Lazy Evaluation: From natural semantics to a machine-checked compiler transformation

Joachim Breitner
Programming Paradigms Group
Karlsruhe Institute for Technology
breitner@kit.edu

June 29, 2016

This document contains a rendering of the formal theories corresponding to doctorla thesis "Lazy Evaluation: From natural semantics to a machine-checked compiler transformation" by Joachim Breitner. See http://www.joachim-breitner.de/thesis for more details.

These theories are also contained in the Archive of Formal Proofs; please see and cite http://afp.sourceforge.net/entries/Launchbury.shtml resp. http://afp.sourceforge.net/entries/Call_Arity.shtml.

Contents

1 AList-Utils.tex

theory *AList−Utils*
imports *Main* ~~/*src*/*HOL*/*Library*/*AList*
begin
declare *implies-True-equals* [*simp*] *False-implies-equals*[*simp*]

We want to have *delete* and *update* back in the namespace.

abbreviation *delete* **where** *delete* ≡ *AList.delete*
abbreviation *update* **where** *update* ≡ *AList.update*
abbreviation *restrictA* **where** *restrictA* ≡ *AList.restrict*
abbreviation *clearjunk* **where** *clearjunk* ≡ *AList.clearjunk*

lemmas *restrict-eq* = *AList.restrict-eq*
  and *delete-eq* = *AList.delete-eq*

lemma *restrictA-append*: *restrictA S* (*a*@*b*) = *restrictA S a* @ *restrictA S b*
  unfolding *restrict-eq* **by** (*rule filter-append*)

lemma *length-restrictA-le*: *length* (*restrictA S a*) ≤ *length a*
  by (*metis length-filter-le restrict-eq*)

1.1 The domain of an associative list

definition *domA*
  where *domA h* = *fst* ‘ *set h*

lemma *domA-append*[*simp*]:*domA* (*a* @ *b*) = *domA a* ∪ *domA b*
  and [*simp*]:*domA* ((*v*,*e*) # *h*) = *insert v* (*domA h*)
  and [*simp*]:*domA* (*p* # *h*) = *insert* (*fst p*) (*domA h*)
  and [*simp*]:*domA* [] = {}
  by (*auto simp add*: *domA-def*)

lemma *domA-from-set*:
  (*x*, *e*) ∈ *set h* ⟹ *x* ∈ *domA h*
by (*induct h*, *auto*)

lemma *finite-domA*[*simp*]:
  finite (*domA* Γ)
  by (*auto simp add*: *domA-def*)

lemma *domA-delete*[*simp*]:
  domA (*delete x* Γ) = *domA* Γ − {*x*}
  by (*induct* Γ) *auto*

lemma *domA-restrictA*[*simp*]:

$domA\ (restrictA\ S\ \Gamma) = domA\ \Gamma \cap S$
by $(induct\ \Gamma)$ *auto*

lemma *delete-not-domA*[*simp*]:
$x \notin domA\ \Gamma \Longrightarrow\ delete\ x\ \Gamma = \Gamma$
by $(induct\ \Gamma)$ *auto*

lemma *deleted-not-domA*: $x \notin domA\ (delete\ x\ \Gamma)$
by $(induct\ \Gamma)$ *auto*

lemma *dom-map-of-conv-domA*:
$dom\ (map\text{-}of\ \Gamma) = domA\ \Gamma$
by $(induct\ \Gamma)$ (*auto simp add*: *dom-if*)

lemma *domA-map-of-Some-the*:
$x \in domA\ \Gamma \Longrightarrow map\text{-}of\ \Gamma\ x = Some\ (the\ (map\text{-}of\ \Gamma\ x))$
by $(induct\ \Gamma)$ (*auto simp add*: *dom-if*)

lemma *domA-clearjunk*[*simp*]: $domA\ (clearjunk\ \Gamma) = domA\ \Gamma$
unfolding *domA-def* **using** *dom-clearjunk*.

lemma *the-map-option-domA*[*simp*]: $x \in domA\ \Gamma \Longrightarrow the\ (map\text{-}option\ f\ (map\text{-}of\ \Gamma\ x)) = f\ (the\ (map\text{-}of\ \Gamma\ x))$
by $(induction\ \Gamma)$ *auto*

lemma *map-of-domAD*: $map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow x \in domA\ \Gamma$
using *dom-map-of-conv-domA* **by** *fastforce*

lemma *restrictA-noop*: $domA\ \Gamma \subseteq S \Longrightarrow restrictA\ S\ \Gamma = \Gamma$
unfolding *restrict-eq* **by** $(induction\ \Gamma)$ *auto*

lemma *restrictA-cong*:
$(\bigwedge x.\ x \in domA\ m1 \Longrightarrow x \in V \longleftrightarrow x \in V') \Longrightarrow m1 = m2 \Longrightarrow restrictA\ V\ m1 = restrictA\ V'\ m2$
unfolding *restrict-eq* **by** (*induction m1 arbitrary*: *m2*) *auto*

1.2 Other lemmas about associative lists

lemma *delete-set-none*: $(map\text{-}of\ l)(x := None) = map\text{-}of\ (delete\ x\ l)$
proof (*induct l*)
case *Nil* **thus** *?case* **by** *simp*
case (*Cons l ls*)
from *this*[*symmetric*]
show *?case*
by (*cases fst* $l = x$) *auto*
qed

lemma *list-size-delete*[*simp*]: $size\text{-}list\ size\ (delete\ x\ l) < Suc\ (size\text{-}list\ size\ l)$
by (*induct l*) *auto*

lemma *delete-append*[*simp*]: *delete* x (*l1* @ *l2*) = *delete* x *l1* @ *delete* x *l2*
unfolding *AList.delete-eq* **by** *simp*

lemma *map-of-delete-insert*:
assumes *map-of* Γ x = *Some* e
shows *map-of* ((x,e) # *delete* x Γ) = *map-of* Γ
using *assms* **by** (*induct* Γ) (*auto split*:*prod.split*)

lemma *map-of-delete-iff* [*simp*]: *map-of* (*delete* x Γ) *xa* = *Some* e $\longleftrightarrow$ (*map-of* Γ *xa* = *Some* e) $\wedge$ *xa* $\neq$ x
by (*metis delete-conv fun-upd-same map-of-delete option.distinct*(*1*))

lemma *map-add-domA*[*simp*]:
$x \in$ *domA* $\Gamma \Longrightarrow$ (*map-of* Δ ++ *map-of* Γ) x = *map-of* Γ x
$x \notin$ *domA* $\Gamma \Longrightarrow$ (*map-of* Δ ++ *map-of* Γ) x = *map-of* Δ x
apply (*metis dom-map-of-conv-domA map-add-dom-app-simps*(*1*))
apply (*metis dom-map-of-conv-domA map-add-dom-app-simps*(*3*))
done

lemma *map-of-empty-iff1* [*simp*]: *map-of* Γ = *empty* $\longleftrightarrow$ Γ = []
by (*cases* Γ) *auto*

lemma *map-of-empty-iff2* [*simp*]: *empty* = *map-of* Γ $\longleftrightarrow$ Γ = []
apply (*subst eq-commute*)
by (*rule map-of-empty-iff1*)

lemma *set-delete-subset*: *set* (*delete* k *al*) $\subseteq$ *set al*
by (*auto simp add*: *delete-eq*)

lemma *dom-delete-subset*: *snd* ‘ *set* (*delete* k *al*) $\subseteq$ *snd* ‘ *set al*
by (*auto simp add*: *delete-eq*)

lemma *map-ran-cong*[*fundef-cong*]:
$[\![$ $\bigwedge$ x . $x \in$ *set m1* $\Longrightarrow$ *f1* (*fst* x) (*snd* x) = *f2* (*fst* x) (*snd* x) ; *m1* = *m2* $]\!]$
$\Longrightarrow$ *map-ran f1 m1* = *map-ran f2 m2*
by (*induction m1 arbitrary*: *m2*) *auto*

lemma *domA-map-ran*[*simp*]: *domA* (*map-ran f m*) = *domA m*
unfolding *domA-def* **by** (*rule dom-map-ran*)

lemma *map-ran-delete*:
map-ran f (*delete* x Γ) = *delete* x (*map-ran f* Γ)
by (*induction* Γ) *auto*

lemma *map-ran-restrictA*:
map-ran f (*restrictA V* Γ) = *restrictA V* (*map-ran f* Γ)

by $(\mathit{induction}\ \Gamma)$ *auto*

lemma *map-ran-append*:
map-ran f $(\Gamma @\Delta)$ = *map-ran f* Γ @ *map-ran f* Δ
by $(\mathit{induction}\ \Gamma)$ *auto*

1.3 Syntax for map comprehensions

definition *mapCollect* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'c\ set$
where *mapCollect f m* = $\{f\ k\ v \mid k\ v\ .\ m\ k = \mathit{Some}\ v\}$

syntax
-MapCollect :: $'c \Rightarrow \mathit{pttrn} => \mathit{pttrn} \Rightarrow 'a \rightharpoonup 'b => 'c\ set$ $((\mathit{1}\{\text{-} \mid/\text{-}/\mapsto/\text{-}/\in/\text{-}/\}))$
translations
$\{e \mid k \mapsto v \in m\}$ == *CONST mapCollect* $(\lambda k\ v.\ e)\ m$

lemma *mapCollect-empty*[*simp*]: $\{f\ k\ v \mid k \mapsto v \in \mathit{empty}\} = \{\}$
unfolding *mapCollect-def* **by** *simp*

lemma *mapCollect-const*[*simp*]:
$m \neq \mathit{empty} \Longrightarrow \{e \mid k\mapsto v\in m\} = \{e\}$
unfolding *mapCollect-def* **by** *auto*

lemma *mapCollect-cong*[*fundef-cong*]:
$(\bigwedge k\ v.\ m1\ k = \mathit{Some}\ v \Longrightarrow f1\ k\ v = f2\ k\ v) \Longrightarrow m1 = m2 \Longrightarrow \mathit{mapCollect}\ f1\ m1 = \mathit{mapCollect}$
$f2\ m2$
unfolding *mapCollect-def* **by** *force*

lemma *mapCollectE*[*elim!*]:
assumes $x \in \{f\ k\ v \mid k \mapsto v \in m\}$
obtains *k v* **where** $m\ k = \mathit{Some}\ v$ **and** $x = f\ k\ v$
using *assms* **by** (*auto simp add*: *mapCollect-def*)

lemma *mapCollectI*[*intro*]:
assumes $m\ k = \mathit{Some}\ v$
shows $f\ k\ v \in \{f\ k\ v \mid k \mapsto v \in m\}$
using *assms* **by** (*auto simp add*: *mapCollect-def*)

lemma *ball-mapCollect*[*simp*]:
$(\forall\ x \in \{f\ k\ v \mid k \mapsto v \in m\}.\ P\ x) \longleftrightarrow (\forall\ k\ v.\ m\ k = \mathit{Some}\ v \longrightarrow P\ (f\ k\ v))$
by (*auto simp add*: *mapCollect-def*)

lemma *image-mapCollect*[*simp*]:
$g\ `\ \{f\ k\ v \mid k \mapsto v \in m\} = \{\ g\ (f\ k\ v) \mid k \mapsto v \in m\}$
by (*auto simp add*: *mapCollect-def*)

lemma *mapCollect-map-upd*[*simp*]:
$\mathit{mapCollect}\ f\ (m(k\mapsto v)) = \mathit{insert}\ (f\ k\ v)\ (\mathit{mapCollect}\ f\ (m(k := \mathit{None})))$

unfolding *mapCollect-def* **by** *auto*

definition *mapCollectFilter* :: $('a \Rightarrow 'b \Rightarrow (bool \times 'c)) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'c\ set$
  where *mapCollectFilter f m* = $\{snd\ (f\ k\ v) \mid k\ v\ .\ m\ k = Some\ v \wedge fst\ (f\ k\ v)\}$

syntax
 -MapCollectFilter :: $'c \Rightarrow pttrn \Rightarrow pttrn \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool \Rightarrow 'c\ set$ ((*1*{- |/-/↦/-/∈/-/./ -}))
translations
  $\{e \mid k{\mapsto}v \in m\ .\ P\ \}$ == *CONST mapCollectFilter* $(\lambda k\ v.\ (P,e))\ m$

lemma *mapCollectFilter-const-False*[*simp*]:
  $\{e \mid k{\mapsto}v \in m\ .\ False\ \} = \{\}$
  unfolding *mapCollect-def mapCollectFilter-def* **by** *simp*

lemma *mapCollectFilter-const-True*[*simp*]:
  $\{e \mid k{\mapsto}v \in m\ .\ True\ \} = \{e \mid k{\mapsto}v \in m\}$
  unfolding *mapCollect-def mapCollectFilter-def* **by** *simp*

end

2 HOLCF-Join.tex

theory *HOLCF−Join*
imports ~~/*src*/*HOL*/*HOLCF*/*HOLCF*
begin

2.1 Binary Joins and compatibility

context *cpo*
begin
definition *join* :: $'a$ => $'a$ => $'a$ (**infixl** ⊔ *80*) **where**
  $x \sqcup y = (if\ \exists\ z.\ \{x,\ y\}$ <<| $z\ then\ lub\ \{x,\ y\}\ else\ x)$

definition *compatible* :: $'a \Rightarrow 'a \Rightarrow bool$
  where *compatible x y* = $(\exists\ z.\ \{x,\ y\}$ <<| $z)$

lemma *compatibleI*:
  assumes $x \sqsubseteq z$
  assumes $y \sqsubseteq z$
  assumes $\bigwedge a.\ [\![\ x \sqsubseteq a\ ;\ y \sqsubseteq a\ ]\!] \Longrightarrow z \sqsubseteq a$
  shows *compatible x y*
proof−
  from *assms*
  have $\{x,y\}$ <<| z

by (*auto intro*: *is-lubI*)
thus *?thesis* **unfolding** *compatible-def* **by** (*metis*)
qed

lemma *is-joinI*:
assumes $x \sqsubseteq z$
assumes $y \sqsubseteq z$
assumes $\bigwedge a.\ [\![x \sqsubseteq a ; y \sqsubseteq a]\!] \Longrightarrow z \sqsubseteq a$
shows $x \sqcup y = z$
proof−
from *assms*
have $\{x,y\}$ <<| z
by (*auto intro*: *is-lubI*)
thus *?thesis* **unfolding** *join-def* **by** (*metis lub-eqI*)
qed

lemma *is-join-and-compatible*:
assumes $x \sqsubseteq z$
assumes $y \sqsubseteq z$
assumes $\bigwedge a.\ [\![x \sqsubseteq a ; y \sqsubseteq a]\!] \Longrightarrow z \sqsubseteq a$
shows *compatible* x y $\wedge$ $x \sqcup y = z$
by (*metis compatibleI is-joinI assms*)

lemma *compatible-sym*: *compatible* x y $\Longrightarrow$ *compatible* y x
unfolding *compatible-def* **by** (*metis insert-commute*)

lemma *compatible-sym-iff*: *compatible* x y $\longleftrightarrow$ *compatible* y x
unfolding *compatible-def* **by** (*metis insert-commute*)

lemma *join-above1*: *compatible* x y $\Longrightarrow$ $x \sqsubseteq x \sqcup y$
unfolding *compatible-def join-def*
apply *auto*
by (*metis is-lubD1 is-ub-insert lub-eqI*)

lemma *join-above2*: *compatible* x y $\Longrightarrow$ $y \sqsubseteq x \sqcup y$
unfolding *compatible-def join-def*
apply *auto*
by (*metis is-lubD1 is-ub-insert lub-eqI*)

lemma *larger-is-join1*: $y \sqsubseteq x \Longrightarrow x \sqcup y = x$
unfolding *join-def*
by (*metis doubleton-eq-iff lub-bin*)

lemma *larger-is-join2*: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
unfolding *join-def*
by (*metis is-lub-bin lub-bin*)

lemma *join-self*[*simp*]: $x \sqcup x = x$
unfolding *join-def* **by** *auto*

end

lemma *join-commute*: *compatible* $x\ y \Longrightarrow x \sqcup y = y \sqcup x$
unfolding *compatible-def* **unfolding** *join-def* **by** (*metis insert-commute*)

lemma *lub-is-join*:
$\{x,\ y\}$ <<| $z \Longrightarrow x \sqcup y = z$
unfolding *join-def* **by** (*metis lub-eqI*)

lemma *compatible-refl*[*simp*]: *compatible* $x\ x$
by (*rule compatibleI*[*OF below-refl below-refl*])

lemma *join-mono*:
assumes *compatible* $a\ b$
and *compatible* $c\ d$
and $a \sqsubseteq c$
and $b \sqsubseteq d$
shows $a \sqcup b \sqsubseteq c \sqcup d$
proof−
from *assms* **obtain** $x\ y$ **where** $\{a,\ b\}$ <<| x $\{c,\ d\}$ <<| y **unfolding** *compatible-def* **by** *auto*
with *assms* **have** $a \sqsubseteq y$ $b \sqsubseteq y$ **by** (*metis below.r-trans is-lubD1 is-ub-insert*)+
with ‹$\{a,\ b\}$ <<| x› **have** $x \sqsubseteq y$ **by** (*metis is-lub-below-iff is-lub-singleton is-ub-insert*)
moreover
from ‹$\{a,\ b\}$ <<| x› ‹$\{c,\ d\}$ <<| y› **have** $a \sqcup b = x$ $c \sqcup d = y$ **by** (*metis lub-is-join*)+
ultimately
show *?thesis* **by** *simp*
qed

lemma
assumes *compatible* $x\ y$
shows *join-above1*: $x \sqsubseteq x \sqcup y$ **and** *join-above2*: $y \sqsubseteq x \sqcup y$
proof−
from *assms* **obtain** z **where** $\{x,y\}$ <<| z **unfolding** *compatible-def* **by** *auto*
hence $x \sqcup y = z$ **and** $x \sqsubseteq z$ **and** $y \sqsubseteq z$ **apply** (*auto intro*: *lub-is-join*) **by** (*metis is-lubD1 is-ub-insert*)+
thus $x \sqsubseteq x \sqcup y$ **and** $y \sqsubseteq x \sqcup y$ **by** *simp-all*
qed

lemma
assumes *compatible* $x\ y$
shows *compatible-above1*: *compatible* $x\ (x \sqcup y)$ **and** *compatible-above2*: *compatible* $y\ (x \sqcup y)$
proof−
from *assms* **obtain** z **where** $\{x,y\}$ <<| z **unfolding** *compatible-def* **by** *auto*
hence $x \sqcup y = z$ **and** $x \sqsubseteq z$ **and** $y \sqsubseteq z$ **apply** (*auto intro*: *lub-is-join*) **by** (*metis is-lubD1 is-ub-insert*)+
thus *compatible* $x\ (x \sqcup y)$ **and** *compatible* $y\ (x \sqcup y)$ **by** (*metis below.r-refl compatibleI*)+
qed

lemma *join-below*:
  assumes *compatible x y*
  and $x \sqsubseteq a$ **and** $y \sqsubseteq a$
  shows $x \sqcup y \sqsubseteq a$
proof −
  from *assms* **obtain** *z* **where** *z*: $\{x,y\}$ <<| *z* **unfolding** *compatible-def* **by** *auto*
  with *assms* **have** $z \sqsubseteq a$ **by** (*metis is-lub-below-iff is-ub-empty is-ub-insert*)
  moreover
  from *z* **have** $x \sqcup y = z$ **by** (*rule lub-is-join*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *join-below-iff*:
  assumes *compatible x y*
  shows $x \sqcup y \sqsubseteq a \longleftrightarrow (x \sqsubseteq a \land y \sqsubseteq a)$
  by (*metis assms below-trans cpo-class.join-above1 cpo-class.join-above2 join-below*)

lemma *join-assoc*:
  assumes *compatible x y*
  assumes *compatible x* $(y \sqcup z)$
  assumes *compatible y z*
  shows $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
  apply (*rule is-joinI*)
  apply (*rule join-mono*[*OF assms*(*1*) *assms*(*2*) *below-refl join-above1*[*OF assms*(*3*)]])
  apply (*rule below-trans*[*OF join-above2*[*OF assms*(*3*)] *join-above2*[*OF assms*(*2*)]])
  apply (*rule join-below*[*OF assms*(*2*)])
  apply (*erule rev-below-trans*)
  apply (*rule join-above1*[*OF assms*(*1*)])
  apply (*rule join-below*[*OF assms*(*3*)])
  apply (*erule rev-below-trans*)
  apply (*rule join-above2*[*OF assms*(*1*)])
  apply *assumption*
  done

lemma *join-idem*[*simp*]: *compatible x y* $\Longrightarrow x \sqcup (x \sqcup y) = x \sqcup y$
  apply (*subst join-assoc*[*symmetric*])
  apply (*rule compatible-refl*)
  apply (*erule compatible-above1*)
  apply *assumption*
  apply (*subst join-self*)
  apply *rule*
  done

lemma *join-bottom*[*simp*]: $x \sqcup \bot = x$ $\bot \sqcup x = x$
  by (*auto intro*: *is-joinI*)

lemma *compatible-adm2*:
  shows *adm* (λ *y*. *compatible x y*)

proof(*rule admI*)
  fix *Y*
  assume *c*: *chain Y* **and** $\forall$ *i*. *compatible x* (*Y i*)
  hence *a*: $\bigwedge$ *i*. *compatible x* (*Y i*) **by** *auto*
  show *compatible x* ($\bigsqcup$ *i*. *Y i*)
  proof(*rule compatibleI*)
    have *c2*: *chain* (λ*i*. *x* $\sqcup$ *Y i*)
      apply (*rule chainI*)
      apply (*rule join-mono*[*OF a a below-refl chainE*[*OF* ‹*chain Y*›]])
      done
    show *x* $\sqsubseteq$ ($\bigsqcup$ *i*. *x* $\sqcup$ *Y i*)
      by (*auto intro*: *admD*[*OF - c2*] *join-above1*[*OF a*])
    show ($\bigsqcup$ *i*. *Y i*) $\sqsubseteq$ ($\bigsqcup$ *i*. *x* $\sqcup$ *Y i*)
      by (*auto intro*: *admD*[*OF - c*] *below-lub*[*OF c2 join-above2*[*OF a*]])
    fix *a*
    assume *x* $\sqsubseteq$ *a* **and** ($\bigsqcup$ *i*. *Y i*) $\sqsubseteq$ *a*
    show ($\bigsqcup$ *i*. *x* $\sqcup$ *Y i*) $\sqsubseteq$ *a*
      apply (*rule lub-below*[*OF c2*])
      apply (*rule join-below*[*OF a* ‹*x* $\sqsubseteq$ *a*›])
      apply (*rule below-trans*[*OF is-ub-thelub*[*OF c*] ‹($\bigsqcup$ *i*. *Y i*) $\sqsubseteq$ *a*›])
      done
  qed
qed

lemma *compatible-adm1*: *adm* (λ *x*. *compatible x y*)
  by (*subst compatible-sym-iff*, *rule compatible-adm2*)

lemma *join-cont1*:
  assumes *chain Y*
  assumes *compat*: $\bigwedge$ *i*. *compatible* (*Y i*) *y*
  shows ($\bigsqcup$*i*. *Y i*) $\sqcup$ *y* = ($\bigsqcup$ *i*. *Y i* $\sqcup$ *y*)
proof−
  have *c*: *chain* (λ*i*. *Y i* $\sqcup$ *y*)
    apply (*rule chainI*)
    apply (*rule join-mono*[*OF compat compat chainE*[*OF* ‹*chain Y*›] *below-refl*])
    done

  show *?thesis*
    apply (*rule is-joinI*)
    apply (*rule lub-mono*[*OF* ‹*chain Y*› *c join-above1*[*OF compat*]])
    apply (*rule below-lub*[*OF c join-above2*[*OF compat*]])
    apply (*rule lub-below*[*OF c*])
    apply (*rule join-below*[*OF compat*])
    apply (*metis lub-below-iff*[*OF* ‹*chain Y*›])
    apply *assumption*
    done
qed

lemma *join-cont2*:

assumes *chain Y*
assumes *compat*: $\bigwedge$ *i. compatible x* (*Y i*)
shows $x \sqcup (\bigsqcup i.\ Y\ i) = (\bigsqcup\ i.\ x \sqcup Y\ i)$
proof −
have *c*: *chain* ($\lambda i.\ x \sqcup Y\ i$)
apply (*rule chainI*)
apply (*rule join-mono*[*OF compat compat below-refl chainE*[*OF* ‹*chain Y*›]])
done

show *?thesis*
apply (*rule is-joinI*)
apply (*rule below-lub*[*OF c join-above1*[*OF compat*]])
apply (*rule lub-mono*[*OF* ‹*chain Y*› *c join-above2*[*OF compat*]])
apply (*rule lub-below*[*OF c*])
apply (*rule join-below*[*OF compat*])
apply *assumption*
apply (*metis lub-below-iff*[*OF* ‹*chain Y*›])
done
qed

lemma *join-cont12*:
assumes *chain Y* **and** *chain Z*
assumes *compat*: $\bigwedge$ *i j. compatible* (*Y i*) (*Z j*)
shows $(\bigsqcup i.\ Y\ i) \sqcup (\bigsqcup i.\ Z\ i) = (\bigsqcup\ i.\ Y\ i\ \sqcup Z\ i)$
proof −
have $(\bigsqcup i.\ Y\ i) \sqcup (\bigsqcup i.\ Z\ i) = (\bigsqcup i.\ Y\ i \sqcup (\bigsqcup j.\ Z\ j))$
by (*rule join-cont1*[*OF* ‹*chain Y*› *admD*[*OF compatible-adm2* ‹*chain Z*› *compat*]])
also have ... $= (\bigsqcup i\ j.\ Y\ i \sqcup Z\ j)$
by (*subst join-cont2*[*OF* ‹*chain Z*› *compat*], *rule*)
also have ... $= (\bigsqcup i.\ Y\ i \sqcup Z\ i)$
apply (*rule diag-lub*)
apply (*rule chainI*, *rule join-mono*[*OF compat compat chainE*[*OF* ‹*chain Y*›] *below-refl*])
apply (*rule chainI*, *rule join-mono*[*OF compat compat below-refl chainE*[*OF* ‹*chain Z*›]])
done
finally show *?thesis*.
qed

context *pcpo*
begin
lemma *bot-compatible*[*simp*]:
compatible x ⊥ *compatible* ⊥ *x*
unfolding *compatible-def* **by** (*metis insert-commute is-lub-bin minimal*)+
end

end

3 HOLCF-Join-Classes.tex

theory *HOLCF−Join−Classes*
imports *HOLCF−Join*
begin

class *Finite-Join-cpo* = *cpo* +
  assumes *all-compatible*: *compatible x y*

lemmas *join-mono* = *join-mono*[*OF all-compatible all-compatible*]
lemmas *join-above1*[*simp*] = *all-compatible*[*THEN join-above1*]
lemmas *join-above2*[*simp*] = *all-compatible*[*THEN join-above2*]
lemmas *join-below*[*simp*] = *all-compatible*[*THEN join-below*]
lemmas *join-below-iff* = *all-compatible*[*THEN join-below-iff*]
lemmas *join-assoc*[*simp*] = *join-assoc*[*OF all-compatible all-compatible all-compatible*]
lemmas *join-comm*[*simp*] = *all-compatible*[*THEN join-commute*]

lemma *join-lc*[*simp*]: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup (z::'a::Finite\text{-}Join\text{-}cpo))$
  by (*metis join-assoc join-comm*)

lemma *join-cont'*: *chain* $Y \Longrightarrow (\bigsqcup\ i.\ Y\ i) \sqcup y = (\bigsqcup\ i.\ Y\ i \sqcup (y::'a::Finite\text{-}Join\text{-}cpo))$
by (*metis all-compatible join-cont1*)

lemma *join-cont1*:
  fixes y :: *'a* :: *Finite-Join-cpo*
  shows *cont* $(\lambda x.\ (x \sqcup y))$
  apply (*rule contI2*)
  apply (*rule monofunI*)
  apply (*metis below-refl join-mono*)
  apply (*rule eq-imp-below*)
  apply (*rule join-cont'*)
  apply *assumption*
  done

lemma *join-cont2*:
  fixes x :: *'a* :: *Finite-Join-cpo*
  shows *cont* $(\lambda y.\ (x \sqcup y))$
  unfolding *join-comm* **by** (*rule join-cont1*)

lemma *join-cont*[*cont2cont*,*simp*]:*cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *cont* $(\lambda x.\ (f\,x \sqcup (g\,x::'a::Finite\text{-}Join\text{-}cpo)))$
   apply (*rule cont2cont-case-prod*[**where** $g = \lambda\ x.\ (f\ x,\ g\ x)$ **and** $f = \lambda\ p\ x\ y\ .\ x \sqcup y$, *simplified*])
  apply (*rule join-cont2*)
  apply (*metis cont2cont-Pair*)
  done

instantiation *fun* :: (*type*, *Finite-Join-cpo*) *Finite-Join-cpo*
begin
  definition *fun-join* :: $('a \Rightarrow 'b) \rightarrow ('a \Rightarrow 'b) \rightarrow ('a \Rightarrow 'b)$

where *fun-join* = (Λ *f g* . (λ *x*. (*f x*) ⊔ (*g x*)))
lemma [*simp*]: (*fun-join*·*f*·*g*) *x* = (*f x*) ⊔ (*g x*)
unfolding *fun-join-def*
apply (*subst beta-cfun*, *intro cont2cont cont2cont-lambda cont2cont-fun*)+
..
instance
apply *standard*
proof(*intro compatibleI exI conjI strip*)
fix *x y*
show *x* ⊑ *fun-join*·*x*·*y* **by** (*auto simp add*: *fun-below-iff*)
show *y* ⊑ *fun-join*·*x*·*y* **by** (*auto simp add*: *fun-below-iff*)
fix *z*
assume *x* ⊑ *z* **and** *y* ⊑ *z*
thus *fun-join*·*x*·*y* ⊑ *z* **by** (*auto simp add*: *fun-below-iff*)
qed
end

instantiation *cfun* :: (*cpo*, *Finite-Join-cpo*) *Finite-Join-cpo*
begin
definition *cfun-join* :: (′*a* → ′*b*) → (′*a* → ′*b*) → (′*a* → ′*b*)
where *cfun-join* = (Λ *f g x*. (*f* · *x*) ⊔ (*g* · *x*))
lemma [*simp*]: *cfun-join*·*f*·*g*·*x* = (*f* · *x*) ⊔ (*g* · *x*)
unfolding *cfun-join-def*
apply (*subst beta-cfun*, *intro cont2cont cont2cont-lambda cont2cont-fun*)+
..
instance
apply *standard*
proof(*intro compatibleI exI conjI strip*)
fix *x y*
show *x* ⊑ *cfun-join*·*x*·*y* **by** (*auto simp add*: *cfun-below-iff*)
show *y* ⊑ *cfun-join*·*x*·*y* **by** (*auto simp add*: *cfun-below-iff*)
fix *z*
assume *x* ⊑ *z* **and** *y* ⊑ *z*
thus *cfun-join*·*x*·*y* ⊑ *z* **by** (*auto simp add*: *cfun-below-iff*)
qed
end

lemma *bot-lub*[*simp*]: *S* <<| ⊥ ⟷ *S* ⊆ {⊥}
by (*auto dest*!: *is-lubD1 is-ubD intro*: *is-lubI is-ubI*)

lemma *compatible-up*[*simp*]: *compatible* (*up*·*x*) (*up*·*y*) ⟷ *compatible x y*
proof
assume *compatible* (*up*·*x*) (*up*·*y*)
then obtain *z*′ **where** *z*′: {*up*·*x*,*up*·*y*} <<| *z*′ **unfolding** *compatible-def* **by** *auto*
then obtain *z* **where** {*up*·*x*,*up*·*y*} <<| *up*·*z* **by** (*cases z*′) *auto*
hence {*x*,*y*} <<| *z*
unfolding *is-lub-def*
apply *auto*
by (*metis up-below*)

thus *compatible x y* **unfolding** *compatible-def* **..**
next
assume *compatible x y*
then obtain *z* **where** *z*: $\{x,y\}$ <<| *z* **unfolding** *compatible-def* **by** *auto*
hence $\{up{\cdot}x, up{\cdot}y\}$ <<| $up{\cdot}z$ **unfolding** *is-lub-def*
apply *auto*
by (*metis not-up-less-UU upE up-below*)
thus *compatible* $(up{\cdot}x)$ $(up{\cdot}y)$ **unfolding** *compatible-def* **..**
qed

instance *u* :: (*Finite-Join-cpo*) *Finite-Join-cpo*
proof
fix *x y* :: $'a_\bot$
show *compatible x y*
apply (*cases x*, *simp*)
apply (*cases y*, *simp*)
apply (*simp add*: *all-compatible*)
done
qed

class *is-unit* = **fixes** *unit* **assumes** *is-unit*: $\bigwedge x.\ x = unit$

instantiation *unit* :: *is-unit*
begin

definition *unit* $\equiv$ ()

instance
by *standard auto*

end

instance *lift* :: (*is-unit*) *Finite-Join-cpo*
proof
fix *x y* :: *'a lift*
show *compatible x y*
apply (*cases x*, *simp*)
apply (*cases y*, *simp*)
apply (*simp add*: *all-compatible*)
apply (*subst is-unit*)
apply (*subst is-unit*) **back**
apply *simp*
done
qed

instance *prod* :: (*Finite-Join-cpo*, *Finite-Join-cpo*) *Finite-Join-cpo*
proof
fix *x y* :: $('a \times 'b)$

let *?z* = (*fst* $x \sqcup$ *fst* y, *snd* $x \sqcup$ *snd* y)
show *compatible* x y
proof(*rule compatibleI*)
show $x \sqsubseteq$ *?z* **by** (*cases* x, *auto*)
show $y \sqsubseteq$ *?z* **by** (*cases* y, *auto*)
fix z'
assume $x \sqsubseteq z'$ **and** $y \sqsubseteq z'$ **thus** *?z* $\sqsubseteq z'$
by (*cases* z', *cases* x, *cases* y, *auto*)
qed
qed

lemma *prod-join*:
fixes x y :: *'a::Finite-Join-cpo* × *'b::Finite-Join-cpo*
shows $x \sqcup y$ = (*fst* $x \sqcup$ *fst* y, *snd* $x \sqcup$ *snd* y)
apply (*rule is-joinI*)
apply (*cases* x, *auto*)[*1*]
apply (*cases* y, *auto*)[*1*]
apply (*cases* x, *cases* y, *case-tac* a, *auto*)[*1*]
done

lemma
fixes x y :: *'a::Finite-Join-cpo* × *'b::Finite-Join-cpo*
shows *fst-join*[*simp*]: *fst* ($x \sqcup y$) = *fst* $x \sqcup$ *fst* y
and *snd-join*[*simp*]: *snd* ($x \sqcup y$) = *snd* $x \sqcup$ *snd* y
unfolding *prod-join* **by** *simp-all*

lemma *fun-meet-simp*[*simp*]: ($f \sqcup g$) x = f $x \sqcup$ (g x::*'a::Finite-Join-cpo*)
proof−
have $f \sqcup g = (\lambda\ x.\ f\ x \sqcup g\ x)$
by (*rule is-joinI*)(*auto simp add*: *fun-below-iff*)
thus *?thesis* **by** *simp*
qed

lemma *fun-upd-meet-simp*[*simp*]: ($f \sqcup g$) (x := y) = f (x := y) $\sqcup$ g (x := y::*'a::Finite-Join-cpo*)
by *auto*

lemma *cfun-meet-simp*[*simp*]: ($f \sqcup g$) · x = f · $x \sqcup$ (g · x::*'a::Finite-Join-cpo*)
proof−
have $f \sqcup g = (\Lambda\ x.\ f \cdot x \sqcup g \cdot x)$
by (*rule is-joinI*)(*auto simp add*: *cfun-below-iff*)
thus *?thesis* **by** *simp*
qed

lemma *cfun-join-below*:
fixes f :: (*'a::Finite-Join-cpo*) → (*'b::Finite-Join-cpo*)
shows $f \cdot x \sqcup f \cdot y \sqsubseteq f \cdot (x \sqcup y)$
by (*intro join-below monofun-cfun-arg join-above1 join-above2*)

lemma *join-self-below*[*iff*]:

$x = x \sqcup y \longleftrightarrow y \sqsubseteq (x{::}'a{::}\mathit{Finite\text{-}Join\text{-}cpo})$
$x = y \sqcup x \longleftrightarrow y \sqsubseteq (x{::}'a{::}\mathit{Finite\text{-}Join\text{-}cpo})$
$x \sqcup y = x \longleftrightarrow y \sqsubseteq (x{::}'a{::}\mathit{Finite\text{-}Join\text{-}cpo})$
$y \sqcup x = x \longleftrightarrow y \sqsubseteq (x{::}'a{::}\mathit{Finite\text{-}Join\text{-}cpo})$
$x \sqcup y \sqsubseteq x \longleftrightarrow y \sqsubseteq (x{::}'a{::}\mathit{Finite\text{-}Join\text{-}cpo})$
$y \sqcup x \sqsubseteq x \longleftrightarrow y \sqsubseteq (x{::}'a{::}\mathit{Finite\text{-}Join\text{-}cpo})$
apply (*metis join-above2 larger-is-join1*)
apply (*metis join-above1 larger-is-join2*)
apply (*metis join-above2 larger-is-join1*)
apply (*metis join-above1 larger-is-join2*)
apply (*metis join-above1 join-above2 below-antisym larger-is-join1*)
apply (*metis join-above2 join-above1 below-antisym larger-is-join2*)
done

lemma *join-bottom-iff* [*iff*]:
$x \sqcup y = \bot \longleftrightarrow x = \bot \wedge (y{::}'a{::}\{\mathit{Finite\text{-}Join\text{-}cpo},\mathit{pcpo}\}) = \bot$
by (*metis all-compatible join-bottom(2) join-comm join-idem*)

class *Join-cpo* = *cpo* +
assumes *exists-lub*: $\exists\, u.\ S <<|\ u$

context *Join-cpo*
begin
subclass *Finite-Join-cpo*
apply *standard*
unfolding *compatible-def*
apply (*rule exists-lub*)
done
end

lemma *below-lubI* [*intro*, *simp*]:
fixes $x :: {'a} :: \mathit{Join\text{-}cpo}$
shows $x \in S \Longrightarrow x \sqsubseteq \mathit{lub}\ S$
by (*metis exists-lub is-ub-thelub-ex*)

lemma *lub-belowI* [*intro*, *simp*]:
fixes $x :: {'a} :: \mathit{Join\text{-}cpo}$
shows $(\bigwedge y.\ y \in S \Longrightarrow y \sqsubseteq x) \Longrightarrow \mathit{lub}\ S \sqsubseteq x$
by (*metis exists-lub is-lub-thelub-ex is-ub-def*)

instance *Join-cpo* $\subseteq$ *pcpo*
apply *standard*
apply (*rule exI*[**where** $x = \mathit{lub}\ \{\}$])
apply *auto*
done

lemma *lub-empty-set* [*simp*]:
$\mathit{lub}\ \{\} = (\bot{::}'a{::}\mathit{Join\text{-}cpo})$

by $(rule\ lub\text{-}eqI)$ *simp*

lemma *lub-insert*[*simp*]:
fixes $x :: 'a :: Join\text{-}cpo$
shows $lub\ (insert\ x\ S) = x \sqcup lub\ S$
by $(rule\ lub\text{-}eqI)$ $(auto\ intro:\ below\text{-}trans[OF\ \text{-}\ join\text{-}above2]\ simp\ add:\ join\text{-}below\text{-}iff\ is\text{-}ub\text{-}def\ is\text{-}lub\text{-}def)$

end

4 Env.tex

theory *Env*
imports *Main HOLCF−Join−Classes*
begin

default-sort *type*

Our type for environments is a function with a pcpo as the co-domain; this theory collects related definitions.

4.1 The domain of a pcpo-valued function

definition $edom :: ('key \Rightarrow 'value::pcpo) \Rightarrow 'key\ set$
where $edom\ m = \{x.\ m\ x \neq \bot\}$

lemma *bot-edom*[*simp*]: $edom\ \bot = \{\}$ **by** $(simp\ add:\ edom\text{-}def)$

lemma *bot-edom2*[*simp*]: $edom\ (\lambda\text{-}\ .\ \bot) = \{\}$ **by** $(simp\ add:\ edom\text{-}def)$

lemma *edomIff*: $(a \in edom\ m) = (m\ a \neq \bot)$ **by** $(simp\ add:\ edom\text{-}def)$
lemma *edom-iff2*: $(m\ a = \bot) \longleftrightarrow (a \notin edom\ m)$ **by** $(simp\ add:\ edom\text{-}def)$

lemma *edom-empty-iff-bot*: $edom\ m = \{\} \longleftrightarrow m = \bot$
by $(metis\ below\text{-}bottom\text{-}iff\ bot\text{-}edom\ edomIff\ empty\text{-}iff\ fun\text{-}belowI)$

lemma *lookup-not-edom*: $x \notin edom\ m \Longrightarrow m\ x = \bot$ **by** $(auto\ iff{:}edomIff)$

lemma *lookup-edom*[*simp*]: $m\ x \neq \bot \Longrightarrow x \in edom\ m$ **by** $(auto\ iff{:}edomIff)$

lemma *edom-mono*: $x \sqsubseteq y \Longrightarrow edom\ x \subseteq edom\ y$
unfolding *edom-def*
by *auto* $(metis\ below\text{-}bottom\text{-}iff\ fun\text{-}belowD)$

lemma *edom-subset-adm*[*simp*]:
  adm ($\lambda ae'$. *edom* $ae' \subseteq S$)
  apply (*rule admI*)
  apply *rule*
  apply (*subst* (*asm*) *edom-def*) **back**
  apply *simp*
  apply (*subst* (*asm*) *lub-fun*) **apply** *assumption*
  apply (*subst* (*asm*) *lub-eq-bottom-iff*)
  apply (*erule ch2ch-fun*)
  unfolding *not-all*
  apply (*erule exE*)
  apply (*rule set-mp*)
  apply (*rule allE*) **apply** *assumption* **apply** *assumption*
  unfolding *edom-def*
  apply *simp*
  done

4.2 Updates

lemma *edom-fun-upd-subset*: *edom* (h ($x := v$)) $\subseteq$ *insert* x (*edom* h)
  by (*auto simp add*: *edom-def*)

declare *fun-upd-same*[*simp*] *fun-upd-other*[*simp*]

4.3 Restriction

definition *env-restr* :: $'a$ *set* $\Rightarrow$ ($'a \Rightarrow$ $'b$::*pcpo*) $\Rightarrow$ ($'a \Rightarrow$ $'b$)
  where *env-restr* S m = (λ x. *if* $x \in S$ *then* m x *else* $\perp$)

abbreviation *env-restr-rev* (**infixl** $f|`$ *110*)
  where *env-restr-rev* m S $\equiv$ *env-restr* S m

notation (*latex* **output**) *env-restr-rev* (-|-)

lemma *env-restr-empty-iff*[*simp*]: m $f|`$ $S = \perp \longleftrightarrow$ *edom* $m \cap S = \{\}$
  apply (*auto simp add*: *edom-def env-restr-def lambda-strict*[*symmetric*] *split*:*if-splits*)
  apply *metis*
  apply (*fastforce simp add*: *edom-def env-restr-def lambda-strict*[*symmetric*] *split*:*if-splits*)
  done
lemmas *env-restr-empty* = *iffD2*[*OF env-restr-empty-iff*, *simp*]

lemma *lookup-env-restr*[*simp*]: $x \in S \Longrightarrow$ (m $f|`$ S) $x = m$ x
  by (*fastforce simp add*: *env-restr-def*)

lemma *lookup-env-restr-not-there*[*simp*]: $x \notin S \Longrightarrow$ (*env-restr* S m) $x = \perp$
  by (*fastforce simp add*: *env-restr-def*)

lemma *lookup-env-restr-eq*: (m $f|`$ S) x = (*if* $x \in S$ *then* m x *else* $\perp$)
  by *simp*

lemma *env-restr-eqI*: $(\bigwedge x.\ x \in S \Longrightarrow m_1\ x = m_2\ x) \Longrightarrow m_1\ f|`\ S = m_2\ f|`\ S$
by (*auto simp add*: *lookup-env-restr-eq*)

lemma *env-restr-eqD*: $m_1\ f|`\ S = m_2\ f|`\ S \Longrightarrow x \in S \Longrightarrow m_1\ x = m_2\ x$
by (*auto dest*!: *fun-cong*[**where** $x = x$])

lemma *env-restr-belowI*: $(\bigwedge x.\ x \in S \Longrightarrow m_1\ x \sqsubseteq m_2\ x) \Longrightarrow m_1\ f|`\ S \sqsubseteq m_2\ f|`\ S$
by (*auto intro*: *fun-belowI simp add*: *lookup-env-restr-eq*)

lemma *env-restr-belowD*: $m_1\ f|`\ S \sqsubseteq m_2\ f|`\ S \Longrightarrow x \in S \Longrightarrow m_1\ x \sqsubseteq m_2\ x$
by (*auto dest*!: *fun-belowD*[**where** $x = x$])

lemma *env-restr-env-restr*[*simp*]:
$x\ f|`\ d2\ f|`\ d1 = x\ f|`\ (d1 \cap d2)$
by (*fastforce simp add*: *env-restr-def*)

lemma *env-restr-env-restr-subset*:
$d1 \subseteq d2 \Longrightarrow x\ f|`\ d2\ f|`\ d1 = x\ f|`\ d1$
by (*metis Int-absorb2 env-restr-env-restr*)

lemma *env-restr-useless*: $edom\ m \subseteq S \Longrightarrow m\ f|`\ S = m$
by (*rule ext*) (*auto simp add*: *lookup-env-restr-eq dest*!: *set-mp*)

lemma *env-restr-UNIV*[*simp*]: $m\ f|`\ UNIV = m$
by (*rule env-restr-useless*) *simp*

lemma *env-restr-fun-upd*[*simp*]: $x \in S \Longrightarrow m1(x := v)\ f|`\ S = (m1\ f|`\ S)(x := v)$
apply (*rule ext*)
apply (*case-tac* $xa = x$)
apply (*auto simp add*: *lookup-env-restr-eq*)
done

lemma *env-restr-fun-upd-other*[*simp*]: $x \notin S \Longrightarrow m1(x := v)\ f|`\ S = m1\ f|`\ S$
apply (*rule ext*)
apply (*case-tac* $xa = x$)
apply (*auto simp add*: *lookup-env-restr-eq*)
done

lemma *env-restr-eq-subset*:
assumes $S \subseteq S'$
and $m1\ f|`\ S' = m2\ f|`\ S'$
shows $m1\ f|`\ S = m2\ f|`\ S$
using *assms*
by (*metis env-restr-env-restr le-iff-inf*)

lemma *env-restr-below-subset*:
assumes $S \subseteq S'$
and $m1\ f|`\ S' \sqsubseteq m2\ f|`\ S'$
shows $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$

using *assms*
by (*auto intro*!: *env-restr-belowI dest*!: *env-restr-belowD*)

lemma *edom-env*[*simp*]:
  $edom\ (m\ f|`\ S) = edom\ m \cap S$
  unfolding *edom-def env-restr-def* **by** *auto*

lemma *env-restr-below-self*: $f\,f|`\ S \sqsubseteq f$
  by (*rule fun-belowI*) (*auto simp add*: *env-restr-def*)

lemma *env-restr-below-trans*:
  $m1\ f|`\ S1 \sqsubseteq m2\ f|`\ S1 \Longrightarrow m2\ f|`\ S2 \sqsubseteq m3\ f|`\ S2 \Longrightarrow m1\ f|`\ (S1 \cap S2) \sqsubseteq m3\ f|`\ (S1 \cap S2)$
by (*auto intro*!: *env-restr-belowI dest*!: *env-restr-belowD elim*: *below-trans*)

lemma *env-restr-cont*: *cont* (*env-restr S*)
  apply (*rule cont2cont-lambda*)
  unfolding *env-restr-def*
  apply (*intro cont2cont cont-fun*)
  done

lemma *env-restr-mono*: $m1 \sqsubseteq m2 \Longrightarrow m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
  by (*metis env-restr-belowI fun-belowD*)

lemma *env-restr-mono2*: $S2 \subseteq S1 \Longrightarrow m\ f|`\ S2 \sqsubseteq m\ f|`\ S1$
  by (*metis env-restr-below-self env-restr-env-restr-subset*)

lemmas *cont-compose*[*OF env-restr-cont*, *cont2cont*, *simp*]

lemma *env-restr-cong*: $(\bigwedge x.\ edom\ m \subseteq S \cap S' \cup -S \cap -S') \Longrightarrow m\ f|`\ S = m\ f|`\ S'$
  by (*rule ext*)(*auto simp add*: *lookup-env-restr-eq edom-def*)

4.4 Deleting

definition *env-delete* :: $'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b{::}pcpo)$
  where $env\text{-}delete\ x\ m = m(x := \bot)$

lemma *lookup-env-delete*[*simp*]:
  $x' \neq x \Longrightarrow env\text{-}delete\ x\ m\ x' = m\ x'$
  by (*simp add*: *env-delete-def*)

lemma *lookup-env-delete-None*[*simp*]:
  $env\text{-}delete\ x\ m\ x = \bot$
  by (*simp add*: *env-delete-def*)

lemma *edom-env-delete*[*simp*]:
  $edom\ (env\text{-}delete\ x\ m) = edom\ m - \{x\}$
  by (*auto simp add*: *env-delete-def edom-def*)

lemma *edom-env-delete-subset*:
edom (*env-delete* x m) $\subseteq$ *edom* m **by** *auto*

lemma *env-delete-fun-upd*[*simp*]:
env-delete x ($m(x := v)$) = *env-delete* x m
by (*auto simp add*: *env-delete-def*)

lemma *env-delete-fun-upd2*[*simp*]:
(*env-delete* x m)$(x := v) = m(x := v)$
by (*auto simp add*: *env-delete-def*)

lemma *env-delete-fun-upd3*[*simp*]:
$x \neq y \Longrightarrow$ *env-delete* x ($m(y := v)$) = (*env-delete* x m)$(y := v)$
by (*auto simp add*: *env-delete-def*)

lemma *env-delete-noop*[*simp*]:
$x \notin$ *edom* $m \Longrightarrow$ *env-delete* x $m = m$
by (*auto simp add*: *env-delete-def edom-def*)

lemma *fun-upd-env-delete*[*simp*]: $x \in$ *edom* $\Gamma \Longrightarrow$ (*env-delete* x Γ)$(x := \Gamma\ x) = \Gamma$
by (*auto*)

lemma *env-restr-env-delete-other*[*simp*]: $x \notin S \Longrightarrow$ *env-delete* x m $f|$‘ $S = m$ $f|$‘ S
apply (*rule ext*)
apply (*auto simp add*: *lookup-env-restr-eq*)
by (*metis lookup-env-delete*)

lemma *env-delete-restr*: *env-delete* x $m = m$ $f|$‘ $(-\{x\})$
by (*auto simp add*: *lookup-env-restr-eq*)

lemma *below-env-deleteI*: $f\ x = \bot \Longrightarrow f \sqsubseteq g \Longrightarrow f \sqsubseteq$ *env-delete* x g
by (*metis env-delete-def env-delete-restr env-restr-mono fun-upd-triv*)

lemma *env-delete-below-cong*[*intro*]:
assumes $x \neq v \Longrightarrow e1\ x \sqsubseteq e2\ x$
shows *env-delete* v $e1$ $x \sqsubseteq$ *env-delete* v $e2$ x
using *assms* **unfolding** *env-delete-def* **by** *auto*

lemma *env-delete-env-restr-swap*:
env-delete x (*env-restr* S e) = *env-restr* S (*env-delete* x e)
by (*metis* (*erased*, *hide-lams*) *env-delete-def env-restr-fun-upd env-restr-fun-upd-other fun-upd-triv lookup-env-restr-eq*)

lemma *env-delete-mono*:
$m \sqsubseteq m' \Longrightarrow$ *env-delete* x $m \sqsubseteq$ *env-delete* x m'
unfolding *env-delete-restr*
by (*rule env-restr-mono*)

lemma *env-delete-below-arg*:

env-delete x m $\sqsubseteq$ *m*
unfolding *env-delete-restr*
by (*rule env-restr-below-self*)

4.5 Merging of two functions

We'd like to have some nice syntax for *override-on*.

abbreviation *override-on-syn* (- ++- - [*100*, *0*, *100*] *100*) **where** $f1 \mathrel{++_S} f2 \equiv$ *override-on f1 f2 S*

lemma *override-on-bot*[*simp*]:
$\bot \mathrel{++_S} m = m\ f|`\ S$
$m \mathrel{++_S} \bot = m\ f|`\ (-S)$
by (*auto simp add*: *override-on-def env-restr-def*)

lemma *edom-override-on*[*simp*]: $edom\ (m1 \mathrel{++_S} m2) = (edom\ m1 - S) \cup (edom\ m2 \cap S)$
by (*auto simp add*: *override-on-def edom-def*)

lemma *lookup-override-on-eq*: $(m1 \mathrel{++_S} m2)\ x = (if\ x \in S\ then\ m2\ x\ else\ m1\ x)$
by (*cases* $x \notin S$) *simp-all*

lemma *override-on-upd-swap*:
$x \notin S \Longrightarrow \varrho(x := z) \mathrel{++_S} \varrho' = (\varrho \mathrel{++_S} \varrho')(x := z)$
by (*auto simp add*: *override-on-def edom-def*)

lemma *override-on-upd*:
$x \in S \Longrightarrow \varrho \mathrel{++_S} (\varrho'(x := z)) = (\varrho \mathrel{++_{S - \{x\}}} \varrho')(x := z)$
by (*auto simp add*: *override-on-def edom-def*)

lemma *env-restr-add*: $(m1 \mathrel{++_{S2}} m2)\ f|`\ S = m1\ f|`\ S \mathrel{++_{S2}} m2\ f|`\ S$
by (*auto simp add*: *override-on-def edom-def env-restr-def*)

lemma *env-delete-add*: $env\text{-}delete\ x\ (m1 \mathrel{++_S} m2) = env\text{-}delete\ x\ m1 \mathrel{++_{S - \{x\}}} env\text{-}delete\ x\ m2$
by (*auto simp add*: *override-on-def edom-def env-restr-def env-delete-def*)

4.6 Environments with binary joins

lemma *edom-join*[*simp*]: $edom\ (f \sqcup (g{::}('a{::}type \Rightarrow 'b{::}\{Finite\text{-}Join\text{-}cpo,pcpo\}))) = edom\ f \cup edom\ g$
unfolding *edom-def* **by** *auto*

lemma *env-delete-join*[*simp*]: $env\text{-}delete\ x\ (f \sqcup (g{::}('a{::}type \Rightarrow 'b{::}\{Finite\text{-}Join\text{-}cpo,pcpo\}))) = env\text{-}delete\ x\ f \sqcup env\text{-}delete\ x\ g$
by (*metis env-delete-def fun-upd-meet-simp*)

lemma *env-restr-join*:
fixes $m1\ m2 :: 'a{::}type \Rightarrow 'b{::}\{Finite\text{-}Join\text{-}cpo,pcpo\}$

shows $(m1 \sqcup m2)\ f|`\ S = (m1\ f|`\ S) \sqcup (m2\ f|`\ S\ )$
by (*auto simp add*: *env-restr-def*)

lemma *env-restr-join2*:
fixes m :: $'a$::*type* $\Rightarrow$ $'b$::{*Finite-Join-cpo*,*pcpo*}
shows $m\ f|`\ S \sqcup m\ f|`\ S' = m\ f|`\ (S \cup S')$
by (*auto simp add*: *env-restr-def*)

lemma *join-env-restr-UNIV*:
fixes m :: $'a$::*type* $\Rightarrow$ $'b$::{*Finite-Join-cpo*,*pcpo*}
shows $S1 \cup S2 = UNIV \Longrightarrow (m\ f|`\ S1) \sqcup (m\ f|`\ S2) = m$
by (*fastforce simp add*: *env-restr-def*)

lemma *env-restr-split*:
fixes m :: $'a$::*type* $\Rightarrow$ $'b$::{*Finite-Join-cpo*,*pcpo*}
shows $m = m\ f|`\ S \sqcup m\ f|`\ (-\ S)$
by (*simp add*: *env-restr-join2 Compl-partition*)

lemma *env-restr-below-split*:
$m\ f|`\ S \sqsubseteq m' \Longrightarrow m\ f|`\ (-\ S) \sqsubseteq m' \Longrightarrow m \sqsubseteq m'$
by (*metis ComplI fun-below-iff lookup-env-restr*)

4.7 Singleton environments

definition *esing* :: $'a \Rightarrow 'b$::{*pcpo*} $\rightarrow ('a \Rightarrow 'b)$
where $esing\ x = (\Lambda\ a.\ (\lambda\ y\ .\ (if\ x = y\ then\ a\ else\ \bot)))$

lemma *esing-bot*[*simp*]: $esing\ x \cdot \bot = \bot$
by (*rule ext*)(*simp add*: *esing-def*)

lemma *esing-simps*[*simp*]:
$(esing\ x \cdot n)\ x = n$
$x' \neq x \Longrightarrow (esing\ x \cdot n)\ x' = \bot$
by (*simp-all add*: *esing-def*)

lemma *esing-eq-up-iff*[*simp*]: $(esing\ x\cdot(up\cdot a))\ y = up\cdot a' \longleftrightarrow (x = y \wedge a = a')$
by (*auto simp add*: *fun-below-iff esing-def*)

lemma *esing-below-iff*[*simp*]: $esing\ x \cdot a \sqsubseteq ae \longleftrightarrow a \sqsubseteq ae\ x$
by (*auto simp add*: *fun-below-iff esing-def*)

lemma *edom-esing-subset*: $edom\ (esing\ x\cdot n) \subseteq \{x\}$
unfolding *edom-def esing-def* **by** *auto*

lemma *edom-esing-up*[*simp*]: $edom\ (esing\ x \cdot (up \cdot n)) = \{x\}$
unfolding *edom-def esing-def* **by** *auto*

lemma *env-delete-esing*[*simp*]: $env\text{-}delete\ x\ (esing\ x \cdot n) = \bot$
unfolding *env-delete-def esing-def*

by *auto*

lemma *env-restr-esing*[*simp*]:
$x \in S \Longrightarrow esing\ x{\cdot}v\ f|`\ S = esing\ x{\cdot}v$
by (*auto intro*: *env-restr-useless dest*: *set-mp*[*OF edom-esing-subset*])

lemma *env-restr-esing2*[*simp*]:
$x \notin S \Longrightarrow esing\ x{\cdot}v\ f|`\ S = \bot$
by (*auto dest*: *set-mp*[*OF edom-esing-subset*])

lemma *esing-eq-iff*[*simp*]:
$esing\ x{\cdot}v = esing\ x{\cdot}v' \longleftrightarrow v = v'$
by (*metis esing-simps*(*1*))

end

5 Pointwise.tex

theory *Pointwise* **imports** *Main* **begin**

Lifting a relation to a function.

definition *pointwise* **where** $pointwise\ P\ m\ m' = (\forall\ x.\ P\ (m\ x)\ (m'\ x))$

lemma *pointwiseI*[*intro*]: $(\bigwedge\ x.\ P\ (m\ x)\ (m'\ x)) \Longrightarrow pointwise\ P\ m\ m'$ **unfolding** *pointwise-def*
by *blast*

end

6 HOLCF-Utils.tex

theory *HOLCF−Utils*
imports ~~/*src*/*HOL*/*HOLCF*/*HOLCF Pointwise*
begin

default-sort *type*

lemmas *cont-fun*[*simp*]
lemmas *cont2cont-fun*[*simp*]

lemma *cont-compose2*:
assumes $\bigwedge\ y.\ cont\ (\lambda\ x.\ c\ x\ y)$
assumes $\bigwedge\ x.\ cont\ (\lambda\ y.\ c\ x\ y)$
assumes *cont f*
assumes *cont g*
shows $cont\ (\lambda x.\ c\ (f\ x)\ (g\ x))$

by (*intro cont-apply*[*OF assms*(*4*) *assms*(*2*)]
cont2cont-fun[*OF cont-compose*[*OF* - *assms*(*3*)]]
cont2cont-lambda[*OF assms*(*1*)])

lemma *pointwise-adm*:
fixes P :: *'a*::*pcpo* ⇒ *'b*::*pcpo* ⇒ *bool*
assumes *adm* (λ *x*. *P* (*fst x*) (*snd x*))
shows *adm* (λ *m*. *pointwise P* (*fst m*) (*snd m*))
proof (*rule admI*, *goal-cases*)
case *prems*: (*1 Y*)
show *?case*
apply (*rule pointwiseI*)
apply (*rule admD*[*OF adm-subst*[**where** *t* = λ*p* . (*fst p x*, *snd p x*) **for** *x*, *OF* - *assms*, *simplified*] ‹*chain Y*›])
using *prems*(*2*) **unfolding** *pointwise-def* **apply** *auto*
done
qed

lemma *cfun-beta-Pair*:
assumes *cont* (λ *p*. *f* (*fst p*) (*snd p*))
shows *csplit*·(Λ *a b* . *f a b*)·(*x*, *y*) = *f x y*
apply *simp*
apply (*subst beta-cfun*)
apply (*rule cont2cont-LAM'*)
apply (*rule assms*)
apply (*rule beta-cfun*)
apply (*rule cont2cont-fun*)
using *assms*
unfolding *prod-cont-iff*
apply *auto*
done

lemma *fun-upd-mono*:
$\varrho 1 \sqsubseteq \varrho 2 \Longrightarrow v1 \sqsubseteq v2 \Longrightarrow \varrho 1(x := v1) \sqsubseteq \varrho 2(x := v2)$
apply (*rule fun-belowI*)
apply (*case-tac xa* = *x*)
apply *simp*
apply (*auto elim*:*fun-belowD*)
done

lemma *fun-upd-cont*[*simp*,*cont2cont*]:
assumes *cont f* **and** *cont h*
shows *cont* (λ *x*. (*f x*)(*v* := *h x*) :: *'a* ⇒ *'b*::*pcpo*)
by (*rule cont2cont-lambda*)(*auto simp add*: *assms*)

lemma *fun-upd-belowI*:
assumes $\bigwedge z \,.\, z \neq x \Longrightarrow \varrho\ z \sqsubseteq \varrho'\ z$
assumes $y \sqsubseteq \varrho'\ x$

shows $\varrho(x := y) \sqsubseteq \varrho'$
apply (*rule fun-belowI*)
using *assms*
apply (*case-tac* $xa = x$)
apply *auto*
done

lemma *cont-if-else-above*:
assumes *cont f*
assumes *cont g*
assumes $\bigwedge x.\ f\ x \sqsubseteq g\ x$
assumes $\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow P\ y \Longrightarrow P\ x$
assumes *adm P*
shows *cont* ($\lambda x.$ *if* $P\ x$ *then* $f\ x$ *else* $g\ x$) (**is** *cont ?I*)
proof(*intro contI2 monofunI*)
fix $x\ y$ :: $'a$
assume $x \sqsubseteq y$
with *assms*(*4*)[*OF this*]
show $?I\ x \sqsubseteq ?I\ y$
apply (*auto*)
apply (*rule cont2monofunE*[*OF assms*(*1*)], *assumption*)
apply (*rule below-trans*[*OF cont2monofunE*[*OF assms*(*1*)] *assms*(*3*)], *assumption*)
apply (*rule cont2monofunE*[*OF assms*(*2*)], *assumption*)
done
next
fix Y :: $nat \Rightarrow 'a$
assume *chain Y*
assume *chain* ($\lambda i\ .\ ?I\ (Y\ i)$)

have *ch-f*: $f\ (\bigsqcup\ i.\ Y\ i) \sqsubseteq (\bigsqcup\ i.\ f\ (Y\ i))$ **by** (*metis* ‹*chain Y*› *assms*(*1*) *below-refl cont2contlubE*)

show $?I\ (\bigsqcup\ i.\ Y\ i) \sqsubseteq (\bigsqcup\ i.\ ?I\ (Y\ i))$
proof(*cases* $\forall\ i.\ P\ (Y\ i)$)
case *True* **hence** $P\ (\bigsqcup\ i.\ Y\ i)$ **by** (*metis* ‹*chain Y*› *adm-def assms*(*5*))
with *True ch-f* **show** *?thesis* **by** *auto*
next
case *False*
then obtain j **where** $\neg\ P\ (Y\ j)$ **by** *auto*
hence $*$: $\forall\ i \geq j.\ \neg\ P\ (Y\ i)\ \neg\ P\ (\bigsqcup\ i.\ Y\ i)$
apply (*auto*)
apply (*metis assms*(*4*) *chain-mono*[*OF* ‹*chain Y*›])
apply (*metis assms*(*4*) *is-ub-thelub*[*OF* ‹*chain Y*›])
done

have $?I\ (\bigsqcup\ i.\ Y\ i) = g\ (\bigsqcup\ i.\ Y\ i)$ **using** $*$ **by** *simp*
also have $\ldots = g\ (\bigsqcup\ i.\ Y\ (i + j))$ **by** (*metis lub-range-shift*[*OF* ‹*chain Y*›])
also have $\ldots = (\bigsqcup\ i.\ (g\ (Y\ (i + j))))$ **by** (*rule cont2contlubE*[*OF assms*(*2*) *chain-shift*[*OF* ‹*chain Y*›]])

also have ... = ($\bigsqcup$ *i*. (*?I* (*Y* (*i* + *j*)))) **using** ∗ **by** *auto*
also have ... = ($\bigsqcup$ *i*. (*?I* (*Y i*))) **by** (*metis lub-range-shift*[*OF* ‹*chain* (λ*i* . *?I* (*Y i*))›])
finally show *?thesis* **by** *simp*
qed
qed

fun *up2option* :: ′*a*::*cpo*$_\perp$ ⇒ ′*a option*
where *up2option Ibottom* = *None*
| *up2option* (*Iup a*) = *Some a*

lemma *up2option-simps*[*simp*]:
up2option ⊥ = *None*
up2option (*up*·*x*) = *Some x*
unfolding *up-def* **by** (*simp-all add*: *cont-Iup inst-up-pcpo*)

fun *option2up* :: ′*a option* ⇒ ′*a*::*cpo*$_\perp$
where *option2up None* = ⊥
| *option2up* (*Some a*) = *up*·*a*

lemma *option2up-up2option*[*simp*]:
option2up (*up2option x*) = *x*
by (*cases x*) *auto*
lemma *up2option-option2up*[*simp*]:
up2option (*option2up x*) = *x*
by (*cases x*) *auto*

lemma *adm-subst2*: *cont f* ⟹ *cont g* ⟹ *adm* (λ*x*. *f* (*fst x*) = *g* (*snd x*))
apply (*rule admI*)
apply (*simp add*:
cont2contlubE[**where** *f* = *f*] *cont2contlubE*[**where** *f* = *g*]
cont2contlubE[**where** *f* = *snd*] *cont2contlubE*[**where** *f* = *fst*]
)
done

6.1 Composition of fun and cfun

lemma *cont2cont-comp* [*simp*, *cont2cont*]:
assumes *cont f*
assumes ⋀ *x*. *cont* (*f x*)
assumes *cont g*
shows *cont* (λ *x*. (*f x*) ∘ (*g x*))
unfolding *comp-def*
by (*rule cont2cont-lambda*)
(*intro cont2cont* ‹*cont g*› ‹*cont f*› *cont-compose2*[*OF cont2cont-fun*[*OF assms*(*1*)] *assms*(*2*)]
cont2cont-fun)

definition *cfun-comp* :: (′*a*::*pcpo* → ′*b*::*pcpo*) → (′*c*::*type* ⇒ ′*a*) → (′*c*::*type* ⇒ ′*b*)
where *cfun-comp* = (Λ *f* ϱ. (λ *x*. *f*·*x*) ∘ ϱ)

lemma [*simp*]: $cfun\text{-}comp\cdot f\cdot(\varrho(x := v)) = (cfun\text{-}comp\cdot f\cdot\varrho)(x := f\cdot v)$
  unfolding *cfun-comp-def* **by** *auto*

lemma *cfun-comp-app*[*simp*]: $(cfun\text{-}comp\cdot f\cdot\varrho)\ x = f\cdot(\varrho\ x)$
  unfolding *cfun-comp-def* **by** *auto*

lemma *fix-eq-fix*:
  $f\cdot(fix\cdot g) \sqsubseteq fix\cdot g \Longrightarrow g\cdot(fix\cdot f) \sqsubseteq fix\cdot f \Longrightarrow fix\cdot f = fix\cdot g$
  by (*metis fix-least-below below-antisym*)

6.2 Additional transitivity rules

These collect side-conditions of the form *cont f*, so the usual way to discharge them is to write *by this* (*intro cont2cont*)+ at the end.

lemma *below-trans-cong*[*trans*]:
  $a \sqsubseteq f\ x \Longrightarrow x \sqsubseteq y \Longrightarrow cont\ f \Longrightarrow a \sqsubseteq f\ y$
by (*metis below-trans cont2monofunE*)

lemma *not-bot-below-trans*[*trans*]:
  $a \neq \bot \Longrightarrow a \sqsubseteq b \Longrightarrow b \neq \bot$
  by (*metis below-bottom-iff*)

lemma *not-bot-below-trans-cong*[*trans*]:
  $f\ a \neq \bot \Longrightarrow a \sqsubseteq b \Longrightarrow cont\ f \Longrightarrow f\ b \neq \bot$
  by (*metis below-bottom-iff cont2monofunE*)

end

7 EvalHeap.tex

theory *EvalHeap*
  imports *AList−Utils Env ../Nominal2/Nominal2 HOLCF−Utils*
begin

7.1 Conversion from heaps to environments

fun
  $evalHeap :: ('var \times 'exp)\ list \Rightarrow ('exp \Rightarrow 'value::\{pure,pcpo\}) \Rightarrow 'var \Rightarrow 'value$
where
  $evalHeap\ []\ \text{-} = \bot$
| $evalHeap\ ((x,e)\#h)\ eval = (evalHeap\ h\ eval)\ (x := eval\ e)$

lemma *cont2cont-evalHeap*[*simp*, *cont2cont*]:
  $(\bigwedge\ e\ .\ e \in snd\ `\ set\ h \Longrightarrow cont\ (\lambda\varrho.\ eval\ \varrho\ e)) \Longrightarrow cont\ (\lambda\ \varrho.\ evalHeap\ h\ (eval\ \varrho))$
  by(*induct h*, *auto*)

lemma *evalHeap-eqvt*[*eqvt*]:

$\pi \cdot$ *evalHeap h eval* $=$ *evalHeap* $(\pi \cdot h)$ $(\pi \cdot eval)$
by (*induct h*) (*auto simp add*:*fun-upd-eqvt simp del*: *fun-upd-apply*)

lemma *edom-evalHeap-subset*:*edom* (*evalHeap h eval*) $\subseteq$ *domA h*
by (*induct h eval rule*:*evalHeap.induct*) (*auto dest*:*set-mp*[*OF edom-fun-upd-subset*] *simp del*: *fun-upd-apply*)

lemma *evalHeap-cong*[*fundef-cong*]:
$\llbracket$ *heap1* $=$ *heap2* ; ($\bigwedge$ *e*. *e* $\in$ *snd* ' *set heap2* $\Longrightarrow$ *eval1 e* $=$ *eval2 e*) $\rrbracket$
$\Longrightarrow$ *evalHeap heap1 eval1* $=$ *evalHeap heap2 eval2*
by (*induct heap2 eval2 arbitrary*:*heap1 rule*:*evalHeap.induct*, *auto*)

lemma *lookupEvalHeap*:
assumes *v* $\in$ *domA h*
shows (*evalHeap h f*) *v* $=$ *f* (*the* (*map-of h v*))
using *assms*
by (*induct h f rule*: *evalHeap.induct*) *auto*

lemma *lookupEvalHeap'*:
assumes *map-of* Γ *v* $=$ *Some e*
shows (*evalHeap* Γ *f*) *v* $=$ *f e*
using *assms*
by (*induct* Γ *f rule*: *evalHeap.induct*) *auto*

lemma *lookupEvalHeap-other*[*simp*]:
assumes *v* $\notin$ *domA* Γ
shows (*evalHeap* Γ *f*) *v* $= \bot$
using *assms*
by (*induct* Γ *f rule*: *evalHeap.induct*) *auto*

lemma *env-restr-evalHeap-noop*:
domA h $\subseteq$ *S* $\Longrightarrow$ *env-restr S* (*evalHeap h eval*) $=$ *evalHeap h eval*
apply (*rule ext*)
apply (*case-tac x* $\in$ *S*)
apply (*auto simp add*: *lookupEvalHeap intro*: *lookupEvalHeap-other*)
done

lemma *env-restr-evalHeap-same*[*simp*]:
env-restr (*domA h*) (*evalHeap h eval*) $=$ *evalHeap h eval*
by (*simp add*: *env-restr-evalHeap-noop*)

lemma *evalHeap-cong'*:
$\llbracket$ ($\bigwedge$ *x*. *x* $\in$ *domA heap* $\Longrightarrow$ *eval1* (*the* (*map-of heap x*)) $=$ *eval2* (*the* (*map-of heap x*))) $\rrbracket$
$\Longrightarrow$ *evalHeap heap eval1* $=$ *evalHeap heap eval2*
apply (*rule ext*)
apply (*case-tac x* $\in$ *domA heap*)
apply (*auto simp add*: *lookupEvalHeap*)
done

lemma *lookupEvalHeapNotAppend*[*simp*]:
assumes $x \notin$ *domA* Γ
shows (*evalHeap* (Γ@*h*) *f*) x = *evalHeap h f x*
using *assms* **by** (*induct* Γ, *auto*)

lemma *evalHeap-delete*[*simp*]: *evalHeap* (*delete x* Γ) *eval* = *env-delete x* (*evalHeap* Γ *eval*)
by (*induct* Γ) *auto*

lemma *evalHeap-mono*:
$x \notin$ *domA* $\Gamma \Longrightarrow$
evalHeap Γ *eval* $\sqsubseteq$ *evalHeap* ((*x*, *e*) # Γ) *eval*
apply *simp*
apply (*rule fun-belowI*)
apply (*case-tac xa* $\in$ *domA* Γ)
apply (*case-tac xa* = *x*)
apply *auto*
done

