv m\ \sigma$
declare $run-def[simp]$

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

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

17.3 Denotational semantics

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

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

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

```

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

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

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

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

type-synonym env = val list

fun E :: exp ⇒ env ⇒ val M where
  Enat: E (ENat n) ρ = return (VNat n) |
  Evar: E (EVar n) ρ = (if n < length ρ then down (ρ!n) else return Wrong) |
  Elam: E (ELam A e) ρ = (L ← vals;
    t ← mapM L (λ v. (μ ← tables; (v',μ') ← choose (run μ (E e (v#ρ))));
    return (VPair v (VFun μ),VPair v' (VFun μ')));
    return (VFun t)) |
  Eapp: E (EApp e1 e2) ρ = apply-fun (E e1 ρ) (E e2 ρ) |
  Eprim: E (EPrim f e1 e2) ρ = (v1 := E e1 ρ; v2 := E e2 ρ;
    case (v1, v2) of (VNat n1,VNat n2) ⇒ return (VNat (f n1 n2))
    | - ⇒ return Wrong) |
  Eif: E (EIf e1 e2 e3) ρ = (v1 := E e1 ρ; case v1 of VNat n ⇒ (if n = 0 then E e3 ρ else E e2 ρ)
    | - ⇒ return Wrong) |
  Epair: E (EPair e1 e2) ρ = (v1 := E e1 ρ; v2 := E e2 ρ; return (VPair v1 v2)) |
  Efst: E (EFst e) ρ = (v:=E e ρ; case v of VPair v1 v2 ⇒ return v1 | - ⇒ return Wrong) |
  Esnd: E (ESnd e) ρ = (v:=E e ρ; case v of VPair v1 v2 ⇒ return v2 | - ⇒ return Wrong) |
  Eref: E (ERef e) ρ = (v:=E e ρ; μ ← get-store; a ← choose UNIV;
    if a ∈ sdom μ then zero
    else (- ← put-store (finsert (VAddr a,v) μ);
    return (VAddr a))) |
  Eread: E (ERead e) ρ = (v := E e ρ; case v of VAddr a ⇒ read a | - ⇒ return Wrong) |
  Ewrite: E (EWrite e1 e2) ρ = (v1 := E e1 ρ; v2 := E e2 ρ;
    case v1 of VAddr a ⇒ update a v2 | - ⇒ return Wrong)

end
theory MutableRefProps
  imports MutableRef
begin

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

```

le-fun-any-inv[elim!]: $VFun\ t \sqsubseteq v$ **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*: $\llbracket v1 \sqsubseteq v2; v2 \sqsubseteq v3 \rrbracket \implies v1 \sqsubseteq v3$
by (*induction v2 arbitrary: v1 v3*) *blast+*

proposition *val-le-antisymm*: $\llbracket v1 \sqsubseteq v2; v2 \sqsubseteq v1 \rrbracket \implies v1 = v2$
by (*induction v1 arbitrary: v2*) *blast+*

end