7.2 Reordering lemmas

lemma *evalHeap-reorder*:
assumes *map-of* Γ = *map-of* Δ
shows *evalHeap* Γ *h* = *evalHeap* Δ *h*
proof (*rule ext*)
from *assms*
have $*$: *domA* Γ = *domA* Δ **by** (*metis dom-map-of-conv-domA*)

fix *x*
show *evalHeap* Γ *h x* = *evalHeap* Δ *h x*
using *assms*(*1*) $*$
apply (*cases x* $\in$ *domA* Γ)
apply (*auto simp add*: *lookupEvalHeap*)
done
qed

lemma *evalHeap-reorder-head*:
assumes $x \neq y$
shows *evalHeap* ((*x*,*e1*)#(*y*,*e2*)#Γ) *eval* = *evalHeap* ((*y*,*e2*)#(*x*,*e1*)#Γ) *eval*
by (*rule evalHeap-reorder*) (*simp add*: *fun-upd-twist*[*OF assms*])

lemma *evalHeap-reorder-head-append*:
assumes $x \notin$ *domA* Γ
shows *evalHeap* ((*x*,*e*)#Γ@Δ) *eval* = *evalHeap* (Γ @ ((*x*,*e*)#Δ)) *eval*
by (*rule evalHeap-reorder*) (*simp*, *metis assms dom-map-of-conv-domA map-add-upd-left*)

lemma *evalHeap-subst-exp*:
assumes *eval e* = *eval e'*
shows *evalHeap* ((*x*,*e*)#Γ) *eval* = *evalHeap* ((*x*,*e'*)#Γ) *eval*
by (*simp add*: *assms*)

end

8 Nominal-Utils.tex

theory *Nominal−Utils*
imports *../Nominal2/Nominal2 ~~/src/HOL/Library/AList*
begin

8.1 Lemmas helping with equivariance proofs

lemma *perm-rel-lemma*:
  assumes $\bigwedge \pi\ x\ y.\ r\ (\pi \cdot x)\ (\pi \cdot y) \Longrightarrow r\ x\ y$
  shows $r\ (\pi \cdot x)\ (\pi \cdot y) \longleftrightarrow r\ x\ y$ (**is** $?l \longleftrightarrow ?r$)
by (*metis* (*full-types*) *assms permute-minus-cancel*(*2*))

lemma *perm-rel-lemma2*:
  assumes $\bigwedge \pi\ x\ y.\ r\ x\ y \Longrightarrow r\ (\pi \cdot x)\ (\pi \cdot y)$
  shows $r\ x\ y \longleftrightarrow r\ (\pi \cdot x)\ (\pi \cdot y)$ (**is** $?l \longleftrightarrow ?r$)
by (*metis* (*full-types*) *assms permute-minus-cancel*(*2*))

lemma *fun-eqvtI*:
  assumes *f-eqvt*[*eqvt*]: $(\bigwedge p\ x.\ p \cdot (f\ x) = f\ (p \cdot x))$
  shows $p \cdot f = f$ **by** *perm-simp rule*

lemma *eqvt-at-apply*:
  assumes *eqvt-at f x*
  shows $(p \cdot f)\ x = f\ x$
by (*metis* (*hide-lams*, *no-types*) *assms eqvt-at-def permute-fun-def permute-minus-cancel*(*1*))

lemma *eqvt-at-apply'*:
  assumes *eqvt-at f x*
  shows $p \cdot f\ x = f\ (p \cdot x)$
by (*metis* (*hide-lams*, *no-types*) *assms eqvt-at-def*)

lemma *eqvt-at-apply''*:
  assumes *eqvt-at f x*
  shows $(p \cdot f)\ (p \cdot x) = f\ (p \cdot x)$
by (*metis* (*hide-lams*, *no-types*) *assms eqvt-at-def permute-fun-def permute-minus-cancel*(*1*))

lemma *size-list-eqvt*[*eqvt*]: $p \cdot \mathit{size\text{-}list}\ f\ x = \mathit{size\text{-}list}\ (p \cdot f)\ (p \cdot x)$
proof (*induction x*)
  case (*Cons x xs*)
  have $f\ x = p \cdot (f\ x)$ **by** (*simp add*: *permute-pure*)
  also have $... = (p \cdot f)\ (p \cdot x)$ **by** *simp*
  with *Cons*
  show *?case* **by** (*auto simp add*: *permute-pure*)

qed *simp*

8.2 Freshness via equivariance

lemma *eqvt-fresh-cong1*: $(\bigwedge p\ x.\ p \cdot (f\ x) = f\ (p \cdot x)) \Longrightarrow a \sharp x \Longrightarrow a \sharp f\ x$
  apply (*rule fresh-fun-eqvt-app*[*of f*])
  apply (*rule eqvtI*)
  apply (*rule eq-reflection*)
  apply (*rule ext*)
  apply (*metis permute-fun-def permute-minus-cancel*(*1*))
  apply *assumption*
  done

lemma *eqvt-fresh-cong2*:
  assumes *eqvt*: $(\bigwedge p\ x\ y.\ p \cdot (f\ x\ y) = f\ (p \cdot x)\ (p \cdot y))$
  and *fresh1*: $a \sharp x$ **and** *fresh2*: $a \sharp y$
  shows $a \sharp f\ x\ y$
proof−
  have *eqvt* $(\lambda\ (x,y).\ f\ x\ y)$
    using *eqvt*
    apply −
    apply (*auto simp add*: *eqvt-def*)
    apply (*rule ext*)
    apply *auto*
    by (*metis permute-minus-cancel*(*1*))
  moreover
  have $a \sharp (x,\ y)$ **using** *fresh1 fresh2* **by** *auto*
  ultimately
  have $a \sharp (\lambda\ (x,y).\ f\ x\ y)\ (x,\ y)$ **by** (*rule fresh-fun-eqvt-app*)
  thus *?thesis* **by** *simp*
qed

lemma *eqvt-fresh-star-cong1*:
  assumes *eqvt*: $(\bigwedge p\ x.\ p \cdot (f\ x) = f\ (p \cdot x))$
  and *fresh1*: $a \sharp{*}\ x$
  shows $a \sharp{*}\ f\ x$
  by (*metis fresh-star-def eqvt-fresh-cong1 assms*)

lemma *eqvt-fresh-star-cong2*:
  assumes *eqvt*: $(\bigwedge p\ x\ y.\ p \cdot (f\ x\ y) = f\ (p \cdot x)\ (p \cdot y))$
  and *fresh1*: $a \sharp{*}\ x$ **and** *fresh2*: $a \sharp{*}\ y$
  shows $a \sharp{*}\ f\ x\ y$
  by (*metis fresh-star-def eqvt-fresh-cong2 assms*)

lemma *eqvt-fresh-cong3*:
  assumes *eqvt*: $(\bigwedge p\ x\ y\ z.\ p \cdot (f\ x\ y\ z) = f\ (p \cdot x)\ (p \cdot y)\ (p \cdot z))$
  and *fresh1*: $a \sharp x$ **and** *fresh2*: $a \sharp y$ **and** *fresh3*: $a \sharp z$
  shows $a \sharp f\ x\ y\ z$
proof−

have *eqvt* (λ (x,y,z). $f\ x\ y\ z$)
using *eqvt*
apply $-$
apply (*auto simp add*: *eqvt-def*)
apply (*rule ext*)
apply *auto*
by (*metis permute-minus-cancel*(*1*))
moreover
have $a \sharp (x,\ y,\ z)$ **using** *fresh1 fresh2 fresh3* **by** *auto*
ultimately
have $a \sharp (\lambda\ (x,y,z).\ f\ x\ y\ z)\ (x,\ y,\ z)$ **by** (*rule fresh-fun-eqvt-app*)
thus *?thesis* **by** *simp*
qed

lemma *eqvt-fresh-star-cong3*:
assumes *eqvt*: $(\bigwedge p\ x\ y\ z.\ p \cdot (f\ x\ y\ z) = f\ (p \cdot x)\ (p \cdot y)\ (p \cdot z))$
and *fresh1*: $a \sharp* x$ **and** *fresh2*: $a \sharp* y$ **and** *fresh3*: $a \sharp* z$
shows $a \sharp* f\ x\ y\ z$
by (*metis fresh-star-def eqvt-fresh-cong3 assms*)

8.3 Additional simplification rules

lemma *not-self-fresh*[*simp*]: *atom* $x \sharp x \longleftrightarrow$ *False*
by (*metis fresh-at-base*(*2*))

lemma *fresh-star-singleton*: $\{\ x\ \} \sharp* e \longleftrightarrow x \sharp e$
by (*simp add*: *fresh-star-def*)

8.4 Additional equivariance lemmas

lemma *eqvt-cases*:
fixes $f\ x\ \pi$
assumes *eqvt*: $\bigwedge x.\ \pi \cdot f\ x = f\ (\pi \cdot x)$
obtains $f\ x\ f\ (\pi \cdot x) \mid \neg\ f\ x \quad \neg\ f\ (\pi \cdot x)$
using *assms*[*symmetric*]
by (*cases* $f\ x$) *auto*

lemma *range-eqvt*: $\pi \cdot$ *range* $Y =$ *range* $(\pi \cdot Y)$
unfolding *image-eqvt UNIV-eqvt* **..**

lemma *case-option-eqvt*[*eqvt*]:
$\pi \cdot$ *case-option* $d\ f\ x =$ *case-option* $(\pi \cdot d)\ (\pi \cdot f)\ (\pi \cdot x)$
by(*cases* x)(*simp-all*)

lemma *supp-option-eqvt*:
supp (*case-option* $d\ f\ x$) $\subseteq$ *supp* $d \cup$ *supp* $f \cup$ *supp* x
apply (*cases* x)
apply (*auto simp add*: *supp-Some*)
apply (*metis* (*mono-tags*) *Un-iff subsetCE supp-fun-app*)
done

lemma *funpow-eqvt*[*simp*,*eqvt*]:
  $\pi \bullet ((f :: 'a \Rightarrow 'a{::}pt)\ \hat{}\hat{}\ n) = (\pi \bullet f)\ \hat{}\hat{}\ (\pi \bullet n)$
 apply (*induct n*)
 apply *simp*
 apply (*rule ext*)
 apply *simp*
 apply *perm-simp*
 apply *simp*
 done

lemma *delete-eqvt*[*eqvt*]:
  $\pi \bullet$ *AList.delete* $x\ \Gamma$ = *AList.delete* $(\pi \bullet x)\ (\pi \bullet \Gamma)$
by (*induct* Γ, *auto*)

lemma *restrict-eqvt*[*eqvt*]:
  $\pi \bullet$ *AList.restrict* $S\ \Gamma$ = *AList.restrict* $(\pi \bullet S)\ (\pi \bullet \Gamma)$
unfolding *AList.restrict-eq* **by** *perm-simp rule*

lemma *supp-restrict*:
  supp (*AList.restrict* $S\ \Gamma$) $\subseteq$ *supp* Γ
 by (*induction* Γ) (*auto simp add*: *supp-Pair supp-Cons*)

lemma *clearjunk-eqvt*[*eqvt*]:
  $\pi \bullet$ *AList.clearjunk* Γ = *AList.clearjunk* $(\pi \bullet \Gamma)$
  by (*induction* Γ *rule*: *clearjunk.induct*) *auto*

lemma *map-ran-eqvt*[*eqvt*]:
  $\pi \bullet$ *map-ran* $f\ \Gamma$ = *map-ran* $(\pi \bullet f)\ (\pi \bullet \Gamma)$
by (*induct* Γ, *auto*)

lemma *dom-perm*:
  dom $(\pi \bullet f) = \pi \bullet$ (*dom* f)
  unfolding *dom-def* **by** (*perm-simp*) (*simp*)

lemmas *dom-perm-rev*[*simp*,*eqvt*] = *dom-perm*[*symmetric*]

lemma *ran-perm*[*simp*]:
  $\pi \bullet$ (*ran* f) = *ran* $(\pi \bullet f)$
  unfolding *ran-def* **by** (*perm-simp*) (*simp*)

lemma *map-add-eqvt*[*eqvt*]:
  $\pi \bullet (m1 \mathbin{++} m2) = (\pi \bullet m1) \mathbin{++} (\pi \bullet m2)$
  unfolding *map-add-def*
  by (*perm-simp*, *rule*)

lemma *map-of-eqvt*[*eqvt*]:
  $\pi \bullet$ *map-of* l = *map-of* $(\pi \bullet l)$
  apply (*induct* l)

apply (*simp add*: *permute-fun-def*)
apply *simp*
apply *perm-simp*
apply *auto*
done

lemma *concat-eqvt*[*eqvt*]: $\pi \cdot concat\ l = concat\ (\pi \cdot l)$
by (*induction l*)(*auto simp add*: *append-eqvt*)

lemma *tranclp-eqvt*[*eqvt*]: $\pi \cdot tranclp\ P\ v_1\ v_2 = tranclp\ (\pi \cdot P)\ (\pi \cdot v_1)\ (\pi \cdot v_2)$
unfolding *tranclp-def* **by** *perm-simp rule*

lemma *rtranclp-eqvt*[*eqvt*]: $\pi \cdot rtranclp\ P\ v_1\ v_2 = rtranclp\ (\pi \cdot P)\ (\pi \cdot v_1)\ (\pi \cdot v_2)$
unfolding *rtranclp-def* **by** *perm-simp rule*

lemma *Set-filter-eqvt*[*eqvt*]: $\pi \cdot Set.filter\ P\ S = Set.filter\ (\pi \cdot P)\ (\pi \cdot S)$
unfolding *Set.filter-def*
by *perm-simp rule*

lemma *Sigma-eqvt'*[*eqvt*]: $\pi \cdot Sigma = Sigma$
apply (*rule ext*)
apply (*rule ext*)
apply (*subst permute-fun-def*)
apply (*subst permute-fun-def*)
unfolding *Sigma-def*
apply *perm-simp*
apply (*simp add*: *permute-self*)
done

lemma *override-on-eqvt*[*eqvt*]:
$\pi \cdot (override\text{-}on\ m1\ m2\ S) = override\text{-}on\ (\pi \cdot m1)\ (\pi \cdot m2)\ (\pi \cdot S)$
by (*auto simp add*: *override-on-def*)

lemma *card-eqvt*[*eqvt*]:
$\pi \cdot (card\ S) = card\ (\pi \cdot S)$
by (*cases finite S*, *induct rule*: *finite-induct*) (*auto simp add*: *card-insert-if mem-permute-iff permute-pure*)

lemma *Projl-permute*:
assumes *a*: $\exists y.\ f = Inl\ y$
shows $(p \cdot (Sum\text{-}Type.projl\ f)) = Sum\text{-}Type.projl\ (p \cdot f)$
using *a* **by** *auto*

lemma *Projr-permute*:
assumes *a*: $\exists y.\ f = Inr\ y$
shows $(p \cdot (Sum\text{-}Type.projr\ f)) = Sum\text{-}Type.projr\ (p \cdot f)$
using *a* **by** *auto*

8.5 Freshness lemmas

lemma *fresh-list-elem*:
  assumes $a \sharp \Gamma$
  and $e \in \mathit{set}\ \Gamma$
  shows $a \sharp e$
using *assms*
by(*induct* Γ)(*auto simp add*: *fresh-Cons*)

lemma *set-not-fresh*:
  $x \in \mathit{set}\ L \Longrightarrow \neg(\mathit{atom}\ x \sharp L)$
  by (*metis fresh-list-elem not-self-fresh*)

lemma *pure-fresh-star*[*simp*]: $a \sharp{*}\ (x :: 'a :: \mathit{pure})$
  by (*simp add*: *fresh-star-def pure-fresh*)

lemma *supp-set-mem*: $x \in \mathit{set}\ L \Longrightarrow \mathit{supp}\ x \subseteq \mathit{supp}\ L$
  by (*induct L*) (*auto simp add*: *supp-Cons*)

lemma *set-supp-mono*: $\mathit{set}\ L \subseteq \mathit{set}\ L2 \Longrightarrow \mathit{supp}\ L \subseteq \mathit{supp}\ L2$
  by (*induct L*)(*auto simp add*: *supp-Cons supp-Nil dest*:*supp-set-mem*)

lemma *fresh-star-at-base*:
  fixes $x :: 'a :: \mathit{at\text{-}base}$
  shows $S \sharp{*}\ x \longleftrightarrow \mathit{atom}\ x \notin S$
  by (*metis fresh-at-base*(*2*) *fresh-star-def*)

8.6 Freshness and support for subsets of variables

lemma *supp-mono*: *finite* $(B::'a::\mathit{fs}\ \mathit{set}) \Longrightarrow A \subseteq B \Longrightarrow \mathit{supp}\ A \subseteq \mathit{supp}\ B$
  by (*metis infinite-super subset-Un-eq supp-of-finite-union*)

lemma *fresh-subset*:
  $\mathit{finite}\ B \Longrightarrow x \sharp (B :: 'a::\mathit{at\text{-}base}\ \mathit{set}) \Longrightarrow A \subseteq B \Longrightarrow x \sharp A$
  by (*auto dest*:*supp-mono simp add*: *fresh-def*)

lemma *fresh-star-subset*:
  $\mathit{finite}\ B \Longrightarrow x \sharp{*}\ (B :: 'a::\mathit{at\text{-}base}\ \mathit{set}) \Longrightarrow A \subseteq B \Longrightarrow x \sharp{*}\ A$
  by (*metis fresh-star-def fresh-subset*)

lemma *fresh-star-set-subset*:
  $x \sharp{*}\ (B :: 'a::\mathit{at\text{-}base}\ \mathit{list}) \Longrightarrow \mathit{set}\ A \subseteq \mathit{set}\ B \Longrightarrow x \sharp{*}\ A$
  by (*metis fresh-star-set fresh-star-subset*[*OF finite-set*])

8.7 The set of free variables of an expression

definition $\mathit{fv} :: 'a::\mathit{pt} \Rightarrow 'b::\mathit{at\text{-}base}\ \mathit{set}$
  where $\mathit{fv}\ e = \{v.\ \mathit{atom}\ v \in \mathit{supp}\ e\}$

lemma *fv-eqvt*[*simp*,*eqvt*]: $\pi \cdot (\mathit{fv}\ e) = \mathit{fv}\ (\pi \cdot e)$

unfolding *fv-def* **by** *simp*

lemma *fv-Nil*[*simp*]: *fv* [] = {}
  by (*auto simp add*: *fv-def supp-Nil*)
lemma *fv-Cons*[*simp*]: *fv* (*x* # *xs*) = *fv x* ∪ *fv xs*
  by (*auto simp add*: *fv-def supp-Cons*)
lemma *fv-Pair*[*simp*]: *fv* (*x*, *y*) = *fv x* ∪ *fv y*
  by (*auto simp add*: *fv-def supp-Pair*)
lemma *fv-append*[*simp*]: *fv* (*x* @ *y*) = *fv x* ∪ *fv y*
  by (*auto simp add*: *fv-def supp-append*)
lemma *fv-at-base*[*simp*]: *fv a* = {*a*::*'a*::*at-base*}
  by (*auto simp add*: *fv-def supp-at-base*)
lemma *fv-pure*[*simp*]: *fv* (*a*::*'a*::*pure*) = {}
  by (*auto simp add*: *fv-def pure-supp*)

lemma *fv-set-at-base*[*simp*]: *fv* (*l* :: (*'a* :: *at-base*) *list*) = *set l*
  by (*induction l*) *auto*

lemma *flip-not-fv*: *a* ∉ *fv x* ⟹ *b* ∉ *fv x* ⟹ (*a* ↔ *b*) • *x* = *x*
  by (*metis flip-def fresh-def fv-def mem-Collect-eq swap-fresh-fresh*)

lemma *fv-not-fresh*: *atom x* ♯ *e* ⟷ *x* ∉ *fv e*
  unfolding *fv-def fresh-def* **by** *blast*

lemma *fresh-fv*: *finite* (*fv e* :: *'a set*) ⟹ *atom* (*x* :: (*'a*::*at-base*)) ♯ (*fv e* :: *'a set*) ⟷ *atom x* ♯ *e*
  unfolding *fv-def fresh-def*
  by (*auto simp add*: *supp-finite-set-at-base*)

lemma *finite-fv*[*simp*]: *finite* (*fv* (*e*::*'a*::*fs*) :: (*'b*::*at-base*) *set*)
proof−
  have *finite* (*supp e*) **by** (*metis finite-supp*)
  hence *finite* (*atom* −` *supp e* :: *'b set*)
    apply (*rule finite-vimageI*)
    apply (*rule inj-onI*)
    apply (*simp*)
    done
  moreover
  have (*atom* −` *supp e* :: *'b set*) = *fv e* **unfolding** *fv-def* **by** *auto*
  ultimately
  show *?thesis* **by** *simp*
qed

definition *fv-list* :: *'a*::*fs* ⇒ *'b*::*at-base list*
  where *fv-list e* = (*SOME l*. *set l* = *fv e*)

lemma *set-fv-list*[*simp*]: *set* (*fv-list e*) = (*fv e* :: (*'b*::*at-base*) *set*)
proof−
  have *finite* (*fv e* :: *'b set*) **by** (*rule finite-fv*)

from *finite-list*[*OF finite-fv*]
obtain l **where** $set\ l = (fv\ e :: 'b\ set)$**..**
thus *?thesis*
unfolding *fv-list-def* **by** (*rule someI*)
qed

lemma *fresh-fv-list*[*simp*]:
$a \sharp (fv\text{-}list\ e :: 'b::at\text{-}base\ list) \longleftrightarrow a \sharp (fv\ e :: 'b::at\text{-}base\ set)$
proof−
have $a \sharp (fv\text{-}list\ e :: 'b::at\text{-}base\ list) \longleftrightarrow a \sharp set\ (fv\text{-}list\ e :: 'b::at\text{-}base\ list)$
by (*rule fresh-set*[*symmetric*])
also have $\ldots \longleftrightarrow a \sharp (fv\ e :: 'b::at\text{-}base\ set)$ **by** *simp*
finally show *?thesis***.**
qed

8.8 Other useful lemmas

lemma *pure-permute-id*: $permute\ p = (\lambda\ x.\ (x::'a::pure))$
by *rule* (*simp add*: *permute-pure*)

lemma *supp-set-elem-finite*:
assumes *finite S*
and $(m::'a::fs) \in S$
and $y \in supp\ m$
shows $y \in supp\ S$
using *assms supp-of-finite-sets*
by *auto*

lemmas *fresh-star-Cons* = *fresh-star-list*(*2*)

lemma *mem-permute-set*:
shows $x \in p \cdot S \longleftrightarrow (- p \cdot x) \in S$
by (*metis mem-permute-iff permute-minus-cancel*(*2*))

lemma *flip-set-both-not-in*:
assumes $x \notin S$ **and** $x' \notin S$
shows $((x' \leftrightarrow x) \cdot S) = S$
unfolding *permute-set-def*
by (*auto*) (*metis assms flip-at-base-simps*(*3*))+

lemma *inj-atom*: *inj atom* **by** (*metis atom-eq-iff injI*)

lemmas *image-Int*[*OF inj-atom*, *simp*]

lemma *eqvt-uncurry*: $eqvt\ f \Longrightarrow eqvt\ (case\text{-}prod\ f)$
unfolding *eqvt-def*
by *perm-simp simp*

lemma *supp-fun-app-eqvt2*:

assumes *a*: *eqvt f*
shows *supp* (*f x y*) ⊆ *supp x* ∪ *supp y*
proof−
from *supp-fun-app-eqvt*[*OF eqvt-uncurry* [*OF a*]]
have *supp* (*case-prod f* (*x*,*y*)) ⊆ *supp* (*x*,*y*)**.**
thus *?thesis* **by** (*simp add*: *supp-Pair*)
qed

lemma *supp-fun-app-eqvt3*:
assumes *a*: *eqvt f*
shows *supp* (*f x y z*) ⊆ *supp x* ∪ *supp y* ∪ *supp z*
proof−
from *supp-fun-app-eqvt2*[*OF eqvt-uncurry* [*OF a*]]
have *supp* (*case-prod f* (*x*,*y*) *z*) ⊆ *supp* (*x*,*y*) ∪ *supp z***.**
thus *?thesis* **by** (*simp add*: *supp-Pair*)
qed

lemma *permute-0*[*simp*]: *permute 0* = (λ *x*. *x*)
by *auto*
lemma *permute-comp*[*simp*]: *permute x* ∘ *permute y* = *permute* (*x* + *y*) **by** *auto*

lemma *map-permute*: *map* (*permute p*) = *permute p*
apply *rule*
apply (*induct-tac x*)
apply *auto*
done

lemma *fresh-star-restrictA*[*intro*]: *a* ♯∗ Γ ⟹ *a* ♯∗ *AList.restrict V* Γ
by (*induction* Γ) (*auto simp add*: *fresh-star-Cons*)

lemma *Abs-lst-Nil-eq*[*simp*]: [[]]*lst*. (*x*::′*a*::*fs*) = [*xs*]*lst*. *x*′ ⟷ (([],*x*) = (*xs*, *x*′))
apply *rule*
apply (*frule Abs-lst-fcb2*[**where** *f* = λ *x y* - . (*x*,*y*) **and** *as* = [] **and** *bs* = *xs* **and** *c* = ()])
apply (*auto simp add*: *fresh-star-def*)
done

lemma *Abs-lst-Nil-eq2*[*simp*]: [*xs*]*lst*. (*x*::′*a*::*fs*) = [[]]*lst*. *x*′ ⟷ ((*xs*,*x*) = ([], *x*′))
by (*subst eq-commute*) *auto*

end

9 AList-Utils-Nominal.tex

theory *AList−Utils−Nominal*
imports *AList−Utils Nominal−Utils*
begin

9.1 Freshness lemmas related to associative lists

lemma *domA-not-fresh*:
$x \in domA\ \Gamma \Longrightarrow \neg(atom\ x\ \sharp\ \Gamma)$
by (*induct* Γ, *auto simp add*: *fresh-Cons fresh-Pair*)

lemma *fresh-delete*:
assumes $a\ \sharp\ \Gamma$
shows $a\ \sharp\ delete\ v\ \Gamma$
using *assms*
by(*induct* Γ)(*auto simp add*: *fresh-Cons*)

lemma *fresh-star-delete*:
assumes $S\ \sharp{*}\ \Gamma$
shows $S\ \sharp{*}\ delete\ v\ \Gamma$
using *assms fresh-delete* **unfolding** *fresh-star-def* **by** *fastforce*

lemma *fv-delete-subset*:
$fv\ (delete\ v\ \Gamma) \subseteq fv\ \Gamma$
using *fresh-delete* **unfolding** *fresh-def fv-def* **by** *auto*

lemma *fresh-heap-expr*:
assumes $a\ \sharp\ \Gamma$
and $(x,e) \in set\ \Gamma$
shows $a\ \sharp\ e$
using *assms*
by (*metis fresh-list-elem fresh-Pair*)

lemma *fresh-heap-expr'*:
assumes $a\ \sharp\ \Gamma$
and $e \in snd\ `\ set\ \Gamma$
shows $a\ \sharp\ e$
using *assms*
by (*induct* Γ, *auto simp add*: *fresh-Cons fresh-Pair*)

lemma *fresh-star-heap-expr'*:
assumes $S\ \sharp{*}\ \Gamma$
and $e \in snd\ `\ set\ \Gamma$
shows $S\ \sharp{*}\ e$
using *assms*
by (*metis fresh-star-def fresh-heap-expr'*)

lemma *fresh-map-of*:
assumes $x \in domA\ \Gamma$

assumes $a \sharp \Gamma$
shows $a \sharp$ *the* (*map-of* Γ x)
using *assms*
by (*induct* Γ)(*auto simp add*: *fresh-Cons fresh-Pair*)

lemma *fresh-star-map-of*:
assumes $x \in$ *domA* Γ
assumes $a \sharp* \Gamma$
shows $a \sharp*$ *the* (*map-of* Γ x)
using *assms* **by** (*simp add*: *fresh-star-def fresh-map-of*)

lemma *domA-fv-subset*: *domA* $\Gamma \subseteq$ *fv* Γ
by (*induction* Γ) *auto*

lemma *map-of-fv-subset*: $x \in$ *domA* $\Gamma \Longrightarrow$ *fv* (*the* (*map-of* Γ x)) $\subseteq$ *fv* Γ
by (*induction* Γ) *auto*

lemma *map-of-Some-fv-subset*: *map-of* Γ x = *Some* $e \Longrightarrow$ *fv* $e \subseteq$ *fv* Γ
by (*metis domA-from-set map-of-fv-subset map-of-SomeD option.sel*)

9.2 Equivariance lemmas

lemma *domA*[*eqvt*]:
$\pi \cdot$ *domA* Γ = *domA* ($\pi \cdot \Gamma$)
by (*simp add*: *domA-def*)

lemma *mapCollect*[*eqvt*]:
$\pi \cdot$ *mapCollect f m* = *mapCollect* ($\pi \cdot f$) ($\pi \cdot m$)
unfolding *mapCollect-def*
by *perm-simp rule*

9.3 Freshness and distinctness

lemma *fresh-distinct*:
assumes *atom* ‘ $S \sharp* \Gamma$
shows $S \cap$ *domA* $\Gamma = \{\}$
proof−
{ fix x
assume $x \in S$
moreover
assume $x \in$ *domA* Γ
hence *atom* $x \in$ *supp* Γ
by (*induct* Γ)(*auto simp add*: *supp-Cons domA-def supp-Pair supp-at-base*)
ultimately
have *False*
using *assms*
by (*simp add*: *fresh-star-def fresh-def*)
}
thus $S \cap$ *domA* $\Gamma = \{\}$ **by** *auto*
qed

lemma *fresh-distinct-list*:
assumes *atom* ‘ S $\sharp*$ l
shows $S \cap$ *set* $l = \{\}$
using *assms*
by (*metis disjoint-iff-not-equal fresh-list-elem fresh-star-def image-eqI not-self-fresh*)

lemma *fresh-distinct-fv*:
assumes *atom* ‘ S $\sharp*$ l
shows $S \cap$ *fv* $l = \{\}$
using *assms*
by (*metis disjoint-iff-not-equal fresh-star-def fv-not-fresh image-eqI*)

9.4 Pure codomains

lemma *domA-fv-pure*:
fixes Γ :: (*'a::at-base* $\times$ *'b::pure*) *list*
shows *fv* Γ = *domA* Γ
apply (*induct* Γ)
apply *simp*
apply (*case-tac a*)
apply (*simp*)
done

lemma *domA-fresh-pure*:
fixes Γ :: (*'a::at-base* $\times$ *'b::pure*) *list*
shows $x \in$ *domA* $\Gamma \longleftrightarrow \neg$(*atom* x $\sharp$ Γ)
unfolding *domA-fv-pure*[*symmetric*]
by (*auto simp add*: *fv-def fresh-def*)

end

10 Nominal-HOLCF.tex

theory *Nominal−HOLCF*
imports
Nominal−Utils HOLCF−Utils
begin

10.1 Type class of continuous permutations and variations thereof

class *cont-pt* =
cpo +
pt +
assumes *perm-cont*: $\bigwedge p.$ *cont* ((*permute p*) :: *'a::{cpo,pt}* $\Rightarrow$ *'a*)

class *discr-pt* =
discrete-cpo +

```
  pt

class pcpo-pt =
  cont-pt +
  pcpo

instance pcpo-pt ⊆ cont-pt
 by standard (auto intro: perm-cont)

instance discr-pt ⊆ cont-pt
 by standard auto

lemma (in cont-pt) perm-cont-simp[simp]: π • x ⊑ π • y ⟷ x ⊑ y
  apply rule
  apply (drule cont2monofunE[OF perm-cont, of - - −π], simp)[1]
  apply (erule cont2monofunE[OF perm-cont, of - - π])
  done

lemma (in cont-pt) perm-below-to-right: π • x ⊑ y ⟷ x ⊑ − π •  y
  by (metis perm-cont-simp pt-class.permute-minus-cancel(2))

lemma perm-is-ub-simp[simp]: π • S <| π • (x::'a::cont-pt) ⟷ S <| x
  by (auto simp add: is-ub-def permute-set-def)

lemma perm-is-ub-eqvt[simp,eqvt]: S <| (x::'a::cont-pt) ⟹ π • S <| π • x
  by simp

lemma perm-is-lub-simp[simp]: π • S <<| π • (x::'a::cont-pt) ⟷ S <<| x
  apply (rule perm-rel-lemma)
  by (metis is-lubI is-lubD1 is-lubD2 perm-cont-simp perm-is-ub-simp)

lemma perm-is-lub-eqvt[simp,eqvt]: S <<| (x::'a::cont-pt) ==> π • S <<| π • x
  by simp

lemmas perm-cont2cont[simp,cont2cont] = cont-compose[OF perm-cont]

lemma perm-still-cont: cont (π • f) = cont (f :: ('a :: cont-pt) ⇒ ('b :: cont-pt))
proof
  have imp:⋀ (f :: 'a ⇒ 'b) π. cont f ⟹ cont (π • f)
    unfolding permute-fun-def
    by (metis cont-compose perm-cont)
  show cont f ⟹ cont (π • f) using imp[of f π].
  show cont (π • f) ⟹ cont (f) using imp[of π • f −π] by simp
qed

lemma perm-bottom[simp,eqvt]: π • ⊥ = (⊥::'a::{cont-pt,pcpo})
  proof−
  have ⊥ ⊑ −π • (⊥::'a::{cont-pt,pcpo}) by simp
  hence π • ⊥ ⊑ π • (−π • (⊥::'a::{cont-pt,pcpo})) by(rule cont2monofunE[OF perm-cont])
```

hence $\pi \cdot \bot \sqsubseteq (\bot::'a::\{cont\text{-}pt,pcpo\})$ **by** *simp*
thus $\pi \cdot \bot = (\bot::'a::\{cont\text{-}pt,pcpo\})$ **by** *simp*
qed

lemma *bot-supp*[*simp*]: *supp* $(\bot :: 'a :: pcpo\text{-}pt) = \{\}$
by (*rule supp-fun-eqvt*) (*simp add*: *eqvt-def*)

lemma *bot-fresh*[*simp*]: $a \sharp (\bot :: 'a :: pcpo\text{-}pt)$
by (*simp add*: *fresh-def*)

lemma *bot-fresh-star*[*simp*]: $a \sharp* (\bot :: 'a :: pcpo\text{-}pt)$
by (*simp add*: *fresh-star-def*)

lemma *below-eqvt* [*eqvt*]:
$\pi \cdot (x \sqsubseteq y) = (\pi \cdot x \sqsubseteq \pi \cdot (y::'a::cont\text{-}pt))$ **by** (*auto simp add*: *permute-pure*)

lemma *lub-eqvt*[*simp*]:
$(\exists\ z.\ S <<|\ (z::'a::\{cont\text{-}pt\})) \Longrightarrow \pi \cdot lub\ S = lub\ (\pi \cdot S)$
by (*metis lub-eqI perm-is-lub-simp*)

lemma *chain-eqvt*[*eqvt*]:
fixes $F :: nat \Rightarrow 'a::cont\text{-}pt$
shows $chain\ F \Longrightarrow chain\ (\pi \cdot F)$
apply (*rule chainI*)
apply (*drule-tac* $i = i$ **in** *chainE*)
apply (*subst* (*asm*) *perm-cont-simp*[*symmetric*, **where** $\pi = \pi$])
by (*metis permute-fun-app-eq permute-pure*)

10.2 Instance for *cfun*

instantiation *cfun* :: (*cont-pt*, *cont-pt*) *pt*
begin
definition $p \cdot (f :: 'a \rightarrow 'b) = (\Lambda\ x.\ p \cdot (f \cdot (-\ p \cdot x)))$

instance
apply *standard*
apply (*simp add*: *permute-cfun-def eta-cfun*)
apply (*simp add*: *permute-cfun-def cfun-eqI minus-add*)
done
end

lemma *permute-cfun-eq*: $permute\ p = (\lambda\ f.\ (Abs\text{-}cfun\ (permute\ p))\ oo\ f\ oo\ (Abs\text{-}cfun\ (permute\ (-p))))$
by (*rule*, *rule cfun-eqI*, *auto simp add*: *permute-cfun-def*)

lemma *Cfun-app-eqvt*[*eqvt*]:
$\pi \cdot (f \cdot x) = (\pi \cdot f) \cdot (\pi \cdot x)$
unfolding *permute-cfun-def*
by *auto*

lemma *permute-Lam*: *cont f* $\Longrightarrow p \bullet (\Lambda\ x.\ f\ x) = (\Lambda\ x.\ (p \bullet f)\ x)$
  apply (*rule cfun-eqI*)
  unfolding *permute-cfun-def*
  by (*metis Abs-cfun-inverse2 eqvt-lambda unpermute-def*)

lemma *Abs-cfun-eqvt*: *cont f* $\Longrightarrow (p \bullet \mathit{Abs\text{-}cfun})\ f = \mathit{Abs\text{-}cfun}\ f$
  apply (*subst permute-fun-def*)
  by (*metis permute-Lam perm-still-cont permute-minus-cancel*(*1*))

lemma *cfun-eqvtI*: $(\bigwedge x.\ p \bullet (f \cdot x) = f' \cdot (p \bullet x)) \Longrightarrow p \bullet f = f'$
  by (*metis Cfun-app-eqvt cfun-eqI permute-minus-cancel*(*1*))

lemma *ID-eqvt*[*eqvt*]: $\pi \bullet \mathit{ID} = \mathit{ID}$
  unfolding *ID-def*
  apply *perm-simp*
  apply (*simp add*: *Abs-cfun-eqvt*)
  done

instance *cfun* :: (*cont-pt*, *cont-pt*) *cont-pt*
  by *standard* (*subst permute-cfun-eq*, *auto*)

instance *cfun* :: ({*pure*,*cont-pt*}, {*pure*,*cont-pt*}) *pure*
  by *standard* (*auto simp add*: *permute-cfun-def permute-pure Cfun.cfun.Rep-cfun-inverse*)

instance *cfun* :: (*cont-pt*, *pcpo-pt*) *pcpo-pt*
  by *standard*

10.3 Instance for *fun*

lemma *permute-fun-eq*: *permute p* $= (\lambda\ f.\ (\mathit{permute}\ p) \circ f \circ (\mathit{permute}\ (-p)))$
  by (*rule*, *rule*, *metis comp-apply eqvt-lambda unpermute-def*)

instance *fun* :: (*pt*, *cont-pt*) *cont-pt*
  apply *standard*
  apply (*rule cont2cont-lambda*)
  apply (*subst permute-fun-def*)
  apply (*rule perm-cont2cont*)
  apply (*rule cont-fun*)
  done

lemma *fix-eqvt*[*eqvt*]:
  $\pi \bullet \mathit{fix} = (\mathit{fix} :: ('a \to 'a) \to 'a{::}\{\mathit{cont\text{-}pt},\mathit{pcpo}\})$
apply (*rule cfun-eqI*)
apply (*subst permute-cfun-def*)
apply *simp*
apply (*rule parallel-fix-ind*[*OF adm-subst2*])
apply (*auto simp add*: *permute-self*)
done

10.4 Instance for u

instantiation u :: ($cont\text{-}pt$) pt
begin
  definition $p \bullet (x :: 'a\ u) = fup\cdot(\Lambda\ x.\ up\cdot(p \bullet x))\cdot x$
  instance
  apply *standard*
  apply (*case-tac x*) **apply** (*auto simp add*: *permute-u-def*)
  apply (*case-tac x*) **apply** (*auto simp add*: *permute-u-def*)
  done
end

instance u :: (*cont-pt*) *cont-pt*
proof
  fix p

  have $permute\ p = (\lambda\ x.\ fup\cdot(\Lambda\ x.\ up\cdot(p \bullet x))\cdot(x{::}\ 'a\ u))$
    by (*rule ext*, *rule permute-u-def*)
  moreover have $cont\ (\lambda\ x.\ fup\cdot(\Lambda\ x.\ up\cdot(p \bullet x))\cdot(x{::}\ 'a\ u))$ **by** *simp*
  ultimately show $cont\ (permute\ p :: 'a\ u \Rightarrow 'a\ u)$ **by** *simp*
qed

instance u :: (*cont-pt*) *pcpo-pt* **..**

class *pure-cont-pt* = *pure* + *cont-pt*

instance u :: (*pure-cont-pt*) *pure*
  apply *standard*
  apply (*case-tac x*)
  apply (*auto simp add*: *permute-u-def permute-pure*)
  done

lemma *up-eqvt*[*eqvt*]: $\pi \bullet up = up$
  apply (*rule cfun-eqI*)
  apply (*subst permute-cfun-def*, *simp*)
  apply (*simp add*: *permute-u-def*)
  done

lemma *fup-eqvt*[*eqvt*]: $\pi \bullet fup = fup$
  apply (*rule cfun-eqI*)
  apply (*rule cfun-eqI*)
  apply (*subst permute-cfun-def*, *simp*)
  apply (*subst permute-cfun-def*, *simp*)
  apply (*case-tac xa*)
  apply *simp*
  apply (*simp add*: *permute-self*)
  done

10.5 Instance for *lift*

instantiation *lift* :: (*pt*) *pt*
begin
  definition $p \cdot (x :: 'a\ lift) = case\text{-}lift\ \bot\ (\lambda\ x.\ Def\ (p \cdot x))\ x$
  instance
  apply *standard*
  apply (*case-tac x*) **apply** (*auto simp add*: *permute-lift-def*)
  apply (*case-tac x*) **apply** (*auto simp add*: *permute-lift-def*)
  done
end

instance *lift* :: (*pt*) *cont-pt*
proof
  fix *p*

  have $permute\ p = (\lambda\ x.\ case\text{-}lift\ \bot\ (\lambda\ x.\ Def\ (p \cdot x))\ (x{::}'a\ lift))$
    by (*rule ext*, *rule permute-lift-def*)
  moreover have $cont\ (\lambda\ x.\ case\text{-}lift\ \bot\ (\lambda\ x.\ Def\ (p \cdot x))\ (x{::}'a\ lift))$ **by** *simp*
  ultimately show $cont\ (permute\ p :: 'a\ lift \Rightarrow 'a\ lift)$ **by** *simp*
qed

instance *lift* :: (*pt*) *pcpo-pt* **..**

instance *lift* :: (*pure*) *pure*
  apply *standard*
  apply (*case-tac x*)
  apply (*auto simp add*: *permute-lift-def permute-pure*)
  done

lemma *Def-eqvt*[*eqvt*]: $\pi \cdot (Def\ x) = Def\ (\pi \cdot x)$
  by (*simp add*: *permute-lift-def*)

lemma *case-lift-eqvt*[*eqvt*]: $\pi \cdot case\text{-}lift\ d\ f\ x = case\text{-}lift\ (\pi \cdot d)\ (\pi \cdot f)\ (\pi \cdot x)$
  by (*cases x*) (*auto simp add*: *permute-self*)

10.6 Instance for *prod*

instance *prod* :: (*cont-pt*, *cont-pt*) *cont-pt*
proof
  fix *p*

  have $permute\ p = (\lambda\ (x :: ('a,\ 'b)\ prod).\ (p \cdot fst\ x,\ p \cdot snd\ x))$ **by** *auto*
  moreover have *cont* ... **by** (*intro cont2cont*)
  ultimately show $cont\ (permute\ p :: ('a,'b)\ prod\ \Rightarrow ('a,'b)\ prod)$ **by** *simp*
qed

end

11 Env-HOLCF.tex

theory *Env−HOLCF*
  imports *Env HOLCF−Utils*
begin

11.1 Continuity and pcpo-valued functions

lemma *override-on-belowI*:
  assumes $\bigwedge a.\ a \in S \Longrightarrow y\ a \sqsubseteq z\ a$
  and $\bigwedge a.\ a \notin S \Longrightarrow x\ a \sqsubseteq z\ a$
  shows $x \mathbin{++_S} y \sqsubseteq z$
  using *assms*
  apply −
  apply (*rule fun-belowI*)
  apply (*case-tac xa* ∈ *S*)
  apply *auto*
  done

lemma *override-on-cont1*: *cont* $(\lambda\ x.\ x \mathbin{++_S} m)$
  by (*rule cont2cont-lambda*) (*auto simp add*: *override-on-def*)

lemma *override-on-cont2*: *cont* $(\lambda\ x.\ m \mathbin{++_S} x)$
  by (*rule cont2cont-lambda*) (*auto simp add*: *override-on-def*)

lemma *override-on-cont2cont*[*simp*, *cont2cont*]:
  assumes *cont f*
  assumes *cont g*
  shows *cont* $(\lambda\ x.\ f\ x \mathbin{++_S} g\ x)$
by (*rule cont-apply*[*OF assms*(*1*) *override-on-cont1 cont-compose*[*OF override-on-cont2 assms*(*2*)]])

lemma *override-on-mono*:
  assumes $x1 \sqsubseteq (x2 :: 'a::type \Rightarrow 'b::cpo)$
  assumes $y1 \sqsubseteq y2$
  shows $x1 \mathbin{++_S} y1 \sqsubseteq x2 \mathbin{++_S} y2$
by (*rule below-trans*[*OF cont2monofunE*[*OF override-on-cont1 assms*(*1*)] *cont2monofunE*[*OF override-on-cont2 assms*(*2*)]]])

lemma *fun-upd-below-env-deleteI*:
  assumes *env-delete x* $\varrho \sqsubseteq$ *env-delete x* ϱ'
  assumes $y \sqsubseteq \varrho'\ x$
  shows $\varrho(x := y) \sqsubseteq \varrho'$
  using *assms*
  apply (*auto intro*!: *fun-upd-belowI* *simp add*: *env-delete-def*)
  by (*metis fun-belowD fun-upd-other*)

lemma *fun-upd-belowI2*:
  assumes $\bigwedge z\ .\ z \neq x \Longrightarrow \varrho\ z \sqsubseteq \varrho'\ z$
  assumes $\varrho\ x \sqsubseteq y$
  shows $\varrho \sqsubseteq \varrho'(x := y)$

apply (*rule fun-belowI*)
using *assms* **by** *auto*

lemma *env-restr-belowI*:
assumes $\bigwedge x.\ x \in S \Longrightarrow (m1\ f|`\ S)\ x \sqsubseteq (m2\ f|`\ S)\ x$
shows $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
apply (*rule fun-belowI*)
by (*metis assms below-bottom-iff lookup-env-restr-not-there*)

lemma *env-restr-belowI2*:
assumes $\bigwedge x.\ x \in S \Longrightarrow m1\ x \sqsubseteq m2\ x$
shows $m1\ f|`\ S \sqsubseteq m2$
by (*rule fun-belowI*)
(*simp add*: *assms env-restr-def*)

lemma *env-restr-below-itself*:
shows $m\ f|`\ S \sqsubseteq m$
apply (*rule fun-belowI*)
apply (*case-tac* $x \in S$)
apply *auto*
done

lemma *env-restr-cont*: *cont* (*env-restr S*)
apply (*rule cont2cont-lambda*)
apply (*case-tac* $y \in S$)
apply *auto*
done

lemma *env-restr-belowD*:
assumes $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
assumes $x \in S$
shows $m1\ x \sqsubseteq m2\ x$
using *fun-belowD*[*OF assms*(*1*), **where** $x = x$] *assms*(*2*) **by** *simp*

lemma *env-restr-eqD*:
assumes $m1\ f|`\ S = m2\ f|`\ S$
assumes $x \in S$
shows $m1\ x = m2\ \ x$
by (*metis assms*(*1*) *assms*(*2*) *lookup-env-restr*)

lemma *env-restr-below-subset*:
assumes $S \subseteq S'$
and $m1\ f|`\ S' \sqsubseteq m2\ f|`\ S'$
shows $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
using *assms*
by (*auto intro*!: *env-restr-belowI dest*: *env-restr-belowD*)

lemma *override-on-below-restrI*:
  assumes $x\, f|`\ (-S) \sqsubseteq z\, f|`\ (-S)$
  and $y\, f|`\ S \sqsubseteq z\, f|`\ S$
  shows $x\ {++}_S\ y \sqsubseteq z$
using *assms*
by (*auto intro*: *override-on-belowI dest*:*env-restr-belowD*)

lemma *fmap-below-add-restrI*:
  assumes $x\, f|`\ (-S) \sqsubseteq y\, f|`\ (-S)$
  and $x\, f|`\ S \sqsubseteq z\, f|`\ S$
  shows $x \sqsubseteq y\ {++}_S\ z$
using *assms*
by (*auto intro*!: *fun-belowI dest*:*env-restr-belowD simp add*: *lookup-override-on-eq*)

lemmas *env-restr-cont2cont*[*simp*,*cont2cont*] = *cont-compose*[*OF env-restr-cont*]

lemma *env-delete-cont*: *cont* (*env-delete x*)
  apply (*rule cont2cont-lambda*)
  apply (*case-tac* $y = x$)
  apply *auto*
  done
lemmas *env-delete-cont2cont*[*simp*,*cont2cont*] = *cont-compose*[*OF env-delete-cont*]

end

12 HasESem.tex

theory *HasESem*
imports *Nominal−HOLCF Env−HOLCF*
begin

A local to work abstract in the expression type and semantics.

locale *has-ESem* =
  fixes *ESem* :: *'exp*::*pt* ⇒ (*'var*::*at-base* ⇒ *'value*) → *'value*::{*pure*,*pcpo*}
begin
  abbreviation *ESem-syn* ($[\![$ - $]\!]_{\_}$ [*0*,*0*] *110*) **where** $[\![e]\!]_\varrho \equiv$ *ESem* $e \cdot \varrho$
end

locale *has-ignore-fresh-ESem* = *has-ESem* +
  assumes *fv-supp*: *supp e* = *atom* ` (*fv e* :: *'b set*)
  assumes *ESem-considers-fv*: $[\![\ e\ ]\!]_\varrho = [\![\ e\ ]\!]_{\varrho\ f|`\ (fv\ e)}$

end

13 Iterative.tex

theory *Iterative*
imports *Env−HOLCF*
begin

A setup for defining a fixed point of mutual recursive environments iteratively

locale *iterative* =
  fixes ϱ :: $'a{::}type \Rightarrow {'b}{::}pcpo$
   and $e1$ :: $('a \Rightarrow 'b) \rightarrow ('a \Rightarrow 'b)$
   and $e2$ :: $('a \Rightarrow 'b) \rightarrow 'b$
   and S :: $'a\ set$ **and** x :: $'a$
  assumes $ne{:}x \notin S$
begin
  abbreviation $L == (\Lambda\ \varrho'.\ (\varrho\ {+}{+}_S\ e1 \cdot \varrho')(x := e2 \cdot \varrho'))$
  abbreviation $H == (\lambda\ \varrho'.\ \Lambda\ \varrho''.\ \varrho'\ {+}{+}_S\ e1 \cdot \varrho'')$
  abbreviation $R == (\Lambda\ \varrho'.\ (\varrho\ {+}{+}_S\ (\mathit{fix} \cdot (H\ \varrho')))(x := e2 \cdot \varrho'))$
  abbreviation $R' == (\Lambda\ \varrho'.\ (\varrho\ {+}{+}_S\ (\mathit{fix} \cdot (H\ \varrho')))(x := e2 \cdot (\mathit{fix} \cdot (H\ \varrho'))))$

  lemma *split-x*:
    fixes y
    obtains $y = x$ **and** $y \notin S \mid y \in S$ **and** $y \neq x \mid y \notin S$ **and** $y \neq x$ **using** *ne* **by** *blast*
  lemmas *below* = *fun-belowI*[*OF split-x*, **where** $y1 = \lambda x.\ x$]
  lemmas *eq* = *ext*[*OF split-x*, **where** $y1 = \lambda x.\ x$]

  lemma *lookup-fix*[*simp*]:
    fixes y **and** F :: $('a \Rightarrow 'b) \rightarrow ('a \Rightarrow 'b)$
    shows $(\mathit{fix} \cdot F)\ y = (F \cdot (\mathit{fix} \cdot F))\ y$
    by (*subst fix-eq*, *rule*)

  lemma *R-S*: $\bigwedge\ y.\ y \in S \Longrightarrow (\mathit{fix} \cdot R)\ y = (e1 \cdot (\mathit{fix} \cdot (H\ (\mathit{fix} \cdot R))))\ y$
    by (*case-tac y rule*: *split-x*) *simp-all*

  lemma *R'-S*: $\bigwedge\ y.\ y \in S \Longrightarrow (\mathit{fix} \cdot R')\ y = (e1 \cdot (\mathit{fix} \cdot (H\ (\mathit{fix} \cdot R'))))\ y$
    by (*case-tac y rule*: *split-x*) *simp-all*

  lemma *HR-is-R*[*simp*]: $\mathit{fix}{\cdot}(H\ (\mathit{fix} \cdot R)) = \mathit{fix} \cdot R$
    by (*rule eq*) *simp-all*

  lemma *HR'-is-R'*[*simp*]: $\mathit{fix} \cdot (H\ (\mathit{fix} \cdot R')) = \mathit{fix} \cdot R'$
    by (*rule eq*) *simp-all*

  lemma *H-noop*:
    fixes $\varrho'\ \varrho''$
    assumes $\bigwedge\ y.\ y \in S \Longrightarrow y \neq x \Longrightarrow (e1 \cdot \varrho'')\ y \sqsubseteq \varrho'\ y$
    shows $H\ \varrho' \cdot \varrho'' \sqsubseteq \varrho'$
      using *assms*
      by −(*rule below*, *simp-all*)

lemma *HL-is-L*[*simp*]: *fix* · (*H* (*fix* · *L*)) = *fix* · *L*
proof (*rule below-antisym*)
  show *fix* · (*H* (*fix* · *L*)) ⊑ *fix* · *L*
    by (*rule fix-least-below*[*OF H-noop*]) *simp*
  hence ∗: *e2* · (*fix* · (*H* (*fix* · *L*))) ⊑ *e2* · (*fix* · *L*) **by** (*rule monofun-cfun-arg*)

  show *fix* · *L* ⊑ *fix* · (*H* (*fix* · *L*))
    by (*rule fix-least-below*[*OF below*]) (*simp-all add*: *ne* ∗)
qed

lemma *iterative-override-on*:
  shows *fix* · *L* = *fix* · *R*
proof(*rule below-antisym*)
  show *fix* · *R* ⊑ *fix* · *L*
    by (*rule fix-least-below*[*OF below*]) *simp-all*

  show *fix* · *L* ⊑ *fix* · *R*
    apply (*rule fix-least-below*[*OF below*])
    apply *simp*
    apply (*simp del*: *lookup-fix add*: *R-S*)
    apply *simp*
    done
qed

lemma *iterative-override-on'*:
  shows *fix* · *L* = *fix* · *R'*
proof(*rule below-antisym*)
  show *fix* · *R'* ⊑ *fix* · *L*
    by (*rule fix-least-below*[*OF below*]) *simp-all*

  show *fix* · *L* ⊑ *fix* · *R'*
    apply (*rule fix-least-below*[*OF below*])
    apply *simp*
    apply (*simp del*: *lookup-fix add*: *R'-S*)
    apply *simp*
    done
qed
end

end

14 Env-Nominal.tex

theory *Env−Nominal*
  imports *Env Nominal−Utils Nominal−HOLCF*
begin

14.1 Equivariance lemmas

lemma *edom-perm*:
  fixes f :: $'a$::*pt* $\Rightarrow$ $'b$::{*pcpo-pt*}
  shows *edom* $(\pi \cdot f) = \pi \cdot$ (*edom* f)
  by (*simp add*: *edom-def*)

lemmas *edom-perm-rev*[*simp*,*eqvt*] = *edom-perm*[*symmetric*]

lemma *mem-edom-perm*[*simp*]:
  fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*}
  shows $xa \in$ *edom* $(p \cdot \varrho) \longleftrightarrow - p \cdot xa \in$ *edom* ϱ
  by (*metis* (*mono-tags*) *edom-perm-rev mem-Collect-eq permute-set-eq*)

lemma *env-restr-eqvt*[*eqvt*]:
  fixes m :: $'a$::*pt* $\Rightarrow$ $'b$::{*cont-pt*,*pcpo*}
  shows $\pi \cdot m\ f|\text{`}\ d = (\pi \cdot m)\ f|\text{`}\ (\pi \cdot d)$
  by (*auto simp add*: *env-restr-def*)

lemma *env-delete-eqvt*[*eqvt*]:
  fixes m :: $'a$::*pt* $\Rightarrow$ $'b$::{*cont-pt*,*pcpo*}
  shows $\pi \cdot$ *env-delete* x m = *env-delete* $(\pi \cdot x)$ $(\pi \cdot m)$
  by (*auto simp add*: *env-delete-def*)

lemma *esing-eqvt*[*eqvt*]: $\pi \cdot$ (*esing* x) = *esing* $(\pi \cdot x)$
  unfolding *esing-def*
  apply *perm-simp*
  apply (*simp add*: *Abs-cfun-eqvt*)
  done

14.2 Permutation and restriction

lemma *env-restr-perm*:
  fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*,*pure*}
  assumes *supp* p $\sharp\!*$ S **and** [*simp*]: *finite* S
  shows $(p \cdot \varrho)\ f|\text{`}\ S = \varrho\ f|\text{`}\ S$
using *assms*
apply −
apply (*rule ext*)
apply (*case-tac* $x \in S$)
apply (*simp*)
apply (*subst permute-fun-def*)
apply (*simp add*: *permute-pure*)
apply (*subst perm-supp-eq*)
apply (*auto simp add*:*perm-supp-eq supp-minus-perm fresh-star-def fresh-def supp-set-elem-finite*)
done

lemma *env-restr-perm'*:
  fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*,*pure*}
  assumes *supp* p $\sharp\!*$ S **and** [*simp*]: *finite* S

shows $p \cdot (\varrho\ f|^{\backprime}\ S) = \varrho\ f|^{\backprime}\ S$
by (*simp add*: *perm-supp-eq*[*OF assms*(*1*)] *env-restr-perm*[*OF assms*])

lemma *env-restr-flip*:
fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*,*pure*}
assumes $x \notin S$ **and** $x' \notin S$
shows $((x' \leftrightarrow x) \cdot \varrho)\ f|^{\backprime}\ S = \varrho\ f|^{\backprime}\ S$
using *assms*
apply −
apply *rule*
apply (*auto simp add*: *permute-flip-at env-restr-def split*:*if-splits*)
by (*metis eqvt-lambda flip-at-base-simps*(*3*) *minus-flip permute-pure unpermute-def*)

lemma *env-restr-flip′*:
fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*,*pure*}
assumes $x \notin S$ **and** $x' \notin S$
shows $(x' \leftrightarrow x) \cdot (\varrho\ f|^{\backprime}\ S) = \varrho\ f|^{\backprime}\ S$
by (*simp add*: *flip-set-both-not-in*[*OF assms*] *env-restr-flip*[*OF assms*])

14.3 Pure codomains

lemma *edom-fv-pure*:
fixes f :: ($'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo*,*pure*})
assumes *finite* (*edom f*)
shows *fv f* $\subseteq$ *edom f*
using *assms*
proof (*induction edom f arbitrary*: *f*)
case *empty*
hence $f = \bot$ **unfolding** *edom-def* **by** *auto*
thus *?case* **by** (*auto simp add*: *fv-def fresh-def supp-def*)
next
case (*insert x S*)
have $f = (\textit{env-delete}\ x\ f)(x := f\ x)$ **by** *auto*
hence $\textit{fv}\ f \subseteq \textit{fv}\ (\textit{env-delete}\ x\ f) \cup \textit{fv}\ x \cup \textit{fv}\ (f\ x)$
using *eqvt-fresh-cong3*[**where** $f = \textit{fun-upd}$ **and** $x = \textit{env-delete}\ x\ f$ **and** $y = x$ **and** $z = f\ x$, *OF fun-upd-eqvt*]
apply (*auto simp add*: *fv-def fresh-def*)
by (*metis fresh-def pure-fresh*)
also

from ‹*insert x S* = *edom f*› **and** ‹$x \notin S$›
have $S = \textit{edom}\ (\textit{env-delete}\ x\ f)$ **by** *auto*
hence $\textit{fv}\ (\textit{env-delete}\ x\ f) \subseteq \textit{edom}\ (\textit{env-delete}\ x\ f)$ **by** (*rule insert*)
also
have $\textit{fv}\ (f\ x) = \{\}$ **by** (*rule fv-pure*)
also
from ‹*insert x S* = *edom f*› **have** $x \in \textit{edom}\ f$ **by** *auto*
hence $\textit{edom}\ (\textit{env-delete}\ x\ f) \cup \textit{fv}\ x \cup \{\} \subseteq \textit{edom}\ f$ **by** *auto*
finally

show *?case* **by** *this* (*intro Un-mono subset-refl*)
qed

end

15 HeapSemantics.tex

theory *HeapSemantics*
imports *EvalHeap AList−Utils−Nominal HasESem Iterative Env−Nominal*
begin

15.1 A locale for heap semantics, abstract in the expression semantics

context *has-ESem*
begin

abbreviation *EvalHeapSem-syn* ($[\![$ - $]\!]_{-}$ [*0*,*0*] *110*)
where *EvalHeapSem-syn* $\Gamma\ \varrho \equiv$ *evalHeap* $\Gamma\ (\lambda\ e.\ [\![e]\!]_{\varrho})$

definition
HSem :: $('var \times 'exp)\ list \Rightarrow ('var \Rightarrow 'value) \rightarrow ('var \Rightarrow 'value)$
where *HSem* $\Gamma = (\Lambda\ \varrho\ .\ (\mu\ \varrho'.\ \varrho\ {++}_{domA\ \Gamma}\ [\![\Gamma]\!]_{\varrho'}))$

abbreviation *HSem-syn* ($\{\!\vert$ - $\vert\!\}$- [*0*,*60*] *60*)
where $\{\!\vert\Gamma\vert\!\}\varrho \equiv$ *HSem* $\Gamma \cdot \varrho$

lemma *HSem-def'*: $\{\!\vert\Gamma\vert\!\}\varrho = (\mu\ \varrho'.\ \varrho\ {++}_{domA\ \Gamma}\ [\![\Gamma]\!]_{\varrho'})$
unfolding *HSem-def* **by** *simp*

15.2 Induction and other lemmas about *HSem*

lemma *HSem-ind*:
assumes *adm P*
assumes $P\ \bot$
assumes *step*: $\bigwedge\ \varrho'.\ P\ \varrho' \Longrightarrow\ P\ (\varrho\ {++}_{domA\ \Gamma}\ [\![\Gamma]\!]_{\varrho'})$
shows $P\ (\{\!\vert\Gamma\vert\!\}\varrho)$
unfolding *HSem-def'*
apply (*rule fix-ind*[*OF assms*(*1*), *OF assms*(*2*)])
using *step* **by** *simp*

lemma *HSem-below*:
assumes *rho*: $\bigwedge x.\ x \notin domA\ h \Longrightarrow \varrho\ x \sqsubseteq r\ x$
assumes *h*: $\bigwedge x.\ x \in domA\ h \Longrightarrow [\![the\ (map\text{-}of\ h\ x)]\!]_{r} \sqsubseteq r\ x$
shows $\{\!\vert h\vert\!\}\varrho \sqsubseteq r$
proof (*rule HSem-ind*, *goal-cases*)
case *1* **show** *?case* **by** (*auto*)

next
  case *2* **show** *?case* **by** (*rule minimal*)
next
  case (*3* ϱ')
    show *?case*
    by (*rule override-on-belowI*)
      (*auto simp add*: *lookupEvalHeap below-trans*[*OF monofun-cfun-arg*[*OF* ‹$\varrho' \sqsubseteq r$›] *h*] *rho*)
qed

lemma *HSem-bot-below*:
  assumes *h*: $\bigwedge x.\ x \in domA\ h \Longrightarrow [\![the\ (map\text{-}of\ h\ x)]\!]_r \sqsubseteq r\ x$
  shows $\{\!|h|\!\}\bot \sqsubseteq r$
  using *assms*
by (*metis HSem-below fun-belowD minimal*)

lemma *HSem-bot-ind*:
  assumes *adm P*
  assumes $P\ \bot$
  assumes *step*: $\bigwedge\ \varrho'.\ P\ \varrho' \Longrightarrow P\ ([\![\ \Gamma\ ]\!]_{\varrho'})$
  shows $P\ (\{\!|\Gamma|\!\}\bot)$
  apply (*rule HSem-ind*[*OF assms*(*1*,*2*)])
  apply (*drule assms*(*3*))
  apply *simp*
  done

lemma *parallel-HSem-ind*:
  assumes *adm* $(\lambda\varrho'.\ P\ (fst\ \varrho')\ (snd\ \varrho'))$
  assumes $P\ \bot\ \bot$
  assumes *step*: $\bigwedge y\ z.\ P\ y\ z \Longrightarrow$
    $P\ (\varrho_1 +\!+_{domA\ \Gamma_1}\ [\![\Gamma_1]\!]y)\ (\varrho_2 +\!+_{domA\ \Gamma_2}\ [\![\Gamma_2]\!]z)$
  shows $P\ (\{\!|\Gamma_1|\!\}\varrho_1)\ (\{\!|\Gamma_2|\!\}\varrho_2)$
  unfolding *HSem-def'*
  apply (*rule parallel-fix-ind*[*OF assms*(*1*), *OF assms*(*2*)])
  using *step* **by** *simp*

lemma *HSem-eq*:
  shows $\{\!|\Gamma|\!\}\varrho = \varrho +\!+_{domA\ \Gamma}\ [\![\Gamma]\!]_{\{\!|\Gamma|\!\}\varrho}$
  unfolding *HSem-def'*
  by (*subst fix-eq*) *simp*

lemma *HSem-bot-eq*:
  shows $\{\!|\Gamma|\!\}\bot = [\![\Gamma]\!]_{\{\!|\Gamma|\!\}\bot}$
  by (*subst HSem-eq*) *simp*

lemma *lookup-HSem-other*:
  assumes $y \notin domA\ h$
  shows $(\{\!|h|\!\}\varrho)\ y = \varrho\ y$
  apply (*subst HSem-eq*)
  using *assms* **by** *simp*

lemma *lookup-HSem-heap*:
  assumes $y \in domA\ h$
  shows $(\{\!\!\{h\}\!\!\}\varrho)\ y = [\![\ the\ (map\text{-}of\ h\ y)\ ]\!]_{\{\!\!\{h\}\!\!\}\varrho}$
  apply (*subst HSem-eq*)
  using *assms* **by** (*simp add*: *lookupEvalHeap*)

lemma *HSem-edom-subset*: $edom\ (\{\!\!\{\Gamma\}\!\!\}\varrho) \subseteq edom\ \varrho \cup domA\ \Gamma$
  apply *rule*
  unfolding *edomIff*
  apply (*case-tac* $x \in domA\ \Gamma$)
  apply (*auto simp add*: *lookup-HSem-other*)
  done

lemma *env-restr-override-onI*:$-S2 \subseteq S \Longrightarrow env\text{-}restr\ S\ \varrho 1 +\!+_{S2}\ \varrho 2 = \varrho 1 +\!+_{S2}\ \varrho 2$
  by (*rule ext*) (*auto simp add*: *lookup-override-on-eq*)

lemma *HSem-restr*:
  $\{\!\!\{h\}\!\!\}(\varrho\ f|`\ (-\ domA\ h)) = \{\!\!\{h\}\!\!\}\varrho$
  apply (*rule parallel-HSem-ind*)
  apply *simp*
  apply *auto*[*1*]
  apply (*subst env-restr-override-onI*)
  apply *simp-all*
  done

lemma *HSem-restr-cong*:
  assumes $\varrho\ f|`\ (-\ domA\ h) = \varrho'\ f|`\ (-\ domA\ h)$
  shows $\{\!\!\{h\}\!\!\}\varrho = \{\!\!\{h\}\!\!\}\varrho'$
  apply (*subst* (*1 2*) *HSem-restr*[*symmetric*])
  by (*simp add*: *assms*)

lemma *HSem-restr-cong-below*:
  assumes $\varrho\ f|`\ (-\ domA\ h) \sqsubseteq \varrho'\ f|`\ (-\ domA\ h)$
  shows $\{\!\!\{h\}\!\!\}\varrho \sqsubseteq \{\!\!\{h\}\!\!\}\varrho'$
  by (*subst* (*1 2*) *HSem-restr*[*symmetric*]) (*rule monofun-cfun-arg*[*OF assms*])

lemma *HSem-reorder*:
  assumes $map\text{-}of\ \Gamma = map\text{-}of\ \Delta$
  shows $\{\!\!\{\Gamma\}\!\!\}\varrho = \{\!\!\{\Delta\}\!\!\}\varrho$
by (*simp add*: *HSem-def′ evalHeap-reorder*[*OF assms*] *assms dom-map-of-conv-domA*[*symmetric*])

lemma *HSem-reorder-head*:
  assumes $x \neq y$
  shows $\{\!\!\{(x,e1)\#(y,e2)\#\Gamma\}\!\!\}\varrho = \{\!\!\{(y,e2)\#(x,e1)\#\Gamma\}\!\!\}\varrho$
proof−
  have $set\ ((x,e1)\#(y,e2)\#\Gamma) = set\ ((y,e2)\#(x,e1)\#\Gamma)$
    by *auto*
  thus *?thesis*

unfolding *HSem-def evalHeap-reorder-head*[*OF assms*]
by (*simp add*: *domA-def*)
qed

lemma *HSem-reorder-head-append*:
assumes $x \notin domA\ \Gamma$
shows $\{\!|(x,e)\#\Gamma @\Delta|\!\}\varrho = \{\!|\Gamma\ @\ ((x,e)\#\Delta)|\!\}\varrho$
proof−
have $set\ ((x,e)\#\Gamma@\Delta) = set\ (\Gamma\ @\ ((x,e)\#\Delta))$ **by** *auto*
thus *?thesis*
unfolding *HSem-def evalHeap-reorder-head-append*[*OF assms*]
by *simp*
qed

lemma *env-restr-HSem*:
assumes $domA\ \Gamma \cap S = \{\}$
shows $(\{\!|\ \Gamma\ |\!\}\varrho)\ f|`\ S = \varrho\ f|`\ S$
proof (*rule env-restr-eqI*)
fix x
assume $x \in S$
hence $x \notin domA\ \Gamma$ **using** *assms* **by** *auto*
thus $(\{\!|\ \Gamma\ |\!\}\varrho)\ x = \varrho\ x$
by (*rule lookup-HSem-other*)
qed

lemma *env-restr-HSem-noop*:
assumes $domA\ \Gamma \cap edom\ \varrho = \{\}$
shows $(\{\!|\ \Gamma\ |\!\}\varrho)\ f|`\ edom\ \varrho = \varrho$
by (*simp add*: *env-restr-HSem*[*OF assms*] *env-restr-useless*)

lemma *HSem-Nil*[*simp*]: $\{\!|[]|\!\}\varrho = \varrho$
by (*subst HSem-eq*, *simp*)

15.3 Substitution

lemma *HSem-subst-exp*:
assumes $\bigwedge\varrho'.\ [\![\ e\ ]\!]_{\varrho'} = [\![\ e'\ ]\!]_{\varrho'}$
shows $\{\!|(x,\ e)\ \#\ \Gamma|\!\}\varrho = \{\!|(x,\ e')\ \#\ \Gamma|\!\}\varrho$
by (*rule parallel-HSem-ind*) (*auto simp add*: *assms evalHeap-subst-exp*)

lemma *HSem-subst-expr-below*:
assumes *below*: $[\![\ e1\ ]\!]_{\{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho} \sqsubseteq [\![\ e2\ ]\!]_{\{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho}$
shows $\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho \sqsubseteq \{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho$
by (*rule HSem-below*) (*auto simp add*: *lookup-HSem-heap below lookup-HSem-other*)

lemma *HSem-subst-expr*:
assumes *below1*: $[\![\ e1\ ]\!]_{\{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho} \sqsubseteq [\![\ e2\ ]\!]_{\{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho}$
assumes *below2*: $[\![\ e2\ ]\!]_{\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho} \sqsubseteq [\![\ e1\ ]\!]_{\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho}$
shows $\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho = \{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho$

by (*metis assms HSem-subst-expr-below below-antisym*)

15.4 Re-calculating the semantics of the heap is idempotent

lemma *HSem-redo*:
  shows $\{\!\!\{\Gamma\}\!\!\}(\{\!\!\{\Gamma @ \Delta\}\!\!\}\varrho)\ f|`\ (edom\ \varrho \cup domA\ \Delta) = \{\!\!\{\Gamma @ \Delta\}\!\!\}\varrho$ (**is** *?LHS = ?RHS*)
proof (*rule below-antisym*)
  show $?LHS \sqsubseteq ?RHS$
  by (*rule HSem-below*)
    (*auto simp add*: *lookup-HSem-heap fun-belowD*[*OF env-restr-below-itself*])

  show $?RHS \sqsubseteq ?LHS$
  proof(*rule HSem-below*, *goal-cases*)
    case (*1 x*)
    thus *?case*
      by (*cases* $x \notin edom\ \varrho$) (*auto simp add*: *lookup-HSem-other dest*:*lookup-not-edom*)
  next
    case *prems*: (*2 x*)
    thus *?case*
    proof(*cases* $x \in domA\ \Gamma$)
    case *True*
      thus *?thesis* **by** (*auto simp add*: *lookup-HSem-heap*)
    next
    case *False*
      hence *delta*: $x \in domA\ \Delta$ **using** *prems* **by** *auto*
      with *False* ‹$?LHS \sqsubseteq ?RHS$›
    show *?thesis* **by** (*auto simp add*: *lookup-HSem-other lookup-HSem-heap monofun-cfun-arg*)
    qed
  qed
qed

15.5 Iterative definition of the heap semantics

lemma *iterative-HSem*:
  assumes $x \notin domA\ \Gamma$
  shows $\{\!\!\{(x,e)\ \#\ \Gamma\}\!\!\}\varrho = (\mu\ \varrho'.\ (\varrho\ {+\!+}_{domA\ \Gamma}\ (\{\!\!\{\Gamma\}\!\!\}\varrho'))(\ x := [\![e]\!]_{\varrho'}))$
proof−
  from *assms*
  interpret *iterative*
    where $e1 = \Lambda\ \varrho'.\ [\![\Gamma]\!]_{\varrho'}$
    and $e2 = \Lambda\ \varrho'.\ [\![e]\!]_{\varrho'}$
    and $S = domA\ \Gamma$
    and $x = x$ **by** *unfold-locales*

  have $\{\!\!\{(x,e)\ \#\ \Gamma\}\!\!\}\varrho = fix \cdot L$
    by (*simp add*: *HSem-def' override-on-upd ne*)
  also have $\ldots = fix \cdot R$
    by (*rule iterative-override-on*)
  also have $\ldots = (\mu\ \varrho'.\ (\varrho\ {+\!+}_{domA\ \Gamma}\ (\{\!\!\{\Gamma\}\!\!\}\varrho'))(\ x := [\![e]\!]_{\varrho'}))$

by (*simp add*: *HSem-def'*)
finally show *?thesis*.
qed

lemma *iterative-HSem'*:
assumes $x \notin domA\ \Gamma$
shows $(\mu\ \varrho'.\ (\varrho\ {++}_{domA\ \Gamma}\ (\{\!\!\{\Gamma\}\!\!\}\varrho'))(\ x := [\![e]\!]_{\varrho'}))$
$= (\mu\ \varrho'.\ (\varrho\ {++}_{domA\ \Gamma}\ (\{\!\!\{\Gamma\}\!\!\}\varrho'))(\ x := [\![e]\!]_{\{\!\!\{\Gamma\}\!\!\}\varrho'}))$
proof−
from *assms*
interpret *iterative*
where $e1 = \Lambda\ \varrho'.\ [\![\Gamma]\!]_{\varrho'}$
and $e2 = \Lambda\ \varrho'.\ [\![e]\!]_{\varrho'}$
and $S = domA\ \Gamma$
and $x = x$ **by** *unfold-locales*

have $(\mu\ \varrho'.\ (\varrho\ {++}_{domA\ \Gamma}\ (\{\!\!\{\Gamma\}\!\!\}\varrho'))(\ x := [\![e]\!]_{\varrho'})) = fix \cdot R$
by (*simp add*: *HSem-def'*)
also have $\ldots = fix \cdot L$
by (*rule iterative-override-on*[*symmetric*])
also have $\ldots = fix \cdot R'$
by (*rule iterative-override-on'*)
also have $\ldots = (\mu\ \varrho'.\ (\varrho\ {++}_{domA\ \Gamma}\ (\{\!\!\{\Gamma\}\!\!\}\varrho'))(\ x := [\![e]\!]_{\{\!\!\{\Gamma\}\!\!\}\varrho'}))$
by (*simp add*: *HSem-def'*)
finally
show *?thesis*.
qed

15.6 Fresh variables on the heap are irrelevant

lemma *HSem-ignores-fresh-restr'*:
assumes $fv\ \Gamma \subseteq S$
assumes $\bigwedge x\ \varrho.\ x \in domA\ \Gamma \Longrightarrow [\![\ the\ (map\text{-}of\ \Gamma\ x)\ ]\!]_{\varrho} = [\![\ the\ (map\text{-}of\ \Gamma\ x)\ ]\!]_{\varrho\ f|`\ (fv\ (the\ (map\text{-}of\ \Gamma\ x)))}$
shows $(\{\!\!\{\Gamma\}\!\!\}\varrho)\ f|`\ S = \{\!\!\{\Gamma\}\!\!\}\varrho\ f|`\ S$
proof(*induction rule*: *parallel-HSem-ind*[*case-names adm base step*])
case *adm* **thus** *?case* **by** *simp*
next
case *base*
show *?case* **by** *simp*
next
case (*step y z*)
have $[\![\ \Gamma\ ]\!]_y = [\![\ \Gamma\ ]\!]_z$
proof(*rule evalHeap-cong'*)
fix x
assume $x \in domA\ \Gamma$
hence $fv\ (the\ (map\text{-}of\ \Gamma\ x)) \subseteq fv\ \Gamma$ **by** (*rule map-of-fv-subset*)
with *assms*(*1*)
have $fv\ (the\ (map\text{-}of\ \Gamma\ x)) \cap S = fv\ (the\ (map\text{-}of\ \Gamma\ x))$ **by** *auto*

with *step*
have $y\ f|`\ fv\ (the\ (map\text{-}of\ \Gamma\ x)) = z\ f|`\ fv\ (the\ (map\text{-}of\ \Gamma\ x))$ **by** *auto*
with $\langle x \in domA\ \Gamma\rangle$
show $[\![\ the\ (map\text{-}of\ \Gamma\ x)\ ]\!]_y = [\![\ the\ (map\text{-}of\ \Gamma\ x)\ ]\!]_z$
by $(subst\ (1\ 2)\ assms(2)[OF\ \langle x \in domA\ \Gamma\rangle])\ simp$
qed
moreover
have $domA\ \Gamma \subseteq S$ **using** *domA-fv-subset assms(1)* **by** *auto*
ultimately
show *?case* **by** (*simp add*: *env-restr-add env-restr-evalHeap-noop*)
qed
end

15.7 Freshness

context *has-ignore-fresh-ESem* **begin**

lemma *ESem-fresh-cong*:
assumes $\varrho\ f|`\ (fv\ e) = \varrho'\ f|`\ (fv\ e)$
shows $[\![\ e\ ]\!]_\varrho = [\![\ e\ ]\!]_{\varrho'}$
by (*metis assms ESem-considers-fv*)

lemma *ESem-fresh-cong-subset*:
assumes $fv\ e \subseteq S$
assumes $\varrho\ f|`\ S = \varrho'\ f|`\ S$
shows $[\![\ e\ ]\!]_\varrho = [\![\ e\ ]\!]_{\varrho'}$
by (*rule ESem-fresh-cong*[*OF env-restr-eq-subset*[*OF assms*]])

lemma *ESem-fresh-cong-below*:
assumes $\varrho\ f|`\ (fv\ e) \sqsubseteq \varrho'\ f|`\ (fv\ e)$
shows $[\![\ e\ ]\!]_\varrho \sqsubseteq [\![\ e\ ]\!]_{\varrho'}$
by (*metis assms ESem-considers-fv monofun-cfun-arg*)

lemma *ESem-fresh-cong-below-subset*:
assumes $fv\ e \subseteq S$
assumes $\varrho\ f|`\ S \sqsubseteq \varrho'\ f|`\ S$
shows $[\![\ e\ ]\!]_\varrho \sqsubseteq [\![\ e\ ]\!]_{\varrho'}$
by (*rule ESem-fresh-cong-below*[*OF env-restr-below-subset*[*OF assms*]])

lemma *ESem-ignores-fresh-restr*:
assumes $atom\ `\ S \sharp{*}\ e$
shows $[\![\ e\ ]\!]_\varrho = [\![\ e\ ]\!]_{\varrho\ f|`\ (-\ S)}$
proof−
have $fv\ e \cap -\ S = fv\ e$ **using** *assms* **by** (*auto simp add*: *fresh-def fresh-star-def fv-supp*)
thus *?thesis* **by** (*subst* (*1 2*) *ESem-considers-fv*) *simp*
qed

lemma *ESem-ignores-fresh-restr'*:
assumes $atom\ `\ (edom\ \varrho - S) \sharp{*}\ e$

shows $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho\ f|`\ S}$
proof−
have $[\![\ e\ ]\!]_{\varrho} =\ [\![\ e\ ]\!]_{\varrho\ f|`\ (-\ (edom\ \varrho\ -\ S))}$
by (*rule ESem-ignores-fresh-restr*[*OF assms*])
also have $\varrho\ f|`\ (-\ (edom\ \varrho\ -\ S)) = \varrho\ f|`\ S$
by (*rule ext*) (*auto simp add*: *lookup-env-restr-eq dest*: *lookup-not-edom*)
finally show *?thesis*.
qed

lemma *HSem-ignores-fresh-restr''*:
assumes $fv\ \Gamma \subseteq S$
shows $(\{\!|\Gamma|\!\}\varrho)\ f|`\ S = \{\!|\Gamma|\!\}\varrho\ f|`\ S$
by (*rule HSem-ignores-fresh-restr'*[*OF assms*(*1*) *ESem-considers-fv*])

lemma *HSem-ignores-fresh-restr*:
assumes $atom\ `\ S\ \sharp{*}\ \Gamma$
shows $(\{\!|\Gamma|\!\}\varrho)\ f|`\ (-\ S) = \{\!|\Gamma|\!\}\varrho\ f|`\ (-\ S)$
proof−
from *assms* **have** $fv\ \Gamma \subseteq -\ S$ **by** (*auto simp add*: *fv-def fresh-star-def fresh-def*)
thus *?thesis* **by** (*rule HSem-ignores-fresh-restr''*)
qed

lemma *HSem-fresh-cong-below*:
assumes $\varrho\ f|`\ ((S \cup fv\ \Gamma)\ -\ domA\ \Gamma) \sqsubseteq \varrho'\ f|`\ ((S \cup fv\ \Gamma)\ -\ domA\ \Gamma)$
shows $(\{\!|\Gamma|\!\}\varrho)\ f|`\ S \sqsubseteq (\{\!|\Gamma|\!\}\varrho')\ f|`\ S$
proof−
from *assms*
have $\{\!|\Gamma|\!\}(\varrho\ f|`\ (S \cup fv\ \Gamma)) \sqsubseteq \{\!|\Gamma|\!\}(\varrho'\ f|`\ (S \cup fv\ \Gamma))$
by (*auto intro*: *HSem-restr-cong-below simp add*: *Diff-eq inf-commute*)
hence $(\{\!|\Gamma|\!\}\varrho)\ f|`\ (S \cup fv\ \Gamma) \sqsubseteq (\{\!|\Gamma|\!\}\varrho')\ f|`\ (S \cup fv\ \Gamma)$
by (*subst* (*1 2*) *HSem-ignores-fresh-restr''*) *simp-all*
thus *?thesis*
by (*rule env-restr-below-subset*[*OF Un-upper1*])
qed

lemma *HSem-fresh-cong*:
assumes $\varrho\ f|`\ ((S \cup fv\ \Gamma)\ -\ domA\ \Gamma) = \varrho'\ f|`\ ((S \cup fv\ \Gamma)\ -\ domA\ \Gamma)$
shows $(\{\!|\Gamma|\!\}\varrho)\ f|`\ S = (\{\!|\Gamma|\!\}\varrho')\ f|`\ S$
apply (*rule below-antisym*)
apply (*rule HSem-fresh-cong-below*[*OF eq-imp-below*[*OF assms*]])
apply (*rule HSem-fresh-cong-below*[*OF eq-imp-below*[*OF assms*[*symmetric*]]])
done

15.8 Adding a fresh variable to a heap does not affect its semantics

lemma *HSem-add-fresh'*:
assumes *fresh*: $atom\ x\ \sharp\ \Gamma$
assumes $x \notin edom\ \varrho$
assumes *step*: $\bigwedge e\ \varrho'.\ e \in snd\ `\ set\ \Gamma \Longrightarrow [\![\ e\ ]\!]_{\varrho'} = [\![\ e\ ]\!]_{env\text{-}delete\ x\ \varrho'}$

shows *env-delete* x ($\{\!\!\{(x,\ e)\ \#\ \Gamma\}\!\!\}\varrho) = \{\!\!\{\Gamma\}\!\!\}\varrho$
proof (*rule parallel-HSem-ind*, *goal-cases*)
case *1* **show** *?case* **by** *simp*
next
case *2* **show** *?case* **by** *auto*
next
case *prems*: (*3 y z*)
have *env-delete* $x\ \varrho = \varrho$ **using** ‹$x \notin$ *edom* ϱ› **by** (*rule env-delete-noop*)
moreover
from *fresh* **have** $x \notin$ *domA* Γ **by** (*metis domA-not-fresh*)
hence *env-delete* x ($[\![\ (x,\ e)\ \#\ \Gamma\ ]\!]_y) = [\![\ \Gamma\ ]\!]_y$
by (*auto intro*: *env-delete-noop dest*: *set-mp*[*OF edom-evalHeap-subset*])
moreover
have $\ldots = [\![\ \Gamma\ ]\!]_z$
apply (*rule evalHeap-cong*[*OF refl*])
apply (*subst* (*1*) *step*, *assumption*)
using *prems*(*1*) **apply** *auto*
done
ultimately
show *?case* **using** ‹$x \notin$ *domA* Γ›
by (*simp add*: *env-delete-add*)
qed

lemma *HSem-add-fresh*:
assumes *atom* $x \sharp \Gamma$
assumes $x \notin$ *edom* ϱ
shows *env-delete* x ($\{\!\!\{(x,\ e)\ \#\ \Gamma\}\!\!\}\varrho) = \{\!\!\{\Gamma\}\!\!\}\varrho$
proof(*rule HSem-add-fresh'*[*OF assms*], *goal-cases*)
case (*1 e* ϱ')
assume $e \in$ *snd* ‘ *set* Γ
hence *atom* $x \sharp e$ **by** (*metis assms*(*1*) *fresh-heap-expr'*)
hence $x \notin$ *fv e* **by** (*simp add*: *fv-def fresh-def*)
thus *?case*
by (*rule ESem-fresh-cong*[*OF env-restr-env-delete-other*[*symmetric*]])
qed

15.9 Mutual recursion with fresh variables

lemma *HSem-subset-below*:
assumes *fresh*: *atom* ‘ *domA* $\Gamma \sharp{*}\ \Delta$
shows $\{\!\!\{\Delta\}\!\!\}(\varrho\ f|$‘ $(-\ domA\ \Gamma)) \sqsubseteq (\{\!\!\{\Delta @ \Gamma\}\!\!\}\varrho)\ \ f|$‘ $(-\ domA\ \Gamma)$
proof(*rule HSem-below*)
fix x
assume [*simp*]: $x \in$ *domA* Δ
with *assms* **have** $*$: *atom* ‘ *domA* $\Gamma \sharp{*}$ *the* (*map-of* Δ x) **by** (*metis fresh-star-map-of*)
hence [*simp*]: $x \notin$ *domA* Γ **using** *fresh* ‹$x \in$ *domA* Δ› **by** (*metis fresh-star-def domA-not-fresh image-eqI*)
show $[\![$ *the* (*map-of* Δ x) $]\!]_{(\{\!\!\{\Delta\ @\ \Gamma\}\!\!\}\varrho)\ f|‘\ (-\ domA\ \Gamma)} \sqsubseteq ((\{\!\!\{\Delta\ @\ \Gamma\}\!\!\}\varrho)\ f|$‘ $(-\ domA\ \Gamma))\ x$
by (*simp add*: *lookup-HSem-heap ESem-ignores-fresh-restr*[*OF* $*$, *symmetric*])

qed (*simp add*: *lookup-HSem-other lookup-env-restr-eq*)

In the following lemma we show that the semantics of fresh variables can be be calculated together with the presently bound variables, or separately.

lemma *HSem-merge*:
  assumes *fresh*: *atom* ‘ *domA* Γ $\sharp*$ Δ
  shows $\{\!|\Gamma|\!\}\{\!|\Delta|\!\}\varrho = \{\!|\Gamma@\Delta|\!\}\varrho$
proof(*rule below-antisym*)
  have *map-of-eq*: *map-of* (Δ @ Γ) = *map-of* (Γ @ Δ)
  proof
    fix x
    show *map-of* (Δ @ Γ) x = *map-of* (Γ @ Δ) x
    proof (*cases* $x \in$ *domA* Γ)
      case *True*
      hence $x \notin$ *domA* Δ **by** (*metis fresh-distinct fresh IntI equals0D*)
      thus *map-of* (Δ @ Γ) x = *map-of* (Γ @ Δ) x
        by (*simp add*: *map-add-dom-app-simps dom-map-of-conv-domA*)
    next
      case *False*
      thus *map-of* (Δ @ Γ) x = *map-of* (Γ @ Δ) x
        by (*simp add*: *map-add-dom-app-simps dom-map-of-conv-domA*)
    qed
  qed

  show $\{\!|\Gamma|\!\}\{\!|\Delta|\!\}\varrho \sqsubseteq \{\!|\Gamma@\Delta|\!\}\varrho$
  proof(*rule HSem-below*)
    fix x
    assume [*simp*]:$x \notin$ *domA* Γ

    have ($\{\!|\Delta|\!\}\varrho$) x = (($\{\!|\Delta|\!\}\varrho$) $f|$‘ ($-$ *domA* Γ)) x **by** *simp*
    also have ... = ($\{\!|\Delta|\!\}$(ϱ $f|$‘ ($-$ *domA* Γ))) x
      by (*rule arg-cong*[*OF HSem-ignores-fresh-restr*[*OF fresh*]])
    also have ... $\sqsubseteq$ (($\{\!|\Delta@\Gamma|\!\}\varrho$) $f|$‘ ($-$ *domA* Γ)) x
      by (*rule fun-belowD*[*OF HSem-subset-below*[*OF fresh*]])
    also have ... = ($\{\!|\Delta@\Gamma|\!\}\varrho$) x **by** *simp*
    also have ... = ($\{\!|\Gamma$ @ $\Delta|\!\}\varrho$) x **by** (*rule arg-cong*[*OF HSem-reorder*[*OF map-of-eq*]])
    finally
    show ($\{\!|\Delta|\!\}\varrho$) x $\sqsubseteq$ ($\{\!|\Gamma$ @ $\Delta|\!\}\varrho$) x**.**
  qed (*auto simp add*: *lookup-HSem-heap lookup-env-restr-eq*)

  have $*$: $\bigwedge$ x. $x \in$ *domA* $\Delta \Longrightarrow x \notin$ *domA* Γ
    using *fresh* **by** (*auto simp add*: *fresh-Pair fresh-star-def domA-not-fresh*)

  have *foo*: *edom* ϱ $\cup$ *domA* Δ $\cup$ *domA* Γ $-$ (*edom* ϱ $\cup$ *domA* Δ $\cup$ *domA* Γ) $\cap$ $-$ *domA* Γ = *domA* Γ **by** *auto*
  have *foo2*:(*edom* ϱ $\cup$ *domA* Δ $-$ (*edom* ϱ $\cup$ *domA* Δ) $\cap$ $-$ *domA* Γ) $\subseteq$ *domA* Γ **by** *auto*

  { fix x
    assume $x \in$ *domA* Δ

hence $*$: *atom ‘ domA* Γ $\sharp*$ *the* (*map-of* Δ *x*)
by (*rule fresh-star-map-of*[*OF - fresh*])

have $[\![$ *the* (*map-of* Δ *x*) $]\!]_{\{\!\mid\Gamma\mid\!\}\{\!\mid\Delta\mid\!\}\varrho}$ = $[\![$ *the* (*map-of* Δ *x*) $]\!]_{(\{\!\mid\Gamma\mid\!\}\{\!\mid\Delta\mid\!\}\varrho)\ f|`\ (-\ domA\ \Gamma)}$
by (*rule ESem-ignores-fresh-restr*[*OF* $*$])
also have $(\{\!\mid\Gamma\mid\!\}\{\!\mid\Delta\mid\!\}\varrho)\ f|`\ (-\ domA\ \Gamma) = (\{\!\mid\Delta\mid\!\}\varrho)\ f|`\ (-\ domA\ \Gamma)$
by (*simp add*: *env-restr-HSem*)
also have $[\![$ *the* (*map-of* Δ *x*) $]\!]_{\ldots}$ = $[\![$ *the* (*map-of* Δ *x*) $]\!]_{\{\!\mid\Delta\mid\!\}\varrho}$
by (*rule ESem-ignores-fresh-restr*[*symmetric*, *OF* $*$])
finally
have $[\![$ *the* (*map-of* Δ *x*) $]\!]_{\{\!\mid\Gamma\mid\!\}\{\!\mid\Delta\mid\!\}\varrho}$ = $[\![$ *the* (*map-of* Δ *x*) $]\!]_{\{\!\mid\Delta\mid\!\}\varrho}$.
}
thus $\{\!\mid\Gamma @ \Delta\mid\!\}\varrho \sqsubseteq \{\!\mid\Gamma\mid\!\}\{\!\mid\Delta\mid\!\}\varrho$
by $-$(*rule HSem-below*, *auto simp add*: *lookup-HSem-other lookup-HSem-heap* $*$)
qed
end

15.10 Parallel induction

lemma *parallel-HSem-ind-different-ESem*:
assumes *adm* ($\lambda\varrho'$. *P* (*fst* ϱ') (*snd* ϱ'))
assumes *P* $\bot$ $\bot$
assumes $\bigwedge$*y z*. *P y z* $\Longrightarrow$ *P* (ϱ $++_{domA\ h}$ *evalHeap h* (λ*e*. *ESem1 e* $\cdot$ *y*)) ($\varrho 2$ $++_{domA\ h2}$ *evalHeap h2* (λ*e*. *ESem2 e* $\cdot$ *z*))
shows *P* (*has-ESem.HSem ESem1 h*$\cdot\varrho$) (*has-ESem.HSem ESem2 h2*$\cdot\varrho 2$)
proof$-$
interpret *HSem1*: *has-ESem ESem1*.
interpret *HSem2*: *has-ESem ESem2*.

show *?thesis*
unfolding *HSem1.HSem-def' HSem2.HSem-def'*
apply (*rule parallel-fix-ind*[*OF assms*(*1*)])
apply (*rule assms*(*2*))
apply *simp*
apply (*erule assms*(*3*))
done
qed

15.11 Congruence rule

lemma *HSem-cong*[*fundef-cong*]:
$[\![$ ($\bigwedge$ *e*. *e* $\in$ *snd ‘ set heap2* $\Longrightarrow$ *ESem1 e* = *ESem2 e*); *heap1* = *heap2* $]\!]$
$\Longrightarrow$ *has-ESem.HSem ESem1 heap1* = *has-ESem.HSem ESem2 heap2*
unfolding *has-ESem.HSem-def*
by (*auto cong*:*evalHeap-cong*)

15.12 Equivariance of the heap semantics

lemma *HSem-eqvt*[*eqvt*]:

$\pi \cdot$ *has-ESem.HSem ESem* $\Gamma =$ *has-ESem.HSem* $(\pi \cdot$ *ESem*$)$ $(\pi \cdot \Gamma)$
proof−
show *?thesis*
unfolding *has-ESem.HSem-def*
apply (*subst permute-Lam*, *simp*)
apply (*subst eqvt-lambda*)
apply (*simp add*: *Cfun-app-eqvt permute-Lam*)
done
qed

end

16 Vars.tex

theory *Vars*
imports *../Nominal2/Nominal2*
begin

The type of variables is abstract and provided by the Nominal package. All we know is that it is countable.

atom-decl *var*

end

17 Terms.tex

theory *Terms*
imports *Nominal−Utils Vars AList−Utils−Nominal*
begin

17.1 Expressions

This is the main data type of the development; our minimal lambda calculus with recursive let-bindings. It is created using the nominal_datatype command, which creates alpha-equivalence classes.

The package does not support nested recursion, so the bindings of the let cannot simply be of type (*var*, *exp*) *list*. Instead, the definition of lists have to be inlined here, as the custom type *assn*. Later we create conversion functions between these two types, define a properly typed *let* and redo the various lemmas in terms of that, so that afterwards, the type *assn* is no longer referenced.

nominal-datatype *exp* =
Var var
| *App exp var*

| *LetA as::assn body::exp* **binds** *bn as* **in** *body as*
| *Lam x::var body::exp* **binds** *x* **in** *body* (*Lam* [-]. - [*100*, *100*] *100*)
| *Bool bool*
| *IfThenElse exp exp exp* (((-)/ ? (-)/ : (-)) [*0*, *0*, *10*] *10*)
and *assn* =
  ANil | *ACons var exp assn*
binder
  bn :: *assn* ⇒ *atom list*
where *bn ANil* = [] | *bn* (*ACons x t as*) = (*atom x*) # (*bn as*)

notation (*latex* **output**) *Terms.Var* (-)
notation (*latex* **output**) *Terms.App* (- -)
notation (*latex* **output**) *Terms.Lam* (λ-. - [*100*, *100*] *100*)

type-synonym *heap* = (*var* × *exp*) *list*

lemma *exp-assn-size-eqvt*[*eqvt*]: *p* • (*size* :: *exp* ⇒ *nat*) = *size*
  by (*metis exp-assn.size-eqvt*(*1*) *fun-eqvtI permute-pure*)

17.2 Rewriting in terms of heaps

We now work towards using *heap* instead of *assn*. All this could be skipped if Nominal supported nested recursion.

Conversion from *assn* to *heap*.

nominal-function *asToHeap* :: *assn* ⇒ *heap*
 where *ANilToHeap*: *asToHeap ANil* = []
 | *AConsToHeap*: *asToHeap* (*ACons v e as*) = (*v*, *e*) # *asToHeap as*
unfolding *eqvt-def asToHeap-graph-aux-def*
apply *rule*
apply *simp*
apply *rule*
apply(*case-tac x rule*: *exp-assn.exhaust*(*2*))
apply *auto*
done
nominal-termination(*eqvt*) **by** *lexicographic-order*

lemma *asToHeap-eqvt*: *eqvt asToHeap*
  unfolding *eqvt-def*
  by (*auto simp add*: *permute-fun-def asToHeap.eqvt*)

The other direction.

fun *heapToAssn* :: *heap* ⇒ *assn*
  where *heapToAssn* [] = *ANil*
  | *heapToAssn* ((*v*,*e*)#Γ) = *ACons v e* (*heapToAssn* Γ)

declare *heapToAssn.simps*[*simp del*]

lemma *heapToAssn-eqvt*[*simp*,*eqvt*]: $p \cdot heapToAssn\ \Gamma = \ heapToAssn\ (p \cdot \Gamma)$
by (*induct* Γ *rule*: *heapToAssn.induct*)
(*auto simp add*: *heapToAssn.simps*)

lemma *bn-heapToAssn*: $bn\ (heapToAssn\ \Gamma) = map\ (\lambda x.\ atom\ (fst\ x))\ \Gamma$
by (*induct rule*: *heapToAssn.induct*)
(*auto simp add*: *heapToAssn.simps exp-assn.bn-defs*)

lemma *set-bn-to-atom-domA*:
set (*bn as*) = *atom* ' *domA* (*asToHeap as*)
by (*induct as rule*: *asToHeap.induct*)
(*auto simp add*: *exp-assn.bn-defs*)

They are inverse to each other.

lemma *heapToAssn-asToHeap*[*simp*]:
heapToAssn (*asToHeap as*) = *as*
by (*induct rule*: *exp-assn.inducts*(*2*)[*of* λ - . *True*])
(*auto simp add*: *heapToAssn.simps*)

lemma *asToHeap-heapToAssn*[*simp*]:
asToHeap (*heapToAssn as*) = *as*
by (*induct rule*: *heapToAssn.induct*)
(*auto simp add*: *heapToAssn.simps*)

lemma *heapToAssn-inject*[*simp*]:
$heapToAssn\ x = heapToAssn\ y \longleftrightarrow x = y$
by (*metis asToHeap-heapToAssn*)

They are transparent to various notions from the Nominal package.

lemma *supp-heapToAssn*: *supp* (*heapToAssn* Γ) = *supp* Γ
by (*induct rule*: *heapToAssn.induct*)
(*simp-all add*: *heapToAssn.simps exp-assn.supp supp-Nil supp-Cons supp-Pair sup-assoc*)

lemma *supp-asToHeap*: *supp* (*asToHeap as*) = *supp as*
by (*induct as rule*: *asToHeap.induct*)
(*simp-all add*: *exp-assn.supp supp-Nil supp-Cons supp-Pair sup-assoc*)

lemma *fv-asToHeap*: *fv* (*asToHeap* Γ) = *fv* Γ
unfolding *fv-def* **by** (*auto simp add*: *supp-asToHeap*)

lemma *fv-heapToAssn*: *fv* (*heapToAssn* Γ) = *fv* Γ
unfolding *fv-def* **by** (*auto simp add*: *supp-heapToAssn*)

lemma [*simp*]: *size* (*heapToAssn* Γ) = *size-list* (λ (*v*,*e*) . *size e*) Γ
by (*induct rule*: *heapToAssn.induct*)
(*simp-all add*: *heapToAssn.simps*)

lemma *Lam-eq-same-var*[*simp*]: $Lam\ [y].\ e = Lam\ [y].\ e' \longleftrightarrow \ e = e'$

by *auto* (*metis fresh-PairD*(*2*) *obtain-fresh*)

Now we define the Let constructor in the form that we actually want.

hide-const *HOL.Let*
definition *Let* :: *heap* $\Rightarrow$ *exp* $\Rightarrow$ *exp*
 where *Let* Γ *e* = *LetA* (*heapToAssn* Γ) *e*

notation (*latex* **output**) *Let* (*let* - *in* -)

abbreviation
 LetBe :: *var*$\Rightarrow$*exp*$\Rightarrow$*exp*$\Rightarrow$*exp* (*let* - *be* - *in* - [*100*,*100*,*100*] *100*)
where
 let x be t1 in t2 $\equiv$ *Let* [(*x*,*t1*)] *t2*

We rewrite all (relevant) lemmas about *LetA* in terms of *Let*.

lemma *size-Let*[*simp*]: *size* (*Let* Γ *e*) = *size-list* (λ*p*. *size* (*snd p*)) Γ + *size e* + *Suc 0*
 unfolding *Let-def* **by** (*auto simp add*: *split-beta'*)

lemma *Let-distinct*[*simp*]:
 Var v $\neq$ *Let* Γ *e*
 Let Γ *e* $\neq$ *Var v*
 App e v $\neq$ *Let* Γ *e'*
 Lam [*v*]. *e'* $\neq$ *Let* Γ *e*
 Let Γ *e* $\neq$ *Lam* [*v*]. *e'*
 Let Γ *e'* $\neq$ *App e v*
 Bool b $\neq$ *Let* Γ *e*
 Let Γ *e* $\neq$ *Bool b*
 (*scrut ? e1* : *e2*) $\neq$ *Let* Γ *e*
 Let Γ *e* $\neq$ (*scrut ? e1* : *e2*)
 unfolding *Let-def* **by** *simp-all*

lemma *Let-perm-simps*[*simp*,*eqvt*]:
 p $\cdot$ *Let* Γ *e* = *Let* (*p* $\cdot$ Γ) (*p* $\cdot$ *e*)
 unfolding *Let-def* **by** *simp*

lemma *Let-supp*:
 supp (*Let* Γ *e*) = (*supp e* $\cup$ *supp* Γ) − *atom* ‘ (*domA* Γ)
 unfolding *Let-def* **by** (*simp add*: *exp-assn.supp supp-heapToAssn bn-heapToAssn domA-def image-image*)

lemma *Let-fresh*[*simp*]:
 a $\sharp$ *Let* Γ *e* = (*a* $\sharp$ *e* $\wedge$ *a* $\sharp$ Γ $\vee$ *a* $\in$ *atom* ‘ *domA* Γ)
 unfolding *fresh-def* **by** (*auto simp add*: *Let-supp*)

lemma *Abs-eq-cong*:
 assumes $\bigwedge$ *p*. (*p* $\cdot$ *x* = *x'*) $\longleftrightarrow$ (*p* $\cdot$ *y* = *y'*)
 assumes *supp y* = *supp x*
 assumes *supp y'* = *supp x'*

shows ([a]lst. x = [a']lst. x') $\longleftrightarrow$ ([a]lst. y = [a']lst. y')
by (*simp add*: *Abs-eq-iff alpha-lst assms*)

lemma *Let-eq-iff*[*simp*]:
(*Let* Γ *e* = *Let* Γ' *e'*) = ([*map* (λx. *atom* (*fst x*)) Γ]*lst*. (*e*, Γ) = [*map* (λx. *atom* (*fst x*)) Γ']*lst*. (*e'*,Γ'))
unfolding *Let-def*
apply (*simp add*: *bn-heapToAssn*)
apply (*rule Abs-eq-cong*)
apply (*simp-all add*: *supp-Pair supp-heapToAssn*)
done

lemma *exp-strong-exhaust*:
fixes *c* :: *'a* :: *fs*
assumes $\bigwedge$*var*. *y* = *Var var* $\Longrightarrow$ *P*
assumes $\bigwedge$*exp var*. *y* = *App exp var* $\Longrightarrow$ *P*
assumes $\bigwedge\Gamma$ *exp*. *atom* ‘ *domA* Γ $\sharp*$ *c* $\Longrightarrow$ *y* = *Let* Γ *exp* $\Longrightarrow$ *P*
assumes $\bigwedge$*var exp*. {*atom var*} $\sharp*$ *c* $\Longrightarrow$ *y* = *Lam* [*var*]. *exp* $\Longrightarrow$ *P*
assumes $\bigwedge$ *b*. (*y* = *Bool b*) $\Longrightarrow$ *P*
assumes $\bigwedge$ *scrut e1 e2*. *y* = (*scrut ? e1* : *e2*) $\Longrightarrow$ *P*
shows *P*
apply (*rule exp-assn.strong-exhaust*(*1*)[**where** *y* = *y* **and** *c* = *c*])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*) *set-bn-to-atom-domA Let-def heapToAssn-asToHeap*)
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
done

And finally the induction rules with *Let*.

lemma *exp-heap-induct*[*case-names Var App Let Lam Bool IfThenElse Nil Cons*]:
assumes $\bigwedge$*b var*. *P1* (*Var var*)
assumes $\bigwedge$*exp var*. *P1 exp* $\Longrightarrow$ *P1* (*App exp var*)
assumes $\bigwedge\Gamma$ *exp*. *P2* Γ $\Longrightarrow$ *P1 exp* $\Longrightarrow$ *P1* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp*. *P1 exp* $\Longrightarrow$ *P1* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$ *b*. *P1* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2*. *P1 scrut* $\Longrightarrow$ *P1 e1* $\Longrightarrow$ *P1 e2* $\Longrightarrow$ *P1* (*scrut ? e1* : *e2*)
assumes *P2* []
assumes $\bigwedge$*var exp* Γ. *P1 exp* $\Longrightarrow$ *P2* Γ $\Longrightarrow$ *P2* ((*var*, *exp*)#Γ)
shows *P1 e* **and** *P2* Γ
proof−
have *P1 e* **and** *P2* (*asToHeap* (*heapToAssn* Γ))
apply (*induct rule*: *exp-assn.inducts*[*of P1* λ *assn*. *P2* (*asToHeap assn*)])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*) *Let-def heapToAssn-asToHeap*)
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))

apply (*metis assms(6)*)
apply (*metis assms(7) asToHeap.simps(1)*)
apply (*metis assms(8) asToHeap.simps(2)*)
done
thus *P1 e* **and** *P2* Γ **unfolding** *asToHeap-heapToAssn*.
qed

lemma *exp-heap-strong-induct*[*case-names Var App Let Lam Bool IfThenElse Nil Cons*]:
assumes $\bigwedge$*var c. P1 c (Var var)*
assumes $\bigwedge$*exp var c.* ($\bigwedge$*c. P1 c exp*) $\Longrightarrow$ *P1 c (App exp var)*
assumes $\bigwedge\Gamma$ *exp c. atom ‘ domA* Γ $\sharp*$ *c* $\Longrightarrow$ ($\bigwedge$*c. P2 c* Γ) $\Longrightarrow$ ($\bigwedge$*c. P1 c exp*) $\Longrightarrow$ *P1 c (Let* Γ *exp)*
assumes $\bigwedge$*var exp c.* {*atom var*} $\sharp*$ *c* $\Longrightarrow$ ($\bigwedge$*c. P1 c exp*) $\Longrightarrow$ *P1 c (Lam [var]. exp)*
assumes $\bigwedge$ *b c. P1 c (Bool b)*
assumes $\bigwedge$ *scrut e1 e2 c.* ($\bigwedge$ *c. P1 c scrut*) $\Longrightarrow$ ($\bigwedge$ *c. P1 c e1*) $\Longrightarrow$ ($\bigwedge$ *c. P1 c e2*) $\Longrightarrow$ *P1 c (scrut ? e1 : e2)*
assumes $\bigwedge$*c. P2 c []*
assumes $\bigwedge$*var exp* Γ *c.* ($\bigwedge$*c. P1 c exp*) $\Longrightarrow$ ($\bigwedge$*c. P2 c* Γ) $\Longrightarrow$ *P2 c ((var,exp)#*Γ*)*
fixes *c :: 'a :: fs*
shows *P1 c e* **and** *P2 c* Γ
proof−
have *P1 c e* **and** *P2 c (asToHeap (heapToAssn* Γ*))*
apply (*induct rule: exp-assn.strong-induct[of P1* λ *c assn. P2 c (asToHeap assn)]*)
apply (*metis assms(1)*)
apply (*metis assms(2)*)
apply (*metis assms(3) set-bn-to-atom-domA Let-def heapToAssn-asToHeap*)
apply (*metis assms(4)*)
apply (*metis assms(5)*)
apply (*metis assms(6)*)
apply (*metis assms(7) asToHeap.simps(1)*)
apply (*metis assms(8) asToHeap.simps(2)*)
done
thus *P1 c e* **and** *P2 c* Γ **unfolding** *asToHeap-heapToAssn*.
qed

17.3 Nice induction rules

These rules can be used instead of the original induction rules, which require a separate goal for *assn*.

lemma *exp-induct*[*case-names Var App Let Lam Bool IfThenElse*]:
assumes $\bigwedge$*var. P (Var var)*
assumes $\bigwedge$*exp var. P exp* $\Longrightarrow$ *P (App exp var)*
assumes $\bigwedge\Gamma$ *exp.* ($\bigwedge$ *x. x* $\in$ *domA* Γ $\Longrightarrow$ *P (the (map-of* Γ *x)))* $\Longrightarrow$ *P exp* $\Longrightarrow$ *P (Let* Γ *exp)*
assumes $\bigwedge$*var exp. P exp* $\Longrightarrow$ *P (Lam [var]. exp)*
assumes $\bigwedge$ *b. P (Bool b)*
assumes $\bigwedge$ *scrut e1 e2. P scrut* $\Longrightarrow$ *P e1* $\Longrightarrow$ *P e2* $\Longrightarrow$ *P (scrut ? e1 : e2)*
shows *P exp*

apply (*rule exp-heap-induct*[*of P* λ Γ. ($\forall$ *x* $\in$ *domA* Γ. *P* (*the* (*map-of* Γ *x*)))])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*))
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply *auto*
done

lemma *exp-strong-induct-set*[*case-names Var App Let Lam Bool IfThenElse*]:
assumes $\bigwedge$*var c. P c* (*Var var*)
assumes $\bigwedge$*exp var c.* ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*App exp var*)
assumes $\bigwedge$$\Gamma$ *exp c.*
atom ‘ *domA* Γ $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c x e.* (*x*,*e*) $\in$ *set* Γ $\Longrightarrow$ *P c e*) $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp c.* {*atom var*} $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b c. P c* (*Bool b*)
assumes $\bigwedge$*scrut e1 e2 c.* ($\bigwedge$ *c. P c scrut*) $\Longrightarrow$ ($\bigwedge$ *c. P c e1*) $\Longrightarrow$ ($\bigwedge$ *c. P c e2*) $\Longrightarrow$ *P c* (*scrut ? e1 : e2*)
shows *P* (*c*::′*a*::*fs*) *exp*
apply (*rule exp-heap-strong-induct*(*1*)[*of P* λ *c* Γ. ($\forall$ (*x*,*e*) $\in$ *set* Γ. *P c e*)])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*) *split-conv*)
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply *auto*
done

lemma *exp-strong-induct*[*case-names Var App Let Lam Bool IfThenElse*]:
assumes $\bigwedge$*var c. P c* (*Var var*)
assumes $\bigwedge$*exp var c.* ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*App exp var*)
assumes $\bigwedge$$\Gamma$ *exp c.*
atom ‘ *domA* Γ $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c x. x* $\in$ *domA* Γ $\Longrightarrow$ *P c* (*the* (*map-of* Γ *x*))) $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp c.* {*atom var*} $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b c. P c* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2 c.* ($\bigwedge$ *c. P c scrut*) $\Longrightarrow$ ($\bigwedge$ *c. P c e1*) $\Longrightarrow$ ($\bigwedge$ *c. P c e2*) $\Longrightarrow$ *P c* (*scrut ? e1 : e2*)
shows *P* (*c*::′*a*::*fs*) *exp*
apply (*rule exp-heap-strong-induct*(*1*)[*of P* λ *c* Γ. ($\forall$ *x* $\in$ *domA* Γ. *P c* (*the* (*map-of* Γ *x*)))])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*))
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))

apply (*metis assms*(*6*))
apply *auto*
done

17.4 Testing alpha equivalence

lemma *alpha-test*:
shows *Lam* [*x*]. (*Var x*) = *Lam* [*y*]. (*Var y*)
by (*simp add*: *Abs1-eq-iff fresh-at-base pure-fresh*)

lemma *alpha-test2*:
shows *let x be* (*Var x*) *in* (*Var x*) = *let y be* (*Var y*) *in* (*Var y*)
by (*simp add*: *fresh-Cons fresh-Nil Abs1-eq-iff fresh-Pair add*: *fresh-at-base pure-fresh*)

lemma *alpha-test3*:
shows
Let [(*x*, *Var y*), (*y*, *Var x*)] (*Var x*)
=
Let [(*y*, *Var x*), (*x*, *Var y*)] (*Var y*) (**is** *Let ?la ?lb* = -)
by (*simp add*: *bn-heapToAssn Abs1-eq-iff fresh-Pair fresh-at-base*
Abs-swap2[*of atom x* (*?lb*, [(*x*, *Var y*), (*y*, *Var x*)]) [*atom x*, *atom y*] *atom y*])

17.5 Free variables

lemma *fv-supp-exp*: *supp e* = *atom* ‘ (*fv* (*e*::*exp*) :: *var set*) **and** *fv-supp-as*: *supp as* = *atom* ‘ (*fv* (*as*::*assn*) :: *var set*)
by (*induction e* **and** *as rule*:*exp-assn.inducts*)
(*auto simp add*: *fv-def exp-assn.supp supp-at-base pure-supp*)

lemma *fv-supp-heap*: *supp* (Γ::*heap*) = *atom* ‘ (*fv* Γ :: *var set*)
by (*metis fv-def fv-supp-as supp-heapToAssn*)

lemma *fv-Lam*[*simp*]: *fv* (*Lam* [*x*]. *e*) = *fv e* − {*x*}
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp*)
lemma *fv-Var*[*simp*]: *fv* (*Var x*) = {*x*}
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp supp-at-base pure-supp*)
lemma *fv-App*[*simp*]: *fv* (*App e x*) = *insert x* (*fv e*)
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp supp-at-base*)
lemma *fv-Let*[*simp*]: *fv* (*Let* Γ *e*) = (*fv* Γ ∪ *fv e*) − *domA* Γ
unfolding *fv-def* **by** (*auto simp add*: *Let-supp exp-assn.supp supp-at-base set-bn-to-atom-domA*)
lemma *fv-Bool*[*simp*]: *fv* (*Bool b*) = {}
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp pure-supp*)
lemma *fv-IfThenElse*[*simp*]: *fv* (*scrut ? e1 : e2*) = *fv scrut* ∪ *fv e1* ∪ *fv e2*
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp*)

lemma *fv-delete-heap*:
assumes *map-of* Γ *x* = *Some e*
shows *fv* (*delete x* Γ, *e*) ∪ {*x*} ⊆ (*fv* (Γ, *Var x*) :: *var set*)
proof−

have *fv* (*delete* x Γ) $\subseteq$ *fv* Γ **by** (*metis fv-delete-subset*)
moreover
have (x,e) $\in$ *set* Γ **by** (*metis assms map-of-SomeD*)
hence *fv* $e \subseteq$ *fv* Γ **by** (*metis assms domA-from-set map-of-fv-subset option.sel*)
moreover
have $x \in$ *fv* (*Var* x) **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

17.6 Lemmas helping with nominal definitions

lemma *eqvt-lam-case*:
assumes *Lam* [x]. e = *Lam* [x']. e'
assumes $\bigwedge \pi$. *supp* $(-\pi)$ $\sharp*$ (*fv* (*Lam* [x]. e) :: *var set*) $\Longrightarrow$
supp π $\sharp*$ (*Lam* [x]. e) $\Longrightarrow$
F ($\pi \cdot e$) ($\pi \cdot x$) (*Lam* [x]. e) = F e x (*Lam* [x]. e)
shows F e x (*Lam* [x]. e) = F e' x' (*Lam* [x']. e')
proof−

from *assms*(*1*)
have [[*atom* x]]*lst*. (e, x) = [[*atom* x']]*lst*. (e', x') **by** *auto*
then obtain p
where (*supp* (e, x) − {*atom* x}) $\sharp*$ p
and [*simp*]: $p \cdot x = x'$
and [*simp*]: $p \cdot e = e'$
unfolding *Abs-eq-iff*(*3*) *alpha-lst.simps* **by** *auto*

from ‹- $\sharp*$ p›
have $*$: *supp* $(-p)$ $\sharp*$ (*fv* (*Lam* [x]. e) :: *var set*)
by (*auto simp add*: *fresh-star-def fresh-def supp-finite-set-at-base supp-Pair fv-supp-exp fv-supp-heap supp-minus-perm*)

from ‹- $\sharp*$ p›
have $**$: *supp* p $\sharp*$ *Lam* [x]. e
by (*auto simp add*: *fresh-star-def fresh-def supp-Pair fv-supp-exp*)

have F e x (*Lam* [x]. e) = F ($p \cdot e$) ($p \cdot x$) (*Lam* [x]. e) **by** (*rule assms*(*2*)[*OF* $*$ $**$, *symmetric*])
also have ... = F e' x' (*Lam* [x']. e') **by** (*simp add*: *assms*(*1*))
finally show *?thesis*.
qed

lemma *eqvt-let-case*:
assumes *Let as body* = *Let as' body'*
assumes $\bigwedge \pi$.
supp $(-\pi)$ $\sharp*$ (*fv* (*Let as body*) :: *var set*) $\Longrightarrow$
supp π $\sharp*$ *Let as body* $\Longrightarrow$

F ($\pi \cdot as$) ($\pi \cdot body$) ($Let\ as\ body$) = $F\ as\ body$ ($Let\ as\ body$)
shows $F\ as\ body$ ($Let\ as\ body$) = $F\ as'\ body'$ ($Let\ as'\ body'$)
proof−
from *assms*(*1*)
have [*map* (λ *p*. *atom* (*fst p*)) *as*]*lst*. (*body*, *as*) = [*map* (λ *p*. *atom* (*fst p*)) *as'*]*lst*. (*body'*, *as'*) **by** *auto*
then obtain *p*
where (*supp* (*body*, *as*) − *atom* ' *domA as*) ♯* *p*
and [*simp*]: *p* · *body* = *body'*
and [*simp*]: *p* · *as* = *as'*
unfolding *Abs-eq-iff*(*3*) *alpha-lst.simps* **by** (*auto simp add*: *domA-def image-image*)

from ‹- ♯* *p*›
have *: *supp* (−*p*) ♯* (*fv* (*Terms.Let as body*) :: *var set*)
by (*auto simp add*: *fresh-star-def fresh-def supp-finite-set-at-base supp-Pair fv-supp-exp fv-supp-heap supp-minus-perm*)

from ‹- ♯* *p*›
have **: *supp p* ♯* *Terms.Let as body*
by (*auto simp add*: *fresh-star-def fresh-def supp-Pair fv-supp-exp fv-supp-heap*)

have *F as body* (*Let as body*) = *F* (*p* · *as*) (*p* · *body*) (*Let as body*) **by** (*rule assms*(*2*)[*OF* * **, *symmetric*])
also have ... = *F as' body'* (*Let as' body'*) **by** (*simp add*: *assms*(*1*))
finally show *?thesis*.
qed

17.7 A smart constructor for lets

Certian program transformations might change the bound variables, possibly making it an empty list. This smart constructor avoids the empty let in the resulting expression. Semantically, it should not make a difference.

definition *SmartLet* :: *heap* => *exp* => *exp*
where *SmartLet* Γ *e* = (*if* Γ = [] *then e else Let* Γ *e*)

lemma *SmartLet-eqvt*[*eqvt*]: π · (*SmartLet* Γ *e*) = *SmartLet* (π · Γ) (π · *e*)
unfolding *SmartLet-def* **by** *perm-simp rule*

lemma *SmartLet-supp*:
supp (*SmartLet* Γ *e*) = (*supp e* ∪ *supp* Γ) − *atom* ' (*domA* Γ)
unfolding *SmartLet-def* **using** *Let-supp* **by** (*auto simp add*: *supp-Nil*)

lemma *fv-SmartLet*[*simp*]: *fv* (*SmartLet* Γ *e*) = (*fv* Γ ∪ *fv e*) − *domA* Γ
unfolding *SmartLet-def* **by** *auto*

17.8 A predicate for value expressions

nominal-function *isLam* :: *exp* ⇒ *bool* **where**

isLam (*Var x*) = *False* |
isLam (*Lam* [*x*]. *e*) = *True* |
isLam (*App e x*) = *False* |
isLam (*Let as e*) = *False* |
isLam (*Bool b*) = *False* |
isLam (*scrut ? e1* : *e2*) = *False*
unfolding *isLam-graph-aux-def eqvt-def*
apply *simp*
apply *simp*
apply (*metis exp-strong-exhaust*)
apply *auto*
done
nominal-termination (*eqvt*) **by** *lexicographic-order*

lemma *isLam-Lam*: *isLam* (*Lam* [*x*]. *e*) **by** *simp*

lemma *isLam-obtain-fresh*:
assumes *isLam z*
obtains *y e′*
where *z* = (*Lam* [*y*]. *e′*) **and** *atom y* ♯ (*c*::′*a*::*fs*)
using *assms* **by** (*nominal-induct z avoiding*: *c rule*:*exp-strong-induct*) *auto*

nominal-function *isVal* :: *exp* ⇒ *bool* **where**
isVal (*Var x*) = *False* |
isVal (*Lam* [*x*]. *e*) = *True* |
isVal (*App e x*) = *False* |
isVal (*Let as e*) = *False* |
isVal (*Bool b*) = *True* |
isVal (*scrut ? e1* : *e2*) = *False*
unfolding *isVal-graph-aux-def eqvt-def*
apply *simp*
apply *simp*
apply (*metis exp-strong-exhaust*)
apply *auto*
done
nominal-termination (*eqvt*) **by** *lexicographic-order*

lemma *isVal-Lam*: *isVal* (*Lam* [*x*]. *e*) **by** *simp*
lemma *isVal-Bool*: *isVal* (*Bool b*) **by** *simp*

17.9 The notion of thunks

definition *thunks* :: *heap* ⇒ *var set* **where**
thunks Γ = {*x* . *case map-of* Γ *x of Some e* ⇒ ¬ *isVal e* | *None* ⇒ *False*}

lemma *thunks-Nil*[*simp*]: *thunks* [] = {} **by** (*auto simp add*: *thunks-def*)

lemma *thunks-domA*: *thunks* Γ ⊆ *domA* Γ
by (*induction* Γ) (*auto simp add*: *thunks-def*)

lemma *thunks-Cons*: *thunks* $((x,e)\#\Gamma) = ($*if isVal e then thunks* $\Gamma - \{x\}$ *else insert x* (*thunks* Γ))
  by (*auto simp add*: *thunks-def*)

lemma *thunks-append*[*simp*]: *thunks* $(\Delta @\Gamma)$ = *thunks* $\Delta \cup$ (*thunks* Γ − *domA* Δ)
  by (*induction* Δ) (*auto simp add*: *thunks-def*)

lemma *thunks-delete*[*simp*]: *thunks* (*delete x* Γ) = *thunks* $\Gamma - \{x\}$
  by (*induction* Γ) (*auto simp add*: *thunks-def*)

lemma *thunksI*[*intro*]: *map-of* Γ *x* = *Some e* $\Longrightarrow \neg$ *isVal e* $\Longrightarrow x \in$ *thunks* Γ
  by (*induction* Γ) (*auto simp add*: *thunks-def*)

lemma *thunksE*[*intro*]: $x \in$ *thunks* $\Gamma \Longrightarrow$ *map-of* Γ *x* = *Some e* $\Longrightarrow \neg$ *isVal e*
  by (*induction* Γ) (*auto simp add*: *thunks-def*)

lemma *thunks-cong*: *map-of* Γ = *map-of* $\Delta \Longrightarrow$ *thunks* Γ = *thunks* Δ
  by (*simp add*: *thunks-def*)

lemma *thunks-eqvt*[*eqvt*]:
  $\pi \cdot$ *thunks* Γ = *thunks* $(\pi \cdot \Gamma)$
    unfolding *thunks-def*
    by *perm-simp rule*

17.10 Non-recursive Let bindings

definition *nonrec* :: *heap* $\Rightarrow$ *bool* **where**
  nonrec $\Gamma = (\exists\ x\ e.\ \Gamma = [(x,e)] \wedge x \notin fv\ e)$

lemma *nonrecE*:
  assumes *nonrec* Γ
  obtains *x e* **where** $\Gamma = [(x,e)]$ **and** $x \notin fv\ e$
  using *assms*
  unfolding *nonrec-def*
  by *blast*

lemma *nonrec-eqvt*[*eqvt*]:
  nonrec $\Gamma \Longrightarrow$ *nonrec* $(\pi \cdot \Gamma)$
  apply (*erule nonrecE*)
  apply (*auto simp add*: *nonrec-def fv-def fresh-def*)
  apply (*metis fresh-at-base-permute-iff fresh-def*)
  done

lemma *exp-induct-rec*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:
  assumes $\bigwedge var.\ P\ (Var\ var)$
  assumes $\bigwedge exp\ var.\ P\ exp \Longrightarrow P\ (App\ exp\ var)$
  assumes $\bigwedge\Gamma\ exp.\ \neg\ nonrec\ \Gamma \Longrightarrow (\bigwedge\ x.\ x \in domA\ \Gamma \Longrightarrow\ P\ (the\ (map\text{-}of\ \Gamma\ x))) \Longrightarrow P\ exp$

$\Longrightarrow$ *P* (*Let* Γ *exp*)
assumes $\bigwedge$*x e exp*. *x* ∉ *fv e* $\Longrightarrow$ *P e* $\Longrightarrow$ *P exp* $\Longrightarrow$ *P* (*let x be e in exp*)
assumes $\bigwedge$*var exp*. *P exp* $\Longrightarrow$ *P* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b*. *P* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2*. *P scrut* $\Longrightarrow$ *P e1* $\Longrightarrow$ *P e2* $\Longrightarrow$ *P* (*scrut ? e1 : e2*)
shows *P exp*
apply (*rule exp-induct*[*of P*])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*case-tac nonrec* Γ)
apply (*erule nonrecE*)
apply *simp*
apply (*metis assms*(*4*))
apply (*metis assms*(*3*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply (*metis assms*(*7*))
done

lemma *exp-strong-induct-rec*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:
assumes $\bigwedge$*var c*. *P c* (*Var var*)
assumes $\bigwedge$*exp var c*. ($\bigwedge$*c*. *P c exp*) $\Longrightarrow$ *P c* (*App exp var*)
assumes $\bigwedge$Γ *exp c*.
atom ‘ *domA* Γ ♯* *c* $\Longrightarrow$ ¬ *nonrec* Γ $\Longrightarrow$ ($\bigwedge$*c x*. *x* ∈ *domA* Γ $\Longrightarrow$ *P c* (*the* (*map-of* Γ *x*)))
$\Longrightarrow$ ($\bigwedge$*c*. *P c exp*) $\Longrightarrow$ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*x e exp c*. {*atom x*} ♯* *c* $\Longrightarrow$ *x* ∉ *fv e* $\Longrightarrow$ ($\bigwedge$ *c*. *P c e*) $\Longrightarrow$ ($\bigwedge$ *c*. *P c exp*) $\Longrightarrow$ *P c* (*let x be e in exp*)
assumes $\bigwedge$*var exp c*. {*atom var*} ♯* *c* $\Longrightarrow$ ($\bigwedge$*c*. *P c exp*) $\Longrightarrow$ *P c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b c*. *P c* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2 c*. ($\bigwedge$ *c*. *P c scrut*) $\Longrightarrow$ ($\bigwedge$ *c*. *P c e1*) $\Longrightarrow$ ($\bigwedge$ *c*. *P c e2*) $\Longrightarrow$ *P c* (*scrut ? e1 : e2*)
shows *P* (*c*::′*a*::*fs*) *exp*
apply (*rule exp-strong-induct*[*of P*])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*case-tac nonrec* Γ)
apply (*erule nonrecE*)
apply *simp*
apply (*metis assms*(*4*))
apply (*metis assms*(*3*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply (*metis assms*(*7*))
done

lemma *exp-strong-induct-rec-set*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:
assumes $\bigwedge$*var c*. *P c* (*Var var*)
assumes $\bigwedge$*exp var c*. ($\bigwedge$*c*. *P c exp*) $\Longrightarrow$ *P c* (*App exp var*)
assumes $\bigwedge$Γ *exp c*.

atom ` domA Γ $\sharp$* *c* $\Longrightarrow$ $\neg$ *nonrec* Γ $\Longrightarrow$ ($\bigwedge$*c x e*. (*x*,*e*) $\in$ *set* Γ $\Longrightarrow$ *P c e*) $\Longrightarrow$ ($\bigwedge$*c*. *P c exp*) $\Longrightarrow$ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*x e exp c*. {*atom x*} $\sharp$* *c* $\Longrightarrow$ *x* $\notin$ *fv e* $\Longrightarrow$ ($\bigwedge$ *c*. *P c e*) $\Longrightarrow$ ($\bigwedge$ *c*. *P c exp*) $\Longrightarrow$ *P c* (*let x be e in exp*)
assumes $\bigwedge$*var exp c*. {*atom var*} $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c*. *P c exp*) $\Longrightarrow$ *P c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b c*. *P c* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2 c*. ($\bigwedge$ *c*. *P c scrut*) $\Longrightarrow$ ($\bigwedge$ *c*. *P c e1*) $\Longrightarrow$ ($\bigwedge$ *c*. *P c e2*) $\Longrightarrow$ *P c* (*scrut ? e1 : e2*)
shows *P* (*c*::*'a*::*fs*) *exp*
apply (*rule exp-strong-induct-set*(*1*)[*of P*])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*case-tac nonrec* Γ)
apply (*erule nonrecE*)
apply *simp*
apply (*metis assms*(*4*))
apply (*metis assms*(*3*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply (*metis assms*(*7*))
done

17.11 Renaming a lambda-bound variable

lemma *change-Lam-Variable*:
assumes *y'* $\neq$ *y* $\Longrightarrow$ *atom y'* $\sharp$ (*e*, *y*)
shows *Lam* [*y*]. *e* = *Lam* [*y'*]. ((*y* $\leftrightarrow$ *y'*) $\cdot$ *e*)
proof(*cases y'* = *y*)
case *True* **thus** *?thesis* **by** *simp*
next
case *False*
from *assms*[*OF this*]
have (*y* $\leftrightarrow$ *y'*) $\cdot$ (*Lam* [*y*]. *e*) = *Lam* [*y*]. *e*
by −(*rule flip-fresh-fresh*, (*simp add*: *fresh-Pair*)+)
moreover
have (*y* $\leftrightarrow$ *y'*) $\cdot$ (*Lam* [*y*]. *e*) = *Lam* [*y'*]. ((*y* $\leftrightarrow$ *y'*) $\cdot$ *e*)
by *simp*
ultimately
show *Lam* [*y*]. *e* = *Lam* [*y'*]. ((*y* $\leftrightarrow$ *y'*) $\cdot$ *e*) **by** (*simp add*: *fresh-Pair*)
qed

end

18 AbstractDenotational.tex

theory *AbstractDenotational*
imports *HeapSemantics Terms*

begin

18.1 The denotational semantics for expressions

Because we need to define two semantics later on, we are abstract in the actual domain.

locale *semantic-domain* =
  fixes *Fn* :: (*'Value* → *'Value*) → (*'Value*::{*pcpo-pt*,*pure*})
  fixes *Fn-project* :: *'Value* → (*'Value* → *'Value*)
  fixes *B* :: *bool discr* → *'Value*
  fixes *B-project* :: *'Value* → *'Value* → *'Value* → *'Value*
  fixes *tick* :: *'Value* → *'Value*
begin

nominal-function
  ESem :: *exp* ⇒ (*var* ⇒ *'Value*) → *'Value*
where
 ESem (*Lam* [*x*]. *e*) = (Λ ϱ. *tick*·(*Fn*·(Λ *v*. *ESem e*·((ϱ *f*|' *fv* (*Lam* [*x*]. *e*))(*x* := *v*)))))
| *ESem* (*App e x*) = (Λ ϱ. *tick*·(*Fn-project*·(*ESem e*·ϱ)·(ϱ *x*)))
| *ESem* (*Var x*) = (Λ ϱ. *tick*·(ϱ *x*))
| *ESem* (*Let as body*) = (Λ ϱ. *tick*·(*ESem body*·(*has-ESem.HSem ESem as*·(ϱ *f*|' *fv* (*Let as body*)))))
| *ESem* (*Bool b*) = (Λ ϱ. *tick*·(*B*·(*Discr b*)))
| *ESem* (*scrut ? e1* : *e2*) = (Λ ϱ. *tick*·((*B-project*·(*ESem scrut*·ϱ))·(*ESem e1*·ϱ)·(*ESem e2*·ϱ)))
proof *goal-cases*

The following proofs discharge technical obligations generated by the Nominal package.

case *1* **thus** *?case*
  unfolding *eqvt-def ESem-graph-aux-def*
  apply *rule*
  apply (*perm-simp*)
  apply (*simp add*: *Abs-cfun-eqvt*)
  apply (*simp add*: *unpermute-def permute-pure*)
  done
next
case (*3 P x*)
  thus *?case* **by** (*metis* (*poly-guards-query*) *exp-strong-exhaust*)
next

case *prems*: (*4 x e x' e'*)
  from *prems*(*5*)
  show *?case*
  proof (*rule eqvt-lam-case*)
    fix π :: *perm*
    assume ∗: *supp* (−π) ♯∗ (*fv* (*Lam* [*x*]. *e*) :: *var set*)
    { fix ϱ *v*
      have *ESem-sumC* (π • *e*)·((ϱ *f*|' *fv* (*Lam* [*x*]. *e*))((π • *x*) := *v*)) = − π • *ESem-sumC* (π • *e*)·((ϱ *f*|' *fv* (*Lam* [*x*]. *e*))((π • *x*) := *v*))
        by (*simp add*: *permute-pure*)

also have ... = *ESem-sumC* $e\cdot((-\ \pi \cdot (\varrho\ f|`\ fv\ (Lam\ [x].\ e)))(x := v))$ **by** (*simp add*: *pemute-minus-self eqvt-at-apply*[*OF prems*(*1*)])
also have $-\ \pi \cdot (\varrho\ f|`\ fv\ (Lam\ [x].\ e)) = (\varrho\ f|`\ fv\ (Lam\ [x].\ e))$ **by** (*rule env-restr-perm'*[*OF* $*$]) *auto*
finally have *ESem-sumC* $(\pi \cdot e)\cdot((\varrho\ f|`\ fv\ (Lam\ [x].\ e))((\pi \cdot x) := v)) =$ *ESem-sumC* $e\cdot((\varrho\ f|`\ fv\ (Lam\ [x].\ e))(x := v))$**.**
}
thus $(\Lambda\ \varrho.\ tick\cdot(Fn\cdot(\Lambda\ v.$ *ESem-sumC* $(\pi \cdot e)\cdot((\varrho\ f|`\ fv\ (Lam\ [x].\ e))(\pi \cdot x := v))))) = (\Lambda\ \varrho.\ tick\cdot(Fn\cdot(\Lambda\ v.$ *ESem-sumC* $e\cdot((\varrho\ f|`\ fv\ (Lam\ [x].\ e))(x := v)))))$ **by** *simp*
qed
next

case *prems*: (*19 as body as' body'*)
from *prems*(*9*)
show *?case*
proof (*rule eqvt-let-case*)
fix π :: *perm*
assume $*$: *supp* $(-\pi)$ $\sharp*$ (*fv* (*Terms.Let as body*) :: *var set*)

{ fix ϱ
have *ESem-sumC* $(\pi \cdot body)\cdot($*has-ESem.HSem ESem-sumC* $(\pi \cdot as)\cdot(\varrho\ f|`\ fv$ (*Terms.Let as body*)))
$= -\ \pi \cdot$ *ESem-sumC* $(\pi \cdot body)\cdot($*has-ESem.HSem ESem-sumC* $(\pi \cdot as)\cdot(\varrho\ f|`\ fv$ (*Terms.Let as body*)))
by (*rule permute-pure*[*symmetric*])
also have ... = $(-\ \pi \cdot$ *ESem-sumC*) $body\cdot($*has-ESem.HSem* $(-\ \pi \cdot$ *ESem-sumC*) $as\cdot(-\ \pi \cdot \varrho\ f|`\ fv$ (*Terms.Let as body*)))
by (*simp add*: *pemute-minus-self*)
also have $(-\ \pi \cdot$ *ESem-sumC*) *body* = *ESem-sumC body*
by (*rule eqvt-at-apply*[*OF* ‹*eqvt-at ESem-sumC body*›])
also have *has-ESem.HSem* $(-\ \pi \cdot$ *ESem-sumC*) *as* = *has-ESem.HSem ESem-sumC as*
by (*rule HSem-cong*[*OF eqvt-at-apply*[*OF prems*(*2*)] *refl*])
also have $-\ \pi \cdot \varrho\ f|`\ fv$ (*Let as body*) $= \varrho\ f|`\ fv$ (*Let as body*)
by (*rule env-restr-perm'*[*OF* $*$], *simp*)
finally have *ESem-sumC* $(\pi \cdot body)\cdot($*has-ESem.HSem ESem-sumC* $(\pi \cdot as)\cdot(\varrho\ f|`\ fv$ (*Let as body*))) = *ESem-sumC* $body\cdot($*has-ESem.HSem ESem-sumC* $as\cdot(\varrho\ f|`\ fv$ (*Let as body*)))**.**
}
thus $(\Lambda\ \varrho.\ tick\cdot($*ESem-sumC* $(\pi \cdot body)\cdot($*has-ESem.HSem ESem-sumC* $(\pi \cdot as)\cdot(\varrho\ f|`\ fv$ (*Let as body*))))) =
$(\Lambda\ \varrho.\ tick\cdot($*ESem-sumC* $body\cdot($*has-ESem.HSem ESem-sumC* $as\cdot(\varrho\ f|`\ fv$ (*Let as body*))))) **by** (*simp only*:)
qed
qed *auto*

nominal-termination (**in** *semantic-domain*) (*no-eqvt*) **by** *lexicographic-order*

sublocale *has-ESem ESem***.**

abbreviation *ESem-syn'* ($[\![\text{-}]\!]_{\text{-}}$ [*60*,*60*] *60*) **where** $[\![e]\!]_{\varrho} \equiv ESem\ e \cdot \varrho$

abbreviation *EvalHeapSem-syn′* ($[\![$ - $]\!]_-$ *[0,0] 110*) **where** $[\![\Gamma]\!]_\varrho \equiv$ *evalHeap* Γ (λ *e*. $[\![e]\!]_\varrho$)
abbreviation *AHSem-syn* ($\{\!|$-$|\!\}$- *[60,60] 60*) **where** $\{\!|\Gamma|\!\}\varrho \equiv$ *HSem* $\Gamma \cdot \varrho$
abbreviation *AHSem-bot* ($\{\!|$-$|\!\}$ *[60] 60*) **where** $\{\!|\Gamma|\!\} \equiv \{\!|\Gamma|\!\}\bot$

end
end

19 Substitution.tex

theory *Substitution*
imports *Terms*
begin

Defining a substitution function on terms turned out to be slightly tricky.

fun
  subst-var :: *var* ⇒ *var* ⇒ *var* ⇒ *var* (-[-::v=-] *[1000,100,100] 1000*)
where *x*[*y* ::*v*= *z*] = (*if x* = *y then z else x*)

nominal-function (*default case-sum* (λ*x*. *Inl undefined*) (λ*x*. *Inr undefined*),
    invariant λ *a r* . (∀ Γ *y z* . ((*a* = *Inr* (Γ, *y*, *z*) ∧ *atom* ‘ *domA* Γ ♯∗ (*y*, *z*)) ⟶ *map* (λ*x* . *atom* (*fst x*)) (*Sum-Type.projr r*) = *map* (λ*x* . *atom* (*fst x*)) Γ)))
  subst :: *exp* ⇒ *var* ⇒ *var* ⇒ *exp* (-[-::=-] *[1000,100,100] 1000*)
and
  subst-heap :: *heap* ⇒ *var* ⇒ *var* ⇒ *heap* (-[-::h=-] *[1000,100,100] 1000*)
where
  (*Var x*)[*y* ::= *z*] = *Var* (*x*[*y* ::*v*= *z*])
 | (*App e v*)[*y* ::= *z*] = *App* (*e*[*y* ::= *z*]) (*v*[*y* ::*v*= *z*])
 | *atom* ‘ *domA* Γ ♯∗ (*y*,*z*) ⟹
    (*Let* Γ *body*)[*y* ::= *z*] = *Let* (Γ[*y* ::*h*= *z*]) (*body*[*y* ::= *z*])
 | *atom x* ♯ (*y*,*z*) ⟹ (*Lam* [*x*].*e*)[*y* ::= *z*] = *Lam* [*x*].(*e*[*y*::=*z*])
 | (*Bool b*)[*y* ::= *z*] = *Bool b*
 | (*scrut ? e1 : e2*)[*y* ::= *z*] = (*scrut*[*y* ::= *z*] *?* *e1*[*y* ::= *z*] : *e2*[*y* ::= *z*])
 | [][*y* ::*h*= *z*] = []
 | ((*v*,*e*)# Γ)[*y* ::*h*= *z*] = (*v*, *e*[*y* ::= *z*])# (Γ[*y* ::*h*= *z*])
proof *goal-cases*

have *eqvt-at-subst*: ⋀ *e y z* . *eqvt-at subst-subst-heap-sumC* (*Inl* (*e*, *y*, *z*)) ⟹ *eqvt-at* (λ(*a*, *b*, *c*). *subst a b c*) (*e*, *y*, *z*)
  apply(*simp add*: *eqvt-at-def subst-def*)
  apply(*rule*)
  apply(*subst Projl-permute*)
  apply(*thin-tac* -)+
  apply (*simp add*: *subst-subst-heap-sumC-def*)
  apply (*simp add*: *THE-default-def*)
  apply (*case-tac Ex1* (*subst-subst-heap-graph* (*Inl* (*e*, *y*, *z*))))
  apply(*simp*)
  apply(*auto*)[*1*]

apply (*erule-tac x=x* **in** *allE*)
apply *simp*
apply(*cases rule*: *subst-subst-heap-graph.cases*)
apply(*assumption*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply (*metis Inr-not-Inl*)
apply (*metis Inr-not-Inl*)
apply(*simp*)
apply(*perm-simp*)
apply(*simp*)
done

have *eqvt-at-subst-heap*: $\bigwedge \Gamma\ y\ z\ .\ eqvt\text{-}at\ subst\text{-}subst\text{-}heap\text{-}sumC\ (Inr\ (\Gamma,\ y,\ z)) \Longrightarrow eqvt\text{-}at\ (\lambda(a,\ b,\ c).\ subst\text{-}heap\ a\ b\ c)\ (\Gamma,\ y,\ z)$
apply(*simp add*: *eqvt-at-def subst-heap-def*)
apply(*rule*)
apply(*subst Projr-permute*)
apply(*thin-tac -*)+
apply (*simp add*: *subst-subst-heap-sumC-def*)
apply (*simp add*: *THE-default-def*)

```
  apply (case-tac Ex1 (subst-subst-heap-graph (Inr (Γ, y, z))))
  apply(simp)
  apply(auto)[1]
  apply (erule-tac x=x in allE)
  apply simp
  apply(cases rule: subst-subst-heap-graph.cases)
  apply(assumption)
  apply (metis (mono-tags) Inr-not-Inl)+
  apply(rule-tac x=Sum-Type.projr x in exI)
  apply(clarify)
  apply (rule the1-equality)
  apply auto[1]
  apply(simp (no-asm) only: sum.sel)

  apply(rule-tac x=Sum-Type.projr x in exI)
  apply(clarify)
  apply (rule the1-equality)
  apply auto[1]
  apply(simp (no-asm) only: sum.sel)

  apply(simp)
  apply(perm-simp)
  apply(simp)
done

{

case 1 thus ?case
  unfolding eqvt-def subst-subst-heap-graph-aux-def
  by simp


next case 2 thus ?case
 by (induct rule: subst-subst-heap-graph.induct)(auto simp add: exp-assn.bn-defs fresh-star-insert)


next case prems: (3 P x) show ?case
  proof(cases x)
  case (Inl a) thus P
    proof(cases a)
    case (fields a1 a2 a3)
    thus P using Inl prems
      apply (rule-tac y =a1 and c =(a2, a3) in exp-strong-exhaust)
      apply (auto simp add: fresh-star-def)
    done
  qed
  next
  case (Inr a) thus P
    proof (cases a)
```

case (*fields a1 a2 a3*)
thus P **using** *Inr prems*
by (*metis heapToAssn.cases*)
qed
qed

next case (*19 e y2 z2* Γ *e2 y z as2*) **thus** *?case*
apply $-$
apply (*drule eqvt-at-subst*)+
apply (*drule eqvt-at-subst-heap*)+
apply (*simp only*: *meta-eq-to-obj-eq*[*OF subst-def*, *symmetric*, *unfolded fun-eq-iff*]
meta-eq-to-obj-eq[*OF subst-heap-def*, *symmetric*, *unfolded fun-eq-iff*])

apply (*auto simp add*: *Abs-fresh-iff*)
apply (*drule-tac*
$c = (y,\ z)$ **and**
$as = (map\ (\lambda x.\ atom\ (fst\ x))\ e)$ **and**
$bs = (map\ (\lambda x.\ atom\ (fst\ x))\ e2)$ **and**
$f = \lambda\ a\ b\ c\ .\ [a]lst.\ (subst\ (fst\ b)\ y\ z,\ subst\text{-}heap\ (snd\ b)\ y\ z\ )$ **in** *Abs-lst-fcb2*)
apply (*simp add*: *perm-supp-eq fresh-Pair fresh-star-def Abs-fresh-iff*)
apply (*metis domA-def image-image image-set*)
apply (*metis domA-def image-image image-set*)
apply (*simp add*: *eqvt-at-def*, *simp add*: *fresh-star-Pair perm-supp-eq*)
apply (*simp add*: *eqvt-at-def*, *simp add*: *fresh-star-Pair perm-supp-eq*)
apply (*simp add*: *eqvt-at-def*)
done

next case (*25 x2 y2 z2 e2 x y z e*) **thus** *?case*
apply $-$
apply (*drule eqvt-at-subst*)+
apply (*simp only*: *Abs-fresh-iff meta-eq-to-obj-eq*[*OF subst-def*, *symmetric*, *unfolded fun-eq-iff*])

apply (*simp add*: *eqvt-at-def*)
apply *rule*
apply (*erule-tac* $x = (x2 \leftrightarrow c)$ **in** *allE*)
apply (*erule-tac* $x = (x \leftrightarrow c)$ **in** *allE*)
apply *auto*
done
}
qed(*auto*)

nominal-termination (*eqvt*) **by** *lexicographic-order*

lemma shows
True **and** *bn-subst*[*simp*]: *domA* (*subst-heap* Γ *y z*) = *domA* Γ
by(*induct rule*:*subst-subst-heap.induct*)
(*auto simp add*: *exp-assn.bn-defs fresh-star-insert*)

lemma *subst-noop*[*simp*]:
shows $e[y ::= y] = e$ **and** $\Gamma[y::h=y]= \Gamma$
by(*induct e y y* **and** Γ *y y rule*:*subst-subst-heap.induct*)
(*auto simp add*:*fresh-star-Pair exp-assn.bn-defs*)

lemma *subst-is-fresh*[*simp*]:
assumes *atom* $y \sharp z$
shows
atom $y \sharp e[y ::= z]$
and
atom ‘ *domA* $\Gamma \sharp* y \Longrightarrow$ *atom* $y \sharp \Gamma[y::h=z]$
using *assms*
by(*induct e y z* **and** Γ *y z rule*:*subst-subst-heap.induct*)
(*auto simp add*:*fresh-at-base fresh-star-Pair fresh-star-insert fresh-Nil fresh-Cons pure-fresh*)

lemma
subst-pres-fresh: *atom* $x \sharp e \vee x = y \Longrightarrow$ *atom* $x \sharp z \Longrightarrow$ *atom* $x \sharp e[y ::= z]$
and
atom $x \sharp \Gamma \vee x = y \Longrightarrow$ *atom* $x \sharp z \Longrightarrow x \notin$ *domA* $\Gamma \Longrightarrow$ *atom* $x \sharp (\Gamma[y ::h= z])$
by(*induct e y z* **and** Γ *y z rule*:*subst-subst-heap.induct*)
(*auto simp add*:*fresh-star-Pair exp-assn.bn-defs fresh-Cons fresh-Nil pure-fresh*)

lemma *subst-fresh-noop*: *atom* $x \sharp e \Longrightarrow e[x ::= y] = e$
and *subst-heap-fresh-noop*: *atom* $x \sharp \Gamma \Longrightarrow \Gamma[x ::h= y] = \Gamma$
by (*nominal-induct e* **and** Γ *avoiding*: *x y rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-def fresh-Pair fresh-at-base fresh-Cons simp del*: *exp-assn.eq-iff*)

lemma *supp-subst-eq*: *supp* $(e[y::=x])$ = (*supp* e − {*atom* y}) ∪ (*if atom* $y \in$ *supp* e *then* {*atom* x} *else* {})
and *atom* ‘ *domA* $\Gamma \sharp* y \Longrightarrow$ *supp* $(\Gamma[y::h=x])$ = (*supp* Γ − {*atom* y}) ∪ (*if atom* $y \in$ *supp* Γ *then* {*atom* x} *else* {})
by (*nominal-induct e* **and** Γ *avoiding*: *x y rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-def fresh-Pair supp-Nil supp-Cons supp-Pair fresh-Cons exp-assn.supp Let-supp supp-at-base pure-supp simp del*: *exp-assn.eq-iff*)

lemma *supp-subst*: *supp* $(e[y::=x]) \subseteq$ (*supp* e − {*atom* y}) ∪ {*atom* x}
using *supp-subst-eq* **by** *auto*

lemma *fv-subst-eq*: *fv* $(e[y::=x])$ = (*fv* $e - \{y\}) \cup$ (*if* $y \in$ *fv* e *then* $\{x\}$ *else* {})
and *atom* ‘ *domA* $\Gamma \sharp* y \Longrightarrow$ *fv* $(\Gamma[y::h=x])$ = (*fv* $\Gamma - \{y\}) \cup$ (*if* $y \in$ *fv* Γ *then* $\{x\}$ *else* {})
by (*nominal-induct e* **and** Γ *avoiding*: *x y rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-def fresh-Pair supp-Nil supp-Cons supp-Pair fresh-Cons exp-assn.supp Let-supp supp-at-base simp del*: *exp-assn.eq-iff*)

lemma *fv-subst-subset*: *fv* $(e[y ::= x]) \subseteq$ (*fv* $e - \{y\}) \cup \{x\}$
using *fv-subst-eq* **by** *auto*

lemma *fv-subst-int*: $x \notin S \Longrightarrow y \notin S \Longrightarrow$ *fv* $(e[y ::= x]) \cap S =$ *fv* $e \cap S$

by (*auto simp add*: *fv-subst-eq*)

lemma *fv-subst-int2*: $x \notin S \Longrightarrow y \notin S \Longrightarrow S \cap fv\ (e[y ::= x]) = S \cap fv\ e$
by (*auto simp add*: *fv-subst-eq*)

lemma *subst-swap-same*: $atom\ x \sharp e \Longrightarrow (x \leftrightarrow y) \cdot e = e[y ::=x]$
and $atom\ x \sharp \Gamma \Longrightarrow atom\ `domA\ \Gamma \sharp{*}\ y \Longrightarrow (x \leftrightarrow y) \cdot \Gamma = \Gamma[y ::h= x]$
by (*nominal-induct* e **and** Γ *avoiding*: $x\ y$ *rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-Pair fresh-star-at-base fresh-Cons pure-fresh permute-pure simp del*: *exp-assn.eq-iff*)

lemma *subst-subst-back*: $atom\ x \sharp e \Longrightarrow e[y::=x][x::=y] = e$
and $atom\ x \sharp \Gamma \Longrightarrow atom\ `domA\ \Gamma \sharp{*}\ y \Longrightarrow \Gamma[y::h=x][x::h=y] = \Gamma$
by(*nominal-induct* e **and** Γ *avoiding*: $x\ y$ *rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-Pair fresh-star-at-base fresh-star-Cons fresh-Cons exp-assn.bn-defs simp del*: *exp-assn.eq-iff*)

lemma *subst-heap-delete*[*simp*]: $(delete\ x\ \Gamma)[y ::h= z] = delete\ x\ (\Gamma[y ::h= z])$
by (*induction* Γ) *auto*

lemma *subst-nil-iff*[*simp*]: $\Gamma[x ::h= z] = [] \longleftrightarrow \Gamma = []$
by (*cases* Γ) *auto*

lemma *subst-SmartLet*[*simp*]:
$atom\ `\ domA\ \Gamma \sharp{*}\ (y,z) \Longrightarrow (SmartLet\ \Gamma\ body)[y ::= z] = SmartLet\ (\Gamma[y ::h= z])\ (body[y ::= z])$
unfolding *SmartLet-def* **by** *auto*

lemma *subst-let-be*[*simp*]:
$atom\ x' \sharp y \Longrightarrow atom\ x' \sharp x \Longrightarrow (let\ x'\ be\ e\ in\ exp\ )[y::=x] = (let\ x'\ be\ e[y::=x]\ in\ exp[y::=x])$
by (*simp add*: *fresh-star-def fresh-Pair*)

lemma *isLam-subst*[*simp*]: $isLam\ e[x::=y] = isLam\ e$
by (*nominal-induct e avoiding*: $x\ y$ *rule*: *exp-strong-induct*)
(*auto simp add*: *fresh-star-Pair*)

lemma *isVal-subst*[*simp*]: $isVal\ e[x::=y] = isVal\ e$
by (*nominal-induct e avoiding*: $x\ y$ *rule*: *exp-strong-induct*)
(*auto simp add*: *fresh-star-Pair*)

lemma *thunks-subst*[*simp*]:
$thunks\ \Gamma[y::h=x] = thunks\ \Gamma$
by (*induction* Γ) (*auto simp add*: *thunks-Cons*)

lemma *map-of-subst*:
$map\text{-}of\ (\Gamma[x::h=y])\ k = map\text{-}option\ (\lambda\ e\ .\ e[x::=y])\ (map\text{-}of\ \Gamma\ k)$
by (*induction* Γ) *auto*

lemma *mapCollect-subst*[*simp*]:

$\{e\ k\ v \mid k{\mapsto}v{\in}\mathit{map\text{-}of}\ \Gamma[x{::}h{=}y]\} = \{e\ k\ v[x{::=}y] \mid k{\mapsto}v{\in}\mathit{map\text{-}of}\ \Gamma\}$
by (*auto simp add*: *map-of-subst*)

lemma *subst-eq-Cons*:
$\Gamma[x{::}h{=}y] = (x', e)\#\Delta \longleftrightarrow (\exists\ e'\ \Gamma'.\ \Gamma = (x',e')\#\Gamma' \wedge e'[x{::=}y] = e \wedge \Gamma'[x{::}h{=}y] = \Delta)$
by (*cases* Γ) *auto*

lemma *nonrec-subst*:
$\mathit{atom}\ `\ \mathit{domA}\ \Gamma\ \sharp{*}\ x \Longrightarrow \mathit{atom}\ `\ \mathit{domA}\ \Gamma\ \sharp{*}\ y \Longrightarrow \mathit{nonrec}\ \Gamma[x{::}h{=}y] \longleftrightarrow \mathit{nonrec}\ \Gamma$
by (*auto simp add*: *nonrec-def fresh-star-def subst-eq-Cons fv-subst-eq*)

end

20 Abstract-Denotational-Props.tex

theory *Abstract−Denotational−Props*
imports *AbstractDenotational Substitution*
begin

context *semantic-domain*
begin

20.1 The semantics ignores fresh variables

lemma *ESem-considers-fv'*: $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho\ f|`\ (fv\ e)}$
proof (*induct e arbitrary*: ϱ *rule*:*exp-induct*)
case *Var*
show *?case* **by** *simp*
next
have [*simp*]: $\bigwedge\ S\ x.\ S \cap \mathit{insert}\ x\ S = S$ **by** *auto*
case *App*
show *?case*
by (*simp*, *subst* (*1 2*) *App*, *simp*)
next
case (*Lam x e*)
show *?case* **by** *simp*
next
case (*IfThenElse scrut* e_1 e_2)
have [*simp*]: $(fv\ scrut \cap (fv\ scrut \cup fv\ e_1 \cup fv\ e_2)) = fv\ scrut$ **by** *auto*
have [*simp*]: $(fv\ e_1 \cap (fv\ scrut \cup fv\ e_1 \cup fv\ e_2)) = fv\ e_1$ **by** *auto*
have [*simp*]: $(fv\ e_2 \cap (fv\ scrut \cup fv\ e_1 \cup fv\ e_2)) = fv\ e_2$ **by** *auto*
show *?case*
apply *simp*
apply (*subst* (*1 2*) *IfThenElse*(*1*))
apply (*subst* (*1 2*) *IfThenElse*(*2*))
apply (*subst* (*1 2*) *IfThenElse*(*3*))
apply (*simp*)
done

next
  case (*Let as e*)

  have $[\![e]\!]_{\{\!|as|\!\}\varrho} = [\![e]\!]_{(\{\!|as|\!\}\varrho)\ f|`\ (fv\ as \cup fv\ e)}$
    apply (*subst* (*1 2*) *Let*(*2*))
    apply *simp*
    apply (*metis inf-sup-absorb sup-commute*)
    done
  also
  have *fv as* ⊆ *fv as* ∪ *fv e* **by** (*rule inf-sup-ord*(*3*))
  hence $(\{\!|as|\!\}\varrho)\ f|`\ (fv\ as \cup fv\ e) = \{\!|as|\!\}(\varrho\ f|`\ (fv\ as \cup fv\ e))$
    by (*rule HSem-ignores-fresh-restr'*[*OF - Let*(*1*)])
  also
  have $\{\!|as|\!\}(\varrho\ f|`\ (fv\ as \cup fv\ e)) = \{\!|as|\!\}\varrho\ f|`\ (fv\ as \cup fv\ e - domA\ as)$
    by (*rule HSem-restr-cong*) (*auto simp add*: *lookup-env-restr-eq*)
  finally
  show *?case* **by** *simp*
qed *auto*

sublocale *has-ignore-fresh-ESem ESem*
  by *standard* (*rule fv-supp-exp*, *rule ESem-considers-fv'*)

20.2 Nicer equations for ESem, without freshness requirements

lemma *ESem-Lam*[*simp*]: $[\![\ Lam\ [x].\ e\ ]\!]_{\varrho} = tick \cdot (Fn \cdot (\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho(x\ :=\ v)}))$
proof−
  have $*$: $\bigwedge\ v.\ ((\varrho\ f|`\ (fv\ e - \{x\}))(x := v))\ f|`\ fv\ e = (\varrho(x := v))\ f|`\ fv\ e$
    by (*rule ext*) (*auto simp add*: *lookup-env-restr-eq*)

  have $[\![\ Lam\ [x].\ e\ ]\!]_{\varrho} = [\![\ Lam\ [x].\ e\ ]\!]_{env\text{-}delete\ x\ \varrho}$
    by (*rule ESem-fresh-cong*) *simp*
  also have $\ldots = tick \cdot (Fn \cdot (\Lambda\ v.\ [\![\ e\ ]\!]_{(\varrho\ f|`\ (fv\ e\ -\ \{x\}))(x\ :=\ v)}))$
    by *simp*
  also have $\ldots = tick \cdot (Fn \cdot (\Lambda\ v.\ [\![\ e\ ]\!]_{((\varrho\ f|`\ (fv\ e\ -\ \{x\}))(x\ :=\ v))\ f|`\ fv\ e}))$
    by (*subst ESem-considers-fv*, *rule*)
  also have $\ldots = tick \cdot (Fn \cdot (\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho(x\ :=\ v)\ f|`\ fv\ e}))$
    unfolding $*$**..**
  also have $\ldots = tick \cdot (Fn \cdot (\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho(x\ :=\ v)}))$
    unfolding *ESem-considers-fv*[*symmetric*]**..**
  finally show *?thesis***.**
qed
declare *ESem.simps*(*1*)[*simp del*]

lemma *ESem-Let*[*simp*]: $[\![Let\ as\ body]\!]_{\varrho} = tick \cdot ([\![body]\!]_{\{\!|as|\!\}\varrho})$
proof−
  have $[\![\ Let\ as\ body\ ]\!]_{\varrho} = tick \cdot ([\![body]\!]_{\{\!|as|\!\}(\varrho\ f|`\ fv\ (Let\ as\ body))})$
    by *simp*
  also have $\{\!|as|\!\}(\varrho\ f|`\ fv(Let\ as\ body)) = \{\!|as|\!\}(\varrho\ f|`\ (fv\ as \cup fv\ body))$

by (*rule HSem-restr-cong*) (*auto simp add*: *lookup-env-restr-eq*)
also have ... = ($\{\!|as|\!\}\varrho$) $f|$‘ (*fv as* ∪ *fv body*)
by (*rule HSem-ignores-fresh-restr′*[*symmetric*, *OF* - *ESem-considers-fv*]) *simp*
also have $[\![body]\!]_{\ldots}$ = $[\![body]\!]_{\{\!|as|\!\}\varrho}$
by (*rule ESem-fresh-cong*) (*auto simp add*: *lookup-env-restr-eq*)
finally show *?thesis*.
qed
declare *ESem.simps*(*4*)[*simp del*]

20.3 Denotation of Substitution

lemma *ESem-subst-same*: $\varrho\ x = \varrho\ y \Longrightarrow [\![\ e\ ]\!]_{\varrho} = [\![\ e[x{::=}\ y]\ ]\!]_{\varrho}$
and
$\varrho\ x = \varrho\ y \Longrightarrow ([\![\ as\ ]\!]_{\varrho}) = [\![\ as[x{::}h{=}y]\ ]\!]_{\varrho}$
proof (*nominal-induct e* **and** *as avoiding*: *x y arbitrary*: ϱ **and** ϱ *rule*:*exp-heap-strong-induct*)
case *Var* **thus** *?case* **by** *auto*
next
case *App*
from *App*(*1*)[*OF App*(*2*)] *App*(*2*)
show *?case* **by** *auto*
next
case (*Let as exp x y* ϱ)
from ‹*atom* ‘ *domA as* ♯∗ *x*› ‹*atom* ‘ *domA as* ♯∗ *y*›
have *x* ∉ *domA as y* ∉ *domA as*
by (*metis fresh-star-at-base imageI*)+
hence [*simp*]:*domA* (*as*[*x*::*h*=*y*]) = *domA as*
by (*metis bn-subst*)

from ‹$\varrho\ x = \varrho\ y$›
have ($\{\!|as|\!\}\varrho$) *x* = ($\{\!|as|\!\}\varrho$) *y*
using ‹*x* ∉ *domA as*› ‹*y* ∉ *domA as*›
by (*simp add*: *lookup-HSem-other*)
hence $[\![exp]\!]_{\{\!|as|\!\}\varrho} = [\![exp[x{::=}y]]\!]_{\{\!|as|\!\}\varrho}$
by (*rule Let*)
moreover
from ‹$\varrho\ x = \varrho\ y$›
have $\{\!|as|\!\}\varrho = \{\!|as[x{::}h{=}y]|\!\}\varrho$ **and** ($\{\!|as|\!\}\varrho$) *x* = ($\{\!|as[x{::}h{=}y]|\!\}\varrho$) *y*
apply (*induction rule*: *parallel-HSem-ind*)
apply (*intro adm-lemmas cont2cont cont2cont-fun*)
apply *simp*
apply *simp*
apply *simp*
apply (*erule arg-cong*[*OF Let*(*3*)])
using ‹*x* ∉ *domA as*› ‹*y* ∉ *domA as*›
apply *simp*
done
ultimately
show *?case* **using** *Let*(*1*,*2*,*3*) **by** (*simp add*: *fresh-star-Pair*)
next

case (*Lam var exp x y* ϱ)
  from ‹ϱ x = ϱ y›
  have $\bigwedge v.$ (ϱ(*var* := *v*)) *x* = (ϱ(*var* := *v*)) *y*
    using *Lam*(*1*,*2*) **by** (*simp add*: *fresh-at-base*)
  hence $\bigwedge$ *v*. $[\![exp]\!]_{\varrho(var\ :=\ v)} = [\![exp[x::=y]]\!]_{\varrho(var\ :=\ v)}$
    by (*rule Lam*)
  thus *?case* **using** *Lam*(*1*,*2*) **by** *simp*
next
case *IfThenElse*
  from *IfThenElse*(*1*)[*OF IfThenElse*(*4*)] *IfThenElse*(*2*)[*OF IfThenElse*(*4*)] *IfThenElse*(*3*)[*OF IfThenElse*(*4*)]
  show *?case*
    by *simp*
next
case *Nil* **thus** *?case* **by** *auto*
next
case *Cons*
  from *Cons*(*1*,*2*)[*OF Cons*(*3*)] *Cons*(*3*)
  show *?case* **by** *auto*
qed *auto*

lemma *ESem-subst*:
  shows $[\![\ e\ ]\!]_{\sigma(x\ :=\ \sigma\ y)} = [\![\ e[x::=\ y]\ ]\!]_{\sigma}$
proof(*cases x* = *y*)
  case *False*
  hence [*simp*]: $x \notin$ *fv e*[*x*::=*y*] **by** (*auto simp add*: *fv-def supp-subst supp-at-base dest*: *set-mp*[*OF supp-subst*])

  have $[\![\ e\ ]\!]_{\sigma(x\ :=\ \sigma\ y)} = [\![\ e[x::=\ y]\ ]\!]_{\sigma(x\ :=\ \sigma\ y)}$
    by (*rule ESem-subst-same*) *simp*
  also have ... = $[\![\ e[x::=\ y]\ ]\!]_{\sigma}$
    by (*rule ESem-fresh-cong*) *simp*
  finally
  show *?thesis*.
next
  case *True*
  thus *?thesis* **by** *simp*
qed

end

end

21 Value.tex

theory *Value*
  imports ~~/*src*/*HOL*/*HOLCF*/*HOLCF*

begin

21.1 The semantic domain for values and environments

domain *Value* = *Fn* (**lazy** *Value* → *Value*) | *B* (**lazy** *bool discr*)

fixrec *Fn-project* :: *Value* → *Value* → *Value*
where *Fn-project*·(*Fn*·*f*) = *f*

abbreviation *Fn-project-abbr* (**infix** ↓*Fn* *55*)
where *f* ↓*Fn* *v* ≡ *Fn-project*·*f*·*v*

lemma [*simp*]:
⊥ ↓*Fn* *x* = ⊥
(*B*·*b*) ↓*Fn* *x* = ⊥
by (*fixrec-simp*)+

fixrec *B-project* :: *Value* → *Value* → *Value* → *Value* **where**
B-project·(*B*·*db*)·v_1·v_2 = (*if undiscr db then* v_1 *else* v_2)

lemma [*simp*]:
B-project·(*B*·(*Discr b*))·v_1·v_2 = (*if b then* v_1 *else* v_2)
B-project·⊥·v_1·v_2 = ⊥
B-project·(*Fn*·*f*)·v_1·v_2 = ⊥
by *fixrec-simp*+

A chain in the domain *Value* is either always bottom, or eventually *Fn* of another chain

lemma *Value-chainE*[*consumes 1*, *case-names bot B Fn*]:
assumes *chain Y*
obtains *Y* = (λ - . ⊥) |
n b **where** *Y* = (λ *m*. (*if m* < *n then* ⊥ *else B*·*b*)) |
n Y' **where** *Y* = (λ *m*. (*if m* < *n then* ⊥ *else Fn*·(*Y'* (*m*−*n*)))) *chain Y'*
proof(*cases Y* = (λ - . ⊥))
case *True*
thus *?thesis* **by** (*rule that*(*1*))
next
case *False*
hence ∃ *i*. *Y i* ≠ ⊥ **by** *auto*
hence ∃ *n*. *Y n* ≠ ⊥ ∧ (∀ *m*. *Y m* ≠ ⊥ ⟶ *m* ≥ *n*)
by (*rule exE*)(*rule ex-has-least-nat*)
then obtain *n* **where** *Y n* ≠ ⊥ **and** ∀ *m*. *m* < *n* ⟶ *Y m* = ⊥ **by** *fastforce*
hence (∃ *f*. *Y n* = *Fn*·*f*) ∨ (∃ *b*. *Y n* = *B*·*b*) **by** (*metis Value.exhaust*)
thus *?thesis*
proof
assume (∃ *f*. *Y n* = *Fn*·*f*)
then obtain *f* **where** *Y n* = *Fn* · *f* **by** *blast*
{
fix *i*
from ‹*chain Y*› **have** *Y n* ⊑ *Y* (*i*+*n*) **by** (*metis chain-mono le-add2*)

with ‹$Y\ n$ = -›
have $\exists\ g.\ (Y\ (i{+}n) = Fn \cdot g)$
by (*metis Value.dist-les*(*1*) *Value.exhaust below-bottom-iff*)
}
then obtain Y' **where** Y': $\bigwedge\ i.\ Y\ (i + n) = Fn \cdot (Y'\ i)$ **by** *metis*

have $Y = (\lambda m.\ \mathit{if}\ m < n\ \mathit{then}\ \bot\ \mathit{else}\ Fn{\cdot}(Y'\ (m - n)))$
using ‹$\forall\, m.$ -› Y' **by** (*metis add-diff-inverse add.commute*)
moreover
have*chain* Y' **using** ‹*chain* Y›
by (*auto intro*!:*chainI elim*: *chainE simp add*: *Value.inverts*[*symmetric*] Y'[*symmetric*] *simp del*: *Value.inverts*)
ultimately
show *?thesis* **by** (*rule that*(*3*))
next
assume $(\exists\ b.\ Y\ n = B{\cdot}b)$
then obtain b **where** $Y\ n = B{\cdot}b$ **by** *blast*
{
fix i
from ‹*chain* Y› **have** $Y\ n \sqsubseteq Y\ (i{+}n)$ **by** (*metis chain-mono le-add2*)
with ‹$Y\ n$ = -›
have $Y\ (i{+}n) = B{\cdot}b$
by (*metis Value.dist-les*(*2*) *Value.exhaust Value.inverts*(*2*) *below-bottom-iff discrete-cpo*)
}
hence Y': $\bigwedge\ i.\ Y\ (i + n) = B{\cdot}b$ **by** *metis*

have $Y = (\lambda m.\ \mathit{if}\ m < n\ \mathit{then}\ \bot\ \mathit{else}\ B{\cdot}b)$
using ‹$\forall\, m.$ -› Y' **by** (*metis add-diff-inverse add.commute*)
thus *?thesis* **by** (*rule that*(*2*))
qed
qed

end

22 Value-Nominal.tex

theory *Value−Nominal*
imports *Value Nominal−Utils Nominal−HOLCF*
begin

Values are pure, i.e. contain no variables.

instantiation *Value* :: *pure*
begin
definition $p \bullet (v{::}\mathit{Value}) = v$
instance
apply *standard*

apply (*auto simp add*: *permute-Value-def*)
done
end

instance *Value* :: *pcpo-pt*
by *standard* (*simp add*: *pure-permute-id*)

end

23 Denotational.tex

theory *Denotational*
imports *Abstract−Denotational−Props Value−Nominal*
begin

This is the actual denotational semantics as found in [Lau93].

interpretation *semantic-domain Fn Fn-project B B-project* (Λ *x*. *x*).

abbreviation
ESem-syn′′ :: *exp* ⇒ (*var* => *Value*) ⇒ *Value* (⟦ - ⟧- [*60*,*60*] *60*)
where $[\![\ e\ ]\!]_{\varrho} \equiv ESem\ e \cdot \varrho$

abbreviation *EvalHeapSem-syn′′* (⟦ - ⟧- [*0*,*0*] *110*) **where** $[\![\Gamma]\!]_{\varrho} \equiv evalHeap\ \Gamma\ (\lambda\ e.\ [\![e]\!]_{\varrho})$
abbreviation *HSem-syn′* (⦃-⦄- [*60*,*60*] *60*) **where** $\{\!|\Gamma|\!\}\varrho \equiv HSem\ \Gamma \cdot \varrho$
abbreviation *HSem-bot* (⦃-⦄ [*60*] *60*) **where** $\{\!|\Gamma|\!\} \equiv \{\!|\Gamma|\!\}\bot$

lemma *ESem-simps-as-defined*:
$[\![\ Lam\ [x].\ e\ ]\!]_{\varrho} = Fn\cdot(\Lambda\ v.\ [\![\ e\ ]\!]_{(\varrho\ f|`\ (fv\ (Lam\ [x].e)))(x := v)})$
$[\![\ App\ e\ x\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho} \downarrow Fn\ \varrho\ x$
$[\![\ Var\ x\ ]\!]_{\varrho} = \varrho\ x$
$[\![\ Bool\ b\ ]\!]_{\varrho} = B\cdot(Discr\ b)$
$[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_{\varrho} = B\text{-}project\cdot([\![\ scrut\ ]\!]_{\varrho})\cdot([\![\ e_1\ ]\!]_{\varrho})\cdot([\![\ e_2\ ]\!]_{\varrho})$
$[\![\ Let\ \Gamma\ body\ ]\!]_{\varrho} = [\![body]\!]_{\{\!|\Gamma|\!\}(\varrho\ f|`\ fv\ (Let\ \Gamma\ body))}$
by (*simp-all del*: *ESem-Lam ESem-Let add*: *ESem.simps*(*1*,*4*))

lemma *ESem-simps*:
$[\![\ Lam\ [x].\ e\ ]\!]_{\varrho} = Fn\cdot(\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho(x := v)})$
$[\![\ App\ e\ x\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho} \downarrow Fn\ \varrho\ x$
$[\![\ Var\ x\ ]\!]_{\varrho} = \varrho\ x$
$[\![\ Bool\ b\ ]\!]_{\varrho} = B\cdot(Discr\ b)$
$[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_{\varrho} = B\text{-}project\cdot([\![\ scrut\ ]\!]_{\varrho})\cdot([\![\ e_1\ ]\!]_{\varrho})\cdot([\![\ e_2\ ]\!]_{\varrho})$
$[\![\ Let\ \Gamma\ body\ ]\!]_{\varrho} = [\![body]\!]_{\{\!|\Gamma|\!\}\varrho}$
by *simp-all*

end

24 Launchbury.tex

theory *Launchbury*
imports *Terms Substitution*
begin

24.1 The natural semantics

This is the semantics as in [Lau93], with two differences:

- Explicit freshness requirements for bound variables in the application and the Let rule.
- An additional parameter that stores variables that have to be avoided, but do not occur in the judgement otherwise, follwing [Ses97].

inductive
  $reds :: heap \Rightarrow exp \Rightarrow var\ list \Rightarrow heap \Rightarrow exp \Rightarrow bool$
  $(- : - \Downarrow_{-}\ - : -\ [50,50,50,50]\ 50)$
where
  $Lambda$:
    $\Gamma : (Lam\ [x].\ e) \Downarrow_L \Gamma : (Lam\ [x].\ e)$
  $|\ Application$: $[\![$
    $atom\ y \sharp (\Gamma,e,x,L,\Delta,\Theta,z)$;
    $\Gamma : e \Downarrow_L \Delta : (Lam\ [y].\ e')$;
    $\Delta : e'[y ::= x] \Downarrow_L \Theta : z$
  $]\!] \Longrightarrow$
    $\Gamma : App\ e\ x \Downarrow_L \Theta : z$
  $|\ Variable$: $[\![$
    $map\text{-}of\ \Gamma\ x = Some\ e$; $delete\ x\ \Gamma : e \Downarrow_{x\#L} \Delta : z$
  $]\!] \Longrightarrow$
    $\Gamma : Var\ x \Downarrow_L (x,\ z)\ \#\ \Delta : z$
  $|\ Let$: $[\![$
    $atom\ `\ domA\ \Delta \sharp{*} (\Gamma,\ L)$;
    $\Delta\ @\ \Gamma : body \Downarrow_L \Theta : z$
  $]\!] \Longrightarrow$
    $\Gamma : Let\ \Delta\ body \Downarrow_L \Theta : z$
  $|\ Bool$:
    $\Gamma : Bool\ b \Downarrow_L \Gamma : Bool\ b$
  $|\ IfThenElse$: $[\![$
    $\Gamma : scrut \Downarrow_L \Delta : (Bool\ b)$;
    $\Delta : (if\ b\ then\ e_1\ else\ e_2) \Downarrow_L \Theta : z$
  $]\!] \Longrightarrow$
    $\Gamma : (scrut\ ?\ e_1 : e_2) \Downarrow_L \Theta : z$

equivariance *reds*

nominal-inductive *reds*
  avoids *Application*: *y*
  by (*auto simp add*: *fresh-star-def fresh-Pair*)

24.2 Example evaluations

lemma *eval-test*:
$[] : (\mathit{Let}\ [(x,\ \mathit{Lam}\ [y].\ \mathit{Var}\ y)]\ (\mathit{Var}\ x)) \Downarrow_{[]} [(x,\ \mathit{Lam}\ [y].\ \mathit{Var}\ y)] : (\mathit{Lam}\ [y].\ \mathit{Var}\ y)$
apply(*auto intro*!: *Lambda Application Variable Let*
simp add: *fresh-Pair fresh-Cons fresh-Nil fresh-star-def*)
done

lemma *eval-test2*:
$y \neq x \Longrightarrow n \neq y \Longrightarrow n \neq x \Longrightarrow [] : (\mathit{Let}\ [(x,\ \mathit{Lam}\ [y].\ \mathit{Var}\ y)]\ (\mathit{App}\ (\mathit{Var}\ x)\ x)) \Downarrow_{[]} [(x,\ \mathit{Lam}\ [y].\ \mathit{Var}\ y)] : (\mathit{Lam}\ [y].\ \mathit{Var}\ y)$
by (*auto intro*!: *Lambda Application Variable Let simp add*: *fresh-Pair fresh-at-base fresh-Cons fresh-Nil fresh-star-def pure-fresh*)

24.3 Better introduction rules

This variant do not require freshness.

lemma *reds-ApplicationI*:
assumes $\Gamma : e \Downarrow_L \Delta : \mathit{Lam}\ [y].\ e'$
assumes $\Delta : e'[y{::=}x] \Downarrow_L \Theta : z$
shows $\Gamma : \mathit{App}\ e\ x \Downarrow_L \Theta : z$
proof−
obtain $y' :: \mathit{var}$ **where** $\mathit{atom}\ y' \sharp (\Gamma,\ e,\ x,\ L,\ \Delta,\ \Theta,\ z,\ e')$ **by** (*rule obtain-fresh*)

have *a*: $\mathit{Lam}\ [y'].\ ((y' \leftrightarrow y) \cdot e') = \mathit{Lam}\ [y].\ e'$
using ‹$\mathit{atom}\ y' \sharp$ -›
by (*auto simp add*: *Abs1-eq-iff fresh-Pair fresh-at-base*)

have *b*: $((y' \leftrightarrow y) \cdot e')[y'{::=}x] = e'[y{::=}x]$
proof(*cases* $x = y$)
case *True*
have $\mathit{atom}\ y' \sharp e'$ **using** ‹$\mathit{atom}\ y' \sharp$ -› **by** *simp*
thus *?thesis*
by (*simp add*: *True subst-swap-same subst-subst-back*)
next
case *False*
hence $\mathit{atom}\ y \sharp x$ **by** *simp*

have [*simp*]: $(y' \leftrightarrow y) \cdot x = x$ **using** ‹$\mathit{atom}\ y \sharp$ -› ‹$\mathit{atom}\ y' \sharp$ -›
by (*simp add*: *flip-fresh-fresh fresh-Pair fresh-at-base*)

have $((y' \leftrightarrow y) \cdot e')[y'{::=}x] = (y' \leftrightarrow y) \cdot (e'[y{::=}x])$ **by** *simp*
also have $\ldots = e'[y{::=}x]$
using ‹$\mathit{atom}\ y \sharp$ -› ‹$\mathit{atom}\ y' \sharp$ -›
by (*simp add*: *flip-fresh-fresh fresh-Pair fresh-at-base subst-pres-fresh*)
finally
show *?thesis*.
qed
have $\mathit{atom}\ y' \sharp (\Gamma,\ e,\ x,\ L,\ \Delta,\ \Theta,\ z)$ **using** ‹$\mathit{atom}\ y' \sharp$ -› **by** (*simp add*: *fresh-Pair*)

from *this assms*[*folded a b*]
show *?thesis* **..**
qed

lemma *reds-SmartLet*: $\llbracket$
$atom$ ‘ $domA\ \Delta \sharp{*} (\Gamma, L)$;
$\Delta @ \Gamma : body \Downarrow_L \Theta : z$
$\rrbracket \Longrightarrow$
$\Gamma : SmartLet\ \Delta\ body \Downarrow_L \Theta : z$
unfolding *SmartLet-def*
by (*auto intro*: *reds.Let*)

A single rule for values

lemma *reds-isValI*:
$isVal\ z \Longrightarrow \Gamma : z \Downarrow_L \Gamma : z$
by (*cases z rule*:*isVal.cases*) (*auto intro*: *reds.intros*)

24.4 Properties of the semantics

Heap entries are never removed.

lemma *reds-doesnt-forget*:
$\Gamma : e \Downarrow_L \Delta : z \Longrightarrow domA\ \Gamma \subseteq domA\ \Delta$
by(*induct rule*: *reds.induct*) *auto*

Live variables are not added to the heap.

lemma *reds-avoids-live'*:
assumes $\Gamma : e \Downarrow_L \Delta : z$
shows $(domA\ \Delta - domA\ \Gamma) \cap set\ L = \{\}$
using *assms*
by(*induct rule*:*reds.induct*)
(*auto dest*: *map-of-domAD fresh-distinct-list simp add*: *fresh-star-Pair*)

lemma *reds-avoids-live*:
$\llbracket \Gamma : e \Downarrow_L \Delta : z$;
$x \in set\ L$;
$x \notin domA\ \Gamma$
$\rrbracket \Longrightarrow x \notin domA\ \Delta$
using *reds-avoids-live'* **by** *blast*

Fresh variables either stay fresh or are added to the heap.

lemma *reds-fresh*: $\llbracket \Gamma : e \Downarrow_L \Delta : z$;
$atom\ (x{::}var) \sharp (\Gamma, e)$
$\rrbracket \Longrightarrow atom\ x \sharp (\Delta, z) \vee x \in (domA\ \Delta - set\ L)$
proof(*induct rule*: *reds.induct*)
case (*Lambda* Γ *x e*) **thus** *?case* **by** *auto*
next
case (*Application* $y\ \Gamma\ e\ x'\ L\ \Delta\ \Theta\ z\ e'$)

hence *atom x* ♯ (Δ, *Lam* [*y*]. *e'*) ∨ *x* ∈ *domA* Δ − *set* (*x'* # *L*) **by** (*auto simp add*: *fresh-Pair*)

thus *?case*
proof
assume *atom x* ♯ (Δ, *Lam* [*y*]. *e'*)
hence *atom x* ♯ *e'*[*y* ::= *x'*]
using *Application.prems*
by (*auto intro*: *subst-pres-fresh simp add*: *fresh-Pair*)
thus *?thesis* **using** *Application.hyps*(*5*) ‹*atom x* ♯ (Δ, *Lam* [*y*]. *e'*)› **by** *auto*
next
assume *x* ∈ *domA* Δ − *set* (*x'* # *L*)
thus *?thesis* **using** *reds-doesnt-forget*[*OF Application.hyps*(*4*)] **by** *auto*
qed
next

case(*Variable* Γ *v e L* Δ *z*)
have *atom x* ♯ Γ **and** *atom x* ♯ *v* **using** *Variable.prems*(*1*) **by** (*auto simp add*: *fresh-Pair*)
from *fresh-delete*[*OF this*(*1*)]
have *atom x* ♯ *delete v* Γ.
moreover
have *v* ∈ *domA* Γ **using** *Variable.hyps*(*1*) **by** (*metis domA-from-set map-of-SomeD*)
from *fresh-map-of*[*OF this* ‹*atom x* ♯ Γ›]
have *atom x* ♯ *the* (*map-of* Γ *v*).
hence *atom x* ♯ *e* **using** ‹*map-of* Γ *v* = *Some e*› **by** *simp*
ultimately
have *atom x* ♯ (Δ, *z*) ∨ *x* ∈ *domA* Δ − *set* (*v* # *L*) **using** *Variable.hyps*(*3*) **by** (*auto simp add*: *fresh-Pair*)
thus *?case* **using** ‹*atom x* ♯ *v*› **by** (*auto simp add*: *fresh-Pair fresh-Cons fresh-at-base*)
next

case (*Bool* Γ *b L*)
thus *?case* **by** *auto*
next

case (*IfThenElse* Γ *scrut L* Δ *b* e_1 e_2 Θ *z*)
from ‹*atom x* ♯ (Γ, *scrut* ? e_1 : e_2)›
have *atom x* ♯ (Γ, *scrut*) **and** *atom x* ♯ (e_1, e_2) **by** (*auto simp add*: *fresh-Pair*)
from *IfThenElse.hyps*(*2*)[*OF this*(*1*)]
show *?case*
proof
assume *atom x* ♯ (Δ, *Bool b*) **with** ‹*atom x* ♯ (e_1, e_2)›
have *atom x* ♯ (Δ, *if b then* e_1 *else* e_2) **by** *auto*
from *IfThenElse.hyps*(*4*)[*OF this*]
show *?thesis*.
next
assume *x* ∈ *domA* Δ − *set L*
with *reds-doesnt-forget*[*OF* ‹Δ : (*if b then* e_1 *else* e_2) $\Downarrow_L$ Θ : *z*›]
show *?thesis* **by** *auto*

  qed
next

case ($Let\ \Delta\ \Gamma\ L\ body\ \Theta\ z$)
  show *?case*
  proof ($cases\ x \in domA\ \Delta$)
    case *False*
      hence $atom\ x \sharp \Delta$ **using** *Let.prems* **by**(*auto simp add*: *fresh-Pair*)
      show *?thesis*
        apply(*rule Let.hyps*(*3*))
        using *Let.prems* ‹$atom\ x \sharp \Delta$› *False*
        by (*auto simp add*: *fresh-Pair fresh-append*)
  next
    case *True*
      hence $x \notin set\ L$
        using *Let*(*1*)
        by (*metis fresh-PairD*(*2*) *fresh-star-def image-eqI set-not-fresh*)
      with *True*
      show *?thesis*
      using *reds-doesnt-forget*[*OF Let.hyps*(*2*)] **by** *auto*
  qed
qed

lemma *reds-fresh-fv*: $\llbracket\ \Gamma : e \Downarrow_L \Delta : z;$
   $x \in fv\ (\Delta, z) \wedge (x \notin domA\ \Delta \vee\ x \in set\ L)$
  $\rrbracket \Longrightarrow x \in fv\ (\Gamma, e)$
using *reds-fresh*
unfolding *fv-def fresh-def*
by *blast*

lemma *new-free-vars-on-heap*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  shows $fv\ (\Delta, z) - domA\ \Delta \subseteq fv\ (\Gamma, e) - domA\ \Gamma$
using *reds-fresh-fv*[*OF assms*(*1*)] *reds-doesnt-forget*[*OF assms*(*1*)] **by** *auto*

lemma *reds-pres-closed*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  and $fv\ (\Gamma, e) \subseteq set\ L \cup domA\ \Gamma$
  shows $fv\ (\Delta, z) \subseteq set\ L \cup domA\ \Delta$
using *new-free-vars-on-heap*[*OF assms*(*1*)] *assms*(*2*) **by** *auto*

Reducing the set of variables to avoid is always possible.

lemma *reds-smaller-L*: $\llbracket\ \Gamma : e \Downarrow_L \Delta : z;$
   $set\ L' \subseteq set\ L$
  $\rrbracket \Longrightarrow \Gamma : e \Downarrow_{L'} \Delta : z$
proof(*nominal-induct avoiding* : L' *rule*: *reds.strong-induct*)
case ($Lambda\ \Gamma\ x\ e\ L\ L'$)
  show *?case*
    by (*rule reds.Lambda*)

next
case $(Application\ y\ \Gamma\ e\ xa\ L\ \Delta\ \Theta\ z\ e'\ L')$
 from $Application.hyps(10)[OF\ Application.prems]$ $Application.hyps(12)[OF\ Application.prems]$
  show *?case*
   by (*rule reds-ApplicationI*)
next
case $(Variable\ \Gamma\ xa\ e\ L\ \Delta\ z\ L')$
  have $set\ (xa\ \#\ L') \subseteq set\ (xa\ \#\ L)$
   using *Variable.prems* **by** *auto*
  thus *?case*
   by $(rule\ reds.Variable[OF\ Variable(1)\ Variable.hyps(3)])$
next
case $(Bool\ b)$
  show *?case*..
next
case $(IfThenElse\ \Gamma\ scrut\ L\ \Delta\ b\ e_1\ e_2\ \Theta\ z\ L')$
  thus *?case* **by** (*metis reds.IfThenElse*)
next
case $(Let\ \Delta\ \Gamma\ L\ body\ \Theta\ z\ L')$
  have $atom\ `\ domA\ \Delta\ \sharp{*}\ (\Gamma,\ L')$
   using $Let(1-3)$ *Let.prems*
   by (*auto simp add*: *fresh-star-Pair fresh-star-set-subset*)
  thus *?case*
   by $(rule\ reds.Let[OF\ \text{-}\ Let.hyps(4)[OF\ Let.prems]])$
qed

Things are evaluated to a lambda expression, and the variable can be freely chose.

lemma *result-evaluated*:
 $\Gamma : e \Downarrow_L \Delta : z \Longrightarrow isVal\ z$
by $(induct\ \Gamma\ e\ L\ \Delta\ z\ rule{:}reds.induct)$ (*auto dest*: *reds-doesnt-forget*)

lemma *result-evaluated-fresh*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  obtains $y\ e'$
  where $z = (Lam\ [y].\ e')$ **and** $atom\ y\ \sharp\ (c{::}'a{::}fs)$ | b **where** $z = Bool\ b$
proof−
  from *assms*
  have *isVal z* **by** (*rule result-evaluated*)
  hence $(\exists\ y\ e'.\ z = Lam\ [y].\ e' \wedge atom\ y\ \sharp\ c) \vee (\exists\ b.\ z = Bool\ b)$
   by (*nominal-induct z avoiding*: *c rule*:*exp-strong-induct*) *auto*
  thus *thesis* **using** *that* **by** *blast*
qed

end

25 CorrectnessOriginal.tex

theory *CorrectnessOriginal*
imports *Denotational Launchbury*
begin

This is the main correctness theorem, Theorem 2 from [Lau93].

theorem *correctness*:
  assumes $\Gamma : e \Downarrow_{L} \Delta : v$
  and $\mathit{fv}\ (\Gamma,\ e) \subseteq \mathit{set}\ L \cup \mathit{domA}\ \Gamma$
  shows $[\![e]\!]_{\{\!|\Gamma|\!\}\varrho} = [\![v]\!]_{\{\!|\Delta|\!\}\varrho}$
  and $(\{\!|\Gamma|\!\}\varrho)\ f|`\ \mathit{domA}\ \Gamma = (\{\!|\Delta|\!\}\varrho)\ f|`\ \mathit{domA}\ \Gamma$
  using *assms*
proof(*nominal-induct arbitrary*: ϱ *rule*:*reds.strong-induct*)
case *Lambda*
  case *1* **show** *?case*..
  case *2* **show** *?case*..
next
case $(\mathit{Application}\ y\ \Gamma\ e\ x\ L\ \Delta\ \Theta\ v\ e')$
  have *Gamma-subset*: $\mathit{domA}\ \Gamma \subseteq \mathit{domA}\ \Delta$
    by (*rule reds-doesnt-forget*[*OF Application.hyps*(*8*)])

  case *1*
  hence *prem1*: $\mathit{fv}\ (\Gamma,\ e) \subseteq \mathit{set}\ L \cup \mathit{domA}\ \Gamma$ **and** $x \in \mathit{set}\ L \cup \mathit{domA}\ \Gamma$ **by** *auto*
  moreover
  note *reds-pres-closed*[*OF Application.hyps*(*8*) *prem1*]
  moreover
  note *reds-doesnt-forget*[*OF Application.hyps*(*8*)]
  moreover
  have $\mathit{fv}\ (e'[y{::=}x]) \subseteq \mathit{fv}\ (\mathit{Lam}\ [y].\ e') \cup \{x\}$
    by (*auto simp add*: *fv-subst-eq*)
  ultimately
  have *prem2*: $\mathit{fv}\ (\Delta,\ e'[y{::=}x]) \subseteq \mathit{set}\ L \cup \mathit{domA}\ \Delta$ **by** *auto*

  have $*$: $(\{\!|\Gamma|\!\}\varrho)\ x = (\{\!|\Delta|\!\}\varrho)\ x$
  proof(*cases* $x \in \mathit{domA}\ \Gamma$)
    case *True*
    from *Application.hyps*(*10*)[*OF prem1*, **where** $\varrho = \varrho$]
    have $((\{\!|\Gamma|\!\}\varrho)\ f|`\ \mathit{domA}\ \Gamma)\ x = ((\{\!|\Delta|\!\}\varrho)\ f|`\ \mathit{domA}\ \Gamma)\ x$ **by** *simp*
    with *True* **show** *?thesis* **by** *simp*
  next
    case *False*
    from *False* ‹$x \in \mathit{set}\ L \cup \mathit{domA}\ \Gamma$› *reds-avoids-live*[*OF Application.hyps*(*8*)]
    show *?thesis* **by** (*auto simp add*: *lookup-HSem-other*)
  qed **have** $[\![\ \mathit{App}\ e\ x\ ]\!]_{\{\!|\Gamma|\!\}\varrho} = ([\![\ e\ ]\!]_{\{\!|\Gamma|\!\}\varrho})\ \downarrow\mathit{Fn}\ (\{\!|\Gamma|\!\}\varrho)\ x$
    by *simp*
  also have $\ldots = ([\![\ \mathit{Lam}\ [y].\ e'\ ]\!]_{\{\!|\Delta|\!\}\varrho})\ \downarrow\mathit{Fn}\ (\{\!|\Gamma|\!\}\varrho)\ x$
    using *Application.hyps*(*9*)[*OF prem1*] **by** *simp*

also have ... $= (\llbracket\ \mathit{Lam}\ [y].\ e'\ \rrbracket_{\{\!\!\{\Delta\}\!\!\}\varrho})\ \downarrow\!\mathit{Fn}\ (\{\!\!\{\Delta\}\!\!\}\varrho)\ x$
 unfolding $*$..
also have ... $= (\mathit{Fn}\cdot(\Lambda\ z.\ \llbracket\ e'\ \rrbracket_{(\{\!\!\{\Delta\}\!\!\}\varrho)(y\ :=\ z)}))\ \downarrow\!\mathit{Fn}\ (\{\!\!\{\Delta\}\!\!\}\varrho)\ x$
 by *simp*
also have ... $= \llbracket\ e'\ \rrbracket_{(\{\!\!\{\Delta\}\!\!\}\varrho)(y\ :=\ (\{\!\!\{\Delta\}\!\!\}\varrho)\ x)}$
 by *simp*
also have ... $= \llbracket\ e'[y\ ::=\ x]\ \rrbracket_{\{\!\!\{\Delta\}\!\!\}\varrho}$
 unfolding *ESem-subst*..
also have ... $= \llbracket\ v\ \rrbracket_{\{\!\!\{\Theta\}\!\!\}\varrho}$
 by (*rule Application.hyps*(*12*)[*OF prem2*])
finally
show $\llbracket\ \mathit{App}\ e\ x\ \rrbracket_{\{\!\!\{\Gamma\}\!\!\}\varrho} = \llbracket\ v\ \rrbracket_{\{\!\!\{\Theta\}\!\!\}\varrho}$. **show** $(\{\!\!\{\Gamma\}\!\!\}\varrho)\ f|`\ \mathit{domA}\ \Gamma = (\{\!\!\{\Theta\}\!\!\}\varrho)\ f|`\ \mathit{domA}\ \Gamma$
 using *Application.hyps*(*10*)[*OF prem1*]
  env-restr-eq-subset[*OF Gamma-subset Application.hyps*(*13*)[*OF prem2*]]
 by (*rule trans*)
next
case (*Variable* Γ *x e L* Δ *v*)
hence [*simp*]:$x \in \mathit{domA}\ \Gamma$ **by** (*metis domA-from-set map-of-SomeD*)

let $?\Gamma = \mathit{delete}\ x\ \Gamma$

case *2*
have $x \notin \mathit{domA}\ \Delta$
 by (*rule reds-avoids-live*[*OF Variable.hyps*(*2*)], *simp-all*)

have *subset*: $\mathit{domA}\ ?\Gamma \subseteq \mathit{domA}\ \Delta$
 by (*rule reds-doesnt-forget*[*OF Variable.hyps*(*2*)])

let $?\mathit{new} = \mathit{domA}\ \Delta - \mathit{domA}\ \Gamma$
have $\mathit{fv}\ (?\Gamma,\ e) \cup \{x\} \subseteq \mathit{fv}\ (\Gamma,\ \mathit{Var}\ x)$
 by (*rule fv-delete-heap*[*OF* ‹*map-of* Γ *x = Some e*›])
hence *prem*: $\mathit{fv}\ (?\Gamma,\ e) \subseteq \mathit{set}\ (x\ \#\ L) \cup \mathit{domA}\ ?\Gamma$ **using** *2* **by** *auto*
hence *fv-subset*: $\mathit{fv}\ (?\Gamma,\ e) - \mathit{domA}\ ?\Gamma \subseteq -\ ?\mathit{new}$
 using *reds-avoids-live'*[*OF Variable.hyps*(*2*)] **by** *auto*

have $\mathit{domA}\ \Gamma \subseteq (-?\mathit{new})$ **by** *auto*

have $\{\!\!\{\Gamma\}\!\!\}\varrho = \{\!\!\{(x,e)\ \#\ ?\Gamma\}\!\!\}\varrho$
 by (*rule HSem-reorder*[*OF map-of-delete-insert*[*symmetric*, *OF Variable*(*1*)]])
also have ... $= (\mu\ \varrho'.\ (\varrho\ {+}{+}_{(\mathit{domA}\ ?\Gamma)}\ (\{\!\!\{?\Gamma\}\!\!\}\varrho'))(\ x := \llbracket\ e\ \rrbracket_{\varrho'}))$
 by (*rule iterative-HSem*, *simp*)
also have ... $= (\mu\ \varrho'.\ (\varrho\ {+}{+}_{(\mathit{domA}\ ?\Gamma)}\ (\{\!\!\{?\Gamma\}\!\!\}\varrho'))(\ x := \llbracket\ e\ \rrbracket_{\{\!\!\{?\Gamma\}\!\!\}\varrho'}))$
 by (*rule iterative-HSem'*, *simp*)
finally
have $(\{\!\!\{\Gamma\}\!\!\}\varrho)f|`\ (-\ ?\mathit{new}) = (...)\ f|`\ (-\ ?\mathit{new})$ **by** *simp*
also have ... $= (\mu\ \varrho'.\ (\varrho\ {+}{+}_{\mathit{domA}\ \Delta}\ (\{\!\!\{\Delta\}\!\!\}\varrho'))(\ x := \llbracket\ v\ \rrbracket_{\{\!\!\{\Delta\}\!\!\}\varrho'}))\ f|`\ (-\ ?\mathit{new})$
proof (*induction rule*: *parallel-fix-ind*[**where** $P = \lambda\ x\ y.\ x\ f|`\ (-\ ?\mathit{new}) = y\ f|`\ (-\ ?\mathit{new})$])
 case *1* **show** *?case* **by** *simp*

next
  case *2* **show** *?case* **..**
next
  case (*3* σ σ')
  hence $[\![\ e\ ]\!]_{\{\!|?\Gamma|\!\}\sigma} = [\![\ e\ ]\!]_{\{\!|?\Gamma|\!\}\sigma'}$
    and $(\{\!|?\Gamma|\!\}\sigma)\ f|`\ domA\ ?\Gamma = (\{\!|?\Gamma|\!\}\sigma')\ f|`\ domA\ ?\Gamma$
    using *fv-subset* **by** (*auto intro*: *ESem-fresh-cong HSem-fresh-cong env-restr-eq-subset*[*OF*
- *3*])
  from *trans*[*OF this*(*1*) *Variable*(*3*)[*OF prem*]] *trans*[*OF this*(*2*) *Variable*(*4*)[*OF prem*]]
  have $[\![\ e\ ]\!]_{\{\!|?\Gamma|\!\}\sigma} = [\![\ v\ ]\!]_{\{\!|\Delta|\!\}\sigma'}$
    and $(\{\!|?\Gamma|\!\}\sigma)\ f|`\ domA\ ?\Gamma = (\{\!|\Delta|\!\}\sigma')\ f|`\ domA\ ?\Gamma$**.**
  thus *?case*
    using *subset*
    by (*fastforce simp add*: *lookup-override-on-eq lookup-env-restr-eq dest*: *env-restr-eqD*)
qed
also have $\ldots = (\mu\ \varrho'.\ (\varrho\ {+}{+}_{domA\ \Delta}\ (\{\!|\Delta|\!\}\varrho'))(\ x := [\![\ v\ ]\!]_{\varrho'}))\ f|`\ (-?new)$
  by (*rule arg-cong*[*OF iterative-HSem'*[*symmetric*], *OF* ‹$x \notin domA\ \Delta$›])
also have $\ldots = (\{\!|(x,v)\ \#\ \Delta|\!\}\varrho)\ \ f|`\ (-?new)$
  by (*rule arg-cong*[*OF iterative-HSem*[*symmetric*], *OF* ‹$x \notin domA\ \Delta$›])
finally
show *le*: *?case* **by** (*rule env-restr-eq-subset*[*OF* ‹$domA\ \Gamma \subseteq (-?new)$›])

have $[\![\ Var\ x\ ]\!]_{\{\!|\Gamma|\!\}\varrho} = [\![\ Var\ x\ ]\!]_{\{\!|(x,\ v)\ \#\ \Delta|\!\}\varrho}$
  using *env-restr-eqD*[*OF le*, **where** $x = x$]
  by *simp*
also have $\ldots = [\![\ v\ ]\!]_{\{\!|(x,\ v)\ \#\ \Delta|\!\}\varrho}$
  by (*auto simp add*: *lookup-HSem-heap*)
finally
show $[\![\ Var\ x\ ]\!]_{\{\!|\Gamma|\!\}\varrho} = [\![\ v\ ]\!]_{\{\!|(x,\ v)\ \#\ \Delta|\!\}\varrho}$**.**
next
case (*Bool b*)
  case *1*
  show *?case* **by** *simp*
  case *2*
  show *?case* **by** *simp*
next
case (*IfThenElse* Γ *scrut L* Δ *b* e_1 e_2 Θ *v*)
  have *Gamma-subset*: $domA\ \Gamma \subseteq domA\ \Delta$
    by (*rule reds-doesnt-forget*[*OF IfThenElse.hyps*(*1*)])

  let $?e = if\ b\ then\ e_1\ else\ e_2$

  case *1*
  thm *new-free-vars-on-heap*[*OF IfThenElse.hyps*(*1*)]

  hence *prem1*: $fv\ (\Gamma,\ scrut) \subseteq set\ L \cup domA\ \Gamma$
    and *prem2*: $fv\ (\Delta,\ ?e) \subseteq set\ L \cup domA\ \Delta$
    and $fv\ ?e \subseteq domA\ \Gamma \cup set\ L$

```
    using new-free-vars-on-heap[OF IfThenElse.hyps(1)] Gamma-subset by auto

  have ⟦ (scrut ? e₁ : e₂) ⟧_{⦃Γ⦄ϱ} = B-project·(⟦ scrut ⟧_{⦃Γ⦄ϱ})·(⟦ e₁ ⟧_{⦃Γ⦄ϱ})·(⟦ e₂ ⟧_{⦃Γ⦄ϱ}) by simp
  also have ... = B-project·(⟦ Bool b ⟧_{⦃Δ⦄ϱ})·(⟦ e₁ ⟧_{⦃Γ⦄ϱ})·(⟦ e₂ ⟧_{⦃Γ⦄ϱ})
    unfolding IfThenElse.hyps(2)[OF prem1]..
  also have ... = ⟦ ?e ⟧_{⦃Γ⦄ϱ} by simp
  also have ... = ⟦ ?e ⟧_{⦃Δ⦄ϱ}
    proof(rule ESem-fresh-cong-subset[OF ‹fv ?e ⊆ domA Γ ∪ set L› env-restr-eqI])
      fix x
      assume x ∈ domA Γ ∪ set L
      thus (⦃Γ⦄ϱ) x = (⦃Δ⦄ϱ) x
      proof(cases x ∈ domA Γ)
        assume x ∈ domA Γ
        from IfThenElse.hyps(3)[OF prem1]
        have ((⦃Γ⦄ϱ) f|` domA Γ) x = ((⦃Δ⦄ϱ) f|` domA Γ) x by simp
        with ‹x ∈ domA Γ› show ?thesis by simp
      next
        assume x ∉ domA Γ
        from this ‹x ∈ domA Γ ∪ set L› reds-avoids-live[OF IfThenElse.hyps(1)]
        show ?thesis
          by (simp add: lookup-HSem-other)
      qed
    qed
  also have ... = ⟦ v ⟧_{⦃Θ⦄ϱ}
    unfolding IfThenElse.hyps(5)[OF prem2]..
  finally
  show ?case.
  thm env-restr-eq-subset
  show (⦃Γ⦄ϱ) f|` domA Γ = (⦃Θ⦄ϱ) f|` domA Γ
    using IfThenElse.hyps(3)[OF prem1]
          env-restr-eq-subset[OF Gamma-subset IfThenElse.hyps(6)[OF prem2]]
    by (rule trans)
next
case (Let as Γ L body Δ v)
  case 1
  { fix a
    assume a: a ∈ domA  as
    have atom a ♯ Γ
      by (rule Let(1)[unfolded fresh-star-def, rule-format, OF imageI[OF a]])
    hence a ∉ domA Γ
      by (metis domA-not-fresh)
  }
  note * = this


  have fv (as @ Γ, body) − domA (as @ Γ) ⊆  fv (Γ, Let as body) − domA Γ
    by auto
  with 1 have prem: fv (as @ Γ, body) ⊆ set L ∪ domA (as @ Γ) by auto
```

have *f1*: *atom* ' *domA as* $\sharp*$ Γ
using *Let*(*1*) **by** (*simp add*: *set-bn-to-atom-domA*)

have $[\![$ *Let as body* $]\!]_{\{\!\!\{\Gamma\}\!\!\}\varrho}$ = $[\![$ *body* $]\!]_{\{\!\!\{as\}\!\!\}\{\!\!\{\Gamma\}\!\!\}\varrho}$
by (*simp*)
also have ... = $[\![$ *body* $]\!]_{\{\!\!\{as\ @\ \Gamma\}\!\!\}\varrho}$
by (*rule arg-cong*[*OF HSem-merge*[*OF f1*]])
also have ... = $[\![$ *v* $]\!]_{\{\!\!\{\Delta\}\!\!\}\varrho}$
by (*rule Let.hyps*(*4*)[*OF prem*])
finally
show *?case*.

have ($\{\!\!\{\Gamma\}\!\!\}\varrho$) *f*|' (*domA* Γ) = ($\{\!\!\{as\}\!\!\}$($\{\!\!\{\Gamma\}\!\!\}\varrho$)) *f*|' (*domA* Γ)
apply (*rule ext*)
apply (*case-tac x* $\in$ *domA as*)
apply (*auto simp add*: *lookup-HSem-other lookup-env-restr-eq* $*$)
done
also have ... = ($\{\!\!\{as\ @\ \Gamma\}\!\!\}\varrho$) *f*|' (*domA* Γ)
by (*rule arg-cong*[*OF HSem-merge*[*OF f1*]])
also have ... = ($\{\!\!\{\Delta\}\!\!\}\varrho$) *f*|' (*domA* Γ)
by (*rule env-restr-eq-subset*[*OF - Let.hyps*(*5*)[*OF prem*]]) *simp*
finally
show ($\{\!\!\{\Gamma\}\!\!\}\varrho$) *f*|' *domA* Γ = ($\{\!\!\{\Delta\}\!\!\}\varrho$) *f*|' *domA* Γ.
qed

end

26 Mono-Nat-Fun.tex

theory *Mono−Nat−Fun*
imports ~~/*src*/*HOL*/*Library*/*Infinite-Set*
begin

The following lemma proves that a monotonous function from and to the natural numbers is either eventually constant or unbounded.

lemma *nat-mono-characterization*:
fixes *f* :: *nat* $\Rightarrow$ *nat*
assumes *mono f*
obtains *n* **where** $\bigwedge m$. $n \leq m \Longrightarrow f\,n = f\,m$ | $\bigwedge$ *m* . $\exists$ *n* . $m \leq f\,n$
proof (*cases finite* (*range f*))
case *True*
from *Max-in*[*OF True*]
obtain *n* **where** *Max*: *f n* = *Max* (*range f*) **by** *auto*
show *thesis*
proof(*rule that*(*1*))
fix *m*

assume $n \leq m$
hence $f\ n \leq f\ m$ **using** ‹*mono f*› **by** (*metis monoD*)
also
have $f\ m \leq f\ n$ **unfolding** *Max* **by** (*rule Max-ge*[*OF True rangeI*])
finally
show $f\ n = f\ m$.
qed
next
case *False*
thus *thesis* **by** (*fastforce intro*: *that*(*2*) *simp add*: *infinite-nat-iff-unbounded-le*)
qed

end

27 C.tex

theory *C*
imports ~~/*src*/*HOL*/*HOLCF*/*HOLCF Mono*−*Nat*−*Fun*
begin

default-sort *cpo*

The initial solution to the domain equation $C = C_\perp$, i.e. the completion of the natural numbers.

domain $C = C$ (**lazy** C)

lemma *below-C*: $x \sqsubseteq C{\cdot}x$
by (*induct x*) *auto*

definition *Cinf* (C^∞) **where** $C^\infty = \mathit{fix}{\cdot}C$

lemma *C-Cinf*[*simp*]: $C{\cdot}C^\infty = C^\infty$ **unfolding** *Cinf-def* **by** (*rule fix-eq*[*symmetric*])

abbreviation *Cpow* (C^{-}) **where** $C^n \equiv \mathit{iterate}\ n{\cdot}C{\cdot}\bot$

lemma *C-below-C*[*simp*]: $(C^i \sqsubseteq C^j) \longleftrightarrow i \leq j$
apply (*induction i arbitrary*: *j*)
apply *simp*
apply (*case-tac j*, *auto*)
done

lemma *below-Cinf*[*simp*]: $r \sqsubseteq C^\infty$
apply (*induct r*)
apply *simp-all*[*2*]
apply (*metis* (*full-types*) *C-Cinf monofun-cfun-arg*)
done

lemma *C-eq-Cinf*[*simp*]: $C^i \neq C^\infty$
  by (*metis C-below-C Suc-n-not-le-n below-Cinf*)

lemma *Cinf-eq-C*[*simp*]: $C^\infty = C \cdot r \longleftrightarrow C^\infty = r$
  by (*metis C.injects C-Cinf*)

lemma *C-eq-C*[*simp*]: $(C^i = C^j) \longleftrightarrow i = j$
  by (*metis C-below-C le-antisym le-refl*)

lemma *case-of-C-below*: $(case\ r\ of\ C\cdot y \Rightarrow x) \sqsubseteq x$
  by (*cases r*) *auto*

lemma *C-case-below*: $C\text{-}case \cdot f \sqsubseteq f$
  by (*metis cfun-belowI C.case-rews(2) below-C monofun-cfun-arg*)

lemma *C-case-bot*[*simp*]: $C\text{-}case \cdot \bot = \bot$
  apply (*subst eq-bottom-iff*)
  apply (*rule C-case-below*)
  done

lemma *C-case-cong*:
  assumes $\bigwedge r'.\ r = C\cdot r' \Longrightarrow f\cdot r' = g\cdot r'$
  shows $C\text{-}case\cdot f\cdot r = C\text{-}case\cdot g\cdot r$
using *assms* **by** (*cases r*) *auto*

lemma *C-cases*:
  obtains *n* **where** $r = C^n \mid r = C^\infty$
proof−
  { fix *m*
    have $\exists\ n.\ C\text{-}take\ m \cdot r = C^n$
    proof (*rule C.finite-induct*)
      have $\bot = C^0$ **by** *simp*
      thus $\exists n.\ \bot = C^n$**..**
    next
      fix *r*
      show $\exists n.\ r = C^n \Longrightarrow \exists n.\ C\cdot r = C^n$
        by (*auto simp del*: *iterate-Suc simp add*: *iterate-Suc*[*symmetric*])
    qed
  }
  then obtain *f* **where** *take*: $\bigwedge m.\ C\text{-}take\ m \cdot r = C^{f\ m}$ **by** *metis*
  have *chain* $(\lambda\ m.\ C^{f\ m})$ **using** *ch2ch-Rep-cfunL*[*OF C.chain-take*, **where** *x*=*r*, *unfolded take*]**.**
  hence *mono f* **by** (*auto simp add*: *mono-iff-le-Suc chain-def elim*!:*chainE*)
  have *r*: $r = (\bigsqcup\ m.\ C^{f\ m})$ **by** (*metis* (*lifting*) *take C.reach lub-eq*)
  from ‹*mono f*›
  show *thesis*
  proof(*rule nat-mono-characterization*)
    fix *n*

assume *n*: $\bigwedge$ *m*. $n \leq m$ ==> $f\ n = f\ m$
have *max-in-chain n* (λ *m*. $C^{f\ m}$)
apply (*rule max-in-chainI*)
apply *simp*
apply (*erule n*)
done
hence ($\bigsqcup$ *m*. $C^{f\ m}$) = $C^{f\ n}$ **unfolding** *maxinch-is-thelub*[*OF* ‹*chain* -›]**.**
thus *?thesis* **using** *that* **unfolding** *r* **by** *blast*
next
assume $\bigwedge$*m*. $\exists$ *n*. $m \leq f\ n$
hence $\bigwedge$ *n*. $C^n \sqsubseteq r$ **unfolding** *r* **by** (*fastforce intro*: *below-lub*[*OF* ‹*chain* -›])
hence ($\bigsqcup$ *n*. C^n) $\sqsubseteq r$
by (*rule lub-below*[*OF chain-iterate*])
hence $C^\infty \sqsubseteq r$ **unfolding** *Cinf-def fix-def2***.**
hence $C^\infty = r$ **using** *below-Cinf* **by** (*metis below-antisym*)
thus *thesis* **using** *that* **by** *blast*
qed
qed

lemma *C-case-Cinf*[*simp*]: *C-case* · *f* · C^∞ = *f* · C^∞
unfolding *Cinf-def*
by (*subst fix-eq*) *simp*

end

28 CValue.tex

theory *CValue*
imports *C*
begin

domain *CValue*
= *CFn* (**lazy** (*C* → *CValue*) → (*C* → *CValue*))
| *CB* (**lazy** *bool discr*)

fixrec *CFn-project* :: *CValue* → (*C* → *CValue*) → (*C* → *CValue*)
where *CFn-project*·(*CFn*·*f*)·*v* = *f* · *v*

abbreviation *CFn-project-abbr* (**infix** ↓*CFn 55*)
where *f* ↓*CFn v* ≡ *CFn-project*·*f*·*v*

lemma *CFn-project-strict*[*simp*]:
⊥ ↓*CFn v* = ⊥
CB·*b* ↓*CFn v* = ⊥
by (*fixrec-simp*)+

lemma *CB-below*[*simp*]: *CB*·*b* ⊑ *v* ⟷ *v* = *CB*·*b* by $(cases\ v)\ auto$

fixrec *CB-project* :: *CValue* → *CValue* → *CValue* → *CValue* **where**
CB-project·(*CB*·*db*)·v_1·v_2 = (*if undiscr db then* v_1 *else* v_2)

lemma [*simp*]:
CB-project·(*CB*·(*Discr b*))·v_1·v_2 = (*if b then* v_1 *else* v_2)
CB-project·⊥·v_1·v_2 = ⊥
CB-project·(*CFn*·*f*)·v_1·v_2 = ⊥
by *fixrec-simp*+

lemma *CB-project-not-bot*:
CB-project·*scrut*·v_1·v_2 ≠ ⊥ ⟷ (∃ *b*. *scrut* = *CB*·(*Discr b*) ∧ (*if b then* v_1 *else* v_2) ≠ ⊥)
apply (*cases scrut*)
apply *simp*
apply *simp*
by (*metis* (*poly-guards-query*) *CB-project.simps CValue.injects*(*2*) *discr.exhaust undiscr-Discr*)

HOLCF provides us *CValue-take*::*nat* ⇒ *CValue* → *CValue*; we want a similar function for *C* → *CValue*.

abbreviation *C-to-CValue-take* :: *nat* ⇒ (*C* → *CValue*) → (*C* → *CValue*)
where *C-to-CValue-take n* ≡ *cfun-map*·*ID*·(*CValue-take n*)

lemma *C-to-CValue-chain-take*: *chain C-to-CValue-take*
by (*auto intro*: *chainI cfun-belowI chainE*[*OF CValue.chain-take*] *monofun-cfun-fun*)

lemma *C-to-CValue-reach*: (⨆ *n*. *C-to-CValue-take n*·*x*) = *x*
by (*auto intro*: *cfun-eqI simp add*: *contlub-cfun-fun*[*OF ch2ch-Rep-cfunL*[*OF C-to-CValue-chain-take*]] *CValue.reach*)

end

29 CValue-Nominal.tex

theory *CValue−Nominal*
imports *CValue Nominal−Utils Nominal−HOLCF*
begin

instantiation *C* :: *pure*
begin
definition *p* • (*c*::*C*) = *c*
instance by *standard* (*auto simp add*: *permute-C-def*)
end
instance *C* :: *pcpo-pt*
by *standard* (*simp add*: *pure-permute-id*)

instantiation *CValue* :: *pure*
begin
  definition $p \cdot (v{::}CValue) = v$
instance
  apply *standard*
  apply (*auto simp add*: *permute-CValue-def*)
  done
end

instance *CValue* :: *pcpo-pt*
  by *standard* (*simp add*: *pure-permute-id*)

end

30 HOLCF-Meet.tex

theory *HOLCF−Meet*
imports ~~/*src*/*HOL*/*HOLCF*/*HOLCF*
begin

This theory defines the $\sqcap$ operator on HOLCF domains, and introduces a type class for domains where all finite meets exist.

30.1 Towards meets: Lower bounds

context *po*
begin
definition *is-lb* :: *'a set* ⇒ *'a* ⇒ *bool* (**infix** >| *55*) **where**
  $S >| x \longleftrightarrow (\forall y{\in}S.\ x \sqsubseteq y)$

lemma *is-lbI*: $(!!x.\ x \in S ==> l \sqsubseteq x) ==> S >| l$
  by (*simp add*: *is-lb-def*)

lemma *is-lbD*: $[|S >| l;\ x \in S|] ==> l \sqsubseteq x$
  by (*simp add*: *is-lb-def*)

lemma *is-lb-empty* [*simp*]: $\{\} >| l$
  unfolding *is-lb-def* **by** *fast*

lemma *is-lb-insert* [*simp*]: $(insert\ x\ A) >| y = (y \sqsubseteq x \wedge A >| y)$
  unfolding *is-lb-def* **by** *fast*

lemma *is-lb-downward*: $[|S >| l;\ y \sqsubseteq l|] ==> S >| y$
  unfolding *is-lb-def* **by** (*fast intro*: *below-trans*)

30.2 Greatest lower bounds

definition *is-glb* :: $'a$ *set* $\Rightarrow$ $'a$ $\Rightarrow$ *bool* (**infix** >>| *55*) **where**
  S >>| x $\longleftrightarrow$ S >| x $\wedge$ ($\forall$ u. S >| u --> $u \sqsubseteq x$)

definition *glb* :: $'a$ *set* $\Rightarrow$ $'a$ ($\bigsqcap$- [*60*]*60*) **where**
  glb S = (*THE* x. S >>| x)

Access to the definition as inference rule

lemma *is-glbD1*: S >>| x ==> S >| x
  unfolding *is-glb-def* **by** *fast*

lemma *is-glbD2*: [|S >>| x; S >| u|] ==> $u \sqsubseteq x$
  unfolding *is-glb-def* **by** *fast*

lemma (**in** *po*) *is-glbI*: [|S >| x; !!u. S >| u ==> $u \sqsubseteq x$|] ==> S >>| x
  unfolding *is-glb-def* **by** *fast*

lemma *is-glb-above-iff*: S >>| x ==> $u \sqsubseteq x$ $\longleftrightarrow$ S >| u
  unfolding *is-glb-def is-lb-def* **by** (*metis below-trans*)

glbs are unique

lemma *is-glb-unique*: [|S >>| x; S >>| y|] ==> $x = y$
  unfolding *is-glb-def is-lb-def* **by** (*blast intro*: *below-antisym*)

technical lemmas about *glb* and *op* >>|

lemma *is-glb-glb*: M >>| x ==> M >>| *glb* M
  unfolding *glb-def* **by** (*rule theI* [*OF* - *is-glb-unique*])

lemma *glb-eqI*: M >>| l ==> *glb* $M = l$
  by (*rule is-glb-unique* [*OF is-glb-glb*])

lemma *is-glb-singleton*: $\{x\}$ >>| x
  by (*simp add*: *is-glb-def*)

lemma *glb-singleton* [*simp*]: *glb* $\{x\} = x$
  by (*rule is-glb-singleton* [*THEN glb-eqI*])

lemma *is-glb-bin*: $x \sqsubseteq y$ ==> $\{x, y\}$ >>| x
  by (*simp add*: *is-glb-def*)

lemma *glb-bin*: $x \sqsubseteq y$ ==> *glb* $\{x, y\} = x$
  by (*rule is-glb-bin* [*THEN glb-eqI*])

lemma *is-glb-maximal*: [|S >| x; $x \in S$|] ==> S >>| x
  by (*erule is-glbI*, *erule* (*1*) *is-lbD*)

lemma *glb-maximal*: [|S >| x; $x \in S$|] ==> *glb* $S = x$

by (*rule is-glb-maximal* [*THEN glb-eqI*])

lemma *glb-above*: $S >>| z \Longrightarrow x \sqsubseteq glb\ S \longleftrightarrow S >| x$
by (*metis glb-eqI is-glb-above-iff*)
end

lemma (**in** *cpo*) *Meet-insert*: $S >>| l \Longrightarrow \{x, l\} >>| l2 \Longrightarrow insert\ x\ S >>| l2$
apply (*rule is-glbI*)
apply (*metis is-glb-above-iff is-glb-def is-lb-insert*)
by (*metis is-glb-above-iff is-glb-def is-glb-singleton is-lb-insert*)

Binary, hence finite meets.

class *Finite-Meet-cpo* = *cpo* +
assumes *binary-meet-exists*: $\exists\ l.\ l \sqsubseteq x \wedge l \sqsubseteq y \wedge (\forall\ z.\ z \sqsubseteq x \longrightarrow z \sqsubseteq y \longrightarrow z \sqsubseteq l)$
begin

lemma *binary-meet-exists'*: $\exists l.\ \{x, y\} >>| l$
using *binary-meet-exists*[*of x y*]
unfolding *is-glb-def is-lb-def*
by *auto*

lemma *finite-meet-exists*:
assumes $S \neq \{\}$
and *finite S*
shows $\exists x.\ S >>| x$
using ‹$S \neq \{\}$›
apply (*induct rule*: *finite-induct*[*OF* ‹*finite S*›])
apply (*erule notE*, *rule refl*)[*1*]
apply (*case-tac* $F = \{\}$)
apply (*metis is-glb-singleton*)
apply (*metis Meet-insert binary-meet-exists'*)
done
end

definition *meet* :: *'a::cpo* ⇒ *'a* ⇒ *'a* (**infix** ⊓ *80*) **where**
$x \sqcap y = (if\ \exists\ z.\ \{x, y\} >>| z\ then\ glb\ \{x, y\}\ else\ x)$

lemma *meet-def'*: $(x::'a::Finite\text{-}Meet\text{-}cpo) \sqcap y = glb\ \{x, y\}$
unfolding *meet-def* **by** (*metis binary-meet-exists'*)

lemma *meet-comm*: $(x::'a::Finite\text{-}Meet\text{-}cpo) \sqcap y = y \sqcap x$ **unfolding** *meet-def'* **by** (*metis insert-commute*)

lemma *meet-bot1*[*simp*]:
fixes *y* :: *'a* :: {*Finite-Meet-cpo*,*pcpo*}
shows $(\bot \sqcap y) = \bot$ **unfolding** *meet-def'* **by** (*metis minimal po-class.glb-bin*)
lemma *meet-bot2*[*simp*]:
fixes *x* :: *'a* :: {*Finite-Meet-cpo*,*pcpo*}
shows $(x \sqcap \bot) = \bot$ **by** (*metis meet-bot1 meet-comm*)

lemma *meet-below1*[*intro*]:
fixes $x\ y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
assumes $x \sqsubseteq z$
shows $(x \sqcap y) \sqsubseteq z$ **unfolding** *meet-def'* **by** (*metis assms binary-meet-exists' below-trans glb-eqI is-glbD1 is-lb-insert*)
lemma *meet-below2*[*intro*]:
fixes $x\ y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
assumes $y \sqsubseteq z$
shows $(x \sqcap y) \sqsubseteq z$ **unfolding** *meet-def'* **by** (*metis assms binary-meet-exists' below-trans glb-eqI is-glbD1 is-lb-insert*)

lemma *meet-above-iff*:
fixes $x\ y\ z :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
shows $z \sqsubseteq x \sqcap y \longleftrightarrow z \sqsubseteq x \wedge z \sqsubseteq y$
proof−
obtain g **where** $\{x,y\} >>|\ g$ **by** (*metis binary-meet-exists'*)
thus *?thesis*
unfolding *meet-def'* **by** (*simp add*: *glb-above*)
qed

lemma *below-meet*[*simp*]:
fixes $x\ y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
assumes $x \sqsubseteq z$
shows $(x \sqcap z) = x$ **by** (*metis assms glb-bin meet-def'*)

lemma *below-meet2*[*simp*]:
fixes $x\ y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
assumes $z \sqsubseteq x$
shows $(x \sqcap z) = z$ **by** (*metis assms below-meet meet-comm*)

lemma *meet-aboveI*:
fixes $x\ y\ z :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
shows $z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq x \sqcap y$ **by** (*simp add*: *meet-above-iff*)

lemma *is-meetI*:
fixes $x\ y\ z :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
assumes $z \sqsubseteq x$
assumes $z \sqsubseteq y$
assumes $\bigwedge a.\ [\![\ a \sqsubseteq x\ ;\ a \sqsubseteq y\ ]\!] \Longrightarrow a \sqsubseteq z$
shows $x \sqcap y = z$
by (*metis assms below-antisym meet-above-iff below-refl*)

lemma *meet-assoc*[*simp*]: $((x::'a::\mathit{Finite\text{-}Meet\text{-}cpo}) \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
apply (*rule is-meetI*)
apply (*metis below-refl meet-above-iff*)
apply (*metis below-refl meet-below2*)
apply (*metis meet-above-iff*)
done

lemma *meet-self* [*simp*]: $r \sqcap r = (r{::}'a{::}Finite\text{-}Meet\text{-}cpo)$
by (*metis below-refl is-meetI*)

lemma [*simp*]: $(r{::}'a{::}Finite\text{-}Meet\text{-}cpo) \sqcap (r \sqcap x) = r \sqcap x$
by (*metis below-refl is-meetI meet-below1*)

lemma *meet-monofun1*:
fixes $y :: 'a :: Finite\text{-}Meet\text{-}cpo$
shows *monofun* $(\lambda x.\ (x \sqcap y))$
by (*rule monofunI*)(*auto simp add*: *meet-above-iff*)

lemma *chain-meet1*:
fixes $y :: 'a :: Finite\text{-}Meet\text{-}cpo$
assumes *chain Y*
shows *chain* $(\lambda\ i.\ Y\ i \sqcap y)$
by (*rule chainI*) (*auto simp add*: *meet-above-iff intro*: *chainI chainE*[*OF assms*])

class *cont-binary-meet* = *Finite-Meet-cpo* +
assumes *meet-cont'*: $chain\ Y \Longrightarrow (\bigsqcup\ i.\ Y\ i) \sqcap y = (\bigsqcup\ i.\ Y\ i \sqcap y)$

lemma *meet-cont1*:
fixes $y :: 'a :: cont\text{-}binary\text{-}meet$
shows *cont* $(\lambda x.\ (x \sqcap y))$
by (*rule contI2*[*OF meet-monofun1*]) (*simp add*: *meet-cont'*)

lemma *meet-cont2*:
fixes $x :: 'a :: cont\text{-}binary\text{-}meet$
shows *cont* $(\lambda y.\ (x \sqcap y))$ **by** (*subst meet-comm*, *rule meet-cont1*)

lemma *meet-cont*[*cont2cont*,*simp*]:*cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *cont* $(\lambda x.\ (f\,x \sqcap (g\,x{::}'a{::}cont\text{-}binary\text{-}meet)))$
apply (*rule cont2cont-case-prod*[**where** $g = \lambda\ x.\ (f\ x,\ g\ x)$ **and** $f = \lambda\ p\ x\ y\ .\ x \sqcap y$, *simplified*])
apply (*rule meet-cont1*)
apply (*rule meet-cont2*)
apply (*metis cont2cont-Pair*)
done

end

31 C-Meet.tex

theory *C−Meet*
imports *C HOLCF−Meet*
begin

instantiation *C* :: *Finite-Meet-cpo* **begin**
fixrec *C-meet* :: $C \rightarrow C \rightarrow C$

where $C\text{-}meet{\cdot}(C{\cdot}a){\cdot}(C{\cdot}b) = C{\cdot}(C\text{-}meet{\cdot}a{\cdot}b)$

lemma[*simp*]: $C\text{-}meet{\cdot}\bot{\cdot}y = \bot$ $C\text{-}meet{\cdot}x{\cdot}\bot = \bot$ **by** (*fixrec-simp*, *cases x*, *fixrec-simp*+)

instance
apply *standard*
proof(*intro exI conjI strip*)
  fix *x y*
  {
    fix *t*
    have $(t \sqsubseteq C\text{-}meet{\cdot}x{\cdot}y) = (t \sqsubseteq x \wedge t \sqsubseteq y)$
    proof (*induct t rule*:*C*.*take-induct*)
      fix *n*
      show $(C\text{-}take\ n{\cdot}t \sqsubseteq C\text{-}meet{\cdot}x{\cdot}y) = (C\text{-}take\ n{\cdot}t \sqsubseteq x \wedge C\text{-}take\ n{\cdot}t \sqsubseteq y)$
      proof (*induct n arbitrary*: *t x y rule*:*nat-induct*)
        case *0* **thus** *?case* **by** *auto*
        next
        case (*Suc n t x y*)
          with *C*.*nchotomy*[*of t*] *C*.*nchotomy*[*of x*] *C*.*nchotomy*[*of y*]
          show *?case* **by** *fastforce*
      qed
    qed *auto*
  } **note** $* =$ *this*
  show $C\text{-}meet{\cdot}x{\cdot}y \sqsubseteq x$ **using** $*$ **by** *auto*
  show $C\text{-}meet{\cdot}x{\cdot}y \sqsubseteq y$ **using** $*$ **by** *auto*
  fix *z*
  assume $z \sqsubseteq x$ **and** $z \sqsubseteq y$
  thus $z \sqsubseteq C\text{-}meet{\cdot}x{\cdot}y$ **using** $*$ **by** *auto*
qed
end

lemma *C-meet-is-meet*: $(z \sqsubseteq C\text{-}meet{\cdot}x{\cdot}y) = (z \sqsubseteq x \wedge z \sqsubseteq y)$
proof (*induct z rule*:*C*.*take-induct*)
  fix *n*
  show $(C\text{-}take\ n{\cdot}z \sqsubseteq C\text{-}meet{\cdot}x{\cdot}y) = (C\text{-}take\ n{\cdot}z \sqsubseteq x \wedge C\text{-}take\ n{\cdot}z \sqsubseteq y)$
  proof (*induct n arbitrary*: *z x y rule*:*nat-induct*)
    case *0* **thus** *?case* **by** *auto*
    next
    case (*Suc n z x y*) **thus** *?case*
      apply −
      apply (*cases z*, *simp*)
      apply (*cases x*, *simp*)
      apply (*cases y*, *simp*)
      apply (*fastforce simp add*: *cfun-below-iff*)
      done
  qed
qed *auto*

instance *C* :: *cont-binary-meet*

proof (*standard*, *goal-cases*)
  have [*simp*]:$\bigwedge$ *x y*. $x \sqcap y$ = *C-meet*$\cdot x \cdot y$
    using *C-meet-is-meet*
    by (*blast intro*: *is-meetI*)
  case *1* **thus** *?case*
    by (*simp add*: *ch2ch-Rep-cfunR contlub-cfun-arg contlub-cfun-fun*)
qed

lemma [*simp*]: $C \cdot r \sqcap r = r$
  by (*auto intro*: *is-meetI simp add*: *below-C*)

lemma [*simp*]: $r \sqcap C \cdot r = r$
  by (*auto intro*: *is-meetI simp add*: *below-C*)

lemma [*simp*]: $C \cdot r \sqcap C \cdot r' = C \cdot (r \sqcap r')$
  apply (*rule is-meetI*)
  apply (*metis below-refl meet-below1 monofun-cfun-arg*)
  apply (*metis below-refl meet-below2 monofun-cfun-arg*)
  apply (*case-tac a*)
  apply *auto*
  by (*metis meet-above-iff*)

end

32 C-restr.tex

theory *C−restr*
imports *C C−Meet HOLCF−Utils*
begin

32.1 The demand of a C-function

The demand is the least amount of resources required to produce a non-bottom element, if at all.

definition *demand* :: $(C \to {'a}{::}pcpo) \Rightarrow C$ **where**
  $demand\ f = (if\ f \cdot C^\infty \neq \bot\ then\ C^{(LEAST\ n.\ f \cdot C^n \neq \bot)}\ else\ C^\infty)$

Because of continuity, a non-bottom value can always be obtained with finite resources.

lemma *finite-resources-suffice*:
  assumes $f \cdot C^\infty \neq \bot$
  obtains n **where** $f \cdot C^n \neq \bot$
proof−
  {
  assume $\forall n.\ f \cdot (C^n) = \bot$
  hence $f \cdot C^\infty \sqsubseteq \bot$
    by (*auto intro*: *lub-below*[*OF ch2ch-Rep-cfunR*[*OF chain-iterate*]]

simp add: *Cinf-def fix-def2 contlub-cfun-arg*[*OF chain-iterate*])
with *assms* **have** *False* **by** *simp*
}
thus *?thesis* **using** *that* **by** *blast*
qed

Because of monotonicity, a non-bottom value can always be obtained with more resources.

lemma *more-resources-suffice*:
assumes $f \cdot r \neq \bot$ **and** $r \sqsubseteq r'$
shows $f \cdot r' \neq \bot$
using *assms*(*1*) *monofun-cfun-arg*[*OF assms*(*2*), **where** $f = f$]
by *auto*

lemma *infinite-resources-suffice*:
shows $f \cdot r \neq \bot \Longrightarrow f \cdot C^\infty \neq \bot$
by (*erule more-resources-suffice*[*OF - below-Cinf*])

lemma *demand-suffices*:
assumes $f \cdot C^\infty \neq \bot$
shows $f \cdot (\textit{demand } f) \neq \bot$
apply (*simp add*: *assms demand-def*)
apply (*rule finite-resources-suffice*[*OF assms*])
apply (*rule LeastI*)
apply *assumption*
done

lemma *not-bot-demand*:
$f \cdot r \neq \bot \longleftrightarrow \textit{demand } f \neq C^\infty \wedge \textit{demand } f \sqsubseteq r$
proof(*intro iffI*)
assume $f \cdot r \neq \bot$
thus $\textit{demand } f \neq C^\infty \wedge \textit{demand } f \sqsubseteq r$
apply (*cases r rule*:*C-cases*)
apply (*auto intro*: *Least-le simp add*: *demand-def dest*: *infinite-resources-suffice*)
done
next
assume $*$: $\textit{demand } f \neq C^\infty \wedge \textit{demand } f \sqsubseteq r$
then have $f \cdot C^\infty \neq \bot$ **by** (*auto intro*: *Least-le simp add*: *demand-def dest*: *infinite-resources-suffice*)
hence $f \cdot (\textit{demand } f) \neq \bot$ **by** (*rule demand-suffices*)
moreover from $*$ **have** $\textit{demand } f \sqsubseteq r$**..**
ultimately
show $f \cdot r \neq \bot$ **by** (*rule more-resources-suffice*)
qed

lemma *infinity-bot-demand*:
$f \cdot C^\infty = \bot \longleftrightarrow \textit{demand } f = C^\infty$
by (*metis below-Cinf not-bot-demand*)

lemma *demand-suffices'*:

assumes *demand* $f = C^{n}$
shows $f \cdot (demand\ f) \neq \bot$
by (*metis C-eq-Cinf assms demand-suffices infinity-bot-demand*)

lemma *demand-Suc-Least*:
assumes [*simp*]: $f \cdot \bot = \bot$
assumes *demand* $f \neq C^{\infty}$
shows *demand* $f = C^{(Suc\ (LEAST\ n.\ f \cdot C^{Suc\ n} \neq \bot))}$
proof−
from *assms*
have *demand* $f = C^{(LEAST\ n.\ f \cdot C^{n} \neq \bot)}$ **by** (*auto simp add*: *demand-def*)
also
then obtain n **where** $f \cdot C^{n} \neq \bot$ **by** (*metis demand-suffices′*)
hence $(LEAST\ n.\ f \cdot C^{n} \neq \bot) = Suc\ (LEAST\ n.\ f \cdot C^{Suc\ n} \neq \bot)$
apply (*rule Least-Suc*) **by** *simp*
finally show *?thesis*.
qed

lemma *demand-C-case*[*simp*]: *demand* $(C\text{-}case \cdot f) = C \cdot (demand\ f)$
proof(*cases demand* $(C\text{-}case \cdot f) = C^{\infty}$)
case *True*
then have $C\text{-}case \cdot f \cdot C^{\infty} = \bot$
by (*metis infinity-bot-demand*)
with *True*
show *?thesis* **apply** *auto* **by** (*metis infinity-bot-demand*)
next
case *False*
hence *demand* $(C\text{-}case \cdot f) = C^{Suc\ (LEAST\ n.\ (C\text{-}case \cdot f) \cdot C^{Suc\ n} \neq \bot)}$
by (*rule demand-Suc-Least*[*OF C.case-rews*(*1*)])
also have $\ldots = C \cdot C^{LEAST\ n.\ f \cdot C^{n} \neq \bot}$ **by** *simp*
also have $\ldots = C \cdot (demand\ f)$
using *False* **unfolding** *demand-def* **by** *auto*
finally show *?thesis*.
qed

lemma *demand-contravariant*:
assumes $f \sqsubseteq g$
shows *demand* $g \sqsubseteq$ *demand* f
proof(*cases demand f rule*:*C-cases*)
fix n
assume *demand* $f = C^{n}$
hence $f \cdot (demand\ f) \neq \bot$ **by** (*metis demand-suffices′*)
also note *monofun-cfun-fun*[*OF assms*]
finally have $g \cdot (demand\ f) \neq \bot$ **by** *this* (*intro cont2cont*)
thus *demand* $g \sqsubseteq$ *demand* f **unfolding** *not-bot-demand* **by** *auto*
qed *auto*

32.2 Restricting functions with domain C

fixrec *C-restr* :: $C \rightarrow (C \rightarrow 'a::pcpo) \rightarrow (C \rightarrow 'a)$
  where $C\text{-}restr \cdot r \cdot f \cdot r' = (f \cdot (r \sqcap r'))$

abbreviation *C-restr-syn* :: $(C \rightarrow 'a::pcpo) \Rightarrow C \Rightarrow (C \rightarrow 'a)$ (*-|-* [*111*,*110*] *110*)
  where $f|_r \equiv C\text{-}restr \cdot r \cdot f$

lemma [*simp*]: $\bot|_r = \bot$ **by** *fixrec-simp*
lemma [*simp*]: $f \cdot \bot = \bot \Longrightarrow f|_\bot = \bot$ **by** *fixrec-simp*

lemma *C-restr-C-restr*[*simp*]: $(v|_{r'})|_r = v|_{(r' \sqcap r)}$
  by (*rule cfun-eqI*) *simp*

lemma *C-restr-eqD*:
  assumes $f|_r = g|_r$
  assumes $r' \sqsubseteq r$
  shows $f \cdot r' = g \cdot r'$
by (*metis C-restr.simps assms below-refl is-meetI*)

lemma *C-restr-eq-lower*:
  assumes $f|_r = g|_r$
  assumes $r' \sqsubseteq r$
  shows $f|_{r'} = g|_{r'}$
by (*metis C-restr-C-restr assms below-refl is-meetI*)

lemma *C-restr-below*[*intro*, *simp*]:
  $x|_r \sqsubseteq x$
  apply (*rule cfun-belowI*)
  apply *simp*
  by (*metis below-refl meet-below2 monofun-cfun-arg*)

lemma *C-restr-below-cong*:
  $(\bigwedge r'.\ r' \sqsubseteq r \Longrightarrow f \cdot r' \sqsubseteq g \cdot r') \Longrightarrow f|_r \sqsubseteq g|_r$
  apply (*rule cfun-belowI*)
  apply *simp*
  by (*metis below-refl meet-below1*)

lemma *C-restr-cong*:
  $(\bigwedge r'.\ r' \sqsubseteq r \Longrightarrow f \cdot r' = g \cdot r') \Longrightarrow f|_r = g|_r$
  apply (*intro below-antisym C-restr-below-cong*)
  by (*metis below-refl*)+

lemma *C-restr-C-cong*:
  $(\bigwedge r'.\ r' \sqsubseteq r \Longrightarrow f \cdot (C \cdot r') = g \cdot (C \cdot r')) \Longrightarrow f \cdot \bot = g \cdot \bot \Longrightarrow f|_{C \cdot r} = g|_{C \cdot r}$
  apply (*rule C-restr-cong*)
  by (*case-tac* r', *auto*)

lemma *C-restr-C-case*[*simp*]:
$(C\text{-}case{\cdot}f)|_{C{\cdot}r} = C\text{-}case{\cdot}(f|_r)$
apply (*rule cfun-eqI*)
apply *simp*
apply (*case-tac x*)
apply *simp*
apply *simp*
done

lemma *C-restr-bot-demand*:
assumes $C{\cdot}r \sqsubseteq demand\ f$
shows $f|_r = \bot$
proof(*rule cfun-eqI*)
fix r'
have $f{\cdot}(r \sqcap r') = \bot$
proof(*rule classical*)
have $r \sqsubseteq C \cdot r$ **by** (*rule below-C*)
also
note *assms*
also
assume $*$: $f{\cdot}(r \sqcap r') \neq \bot$
hence $demand\ f \sqsubseteq (r \sqcap r')$ **unfolding** *not-bot-demand* **by** *auto*
hence $demand\ f \sqsubseteq r$ **by** (*metis below-refl meet-below1 below-trans*)
finally (*below-antisym*) **have** $r = demand\ f$ **by** *this* (*intro cont2cont*)
with *assms*
have $demand\ f = C^\infty$ **by** (*cases demand f rule*: *C-cases*) (*auto simp add*: *iterate-Suc*[*symmetric*] *simp del*: *iterate-Suc*)
thus $f{\cdot}(r \sqcap r') = \bot$ **by** (*metis not-bot-demand*)
qed
thus $(f|_r){\cdot}r' = \bot{\cdot}r'$ **by** *simp*
qed

32.3 Restricting maps of C-ranged functions

definition *env-C-restr* :: $C \to ('var::type \Rightarrow (C \to 'a::pcpo)) \to ('var \Rightarrow (C \to 'a))$ **where**
env-C-restr $= (\Lambda\ r\ f.\ \ cfun\text{-}comp{\cdot}(C\text{-}restr{\cdot}r){\cdot}f)$

abbreviation *env-C-restr-syn* :: $('var::type \Rightarrow (C \to 'a::pcpo)) \Rightarrow C \Rightarrow\ ('var \Rightarrow (C \to 'a))$ (-|°- [*111*,*110*] *110*)
where $f|^\circ{}_r \equiv env\text{-}C\text{-}restr{\cdot}r{\cdot}f$

lemma *env-C-restr-upd*[*simp*]: $(\varrho(x := v))|^\circ{}_r = (\varrho|^\circ{}_r)(x := v|_r)$
unfolding *env-C-restr-def* **by** *simp*

lemma *env-C-restr-lookup*[*simp*]: $(\varrho|^\circ{}_r)\ v = \varrho\ v|_r$
unfolding *env-C-restr-def* **by** *simp*

lemma *env-C-restr-bot*[*simp*]: $\bot|^\circ{}_r = \bot$

unfolding *env-C-restr-def* **by** *auto*

lemma *env-C-restr-restr-below*[*intro*]: $\varrho|^\circ{}_r \sqsubseteq \varrho$
by (*auto intro*: *fun-belowI*)

lemma *env-C-restr-env-C-restr*[*simp*]: $(v|^\circ{}_{r'})|^\circ{}_r = v|^\circ{}_{(r' \sqcap r)}$
unfolding *env-C-restr-def* **by** *auto*

lemma *env-C-restr-cong*:
$(\bigwedge x\ r'.\ r' \sqsubseteq r \Longrightarrow f\ x \cdot r' = g\ x \cdot r') \Longrightarrow f|^\circ{}_r = g|^\circ{}_r$
unfolding *env-C-restr-def*
by (*rule ext*) (*auto intro*: *C-restr-cong*)

end

33 ResourcedDenotational.tex

theory *ResourcedDenotational*
imports *Abstract−Denotational−Props CValue−Nominal C−restr*
begin

type-synonym $CEnv = var \Rightarrow (C \rightarrow CValue)$

interpretation *semantic-domain*
$\Lambda\ f\ .\ \Lambda\ r.\ CFn\cdot(\Lambda\ v.\ (f\cdot(v))|_r)$
$\Lambda\ x\ y.\ (\Lambda\ r.\ (x\cdot r \downarrow CFn\ y|_r)\cdot r)$
$\Lambda\ b\ r.\ CB\cdot b$
$\Lambda\ scrut\ v1\ v2\ r.\ CB\text{-}project\cdot(scrut\cdot r)\cdot(v1\cdot r)\cdot(v2\cdot r)$
C-case.

abbreviation *ESem-syn''* ($\mathcal{N}[\![\ \text{-}\ ]\!]_{\text{-}}$ [*60*,*60*] *60*) **where** $\mathcal{N}[\![\ e\ ]\!]_\varrho \equiv ESem\ e \cdot \varrho$
abbreviation *EvalHeapSem-syn''* ($\boldsymbol{\mathcal{N}}[\![\ \text{-}\ ]\!]_{\text{-}}$ [*0*,*0*] *110*) **where** $\boldsymbol{\mathcal{N}}[\![\Gamma]\!]_\varrho \equiv evalHeap\ \Gamma\ (\lambda\ e.\ \mathcal{N}[\![e]\!]_\varrho)$
abbreviation *HSem-syn'* ($\mathcal{N}\{\!\!\{\text{-}\}\!\!\}_{\text{-}}$ [*60*,*60*] *60*) **where** $\mathcal{N}\{\!\!\{\Gamma\}\!\!\}_\varrho \equiv HSem\ \Gamma \cdot \varrho$
abbreviation *HSem-bot* ($\mathcal{N}\{\!\!\{\text{-}\}\!\!\}$ [*60*] *60*) **where** $\mathcal{N}\{\!\!\{\Gamma\}\!\!\} \equiv \mathcal{N}\{\!\!\{\Gamma\}\!\!\}\bot$

Here we re-state the simplification rules, cleaned up by beta-reducing the locale parameters.

lemma *CESem-simps*:
$\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_\varrho\ = (\Lambda\ (C\cdot r).\ CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)})|_r))$
$\mathcal{N}[\![\ App\ e\ x\ ]\!]_\varrho\ = (\Lambda\ (C\cdot r).\ ((\mathcal{N}[\![\ e\ ]\!]_\varrho)\cdot r \downarrow CFn\ \varrho\ x|_r)\cdot r)$
$\mathcal{N}[\![\ Var\ x\ ]\!]_\varrho\ = (\Lambda\ (C\cdot r).\ (\varrho\ \ x) \cdot r)$
$\mathcal{N}[\![\ Bool\ b\ ]\!]_\varrho\ = (\Lambda\ (C\cdot r).\ CB\cdot(Discr\ b))$
$\mathcal{N}[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_\varrho\ = (\Lambda\ (C\cdot r).\ CB\text{-}project\cdot((\mathcal{N}[\![\ scrut\ ]\!]_\varrho)\cdot r)\cdot((\mathcal{N}[\![\ e_1\ ]\!]_\varrho)\cdot r)\cdot((\mathcal{N}[\![\ e_2\ ]\!]_\varrho)\cdot r))$
$\mathcal{N}[\![\ Let\ as\ body\ ]\!]_\varrho = (\Lambda\ (C\cdot r).\ (\mathcal{N}[\![body]\!]_{\mathcal{N}\{\!\!\{as\}\!\!\}\varrho}) \cdot r)$
by (*auto simp add*: *eta-cfun*)

lemma *CESem-bot*[*simp*]:$(\mathcal{N}[\![\ e\ ]\!]_{\sigma})\cdot\bot = \bot$
  by (*nominal-induct e arbitrary*: σ *rule*: *exp-strong-induct*) *auto*

lemma *CHSem-bot*[*simp*]:$((\mathcal{N}\{\!|\ \Gamma\ |\!\})\ x)\cdot\bot = \bot$
  by (*cases* $x \in$ *domA* Γ) (*auto simp add*: *lookup-HSem-heap lookup-HSem-other*)

Sometimes we do not care much about the resource usage and just want a simpler formula.

lemma *CESem-simps-no-tick*:
  $(\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho})\cdot r \sqsubseteq CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)})|_r)$
  $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\varrho})\cdot r \quad \sqsubseteq ((\mathcal{N}[\![\ e\ ]\!]_{\varrho})\cdot r \downarrow CFn\ \varrho\ x|_r)\cdot r$
  $\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho} \qquad \sqsubseteq \varrho\ x$
  $(\mathcal{N}[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_{\varrho})\cdot r \sqsubseteq CB\text{-}project\cdot((\mathcal{N}[\![\ scrut\ ]\!]_{\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_1\ ]\!]_{\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_2\ ]\!]_{\varrho})\cdot r)$
  $\mathcal{N}[\![\ Let\ as\ body\ ]\!]_{\varrho} \quad \sqsubseteq \ \mathcal{N}[\![body]\!]_{\mathcal{N}\{\!|as|\!\}\varrho}$
  apply −
  apply (*rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
  apply (*rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
  apply (*rule cfun-belowI*, *rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
  apply (*rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
  apply (*rule cfun-belowI*, *rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
  done

lemma *CELam-no-restr*: $(\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho})\cdot r \sqsubseteq CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)}))$
proof−
 have $(\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho})\cdot r \sqsubseteq CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)})|_r)$ **by** (*rule CESem-simps-no-tick*)
  also have $\ldots \sqsubseteq CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)}))$
   by (*intro cont2cont monofun-LAM below-trans*[*OF C-restr-below*] *monofun-cfun-arg below-refl fun-upd-mono*)
  finally show *?thesis* **by** *this* (*intro cont2cont*)
qed

lemma *CEApp-no-restr*: $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\varrho})\cdot r \quad \sqsubseteq ((\mathcal{N}[\![\ e\ ]\!]_{\varrho})\cdot r \downarrow CFn\ \varrho\ x)\cdot r$
proof−
  have $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\varrho})\cdot r \ \sqsubseteq ((\mathcal{N}[\![\ e\ ]\!]_{\varrho})\cdot r \downarrow CFn\ \varrho\ x|_r)\cdot r$ **by** (*rule CESem-simps-no-tick*)
  also have $\varrho\ x|_r \sqsubseteq \varrho\ x$ **by** (*rule C-restr-below*)
  finally show *?thesis* **by** *this* (*intro cont2cont*)
qed

end

34 CorrectnessResourced.tex

theory *CorrectnessResourced*
  imports *ResourcedDenotational Launchbury*
begin

theorem *correctness*:
  assumes $\Gamma : e \Downarrow_{L} \Delta : z$
  and *fv* $(\Gamma, e) \subseteq$ *set* $L \cup$ *domA* Γ
  shows $\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho} \sqsubseteq \mathcal{N}[\![z]\!]_{\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho}$ **and** $(\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho)\ f|`\ domA\ \Gamma \sqsubseteq (\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho)\ f|`\ domA\ \Gamma$
  using *assms*
proof(*nominal-induct arbitrary*: ϱ *rule*:*reds.strong-induct*)
case *Lambda*
  case *1* **show** *?case*..
  case *2* **show** *?case*..
next
case (*Application* $y\ \Gamma\ e\ x\ L\ \Delta\ \Theta\ z\ e'$)
  have *Gamma-subset*: *domA* $\Gamma \subseteq$ *domA* Δ
    by (*rule reds-doesnt-forget*[*OF Application.hyps*(*8*)])

  case *1*
  hence *prem1*: *fv* $(\Gamma, e) \subseteq$ *set* $L \cup$ *domA* Γ **and** $x \in$ *set* $L \cup$ *domA* Γ **by** *auto*
  moreover
  note *reds-pres-closed*[*OF Application.hyps*(*8*) *prem1*]
  moreover
  note *reds-doesnt-forget*[*OF Application.hyps*(*8*)]
  moreover
  have *fv* $(e'[y{::=}x]) \subseteq$ *fv* (*Lam* $[y].\ e') \cup \{x\}$
    by (*auto simp add*: *fv-subst-eq*)
  ultimately
  have *prem2*: *fv* $(\Delta, e'[y{::=}x]) \subseteq$ *set* $L \cup$ *domA* Δ **by** *auto*

  have $*$: $(\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho)\ x \sqsubseteq (\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho)\ x$
  proof(*cases* $x \in$ *domA* Γ)
    case *True*
    thus *?thesis*
      using *fun-belowD*[*OF Application.hyps*(*10*)[*OF prem1*], **where** $\varrho 1 = \varrho$ **and** $x = x$]
      by *simp*
  next
    case *False*
    from *False* ‹$x \in$ *set* $L \cup$ *domA* Γ› *reds-avoids-live*[*OF Application.hyps*(*8*)]
    show *?thesis* **by** (*auto simp add*: *lookup-HSem-other*)
  qed

  {
  fix r
  have $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho})\cdot r \sqsubseteq ((\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho})\cdot r \downarrow CFn\ (\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho)\ x)\cdot r$
    by (*rule CEApp-no-restr*)
  also have $((\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho})) \sqsubseteq ((\mathcal{N}[\![\ Lam\ [y].\ e'\ ]\!]_{\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho}))$
    using *Application.hyps*(*9*)[*OF prem1*].
  also note ‹$((\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho)\ x) \sqsubseteq (\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho)\ x$›
  also have $(\mathcal{N}[\![\ Lam\ [y].\ e'\ ]\!]_{\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho})\cdot r \sqsubseteq (CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e'\ ]\!]_{(\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho)(y\ :=\ v)})))$
    by (*rule CELam-no-restr*)
  also have $CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e'\ ]\!]_{(\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho)(y\ :=\ v)})) \downarrow CFn\ ((\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho)\ x) = (\mathcal{N}[\![\ e'\ ]\!]_{(\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho)(y\ :=\ (\mathcal{N}\{\!\!\{\Delta\}\!\!\}\varrho)\ x)})$

by *simp*
also have ... $= (\mathcal{N}[\![\ e'[y ::= x]\ ]\!]_{(\mathcal{N}\{\!\![\Delta]\!\!\}\varrho)})$
unfolding *ESem-subst*..
also have ... $\sqsubseteq \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!\![\Theta]\!\!\}\varrho}$
using *Application.hyps*(*12*)[*OF prem2*].
finally
have $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\mathcal{N}\{\!\![\Gamma]\!\!\}\varrho})\cdot r \sqsubseteq (\mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!\![\Theta]\!\!\}\varrho})\cdot r$ **by** *this* (*intro cont2cont*)+
}
thus *?case* **by** (*rule cfun-belowI*)

show $(\mathcal{N}\{\!\![\Gamma]\!\!\}\varrho)\ f|`\ (domA\ \Gamma) \sqsubseteq (\mathcal{N}\{\!\![\Theta]\!\!\}\varrho)\ \ f|`\ (domA\ \Gamma)$
using *Application.hyps*(*10*)[*OF prem1*]
env-restr-below-subset[*OF Gamma-subset Application.hyps*(*13*)[*OF prem2*]]
by (*rule below-trans*)
next
case (*Variable* Γ *x e L* Δ *z*)
hence [*simp*]:$x \in domA\ \Gamma$
by (*metis domA-from-set map-of-SomeD*)

case *2*

have $x \notin domA\ \Delta$
by (*rule reds-avoids-live*[*OF Variable.hyps*(*2*)], *simp-all*)

have *subset*: $domA\ (delete\ x\ \Gamma) \subseteq domA\ \Delta$
by (*rule reds-doesnt-forget*[*OF Variable.hyps*(*2*)])

let $?new = domA\ \Delta\ -\ domA\ \Gamma$
have $fv\ (delete\ x\ \Gamma,\ e) \cup \{x\} \subseteq fv\ (\Gamma,\ Var\ x)$
by (*rule fv-delete-heap*[*OF* ‹*map-of* Γ *x = Some e*›])
hence *prem*: $fv\ (delete\ x\ \Gamma,\ e) \subseteq set\ (x\ \#\ L) \cup domA\ (delete\ x\ \Gamma)$ **using** *2* **by** *auto*
hence *fv-subset*: $fv\ (delete\ x\ \Gamma,\ e) - domA\ (delete\ x\ \Gamma) \subseteq -\ ?new$
using *reds-avoids-live'*[*OF Variable.hyps*(*2*)] **by** *auto*

have $domA\ \Gamma \subseteq (-?new)$ **by** *auto*

have $\mathcal{N}\{\!\![\Gamma]\!\!\}\varrho = \mathcal{N}\{\!\![(x,e)\ \#\ delete\ x\ \Gamma]\!\!\}\varrho$
by (*rule HSem-reorder*[*OF map-of-delete-insert*[*symmetric*, *OF Variable*(*1*)]])
also have ... $= (\mu\ \varrho'.\ (\varrho\ {++}_{(domA\ (delete\ x\ \Gamma))}\ (\mathcal{N}\{\!\![delete\ x\ \Gamma]\!\!\}\varrho'))(\ x := \mathcal{N}[\![\ e\ ]\!]_{\varrho'}))$
by (*rule iterative-HSem*, *simp*)
also have ... $= (\mu\ \varrho'.\ (\varrho\ {++}_{(domA\ (delete\ x\ \Gamma))}\ (\mathcal{N}\{\!\![delete\ x\ \Gamma]\!\!\}\varrho'))(\ x := \mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!\![delete\ x\ \Gamma]\!\!\}\varrho'}))$
by (*rule iterative-HSem'*, *simp*)
finally
have $(\mathcal{N}\{\!\![\Gamma]\!\!\}\varrho)f|`\ (-\ ?new) \sqsubseteq (...)\ f|`\ (-\ ?new)$ **by** (*rule ssubst*) (*rule below-refl*)
also have ... $\sqsubseteq (\mu\ \varrho'.\ (\varrho\ {++}_{domA\ \Delta}\ (\mathcal{N}\{\!\![\Delta]\!\!\}\varrho'))(\ x := \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!\![\Delta]\!\!\}\varrho'}))\ f|`\ (-\ ?new)$

proof (*induction rule*: *parallel-fix-ind*[**where** $P = \lambda\ x\ y.\ x\ f|`\ (-\ ?new) \sqsubseteq y\ f|`\ (-\ ?new)$])
case *1* **show** *?case* **by** *simp*
next
case *2* **show** *?case* **..**
next
case ($3\ \sigma\ \sigma'$)
hence $\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma} \sqsubseteq \mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma'}$
and $(\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma)\ f|`\ domA\ (delete\ x\ \Gamma) \sqsubseteq (\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma')\ f|`\ domA\ (delete\ x\ \Gamma)$
using *fv-subset* **by** (*auto intro*: *ESem-fresh-cong-below HSem-fresh-cong-below env-restr-below-subset*[*OF* - *3*])
from *below-trans*[*OF this*(*1*) *Variable*(*3*)[*OF prem*]] *below-trans*[*OF this*(*2*) *Variable*(*4*)[*OF prem*]]
have $\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma} \sqsubseteq \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}\sigma'}$
and $(\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma)\ f|`\ domA\ (delete\ x\ \Gamma) \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\}\sigma')\ f|`\ domA\ (delete\ x\ \Gamma)$**.**
thus *?case*
using *subset*
by (*auto intro*!: *fun-belowI simp add*: *lookup-override-on-eq lookup-env-restr-eq elim*: *env-restr-belowD*)
qed
also have ... $= (\mu\ \varrho'.\ (\varrho +\!\!+_{domA\ \Delta} (\mathcal{N}\{\!|\Delta|\!\}\varrho'))(\ x := \mathcal{N}[\![\ z\ ]\!]_{\varrho'}))\ f|`\ (-?new)$
by (*rule arg-cong*[*OF iterative-HSem'*[*symmetric*], *OF* ‹$x \notin domA\ \Delta$›])
also have ... $= (\mathcal{N}\{\!|(x,z)\ \#\ \Delta|\!\}\varrho)\ \ f|`\ (-?new)$
by (*rule arg-cong*[*OF iterative-HSem*[*symmetric*], *OF* ‹$x \notin domA\ \Delta$›])
finally
show *le*: *?case* **by** (*rule env-restr-below-subset*[*OF* ‹$domA\ \Gamma \subseteq (-?new)$›]) (*intro cont2cont*)+

have $\mathcal{N}[\![\ Var\ x\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho} \sqsubseteq (\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ x$ **by** (*rule CESem-simps-no-tick*)
also have ... $\sqsubseteq (\mathcal{N}\{\!|(x,\ z)\ \#\ \Delta|\!\}\varrho)\ x$
using *fun-belowD*[*OF le*, **where** $x = x$] **by** *simp*
also have ... $= \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|(x,\ z)\ \#\ \Delta|\!\}\varrho}$
by (*simp add*: *lookup-HSem-heap*)
finally
show $\mathcal{N}[\![\ Var\ x\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho} \sqsubseteq \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|(x,\ z)\ \#\ \Delta|\!\}\varrho}$ **by** *this* (*intro cont2cont*)+
next
case (*Bool b*)
case *1*
show *?case* **by** *simp*
case *2*
show *?case* **by** *simp*
next
case (*IfThenElse* Γ *scrut* $L\ \Delta\ b\ e_1\ e_2\ \Theta\ z$)
have *Gamma-subset*: $domA\ \Gamma \subseteq domA\ \Delta$
by (*rule reds-doesnt-forget*[*OF IfThenElse.hyps*(*1*)])

let $?e = if\ b\ then\ e_1\ else\ e_2$

case *1*
thm *new-free-vars-on-heap*[*OF IfThenElse.hyps*(*1*)]

hence *prem1*: *fv* $(\Gamma,\ scrut) \subseteq$ *set* $L \cup$ *domA* Γ
and *prem2*: *fv* $(\Delta,\ ?e) \subseteq$ *set* $L \cup$ *domA* Δ
and *fv* $?e \subseteq$ *domA* $\Gamma \cup$ *set* L
using *new-free-vars-on-heap*[*OF IfThenElse.hyps*(*1*)] *Gamma-subset* **by** *auto*

{
fix r
have $(\mathcal{N}[\![\ (scrut\ ?\ e_1 : e_2)\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r \sqsubseteq$ *CB-project*$\cdot((\mathcal{N}[\![\ scrut\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_1\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)\cdot((\mathcal{N}[\![$
$e_2\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)$
by (*rule CESem-simps-no-tick*)
also have $\ldots \sqsubseteq$ *CB-project*$\cdot((\mathcal{N}[\![\ Bool\ b\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_1\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_2\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)$
by (*intro monofun-cfun-fun monofun-cfun-arg IfThenElse.hyps*(*2*)[*OF prem1*])
also have $\ldots = (\mathcal{N}[\![\ ?e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r$ **by** (*cases r*) *simp-all*
also have $\ldots \sqsubseteq (\mathcal{N}[\![\ ?e\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho})\cdot r$
proof(*rule monofun-cfun-fun*[*OF ESem-fresh-cong-below-subset*[*OF* ‹*fv* $?e \subseteq$ *domA* $\Gamma \cup$ *set*
L› *Env.env-restr-belowI*]])
fix x
assume $x \in$ *domA* $\Gamma \cup$ *set* L
thus $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ x \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\}\varrho)\ x$
proof(*cases* $x \in$ *domA* Γ)
assume $x \in$ *domA* Γ
from *IfThenElse.hyps*(*3*)[*OF prem1*]
have $((\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ f|`\ $ *domA* $\Gamma)\ x \sqsubseteq ((\mathcal{N}\{\!|\Delta|\!\}\varrho)\ f|`\ $ *domA* $\Gamma)\ x$ **by** (*rule fun-belowD*)
with ‹$x \in$ *domA* Γ› **show** *?thesis* **by** *simp*
next
assume $x \notin$ *domA* Γ
from *this* ‹$x \in$ *domA* $\Gamma \cup$ *set* L› *reds-avoids-live*[*OF IfThenElse.hyps*(*1*)]
show *?thesis*
by (*simp add*: *lookup-HSem-other*)
qed
qed
also have $\ldots \sqsubseteq (\mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|\Theta|\!\}\varrho})\cdot r$
by (*intro monofun-cfun-fun monofun-cfun-arg IfThenElse.hyps*(*5*)[*OF prem2*])
finally
have $(\mathcal{N}[\![\ (scrut\ ?\ e_1 : e_2)\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r \sqsubseteq (\mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|\Theta|\!\}\varrho})\cdot r$ **by** *this* (*intro cont2cont*)+
}
thus *?case* **by** (*rule cfun-belowI*)

show $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ f|`\ ($*domA* $\Gamma) \sqsubseteq (\mathcal{N}\{\!|\Theta|\!\}\varrho)\ f|`\ ($*domA* $\Gamma)$
using *IfThenElse.hyps*(*3*)[*OF prem1*]
env-restr-below-subset[*OF Gamma-subset IfThenElse.hyps*(*6*)[*OF prem2*]]
by (*rule below-trans*)
next
case (*Let as* Γ *L body* Δ z)
case *1*
have $*$: *domA as* $\cap$ *domA* $\Gamma = \{\}$ **by** (*metis Let.hyps*(*1*) *fresh-distinct*)

have *fv* (*as* @ Γ, *body*) − *domA* (*as* @ Γ) ⊆ *fv* (Γ, *Let as body*) − *domA* Γ
by *auto*
with *1* **have** *prem*: *fv* (*as* @ Γ, *body*) ⊆ *set L* ∪ *domA* (*as* @ Γ) **by** *auto*

have *f1*: *atom* ‘ *domA as* ♯* Γ
using *Let*(*1*) **by** (*simp add*: *set-bn-to-atom-domA*)

have $\mathcal{N}[\![\textit{Let as body}]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho} \sqsubseteq \mathcal{N}[\![\textit{body}]\!]_{\mathcal{N}\{\!|as|\!\}\mathcal{N}\{\!|\Gamma|\!\}\varrho}$
by (*rule CESem-simps-no-tick*)
also have ... $= \mathcal{N}[\![\textit{body}]\!]_{\mathcal{N}\{\!|as \ @ \ \Gamma|\!\}\varrho}$
by (*rule arg-cong*[*OF HSem-merge*[*OF f1*]])
also have ... $\sqsubseteq \mathcal{N}[\![z]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho}$
by (*rule Let.hyps*(*4*)[*OF prem*])
finally
show *?case* **by** *this* (*intro cont2cont*)+

have $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)$ *f*|‘ (*domA* Γ) = $(\mathcal{N}\{\!|as|\!\}(\mathcal{N}\{\!|\Gamma|\!\}\varrho))$ *f*|‘ (*domA* Γ)
unfolding *env-restr-HSem*[*OF* ∗]**..**
also have $\mathcal{N}\{\!|as|\!\}(\mathcal{N}\{\!|\Gamma|\!\}\varrho) = (\mathcal{N}\{\!|as \ @ \ \Gamma|\!\}\varrho)$
by (*rule HSem-merge*[*OF f1*])
also have ... *f*|‘ *domA* Γ ⊑ $(\mathcal{N}\{\!|\Delta|\!\}\varrho)$ *f*|‘ *domA* Γ
by (*rule env-restr-below-subset*[*OF - Let.hyps*(*5*)[*OF prem*]]) *simp*
finally
show $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)$ *f*|‘ *domA* Γ ⊑ $(\mathcal{N}\{\!|\Delta|\!\}\varrho)$ *f*|‘ *domA* Γ**.**
qed

corollary *correctness-empty-env*:
assumes $\Gamma : e \Downarrow_L \Delta : z$
and *fv* (Γ, *e*) ⊆ *set L*
shows $\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}} \sqsubseteq \mathcal{N}[\![z]\!]_{\mathcal{N}\{\!|\Delta|\!\}}$ **and** $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$
proof−
from *assms*(*2*) **have** *fv* (Γ, *e*) ⊆ *set L* ∪ *domA* Γ **by** *auto*
note *corr* = *correctness*[*OF assms*(*1*) *this*, **where** ϱ = ⊥]

show $\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}} \sqsubseteq \mathcal{N}[\![z]\!]_{\mathcal{N}\{\!|\Delta|\!\}}$ **using** *corr*(*1*)**.**

have $\mathcal{N}\{\!|\Gamma|\!\} = (\mathcal{N}\{\!|\Gamma|\!\})$ *f*|‘ *domA* Γ
using *env-restr-useless*[*OF HSem-edom-subset*, **where** ϱ*1* = ⊥] **by** *simp*
also have ... ⊑ $(\mathcal{N}\{\!|\Delta|\!\})$ *f*|‘ *domA* Γ **using** *corr*(*2*)**.**
also have ... ⊑ $\mathcal{N}\{\!|\Delta|\!\}$ **by** (*rule env-restr-below-itself*)
finally show $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$ **by** *this* (*intro cont2cont*)+
qed

end

35 ResourcedAdequacy.tex

theory *ResourcedAdequacy*
imports *ResourcedDenotational Launchbury AList−Utils CorrectnessResourced*
begin

lemma *demand-not-0*: *demand* $(\mathcal{N}[\![e]\!]_\varrho) \neq \bot$
proof
  assume *demand* $(\mathcal{N}[\![e]\!]_\varrho) = \bot$
  with *demand-suffices'*[**where** $n = 0$, *simplified*, *OF this*]
  have $(\mathcal{N}[\![e]\!]_\varrho)\cdot\bot \neq \bot$ **by** *simp*
  thus *False* **by** *simp*
qed

The semantics of an expression, given only r resources, will only use values from the environment with less resources.

lemma *restr-can-restrict-env*: $(\mathcal{N}[\![\ e\ ]\!]_\varrho)|_{C\cdot r} = (\mathcal{N}[\![\ e\ ]\!]_{\varrho|^\circ r})|_{C\cdot r}$
proof(*induction e arbitrary*: ϱ *r rule*: *exp-induct*)
  case (*Var x*)
  show *?case*
  proof (*rule C-restr-C-cong*)
    fix r'
    assume $r' \sqsubseteq r$
    have $(\mathcal{N}[\![\ Var\ x\ ]\!]_\varrho)\cdot(C\cdot r') = \varrho\ x\cdot r'$ **by** *simp*
    also have $\ldots = ((\varrho\ x)|_r)\cdot r'$ **using** ‹$r' \sqsubseteq r$› **by** *simp*
    also have $\ldots = (\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho|^\circ r})\cdot(C\cdot r')$ **by** *simp*
    finally show $(\mathcal{N}[\![\ Var\ x\ ]\!]_\varrho)\cdot(C\cdot r') = (\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho|^\circ r})\cdot(C\cdot r')$.
  qed *simp*
next
  case (*Lam x e*)
  show *?case*
  proof(*rule C-restr-C-cong*)
    fix r'
    assume $r' \sqsubseteq r$
    hence $r' \sqsubseteq C\cdot r$ **by** (*metis below-C below-trans*)
    {
      fix v
      have $\varrho(x := v)|^\circ r = (\varrho|^\circ r)(x := v)|^\circ r$
        by *simp*
      hence $(\mathcal{N}[\![\ e\ ]\!]_{\varrho(x := v)})|_{r'} = (\mathcal{N}[\![\ e\ ]\!]_{(\varrho|^\circ r)(x := v)})|_{r'}$
        by (*subst* (*1 2*) *C-restr-eq-lower*[*OF Lam* ‹$r' \sqsubseteq C\cdot r$›]) *simp*
    }
    thus $(\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_\varrho)\cdot(C\cdot r') = (\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho|^\circ r})\cdot(C\cdot r')$
      by *simp*
  qed *simp*
next
  case (*App e x*)
  show *?case*

proof (*rule C-restr-C-cong*)
  fix r'
  assume $r' \sqsubseteq r$
  hence $r' \sqsubseteq C \cdot r$ **by** (*metis below-C below-trans*)
  hence $(\mathcal{N}[\![e]\!]_{\varrho}) \cdot r' = (\mathcal{N}[\![e]\!]_{\varrho|^{\circ}r}) \cdot r'$
    by (*rule C-restr-eqD*[*OF App*])
  thus $(\mathcal{N}[\![App\ e\ x]\!]_{\varrho}) \cdot (C \cdot r') = (\mathcal{N}[\![App\ e\ x]\!]_{\varrho|^{\circ}r}) \cdot (C \cdot r')$
    using $\langle r' \sqsubseteq r \rangle$ **by** *simp*
qed *simp*
next
  case (*Bool b*)
  show *?case* **by** *simp*
next
  case (*IfThenElse scrut* e_1 e_2)
  show *?case*
  proof (*rule C-restr-C-cong*)
    fix r'
    assume $r' \sqsubseteq r$
    hence $r' \sqsubseteq C \cdot r$ **by** (*metis below-C below-trans*)

    have $(\mathcal{N}[\![scrut]\!]_{\varrho}) \cdot r' = (\mathcal{N}[\![scrut]\!]_{\varrho|^{\circ}r}) \cdot r'$
      using $\langle r' \sqsubseteq C \cdot r \rangle$ **by** (*rule C-restr-eqD*[*OF IfThenElse*(*1*)])
    moreover
    have $(\mathcal{N}[\![e_1]\!]_{\varrho}) \cdot r' = (\mathcal{N}[\![e_1]\!]_{\varrho|^{\circ}r}) \cdot r'$
      using $\langle r' \sqsubseteq C \cdot r \rangle$ **by** (*rule C-restr-eqD*[*OF IfThenElse*(*2*)])
    moreover
    have $(\mathcal{N}[\![e_2]\!]_{\varrho}) \cdot r' = (\mathcal{N}[\![e_2]\!]_{\varrho|^{\circ}r}) \cdot r'$
      using $\langle r' \sqsubseteq C \cdot r \rangle$ **by** (*rule C-restr-eqD*[*OF IfThenElse*(*3*)])
    ultimately
    show $(\mathcal{N}[\![(scrut\ ?\ e_1 : e_2)]\!]_{\varrho}) \cdot (C \cdot r') = (\mathcal{N}[\![(scrut\ ?\ e_1 : e_2)]\!]_{\varrho|^{\circ}r}) \cdot (C \cdot r')$
      using $\langle r' \sqsubseteq r \rangle$ **by** *simp*
  qed *simp*
next
  case (*Let* Γ *e*)

The lemma, lifted to heaps

  have *restr-can-restrict-env-heap* : $\bigwedge r.\ (\mathcal{N}\{\!|\Gamma|\!\}\varrho)|^{\circ}r = (\mathcal{N}\{\!|\Gamma|\!\}\varrho|^{\circ}r)|^{\circ}r$
  proof(*rule has-ESem.parallel-HSem-ind*)
    fix ϱ_1 ϱ_2 :: *CEnv* **and** r :: *C*
    assume $\varrho_1|^{\circ}r = \varrho_2|^{\circ}r$

    show $(\varrho ++_{domA\ \Gamma} \boldsymbol{\mathcal{N}}[\![\Gamma]\!]_{\varrho_1})|^{\circ}r = (\varrho|^{\circ}r ++_{domA\ \Gamma} \boldsymbol{\mathcal{N}}[\![\Gamma]\!]_{\varrho_2})|^{\circ}r$
    proof(*rule env-C-restr-cong*)
      fix x **and** r'
      assume $r' \sqsubseteq r$
      hence $r' \sqsubseteq C \cdot r$ **by** (*metis below-C below-trans*)

      show $(\varrho ++_{domA\ \Gamma} \boldsymbol{\mathcal{N}}[\![\Gamma]\!]_{\varrho_1})\ x \cdot r' = (\varrho|^{\circ}r ++_{domA\ \Gamma} \boldsymbol{\mathcal{N}}[\![\Gamma]\!]_{\varrho_2})\ x \cdot r'$

proof(*cases* $x \in$ *domA* Γ)
case *True*
have $(\mathcal{N}[\![$ *the* (*map-of* Γ x) $]\!]_{\varrho_1})\cdot r' = (\mathcal{N}[\![$ *the* (*map-of* Γ x) $]\!]_{\varrho_1|^\circ r})\cdot r'$
by (*rule C-restr-eqD*[*OF Let*(*1*)[*OF True*] $\langle r' \sqsubseteq C\cdot r\rangle$])
also have $\ldots = (\mathcal{N}[\![$ *the* (*map-of* Γ x) $]\!]_{\varrho_2|^\circ r})\cdot r'$
unfolding $\langle \varrho_1|^\circ r = \varrho_2|^\circ r\rangle$**..**
also have $\ldots = (\mathcal{N}[\![$ *the* (*map-of* Γ x) $]\!]_{\varrho_2})\cdot r'$
by (*rule C-restr-eqD*[*OF Let*(*1*)[*OF True*] $\langle r' \sqsubseteq C\cdot r\rangle$, *symmetric*])
finally
show *?thesis* **using** *True* **by** (*simp add*: *lookupEvalHeap*)
next
case *False*
with $\langle r' \sqsubseteq r\rangle$
show *?thesis* **by** *simp*
qed
qed
qed *simp-all*

show *?case*
proof (*rule C-restr-C-cong*)
fix r'
assume $r' \sqsubseteq r$
hence $r' \sqsubseteq C\cdot r$ **by** (*metis below-C below-trans*)

have $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)|^\circ r = (\mathcal{N}\{\!|\Gamma|\!\}(\varrho|^\circ r))|^\circ r$
by (*rule restr-can-restrict-env-heap*)
hence $(\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r' = (\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho|^\circ r})\cdot r'$
by (*subst* (*1 2*) *C-restr-eqD*[*OF Let*(*2*) $\langle r' \sqsubseteq C\cdot r\rangle$]) *simp*

thus $(\mathcal{N}[\![$ *Let* Γ e $]\!]_{\varrho})\cdot(C\cdot r') = (\mathcal{N}[\![$ *Let* Γ e $]\!]_{\varrho|^\circ r})\cdot(C\cdot r')$
using $\langle r' \sqsubseteq r\rangle$ **by** *simp*
qed *simp*
qed

lemma *can-restrict-env*:
$(\mathcal{N}[\![e]\!]_{\varrho})\cdot(C\cdot r) = (\mathcal{N}[\![\ e\ ]\!]_{\varrho|^\circ r})\cdot(C\cdot r)$
by (*rule C-restr-eqD*[*OF restr-can-restrict-env below-refl*])

When an expression e terminates, then we can remove such an expression from the heap and it still terminates. This is the crucial trick to handle black-holing in the resourced semantics.

lemma *add-BH*:
assumes *map-of* Γ x = *Some* e
assumes $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r' \neq \bot$
shows $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\textit{delete}\ x\ \Gamma|\!\}})\cdot r' \neq \bot$
proof−
obtain r **where** r: $C\cdot r$ = *demand* $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})$

using *demand-not-0* **by** (*cases demand* ($\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}}$)) *auto*

from *assms*(*2*)
have $C \cdot r \sqsubseteq r'$ **unfolding** *r not-bot-demand* **by** *simp*

from *assms*(*1*)
have [*simp*]: *the* (*map-of* Γ *x*) = *e* **by** (*metis option.sel*)

from *assms*(*1*)
have [*simp*]: $x \in$ *domA* Γ **by** (*metis domIff dom-map-of-conv-domA not-Some-eq*)

def $ub \equiv \mathcal{N}\{\!|\Gamma|\!\}$ — An upper bound for the induction

have *heaps*: $(\mathcal{N}\{\!|\Gamma|\!\})|^{\circ}{}_{r} \sqsubseteq \mathcal{N}\{\!|$*delete x* $\Gamma|\!\}$ **and** $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq ub$
proof (*induction rule*: *HSem-bot-ind*)
  fix ϱ
  assume $\varrho|^{\circ}{}_{r} \sqsubseteq \mathcal{N}\{\!|$*delete x* $\Gamma|\!\}$
  assume $\varrho \sqsubseteq ub$

  show $(\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho})|^{\circ}{}_{r} \sqsubseteq \mathcal{N}\{\!|$*delete x* $\Gamma|\!\}$
  proof (*rule fun-belowI*)
    fix *y*
    show $((\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho})|^{\circ}{}_{r})$ $y \sqsubseteq (\mathcal{N}\{\!|$*delete x* $\Gamma|\!\})$ *y*
    proof (*cases* $y = x$)
      case *True*
      have $((\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho})|^{\circ}{}_{r})$ $x = (\mathcal{N}[\![\ e\ ]\!]_{\varrho})|_{r}$
        by (*simp add*: *lookupEvalHeap*)
      also have ... $\sqsubseteq (\mathcal{N}[\![\ e\ ]\!]_{ub})|_{r}$
        using ‹$\varrho \sqsubseteq ub$› **by** (*intro monofun-cfun-arg*)
      also have ... $= (\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})|_{r}$
        unfolding *ub-def* **..**
      also have ... $= \bot$
        using *r* **by** (*rule C-restr-bot-demand*[*OF eq-imp-below*])
      also have ... $= (\mathcal{N}\{\!|$*delete x* $\Gamma|\!\})$ *x*
        by (*simp add*: *lookup-HSem-other*)
      finally
      show *?thesis* **unfolding** *True***.**
    next
      case *False*
      show *?thesis*
      proof (*cases* $y \in$ *domA* Γ)
        case *True*
        have $(\mathcal{N}[\![$ *the* (*map-of* Γ *y*) $]\!]_{\varrho})|_{r} = (\mathcal{N}[\![$ *the* (*map-of* Γ *y*) $]\!]_{\varrho|^{\circ}{}_{r}})|_{r}$
          by (*rule C-restr-eq-lower*[*OF restr-can-restrict-env below-C*])
        also have ... $\sqsubseteq \mathcal{N}[\![$ *the* (*map-of* Γ *y*) $]\!]_{\varrho|^{\circ}{}_{r}}$
          by (*rule C-restr-below*)
        also note ‹$\varrho|^{\circ}{}_{r} \sqsubseteq \mathcal{N}\{\!|$*delete x* $\Gamma|\!\}$›
        finally
        show *?thesis*

      using ‹$y \in domA\ \Gamma$› ‹$y \neq x$›
      by (*simp add*: *lookupEvalHeap lookup-HSem-heap*)
    next
      case *False*
      thus *?thesis* **by** *simp*
    qed
   qed
  qed

  from ‹$\varrho \sqsubseteq ub$›
  have $(\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho}) \sqsubseteq (\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{ub})$
   by (*rule cont2monofunE*[*rotated*]) *simp*
  also have $\ldots = ub$
   unfolding *ub-def HSem-bot-eq*[*symmetric*]**..**
  finally
  show $(\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho}) \sqsubseteq ub$**.**
 qed *simp-all*

 from *assms*(*2*)
 have $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot(C\cdot r) \neq \bot$
  unfolding *r*
  by (*rule demand-suffices*[*OF infinite-resources-suffice*])
 also
 have $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot(C\cdot r) = (\mathcal{N}[\![e]\!]_{(\mathcal{N}\{\!|\Gamma|\!\})|^{\circ}r})\cdot(C\cdot r)$
  by (*rule can-restrict-env*)
 also
 have $\ldots \sqsubseteq (\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}})\cdot(C\cdot r)$
  by (*intro monofun-cfun-arg monofun-cfun-fun heaps*)
 also
 have $\ldots \sqsubseteq (\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}})\cdot r'$
  using ‹$C\cdot r \sqsubseteq r'$› **by** (*rule monofun-cfun-arg*)
 finally
 show *?thesis* **by** *this* (*intro cont2cont*)+
qed

The semantics is continuous, so we can apply induction here:

lemma *resourced-adequacy*:
 assumes $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$
 shows $\exists\ \Delta\ v.\ \Gamma : e \Downarrow_{S} \Delta : v$
using *assms*
proof(*induction r arbitrary*: Γ *e S rule*: *C.induct*[*case-names adm bot step*])
 case *adm* **show** *?case* **by** *simp*
next
 case *bot*
 hence *False* **by** *auto*
 thus *?case***..**
next
 case (*step r*)

show *?case*
proof(*cases e rule*:*exp-strong-exhaust*(*1*)[**where** $c = (\Gamma,S)$, *case-names Var App Let Lam Bool IfThenElse*])
case (*Var x*)
let $?e = the\ (map\text{-}of\ \Gamma\ x)$
from *step.prems*[*unfolded Var*]
have $x \in domA\ \Gamma$
by (*auto intro*: *ccontr simp add*: *lookup-HSem-other*)
hence $map\text{-}of\ \Gamma\ x = Some\ ?e$ **by** (*rule domA-map-of-Some-the*)
moreover
from *step.prems*[*unfolded Var*] ‹$map\text{-}of\ \Gamma\ x = Some\ ?e$› ‹$x \in domA\ \Gamma$›
have $(\mathcal{N}[\![?e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$ **by** (*auto simp add*: *lookup-HSem-heap simp del*: *app-strict*)
hence $(\mathcal{N}[\![?e]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}})\cdot r \neq \bot$ **by** (*rule add-BH*[*OF* ‹$map\text{-}of\ \Gamma\ x = Some\ ?e$›])
from *step.IH*[*OF this*]
obtain $\Delta\ v$ **where** $delete\ x\ \Gamma : ?e \Downarrow_{x\ \#\ S} \Delta : v$ **by** *blast*
ultimately
have $\Gamma : (Var\ x) \Downarrow_S (x,v)\ \#\ \ \Delta : v$ **by** (*rule Variable*)
thus *?thesis* **using** *Var* **by** *auto*
next
case (*App* e' *x*)
have $finite\ (set\ S \cup fv\ (\Gamma,\ e'))$ **by** *simp*
from *finite-list*[*OF this*]
obtain S' **where** S': $set\ S' = set\ S \cup fv\ (\Gamma,\ e')$**..**

from *step.prems*[*unfolded App*]
have *prem*: $((\mathcal{N}[\![\ e'\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \downarrow CFn\ (\mathcal{N}\{\!|\Gamma|\!\})\ x|_r)\cdot r \neq \bot$ **by** (*auto simp del*: *app-strict*)
hence $(\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$ **by** *auto*
from *step.IH*[*OF this*]
obtain $\Delta\ v$ **where** *lhs'*: $\Gamma : e' \Downarrow_{S'} \Delta : v$ **by** *blast*

have $fv\ (\Gamma,\ e') \subseteq set\ S'$ **using** S' **by** *auto*
from *correctness-empty-env*[*OF lhs' this*]
have *correct1*: $\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!|\Gamma|\!\}} \sqsubseteq \mathcal{N}[\![v]\!]_{\mathcal{N}\{\!|\Delta|\!\}}$ **and** $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$ **by** *auto*

from *prem*
have $((\mathcal{N}[\![\ v\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r \downarrow CFn\ (\mathcal{N}\{\!|\Gamma|\!\})\ x|_r)\cdot r \neq \bot$
by (*rule not-bot-below-trans*)(*intro correct1 monofun-cfun-fun monofun-cfun-arg*)
with *result-evaluated*[*OF lhs'*]
have *isLam v* **by** (*cases r*, *auto*, *cases v rule*: *isVal.cases*, *auto*)
then obtain $y\ e''$ **where** n': $v = (Lam\ [y].\ e'')$ **by** (*rule isLam-obtain-fresh*)
with *lhs'*
have *lhs*: $\Gamma : e' \Downarrow_{S'} \Delta : Lam\ [y].\ e''$ **by** *simp*

have $((\mathcal{N}[\![\ v\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r \downarrow CFn\ (\mathcal{N}\{\!|\Gamma|\!\})\ x|_r)\cdot r \neq \bot$ **by** *fact*
also have $(\mathcal{N}\{\!|\Gamma|\!\})\ x|_r \sqsubseteq (\mathcal{N}\{\!|\Gamma|\!\})\ x$ **by** (*rule C-restr-below*)
also note ‹$v = \text{-}$›
also note ‹$(\mathcal{N}\{\!|\Gamma|\!\}) \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\})$›
also have $(\mathcal{N}[\![\ Lam\ [y].\ e''\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r \sqsubseteq CFn\cdot(\Lambda\ v.\ \mathcal{N}[\![e'']\!]_{(\mathcal{N}\{\!|\Delta|\!\})(y\ :=\ v)})$

by (*rule CELam-no-restr*)
also have $(\ldots \downarrow CFn\ (\mathcal{N}\{\!|\Delta|\!\})\ x)\cdot r = (\mathcal{N}[\![e']\!]_{(\mathcal{N}\{\!|\Delta|\!\})(y\ :=\ ((\mathcal{N}\{\!|\Delta|\!\})\ x))})\cdot r$ **by** *simp*
also have $\ldots = (\mathcal{N}[\![e''[y::=x]]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r$
unfolding *ESem-subst*..
finally
have $\ldots \neq \bot$ **by** *this* (*intro cont2cont cont-fun*)+
then
obtain $\Theta\ v'$ **where** *rhs*: $\Delta : e''[y::=x] \Downarrow_{S'} \Theta : v'$ **using** *step.IH* **by** *blast*

have $\Gamma : App\ e'\ x \Downarrow_{S'} \Theta : v'$
by (*rule reds-ApplicationI*[*OF lhs rhs*])
hence $\Gamma : App\ e'\ x \Downarrow_{S} \Theta : v'$
apply (*rule reds-smaller-L*) **using** S' **by** *auto*
thus *?thesis* **using** *App* **by** *auto*
next
case (*Lam v e'*)
have $\Gamma : Lam\ [v].\ e' \Downarrow_{S} \Gamma : Lam\ [v].\ e'$ ..
thus *?thesis* **using** *Lam* **by** *blast*
next
case (*Bool b*)
have $\Gamma : Bool\ b \Downarrow_{S} \Gamma : Bool\ b$ **by** *rule*
thus *?thesis* **using** *Bool* **by** *blast*
next
case (*IfThenElse scrut* e_1 e_2)

from *step.prems*[*unfolded IfThenElse*]
have *prem*: $CB\text{-}project\cdot((\mathcal{N}[\![\ scrut\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r)\cdot((\mathcal{N}[\![\ e_1\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r)\cdot((\mathcal{N}[\![\ e_2\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r) \neq \bot$
by (*auto simp del*: *app-strict*)
then obtain b **where**
is-CB: $(\mathcal{N}[\![\ scrut\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r = CB\cdot(Discr\ b)$
and *not-bot2*: $((\mathcal{N}[\![\ (if\ b\ then\ e_1\ else\ e_2)\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r) \neq \bot$
unfolding *CB-project-not-bot* **by** (*auto split*: *if-splits*)

have *finite* (*set* $S \cup$ *fv* (Γ, *scrut*)) **by** *simp*
from *finite-list*[*OF this*]
obtain S' **where** S': *set* S' = *set* $S \cup$ *fv* (Γ, *scrut*)..

from *is-CB* **have** $(\mathcal{N}[\![\ scrut\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$ **by** *simp*
from *step.IH*[*OF this*]
obtain $\Delta\ v$ **where** *lhs'*: $\Gamma : scrut \Downarrow_{S'} \Delta : v$ **by** *blast*
then have *isVal v* **by** (*rule result-evaluated*)

have *fv* (Γ, *scrut*) $\subseteq$ *set* S' **using** S' **by** *simp*
from *correctness-empty-env*[*OF lhs' this*]
have *correct1*: $\mathcal{N}[\![scrut]\!]_{\mathcal{N}\{\!|\Gamma|\!\}} \sqsubseteq \mathcal{N}[\![v]\!]_{\mathcal{N}\{\!|\Delta|\!\}}$ **and** *correct2*: $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$ **by** *auto*

from *correct1*

have $(\mathcal{N}[\![\ \mathit{scrut}\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \sqsubseteq (\mathcal{N}[\![\ v\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r$ **by** (*rule monofun-cfun-fun*)
with *is-CB*
have $(\mathcal{N}[\![\ v\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r = \mathit{CB}\cdot(\mathit{Discr}\ b)$ **by** *simp*
with ‹*isVal v*›
have $v = \mathit{Bool}\ b$ **by** (*cases v rule*: *isVal.cases*) (*case-tac r*, *auto*)+

from *not-bot2* ‹$\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$›
have $(\mathcal{N}[\![\ (\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2)\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r \neq \bot$
  by (*rule not-bot-below-trans*[*OF - monofun-cfun-fun*[*OF monofun-cfun-arg*]])
from *step.IH*[*OF this*]
obtain $\Theta\ v'$ **where** *rhs*: $\Delta : (\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2) \Downarrow_{S'} \Theta : v'$ **by** *blast*

from *lhs'*[*unfolded* ‹$v = \text{-}$›] *rhs*
have $\Gamma : (\mathit{scrut}\ ?\ e_1 : e_2) \Downarrow_{S'} \Theta : v'$ **by** *rule*
hence $\Gamma : (\mathit{scrut}\ ?\ e_1 : e_2) \Downarrow_{S} \Theta : v'$
  apply (*rule reds-smaller-L*) **using** S' **by** *auto*
thus *?thesis* **unfolding** *IfThenElse* **by** *blast*
next
case (*Let* Δ e')
from *step.prems*[*unfolded Let*(*2*)]
have *prem*: $(\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!|\Delta|\!\}\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$
  by (*simp del*: *app-strict*)
also
  have *atom* ‘ *domA* Δ $\sharp*$ Γ **using** *Let*(*1*) **by** (*simp add*: *fresh-star-Pair*)
  hence $\mathcal{N}\{\!|\Delta|\!\}\mathcal{N}\{\!|\Gamma|\!\} = \mathcal{N}\{\!|\Delta\ @\ \Gamma|\!\}$ **by** (*rule HSem-merge*)
finally
have $(\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!|\Delta\ @\ \Gamma|\!\}})\cdot r \neq \bot$**.**
then
obtain $\Theta\ v$ **where** $\Delta\ @\ \Gamma : e' \Downarrow_S \Theta : v$ **using** *step.IH* **by** *blast*
hence $\Gamma : \mathit{Let}\ \Delta\ e' \Downarrow_S \Theta : v$
  by (*rule reds.Let*[*OF Let*(*1*)])
thus *?thesis* **using** *Let* **by** *auto*
qed
qed

end

36 ValueSimilarity.tex

theory *ValueSimilarity*
imports *Value CValue Pointwise*
begin

This theory formalizes Section 3 of [SGHHOM11]. Their domain D is our type *Value*, their domain E is our type *CValue* and A corresponds to $C \to \mathit{CValue}$.

In our case, the construction of the domains was taken care of by the HOLCF package

([Huf12]), so where [SGHHOM11] refers to elements of the domain approximations D_n resp. E_n, these are just elements of *Value* resp. *CValue* here. Therefore the *n-injection* $\phi_n^E\colon E_n \to E$ is the identity here.

The projections correspond to the take-functions generated by the HOLCF package:

$\psi_n^E\colon E \to E_n$	corresponds to	$\mathit{CValue\text{-}take}::\mathit{nat} \Rightarrow \mathit{CValue} \to \mathit{CValue}$
$\psi_n^A\colon A \to A_n$	corresponds to	$\mathit{C\text{-}to\text{-}CValue\text{-}take}::\mathit{nat} \Rightarrow (C \to \mathit{CValue}) \to C \to \mathit{CValue}$
$\psi_n^D\colon D \to D_n$	corresponds to	$\mathit{Value\text{-}take}::\mathit{nat} \Rightarrow \mathit{Value} \to \mathit{Value}.$

The syntactic overloading of $e(a)(c)$ to mean either $\mathsf{Ap}^{\perp}_{E_n}$ or $\mathsf{AP}^{\perp}_{E}$ turns into our non-overloaded $\text{-} \downarrow \mathit{CFn}\ \text{-}::\mathit{CValue} \Rightarrow (C \to \mathit{CValue}) \Rightarrow C \to \mathit{CValue}$.

To have our presentation closer to [SGHHOM11], we introduce some notation:

notation *Value-take* ($\psi^D{}_{\text{-}}$)
notation *C-to-CValue-take* ($\psi^A{}_{\text{-}}$)
notation *CValue-take* ($\psi^E{}_{\text{-}}$)

36.1 A note about section 2.3

Section 2.3 of [SGHHOM11] contains equations (2) and (3) which do not hold in general. We demonstrate that fact here using our corresponding definition, but the counter-example carries over to the original formulation. Lemma (2) is a generalisation of (3) to the resourced semantics, so the counter-example for (3) is the simpler and more educating:

lemma *counter-example*:
  assumes *Equation (3)*: $\bigwedge n\ d\ d'.\ \psi^D{}_n\cdot(d \downarrow \mathit{Fn}\ d') = \psi^D{}_{\mathit{Suc}\ n}\cdot d \downarrow \mathit{Fn}\ \psi^D{}_n\cdot d'$
  shows *False*
proof−
  def $n == 1::\mathit{nat}$
  def $d == \mathit{Fn}\cdot(\Lambda\ e.\ (e \downarrow \mathit{Fn}\ \perp))$
  def $d' == \mathit{Fn}\cdot(\Lambda\ \text{-}.\ \mathit{Fn}\cdot(\Lambda\ \text{-}.\ \perp))$
  have $\mathit{Fn}\cdot(\Lambda\ \text{-}.\ \perp) = \psi^D{}_n\cdot(d \downarrow \mathit{Fn}\ d')$
    by (*simp add*: *d-def d'-def n-def cfun-map-def*)
  also
  have $\ldots = \psi^D{}_{\mathit{Suc}\ n}\cdot d \downarrow \mathit{Fn}\ \psi^D{}_n\cdot d'$
    using *Equation (3)*.
  also have $\ldots = \perp$
    by (*simp add*: *d-def d'-def n-def*)
  finally show *False* **by** *simp*
qed

For completeness, and to avoid making false assertions, the counter-example to equation (2):

lemma *counter-example2*:

assumes *Equation (2)*: $\bigwedge$ *n e a c*. $\psi^E{}_n\cdot((e \downarrow CFn\ a)\cdot c) = (\psi^E{}_{Suc\ n}\cdot e \downarrow CFn\ \psi^A{}_n\cdot a)\cdot c$
shows *False*
proof−
def $n == 1{::}nat$
def $e == CFn\cdot(\Lambda\ e\ r.\ (e\cdot r \downarrow CFn\ \bot)\cdot r)$
def $a == \Lambda\ \text{-}\ .\ CFn\cdot(\Lambda\ \text{-}\ \text{-}.\ CFn\cdot(\Lambda\ \text{-}\ \text{-}.\ \bot)) :: C \to CValue$
fix $c :: C$
have $CFn\cdot(\Lambda\ \text{-}\ \text{-}.\ \bot) = \psi^E{}_n\cdot((e \downarrow CFn\ a)\cdot c)$
by (*simp add*: *e-def a-def n-def cfun-map-def*)
also
have $\ldots = (\psi^E{}_{Suc\ n}\cdot e \downarrow CFn\ \psi^A{}_n\cdot a)\cdot c$
using *Equation (2)*.
also have $\ldots = \bot$
by (*simp add*: *e-def a-def n-def*)
finally show *False* **by** *simp*
qed

A suitable substitute for the lemma can be found in 4.3.5 (1) in [AO93], which in our setting becomes the following (note the extra invocation of $\psi^D{}_n$ on the left hand side):

lemma *Abramsky 4,3,5 (1)*:
$\psi^D{}_n\cdot(d \downarrow Fn\ \psi^D{}_n\cdot d') = \psi^D{}_{Suc\ n}\cdot d \downarrow Fn\ \psi^D{}_n\cdot d'$
by (*cases d*) (*auto simp add*: *Value.take-take*)

The problematic equations are used in the proof of the only-if direction of proposition 9 in [SGHHOM11]. It can be fixed by applying take-induction, which inserts the extra call to $\psi^D{}_n$ in the right spot.

36.2 Working with *Value* and *CValue*

Combined case distinguishing and induction rules.

lemma *value-CValue-cases*:
obtains
$x = \bot\ y = \bot$ |
f **where** $x = Fn\cdot f\ y = \bot$ |
g **where** $x = \bot\ y = CFn\cdot g$ |
$f\ g$ **where** $x = Fn\cdot f\ y = CFn \cdot g$ |
b_1 **where** $x = B\cdot(Discr\ b_1)\ y = \bot$ |
$b_1\ g$ **where** $x = B\cdot(Discr\ b_1)\ y = CFn\cdot g$ |
$b_1\ b_2$ **where** $x = B\cdot(Discr\ b_1)\ y = CB\cdot(Discr\ b_2)$ |
$f\ b_2$ **where** $x = Fn\cdot f\ y = CB\cdot(Discr\ b_2)$ |
b_2 **where** $x = \bot\ y = CB\cdot(Discr\ b_2)$
by (*metis CValue.exhaust Discr-undiscr Value.exhaust*)

lemma *Value-CValue-take-induct*:
assumes *adm* (*case-prod P*)
assumes $\bigwedge$ *n*. $P\ (\psi^D{}_n\cdot x)\ (\psi^A{}_n\cdot y)$
shows *P x y*

proof−
  have *case-prod P* ($\bigsqcup n.\ (\psi^D{}_n \cdot x,\ \psi^A{}_n \cdot y)$)
  by (*rule admD*[*OF* ‹*adm* (*case-prod P*)› *ch2ch-Pair*[*OF ch2ch-Rep-cfunL*[*OF Value.chain-take*] *ch2ch-Rep-cfunL*[*OF C-to-CValue-chain-take*]]])
      (*simp add*: *assms*(*2*))
  hence *case-prod P* (*x*,*y*)
    by (*simp add*: *lub-Pair*[*OF ch2ch-Rep-cfunL*[*OF Value.chain-take*] *ch2ch-Rep-cfunL*[*OF C-to-CValue-chain-take*]]
            Value.reach C-to-CValue-reach)
  thus *?thesis* **by** *simp*
qed

36.3 Restricted similarity is defined recursively

The base case

inductive *similar′-base* :: *Value* ⇒ *CValue* ⇒ *bool* **where**
  bot-similar′-base[*simp*,*intro*]: *similar′-base* ⊥ ⊥

inductive-cases [*elim*!]:
  similar′-base x y

The inductive case

inductive *similar′-step* :: (*Value* ⇒ *CValue* ⇒ *bool*) ⇒ *Value* ⇒ *CValue* ⇒ *bool* **for** *s* **where**
  bot-similar′-step[*intro*!]: *similar′-step s* ⊥ ⊥ |
  bool-similar′-step[*intro*]: *similar′-step s* (*B*·*b*) (*CB*·*b*) |
  Fun-similar′-step[*intro*]: ($\bigwedge x\ y\ .\ s\ x\ (y \cdot C^\infty) \Longrightarrow s\ (f \cdot x)\ (g \cdot y \cdot C^\infty)$) ⟹ *similar′-step s* (*Fn*·*f*) (*CFn*·*g*)

inductive-cases [*elim*!]:
  similar′-step s x ⊥
  similar′-step s ⊥ *y*
  similar′-step s (*B*·*f*) (*CB*·*g*)
  similar′-step s (*Fn*·*f*) (*CFn*·*g*)

We now create the restricted similarity relation, by primitive recursion over n.

This cannot be done using an inductive definition, as it would not be monotone.

fun *similar′* **where**
 similar′ 0 = *similar′-base* |
 similar′ (*Suc n*) = *similar′-step* (*similar′ n*)
declare *similar′.simps*[*simp del*]

abbreviation *similar′-syn* (- ◁▷- - [*50*,*50*,*50*] *50*)
  where *similar′-syn x n y* ≡ *similar′ n x y*

lemma *similar′-botI*[*intro*!,*simp*]: $\bot \mathrel{◁▷_n} \bot$
  by (*cases n*) (*auto simp add*: *similar′.simps*)

lemma *similar′-FnI*[*intro*]:
  assumes $\bigwedge x\ y.\ \ x \lhd\!\rhd_n y\cdot C^\infty \Longrightarrow f\cdot x \lhd\!\rhd_n g\cdot y\cdot C^\infty$
  shows $Fn\cdot f \lhd\!\rhd_{Suc\ n}\ CFn\cdot g$
using *assms* **by** (*auto simp add*: *similar′.simps*)

lemma *similar′-FnE*[*elim*!]:
  assumes $Fn\cdot f \lhd\!\rhd_{Suc\ n}\ CFn\cdot g$
  assumes $(\bigwedge x\ y.\ \ x \lhd\!\rhd_n y\cdot C^\infty \Longrightarrow f\cdot x \lhd\!\rhd_n g\cdot y\cdot C^\infty) \Longrightarrow P$
  shows P
using *assms* **by** (*auto simp add*: *similar′.simps*)

lemma *bot-or-not-bot′*:
  $x \lhd\!\rhd_n y \Longrightarrow (x = \bot \longleftrightarrow y = \bot)$
  by (*cases n*) (*auto simp add*: *similar′.simps elim*: *similar′-base.cases similar′-step.cases*)

lemma *similar′-bot*[*elim-format*, *elim*!]:
    $\bot \lhd\!\rhd_n x \Longrightarrow x = \bot$
    $y \lhd\!\rhd_n \bot \Longrightarrow y = \bot$
by (*metis bot-or-not-bot′*)+

lemma *similar′-typed*[*simp*]:
  $\neg\ B\cdot b \lhd\!\rhd_n CFn\cdot g$
  $\neg\ Fn\cdot f \lhd\!\rhd_n CB\cdot b$
  by (*cases n*, *auto simp add*: *similar′.simps elim*: *similar′-base.cases similar′-step.cases*)+

lemma *similar′-bool*[*simp*]:
  $B\cdot b_1 \lhd\!\rhd_{Suc\ n}\ CB\cdot b_2 \longleftrightarrow b_1 = b_2$
  by (*auto simp add*: *similar′.simps elim*: *similar′-base.cases similar′-step.cases*)

36.4 Moving up and down the similarity relations

These correspond to Lemma 7 in [SGHHOM11].

lemma *similar′-down*: $d \lhd\!\rhd_{Suc\ n}\ e \Longrightarrow \psi^D{}_n\cdot d \lhd\!\rhd_n \psi^E{}_n\cdot e$
  and *similar′-up*: $d \lhd\!\rhd_n e \Longrightarrow \psi^D{}_n\cdot d \lhd\!\rhd_{Suc\ n}\ \psi^E{}_n\cdot e$
proof (*induction n arbitrary*: *d e*)
  case (*Suc n*) **case** *1* **with** *Suc*
  show *?case*
    by (*cases d e rule*:*value-CValue-cases*) *auto*
next
  case (*Suc n*) **case** *2* **with** *Suc*
  show *?case*
    by (*cases d e rule*:*value-CValue-cases*) *auto*
qed *auto*

A generalisation of the above, doing multiple steps at once.

lemma *similar′-up-le*: $n \le m \Longrightarrow \psi^D{}_n\cdot d \lhd\!\rhd_n \psi^E{}_n\cdot e \Longrightarrow \psi^D{}_n\cdot d \lhd\!\rhd_m \psi^E{}_n\cdot e$
  by (*induction rule*: *dec-induct*)

(*auto dest*: *similar′-up simp add*: *Value.take-take CValue.take-take min-absorb2*)

lemma *similar′-down-le*: $n \leq m \Longrightarrow \psi^D{}_m{\cdot}d \lhd\!\rhd_m \psi^E{}_m{\cdot}e \Longrightarrow \psi^D{}_n{\cdot}d \lhd\!\rhd_n \psi^E{}_n{\cdot}e$
by (*induction rule*: *inc-induct*)
(*auto dest*: *similar′-down simp add*: *Value.take-take CValue.take-take min-absorb1*)

lemma *similar′-take*: $d \lhd\!\rhd_n e \Longrightarrow \psi^D{}_n{\cdot}d \lhd\!\rhd_n \psi^E{}_n{\cdot}e$
apply (*drule similar′-up*)
apply (*drule similar′-down*)
apply (*simp add*: *Value.take-take CValue.take-take*)
done

36.5 Admissibility

A technical prerequisite for induction is admissibility of the predicate, i.e. that the predicate holds for the limit of a chain, given that it holds for all elements.

lemma *similar′-base-adm*: *adm* (λ *x*. *similar′-base* (*fst x*) (*snd x*))
proof (*rule admI*, *goal-cases*)
case (*1 Y*)
then have $Y = (\lambda$ - . $\bot)$ **by** (*metis prod.exhaust fst-eqD inst-prod-pcpo similar′-base.simps snd-eqD*)
thus *?case* **by** *auto*
qed

lemma *similar′-step-adm*:
assumes *adm* (λ *x*. *s* (*fst x*) (*snd x*))
shows *adm* (λ *x*. *similar′-step s* (*fst x*) (*snd x*))
proof (*rule admI*, *goal-cases*)
case *prems*: (*1 Y*)
from ‹*chain Y*›
have *chain* (λ *i*. *fst* (*Y i*)) **by** (*rule ch2ch-fst*)
thus *?case*
proof(*cases rule*: *Value-chainE*)
case *bot*
hence $*$: $\bigwedge$ *i*. *fst* (*Y i*) $= \bot$ **by** *metis*
with *prems*(*2*)[*unfolded split-beta*]
have $\bigwedge$*i*. *snd* (*Y i*) $= \bot$ **by** *auto*
hence $Y = (\lambda$ *i*. $(\bot, \bot))$ **using** $*$ **by** (*metis surjective-pairing*)
thus *?thesis* **by** *auto*
next
case (*B n b*)
hence $\forall$ *i*. *fst* (*Y* (*i* + *n*)) = *B*$\cdot$*b* **by** (*metis add.commute not-add-less1*)
with *prems*(*2*)
have $\forall$ *i*. *Y* (*i* + *n*) = (*B*$\cdot$*b*, *CB*$\cdot$*b*)
apply *auto*
apply (*erule-tac x* = *i* + *n* **in** *allE*)
apply (*erule-tac x* = *i* **in** *allE*)
apply (*erule similar′-step.cases*)

apply *auto*
by (*metis fst-conv old.prod.exhaust snd-conv*)
hence *similar′-step s* (*fst* ($\bigsqcup$ *i. Y* (*i* + *n*))) (*snd* ($\bigsqcup$ *i. Y* (*i* + *n*))) **by** *auto*
thus *?thesis*
by (*simp add*: *lub-range-shift*[*OF* ‹*chain Y*›])
next
fix *n*
fix *Y′*
assume *chain Y′* **and** (λ*i. fst* (*Y i*)) = (λ *m.* (*if m* < *n then* ⊥ *else Fn*·(*Y′* (*m*−*n*))))
hence *Y′*: $\bigwedge$ *i. fst* (*Y* (*i*+*n*)) = *Fn*·(*Y′ i*) **by** (*metis add-diff-cancel-right′ not-add-less2*)
with *prems*(*2*)[*unfolded split-beta*]
have $\bigwedge$*i.* ∃ *g′. snd* (*Y* (*i*+*n*)) = *CFn*·*g′*
by −(*erule-tac x* = *i* + *n* **in** *allE*, *auto elim*!: *similar′-step.cases*)
then obtain *Y″* **where** *Y″*: $\bigwedge$ *i. snd* (*Y* (*i*+*n*)) = *CFn*·(*Y″ i*) **by** *metis*
from *prems*(*1*) **have** $\bigwedge$*i. Y i* ⊑ *Y* (*Suc i*)
by (*simp add*: *po-class.chain-def*)
then have ∗: $\bigwedge$*i. Y* (*i* + *n*) ⊑ *Y* (*Suc i* + *n*)
by *simp*
have *chain Y″*
apply (*rule chainI*)
apply (*rule iffD1*[*OF CValue.inverts*(*1*)])
apply (*subst* (*1 2*) *Y″*[*symmetric*])
apply (*rule snd-monofun*)
apply (*rule* ∗)
done

have *similar′-step s* (*Fn*·($\bigsqcup$ *i.* (*Y′ i*))) (*CFn* · ($\bigsqcup$ *i. Y″ i*))
proof (*rule Fun-similar′-step*)
fix *x y*
from *prems*(*2*) *Y′ Y″*
have $\bigwedge$*i. similar′-step s* (*Fn*·(*Y′ i*)) (*CFn*·(*Y″ i*)) **by** *metis*
moreover
assume *s x* (*y*·C^∞)
ultimately
have $\bigwedge$*i. s* (*Y′ i*·*x*) (*Y″ i*·*y*·C^∞) **by** *auto*
hence *case-prod s* ($\bigsqcup$ *i.* ((*Y′ i*)·*x*, (*Y″ i*)·*y*·C^∞))
apply −
apply (*rule admD*[*OF adm-case-prod*[**where** *P* = λ- . *s*, *OF assms*]])
apply (*simp add*: ‹*chain Y′*› ‹*chain Y″*›)
apply *simp*
done
thus *s* (($\bigsqcup$ *i. Y′ i*)·*x*) (($\bigsqcup$ *i. Y″ i*)·*y*·C^∞)
by (*simp add*: *lub-Pair ch2ch-Rep-cfunL contlub-cfun-fun* ‹*chain Y′*› ‹*chain Y″*›)
qed
hence *similar′-step s* (*fst* ($\bigsqcup$ *i. Y* (*i*+*n*))) (*snd* ($\bigsqcup$ *i. Y* (*i*+*n*)))
by (*simp add*: *Y′ Y″*
cont2contlubE[*OF cont-fst chain-shift*[*OF prems*(*1*)]] *cont2contlubE*[*OF cont-snd chain-shift*[*OF prems*(*1*)]]
contlub-cfun-arg[*OF* ‹*chain Y″*›] *contlub-cfun-arg*[*OF* ‹*chain Y′*›])

thus *similar′-step s* (*fst* ($\bigsqcup$ *i*. *Y i*)) (*snd* ($\bigsqcup$ *i*. *Y i*))
by (*simp add*: *lub-range-shift*[*OF* ‹*chain Y*›])
qed
qed

lemma *similar′-adm*: *adm* ($\lambda x.$ *fst x* $\lhd\!\rhd_n$ *snd x*)
by (*induct n*) (*auto simp add*: *similar′.simps intro*: *similar′-base-adm similar′-step-adm*)

lemma *similar′-admI*: *cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *adm* ($\lambda x.$ *f x* $\lhd\!\rhd_n$ *g x*)
by (*rule adm-subst*[*OF - similar′-adm*, **where** $t = \lambda x.$ (*f x*, *g x*), *simplified*]) *auto*

36.6 The real similarity relation

This is the goal of the theory: A relation between *Value* and *CValue*.

definition *similar* :: *Value* $\Rightarrow$ *CValue* $\Rightarrow$ *bool* (**infix** $\lhd\!\rhd$ *50*) **where**
$x \lhd\!\rhd y \longleftrightarrow (\forall n.\ \psi^D{}_n \cdot x \lhd\!\rhd_n \psi^E{}_n \cdot y)$

lemma *similarI*:
$(\bigwedge n.\ \psi^D{}_n \cdot x \lhd\!\rhd_n \psi^E{}_n \cdot y) \Longrightarrow x \lhd\!\rhd y$
unfolding *similar-def* **by** *blast*

lemma *similarE*:
$x \lhd\!\rhd y \Longrightarrow \psi^D{}_n \cdot x \lhd\!\rhd_n \psi^E{}_n \cdot y$
unfolding *similar-def* **by** *blast*

lemma *similar-bot*[*simp*]: $\bot \lhd\!\rhd \bot$ **by** (*auto intro*: *similarI*)

lemma *similar-bool*[*simp*]: $B \cdot b \lhd\!\rhd CB \cdot b$
by (*rule similarI*, *case-tac n*, *auto*)

lemma [*elim-format*, *elim*!]: $x \lhd\!\rhd \bot \Longrightarrow x = \bot$
unfolding *similar-def*
apply (*cases x*)
apply *auto*
apply (*erule-tac x* = *Suc 0* **in** *allE*, *auto*)+
done

lemma [*elim-format*, *elim*!]: $x \lhd\!\rhd CB \cdot b \Longrightarrow x = B \cdot b$
unfolding *similar-def*
apply (*cases x*)
apply *auto*
apply (*erule-tac x* = *Suc 0* **in** *allE*, *auto*)+
done

lemma [*elim-format*, *elim*!]: $\bot \lhd\!\rhd y \Longrightarrow y = \bot$
unfolding *similar-def*
apply (*cases y*)

apply *auto*
apply (*erule-tac* $x = Suc\ 0$ **in** *allE*, *auto*)+
done

lemma [*elim-format*, *elim*!]: $B{\cdot}b \lhd\!\rhd y \Longrightarrow y = CB{\cdot}b$
unfolding *similar-def*
apply (*cases* y)
apply *auto*
apply (*erule-tac* $x = Suc\ 0$ **in** *allE*, *auto*)+
done

lemma *take-similar'-similar*:
assumes $x \lhd\!\rhd_n y$
shows $\psi^D{}_n{\cdot}x \lhd\!\rhd \psi^E{}_n{\cdot}y$
proof(*rule similarI*)
fix m
from *assms*
have $\psi^D{}_n{\cdot}x \lhd\!\rhd_n \psi^E{}_n{\cdot}y$ **by** (*rule similar'-take*)
moreover
have $n \leq m \vee m \leq n$ **by** *auto*
ultimately
show $\psi^D{}_m{\cdot}(\psi^D{}_n{\cdot}x) \lhd\!\rhd_m \psi^E{}_m{\cdot}(\psi^E{}_n{\cdot}y)$
by (*auto elim*: *similar'-up-le similar'-down-le dest*: *similar'-take*
simp add: *min-absorb2 min-absorb1 Value.take-take CValue.take-take*)
qed

lemma *bot-or-not-bot*:
$x \lhd\!\rhd y \Longrightarrow (x = \bot \longleftrightarrow y = \bot)$
by (*cases x y rule*:*value-CValue-cases*) *auto*

lemma *bool-or-not-bool*:
$x \lhd\!\rhd y \Longrightarrow (x = B{\cdot}b) \longleftrightarrow (y = CB{\cdot}b)$
by (*cases x y rule*:*value-CValue-cases*) *auto*

lemma *slimilar-bot-cases*[*consumes 1*, *case-names bot bool Fn*]:
assumes $x \lhd\!\rhd y$
obtains $x = \bot$ $y = \bot$ |
b **where** $x = B{\cdot}(Discr\ b)$ $y = CB{\cdot}(Discr\ b)$ |
f g **where** $x = Fn{\cdot}f$ $y = CFn \cdot g$
using *assms*
by (*metis CValue.exhaust Value.exhaust bool-or-not-bool bot-or-not-bot discr.exhaust*)

lemma *similar-adm*: *adm* ($\lambda x.$ *fst* $x \lhd\!\rhd$ *snd* x)
unfolding *similar-def*
by (*intro adm-lemmas similar'-admI cont2cont*)

lemma *similar-admI*: *cont* $f \Longrightarrow$ *cont* $g \Longrightarrow$ *adm* ($\lambda x.\ f\ x \lhd\!\rhd g\ x$)
by (*rule adm-subst*[*OF - similar-adm*, **where** $t = \lambda x.\ (f\ x,\ g\ x)$, *simplified*]) *auto*

Having constructed the relation we can how show that it indeed is the desired relation, relating $\bot$ with $\bot$ and functions with functions, if they take related arguments to related values. This corresponds to Proposition 9 in [SGHHOM11].

lemma *similar-nice-def*: $x \lhd\rhd y \longleftrightarrow (x = \bot \wedge y = \bot \vee (\exists\ b.\ x = B\cdot(Discr\ b) \wedge y = CB\cdot(Discr\ b)) \vee (\exists\ f\ g.\ x = Fn\cdot f \wedge y = CFn\cdot g \wedge (\forall\ a\ b.\ a \lhd\rhd b\cdot C^\infty \longrightarrow f\cdot a \lhd\rhd g\cdot b\cdot C^\infty)))$
(**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
thus *?R*
proof (*cases x y rule*:*slimilar-bot-cases*)
case *bot* **thus** *?thesis* **by** *simp*
next
case *bool* **thus** *?thesis* **by** *simp*
next
case (*Fn f g*)
note ‹*?L*›[*unfolded Fn*]
have $\forall\ a\ b.\ a \lhd\rhd b\cdot C^\infty \longrightarrow f\cdot a \lhd\rhd g\cdot b\cdot C^\infty$
proof(*intro impI allI*)
fix *a b*
assume $a \lhd\rhd b\cdot C^\infty$

show $f\cdot a \lhd\rhd g\cdot b\cdot C^\infty$
proof(*rule similarI*)
fix *n*
have *adm* $(\lambda(b,\ a).\ \psi^D{}_n\cdot(f\cdot b) \lhd\rhd_n \psi^E{}_n\cdot(g\cdot a\cdot C^\infty))$
by (*intro adm-case-prod similar′-admI cont2cont*)
thus $\psi^D{}_n\cdot(f\cdot a) \lhd\rhd_n \psi^E{}_n\cdot(g\cdot b\cdot C^\infty)$
proof (*induct a b rule*: *Value-CValue-take-induct*[*consumes 1*])

This take induction is required to avoid the wrong equation shown above.

fix *m*

from ‹$a \lhd\rhd b\cdot C^\infty$›
have $\psi^D{}_m\cdot a \lhd\rhd_m \psi^E{}_m\cdot(b\cdot C^\infty)$ **by** (*rule similarE*)
hence $\psi^D{}_m\cdot a \lhd\rhd_{max\ m\ n} \psi^E{}_m\cdot(b\cdot C^\infty)$ **by** (*rule similar′-up-le*[*rotated*]) *auto*
moreover
from ‹$Fn\cdot f \lhd\rhd CFn\cdot g$›
have $\psi^D{}_{Suc\ (max\ m\ n)}\cdot(Fn\cdot f) \lhd\rhd_{Suc\ (max\ m\ n)} \psi^E{}_{Suc\ (max\ m\ n)}\cdot(CFn\cdot g)$ **by** (*rule similarE*)
ultimately
have $\psi^D{}_{max\ m\ n}\cdot(f\cdot(\psi^D{}_{max\ m\ n}\cdot(\psi^D{}_m\cdot a))) \lhd\rhd_{max\ m\ n} \psi^E{}_{max\ m\ n}\cdot(g\cdot(\psi^A{}_{max\ m\ n}\cdot(\psi^A{}_m\cdot b))\cdot C^\infty)$
by *auto*
hence $\psi^D{}_{max\ m\ n}\cdot(f\cdot(\psi^D{}_m\cdot a)) \lhd\rhd_{max\ m\ n} \psi^E{}_{max\ m\ n}\cdot(g\cdot(\psi^A{}_m\cdot b)\cdot C^\infty)$
by (*simp add*: *Value.take-take cfun-map-map CValue.take-take ID-def eta-cfun min-absorb2 min-absorb1*)
thus $\psi^D{}_n\cdot(f\cdot(\psi^D{}_m\cdot a)) \lhd\rhd_n \psi^E{}_n\cdot(g\cdot(\psi^A{}_m\cdot b)\cdot C^\infty)$
by (*rule similar′-down-le*[*rotated*]) *auto*
qed

qed
qed
thus *?thesis* **unfolding** *Fn* **by** *simp*
qed
next
assume *?R*
thus *?L*
proof(*elim conjE disjE exE ssubst*)
show $\bot \lhd\!\rhd \bot$ **by** *simp*
next
fix *b*
show $B{\cdot}(Discr\ b) \lhd\!\rhd CB{\cdot}(Discr\ b)$ **by** *simp*
next
fix *f g*
assume *imp*: $\forall\, a\ b.\ a \lhd\!\rhd b{\cdot}C^{\infty} \longrightarrow f{\cdot}a \lhd\!\rhd g{\cdot}b{\cdot}C^{\infty}$
show $Fn{\cdot}f \lhd\!\rhd CFn{\cdot}g$
proof (*rule similarI*)
fix *n*
show $\psi^D{}_n{\cdot}(Fn{\cdot}f) \lhd\!\rhd_n \psi^E{}_n{\cdot}(CFn{\cdot}g)$
proof(*cases n*)
case *0* **thus** *?thesis* **by** *simp*
next
case (*Suc n*)
{ fix *x y*
assume $x \lhd\!\rhd_n y{\cdot}C^{\infty}$
hence $\psi^D{}_n{\cdot}x \lhd\!\rhd \psi^E{}_n{\cdot}(y{\cdot}C^{\infty})$ **by** (*rule take-similar′-similar*)
hence $f{\cdot}(\psi^D{}_n{\cdot}x) \lhd\!\rhd g{\cdot}(\psi^A{}_n{\cdot}y){\cdot}C^{\infty}$ **using** *imp* **by** *auto*
hence $\psi^D{}_n{\cdot}(f{\cdot}(\psi^D{}_n{\cdot}x)) \lhd\!\rhd_n \psi^E{}_n{\cdot}(g{\cdot}(\psi^A{}_n{\cdot}y){\cdot}C^{\infty})$
by (*rule similarE*)
}
with *Suc*
show *?thesis* **by** *auto*
qed
qed
qed
qed

lemma *similar-FnI*[*intro*]:
assumes $\bigwedge x\ y.\ \ x \lhd\!\rhd y{\cdot}C^{\infty} \Longrightarrow f{\cdot}x \lhd\!\rhd g{\cdot}y{\cdot}C^{\infty}$
shows $Fn{\cdot}f \lhd\!\rhd CFn{\cdot}g$
by (*metis assms similar-nice-def*)

lemma *similar-FnD*[*elim*!]:
assumes $Fn{\cdot}f \lhd\!\rhd CFn{\cdot}g$
assumes $x \lhd\!\rhd y{\cdot}C^{\infty}$
shows $f{\cdot}x \lhd\!\rhd g{\cdot}y{\cdot}C^{\infty}$
using *assms*
by (*subst* (*asm*) *similar-nice-def*) *auto*

lemma *similar-FnE*[*elim*!]:
  assumes $Fn \cdot f \triangleleft\triangleright CFn \cdot g$
  assumes $(\bigwedge x\ y.\ \ x \triangleleft\triangleright y \cdot C^\infty \Longrightarrow f \cdot x \triangleleft\triangleright g \cdot y \cdot C^\infty) \Longrightarrow P$
  shows P
by (*metis assms similar-FnD*)

36.7 The similarity relation lifted pointwise to functions.

abbreviation *fun-similar* :: $('a{::}type \Rightarrow Value) \Rightarrow ('a \Rightarrow (C \to CValue)) \Rightarrow bool$ (**infix** $\triangleleft\triangleright^*$ *50*) **where**
  $\mathit{fun\text{-}similar} \equiv \mathit{pointwise}\ (\lambda x\ y.\ x \triangleleft\triangleright y \cdot C^\infty)$

lemma *fun-similar-fmap-bottom*[*simp*]: $\bot \triangleleft\triangleright^* \bot$
  by *auto*

lemma *fun-similarE*[*elim*]:
  assumes $m \triangleleft\triangleright^* m'$
  assumes $(\bigwedge x.\ (m\ \ x) \triangleleft\triangleright (m'\ x) \cdot C^\infty) \Longrightarrow Q$
  shows Q
  using *assms* **unfolding** *pointwise-def* **by** *blast*

end

37 Denotational-Related.tex

theory *Denotational−Related*
imports *Denotational ResourcedDenotational ValueSimilarity*
begin

Given the similarity relation it is straight-forward to prove that the standard and the resourced denotational semantics produce similar results. (Theorem 10 in [SGHHOM11]).

theorem *denotational-semantics-similar*:
  assumes $\varrho \triangleleft\triangleright^* \sigma$
  shows $[\![e]\!]_\varrho \triangleleft\triangleright (\mathcal{N}[\![e]\!]_\sigma) \cdot C^\infty$
using *assms*
proof(*induct e arbitrary*: $\varrho\ \sigma$ *rule*:*exp-induct*)
  case (*Var v*)
  from *Var* **have** $\varrho\ v \triangleleft\triangleright (\sigma\ \ v) \cdot C^\infty$ **by** *cases auto*
  thus *?case* **by** *simp*
next
  case (*Lam v e*)
  { **fix** *x y*
    assume $x \triangleleft\triangleright y \cdot C^\infty$
    with ‹$\varrho \triangleleft\triangleright^* \sigma$›
    have $\varrho(v := x) \triangleleft\triangleright^* \sigma(v := y)$
      by (*auto 1 4*)
    hence $[\![e]\!]_{\varrho(v\ :=\ x)} \triangleleft\triangleright (\mathcal{N}[\![e]\!]_{\sigma(v\ :=\ y)}) \cdot C^\infty$

```
      by (rule Lam.hyps)
    }
    thus ?case by auto
next
  case (App e v ϱ σ)
  hence App': ⟦e⟧ϱ ◃▹ (𝒩⟦e⟧σ)·C∞ by auto
  thus ?case
  proof (cases rule: slimilar-bot-cases)
    case (Fn f g)
    from ‹ϱ ◃▹* σ›
    have ϱ v ◃▹ (σ v)·C∞  by auto
    thus ?thesis using Fn App' by auto
  qed auto
next
  case (Bool b)
  thus ⟦Bool b⟧ϱ ◃▹ (𝒩⟦Bool b⟧σ)·C∞ by auto
next
  case (IfThenElse scrut e1 e2)
  hence IfThenElse':
    ⟦ scrut ⟧ϱ ◃▹ (𝒩⟦ scrut ⟧σ)·C∞
    ⟦ e1 ⟧ϱ ◃▹ (𝒩⟦ e1 ⟧σ)·C∞
    ⟦ e2 ⟧ϱ ◃▹ (𝒩⟦ e2 ⟧σ)·C∞ by auto
  from IfThenElse'(1)
  show ?case
  proof (cases rule: slimilar-bot-cases)
    case (bool b)
    thus ?thesis using IfThenElse' by auto
  qed auto
next
  case (Let as e ϱ σ)
  have ⦃as⦄ϱ ◃▹* 𝒩⦃as⦄σ
  proof (rule parallel-HSem-ind-different-ESem[OF pointwise-adm[OF similar-admI] fun-similar-fmap-bottom])
    fix ϱ' :: var ⇒ Value and σ' :: var ⇒ C → CValue
    assume ϱ' ◃▹* σ'
    show ϱ ++domA as ⟦ as ⟧ϱ' ◃▹* σ ++domA as evalHeap as (λe. 𝒩⟦ e ⟧σ')
    proof(rule pointwiseI, goal-cases)
      case (1 x)
      show ?case using ‹ϱ ◃▹* σ›
        by (auto simp add: lookup-override-on-eq lookupEvalHeap elim: Let(1)[OF - ‹ϱ' ◃▹* σ'›]
)
    qed
  qed auto
  hence ⟦e⟧⦃as⦄ϱ ◃▹ (𝒩⟦e⟧𝒩⦃as⦄σ)·C∞ by (rule Let(2))
  thus ?case by simp
qed

corollary evalHeap-similar:
  ⋀y z. y ◃▹* z ⟹ ⟦ Γ ⟧y ◃▹* 𝓝⟦ Γ ⟧z
  by (rule pointwiseI)
```

(*case-tac* $x \in$ *domA* Γ, *auto simp add*: *lookupEvalHeap denotational-semantics-similar*)

theorem *heaps-similar*: $\{\!|\Gamma|\!\} \triangleleft\triangleright^* \mathcal{N}\{\!|\Gamma|\!\}$
by (*rule parallel-HSem-ind-different-ESem*[*OF pointwise-adm*[*OF similar-admI*]])
(*auto simp add*: *evalHeap-similar*)

end

38 Adequacy.tex

theory *Adequacy*
imports *ResourcedAdequacy Denotational−Related*
begin

theorem *adequacy*:
assumes $[\![e]\!]_{\{\!|\Gamma|\!\}} \neq \bot$
shows $\exists\ \Delta\ v.\ \Gamma : e \Downarrow_S \Delta : v$
proof−
have $\{\!|\Gamma|\!\} \triangleleft\triangleright^* \mathcal{N}\{\!|\Gamma|\!\}$ **by** (*rule heaps-similar*)
hence $[\![e]\!]_{\{\!|\Gamma|\!\}} \triangleleft\triangleright (\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot C^{\infty}$ **by** (*rule denotational-semantics-similar*)
from *bot-or-not-bot*[*OF this*] *assms*
have $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot C^{\infty} \neq \bot$ **by** *metis*
thus *?thesis* **by** (*rule resourced-adequacy*)
qed

end

39 BalancedTraces.tex

theory *BalancedTraces*
imports *Main*
begin

locale *traces* =
fixes *step* :: $'c$ => $'c$ => *bool* (**infix** $\Rightarrow$ *50*)
begin

abbreviation *steps* (**infix** $\Rightarrow^*$ *50*) **where** *steps* $\equiv$ *step***

inductive *trace* :: $'c \Rightarrow 'c$ *list* $\Rightarrow 'c \Rightarrow$ *bool* **where**
trace-nil[*iff*]: *trace final* [] *final*
| *trace-cons*[*intro*]: *trace conf′ T final* $\Longrightarrow$ *conf* $\Rightarrow$ *conf′* $\Longrightarrow$ *trace conf* (*conf′#T*) *final*

inductive-cases *trace-consE*: *trace conf* (*conf′#T*) *final*

lemma *trace-induct-final*[*consumes 1*, *case-names trace-nil trace-cons*]:

trace x1 x2 final $\Longrightarrow$ *P final* [] *final* $\Longrightarrow$ ($\bigwedge$*conf′ T conf*. *trace conf′ T final* $\Longrightarrow$ *P conf′ T final* $\Longrightarrow$ *conf* $\Rightarrow$ *conf′* $\Longrightarrow$ *P conf* (*conf′* # *T*) *final*) $\Longrightarrow$ *P x1 x2 final*
by (*induction rule*:*trace.induct*) *auto*

lemma *build-trace*:
c $\Rightarrow^*$ *c′* $\Longrightarrow$ $\exists$ *T*. *trace c T c′*
proof(*induction rule*: *converse-rtranclp-induct*)
have *trace c′* [] *c′*..
thus $\exists$ *T*. *trace c′ T c′*..
next
fix *y z*
assume *y* $\Rightarrow$ *z*
assume $\exists$ *T*. *trace z T c′*
then obtain *T* **where** *trace z T c′*..
with ‹*y* $\Rightarrow$ *z*›
have *trace y* (*z*#*T*) *c′* **by** *auto*
thus $\exists$ *T*. *trace y T c′* **by** *blast*
qed

lemma *destruct-trace*: *trace c T c′* $\Longrightarrow$ *c* $\Rightarrow^*$ *c′*
by (*induction rule*:*trace.induct*) *auto*

lemma *traceWhile*:
assumes *trace* c_1 *T* c_4
assumes *P* c_1
assumes $\neg$ *P* c_4
obtains T_1 c_2 c_3 T_2
where $T = T_1$ @ c_3 # T_2 **and** *trace* c_1 T_1 c_2 **and** $\forall x \in$ *set* T_1. *P x* **and** *P* c_2 **and** $c_2 \Rightarrow c_3$ **and** $\neg$ *P* c_3 **and** *trace* c_3 T_2 c_4
proof−
from *assms*
have $\exists$ T_1 c_2 c_3 T_2 . ($T = T_1$ @ c_3 # T_2 $\wedge$ *trace* c_1 T_1 c_2 $\wedge$ ($\forall x \in$ *set* T_1. *P x*) $\wedge$ *P* c_2 $\wedge$ $c_2 \Rightarrow c_3$ $\wedge$ $\neg$ *P* c_3 $\wedge$ *trace* c_3 T_2 c_4)
proof(*induction*)
case *trace-nil* **thus** *?case* **by** *auto*
next
case (*trace-cons conf′ T end conf*)
thus *?case*
proof (*cases P conf′*)
case *True*
from *trace-cons.IH*[*OF True* ‹$\neg$ *P end*›]
obtain T_1 c_2 c_3 T_2 **where** $T = T_1$ @ c_3 # T_2 $\wedge$ *trace conf′* T_1 c_2 $\wedge$ ($\forall x \in$ *set* T_1. *P x*) $\wedge$ *P* c_2 $\wedge$ $c_2 \Rightarrow c_3$ $\wedge$ $\neg$ *P* c_3 $\wedge$ *trace* c_3 T_2 *end* **by** *auto*
with *True*
have *conf′* # *T* = (*conf′* # T_1) @ c_3 # T_2 $\wedge$ *trace conf* (*conf′* # T_1) c_2 $\wedge$ ($\forall x \in$ *set* (*conf′* # T_1). *P x*) $\wedge$ *P* c_2 $\wedge$ $c_2 \Rightarrow c_3$ $\wedge$ $\neg$ *P* c_3 $\wedge$ *trace* c_3 T_2 *end* **by** (*auto intro*: *trace-cons*)
thus *?thesis* **by** *blast*
next
case *False* **with** *trace-cons*

have $conf' \# T = [] @ conf' \# T \wedge (\forall x\in set []. P x) \wedge P conf \wedge conf \Rightarrow conf' \wedge \neg P conf' \wedge trace conf' T end$ **by** *auto*
thus *?thesis* **by** *blast*
qed
qed
thus *?thesis* **by** (*auto intro*: *that*)
qed

lemma *traces-list-all*:
$trace\ c\ T\ c' \Longrightarrow P\ c' \Longrightarrow (\bigwedge c\ c'.\ c \Rightarrow c' \Longrightarrow P\ c' \Longrightarrow P\ c) \Longrightarrow (\forall\ x\in set\ T.\ P\ x) \wedge P\ c$
by (*induction rule*:*trace.induct*) *auto*

lemma *trace-nil*[*simp*]: $trace\ c\ []\ c' \longleftrightarrow c = c'$
by (*metis list.distinct*(*1*) *trace.cases traces.trace-nil*)

end

definition *extends* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$ (**infix** $\lesssim$ *50*) **where**
$S \lesssim S' = (\exists\ S''.\ S' = S'' @ S)$

lemma *extends-refl*[*simp*]: $S \lesssim S$ **unfolding** *extends-def* **by** *auto*
lemma *extends-cons*[*simp*]: $S \lesssim x \# S$ **unfolding** *extends-def* **by** *auto*
lemma *extends-append*[*simp*]: $S \lesssim L @ S$ **unfolding** *extends-def* **by** *auto*
lemma *extends-not-cons*[*simp*]: $\neg (x \# S) \lesssim S$ **unfolding** *extends-def* **by** *auto*
lemma *extends-trans*[*trans*]: $S \lesssim S' \Longrightarrow S' \lesssim S'' \Longrightarrow S \lesssim S''$ **unfolding** *extends-def* **by** *auto*

locale *balance-trace* = *traces* +
fixes *stack* :: $'a \Rightarrow 's\ list$
assumes *one-step-only*: $c \Rightarrow c' \Longrightarrow (stack\ c) = (stack\ c') \vee (\exists\ x.\ stack\ c' = x \# stack\ c) \vee (\exists\ x.\ stack\ c = x \# stack\ c')$
begin

inductive *bal* :: $'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$ **where**
balI[*intro*]: $trace\ c\ T\ c' \Longrightarrow \forall c'\in set\ T.\ stack\ c \lesssim stack\ c' \Longrightarrow stack\ c' = stack\ c \Longrightarrow bal\ c\ T\ c'$

inductive-cases *balE*: $bal\ c\ T\ c'$

lemma *bal-nil*[*simp*]: $bal\ c\ []\ c' \longleftrightarrow c = c'$
by (*auto elim*: *balE trace.cases*)

lemma *bal-stackD*: $bal\ c\ T\ c' \Longrightarrow stack\ c' = stack\ c$ **by** (*auto dest*: *balE*)

lemma *stack-passes-lower-bound*:
assumes $c_3 \Rightarrow c_4$
assumes $stack\ c_2 \lesssim stack\ c_3$
assumes $\neg\ stack\ c_2 \lesssim stack\ c_4$

shows *stack* c_3 = *stack* c_2 **and** *stack* c_4 = *tl* (*stack* c_2)
proof−
  from *one-step-only*[*OF assms*(*1*)]
  have *stack* c_3 = *stack* c_2 ∧ *stack* c_4 = *tl* (*stack* c_2)
  proof(*elim disjE exE*)
    assume *stack* c_3 = *stack* c_4 **with** *assms*(*2*,*3*)
    have *False* **by** *auto*
    thus *?thesis*..
  next
    fix x
    note ⟨*stack* c_2 $\lesssim$ *stack* c_3⟩
    also
    assume *stack* c_4 = x # *stack* c_3
    hence *stack* c_3 $\lesssim$ *stack* c_4 **by** *simp*
    finally
    have *stack* c_2 $\lesssim$ *stack* c_4.
    with *assms*(*3*) **show** *?thesis*..
  next
    fix x
    assume c_3: *stack* c_3 = x # *stack* c_4
    with *assms*(*2*)
    obtain L **where** L: x # *stack* c_4 = L @ *stack* c_2 **unfolding** *extends-def* **by** *auto*
    show *?thesis*
    proof(*cases L*)
      case *Nil* **with** c_3 L **have** *stack* c_3 = *stack* c_2 **by** *simp*
      moreover
      from *Nil* c_3 L **have** *stack* c_4 = *tl* (*stack* c_2) **by** (*cases stack* c_2) *auto*
      ultimately
      show *?thesis*..
    next
      case (*Cons y* L')
      with L **have** *stack* c_4 = L' @ *stack* c_2 **by** *simp*
      hence *stack* c_2 $\lesssim$ *stack* c_4 **by** *simp*
      with *assms*(*3*) **show** *?thesis*..
    qed
  qed
  thus *stack* c_3 = *stack* c_2 **and** *stack* c_4 = *tl* (*stack* c_2) **by** *auto*
qed

lemma *bal-consE*:
  assumes *bal* c_1 (c_2 # T) c_5
  and c_2: *stack* c_2 = s # *stack* c_1
  obtains T_1 c_3 c_4 T_2
  where $T = T_1$ @ c_4 # T_2 **and** *bal* c_2 T_1 c_3 **and** $c_3 \Rightarrow c_4$ *bal* c_4 T_2 c_5
using *assms*(*1*)
proof(*rule balE*)

  assume c_5: *stack* c_5 = *stack* c_1

assume T: ∀ $c' \in$ *set* $(c_2 \# T)$. *stack* $c_1 \lesssim$ *stack* c'

assume *trace* c_1 $(c_2 \# T)$ c_5
hence $c_1 \Rightarrow c_2$ **and** *trace* c_2 T c_5 **by** (*auto elim*: *trace-consE*)

note ‹*trace* c_2 T c_5›
moreover
have *stack* $c_2 \lesssim$ *stack* c_2 **by** *simp*
moreover
have ¬ (*stack* $c_2 \lesssim$ *stack* c_5) **unfolding** c_5 c_2 **by** *simp*
ultimately
obtain T_1 c_3 c_4 T_2
  where $T = T_1$ @ $c_4 \# T_2$ **and** *trace* c_2 T_1 c_3 **and** ∀ $c' \in$ *set* T_1. *stack* $c_2 \lesssim$ *stack* c'
  and *stack* $c_2 \lesssim$ *stack* c_3 **and** $c_3 \Rightarrow c_4$ **and** ¬ *stack* $c_2 \lesssim$ *stack* c_4 **and** *trace* c_4 T_2 c_5
  by (*rule traceWhile*)

show *?thesis*
proof (*rule that*)
  show $T = T_1$ @ $c_4 \# T_2$ **by** *fact*

  from ‹$c_3 \Rightarrow c_4$› ‹*stack* $c_2 \lesssim$ *stack* c_3› ‹¬ *stack* $c_2 \lesssim$ *stack* c_4›
  have *stack* c_3 = *stack* c_2 **and** $c_2{}'$: *stack* c_4 = *tl* (*stack* c_2) **by** (*rule stack-passes-lower-bound*)+

  from ‹*trace* c_2 T_1 c_3› ‹∀ $a \in$ *set* T_1. *stack* $c_2 \lesssim$ *stack* a› *this*(*1*)
  show *bal* c_2 T_1 c_3..

  show $c_3 \Rightarrow c_4$ **by** *fact*

  have c_4: *stack* c_4 = *stack* c_1 **using** c_2 $c_2{}'$ **by** *simp*

  note ‹*trace* c_4 T_2 c_5›
  moreover
  have ∀ $a \in$*set* T_2. *stack* $c_4 \lesssim$ *stack* a **using** c_4 T ‹T = -› **by** *auto*
  moreover
  have *stack* c_5 = *stack* c_4 **unfolding** c_4 c_5..
  ultimately
  show *bal* c_4 T_2 c_5..
qed
qed

end

end

40 SestoftConf.tex

theory *SestoftConf*

imports *Terms Substitution*
begin

datatype *stack-elem* = *Alts exp exp* | *Arg var* | *Upd var* | *Dummy var*

instantiation *stack-elem* :: *pt*
begin
definition $\pi \cdot x$ = (*case x of* (*Alts e1 e2*) ⇒ *Alts* ($\pi \cdot$ *e1*) ($\pi \cdot$ *e2*) | (*Arg v*) ⇒ *Arg* ($\pi \cdot v$) | (*Upd v*) ⇒ *Upd* ($\pi \cdot v$) | (*Dummy v*) ⇒ *Dummy* ($\pi \cdot v$))
instance
  by *standard* (*auto simp add*: *permute-stack-elem-def split*:*stack-elem.split*)
end

lemma *Alts-eqvt*[*eqvt*]: $\pi \cdot$ (*Alts e1 e2*) = *Alts* ($\pi \cdot$ *e1*) ($\pi \cdot$ *e2*)
  and *Arg-eqvt*[*eqvt*]: $\pi \cdot$ (*Arg v*) = *Arg* ($\pi \cdot v$)
  and *Upd-eqvt*[*eqvt*]: $\pi \cdot$ (*Upd v*) = *Upd* ($\pi \cdot v$)
  and *Dummy-eqvt*[*eqvt*]: $\pi \cdot$ (*Dummy v*) = *Dummy* ($\pi \cdot v$)
  by (*auto simp add*: *permute-stack-elem-def split*:*stack-elem.split*)

lemma *supp-Alts*[*simp*]: *supp* (*Alts e1 e2*) = *supp e1* ∪ *supp e2* **unfolding** *supp-def* **by** (*auto simp add*: *Collect-imp-eq Collect-neg-eq*)
lemma *supp-Arg*[*simp*]: *supp* (*Arg v*) = *supp v* **unfolding** *supp-def* **by** *auto*
lemma *supp-Upd*[*simp*]: *supp* (*Upd v*) = *supp v* **unfolding** *supp-def* **by** *auto*
lemma *supp-Dummy*[*simp*]: *supp* (*Dummy v*) = *supp v* **unfolding** *supp-def* **by** *auto*
lemma *fresh-Alts*[*simp*]: *a* ♯ *Alts e1 e2* = (*a* ♯ *e1* ∧ *a* ♯ *e2*) **unfolding** *fresh-def* **by** *auto*
lemma *fresh-star-Alts*[*simp*]: *a* ♯* *Alts e1 e2* = (*a* ♯* *e1* ∧ *a* ♯* *e2*) **unfolding** *fresh-star-def* **by** *auto*
lemma *fresh-Arg*[*simp*]: *a* ♯ *Arg v* = *a* ♯ *v* **unfolding** *fresh-def* **by** *auto*
lemma *fresh-Upd*[*simp*]: *a* ♯ *Upd v* = *a* ♯ *v* **unfolding** *fresh-def* **by** *auto*
lemma *fresh-Dummy*[*simp*]: *a* ♯ *Dummy v* = *a* ♯ *v* **unfolding** *fresh-def* **by** *auto*
lemma *fv-Alts*[*simp*]: *fv* (*Alts e1 e2*) = *fv e1* ∪ *fv e2* **unfolding** *fv-def* **by** *auto*
lemma *fv-Arg*[*simp*]: *fv* (*Arg v*) = *fv v* **unfolding** *fv-def* **by** *auto*
lemma *fv-Upd*[*simp*]: *fv* (*Upd v*) = *fv v* **unfolding** *fv-def* **by** *auto*
lemma *fv-Dummy*[*simp*]: *fv* (*Dummy v*) = *fv v* **unfolding** *fv-def* **by** *auto*

instance *stack-elem* :: *fs*
  by *standard* (*case-tac x*, *auto simp add*: *finite-supp*)

type-synonym *stack* = *stack-elem list*

fun *ap* :: *stack* ⇒ *var set* **where**
  ap [] = {}
| *ap* (*Alts e1 e2* # *S*) = *ap S*
| *ap* (*Arg x* # *S*) = *insert x* (*ap S*)
| *ap* (*Upd x* # *S*) = *ap S*
| *ap* (*Dummy x* # *S*) = *ap S*
fun *upds* :: *stack* ⇒ *var set* **where**
  upds [] = {}
| *upds* (*Alts e1 e2* # *S*) = *upds S*

| *upds* (*Upd x* # *S*) = *insert x* (*upds S*)
| *upds* (*Arg x* # *S*) = *upds S*
| *upds* (*Dummy x* # *S*) = *upds S*
fun *dummies* :: *stack* ⇒ *var set* **where**
dummies [] = {}
| *dummies* (*Alts e1 e2* # *S*) = *dummies S*
| *dummies* (*Upd x* # *S*) = *dummies S*
| *dummies* (*Arg x* # *S*) = *dummies S*
| *dummies* (*Dummy x* # *S*) = *insert x* (*dummies S*)
fun *flattn* :: *stack* ⇒ *var list* **where**
flattn [] = []
| *flattn* (*Alts e1 e2* # *S*) = *fv-list e1* @ *fv-list e2* @ *flattn S*
| *flattn* (*Upd x* # *S*) = *x* # *flattn S*
| *flattn* (*Arg x* # *S*) = *x* # *flattn S*
| *flattn* (*Dummy x* # *S*) = *x* # *flattn S*
fun *upds-list* :: *stack* ⇒ *var list* **where**
upds-list [] = []
| *upds-list* (*Alts e1 e2* # *S*) = *upds-list S*
| *upds-list* (*Upd x* # *S*) = *x* # *upds-list S*
| *upds-list* (*Arg x* # *S*) = *upds-list S*
| *upds-list* (*Dummy x* # *S*) = *upds-list S*

lemma *set-upds-list*[*simp*]:
set (*upds-list S*) = *upds S*
by (*induction S rule*: *upds-list.induct*) *auto*

lemma *ups-fv-subset*: *upds S* ⊆ *fv S*
by (*induction S rule*: *upds.induct*) *auto*
lemma *fresh-distinct-ups*: *atom* ‘ *V* ♯∗ *S* ⟹ *V* ∩ *upds S* = {}
by (*auto dest*!: *fresh-distinct-fv set-mp*[*OF ups-fv-subset*])
lemma *ap-fv-subset*: *ap S* ⊆ *fv S*
by (*induction S rule*: *upds.induct*) *auto*
lemma *dummies-fv-subset*: *dummies S* ⊆ *fv S*
by (*induction S rule*: *dummies.induct*) *auto*

lemma *fresh-flattn*[*simp*]: *atom* (*a*::*var*) ♯ *flattn S* ⟷ *atom a* ♯ *S*
by (*induction S rule*:*flattn.induct*) (*auto simp add*: *fresh-Nil fresh-Cons fresh-append fresh-fv*[*OF finite-fv*])
lemma *fresh-star-flattn*[*simp*]: *atom* ‘ (*as*:: *var set*) ♯∗ *flattn S* ⟷ *atom* ‘ *as* ♯∗ *S*
by (*auto simp add*: *fresh-star-def*)
lemma *fresh-upds-list*[*simp*]: *atom a* ♯ *S* ⟹ *atom* (*a*::*var*) ♯ *upds-list S*
by (*induction S rule*:*upds-list.induct*) (*auto simp add*: *fresh-Nil fresh-Cons fresh-append fresh-fv*[*OF finite-fv*])
lemma *fresh-star-upds-list*[*simp*]: *atom* ‘ (*as*:: *var set*) ♯∗ *S* ⟹ *atom* ‘ (*as*:: *var set*) ♯∗ *upds-list S*
by (*auto simp add*: *fresh-star-def*)

lemma *upds-append*[*simp*]: *upds* (*S*@*S*′) = *upds S* ∪ *upds S*′
by (*induction S rule*: *upds.induct*) *auto*

lemma *upds-map-Dummy*[*simp*]: *upds* (*map Dummy l*) = {}
  by (*induction l*) *auto*

lemma *upds-list-append*[*simp*]: *upds-list* (S@S') = *upds-list S* @ *upds-list* S'
  by (*induction S rule*: *upds.induct*) *auto*
lemma *upds-list-map-Dummy*[*simp*]: *upds-list* (*map Dummy l*) = []
  by (*induction l*) *auto*

lemma *dummies-append*[*simp*]: *dummies* (S@S') = *dummies S* ∪ *dummies* S'
  by (*induction S rule*: *dummies.induct*) *auto*
lemma *dummies-map-Dummy*[*simp*]: *dummies* (*map Dummy l*) = *set l*
  by (*induction l*) *auto*

lemma *map-Dummy-inj*[*simp*]: *map Dummy l* = *map Dummy* l' ⟷ $l = l'$
  apply (*induction l arbitrary*: l')
  apply (*case-tac* [!] l')
  apply *auto*
  done

type-synonym *conf* = (*heap* × *exp* × *stack*)

inductive *boring-step* **where**
  isVal e ⟹ *boring-step* (Γ, *e*, *Upd x* # *S*)

fun *restr-stack* :: *var set* ⇒ *stack* ⇒ *stack*
  where *restr-stack V* [] = []
    | *restr-stack V* (*Alts e1 e2* # *S*) = *Alts e1 e2* # *restr-stack V S*
    | *restr-stack V* (*Arg x* # *S*) = *Arg x* # *restr-stack V S*
    | *restr-stack V* (*Upd x* # *S*) = (*if x* ∈ *V then Upd x* # *restr-stack V S else restr-stack V S*)
    | *restr-stack V* (*Dummy x* # *S*) = *Dummy x* # *restr-stack V S*

lemma *restr-stack-cong*:
  (⋀ *x*. *x* ∈ *upds S* ⟹ *x* ∈ *V* ⟷ *x* ∈ V') ⟹ *restr-stack V S* = *restr-stack* V' *S*
  by (*induction V S rule*: *restr-stack.induct*) *auto*

lemma *upds-restr-stack*[*simp*]: *upds* (*restr-stack V S*) = *upds S* ∩ *V*
  by (*induction V S rule*: *restr-stack.induct*) *auto*

lemma *fresh-star-restict-stack*[*intro*]:
  a ♯* *S* ⟹ *a* ♯* *restr-stack V S*
  by (*induction V S rule*: *restr-stack.induct*) (*auto simp add*: *fresh-star-Cons*)

lemma *restr-stack-restr-stack*[*simp*]:
  restr-stack V (*restr-stack* V' *S*) = *restr-stack* (*V* ∩ V') *S*
  by (*induction V S rule*: *restr-stack.induct*) *auto*

lemma *Upd-eq-restr-stackD*:

assumes *Upd x # S = restr-stack V S'*
shows $x \in V$
using *arg-cong*[**where** f = *upds*, *OF assms*]
by *auto*
lemma *Upd-eq-restr-stackD2*:
assumes *restr-stack V S' = Upd x # S*
shows $x \in V$
using *arg-cong*[**where** f = *upds*, *OF assms*]
by *auto*

lemma *restr-stack-noop*[*simp*]:
restr-stack V S = S $\longleftrightarrow$ *upds S* $\subseteq$ *V*
by (*induction V S rule*: *restr-stack.induct*)
(*auto dest*: *Upd-eq-restr-stackD2*)

40.1 Invariants of the semantics

inductive *invariant* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where $(\bigwedge x\ y.\ rel\ x\ y \Longrightarrow I\ x \Longrightarrow I\ y) \Longrightarrow invariant\ rel\ I$

lemmas *invariant.intros*[*case-names step*]

lemma *invariantE*:
$invariant\ rel\ I \Longrightarrow rel\ x\ y \Longrightarrow I\ x \Longrightarrow I\ y$ **by** (*auto elim*: *invariant.cases*)

lemma *invariant-starE*:
$rtranclp\ rel\ x\ y \Longrightarrow invariant\ rel\ I \Longrightarrow\ I\ x \Longrightarrow I\ y$
by (*induction rule*: *rtranclp.induct*) (*auto elim*: *invariantE*)

lemma *invariant-True*:
invariant rel (λ -. *True*)
by (*auto intro*: *invariant.intros*)

lemma *invariant-conj*:
$invariant\ rel\ I1 \Longrightarrow invariant\ rel\ I2 \Longrightarrow invariant\ rel\ (\lambda\ x.\ I1\ x \wedge I2\ x)$
by (*auto simp add*: *invariant.simps*)

lemma *rtranclp-invariant-induct*[*consumes 3*, *case-names base step*]:
assumes $r^{**}\ a\ b$
assumes *invariant r I*
assumes *I a*
assumes *P a*
assumes $(\bigwedge y\ z.\ r^{**}\ a\ y \Longrightarrow r\ y\ z \Longrightarrow I\ y \Longrightarrow I\ z \Longrightarrow P\ y \Longrightarrow P\ z)$
shows *P b*
proof−
from *assms*(*1*,*3*)
have *P b* **and** *I b*

proof(*induction*)
case *base*
from ‹$P\ a$› **show** $P\ a$.
from ‹$I\ a$› **show** $I\ a$.
next
case (*step y z*)
with ‹$I\ a$› **have** $P\ y$ **and** $I\ y$ **by** *auto*

from *assms*(*2*) ‹$r\ y\ z$› ‹$I\ y$›
show $I\ z$ **by** (*rule invariantE*)

from ‹$r^{**}\ a\ y$› ‹$r\ y\ z$› ‹$I\ y$› ‹$I\ z$› ‹$P\ y$›
show $P\ z$ **by** (*rule assms*(*5*))
qed
thus $P\ b$ **by**−
qed

fun *closed* :: *conf* ⇒ *bool*
where *closed* (Γ, *e*, *S*) ⟷ *fv* (Γ, *e*, *S*) ⊆ *domA* Γ ∪ *upds* *S*

fun *heap-upds-ok* **where** *heap-upds-ok* (Γ,*S*) ⟷ *domA* Γ ∩ *upds* *S* = {} ∧ *distinct* (*upds-list* *S*)

abbreviation *heap-upds-ok-conf* :: *conf* ⇒ *bool*
where *heap-upds-ok-conf* *c* ≡ *heap-upds-ok* (*fst* *c*, *snd* (*snd* *c*))

lemma *heap-upds-okE*: *heap-upds-ok* (Γ, *S*) ⟹ *x* ∈ *domA* Γ ⟹ *x* ∉ *upds* *S*
by *auto*

lemma *heap-upds-ok-Nil*[*simp*]: *heap-upds-ok* (Γ, []) **by** *auto*
lemma *heap-upds-ok-app1*: *heap-upds-ok* (Γ, *S*) ⟹ *heap-upds-ok* (Γ,*Arg* *x* # *S*) **by** *auto*
lemma *heap-upds-ok-app2*: *heap-upds-ok* (Γ, *Arg* *x* # *S*) ⟹ *heap-upds-ok* (Γ, *S*) **by** *auto*
lemma *heap-upds-ok-alts1*: *heap-upds-ok* (Γ, *S*) ⟹ *heap-upds-ok* (Γ,*Alts* *e1* *e2* # *S*) **by** *auto*
lemma *heap-upds-ok-alts2*: *heap-upds-ok* (Γ, *Alts* *e1* *e2* # *S*) ⟹ *heap-upds-ok* (Γ, *S*) **by** *auto*

lemma *heap-upds-ok-append*:
assumes *domA* Δ ∩ *upds* *S* = {}
assumes *heap-upds-ok* (Γ,*S*)
shows *heap-upds-ok* (Δ@Γ, *S*)
using *assms*
unfolding *heap-upds-ok.simps*
by *auto*

lemma *heap-upds-ok-let*:
assumes *atom* ‘ *domA* Δ ♯∗ *S*
assumes *heap-upds-ok* (Γ, *S*)
shows *heap-upds-ok* (Δ @ Γ, *S*)
using *assms*(*2*) *fresh-distinct-fv*[*OF* *assms*(*1*)]

by (*auto intro*: *heap-upds-ok-append dest*: *set-mp*[*OF ups-fv-subset*])

lemma *heap-upds-ok-to-stack*:
  $x \in domA\ \Gamma \Longrightarrow heap\text{-}upds\text{-}ok\ (\Gamma,\ S) \Longrightarrow heap\text{-}upds\text{-}ok\ (delete\ x\ \Gamma,\ Upd\ x\ \#S)$
  by (*auto*)

lemma *heap-upds-ok-to-stack'*:
  $map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow heap\text{-}upds\text{-}ok\ (\Gamma,\ S) \Longrightarrow heap\text{-}upds\text{-}ok\ (delete\ x\ \Gamma,\ Upd\ x\ \#S)$
  by (*metis Domain.DomainI domA-def fst-eq-Domain heap-upds-ok-to-stack map-of-SomeD*)

lemma *heap-upds-ok-delete*:
  $heap\text{-}upds\text{-}ok\ (\Gamma,\ S) \Longrightarrow heap\text{-}upds\text{-}ok\ (delete\ x\ \Gamma,\ S)$
  by *auto*

lemma *heap-upds-ok-restrictA*:
  $heap\text{-}upds\text{-}ok\ (\Gamma,\ S) \Longrightarrow heap\text{-}upds\text{-}ok\ (restrictA\ \ V\ \Gamma,\ S)$
  by *auto*

lemma *heap-upds-ok-restr-stack*:
  $heap\text{-}upds\text{-}ok\ (\Gamma,\ S) \Longrightarrow heap\text{-}upds\text{-}ok\ (\Gamma,\ restr\text{-}stack\ V\ S)$
  apply *auto*
  by (*induction V S rule*: *restr-stack.induct*) *auto*

lemma *heap-upds-ok-to-heap*:
  $heap\text{-}upds\text{-}ok\ (\Gamma,\ Upd\ x\ \#\ S) \Longrightarrow heap\text{-}upds\text{-}ok\ ((x,e)\ \#\ \Gamma,\ S)$
  by *auto*

lemma *heap-upds-ok-reorder*:
  $x \in domA\ \Gamma \Longrightarrow heap\text{-}upds\text{-}ok\ (\Gamma,\ S) \Longrightarrow heap\text{-}upds\text{-}ok\ ((x,e)\ \#\ delete\ x\ \Gamma,\ S)$
  by (*intro heap-upds-ok-to-heap heap-upds-ok-to-stack*)

lemma *heap-upds-ok-upd*:
$heap\text{-}upds\text{-}ok\ (\Gamma,\ Upd\ x\ \#\ S) \Longrightarrow x \notin domA\ \Gamma \wedge x \notin upds\ S$
  by *auto*

lemmas *heap-upds-ok-intros*[*intro*] =
  heap-upds-ok-to-heap heap-upds-ok-to-stack heap-upds-ok-to-stack' heap-upds-ok-reorder
  heap-upds-ok-app1 heap-upds-ok-app2 heap-upds-ok-alts1 heap-upds-ok-alts2 heap-upds-ok-delete
  heap-upds-ok-restrictA heap-upds-ok-restr-stack
  heap-upds-ok-let
lemmas *heap-upds-ok.simps*[*simp del*]

end

41 Sestoft.tex

theory *Sestoft*
imports *SestoftConf*
begin

inductive *step* :: *conf* $\Rightarrow$ *conf* $\Rightarrow$ *bool* (**infix** $\Rightarrow$ *50*) **where**
  app_1: $(\Gamma, App\ e\ x, S) \Rightarrow (\Gamma, e\ , Arg\ x\ \#\ S)$
| app_2: $(\Gamma, Lam\ [y].\ e, Arg\ x\ \#\ S) \Rightarrow (\Gamma, e[y ::= x]\ , S)$
| var_1: $map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow (\Gamma, Var\ x, S) \Rightarrow (delete\ x\ \Gamma, e\ , Upd\ x\ \#\ S)$
| var_2: $x \notin domA\ \Gamma \Longrightarrow isVal\ e \Longrightarrow (\Gamma, e, Upd\ x\ \#\ S) \Rightarrow ((x,e)\#\ \Gamma, e\ , S)$
| let_1: $atom\ `\ domA\ \Delta\ \sharp*\ \Gamma \Longrightarrow atom\ `\ domA\ \Delta\ \sharp*\ S$
$\Longrightarrow (\Gamma, Let\ \Delta\ e, S) \Rightarrow (\Delta @\Gamma, e\ , S)$
| if_1: $(\Gamma, scrut\ ?\ e1 : e2, S) \Rightarrow (\Gamma, scrut, Alts\ e1\ e2\ \#\ S)$
| if_2: $(\Gamma, Bool\ b, Alts\ e1\ e2\ \#\ S) \Rightarrow (\Gamma, if\ b\ then\ e1\ else\ e2, S)$

abbreviation *steps* (**infix** $\Rightarrow^*$ *50*) **where** $steps \equiv step^{**}$

lemma *SmartLet-stepI*:
  $atom\ `\ domA\ \Delta\ \sharp*\ \Gamma \Longrightarrow atom\ `\ domA\ \Delta\ \sharp*\ S \Longrightarrow (\Gamma, SmartLet\ \Delta\ e, S) \Rightarrow^* (\Delta @\Gamma, e\ , S)$
unfolding *SmartLet-def* **by** (*auto intro*: let_1)

lemma *lambda-var*: $map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow isVal\ e \Longrightarrow (\Gamma, Var\ x, S) \Rightarrow^* ((x,e)\ \#\ delete\ x\ \Gamma, e\ , S)$
  by (*rule rtranclp-trans*[*OF r-into-rtranclp r-into-rtranclp*])
    (*auto intro*: var_1 var_2)

lemma let_1*-closed*:
  assumes *closed* $(\Gamma, Let\ \Delta\ e, S)$
  assumes $domA\ \Delta \cap domA\ \Gamma = \{\}$
  assumes $domA\ \Delta \cap upds\ S = \{\}$
  shows $(\Gamma, Let\ \Delta\ e, S) \Rightarrow (\Delta @\Gamma, e\ , S)$
proof
  from $\langle domA\ \Delta \cap domA\ \Gamma = \{\}\rangle$ **and** $\langle domA\ \Delta \cap upds\ S = \{\}\rangle$
  have $domA\ \Delta \cap (domA\ \Gamma \cup upds\ S) = \{\}$ **by** *auto*
  with ‹*closed* -›
  have $domA\ \Delta \cap fv\ (\Gamma, S) = \{\}$ **by** *auto*
  hence $atom\ `\ domA\ \Delta\ \sharp*\ (\Gamma, S)$
    by (*auto simp add*: *fresh-star-def fv-def fresh-def*)
  thus $atom`\ domA\ \Delta\ \sharp*\ \Gamma$ **and** $atom\ `\ domA\ \Delta\ \sharp*\ S$ **by** (*auto simp add*: *fresh-star-Pair*)
qed

An induction rule that skips the annoying case of a lambda taken off the heap

lemma *step-invariant-induction*[*consumes 4*, *case-names* app_1 app_2 *thunk lamvar* var_2 let_1 if_1 if_2 *refl trans*]:
  assumes $c \Rightarrow^* c'$
  assumes $\neg$ *boring-step* c'
  assumes *invariant op* $\Rightarrow$ *I*

assumes $I\ c$
assumes app_1: $\bigwedge\ \Gamma\ e\ x\ S\ .\ I\ (\Gamma,\ App\ e\ x,\ S) \Longrightarrow P\ (\Gamma,\ App\ e\ x,\ S)\ \ (\Gamma,\ e\ ,\ Arg\ x\ \#\ S)$
assumes app_2: $\bigwedge\ \Gamma\ y\ e\ x\ S\ .\ I\ (\Gamma,\ Lam\ [y].\ e,\ Arg\ x\ \#\ S) \Longrightarrow P\ (\Gamma,\ Lam\ [y].\ e,\ Arg\ x\ \#\ S)\ (\Gamma,\ e[y ::= x]\ ,\ S)$
assumes $thunk$: $\bigwedge\ \Gamma\ x\ e\ S\ .\ map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow \neg\ isVal\ e \Longrightarrow I\ (\Gamma,\ Var\ x,\ S) \Longrightarrow P\ (\Gamma,\ Var\ x,\ S)\ (delete\ x\ \Gamma,\ e\ ,\ Upd\ x\ \#\ S)$
assumes $lamvar$: $\bigwedge\ \Gamma\ x\ e\ S\ .\ map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow isVal\ e \Longrightarrow I\ (\Gamma,\ Var\ x,\ S) \Longrightarrow P\ (\Gamma,\ Var\ x,\ S)\ ((x,e)\ \#\ delete\ x\ \Gamma,\ e\ ,\ S)$
assumes var_2: $\bigwedge\ \Gamma\ x\ e\ S\ .\ x \notin domA\ \Gamma \Longrightarrow isVal\ e \Longrightarrow I\ (\Gamma,\ e,\ Upd\ x\ \#\ S) \Longrightarrow P\ (\Gamma,\ e,\ Upd\ x\ \#\ S)\ ((x,e)\#\ \Gamma,\ e\ ,\ S)$
assumes let_1: $\bigwedge\ \Delta\ \Gamma\ e\ S\ .\ atom\ `\ domA\ \Delta\ \sharp{*}\ \Gamma \Longrightarrow atom\ `\ domA\ \Delta\ \sharp{*}\ S \Longrightarrow I\ (\Gamma,\ Let\ \Delta\ e,\ S) \Longrightarrow P\ (\Gamma,\ Let\ \Delta\ e,\ S)\ (\Delta @ \Gamma,\ e,\ S)$
assumes if_1: $\bigwedge \Gamma\ scrut\ e1\ e2\ S.\ I\ \ (\Gamma,\ scrut\ ?\ e1 : e2,\ S) \Longrightarrow P\ (\Gamma,\ scrut\ ?\ e1 : e2,\ S)\ (\Gamma,\ scrut,\ Alts\ e1\ e2\ \#\ S)$
assumes if_2: $\bigwedge \Gamma\ b\ e1\ e2\ S.\ I\ \ (\Gamma,\ Bool\ b,\ Alts\ e1\ e2\ \#\ S) \Longrightarrow P\ (\Gamma,\ Bool\ b,\ Alts\ e1\ e2\ \#\ S)\ (\Gamma,\ if\ b\ then\ e1\ else\ e2,\ S)$
assumes $refl$: $\bigwedge\ c.\ P\ c\ c$
assumes $trans[trans]$: $\bigwedge\ c\ c'\ c''.\ c \Rightarrow^* c' \Longrightarrow c' \Rightarrow^* c'' \Longrightarrow P\ c\ c' \Longrightarrow P\ c'\ c'' \Longrightarrow P\ c\ c''$
shows $P\ c\ c'$
proof−
from *assms(1,3,4)*
have $P\ c\ c' \vee (boring\text{-}step\ c' \wedge (\forall\ \ c''.\ c' \Rightarrow c'' \longrightarrow P\ c\ c''))$
proof(*induction rule*: *rtranclp-invariant-induct*)
case *base*
have $P\ c\ c$ **by** (*rule refl*)
thus *?case*..
next
case (*step y z*)
from *step(5)*
show *?case*
proof
assume $P\ c\ y$

note $t = trans[OF\ \langle c \Rightarrow^* y\rangle\ r\text{-}into\text{-}rtranclp$[**where** $r = step,\ OF\ \langle y \Rightarrow z\rangle]]$

from $\langle y \Rightarrow z\rangle$
show *?thesis*
proof (*cases*)
case app_1 **hence** $P\ y\ z$ **using** *assms(5)* $\langle I\ y\rangle$ **by** *metis*
with $\langle P\ c\ y\rangle$ **show** *?thesis* **by** (*metis t*)
next
case app_2 **hence** $P\ y\ z$ **using** *assms(6)* $\langle I\ y\rangle$ **by** *metis*
with $\langle P\ c\ y\rangle$ **show** *?thesis* **by** (*metis t*)
next
case ($var_1\ \Gamma\ x\ e\ S$)
show *?thesis*
proof (*cases isVal e*)
case *False* **with** var_1 **have** $P\ y\ z$ **using** *assms(7)* $\langle I\ y\rangle$ **by** *metis*
with $\langle P\ c\ y\rangle$ **show** *?thesis* **by** (*metis t*)

next
case *True*
have $*$: $y \Rightarrow^* ((x,e) \# \mathit{delete}\ x\ \Gamma,\ e\ ,\ S)$ **using** var_1 *True lambda-var* **by** *metis*

have *boring-step* $(\mathit{delete}\ x\ \Gamma,\ e,\ \mathit{Upd}\ x \# S)$ **using** *True* **..**
moreover
have $P\ y\ ((x,e) \# \mathit{delete}\ x\ \Gamma,\ e\ ,\ S)$ **using** var_1 *True* $\mathit{assms}(8)$ $\langle I\ y\rangle$ **by** *metis*
with $\langle P\ c\ y\rangle$ **have** $P\ c\ ((x,e) \# \mathit{delete}\ x\ \Gamma,\ e\ ,\ S)$ **by** (*rule trans*[*OF* $\langle c \Rightarrow^* y\rangle$ $*$])
ultimately
show *?thesis* **using** $\mathit{var}_1(2,3)$ *True* **by** (*auto elim*!: *step.cases*)
qed
next
case var_2 **hence** $P\ y\ z$ **using** $\mathit{assms}(9)$ $\langle I\ y\rangle$ **by** *metis*
with $\langle P\ c\ y\rangle$ **show** *?thesis* **by** (*metis t*)
next
case let_1 **hence** $P\ y\ z$ **using** $\mathit{assms}(10)$ $\langle I\ y\rangle$ **by** *metis*
with $\langle P\ c\ y\rangle$ **show** *?thesis* **by** (*metis t*)
next
case if_1 **hence** $P\ y\ z$ **using** $\mathit{assms}(11)$ $\langle I\ y\rangle$ **by** *metis*
with $\langle P\ c\ y\rangle$ **show** *?thesis* **by** (*metis t*)
next
case if_2 **hence** $P\ y\ z$ **using** $\mathit{assms}(12)$ $\langle I\ y\rangle$ **by** *metis*
with $\langle P\ c\ y\rangle$ **show** *?thesis* **by** (*metis t*)
qed
next
assume *boring-step* $y \wedge (\forall c''.\ y \Rightarrow c'' \longrightarrow P\ c\ c'')$
with $\langle y \Rightarrow z\rangle$
have $P\ c\ z$ **by** *blast*
thus *?thesis***..**
qed
qed
with $\mathit{assms}(2)$
show *?thesis* **by** *auto*
qed

lemma *step-induction*[*consumes 2*, *case-names* app_1 app_2 *thunk lamvar* var_2 let_1 if_1 if_2 *refl trans*]:
assumes $c \Rightarrow^* c'$
assumes $\neg$ *boring-step* c'
assumes app_1: $\bigwedge \Gamma\ e\ x\ S\ .\ P\ (\Gamma,\ \mathit{App}\ e\ x,\ S)\ \ (\Gamma,\ e\ ,\ \mathit{Arg}\ x \# S)$
assumes app_2: $\bigwedge \Gamma\ y\ e\ x\ S\ .\ P\ (\Gamma,\ \mathit{Lam}\ [y].\ e,\ \mathit{Arg}\ x \# S)\ (\Gamma,\ e[y ::= x]\ ,\ S)$
assumes *thunk*: $\bigwedge \Gamma\ x\ e\ S\ .\ \mathit{map\text{-}of}\ \Gamma\ x = \mathit{Some}\ e \Longrightarrow \neg\ \mathit{isVal}\ e \Longrightarrow P\ (\Gamma,\ \mathit{Var}\ x,\ S)\ (\mathit{delete}\ x\ \Gamma,\ e\ ,\ \mathit{Upd}\ x \# S)$
assumes *lamvar*: $\bigwedge \Gamma\ x\ e\ S\ .\ \mathit{map\text{-}of}\ \Gamma\ x = \mathit{Some}\ e \Longrightarrow \mathit{isVal}\ e \Longrightarrow P\ (\Gamma,\ \mathit{Var}\ x,\ S)\ ((x,e) \# \mathit{delete}\ x\ \Gamma,\ e\ ,\ S)$
assumes var_2: $\bigwedge \Gamma\ x\ e\ S\ .\ x \notin \mathit{domA}\ \Gamma \Longrightarrow \mathit{isVal}\ e \Longrightarrow P\ (\Gamma,\ e,\ \mathit{Upd}\ x \# S)\ ((x,e) \# \Gamma,\ e\ ,\ S)$
assumes let_1: $\bigwedge \Delta\ \Gamma\ e\ S\ .\ \mathit{atom}\ `\ \mathit{domA}\ \Delta\ \sharp{*}\ \Gamma \Longrightarrow \mathit{atom}\ `\ \mathit{domA}\ \Delta\ \sharp{*}\ S \Longrightarrow P\ (\Gamma,\ \mathit{Let}\ \Delta$

e, S) ($\Delta @ \Gamma$, e, S)
  assumes if_1: $\bigwedge \Gamma$ *scrut e1 e2 S*. P (Γ, *scrut ? e1* : *e2*, S) (Γ, *scrut*, *Alts e1 e2* # S)
  assumes if_2: $\bigwedge \Gamma$ *b e1 e2 S*. P (Γ, *Bool b*, *Alts e1 e2* # S) (Γ, *if b then e1 else e2*, S)
  assumes *refl*: $\bigwedge$ *c*. $P\ c\ c$
  assumes *trans*[*trans*]: $\bigwedge$ $c\ c'\ c''$. $c \Rightarrow^* c' \Longrightarrow c' \Rightarrow^* c'' \Longrightarrow P\ c\ c' \Longrightarrow P\ c'\ c'' \Longrightarrow P\ c\ c''$
  shows $P\ c\ c'$
by (*rule step-invariant-induction*[*OF - - invariant-True*, *simplified*, *OF assms*])

41.1 Equivariance

lemma *step-eqvt*[*eqvt*]: *step x y* $\Longrightarrow$ *step* ($\pi \cdot x$) ($\pi \cdot y$)
  apply (*induction rule*: *step.induct*)
  apply (*perm-simp*, *rule step.intros*)
  apply (*perm-simp*, *rule step.intros*)
  apply (*perm-simp*, *rule step.intros*)
  apply (*rule permute-boolE*[**where** $p = -\pi$], *simp add*: *pemute-minus-self*)
  apply (*perm-simp*, *rule step.intros*)
  apply (*rule permute-boolE*[**where** $p = -\pi$], *simp add*: *pemute-minus-self*)
  apply (*rule permute-boolE*[**where** $p = -\pi$], *simp add*: *pemute-minus-self*)
  apply (*perm-simp*, *rule step.intros*)
  apply (*rule permute-boolE*[**where** $p = -\pi$], *simp add*: *pemute-minus-self*)
  apply (*rule permute-boolE*[**where** $p = -\pi$], *simp add*: *pemute-minus-self*)
  apply (*perm-simp*, *rule step.intros*)
  apply (*perm-simp*, *rule step.intros*)
  done

41.2 Invariants

lemma *closed-invariant*:
  invariant step closed
proof
  fix $c\ c'$
  assume $c \Rightarrow c'$ **and** *closed c*
  thus *closed* c'
  by (*induction rule*: *step.induct*) (*auto simp add*: *fv-subst-eq dest*!: *set-mp*[*OF fv-delete-subset*] *dest*: *set-mp*[*OF map-of-Some-fv-subset*])
qed

lemma *heap-upds-ok-invariant*:
  invariant step heap-upds-ok-conf
proof
  fix $c\ c'$
  assume $c \Rightarrow c'$ **and** *heap-upds-ok-conf c*
  thus *heap-upds-ok-conf* c'
  by (*induction rule*: *step.induct*) *auto*
qed

end

42 SestoftCorrect.tex

theory *SestoftCorrect*
imports *BalancedTraces Launchbury Sestoft*
begin

lemma *lemma-2*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  and $\mathit{fv}\ (\Gamma, e, S) \subseteq \mathit{set}\ L \cup \mathit{domA}\ \Gamma$
  shows $(\Gamma, e, S) \Rightarrow^* (\Delta, z, S)$
using *assms*
proof(*induction arbitrary*: *S rule*:*reds.induct*)
  case (*Lambda* Γ *x e L*)
  show *?case*..
next
  case (*Application y* Γ *e x L* Δ Θ *z e'*)
  from ‹$\mathit{fv}\ (\Gamma, \mathit{App}\ e\ x, S) \subseteq \mathit{set}\ L \cup \mathit{domA}\ \Gamma$›
  have *prem1*: $\mathit{fv}\ (\Gamma, e, \mathit{Arg}\ x \mathbin{\#} S) \subseteq \mathit{set}\ L \cup \mathit{domA}\ \Gamma$ **by** *simp*

  from *prem1 reds-pres-closed*[*OF* ‹$\Gamma : e \Downarrow_L \Delta : \mathit{Lam}\ [y].\ e'$›] *reds-doesnt-forget*[*OF* ‹$\Gamma : e \Downarrow_L \Delta : \mathit{Lam}\ [y].\ e'$›]
  have *prem2*: $\mathit{fv}\ (\Delta, e'[y{::=}x], S) \subseteq \mathit{set}\ L \cup \mathit{domA}\ \Delta$ **by** (*auto simp add*: *fv-subst-eq*)

  have $(\Gamma, \mathit{App}\ e\ x, S) \Rightarrow (\Gamma, e, \mathit{Arg}\ x \mathbin{\#} S)$..
  also have $\ldots \Rightarrow^* (\Delta, \mathit{Lam}\ [y].\ e', \mathit{Arg}\ x\mathbin{\#}\ S)$ **by** (*rule Application.IH*(*1*)[*OF prem1*])
  also have $\ldots \Rightarrow (\Delta, e'[y ::= x], S)$..
  also have $\ldots \Rightarrow^* (\Theta, z, S)$ **by** (*rule Application.IH*(*2*)[*OF prem2*])
  finally show *?case*.
next
case (*Variable* Γ *x e L* Δ *z S*)
  from *Variable*(*2*)
  have *isVal z* **by** (*rule result-evaluated*)

  have $x \notin \mathit{domA}\ \Delta$ **by** (*rule reds-avoids-live*[*OF Variable*(*2*), **where** $x = x$]) *simp-all*

  from ‹$\mathit{fv}\ (\Gamma, \mathit{Var}\ x, S) \subseteq \mathit{set}\ L \cup \mathit{domA}\ \Gamma$›
  have *prem*: $\mathit{fv}\ (\mathit{delete}\ x\ \Gamma, e, \mathit{Upd}\ x \mathbin{\#} S) \subseteq \mathit{set}\ (x\#L) \cup \mathit{domA}\ (\mathit{delete}\ x\ \Gamma)$
   by (*auto dest*: *set-mp*[*OF fv-delete-subset*] *set-mp*[*OF map-of-Some-fv-subset*[*OF* ‹$\mathit{map\text{-}of}\ \Gamma\ x = \mathit{Some}\ e$›]])

  from ‹$\mathit{map\text{-}of}\ \Gamma\ x = \mathit{Some}\ e$›
  have $(\Gamma, \mathit{Var}\ x, S) \Rightarrow (\mathit{delete}\ x\ \Gamma, e, \mathit{Upd}\ x \mathbin{\#} S)$..
  also have $\ldots \Rightarrow^* (\Delta, z, \mathit{Upd}\ x \mathbin{\#} S)$ **by** (*rule Variable.IH*[*OF prem*])
  also have $\ldots \Rightarrow ((x,z)\#\Delta, z, S)$ **using** ‹$x \notin \mathit{domA}\ \Delta$› ‹*isVal z*› **by** (*rule* var_2)
  finally show *?case*.
next
case (*Bool* Γ *b L S*)
  show *?case*..

next
case (*IfThenElse* Γ *scrut* L Δ b e_1 e_2 Θ z S)
  have (Γ, *scrut ?* e_1 : e_2, S) $\Rightarrow$ (Γ, *scrut*, *Alts* e_1 e_2 #S)..
  also
  from *IfThenElse.prems*
  have *prem1*: *fv* (Γ, *scrut*, *Alts* e_1 e_2 #S) $\subseteq$ *set* L $\cup$ *domA* Γ **by** *auto*
  hence (Γ, *scrut*, *Alts* e_1 e_2 #S) $\Rightarrow^*$ (Δ, *Bool* b, *Alts* e_1 e_2 #S)
    by (*rule IfThenElse.IH*)
  also
  have (Δ, *Bool* b, *Alts* e_1 e_2 #S) $\Rightarrow$ (Δ, *if* b *then* e_1 *else* e_2, S)..
  also
  from *prem1 reds-pres-closed*[*OF IfThenElse*(*1*)] *reds-doesnt-forget*[*OF IfThenElse*(*1*)]
  have *prem2*: *fv* (Δ, *if* b *then* e_1 *else* e_2, S) $\subseteq$ *set* L $\cup$ *domA* Δ **by** *auto*
  hence (Δ, *if* b *then* e_1 *else* e_2, S) $\Rightarrow^*$ (Θ, z, S) **by** (*rule IfThenElse.IH*(*2*))
  finally
  show *?case*.
next
case (*Let as* Γ L *body* Δ z S)
  from *Let*(*4*)
  have *prem*: *fv* (*as* @ Γ, *body*, S) $\subseteq$ *set* L $\cup$ *domA* (*as* @ Γ) **by** *auto*

  from *Let*(*1*)
  have *atom* ‘ *domA as* $\sharp*$ Γ **by** (*auto simp add*: *fresh-star-Pair*)
  moreover
  from *Let*(*1*)
  have *domA as* $\cap$ *fv* (Γ, L) = {}
    by (*rule fresh-distinct-fv*)
  hence *domA as* $\cap$ (*set* L $\cup$ *domA* Γ) = {}
    by (*auto dest*: *set-mp*[*OF domA-fv-subset*])
  with *Let*(*4*)
  have *domA as* $\cap$ *fv* S = {}
    by *auto*
  hence *atom* ‘ *domA as* $\sharp*$ S
    by (*auto simp add*: *fresh-star-def fv-def fresh-def*)
  ultimately
  have (Γ, *Terms.Let as body*, S) $\Rightarrow$ (*as*@Γ, *body*, S)..
  also have ... $\Rightarrow^*$ (Δ, z, S)
    by (*rule Let.IH*[*OF prem*])
  finally show *?case*.
qed

type-synonym *trace* = *conf list*

fun *stack* :: *conf* $\Rightarrow$ *stack* **where** *stack* (Γ, e, S) = S

interpretation *traces step*.

abbreviation *trace-syn* (- $\Rightarrow^*_{-}$ - [*50*,*50*,*50*] *50*) **where** *trace-syn* $\equiv$ *trace*

lemma *conf-trace-induct-final*[*consumes 1*, *case-names trace-nil trace-cons*]:
$(\Gamma,\ e,\ S) \Rightarrow^*{}_T$ *final* $\Longrightarrow (\bigwedge \Gamma\ e\ S.$ *final* $= (\Gamma,\ e,\ S) \Longrightarrow P\ \Gamma\ e\ S\ []\ (\Gamma,\ e,\ S)) \Longrightarrow (\bigwedge \Gamma\ e\ S\ T\ \Gamma'\ e'\ S'.\ (\Gamma',\ e',\ S') \Rightarrow^*{}_T$ *final* $\Longrightarrow P\ \Gamma'\ e'\ S'\ T$ *final* $\Longrightarrow (\Gamma,\ e,\ S) \Rightarrow (\Gamma',\ e',\ S') \Longrightarrow P\ \Gamma\ e\ S\ ((\Gamma',\ e',\ S')\ \#\ T)$ *final*$) \Longrightarrow P\ \Gamma\ e\ S\ T$ *final*
by (*induction* $(\Gamma,\ e,\ S)$ *T final arbitrary*: Γ *e S rule*: *trace-induct-final*) *auto*

interpretation *balance-trace step stack*
apply *standard*
apply (*erule step.cases*)
apply *auto*
done

abbreviation *bal-syn* (- $\Rightarrow^{b*}$- - [*50*,*50*,*50*] *50*) **where** *bal-syn* $\equiv$ *bal*

lemma *isVal-stops*:
assumes *isVal e*
assumes $(\Gamma,\ e,\ S) \Rightarrow^{b*}{}_T (\Delta,\ z,\ S)$
shows $T=[]$
using *assms*
apply −
apply (*erule balE*)
apply (*erule trace.cases*)
apply *simp*
apply *auto*
apply (*auto elim*!: *step.cases*)
done

lemma *Ball-subst*[*simp*]:
$(\forall p \in set\ (\Gamma[y::h=x]).\ f\ p) \longleftrightarrow (\forall p \in set\ \Gamma.\ case\ p\ of\ (z,e) \Rightarrow f\ (z,\ e[y::=x]))$
by (*induction* Γ) *auto*

lemma *lemma-3*:
assumes $(\Gamma,\ e,\ S) \Rightarrow^{b*}{}_T (\Delta,\ z,\ S)$
assumes *isVal z*
shows $\Gamma : e \Downarrow_{upds\text{-}list\ S} \Delta : z$
using *assms*
proof(*induction T arbitrary*: Γ *e S* Δ *z rule*: *measure-induct-rule*[**where** *f* = *length*])
case (*less T* Γ *e S* Δ *z*)
from ‹$(\Gamma,\ e,\ S) \Rightarrow^{b*}{}_T (\Delta,\ z,\ S)$›
have $(\Gamma,\ e,\ S) \Rightarrow^*{}_T (\Delta,\ z,\ S)$ **and** $\forall\ c' \in set\ T.\ S \lesssim stack\ c'$ **unfolding** *bal.simps* **by** *auto*

from *this*(*1*)
show *?case*
proof(*cases*)
case *trace-nil*
from ‹*isVal z*› *trace-nil* **show** *?thesis* **by** (*auto intro*: *reds-isValI*)
next
case (*trace-cons conf′ T′*)

from ‹T = $conf'$ # T'› **and** ‹∀ c'∈set T. $S \lesssim stack\ c'$› **have** $S \lesssim stack\ conf'$ **by** *auto*

from ‹(Γ, e, S) ⇒ $conf'$›
show *?thesis*
proof(*cases*)
case (*app*$_1$ *e x*)
obtain T_1 c_3 c_4 T_2
where $T' = T_1$ @ c_4 # T_2 **and** *prem1*: (Γ, e, *Arg x* # S) $\Rightarrow^{b*}{}_{T_1}$ c_3 **and** $c_3 \Rightarrow c_4$ **and** *prem2*: $c_4 \Rightarrow^{b*}{}_{T_2}$ (Δ, z, S)
by (*rule bal-consE*[*OF* ‹*bal* - *T* -›[*unfolded app*$_1$ *trace-cons*]]) (*simp*, *rule*)

from ‹T = -› ‹T' = -› **have** *length* T_1 < *length* T **and** *length* T_2 < *length* T **by** *auto*

from *prem1* **have** *stack* c_3 = *Arg x* # S **by** (*auto dest*: *bal-stackD*)
moreover
from *prem2* **have** *stack* c_4 = S **by** (*auto dest*: *bal-stackD*)
moreover
note ‹$c_3 \Rightarrow c_4$›
ultimately
obtain Δ' y e' **where** c_3 = (Δ', *Lam* [y]. e', *Arg x* # S) **and** c_4 = (Δ', e'[y ::= x], S)
by (*auto elim*!: *step.cases simp del*: *exp-assn.eq-iff*)

from *less*(*1*)[*OF* ‹*length* T_1 < *length* T› *prem1*[*unfolded* ‹c_3 = -› ‹c_4 = -›]]
have Γ : e ⇓$_{upds\text{-}list\ S}$ Δ' : *Lam* [y]. e' **by** *simp*
moreover
from *less*(*1*)[*OF* ‹*length* T_2 < *length* T› *prem2*[*unfolded* ‹c_3 = -› ‹c_4 = -›] ‹*isVal z*›]
have Δ' : e'[y::=x] ⇓$_{upds\text{-}list\ S}$ Δ : z **by** *simp*
ultimately
show *?thesis* **unfolding** *app*$_1$
by (*rule reds-ApplicationI*)
next
case (*app*$_2$ *y e x S'*)
from ‹$conf'$ =-› ‹S = - # S'› ‹$S \lesssim stack\ conf'$›
have *False* **by** (*auto simp add*: *extends-def*)
thus *?thesis*..
next
case (*var*$_1$ *x e*)
obtain T_1 c_3 c_4 T_2
where $T' = T_1$ @ c_4 # T_2 **and** *prem1*: (*delete x* Γ, e, *Upd x* # S) $\Rightarrow^{b*}{}_{T_1}$ c_3 **and** $c_3 \Rightarrow c_4$ **and** *prem2*: $c_4 \Rightarrow^{b*}{}_{T_2}$ (Δ, z, S)
by (*rule bal-consE*[*OF* ‹*bal* - *T* -›[*unfolded var*$_1$ *trace-cons*]]) (*simp*, *rule*)

from ‹T = -› ‹T' = -› **have** *length* T_1 < *length* T **and** *length* T_2 < *length* T **by** *auto*

from *prem1* **have** *stack* c_3 = *Upd x* # S **by** (*auto dest*: *bal-stackD*)
moreover
from *prem2* **have** *stack* c_4 = S **by** (*auto dest*: *bal-stackD*)
moreover

note ‹$c_3 \Rightarrow c_4$›
ultimately
obtain Δ' z' **where** $c_3 = (\Delta', z', Upd\ x\ \#\ S)$ **and** $c_4 = ((x,z')\#\Delta', z', S)$ **and** *isVal* z'
by (*auto elim*!: *step.cases simp del*: *exp-assn.eq-iff*)

from ‹*isVal* z'› **and** *prem2*[*unfolded* ‹c_4 = -›]
have $T_2 = []$ **by** (*rule isVal-stops*)
with *prem2* ‹c_4 = -›
have $z' = z$ **and** $\Delta = (x,z)\#\Delta'$ **by** *auto*

from *less*(*1*)[*OF* ‹*length* T_1 < *length* T› *prem1*[*unfolded* ‹c_3 = -› ‹c_4 = -› ‹z' = -›] ‹*isVal* z›]
have *delete* x $\Gamma : e \Downarrow_{x\ \#\ \mathit{upds\text{-}list}\ S} \Delta' : z$ **by** *simp*
with ‹*map-of* - - = -›
show *?thesis* **unfolding** $var_1(1)$ ‹Δ = -› **by** *rule*
next
case (var_2 x S')
from ‹*conf'* = -› ‹S = - # S'› ‹$S \lesssim$ *stack conf'*›
have *False* **by** (*auto simp add*: *extends-def*)
thus *?thesis*..
next
case (if_1 *scrut* e_1 e_2)
obtain T_1 c_3 c_4 T_2
where $T' = T_1 @ c_4 \# T_2$ **and** *prem1*: $(\Gamma, \mathit{scrut}, \mathit{Alts}\ e_1\ e_2\ \#\ S) \Rightarrow^{b*}{}_{T_1} c_3$ **and** $c_3 \Rightarrow c_4$ **and** *prem2*: $c_4 \Rightarrow^{b*}{}_{T_2} (\Delta, z, S)$
by (*rule bal-consE*[*OF* ‹*bal* - *T* -›[*unfolded* if_1 *trace-cons*]]) (*simp*, *rule*)

from ‹T = -› ‹T' = -› **have** *length* T_1 < *length* T **and** *length* T_2 < *length* T **by** *auto*

from *prem1* **have** *stack* c_3 = *Alts* e_1 e_2 # S **by** (*auto dest*: *bal-stackD*)
moreover
from *prem2* **have** *stack* c_4 = S **by** (*auto dest*: *bal-stackD*)
moreover
note ‹$c_3 \Rightarrow c_4$›
ultimately
obtain Δ' b **where** $c_3 = (\Delta', \mathit{Bool}\ b, \mathit{Alts}\ e_1\ e_2\ \#\ S)$ **and** $c_4 = (\Delta', (\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2), S)$
by (*auto elim*!: *step.cases simp del*: *exp-assn.eq-iff*)

from *less*(*1*)[*OF* ‹*length* T_1 < *length* T› *prem1*[*unfolded* ‹c_3 = -› ‹c_4 = -›] *isVal-Bool*]
have $\Gamma : \mathit{scrut} \Downarrow_{\mathit{upds\text{-}list}\ S} \Delta' : \mathit{Bool}\ b$ **by** *simp*
moreover
from *less*(*1*)[*OF* ‹*length* T_2 < *length* T› *prem2*[*unfolded* ‹c_4 = -›] ‹*isVal* z›]
have $\Delta' : (\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2) \Downarrow_{\mathit{upds\text{-}list}\ S} \Delta : z$.
ultimately
show *?thesis* **unfolding** if_1 **by** (*rule reds.IfThenElse*)
next
case (if_2 *b e1 e2* S')

from ‹$conf' = -$› ‹$S = - \# S'$› ‹$S \lesssim stack\ conf'$›
have *False* **by** (*auto simp add*: *extends-def*)
thus *?thesis*..
next
case (let_1 *as e*)
from ‹$T = conf' \# T'$› **have** *length* $T' <$ *length* T **by** *auto*
moreover
have (as @ Γ, e, S) $\Rightarrow^{b*}{}_{T'}$ (Δ, z, S)
using *trace-cons* ‹$conf' = -$› ‹$\forall\ c' \in set\ T.\ S \lesssim stack\ c'$› **by** *fastforce*
moreover
note ‹*isVal z*›
ultimately
have *as* @ Γ : $e \Downarrow_{upds\text{-}list\ S} \Delta$: z **by** (*rule less*)
moreover
from ‹*atom* ‘ *domA as* $\sharp*$ Γ› ‹*atom* ‘ *domA as* $\sharp*$ S›
have *atom* ‘ *domA as* $\sharp*$ (Γ, *upds-list S*) **by** (*auto simp add*: *fresh-star-Pair*)
ultimately
show *?thesis* **unfolding** let_1 **by** (*rule reds.Let*[*rotated*])
qed
qed
qed

lemma *dummy-stack-extended*:
set S $\subseteq$ *Dummy* ‘ *UNIV* $\Longrightarrow x \notin$ *Dummy* ‘ *UNIV* $\Longrightarrow (S \lesssim x \# S') \longleftrightarrow S \lesssim S'$
apply (*auto simp add*: *extends-def*)
apply (*case-tac* S'')
apply *auto*
done

lemma[*simp*]: *Arg x* $\notin$ *range Dummy* *Upd x* $\notin$ *range Dummy* *Alts* e_1 e_2 $\notin$ *range Dummy* **by** *auto*

lemma *dummy-stack-balanced*:
assumes *set S* $\subseteq$ *Dummy* ‘ *UNIV*
assumes (Γ, e, S) $\Rightarrow^*$ (Δ, z, S)
obtains T **where** (Γ, e, S) $\Rightarrow^{b*}{}_T$ (Δ, z, S)
proof−
from *build-trace*[*OF assms*(*2*)]
obtain T **where** (Γ, e, S) $\Rightarrow^*{}_T$ (Δ, z, S)..
moreover
hence $\forall\ c' \in set\ T.$ *stack* (Γ, e, S) $\lesssim$ *stack* c'
by (*rule conjunct1*[*OF traces-list-all*])
(*auto elim*: *step.cases simp add*: *dummy-stack-extended*[*OF* ‹*set S* $\subseteq$ *Dummy* ‘ *UNIV*›])
ultimately
have (Γ, e, S) $\Rightarrow^{b*}{}_T$ (Δ, z, S)
by (*rule ballI*) *simp*
thus *?thesis* **by** (*rule that*)
qed

end

43 Arity.tex

theory *Arity*
imports *HOLCF−Join−Classes Lifting*
begin

typedef *Arity* = *UNIV* :: *nat set*
  morphisms *Rep-Arity to-Arity* **by** *auto*

setup-lifting *type-definition-Arity*

instantiation *Arity* :: *po*
begin
lift-definition *below-Arity* :: *Arity* ⇒ *Arity* ⇒ *bool* **is** λ x y . $y \leq x$.

instance
apply *standard*
apply ((*transfer*, *auto*)+)
done
end

instance *Arity* :: *chfin*
proof
  fix *S* :: *nat* ⇒ *Arity*
  assume *chain S*
  have *LeastM Rep-Arity* ($\lambda x.$ $x \in$ *range S*) $\in$ *range S*
    by (*rule LeastM-natI*) *auto*
  then obtain *n* **where** *n*: *S n* = *LeastM Rep-Arity* ($\lambda x.$ $x \in$ *range S*) **by** *auto*
  have *max-in-chain n S*
  proof(*rule max-in-chainI*)
    fix *j*
    assume $n \leq j$ **hence** *S n* ⊑ *S j* **using** ‹*chain S*› **by** (*metis chain-mono*)
    also
    have *Rep-Arity* (*S n*) ≤ *Rep-Arity* (*S j*)
      unfolding *n image-def*
      by (*metis* (*lifting*, *full-types*) *LeastM-nat-lemma UNIV-I mem-Collect-eq*)
    hence *S j* ⊑ *S n* **by** *transfer*
    finally
    show *S n* = *S j*.
  qed
  thus ∃ *n*. *max-in-chain n S*..
qed

instance *Arity* :: *cpo* ..

lift-definition *inc-Arity* :: *Arity* ⇒ *Arity* **is** *Suc***.**
lift-definition *pred-Arity* :: *Arity* ⇒ *Arity* **is** $(\lambda\ x\ .\ x - 1)$**.**

lemma *inc-Arity-cont*[*simp*]: *cont inc-Arity*
  apply (*rule chfindom-monofun2cont*)
  apply (*rule monofunI*)
  apply (*transfer*, *simp*)
  done

lemma *pred-Arity-cont*[*simp*]: *cont pred-Arity*
  apply (*rule chfindom-monofun2cont*)
  apply (*rule monofunI*)
  apply (*transfer*, *simp*)
  done

definition *inc* :: *Arity* → *Arity* **where**
  $inc = (\Lambda\ x.\ inc\text{-}Arity\ x)$

definition *pred* :: *Arity* → *Arity* **where**
  $pred = (\Lambda\ x.\ pred\text{-}Arity\ x)$

lemma *inc-inj*[*simp*]: $inc \cdot n = inc \cdot n' \longleftrightarrow n = n'$
  by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *pred-inc*[*simp*]: $pred \cdot (inc \cdot n) = n$
  by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *inc-below-inc*[*simp*]: $inc \cdot a \sqsubseteq inc \cdot b \longleftrightarrow a \sqsubseteq b$
  by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *inc-below-below-pred*[*elim*]:
  $inc \cdot a \sqsubseteq b \Longrightarrow a \sqsubseteq pred \cdot b$
  by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *Rep-Arity-inc*[*simp*]: $Rep\text{-}Arity\ (inc \cdot a') = Suc\ (Rep\text{-}Arity\ a')$
  by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

instantiation *Arity* :: *zero*
begin
lift-definition *zero-Arity* :: *Arity* **is** *0***.**
instance..
end

instantiation *Arity* :: *one*
begin
lift-definition *one-Arity* :: *Arity* **is** *1***.**
instance ..
end

lemma *one-is-inc-zero*: $1 = inc{\cdot}0$
by (*simp add*: *inc-def*, *transfer*, *simp*)

lemma *inc-not-0*[*simp*]: $inc{\cdot}n = 0 \longleftrightarrow False$
by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *pred-0*[*simp*]: $pred{\cdot}0 = 0$
by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *Arity-ind*: $P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (inc{\cdot}n)) \Longrightarrow P\ n$
apply (*simp add*: *inc-def*)
apply *transfer*
by (*rule nat.induct*)

lemma *Arity-total*:
fixes $x\ y$ :: *Arity*
shows $x \sqsubseteq y \vee y \sqsubseteq x$
by *transfer auto*

instance *Arity* :: *Finite-Join-cpo*
proof
fix $x\ y$ :: *Arity*
show *compatible x y* **by** (*metis Arity-total compatibleI*)
qed

lemma *Arity-zero-top*[*simp*]: $(x :: Arity) \sqsubseteq 0$
by *transfer simp*

lemma *Arity-above-top*[*simp*]: $0 \sqsubseteq (a :: Arity) \longleftrightarrow a = 0$
by *transfer simp*

lemma *Arity-zero-join*[*simp*]: $(x :: Arity) \sqcup 0 = 0$
by *transfer simp*
lemma *Arity-zero-join2*[*simp*]: $0 \sqcup (x :: Arity) = 0$
by *transfer simp*

lemma *Arity-up-zero-join*[*simp*]: $(x :: Arity_\bot) \sqcup up{\cdot}0 = up{\cdot}0$
by (*cases x*) *auto*
lemma *Arity-up-zero-join2*[*simp*]: $up{\cdot}0 \sqcup (x :: Arity_\bot) = up{\cdot}0$
by (*cases x*) *auto*
lemma *up-zero-top*[*simp*]: $x \sqsubseteq up{\cdot}(0::Arity)$
by (*cases x*) *auto*
lemma *Arity-above-up-top*[*simp*]: $up{\cdot}0 \sqsubseteq (a :: Arity_\bot) \longleftrightarrow a = up{\cdot}0$
by (*metis Arity-up-zero-join2 join-self-below*(*4*))

lemma *Arity-exhaust*: $(y = 0 \Longrightarrow P) \Longrightarrow (\bigwedge x.\ y = inc \cdot x \Longrightarrow P) \Longrightarrow P$
by (*metis Abs-cfun-inverse2 Arity.inc-def Rep-Arity-inverse inc-Arity.abs-eq inc-Arity-cont*

list-decode.cases zero-Arity-def)

end

44 AEnv.tex

theory *AEnv*
imports *Arity Vars Env*
begin

type-synonym *AEnv* = *var* ⇒ *Arity*$_\bot$

end

45 Arity-Nominal.tex

theory *Arity−Nominal*
imports *Arity Nominal−HOLCF*
begin

lemma *join-eqvt*[*eqvt*]: $\pi \cdot (x \sqcup (y :: 'a :: \{Finite\text{-}Join\text{-}cpo,\ cont\text{-}pt\})) = (\pi \cdot x) \sqcup (\pi \cdot y)$
  by (*rule is-joinI*[*symmetric*]) (*auto simp add*: *perm-below-to-right*)

instantiation *Arity* :: *pure*
begin
  definition $p \cdot (a::Arity) = a$
instance
  apply *standard*
  apply (*auto simp add*: *permute-Arity-def*)
  done
end

instance *Arity* :: *cont-pt* **by** *standard* (*simp add*: *pure-permute-id*)
instance *Arity* :: *pure-cont-pt* **..**

end

46 ArityAnalysisSig.tex

theory *ArityAnalysisSig*

imports *Terms AEnv Arity−Nominal Nominal−HOLCF Substitution*
begin

locale *ArityAnalysis* =
  fixes *Aexp* :: *exp* ⇒ *Arity* → *AEnv*
begin
  abbreviation *Aexp-syn* ($\mathcal{A}_{\_}$)**where** $\mathcal{A}_a$ *e* ≡ *Aexp e·a*
  abbreviation *Aexp-bot-syn* ($\mathcal{A}^{\perp}{}_{\_}$)
    where $\mathcal{A}^{\perp}{}_a$ *e* ≡ *fup·*(*Aexp e*)*·a*

end

locale *ArityAnalysisHeap* =
  fixes *Aheap* :: *heap* ⇒ *exp* ⇒ *Arity* → *AEnv*

locale *EdomArityAnalysis* = *ArityAnalysis* +
  assumes *Aexp-edom*: *edom* ($\mathcal{A}_a$ *e*) ⊆ *fv e*
begin

  lemma *fup-Aexp-edom*: *edom* ($\mathcal{A}^{\perp}{}_a$ *e*) ⊆ *fv e*
    by (*cases a*) (*auto dest*:*set-mp*[*OF Aexp-edom*])

  lemma *Aexp-fresh-bot*[*simp*]: **assumes** *atom v* ♯ *e* **shows** $\mathcal{A}_a$ *e v* = ⊥
  proof−
    from *assms* **have** *v* ∉ *fv e* **by** (*metis fv-not-fresh*)
    with *Aexp-edom* **have** *v* ∉ *edom* ($\mathcal{A}_a$ *e*) **by** *auto*
    thus *?thesis* **unfolding** *edom-def* **by** *simp*
  qed
end

locale *ArityAnalysisHeapEqvt* = *ArityAnalysisHeap* +
  assumes *Aheap-eqvt*[*eqvt*]: π • *Aheap* = *Aheap*

end

47 ArityAnalysisAbinds.tex

theory *ArityAnalysisAbinds*
imports *ArityAnalysisSig*
begin

context *ArityAnalysis*
begin

47.1 Lifting arity analysis to recursive groups

definition *ABind* :: *var* ⇒ *exp* ⇒ (*AEnv* → *AEnv*)
  where *ABind v e* = (Λ *ae*. *fup·*(*Aexp e*)*·*(*ae v*))

lemma *ABind-eq*[*simp*]: *ABind v e* · *ae* = $\mathcal{A}^{\perp}{}_{ae\ v}$ *e*
  unfolding *ABind-def* **by** (*simp add*: *cont-fun*)

fun *ABinds* :: *heap* ⇒ (*AEnv* → *AEnv*)
  where *ABinds* [] = ⊥
    | *ABinds* ((*v*,*e*)#*binds*) = *ABind v e* ⊔ *ABinds* (*delete v binds*)

lemma *ABinds-strict*[*simp*]: *ABinds* Γ·⊥=⊥
  by (*induct* Γ *rule*: *ABinds.induct*) *auto*

lemma *Abinds-reorder1*: *map-of* Γ *v* = *Some e* ⟹ *ABinds* Γ = *ABind v e* ⊔ *ABinds* (*delete v* Γ)
  by (*induction* Γ *rule*: *ABinds.induct*) (*auto simp add*: *delete-twist*)

lemma *ABind-below-ABinds*: *map-of* Γ *v* = *Some e* ⟹ *ABind v e* ⊑ *ABinds* Γ
  by (*metis HOLCF−Join−Classes.join-above1 ArityAnalysis.Abinds-reorder1*)

lemma *Abinds-reorder*: *map-of* Γ = *map-of* Δ ⟹ *ABinds* Γ = *ABinds* Δ
proof (*induction* Γ *arbitrary*: Δ *rule*: *ABinds.induct*)
  case *1* **thus** *?case* **by** *simp*
next
  case (*2 v e* Γ Δ)
  from ‹*map-of* ((*v*, *e*) # Γ) = *map-of* Δ›
  have (*map-of* ((*v*, *e*) # Γ))(*v* := *None*) = (*map-of* Δ)(*v* := *None*) **by** *simp*
  hence *map-of* (*delete v* Γ) = *map-of* (*delete v* Δ) **unfolding** *delete-set-none* **by** *simp*
  hence *ABinds* (*delete v* Γ) = *ABinds* (*delete v* Δ) **by** (*rule 2*)
  moreover
  from ‹*map-of* ((*v*, *e*) # Γ) = *map-of* Δ›
  have *map-of* Δ *v* = *Some e* **by** (*metis map-of-Cons-code*(*2*))
  hence *ABinds* Δ = *ABind v e* ⊔ *ABinds* (*delete v* Δ) **by** (*rule Abinds-reorder1*)
  ultimately
  show *?case* **by** *auto*
qed

lemma *Abinds-env-cong*: (⋀ *x*. *x*∈*domA* Δ ⟹ *ae x* = *ae′ x*) ⟹ *ABinds* Δ·*ae* = *ABinds* Δ·*ae′*
  by (*induct* Δ *rule*: *ABinds.induct*) *auto*

lemma *Abinds-env-restr-cong*: *ae f*|‘ *domA* Δ = *ae′ f*|‘ *domA* Δ ⟹ *ABinds* Δ·*ae* = *ABinds* Δ·*ae′*
  by (*rule Abinds-env-cong*) (*metis env-restr-eqD*)

lemma *ABinds-env-restr*[*simp*]: *ABinds* Δ·(*ae f*|‘ *domA* Δ) = *ABinds* Δ·*ae*
  by (*rule Abinds-env-restr-cong*) *simp*

lemma *Abinds-join-fresh*: *ae′* ‘ (*domA* Δ) ⊆ {⊥} ⟹ *ABinds* Δ·(*ae* ⊔ *ae′*) = (*ABinds* Δ·*ae*)

by (*rule Abinds-env-cong*) *auto*

lemma *ABinds-delete-bot*: $ae\ x = \bot \Longrightarrow ABinds\ (delete\ x\ \Gamma) \cdot ae = ABinds\ \Gamma \cdot ae$
by (*induction* Γ *rule*: *ABinds.induct*) (*auto simp add*: *delete-twist*)

lemma *ABinds-restr-fresh*:
assumes $atom\ `\ S \sharp{*}\ \Gamma$
shows $ABinds\ \Gamma \cdot ae\ f|`\ (-\ S) = ABinds\ \Gamma \cdot (ae\ \ f|`\ (-\ S))\ f|`\ (-\ S)$
using *assms*
apply (*induction* Γ *rule*:*ABinds.induct*)
apply *simp*
apply (*auto simp del*: *fun-meet-simp simp add*: *env-restr-join fresh-star-Pair fresh-star-Cons fresh-star-delete*)
apply (*subst lookup-env-restr*)
apply (*metis* (*no-types*, *hide-lams*) *ComplI fresh-at-base*(*2*) *fresh-star-def imageI*)
apply *simp*
done

lemma *ABinds-restr*:
assumes $domA\ \Gamma \subseteq S$
shows $ABinds\ \Gamma \cdot ae\ f|`\ S = ABinds\ \Gamma \cdot (ae\ \ f|`\ S)\ f|`\ S$
using *assms*
by (*induction* Γ *rule*:*ABinds.induct*) (*fastforce simp del*: *fun-meet-simp simp add*: *env-restr-join*)+

lemma *ABinds-restr-subst*:
assumes $\bigwedge x'\ e\ a.\ (x',e) \in set\ \Gamma \Longrightarrow Aexp\ e[x::=y] \cdot a\ f|`\ S = Aexp\ e \cdot a\ f|`\ S$
assumes $x \notin S$
assumes $y \notin S$
assumes $domA\ \Gamma \subseteq S$
shows $ABinds\ \Gamma[x::h=y] \cdot ae\ f|`\ S = ABinds\ \Gamma \cdot (ae\ \ f|`\ S)\ f|`\ S$
using *assms*
apply (*induction* Γ *rule*:*ABinds.induct*)
apply (*auto simp del*: *fun-meet-simp join-comm simp add*: *env-restr-join*)
apply (*rule arg-cong2*[**where** *f* = *join*])
apply (*case-tac ae v*)
apply (*auto dest*: *set-mp*[*OF set-delete-subset*])
done

lemma *Abinds-append-disjoint*: $domA\ \Delta \cap domA\ \Gamma = \{\} \Longrightarrow ABinds\ (\Delta\ @\ \Gamma) \cdot ae = ABinds\ \Delta \cdot ae \sqcup ABinds\ \Gamma \cdot ae$
proof (*induct* Δ *rule*: *ABinds.induct*)
case *1* **thus** *?case* **by** *simp*
next
case (*2 v e* Δ)
from *2*(*2*)
have $v \notin domA\ \Gamma$ **and** $domA\ (delete\ v\ \Delta) \cap domA\ \Gamma = \{\}$ **by** *auto*
from *2*(*1*)[*OF this*(*2*)]
have $ABinds\ (delete\ v\ \Delta\ @\ \Gamma) \cdot ae = ABinds\ (delete\ v\ \Delta) \cdot ae \sqcup ABinds\ \Gamma \cdot ae$.
moreover

have *delete v* $\Gamma = \Gamma$ **by** (*metis* ‹$v \notin$ *domA* Γ› *delete-not-domA*)
ultimately
show *ABinds* $(((v,\ e)\ \#\ \Delta)\ @\ \Gamma)\cdot ae$ = *ABinds* $((v,\ e)\ \#\ \Delta)\cdot ae \sqcup$ *ABinds* $\Gamma\cdot ae$
by *auto*
qed

lemma *ABinds-restr-subset*: $S \subseteq S' \Longrightarrow$ *ABinds* (*restrictA* S $\Gamma)\cdot ae \sqsubseteq$ *ABinds* (*restrictA* S' $\Gamma)\cdot ae$
by (*induct* Γ *rule*: *ABinds.induct*)
(*auto simp add*: *join-below-iff restr-delete-twist intro*: *below-trans*[*OF - join-above2*])

lemma *ABinds-restrict-edom*: *ABinds* (*restrictA* (*edom ae*) $\Gamma)\cdot ae$ = *ABinds* $\Gamma\cdot ae$
by (*induct* Γ *rule*: *ABinds.induct*) (*auto simp add*: *edom-def restr-delete-twist*)

lemma *ABinds-restrict-below*: *ABinds* (*restrictA* S $\Gamma)\cdot ae \sqsubseteq$ *ABinds* $\Gamma\cdot ae$
by (*induct* Γ *rule*: *ABinds.induct*)
(*auto simp add*: *join-below-iff restr-delete-twist intro*: *below-trans*[*OF - join-above2*] *simp del*: *fun-meet-simp join-comm*)

lemma *ABinds-delete-below*: *ABinds* (*delete x* $\Gamma)\cdot ae \sqsubseteq$ *ABinds* $\Gamma\cdot ae$
by (*induct* Γ *rule*: *ABinds.induct*)
(*auto simp add*: *join-below-iff delete-twist*[**where** $x = x$] *elim*: *below-trans simp del*: *fun-meet-simp*)
end

lemma *ABind-eqvt*[*eqvt*]: $\pi \cdot$ (*ArityAnalysis.ABind Aexp v e*) = *ArityAnalysis.ABind* ($\pi \cdot$ *Aexp*) ($\pi \cdot v$) ($\pi \cdot e$)
apply (*rule cfun-eqvtI*)
unfolding *ArityAnalysis.ABind-eq*
by *perm-simp rule*

lemma *ABinds-eqvt*[*eqvt*]: $\pi \cdot$ (*ArityAnalysis.ABinds Aexp* Γ) = *ArityAnalysis.ABinds* ($\pi \cdot$ *Aexp*) ($\pi \cdot \Gamma$)
apply (*rule cfun-eqvtI*)
apply (*induction* Γ *rule*: *ArityAnalysis.ABinds.induct*)
apply (*simp add*: *ArityAnalysis.ABinds.simps*)
apply (*simp add*: *ArityAnalysis.ABinds.simps*)
apply *perm-simp*
apply *simp*
done

lemma *Abinds-cong*[*fundef-cong*]:
$[\![$ ($\bigwedge$ *e*. $e \in$ *snd* ‘ *set heap2* $\Longrightarrow$ *aexp1 e* = *aexp2 e*) ; *heap1* = *heap2* $]\!]$
$\Longrightarrow$ *ArityAnalysis.ABinds aexp1 heap1* = *ArityAnalysis.ABinds aexp2 heap2*
proof (*induction heap1 arbitrary*:*heap2 rule*:*ArityAnalysis.ABinds.induct*)
case *1*
thus *?case* **by** (*auto simp add*: *ArityAnalysis.ABinds.simps*)
next
case *prems*: (*2 v e as heap2*)

have *snd ‘ set* (*delete v as*) ⊆ *snd ‘ set as* **by** (*rule dom-delete-subset*)
also have ... ⊆ *snd ‘set* ((*v*, *e*) # *as*) **by** *auto*
also note *prems*(*3*)
finally
have (⋀*e*. *e* ∈ *snd ‘ set* (*delete v as*) ⟹ *aexp1 e* = *aexp2 e*) **by** −(*rule prems*, *auto*)
from *prems prems*(*1*)[*OF this refl*] **show** *?case*
by (*auto simp add*: *ArityAnalysis.ABinds.simps ArityAnalysis.ABind-def*)
qed

context *EdomArityAnalysis*
begin
lemma *fup-Aexp-lookup-fresh*: *atom v* ♯ *e* ⟹ (*fup*·(*Aexp e*)·*a*) *v* = ⊥
by (*cases a*) *auto*

lemma *edom-AnalBinds*: *edom* (*ABinds* Γ·*ae*) ⊆ *fv* Γ
by (*induction* Γ *rule*: *ABinds.induct*)
(*auto simp del*: *fun-meet-simp dest*: *set-mp*[*OF fup-Aexp-edom*] *dest*: *set-mp*[*OF fv-delete-subset*])
end

end

48 ArityAnalysisSpec.tex

theory *ArityAnalysisSpec*
imports *ArityAnalysisAbinds*
begin

locale *SubstArityAnalysis* = *EdomArityAnalysis* +
assumes *Aexp-subst-restr*: *x* ∉ *S* ⟹ *y* ∉ *S* ⟹ (*Aexp e*[*x*::=*y*] · *a*) *f*|‘ *S* = (*Aexp e*·*a*) *f*|‘ *S*

locale *ArityAnalysisSafe* = *SubstArityAnalysis* +
assumes *Aexp-Var*: *up* · *n* ⊑ (*Aexp* (*Var x*)·*n*) *x*
assumes *Aexp-App*: *Aexp e* ·(*inc*·*n*) ⊔ *esing x* · (*up*·*0*) ⊑ *Aexp* (*App e x*) · *n*
assumes *Aexp-Lam*: *env-delete y* (*Aexp e* ·(*pred*·*n*)) ⊑ *Aexp* (*Lam* [*y*]. *e*) · *n*
assumes *Aexp-IfThenElse*: *Aexp scrut*·*0* ⊔ *Aexp e1*·*a* ⊔ *Aexp e2*·*a* ⊑ *Aexp* (*scrut ? e1* : *e2*)·*a*

locale *ArityAnalysisHeapSafe* = *ArityAnalysisSafe* + *ArityAnalysisHeapEqvt* +
assumes *edom-Aheap*: *edom* (*Aheap* Γ *e*· *a*) ⊆ *domA* Γ
assumes *Aheap-subst*: *x* ∉ *domA* Γ ⟹ *y* ∉ *domA* Γ ⟹ *Aheap* Γ[*x*::*h*=*y*] *e*[*x* ::=*y*] = *Aheap* Γ *e*

locale *ArityAnalysisLetSafe* = *ArityAnalysisHeapSafe* +
assumes *Aexp-Let*: *ABinds* Γ·(*Aheap* Γ *e*·*a*) ⊔ *Aexp e*·*a* ⊑ *Aheap* Γ *e*·*a* ⊔ *Aexp* (*Let* Γ *e*)·*a*

locale *ArityAnalysisLetSafeNoCard* = *ArityAnalysisLetSafe* +
assumes *Aheap-heap3*: *x* ∈ *thunks* Γ ⟹ (*Aheap* Γ *e*·*a*) *x* = *up*·*0*

context *SubstArityAnalysis*
begin
lemma *Aexp-subst-upd*: $(Aexp\ e[y{::=}x]\cdot n) \sqsubseteq (Aexp\ e\cdot n)(y := \bot,\ x := up\cdot 0)$
proof−
have $Aexp\ e[y{::=}x]\cdot n\ f|`(-\{x,y\}) = Aexp\ e\cdot n\ f|`\ (-\{x,y\})$ **by** (*rule Aexp-subst-restr*) *auto*

show *?thesis*
proof (*rule fun-belowI*)
fix x'
have $x' = x \lor x' = y \lor x' \in (-\{x,y\})$ **by** *auto*
thus $(Aexp\ e[y{::=}x]\cdot n)\ x' \sqsubseteq ((Aexp\ e\cdot n)(y := \bot,\ x := up\cdot 0))\ x'$
proof(*elim disjE*)
assume $x' \in (-\{x,y\})$
moreover
have $Aexp\ e[y{::=}x]\cdot n\ f|`(-\{x,y\}) = Aexp\ e\cdot n\ f|`\ (-\{x,y\})$ **by** (*rule Aexp-subst-restr*) *auto*
note *fun-cong*[*OF this*, **where** $x = x'$]
ultimately
show *?thesis* **by** *auto*
next
assume $x' = x$
thus *?thesis* **by** *simp*
next
assume $x' = y$
thus *?thesis*
using [[*simp-trace*]]
by *simp*
qed
qed
qed

lemma *Aexp-subst*: $Aexp\ (e[y{::=}x])\cdot a \sqsubseteq env\text{-}delete\ y\ ((Aexp\ e)\cdot a) \sqcup esing\ x\cdot(up\cdot 0)$
apply (*rule below-trans*[*OF Aexp-subst-upd*])
apply (*rule fun-belowI*)
apply *auto*
done
end

context *ArityAnalysisSafe*
begin

lemma *Aexp-Var-singleton*: $esing\ x \cdot (up\cdot n) \sqsubseteq Aexp\ (Var\ x) \cdot n$
by (*simp add*: *Aexp-Var*)

lemma *fup-Aexp-Var*: $esing\ x \cdot n \sqsubseteq fup\cdot(Aexp\ (Var\ x))\cdot n$
by (*cases n*) (*simp-all add*: *Aexp-Var*)
end

context *ArityAnalysisLetSafe*
begin
  lemma *Aheap-nonrec*:
    assumes *nonrec* Δ
    shows *Aexp* $e{\cdot}a$ $f|`$ *domA* $\Delta \sqsubseteq$ *Aheap* Δ $e{\cdot}a$
  proof−
    have *ABinds* $\Delta{\cdot}$(*Aheap* Δ $e{\cdot}a$) $\sqcup$ *Aexp* $e{\cdot}a \sqsubseteq$ *Aheap* Δ $e{\cdot}a \sqcup$ *Aexp* (*Let* Δ e)$\cdot a$ **by** (*rule Aexp-Let*)
    note *env-restr-mono*[**where** S = *domA* Δ, *OF this*]
    moreover
    from *assms*
    have *ABinds* $\Delta{\cdot}$(*Aheap* Δ $e{\cdot}a$) $f|`$ *domA* $\Delta = \bot$
      by (*rule nonrecE*) (*auto simp add*: *fv-def fresh-def dest*!: *set-mp*[*OF fup-Aexp-edom*])
    moreover
    have *Aheap* Δ $e{\cdot}a$ $f|`$ *domA* Δ = *Aheap* Δ $e{\cdot}a$
      by (*rule env-restr-useless*[*OF edom-Aheap*])
    moreover
    have (*Aexp* (*Let* Δ e)$\cdot a$) $f|`$ *domA* $\Delta = \bot$
      by (*auto dest*!: *set-mp*[*OF Aexp-edom*])
    ultimately
    show *Aexp* $e{\cdot}a$ $f|`$ *domA* $\Delta \sqsubseteq$ *Aheap* Δ $e{\cdot}a$
      by (*simp add*: *env-restr-join*)
  qed
end

end

49 TrivialArityAnal.tex

theory *TrivialArityAnal*
imports *ArityAnalysisSpec Env*−*Nominal*
begin

definition *Trivial-Aexp* :: *exp* $\Rightarrow$ *Arity* $\to$ *AEnv*
  where *Trivial-Aexp* e = (Λ n. (λ x. *up*$\cdot$*0*) $f|`$ *fv* e)

lemma *Trivial-Aexp-simp*: *Trivial-Aexp* $e \cdot n$ = (λ x. *up*$\cdot$*0*) $f|`$ *fv* e
  unfolding *Trivial-Aexp-def* **by** *simp*

lemma *edom-Trivial-Aexp*[*simp*]: *edom* (*Trivial-Aexp* $e \cdot n$) = *fv* e
  by (*auto simp add*: *edom-def env-restr-def Trivial-Aexp-def*)

lemma *Trivial-Aexp-eq*[*iff*]: *Trivial-Aexp* $e \cdot n$ = *Trivial-Aexp* $e' \cdot n' \longleftrightarrow$ *fv* e = (*fv* e' :: *var set*)
  apply (*auto simp add*: *Trivial-Aexp-simp env-restr-def*)
  apply (*metis up-defined*)+

done

lemma *below-Trivial-Aexp*[*simp*]: $(ae \sqsubseteq \textit{Trivial-Aexp}\ e \cdot n) \longleftrightarrow \textit{edom}\ ae \subseteq \textit{fv}\ e$
by (*auto dest*:*fun-belowD intro*!: *fun-belowI simp add*: *Trivial-Aexp-def env-restr-def edom-def split*:*if-splits*)

interpretation *ArityAnalysis Trivial-Aexp*.
interpretation *EdomArityAnalysis Trivial-Aexp*
by *standard simp*

interpretation *ArityAnalysisSafe Trivial-Aexp*
proof

fix *n x*
show $up \cdot n \sqsubseteq (\textit{Trivial-Aexp}\ (\textit{Var}\ x) \cdot n)\ x$
by (*simp add*: *Trivial-Aexp-simp*)
next
fix *e x n*
show $\textit{Trivial-Aexp}\ e \cdot (inc \cdot n) \sqcup \textit{esing}\ x \cdot (up \cdot 0) \sqsubseteq \textit{Trivial-Aexp}\ (\textit{App}\ e\ x) \cdot n$
by (*auto intro*: *fun-belowI simp add*: *Trivial-Aexp-def env-restr-def*)
next
fix *y e n*
show $\textit{env-delete}\ y\ (\textit{Trivial-Aexp}\ e \cdot (pred \cdot n)) \sqsubseteq \textit{Trivial-Aexp}\ (\textit{Lam}\ [y].\ e) \cdot n$
by (*auto simp add*: *Trivial-Aexp-simp env-delete-restr Diff-eq inf-commute*)
next
fix *x y* :: *var* **and** *S e a*
assume $x \notin S$ **and** $y \notin S$
thus $\textit{Trivial-Aexp}\ e[x::=y] \cdot a\ f|`\ S = \textit{Trivial-Aexp}\ e \cdot a\ f|`\ S$
by (*auto simp add*: *Trivial-Aexp-simp fv-subst-eq intro*!: *arg-cong*[**where** $f = \lambda\ S.\ \textit{env-restr}\ S\ e$ **for** *e*])
next
fix *scrut e1 a e2*
show $\textit{Trivial-Aexp}\ scrut \cdot 0 \sqcup \textit{Trivial-Aexp}\ e1 \cdot a \sqcup \textit{Trivial-Aexp}\ e2 \cdot a \sqsubseteq \textit{Trivial-Aexp}\ (scrut\ ?\ e1 : e2) \cdot a$
by (*auto intro*: *env-restr-mono2 simp add*: *Trivial-Aexp-simp join-below-iff*)
qed

definition *Trivial-Aheap* :: $heap \Rightarrow exp \Rightarrow Arity \rightarrow AEnv$ **where**
$\textit{Trivial-Aheap}\ \Gamma\ e = (\Lambda\ a.\ (\lambda\ x.\ up \cdot 0)\ f|`\ \textit{domA}\ \Gamma)$

lemma *Trivial-Aheap-eqvt*[*eqvt*]: $\pi \bullet (\textit{Trivial-Aheap}\ \Gamma\ e) = \textit{Trivial-Aheap}\ (\pi \bullet \Gamma)\ (\pi \bullet e)$
unfolding *Trivial-Aheap-def*
apply *perm-simp*
apply (*simp add*: *Abs-cfun-eqvt*)
done

lemma *Trivial-Aheap-simp*: $\textit{Trivial-Aheap}\ \Gamma\ e \cdot a = (\lambda\ x.\ up \cdot 0)\ f|`\ \textit{domA}\ \Gamma$

unfolding *Trivial-Aheap-def* **by** *simp*

lemma *Trivial-fup-Aexp-below-fv*: $fup\cdot(Trivial\text{-}Aexp\ e)\cdot a \sqsubseteq (\lambda\ x\ .\ up\cdot 0)\ f|`\ fv\ e$
by (*cases a*)(*auto simp add*: *Trivial-Aexp-simp*)

lemma *Trivial-Abinds-below-fv*: $ABinds\ \Gamma\cdot ae \sqsubseteq (\lambda\ x\ .\ up\cdot 0)\ f|`\ fv\ \Gamma$
by (*induction* Γ *rule*:*ABinds.induct*)
(*auto simp add*: *join-below-iff intro*!: *below-trans*[*OF Trivial-fup-Aexp-below-fv*] *env-restr-mono2 elim*: *below-trans dest*: *set-mp*[*OF fv-delete-subset*] *simp del*: *fun-meet-simp*)

interpretation *ArityAnalysisLetSafe Trivial-Aexp Trivial-Aheap*
proof
fix π
show $\pi \cdot Trivial\text{-}Aheap = Trivial\text{-}Aheap$ **by** *perm-simp rule*
next
fix Γ *e ae* **show** $edom\ (Trivial\text{-}Aheap\ \Gamma\ e\cdot ae) \subseteq domA\ \Gamma$
by (*simp add*: *Trivial-Aheap-simp*)
next
fix Γ :: *heap* **and** *e* **and** *a*
show $ABinds\ \Gamma\cdot(Trivial\text{-}Aheap\ \Gamma\ e\cdot a) \sqcup Trivial\text{-}Aexp\ e\cdot a \sqsubseteq Trivial\text{-}Aheap\ \Gamma\ e\cdot a \sqcup Trivial\text{-}Aexp\ (Terms.Let\ \Gamma\ e)\cdot a$
by (*auto simp add*: *Trivial-Aheap-simp Trivial-Aexp-simp join-below-iff env-restr-join2 intro*!: *env-restr-mono2 below-trans*[*OF Trivial-Abinds-below-fv*])
next
fix *x y* :: *var* **and** Γ :: *heap* **and** *e*
assume $x \notin domA\ \Gamma$ **and** $y \notin domA\ \Gamma$
thus $Trivial\text{-}Aheap\ \Gamma[x::h=y]\ e[x::=y] = Trivial\text{-}Aheap\ \Gamma\ e$
by (*auto intro*: *cfun-eqI simp add*: *Trivial-Aheap-simp*)
qed

end

50 Cardinality-Domain.tex

theory *Cardinality−Domain*
imports *HOLCF−Utils*
begin

type-synonym *oneShot* = *one*
abbreviation *notOneShot* :: *oneShot* **where** *notOneShot* ≡ *ONE*
abbreviation *oneShot* :: *oneShot* **where** *oneShot* ≡ ⊥

type-synonym *two* = $oneShot_{\bot}$
abbreviation *many* :: *two* **where** $many \equiv up\cdot notOneShot$
abbreviation *once* :: *two* **where** $once \equiv up\cdot oneShot$
abbreviation *none* :: *two* **where** *none* ≡ ⊥

lemma *many-max*[*simp*]: $a \sqsubseteq many$ **by** (*cases a*) *auto*

lemma *two-conj*: $c = many \lor c = once \lor c = none$ **by** (*metis Exh-Up one-neq-iffs*(*1*))

lemma *two-cases*[*case-names many once none*]:
  obtains $c = many \mid c = once \mid c = none$ **using** *two-conj* **by** *metis*

definition *two-pred* **where** $two\text{-}pred = (\Lambda\ x.\ if\ x \sqsubseteq once\ then\ \perp\ else\ x)$

lemma *two-pred-simp*: $two\text{-}pred\cdot c = (if\ c \sqsubseteq once\ then\ \perp\ else\ c)$
  unfolding *two-pred-def*
  apply (*rule beta-cfun*)
  apply (*rule cont-if-else-above*)
  apply (*auto elim*: *below-trans*)
  done

lemma *two-pred-simps*[*simp*]:
  $two\text{-}pred\cdot many = many$
  $two\text{-}pred\cdot once = none$
  $two\text{-}pred\cdot none = none$
by (*simp-all add*: *two-pred-simp*)

lemma *two-pred-below-arg*: $two\text{-}pred \cdot f \sqsubseteq f$
  by (*auto simp add*: *two-pred-simp*)

lemma *two-pred-none*: $two\text{-}pred\cdot c = none \longleftrightarrow c \sqsubseteq once$
  by (*auto simp add*: *two-pred-simp*)

definition *record-call* **where** $record\text{-}call\ x = (\Lambda\ ce.\ (\lambda\ y.\ if\ x = y\ then\ two\text{-}pred\cdot(ce\ y)\ else\ ce\ y))$

lemma *record-call-simp*: $(record\text{-}call\ x \cdot f)\ x' = (if\ x = x'\ then\ two\text{-}pred \cdot (f\ x')\ else\ f\ x')$
  unfolding *record-call-def* **by** *auto*

lemma *record-call*[*simp*]: $(record\text{-}call\ x \cdot f)\ x = two\text{-}pred \cdot (f\ x)$
  unfolding *record-call-simp* **by** *auto*

lemma *record-call-other*[*simp*]: $x' \neq x \Longrightarrow (record\text{-}call\ x \cdot f)\ x' = f\ x'$
  unfolding *record-call-simp* **by** *auto*

lemma *record-call-below-arg*: $record\text{-}call\ x \cdot f \sqsubseteq f$
  unfolding *record-call-def*
  by (*auto intro*!: *fun-belowI two-pred-below-arg*)

definition *two-add* :: $two \rightarrow two \rightarrow two$
  where $two\text{-}add = (\Lambda\ x.\ (\Lambda\ y.\ if\ x \sqsubseteq \perp\ then\ y\ else\ (if\ y \sqsubseteq \perp\ then\ x\ else\ many)))$

lemma *two-add-simp*: $two\text{-}add\cdot x\cdot y = (if\ x \sqsubseteq \perp\ then\ y\ else\ (if\ y \sqsubseteq \perp\ then\ x\ else\ many))$
  unfolding *two-add-def*
  apply (*subst beta-cfun*)

apply (*rule cont2cont*)
apply (*rule cont-if-else-above*)
apply (*auto elim*: *below-trans*)[*1*]
apply (*rule cont-if-else-above*)
apply (*auto elim*: *below-trans*)[*8*]
apply (*rule beta-cfun*)
apply (*rule cont-if-else-above*)
apply (*auto elim*: *below-trans*)[*1*]
apply (*rule cont-if-else-above*)
apply *auto*
done

lemma *two-pred-two-add-once*: $c \sqsubseteq$ *two-pred*$\cdot$(*two-add*$\cdot$*once*$\cdot$*c*)
by (*cases c rule*: *two-cases*) (*auto simp add*: *two-add-simp*)

end

51 CardinalityAnalysisSig.tex

theory *CardinalityAnalysisSig*
imports *Arity AEnv Cardinality−Domain SestoftConf*
begin

locale *CardinalityPrognosis* =
fixes *prognosis* :: *AEnv* ⇒ *Arity list* ⇒ *Arity* ⇒ *conf* ⇒ (*var* ⇒ *two*)

locale *CardinalityHeap* =
fixes *cHeap* :: *heap* ⇒ *exp* ⇒ *Arity* → (*var* ⇒ *two*)
end

52 ConstOn.tex

theory *ConstOn*
imports *Main*
begin

definition *const-on* :: ('*a* ⇒ '*b*) ⇒ '*a set* ⇒ '*b* ⇒ *bool*
where *const-on f S x* = (∀ *y* ∈ *S* . *f y* = *x*)

lemma *const-onI*[*intro*]: ($\bigwedge y.\ y \in S \Longrightarrow f\ y = x$) $\Longrightarrow$ *const-on f S x*
by (*simp add*: *const-on-def*)

lemma *const-onD*[*dest*]: *const-on f S x* $\Longrightarrow y \in S \Longrightarrow f\ y = x$
by (*simp add*: *const-on-def*)

lemma *const-on-insert*[*simp*]: *const-on f* (*insert x S*) *y* $\longleftrightarrow$ *const-on f S y* $\wedge$ *f x* = *y*
by *auto*

lemma *const-on-union*[*simp*]: *const-on f* ($S \cup S'$) *y* $\longleftrightarrow$ *const-on f S y* $\wedge$ *const-on f S' y*
by *auto*

lemma *const-on-subset*[*elim*]: *const-on f S y* $\Longrightarrow$ $S' \subseteq S$ $\Longrightarrow$ *const-on f S' y*
by *auto*

end

53 CardinalityAnalysisSpec.tex

theory *CardinalityAnalysisSpec*
imports *ArityAnalysisSpec CardinalityAnalysisSig ConstOn*
begin

locale *CardinalityPrognosisEdom* = *CardinalityPrognosis* +
assumes *edom-prognosis*:
edom (*prognosis ae as a* (Γ, *e*, *S*)) $\subseteq$ *fv* Γ $\cup$ *fv e* $\cup$ *fv S*

locale *CardinalityPrognosisShape* = *CardinalityPrognosis* +
assumes *prognosis-env-cong*: *ae f*|' *domA* Γ = *ae' f*|' *domA* Γ $\Longrightarrow$ *prognosis ae as u* (Γ, *e*, *S*) = *prognosis ae' as u* (Γ, *e*, *S*)
assumes *prognosis-reorder*: *map-of* Γ = *map-of* Δ $\Longrightarrow$ *prognosis ae as u* (Γ, *e*, *S*) = *prognosis ae as u* (Δ, *e*, *S*)
assumes *prognosis-ap*: *const-on* (*prognosis ae as a* (Γ, *e*, *S*)) (*ap S*) *many*
assumes *prognosis-upd*: *prognosis ae as u* (Γ, *e*, *S*) $\sqsubseteq$ *prognosis ae as u* (Γ, *e*, *Upd x* # *S*)
assumes *prognosis-not-called*: *ae x* = $\bot$ $\Longrightarrow$ *prognosis ae as a* (Γ, *e*, *S*) $\sqsubseteq$ *prognosis ae as a* (*delete x* Γ, *e*, *S*)
assumes *prognosis-called*: *once* $\sqsubseteq$ *prognosis ae as a* (Γ, *Var x*, *S*) *x*

locale *CardinalityPrognosisApp* = *CardinalityPrognosis* +
assumes *prognosis-App*: *prognosis ae as* (*inc*$\cdot$*a*) (Γ, *e*, *Arg x* # *S*) $\sqsubseteq$ *prognosis ae as a* (Γ, *App e x*, *S*)

locale *CardinalityPrognosisLam* = *CardinalityPrognosis* +
assumes *prognosis-subst-Lam*: *prognosis ae as* (*pred*$\cdot$*a*) (Γ, *e*[*y*::=*x*], *S*) $\sqsubseteq$ *prognosis ae as a* (Γ, *Lam* [*y*]. *e*, *Arg x* # *S*)

locale *CardinalityPrognosisVar* = *CardinalityPrognosis* +
assumes *prognosis-Var-lam*: *map-of* Γ *x* = *Some e* $\Longrightarrow$ *ae x* = *up*$\cdot$*u* $\Longrightarrow$ *isVal e* $\Longrightarrow$ *prognosis ae as u* (Γ, *e*, *S*) $\sqsubseteq$ *record-call x* $\cdot$ (*prognosis ae as a* (Γ, *Var x*, *S*))
assumes *prognosis-Var-thunk*: *map-of* Γ *x* = *Some e* $\Longrightarrow$ *ae x* = *up*$\cdot$*u* $\Longrightarrow$ $\neg$ *isVal e* $\Longrightarrow$ *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) $\sqsubseteq$ *record-call x* $\cdot$ (*prognosis ae as a* (Γ, *Var x*,

S))
assumes *prognosis-Var2*: *isVal e* $\Longrightarrow$ *x* $\notin$ *domA* Γ $\Longrightarrow$ *prognosis ae as 0* ((*x*, *e*) # Γ, *e*, *S*) $\sqsubseteq$ *prognosis ae as 0* (Γ, *e*, *Upd x* # *S*)

locale *CardinalityPrognosisIfThenElse* = *CardinalityPrognosis* +
assumes *prognosis-IfThenElse*: *prognosis ae* (*a*#*as*) *0* (Γ, *scrut*, *Alts e1 e2* # *S*) $\sqsubseteq$ *prognosis ae as a* (Γ, *scrut ? e1* : *e2*, *S*)
assumes *prognosis-Alts*: *prognosis ae as a* (Γ, *if b then e1 else e2*, *S*) $\sqsubseteq$ *prognosis ae* (*a*#*as*) *0* (Γ, *Bool b*, *Alts e1 e2* # *S*)

locale *CardinalityPrognosisLet* = *CardinalityPrognosis* + *CardinalityHeap* + *ArityAnalysisHeap* +
assumes *prognosis-Let*:
atom ‘ *domA* Δ $\sharp*$ Γ $\Longrightarrow$ *atom* ‘ *domA* Δ $\sharp*$ *S* $\Longrightarrow$ *edom ae* $\subseteq$ *domA* Γ $\cup$ *upds S* $\Longrightarrow$ *prognosis* (*Aheap* Δ *e*$\cdot$*a* $\sqcup$ *ae*) *as a* (Δ @ Γ, *e*, *S*) $\sqsubseteq$ *cHeap* Δ *e*$\cdot$*a* $\sqcup$ *prognosis ae as a* (Γ, *Terms.Let* Δ *e*, *S*)

locale *CardinalityHeapSafe* = *CardinalityHeap* + *ArityAnalysisHeap* +
assumes *Aheap-heap3*: *x* $\in$ *thunks* Γ $\Longrightarrow$ *many* $\sqsubseteq$ (*cHeap* Γ *e*$\cdot$*a*) *x* $\Longrightarrow$ (*Aheap* Γ *e*$\cdot$*a*) *x* = *up*$\cdot$*0*
assumes *edom-cHeap*: *edom* (*cHeap* Δ *e*$\cdot$*a*) = *edom* (*Aheap* Δ *e*$\cdot$*a*)

locale *CardinalityPrognosisSafe* =
CardinalityPrognosisEdom +
CardinalityPrognosisShape +
CardinalityPrognosisApp +
CardinalityPrognosisLam +
CardinalityPrognosisVar +
CardinalityPrognosisLet +
CardinalityPrognosisIfThenElse +
CardinalityHeapSafe +
ArityAnalysisLetSafe

end

54 ArityAnalysisStack.tex

theory *ArityAnalysisStack*
imports *SestoftConf ArityAnalysisSig*
begin

context *ArityAnalysis*
begin
fun *AEstack* :: *Arity list* $\Rightarrow$ *stack* $\Rightarrow$ *AEnv*
where
AEstack - [] = $\bot$
| *AEstack* (*a*#*as*) (*Alts e1 e2* # *S*) = *Aexp e1*$\cdot$*a* $\sqcup$ *Aexp e2*$\cdot$*a* $\sqcup$ *AEstack as S*
| *AEstack as* (*Upd x* # *S*) = *esing x*$\cdot$(*up*$\cdot$*0*) $\sqcup$ *AEstack as S*

| *AEstack as* (*Arg x* # *S*) = *esing x*·(*up*·*0*) ⊔ *AEstack as S*
| *AEstack as* (- # *S*) = *AEstack as S*
end

context *EdomArityAnalysis*
begin
lemma *edom-AEstack*: *edom* (*AEstack as S*) ⊆ *fv S*
by (*induction as S rule*: *AEstack.induct*) (*auto simp del*: *fun-meet-simp dest*!: *set-mp*[*OF Aexp-edom*])
end

end

55 NoCardinalityAnalysis.tex

theory *NoCardinalityAnalysis*
imports *CardinalityAnalysisSpec ArityAnalysisStack*
begin

locale *NoCardinalityAnalysis* = *ArityAnalysisLetSafe* +
assumes *Aheap-thunk*: $x \in thunks\ \Gamma \Longrightarrow (Aheap\ \Gamma\ e{\cdot}a)\ x = up{\cdot}0$
begin

definition *a2c* :: $Arity_\bot \to two$ **where** $a2c = (\Lambda\ a.\ if\ a \sqsubseteq \bot\ then\ \bot\ else\ many)$
lemma *a2c-simp*: $a2c{\cdot}a = (if\ a \sqsubseteq \bot\ then\ \bot\ else\ many)$
unfolding *a2c-def* **by** (*rule beta-cfun*[*OF cont-if-else-above*]) *auto*

lemma *a2c-eqvt*[*eqvt*]: $\pi \cdot a2c = a2c$
unfolding *a2c-def*
apply *perm-simp*
apply (*rule Abs-cfun-eqvt*)
apply (*rule cont-if-else-above*)
apply *auto*
done

definition *ae2ce* :: *AEnv* ⇒ (*var* ⇒ *two*) **where** *ae2ce ae x* = *a2c*·(*ae x*)

lemma *ae2ce-cont*: *cont ae2ce*
by (*auto simp add*: *ae2ce-def*)
lemmas *cont-compose*[*OF ae2ce-cont*, *cont2cont*, *simp*]

lemma *ae2ce-eqvt*[*eqvt*]: $\pi \cdot ae2ce\ ae\ x = ae2ce\ (\pi \cdot ae)\ (\pi \cdot x)$
unfolding *ae2ce-def* **by** *perm-simp rule*

lemma *ae2ce-to-env-restr*: *ae2ce ae* = (λ- . *many*) *f*|‘ *edom ae*
by (*auto simp add*: *ae2ce-def lookup-env-restr-eq edom-def a2c-simp*)

lemma *edom-ae2ce*[*simp*]: *edom* (*ae2ce ae*) = *edom ae*
unfolding *edom-def*
by (*auto simp add*: *ae2ce-def a2c-simp*)

definition *cHeap* :: *heap* ⇒ *exp* ⇒ *Arity* → (*var* ⇒ *two*)
where *cHeap* Γ *e* = (Λ *a*. *ae2ce* (*Aheap* Γ *e*·*a*))
lemma *cHeap-simp*[*simp*]: *cHeap* Γ *e*·*a* = *ae2ce* (*Aheap* Γ *e*·*a*)
unfolding *cHeap-def* **by** *simp*

sublocale *CardinalityHeap cHeap*.

sublocale *CardinalityHeapSafe cHeap Aheap*
apply *standard*
apply (*erule Aheap-thunk*)
apply *simp*
done

fun *prognosis* **where**
prognosis ae as a (Γ, *e*, *S*) = ((λ-. *many*) *f*|‘ (*edom* (*ABinds* Γ·*ae*) ∪ *edom* (*Aexp e*·*a*) ∪ *edom* (*AEstack as S*)))

lemma *record-all-noop*[*simp*]:
record-call x·((λ-. *many*) *f*|‘ *S*) = (λ-. *many*) *f*|‘ *S*
by (*auto simp add*: *record-call-def lookup-env-restr-eq*)

lemma *const-on-restr-constI*[*intro*]:
S′ ⊆ *S* ⟹ *const-on* ((λ -. *x*) *f*|‘ *S*) *S*′ *x*
by *fastforce*

lemma *ap-subset-edom-AEstack*: *ap S* ⊆ *edom* (*AEstack as S*)
by (*induction as S rule*:*AEstack.induct*) (*auto simp del*: *fun-meet-simp*)

sublocale *CardinalityPrognosis prognosis*.

sublocale *CardinalityPrognosisShape prognosis*
proof (*standard*, *goal-cases*)
case *1*
thus *?case* **by** (*simp cong*: *Abinds-env-restr-cong*)
next
case *2*
thus *?case* **by** (*simp cong*: *Abinds-reorder*)
next
case *3*
thus *?case* **by** (*auto dest*: *set-mp*[*OF ap-subset-edom-AEstack*])
next
case *4*
thus *?case* **by** (*auto intro*: *env-restr-mono2*)

next
  case (*5 ae x as a* Γ *e S*)
  from ‹*ae x* = ⊥›
  have *ABinds* (*delete x* Γ)·*ae* = *ABinds* Γ·*ae* **by** (*rule ABinds-delete-bot*)
  thus *?case* **by** *simp*
next
  case (*6 ae as a* Γ *x S*)
  from *Aexp-Var*[**where** *n* = *a* **and** *x* = *x*]
  have (*Aexp* (*Var x*)·*a*) *x* ≠ ⊥ **by** *auto*
  hence *x* ∈ *edom* (*Aexp* (*Var x*)·*a*) **by** (*simp add*: *edomIff*)
  thus *?case* **by** *simp*
qed

sublocale *CardinalityPrognosisApp prognosis*
proof (*standard*, *goal-cases*)
  case *1*
  thus *?case*
    using *edom-mono*[*OF Aexp-App*] **by** (*auto intro*!: *env-restr-mono2*)
qed

sublocale *CardinalityPrognosisLam prognosis*
proof (*standard*, *goal-cases*)
  case (*1 ae as a* Γ *e y x S*)
  have *edom* (*Aexp e*[*y*::=*x*]·(*pred*·*a*)) ⊆ *insert x* (*edom* (*env-delete y* (*Aexp e*·(*pred*·*a*))))
    by (*auto dest*: *set-mp*[*OF edom-mono*[*OF Aexp-subst*]])
  also have ... ⊆ *insert x* (*edom* (*Aexp* (*Lam* [*y*]. *e*)·*a*))
    using *edom-mono*[*OF Aexp-Lam*] **by** *auto*
  finally show *?case* **by** (*auto intro*!: *env-restr-mono2*)
qed

sublocale *CardinalityPrognosisVar prognosis*
proof (*standard*, *goal-cases*)
  case *prems*: *1*
  thus *?case* **by** (*auto intro*!: *env-restr-mono2 simp add*: *Abinds-reorder1*[*OF prems*(*1*)])
next
  case *prems*: *2*
  thus *?case*
    by (*auto intro*!: *env-restr-mono2 simp add*: *Abinds-reorder1*[*OF prems*(*1*)])
      (*metis Aexp-Var edomIff not-up-less-UU*)
next
  case (*3 e x* Γ *ae as S*)
  have *fup*·(*Aexp e*)·(*ae x*) ⊑ *Aexp e*·*0* **by** (*cases ae x*) (*auto intro*: *monofun-cfun-arg*)
  from *edom-mono*[*OF this*]
  show *?case* **by** (*auto intro*!: *env-restr-mono2 dest*: *set-mp*[*OF edom-mono*[*OF ABinds-delete-below*]])
qed

sublocale *CardinalityPrognosisIfThenElse prognosis*
proof (*standard*, *goal-cases*)
  case (*1 ae a as* Γ *scrut e1 e2 S*)

have *edom* (*Aexp scrut*·*0* ⊔ *Aexp e1*·*a* ⊔ *Aexp e2*·*a*) ⊆ *edom* (*Aexp* (*scrut ? e1* : *e2*)·*a*)
  by (*rule edom-mono*[*OF Aexp-IfThenElse*])
thus *?case*
  by (*auto intro*!: *env-restr-mono2*)
next
case (*2 ae as a* Γ *b e1 e2 S*)
show *?case* **by** (*auto intro*!: *env-restr-mono2*)
qed

sublocale *CardinalityPrognosisLet prognosis cHeap Aheap*
proof (*standard*, *goal-cases*)
case *prems*: (*1* Δ Γ *S ae e a as*)

from *set-mp*[*OF prems*(*3*)] *fresh-distinct*[*OF prems*(*1*)] *fresh-distinct-fv*[*OF prems*(*2*)]
have *ae f*|‘ *domA* Δ = ⊥
  by (*auto dest*: *set-mp*[*OF ups-fv-subset*])
hence [*simp*]: *ABinds* Δ·(*ae* ⊔ *Aheap* Δ *e*·*a*) = *ABinds* Δ·(*Aheap* Δ *e*·*a*) **by** (*simp cong*: *Abinds-env-restr-cong add*: *env-restr-join*)

from *fresh-distinct*[*OF prems*(*1*)]
have *Aheap* Δ *e*·*a f*|‘ *domA* Γ = ⊥ **by** (*auto dest*!: *set-mp*[*OF edom-Aheap*])
hence [*simp*]: *ABinds* Γ·(*ae* ⊔ *Aheap* Δ *e*·*a*) = *ABinds* Γ·*ae* **by** (*simp cong*: *Abinds-env-restr-cong add*: *env-restr-join*)

have *edom* (*ABinds* (Δ @ Γ)·(*Aheap* Δ *e*·*a* ⊔ *ae*)) ∪ *edom* (*Aexp e*·*a*) = *edom* (*ABinds* Δ·(*Aheap* Δ *e*·*a*)) ∪ *edom* (*ABinds* Γ·*ae*) ∪ *edom* (*Aexp e*·*a*)
  by (*simp add*: *Abinds-append-disjoint*[*OF fresh-distinct*[*OF prems*(*1*)]] *Un-commute*)
also have . . . = *edom* (*ABinds* Γ·*ae*) ∪ *edom* (*ABinds* Δ·(*Aheap* Δ *e*·*a*) ⊔ *Aexp e*·*a*)
  by *force*
also have . . . ⊆ *edom* (*ABinds* Γ·*ae*) ∪ *edom* (*Aheap* Δ *e*·*a* ⊔ *Aexp* (*Let* Δ *e*)·*a*)
  using *edom-mono*[*OF Aexp-Let*] **by** *force*
also have . . . = *edom* (*Aheap* Δ *e*·*a*) ∪ *edom* (*ABinds* Γ·*ae*) ∪ *edom* (*Aexp* (*Let* Δ *e*)·*a*)
  by *auto*
finally
have *edom* (*ABinds* (Δ @ Γ)·(*Aheap* Δ *e*·*a* ⊔ *ae*)) ∪ *edom* (*Aexp e*·*a*) ⊆ *edom* (*Aheap* Δ *e*·*a*) ∪ *edom* (*ABinds* Γ·*ae*) ∪ *edom* (*Aexp* (*Let* Δ *e*)·*a*)**.**
hence *edom* (*ABinds* (Δ @ Γ)·(*Aheap* Δ *e*·*a* ⊔ *ae*)) ∪ *edom* (*Aexp e*·*a*) ∪ *edom* (*AEstack as S*) ⊆ *edom* (*Aheap* Δ *e*·*a*) ∪ *edom* (*ABinds* Γ·*ae*) ∪ *edom* (*Aexp* (*Let* Δ *e*)·*a*) ∪ *edom* (*AEstack as S*) **by** *auto*
thus *?case* **by** (*simp add*: *ae2ce-to-env-restr env-restr-join2 Un-assoc*[*symmetric*] *env-restr-mono2*)
qed

sublocale *CardinalityPrognosisEdom prognosis*
by *standard* (*auto dest*: *set-mp*[*OF Aexp-edom*] *set-mp*[*OF ap-fv-subset*] *set-mp*[*OF edom-AnalBinds*] *set-mp*[*OF edom-AEstack*])

sublocale *CardinalityPrognosisSafe prognosis cHeap Aheap Aexp*..

end

end

56 TransformTools.tex

theory *TransformTools*
imports *Nominal−HOLCF Terms Substitution Env*
begin

default-sort *type*

fun *lift-transform* :: $('a::cont\text{-}pt \Rightarrow exp \Rightarrow exp) \Rightarrow ('a_\bot \Rightarrow exp \Rightarrow exp)$
  where *lift-transform t Ibottom e = e*
  | *lift-transform t (Iup a) e = t a e*

lemma *lift-transform-simps*[*simp*]:
  lift-transform t $\bot$ *e = e*
  lift-transform t (*up*$\cdot$*a*) *e = t a e*
  apply (*metis inst-up-pcpo lift-transform.simps*(*1*))
  apply (*simp add*: *up-def cont-Iup*)
  done

lemma *lift-transform-eqvt*[*eqvt*]: $\pi \cdot$ *lift-transform t a e = lift-transform* $(\pi \cdot t)$ $(\pi \cdot a)$ $(\pi \cdot e)$
  by (*cases a*) *simp-all*

lemma *lift-transform-fun-cong*[*fundef-cong*]:
  $(\bigwedge a.$ *t1 a e1 = t2 a e1*$) \Longrightarrow$ *a1 = a2* $\Longrightarrow$ *e1 = e2* $\Longrightarrow$ *lift-transform t1 a1 e1 = lift-transform t2 a2 e2*
  by (*cases* (*t2*,*a2*,*e2*) *rule*: *lift-transform.cases*) *auto*

lemma *subst-lift-transform*:
  assumes $\bigwedge a.$ (*t a e*)[*x* ::= *y*] = *t a* (*e*[*x* ::= *y*])
  shows (*lift-transform t a e*)[*x* ::=*y*] = *lift-transform t a* (*e*[*x* ::= *y*])
  using *assms* **by** (*cases a*) *auto*

definition
  map-transform :: $('a::cont\text{-}pt \Rightarrow exp \Rightarrow exp) \Rightarrow (var \Rightarrow 'a_\bot) \Rightarrow heap \Rightarrow heap$
  where *map-transform t ae = map-ran* (λ *x e . lift-transform t* (*ae x*) *e*)

lemma *map-transform-eqvt*[*eqvt*]: $\pi \cdot$ *map-transform t ae = map-transform* $(\pi \cdot t)$ $(\pi \cdot ae)$
  unfolding *map-transform-def* **by** *simp*

lemma *domA-map-transform*[*simp*]: *domA* (*map-transform t ae* Γ) = *domA* Γ
  unfolding *map-transform-def* **by** *simp*

lemma *length-map-transform*[*simp*]: *length* (*map-transform t ae xs*) = *length xs*

unfolding *map-transform-def map-ran-def* **by** *simp*

lemma *map-transform-delete*:
map-transform t ae (*delete x* Γ) = *delete x* (*map-transform t ae* Γ)
unfolding *map-transform-def* **by** (*simp add*: *map-ran-delete*)

lemma *map-transform-restrA*:
map-transform t ae (*restrictA S* Γ) = *restrictA S* (*map-transform t ae* Γ)
unfolding *map-transform-def* **by** (*auto simp add*: *map-ran-restrictA*)

lemma *delete-map-transform-env-delete*:
delete x (*map-transform t* (*env-delete x ae*) Γ) = *delete x* (*map-transform t ae* Γ)
unfolding *map-transform-def* **by** (*induction* Γ) *auto*

lemma *map-transform-Nil*[*simp*]:
map-transform t ae [] = []
unfolding *map-transform-def* **by** *simp*

lemma *map-transform-Cons*:
map-transform t ae ((*x*,*e*)# Γ) = (*x*, *lift-transform t* (*ae x*) *e*) # (*map-transform t ae* Γ)
unfolding *map-transform-def* **by** *simp*

lemma *map-transform-append*:
map-transform t ae (Δ@Γ) = *map-transform t ae* Δ @ *map-transform t ae* Γ
unfolding *map-transform-def* **by** (*simp add*: *map-ran-append*)

lemma *map-transform-fundef-cong*[*fundef-cong*]:
(⋀*x e a*. (*x*,*e*) ∈ *set m1* ⟹ *t1 a e* = *t2 a e*) ⟹ *ae1* = *ae2* ⟹ *m1* = *m2* ⟹ *map-transform t1 ae1 m1* = *map-transform t2 ae2 m2*
by (*induction m2 arbitrary*: *m1*)
(*fastforce simp add*: *map-transform-Nil map-transform-Cons intro*!: *lift-transform-fun-cong*)+

lemma *map-transform-cong*:
(⋀*x*. *x* ∈ *domA m1* ⟹ *ae x* = *ae′ x*) ⟹ *m1* = *m2* ⟹ *map-transform t ae m1* = *map-transform t ae′ m2*
unfolding *map-transform-def* **by** (*auto intro*!: *map-ran-cong dest*: *domA-from-set*)

lemma *map-of-map-transform*: *map-of* (*map-transform t ae* Γ) *x* = *map-option* (*lift-transform t* (*ae x*)) (*map-of* Γ *x*)
unfolding *map-transform-def* **by** (*simp add*: *map-ran-conv*)

lemma *supp-map-transform-step*:
assumes ⋀ *x e a*. (*x*, *e*) ∈ *set* Γ ⟹ *supp* (*t a e*) ⊆ *supp e*
shows *supp* (*map-transform t ae* Γ) ⊆ *supp* Γ
using *assms*
apply (*induction* Γ)
apply (*auto simp add*: *supp-Nil supp-Cons map-transform-Nil map-transform-Cons supp-Pair pure-supp*)
apply (*case-tac ae a*)

apply (*fastforce*)+
done

lemma *subst-map-transform*:
assumes $\bigwedge$ x' *e a*. (x',e) : *set* Γ ⟹ (*t a e*)[*x* ::= *y*] = *t a* (*e*[*x* ::= *y*])
shows (*map-transform t ae* Γ)[*x* ::*h*=*y*] = *map-transform t ae* (Γ[*x* ::*h*= *y*])
using *assms*
apply (*induction* Γ)
apply (*auto simp add*: *map-transform-Nil map-transform-Cons*)
apply (*subst subst-lift-transform*)
apply *auto*
done

locale *supp-bounded-transform* =
fixes *trans* :: ′*a*::*cont-pt* ⇒ *exp* ⇒ *exp*
assumes *supp-trans*: *supp* (*trans a e*) ⊆ *supp e*
begin
lemma *supp-lift-transform*: *supp* (*lift-transform trans a e*) ⊆ *supp e*
by (*cases* (*trans*, *a*, *e*) *rule*:*lift-transform.cases*) (*auto dest*!: *set-mp*[*OF supp-trans*])

lemma *supp-map-transform*: *supp* (*map-transform trans ae* Γ) ⊆ *supp* Γ
unfolding *map-transform-def*
by (*induction* Γ) (*auto simp add*: *supp-Pair supp-Cons dest*!: *set-mp*[*OF supp-lift-transform*])

lemma *fresh-transform*[*intro*]: *a* ♯ *e* ⟹ *a* ♯ *trans n e*
by (*auto simp add*: *fresh-def*) (*auto dest*!: *set-mp*[*OF supp-trans*])

lemma *fresh-star-transform*[*intro*]: *a* ♯∗ *e* ⟹ *a* ♯∗ *trans n e*
by (*auto simp add*: *fresh-star-def*)

lemma *fresh-map-transform*[*intro*]: *a* ♯ Γ ⟹ *a* ♯ *map-transform trans ae* Γ
unfolding *fresh-def* **using** *supp-map-transform* **by** *auto*

lemma *fresh-star-map-transform*[*intro*]: *a* ♯∗ Γ ⟹ *a* ♯∗ *map-transform trans ae* Γ
by (*auto simp add*: *fresh-star-def*)
end

end

57 AbstractTransform.tex

theory *AbstractTransform*
imports *Terms TransformTools*
begin

locale *AbstractAnalProp* =
fixes *PropApp* :: ′*a* ⇒ ′*a*::*cont-pt*

```
  fixes PropLam :: 'a ⇒ 'a
  fixes AnalLet :: heap ⇒ exp ⇒ 'a ⇒ 'b::cont-pt
  fixes PropLetBody :: 'b ⇒ 'a
  fixes PropLetHeap :: 'b⇒ var ⇒ 'a⊥
  fixes PropIfScrut :: 'a ⇒ 'a
  assumes PropApp-eqvt: π • PropApp ≡ PropApp
  assumes PropLam-eqvt: π • PropLam ≡ PropLam
  assumes AnalLet-eqvt: π • AnalLet ≡ AnalLet
  assumes PropLetBody-eqvt: π • PropLetBody ≡ PropLetBody
  assumes PropLetHeap-eqvt: π • PropLetHeap ≡ PropLetHeap
  assumes PropIfScrut-eqvt: π • PropIfScrut ≡ PropIfScrut

locale AbstractAnalPropSubst = AbstractAnalProp +
  assumes AnalLet-subst:  x ∉ domA Γ ⟹ y ∉ domA Γ ⟹ AnalLet (Γ[x::h=y]) (e[x::=y])
a = AnalLet Γ e a

locale AbstractTransform = AbstractAnalProp +
  constrains AnalLet :: heap ⇒ exp ⇒ 'a::pure-cont-pt ⇒ 'b::cont-pt
  fixes TransVar :: 'a ⇒ var ⇒ exp
  fixes TransApp :: 'a ⇒ exp ⇒ var ⇒ exp
  fixes TransLam :: 'a ⇒ var ⇒ exp ⇒ exp
  fixes TransLet :: 'b ⇒ heap ⇒ exp ⇒ exp
  assumes TransVar-eqvt: π • TransVar = TransVar
  assumes TransApp-eqvt: π • TransApp = TransApp
  assumes TransLam-eqvt: π • TransLam = TransLam
  assumes TransLet-eqvt: π • TransLet = TransLet
  assumes SuppTransLam: supp (TransLam a v e) ⊆ supp e − supp v
  assumes SuppTransLet: supp (TransLet b Γ e) ⊆ supp (Γ, e) − atom ‘ domA Γ
begin
  nominal-function transform where
    transform a (App e x) = TransApp a (transform (PropApp a) e) x
  | transform a (Lam [x]. e) = TransLam a x (transform (PropLam a) e)
  | transform a (Var x) = TransVar a x
  | transform a (Let Γ e) = TransLet (AnalLet Γ e a)
        (map-transform transform (PropLetHeap (AnalLet Γ e a)) Γ)
        (transform (PropLetBody (AnalLet Γ e a)) e)
  | transform a (Bool b) = (Bool b)
 | transform a (scrut ? e1 : e2) = (transform (PropIfScrut a) scrut ? transform a e1 : transform
a e2)
proof goal-cases
  case 1
 note PropApp-eqvt[eqvt-raw] PropLam-eqvt[eqvt-raw] PropLetBody-eqvt[eqvt-raw] PropLetHeap-eqvt[eqvt-raw]
AnalLet-eqvt[eqvt-raw] TransVar-eqvt[eqvt]  TransApp-eqvt[eqvt]  TransLam-eqvt[eqvt] TransLet-eqvt[eqvt]
  show ?case
    unfolding eqvt-def transform-graph-aux-def
    apply rule
    apply perm-simp
    apply (rule refl)
    done
```

next
  case *prems*: (*3 P x*)
  show *?case*
  proof (*cases x*)
    fix *a b*
    assume $x = (a, b)$
    thus *?case*
      using *prems*
      apply (*cases b rule*: *Terms.exp-strong-exhaust*)
      apply *auto*
      done
  qed
next
  case *prems*: (*10 a x e a′ x′ e′*)
  from *prems*(*5*)
  have $a' = a$ **and** *Lam* [*x*]. e = *Lam* [*x′*]. e' **by** *simp-all*
  from *this*(*2*)
  show *?case*
  unfolding ‹$a' = a$›
  proof(*rule eqvt-lam-case*)
    fix π :: *perm*

    have *supp* (*TransLam a x* (*transform-sumC* (*PropLam a*, *e*))) ⊆ *supp* (*Lam* [*x*]. *e*)
      apply (*rule subset-trans*[*OF SuppTransLam*])
       apply (*auto simp add*: *exp-assn.supp supp-Pair supp-at-base pure-supp exp-assn.fsupp dest*!: *set-mp*[*OF supp-eqvt-at*[*OF prems*(*1*)], *rotated*])
      done
    moreover
    assume *supp* π ♯* (*Lam* [*x*]. *e*)
    ultimately
     have *: *supp* π ♯* *TransLam a x* (*transform-sumC* (*PropLam a*, *e*)) **by** (*auto simp add*: *fresh-star-def fresh-def*)

   note *PropApp-eqvt*[*eqvt-raw*] *PropLam-eqvt*[*eqvt-raw*] *PropLetBody-eqvt*[*eqvt-raw*] *PropLetHeap-eqvt*[*eqvt-raw*] *TransVar-eqvt*[*eqvt*] *TransApp-eqvt*[*eqvt*] *TransLam-eqvt*[*eqvt*] *TransLet-eqvt*[*eqvt*]

    have *TransLam a* ($\pi \cdot x$) (*transform-sumC* (*PropLam a*, $\pi \cdot e$))
       = *TransLam a* ($\pi \cdot x$) (*transform-sumC* ($\pi \cdot$ (*PropLam a*, *e*)))
      by *perm-simp rule*
    also have ... = *TransLam a* ($\pi \cdot x$) ($\pi \cdot$ *transform-sumC* (*PropLam a*, *e*))
      unfolding *eqvt-at-apply′*[*OF prems*(*1*)] **..**
    also have ... = $\pi \cdot$ (*TransLam a x* (*transform-sumC* (*PropLam a*, *e*)))
      by *simp*
    also have ... = *TransLam a x* (*transform-sumC* (*PropLam a*, *e*))
      by (*rule perm-supp-eq*[*OF* *])
    finally show *TransLam a* ($\pi \cdot x$) (*transform-sumC* (*PropLam a*, $\pi \cdot e$)) = *TransLam a x* (*transform-sumC* (*PropLam a*, *e*)) **by** *simp*
  qed
next

case *prems*: (*19 a as body a′ as′ body′*)
note *PropApp-eqvt*[*eqvt-raw*] *PropLam-eqvt*[*eqvt-raw*] *PropLetBody-eqvt*[*eqvt-raw*] *AnalLet-eqvt*[*eqvt-raw*] *PropLetHeap-eqvt*[*eqvt-raw*] *TransVar-eqvt*[*eqvt*] *TransApp-eqvt*[*eqvt*] *TransLam-eqvt*[*eqvt*] *TransLet-eqvt*[*eqvt*]

from *supp-eqvt-at*[*OF prems*(*1*)]
have $\bigwedge$ *x e a*. (*x*, *e*) ∈ *set as* ⟹ *supp* (*transform-sumC* (*a*, *e*)) ⊆ *supp e*
by (*auto simp add*: *exp-assn.fsupp supp-Pair pure-supp*)
hence *supp-map*: $\bigwedge$*ae*. *supp* (*map-transform* (*λx0 x1*. *transform-sumC* (*x0*, *x1*)) *ae as*) ⊆ *supp as*
by (*rule supp-map-transform-step*)

from *prems*(*9*)
have *a′* = *a* **and** *Terms.Let as body* = *Terms.Let as′ body′* **by** *simp-all*
from *this*(*2*)
show *?case*
unfolding ⟨*a′* = *a*⟩
proof (*rule eqvt-let-case*)
have *supp* (*TransLet* (*AnalLet as body a*) (*map-transform* (*λx0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet as body a*)) *as*) (*transform-sumC* (*PropLetBody* (*AnalLet as body a*), *body*))) ⊆ *supp* (*Let as body*)
by (*auto simp add*: *Let-supp supp-Pair pure-supp exp-assn.fsupp*
dest!: *set-mp*[*OF supp-eqvt-at*[*OF prems*(*2*)], *rotated*] *set-mp*[*OF SuppTransLet*] *set-mp*[*OF supp-map*])
moreover
fix *π* :: *perm*
assume *supp π* ♯∗ *Terms.Let as body*
ultimately
have ∗: *supp π* ♯∗ *TransLet* (*AnalLet as body a*) (*map-transform* (*λx0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet as body a*)) *as*) (*transform-sumC* (*PropLetBody* (*AnalLet as body a*), *body*))
by (*auto simp add*: *fresh-star-def fresh-def*)

have *TransLet* (*AnalLet* (*π* · *as*) (*π* · *body*) *a*) (*map-transform* (*λx0 x1*. (*π* · *transform-sumC*) (*x0*, *x1*)) (*PropLetHeap* (*AnalLet* (*π* · *as*) (*π* · *body*) *a*)) (*π* · *as*)) ((*π* · *transform-sumC*) (*PropLetBody* (*AnalLet* (*π* · *as*) (*π* · *body*) *a*), *π* · *body*)) =
π · *TransLet* (*AnalLet as body a*) (*map-transform* (*λx0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet as body a*)) *as*) (*transform-sumC* (*PropLetBody* (*AnalLet as body a*), *body*))
by (*simp del*: *Let-eq-iff Pair-eqvt add*: *eqvt-at-apply*[*OF prems*(*2*)])
also have . . . = *TransLet* (*AnalLet as body a*) (*map-transform* (*λx0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet as body a*)) *as*) (*transform-sumC* (*PropLetBody* (*AnalLet as body a*), *body*))
by (*rule perm-supp-eq*[*OF* ∗])
also
have *map-transform* (*λx0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet* (*π* · *as*) (*π* · *body*) *a*)) (*π* · *as*)
= *map-transform* (*λx xa*. (*π* · *transform-sumC*) (*x*, *xa*)) (*PropLetHeap* (*AnalLet* (*π* · *as*) (*π* · *body*) *a*)) (*π* · *as*)
apply (*rule map-transform-fundef-cong*[*OF - refl refl*])

apply (*subst* (*asm*) *set-eqvt*[*symmetric*])
apply (*subst* (*asm*) *mem-permute-set*)
apply (*auto simp add*: *permute-self* *dest*: *eqvt-at-apply*′′[*OF prems*(*1*)[**where** *aa* = (− $\pi \cdot a$) **for** *a*], **where** *p* = π, *symmetric*])
done
moreover
have (π · *transform-sumC*) (*PropLetBody* (*AnalLet* (π · *as*) (π · *body*) *a*), π · *body*) = *transform-sumC* (*PropLetBody* (*AnalLet* (π · *as*) (π · *body*) *a*), π · *body*)
using *eqvt-at-apply*′′[*OF prems*(*2*), **where** *p* = π] **by** *perm-simp*
ultimately
show *TransLet* (*AnalLet* (π · *as*) (π · *body*) *a*) (*map-transform* (λ*x0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet* (π · *as*) (π · *body*) *a*)) (π · *as*)) (*transform-sumC* (*PropLetBody* (*AnalLet* (π · *as*) (π · *body*) *a*), π · *body*)) =
TransLet (*AnalLet as body a*) (*map-transform* (λ*x0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet as body a*)) *as*) (*transform-sumC* (*PropLetBody* (*AnalLet as body a*), *body*)) **by** *metis*
qed
qed *auto*
nominal-termination by *lexicographic-order*

lemma *supp-transform*: *supp* (*transform a e*) ⊆ *supp e*
proof−
note *PropApp-eqvt*[*eqvt-raw*] *PropLam-eqvt*[*eqvt-raw*] *PropLetBody-eqvt*[*eqvt-raw*] *AnalLet-eqvt*[*eqvt-raw*] *PropLetHeap-eqvt*[*eqvt-raw*] *TransVar-eqvt*[*eqvt*] *TransApp-eqvt*[*eqvt*] *TransLam-eqvt*[*eqvt*] *TransLet-eqvt*[*eqvt*]
note *transform.eqvt*[*eqvt*]
show *?thesis*
apply (*rule supp-fun-app-eqvt*)
apply (*rule eqvtI*)
apply *perm-simp*
apply (*rule reflexive*)
done
qed

lemma *fv-transform*: *fv* (*transform a e*) ⊆ *fv e*
unfolding *fv-def* **by** (*auto dest*: *set-mp*[*OF supp-transform*])

end

locale *AbstractTransformSubst* = *AbstractTransform* + *AbstractAnalPropSubst* +
assumes *TransVar-subst*: (*TransVar a v*)[*x* ::= *y*] = (*TransVar a v*[*x* ::*v*= *y*])
assumes *TransApp-subst*: (*TransApp a e v*)[*x* ::= *y*] = (*TransApp a e*[*x* ::= *y*] *v*[*x* ::*v*= *y*])
assumes *TransLam-subst*: *atom v* ♯ (*x*,*y*) ⟹ (*TransLam a v e*)[*x* ::= *y*] = (*TransLam a v*[*x* ::*v*= *y*] *e*[*x* ::= *y*])
assumes *TransLet-subst*: *atom* ‘ *domA* Γ ♯∗ (*x*,*y*) ⟹ (*TransLet b* Γ *e*)[*x* ::= *y*] = (*TransLet b* Γ[*x* ::*h*= *y*] *e*[*x* ::= *y*])
begin
lemma *subst-transform*: (*transform a e*)[*x* ::= *y*] = *transform a e*[*x* ::= *y*]
proof (*nominal-induct e avoiding*: *x y arbitrary*: *a rule*: *exp-strong-induct-set*)
case (*Let* Δ *body x y*)

hence ∗: $x \notin domA\ \Delta\ y \notin domA\ \Delta$ **by** (*auto simp add*: *fresh-star-def fresh-at-base*)
hence *AnalLet* $\Delta[x{::}h{=}y]$ *body*$[x{::=}y]$ a = *AnalLet* Δ *body* a **by** (*rule AnalLet-subst*)
with *Let*
show *?case*
apply (*auto simp add*: *fresh-star-Pair TransLet-subst simp del*: *Let-eq-iff*)
apply (*rule fun-cong*[*OF arg-cong*[**where** f = *TransLet* b **for** b]])
apply (*rule subst-map-transform*)
apply *simp*
done
qed (*simp-all add*: *TransVar-subst TransApp-subst TransLam-subst*)
end

locale *AbstractTransformBound* = *AbstractAnalProp* + *supp-bounded-transform* +
constrains *PropApp* :: $'a \Rightarrow 'a$::*pure-cont-pt*
constrains *PropLetHeap* :: $'b$::*cont-pt* $\Rightarrow$ *var* $\Rightarrow$ $'a_\bot$
constrains *trans* :: $'c$::*cont-pt* $\Rightarrow$ *exp* $\Rightarrow$ *exp*
fixes *PropLetHeapTrans* :: $'b \Rightarrow$ *var* $\Rightarrow$ $'c_\bot$
assumes *PropLetHeapTrans-eqvt*: $\pi \cdot$ *PropLetHeapTrans* = *PropLetHeapTrans*
assumes *TransBound-eqvt*: $\pi \cdot$ *trans* = *trans*
begin
sublocale *AbstractTransform PropApp PropLam AnalLet PropLetBody PropLetHeap PropIfScrut*
(λ *a*. *Var*)
(λ *a*. *App*)
(λ *a*. *Terms.Lam*)
(λ *b* Γ *e* . *Let* (*map-transform trans* (*PropLetHeapTrans b*) Γ) *e*)
proof *goal-cases*
case *1*
note *PropApp-eqvt*[*eqvt-raw*] *PropLam-eqvt*[*eqvt-raw*] *PropLetBody-eqvt*[*eqvt-raw*] *PropLetHeap-eqvt*[*eqvt-raw*] *PropIfScrut-eqvt*[*eqvt-raw*]
AnalLet-eqvt[*eqvt-raw*] *PropLetHeapTrans-eqvt*[*eqvt*] *TransBound-eqvt*[*eqvt*]
show *?case*
apply *standard*
apply ((*perm-simp*, *rule*)+)[*4*]
apply (*auto simp add*: *exp-assn.supp supp-at-base*)[*1*]
apply (*auto simp add*: *Let-supp supp-Pair supp-at-base dest*: *set-mp*[*OF supp-map-transform*])[*1*]
done
qed

lemma *isLam-transform*[*simp*]:
isLam (*transform a e*) $\longleftrightarrow$ *isLam e*
by (*induction e rule*:*isLam.induct*) *auto*

lemma *isVal-transform*[*simp*]:
isVal (*transform a e*) $\longleftrightarrow$ *isVal e*
by (*induction e rule*:*isLam.induct*) *auto*

end

locale *AbstractTransformBoundSubst* = *AbstractAnalPropSubst* + *AbstractTransformBound* +

assumes *TransBound-subst*: (*trans a e*)[*x*::=*y*] = *trans a e*[*x*::=*y*]
begin
sublocale *AbstractTransformSubst PropApp PropLam AnalLet PropLetBody PropLetHeap PropIfScrut*
(λ *a*. *Var*)
(λ *a*. *App*)
(λ *a*. *Terms.Lam*)
(λ *b* Γ *e* . *Let* (*map-transform trans* (*PropLetHeapTrans b*) Γ) *e*)
proof *goal-cases*
case *1*
note *PropApp-eqvt*[*eqvt-raw*] *PropLam-eqvt*[*eqvt-raw*] *PropLetBody-eqvt*[*eqvt-raw*] *PropLetHeap-eqvt*[*eqvt-raw*] *PropIfScrut-eqvt*[*eqvt-raw*]
TransBound-eqvt[*eqvt*]
show *?case*
apply *standard*
apply *simp-all*[*3*]
apply (*simp del*: *Let-eq-iff*)
apply (*rule arg-cong*[**where** *f* = λ *x*. *Let x y* **for** *y*])
apply (*rule subst-map-transform*)
apply (*simp add*: *TransBound-subst*)
done
qed
end

end

58 EtaExpansion.tex

theory *EtaExpansion*
imports *Terms Substitution*
begin

definition *fresh-var* :: *exp* $\Rightarrow$ *var* **where**
fresh-var e = (*SOME v*. *v* $\notin$ *fv e*)

lemma *fresh-var-not-free*:
fresh-var e $\notin$ *fv e*
proof−
obtain *v* :: *var* **where** *atom v* $\sharp$ *e* **by** (*rule obtain-fresh*)
hence *v* $\notin$ *fv e* **by** (*metis fv-not-fresh*)
thus *?thesis* **unfolding** *fresh-var-def* **by** (*rule someI*)
qed

lemma *fresh-var-fresh*[*simp*]:
atom (*fresh-var e*) $\sharp$ *e*

by (*metis fresh-var-not-free fv-not-fresh*)

lemma *fresh-var-subst*[*simp*]:
e[*fresh-var e*::=*x*] = *e*
by (*metis fresh-var-fresh subst-fresh-noop*)

fun *eta-expand* :: *nat* ⇒ *exp* ⇒ *exp* **where**
eta-expand 0 e = *e*
| *eta-expand* (*Suc n*) *e* = (*Lam* [*fresh-var e*]. *eta-expand n* (*App e* (*fresh-var e*)))

lemma *eta-expand-eqvt*[*eqvt*]:
$\pi \cdot$ (*eta-expand n e*) = *eta-expand* ($\pi \cdot n$) ($\pi \cdot e$)
apply (*induction n arbitrary*: *e* π)
apply (*auto simp add*: *fresh-Pair permute-pure*)
apply (*metis fresh-at-base-permI fresh-at-base-permute-iff fresh-var-fresh subst-fresh-noop subst-swap-same*)
done

lemma *fresh-eta-expand*[*simp*]: *a* ♯ *eta-expand n e* ⟷ *a* ♯ *e*
apply (*induction n arbitrary*: *e*)
apply (*simp add*: *fresh-Pair*)
apply (*clarsimp simp add*: *fresh-Pair fresh-at-base*)
by (*metis fresh-var-fresh*)

lemma *subst-eta-expand*: (*eta-expand n e*)[*x* ::= *y*] = *eta-expand n* (*e*[*x* ::= *y*])
proof (*induction n arbitrary*: *e*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
obtain *z* :: *var* **where** *atom z* ♯ (*e*, *fresh-var e*, *x*, *y*) **by** (*rule obtain-fresh*)

have (*eta-expand* (*Suc n*) *e*)[*x*::=*y*] = (*Lam* [*fresh-var e*]. *eta-expand n* (*App e* (*fresh-var e*)))[*x*::=*y*] **by** *simp*
also have ... = (*Lam* [*z*]. *eta-expand n* (*App e z*))[*x*::=*y*]
apply (*subst change-Lam-Variable*[**where** *y'* = *z*])
using ‹*atom z* ♯ -›
by (*auto simp add*: *fresh-Pair eta-expand-eqvt pure-fresh permute-pure flip-fresh-fresh intro*!: *eqvt-fresh-cong2*[**where** *f* = *eta-expand*, *OF eta-expand-eqvt*])
also have ... = *Lam* [*z*]. (*eta-expand n* (*App e z*))[*x*::=*y*]
using ‹*atom z* ♯ -› **by** *simp*
also have ... = *Lam* [*z*]. *eta-expand n* (*App e z*)[*x*::=*y*] **unfolding** *Suc.IH*..
also have ... = *Lam* [*z*]. *eta-expand n* (*App e*[*x*::=*y*] *z*)
using ‹*atom z* ♯ -› **by** *simp*
also have ... = *Lam* [*fresh-var* (*e*[*x*::=*y*])]. *eta-expand n* (*App e*[*x*::=*y*] (*fresh-var* (*e*[*x*::=*y*])))
apply (*subst change-Lam-Variable*[**where** *y'* = *fresh-var* (*e*[*x*::=*y*])])
using ‹*atom z* ♯ -›
by (*auto simp add*: *fresh-Pair eqvt-fresh-cong2*[**where** *f* = *eta-expand*, *OF eta-expand-eqvt*] *pure-fresh eta-expand-eqvt flip-fresh-fresh subst-pres-fresh simp del*: *exp-assn.eq-iff*)
also have ... = *eta-expand* (*Suc n*) *e*[*x*::=*y*] **by** *simp*
finally show *?case*.

qed

lemma *isLam-eta-expand*:
  isLam e $\Longrightarrow$ *isLam* (*eta-expand n e*) **and** $n > 0 \Longrightarrow$ *isLam* (*eta-expand n e*)
by (*induction n*) *auto*

lemma *isVal-eta-expand*:
  isVal e $\Longrightarrow$ *isVal* (*eta-expand n e*) **and** $n > 0 \Longrightarrow$ *isVal* (*eta-expand n e*)
by (*induction n*) *auto*

end

59 EtaExpansionSafe.tex

theory *EtaExpansionSafe*
imports *EtaExpansion Sestoft*
begin

theorem *eta-expansion-safe*:
  assumes *set T* $\subseteq$ *range Arg*
  shows (Γ, *eta-expand* (*length T*) *e*, *T@S*) $\Rightarrow^*$ (Γ, *e*, *T@S*)
using *assms*
proof(*induction T arbitrary*: *e*)
  case *Nil* **show** *?case* **by** *simp*
next
  case (*Cons se T*)
  from *Cons*(*2*) **obtain** *x* **where** *se* = *Arg x* **by** *auto*

  from *Cons* **have** *prem*: *set T* $\subseteq$ *range Arg* **by** *simp*

  have (Γ, *eta-expand* (*Suc* (*length T*)) *e*, *Arg x* # *T* @ *S*) = (Γ, *Lam* [*fresh-var e*]. *eta-expand* (*length T*) (*App e* (*fresh-var e*)), *Arg x* # *T* @ *S*) **by** *simp*
  also have ... $\Rightarrow$ (Γ, (*eta-expand* (*length T*) (*App e* (*fresh-var e*)))[*fresh-var e* ::= *x*], *T* @ *S*) **by** (*rule* app_2)
  also have ... = (Γ, (*eta-expand* (*length T*) (*App e x*)), *T* @ *S*) **unfolding** *subst-eta-expand* **by** *simp*
  also have ... $\Rightarrow^*$ (Γ, *App e x*, *T* @ *S*) **by** (*rule Cons.IH*[*OF prem*])
  also have ... $\Rightarrow$ (Γ, *e*, *Arg x* # *T* @ *S*) **by** (*rule* app_1)
  finally show *?case* **using** ‹*se* = -› **by** *simp*
qed

fun *arg-prefix* :: *stack* $\Rightarrow$ *nat* **where**
  arg-prefix [] = *0*
| *arg-prefix* (*Arg x* # *S*) = *Suc* (*arg-prefix S*)
| *arg-prefix* (*Alts e1 e2* # *S*) = *0*
| *arg-prefix* (*Upd x* # *S*) = *0*
| *arg-prefix* (*Dummy x* # *S*) = *0*

theorem *eta-expansion-safe'*:
  assumes $n \leq$ *arg-prefix S*
  shows $(\Gamma,$ *eta-expand n e*, $S) \Rightarrow^* (\Gamma, e, S)$
proof−
  from *assms*
  have *set* (*take n S*) $\subseteq$ *range Arg* **and** *length* (*take n S*) = *n*
    apply (*induction S arbitrary*: *n rule*: *arg-prefix.induct*)
    apply *auto*
    apply (*case-tac n*, *auto*)+
    done
  hence *S* = *take n S* @ *drop n S* **by** (*metis append-take-drop-id*)
  with *eta-expansion-safe*[*OF* ‹- ⊆ -›] ‹*length* - = -›
  show *?thesis* **by** *metis*
qed

end

60 ArityStack.tex

theory *ArityStack*
imports *Arity SestoftConf*
begin

fun *Astack* :: *stack* ⇒ *Arity*
  where *Astack* [] = *0*
    | *Astack* (*Arg x* # *S*) = *inc*·(*Astack S*)
    | *Astack* (*Alts e1 e2* # *S*) = *0*
    | *Astack* (*Upd x* # *S*) = *0*
    | *Astack* (*Dummy x* # *S*) = *0*

lemma *Astack-restr-stack-below*:
  Astack (*restr-stack V S*) ⊑ *Astack S*
  by (*induction V S rule*: *restr-stack.induct*) *auto*

lemma *Astack-map-Dummy*[*simp*]:
  Astack (*map Dummy l*) = *0*
by (*induction l*) *auto*

lemma *Astack-append-map-Dummy*[*simp*]:
  Astack S' = *0* ⟹ *Astack* (*S* @ *S'*) = *Astack S*
by (*induction S rule*: *Astack.induct*) *auto*

end

61 ArityEtaExpansion.tex

theory *ArityEtaExpansion*
imports *EtaExpansion Arity−Nominal TransformTools*
begin

lift-definition *Aeta-expand* :: *Arity* ⇒ *exp* ⇒ *exp* **is** *eta-expand***.**

lemma *Aeta-expand-eqvt*[*eqvt*]: $\pi \cdot$ *Aeta-expand a e* = *Aeta-expand* ($\pi \cdot a$) ($\pi \cdot e$)
  apply (*cases a*)
  apply *simp*
  apply *transfer*
  apply *simp*
  done

lemma *Aeta-expand-0*[*simp*]: *Aeta-expand 0 e* = *e*
  by *transfer simp*

lemma *Aeta-expand-inc*[*simp*]: *Aeta-expand* (*inc*·*n*) *e* = (*Lam* [*fresh-var e*]. *Aeta-expand n* (*App e* (*fresh-var e*)))
  apply (*simp add*: *inc-def*)
  by *transfer simp*

lemma *subst-Aeta-expand*:
  (*Aeta-expand n e*)[*x*::=*y*] = *Aeta-expand n e*[*x*::=*y*]
  by *transfer* (*rule subst-eta-expand*)

lemma *isLam-Aeta-expand*: *isLam e* ⟹ *isLam* (*Aeta-expand a e*)
  by *transfer* (*rule isLam-eta-expand*)

lemma *isVal-Aeta-expand*: *isVal e* ⟹ *isVal* (*Aeta-expand a e*)
  by *transfer* (*rule isVal-eta-expand*)

lemma *Aeta-expand-fresh*[*simp*]: *a* ♯ *Aeta-expand n e* = *a* ♯ *e* **by** *transfer simp*
lemma *Aeta-expand-fresh-star*[*simp*]: *a* ♯* *Aeta-expand n e* = *a* ♯* *e* **by** (*auto simp add*: *fresh-star-def*)

interpretation *supp-bounded-transform Aeta-expand*
  apply *standard*
  using *Aeta-expand-fresh*
  apply (*auto simp add*: *fresh-def*)
  done

end

62 ArityEtaExpansionSafe.tex

theory *ArityEtaExpansionSafe*

imports *EtaExpansionSafe ArityStack ArityEtaExpansion*
begin

lemma *Aeta-expand-safe*:
  assumes *Astack S* $\sqsubseteq$ *a*
  shows (Γ, *Aeta-expand a e*, *S*) $\Rightarrow^*$ (Γ, *e*, *S*)
proof−
  have *arg-prefix S* = *Rep-Arity* (*Astack S*)
   by (*induction S arbitrary*: *a rule*: *arg-prefix.induct*) (*auto simp add*: *Arity.zero-Arity.rep-eq*[*symmetric*])
  also
 from *assms*
  have *Rep-Arity a* $\leq$ *Rep-Arity* (*Astack S*) **by** (*metis below-Arity.rep-eq*)
  finally
  show *?thesis*
    by *transfer* (*rule eta-expansion-safe'*)
qed

end

63 ArityTransform.tex

theory *ArityTransform*
imports *ArityAnalysisSig AbstractTransform ArityEtaExpansionSafe*
begin

context *ArityAnalysisHeapEqvt*
begin
sublocale *AbstractTransformBound*
  λ *a* . *inc*·*a*
  λ *a* . *pred*·*a*
  λ Δ *e a* . (*a*, *Aheap* Δ *e*·*a*)
  fst
  snd
  λ -. *0*
  Aeta-expand
  snd
apply *standard*
apply (((*rule eq-reflection*)?, *perm-simp*, *rule*)+)
done

abbreviation *transform-syn* ($\mathcal{T}_{\text{-}}$) **where** $\mathcal{T}_a \equiv$ *transform a*

lemma *transform-simps*:
  $\mathcal{T}_a$ (*App e x*) = *App* ($\mathcal{T}_{inc \cdot a}$ *e*) *x*
  $\mathcal{T}_a$ (*Lam* [*x*]. *e*) = *Lam* [*x*]. $\mathcal{T}_{pred \cdot a}$ *e*
  $\mathcal{T}_a$ (*Var x*) = *Var x*

$\mathcal{T}_a$ (*Let* Γ *e*) = *Let* (*map-transform Aeta-expand* (*Aheap* Γ *e*·*a*) (*map-transform* (λ*a*. $\mathcal{T}_a$) (*Aheap* Γ *e*·*a*) Γ)) ($\mathcal{T}_a$ *e*)
$\mathcal{T}_a$ (*Bool b*) = *Bool b*
$\mathcal{T}_a$ (*scrut ? e1 : e2*) = ($\mathcal{T}_0$ *scrut ?* $\mathcal{T}_a$ *e1* : $\mathcal{T}_a$ *e2*)
by *simp-all*
end

end

64 AirtyConsistent.tex

theory *ArityConsistent*
imports *ArityAnalysisSpec ArityStack ArityAnalysisStack*
begin

context *ArityAnalysisLetSafe*
begin

type-synonym *astate* = (*AEnv* × *Arity* × *Arity list*)

inductive *stack-consistent* :: *Arity list* ⇒ *stack* ⇒ *bool*
where
stack-consistent [] []
| *Astack S* ⊑ *a* ⟹ *stack-consistent as S* ⟹ *stack-consistent* (*a*#*as*) (*Alts e1 e2* # *S*)
| *stack-consistent as S* ⟹ *stack-consistent as* (*Upd x* # *S*)
| *stack-consistent as S* ⟹ *stack-consistent as* (*Arg x* # *S*)
inductive-simps *stack-consistent-foo*[*simp*]:
stack-consistent [] [] *stack-consistent* (*a*#*as*) (*Alts e1 e2* # *S*) *stack-consistent as* (*Upd x* # *S*) *stack-consistent as* (*Arg x* # *S*)
inductive-cases [*elim*!]: *stack-consistent as* (*Alts e1 e2* # *S*)

inductive *a-consistent* :: *astate* ⇒ *conf* ⇒ *bool* **where**
a-consistentI:
edom ae ⊆ *domA* Γ ∪ *upds S*
⟹ *Astack S* ⊑ *a*
⟹ (*ABinds* Γ·*ae* ⊔ *Aexp e*·*a* ⊔ *AEstack as S*) *f*|' (*domA* Γ ∪ *upds S*) ⊑ *ae*
⟹ *stack-consistent as S*
⟹ *a-consistent* (*ae*, *a*, *as*) (Γ, *e*, *S*)
inductive-cases *a-consistentE*: *a-consistent* (*ae*, *a*, *as*) (Γ, *e*, *S*)

lemma *a-consistent-restrictA*:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *e*, *S*)
assumes *edom ae* ⊆ *V*
shows *a-consistent* (*ae*, *a*, *as*) (*restrictA V* Γ, *e*, *S*)
proof−
have *domA* Γ ∩ *V* ∪ *upds S* ⊆ *domA* Γ ∪ *upds S* **by** *auto*
note ∗ = *below-trans*[*OF env-restr-mono2*[*OF this*]]

show *a-consistent* (*ae*, *a*, *as*) (*restrictA V* Γ, *e*, *S*)
using *assms*
by (*auto simp add*: *a-consistent.simps env-restr-join join-below-iff ABinds-restrict-edom*
intro: ∗ *below-trans*[*OF env-restr-mono*[*OF ABinds-restrict-below*]])
qed

lemma *a-consistent-edom-subsetD*:
a-consistent (*ae*, *a*, *as*) (Γ, *e*, *S*) ⟹ *edom ae* ⊆ *domA* Γ ∪ *upds S*
by (*rule a-consistentE*)

lemma *a-consistent-stackD*:
a-consistent (*ae*, *a*, *as*) (Γ, *e*, *S*) ⟹ *Astack S* ⊑ *a*
by (*rule a-consistentE*)

lemma *a-consistent-app*$_1$:
a-consistent (*ae*, *a*, *as*) (Γ, *App e x*, *S*) ⟹ *a-consistent* (*ae*, *inc*·*a*, *as*) (Γ, *e*, *Arg x* # *S*)
by (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*
dest!: *below-trans*[*OF env-restr-mono*[*OF Aexp-App*]]
elim: *below-trans*)

lemma *a-consistent-app*$_2$:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, (*Lam* [*y*]. *e*), *Arg x* # *S*)
shows *a-consistent* (*ae*, (*pred*·*a*), *as*) (Γ, *e*[*y*::=*x*], *S*)
proof−
have *Aexp* (*e*[*y*::=*x*])·(*pred*·*a*) *f*|‘ (*domA* Γ ∪ *upds S*) ⊑ (*env-delete y* ((*Aexp e*)·(*pred*·*a*)) ⊔ *esing x*·(*up*·*0*)) *f*|‘ (*domA* Γ ∪ *upds S*) **by** (*rule env-restr-mono*[*OF Aexp-subst*])
also have . . . = *env-delete y* ((*Aexp e*)·(*pred*·*a*)) *f*|‘ (*domA* Γ ∪ *upds S*) ⊔ *esing x*·(*up*·*0*) *f*|‘ (*domA* Γ ∪ *upds S*) **by** (*simp add*: *env-restr-join*)
also have *env-delete y* ((*Aexp e*)·(*pred*·*a*)) ⊑ *Aexp* (*Lam* [*y*]. *e*)·*a* **by** (*rule Aexp-Lam*)
also have . . . *f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *ae* **using** *assms* **by** (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
also have *esing x*·(*up*·*0*) *f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *ae* **using** *assms*
by (*cases x*∈*edom ae*) (*auto simp add*: *env-restr-join join-below-iff a-consistent.simps*)
also have *ae* ⊔ *ae* = *ae* **by** *simp*
finally
have *Aexp* (*e*[*y*::=*x*])·(*pred*·*a*) *f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *ae* **by** *this simp-all*
thus *?thesis* **using** *assms*
by (*auto elim*: *below-trans edom-mono simp add*: *join-below-iff env-restr-join a-consistent.simps*)
qed

lemma *a-consistent-thunk-0*:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *Var x*, *S*)
assumes *map-of* Γ *x* = *Some e*
assumes *ae x* = *up*·*0*
shows *a-consistent* (*ae*, *0*, *as*) (*delete x* Γ, *e*, *Upd x* # *S*)
proof−
from *assms*(*2*)

have [*simp*]: $x \in domA\ \Gamma$ **by** (*metis domI dom-map-of-conv-domA*)

from *assms(3)*
have [*simp*]: $x \in edom\ ae$ **by** (*auto simp add*: *edom-def*)

have $x \in domA\ \Gamma$ **by** (*metis assms(2) domI dom-map-of-conv-domA*)
hence [*simp*]: *insert* x ($domA\ \Gamma - \{x\} \cup upds\ S$) $= (domA\ \Gamma \cup upds\ S)$
by *auto*

from *Abinds-reorder1*[*OF* ‹*map-of* $\Gamma\ x = Some\ e$›] ‹$ae\ x = up\cdot 0$›
have *ABinds* (*delete* $x\ \Gamma$)$\cdot ae \sqcup Aexp\ e\cdot 0 = ABinds\ \Gamma\cdot ae$ **by** (*auto intro*: *join-comm*)
moreover have $(\ldots \sqcup AEstack\ as\ S)\ f|`\ (domA\ \Gamma \cup upds\ S) \sqsubseteq ae$
using *assms(1)* **by** (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
ultimately have $((ABinds\ (delete\ x\ \Gamma))\cdot ae \sqcup Aexp\ e\cdot 0 \sqcup AEstack\ as\ S)\ f|`\ (domA\ \Gamma \cup upds\ S) \sqsubseteq ae$ **by** *simp*
then
show *?thesis*
using ‹$ae\ x = up\cdot 0$› *assms(1)*
by (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
qed

lemma *a-consistent-thunk-once*:
assumes *a-consistent* (ae, a, as) (Γ, *Var* x, S)
assumes *map-of* $\Gamma\ x = Some\ e$
assumes [*simp*]: $ae\ x = up\cdot u$
assumes *heap-upds-ok* (Γ, S)
shows *a-consistent* (*env-delete* x ae, u, as) (*delete* $x\ \Gamma$, e, S)
proof−
from *assms(2)*
have [*simp*]: $x \in domA\ \Gamma$ **by** (*metis domI dom-map-of-conv-domA*)

from *assms(1)* **have** *Aexp* (*Var* x)$\cdot a\ f|`\ (domA\ \Gamma \cup upds\ S) \sqsubseteq ae$ **by** (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
from *fun-belowD*[**where** $x = x$, *OF this*]
have (*Aexp* (*Var* x)$\cdot a$) $x \sqsubseteq up\cdot u$ **by** *simp*
from *below-trans*[*OF Aexp-Var this*]
have $a \sqsubseteq u$ **by** *simp*

from ‹*heap-upds-ok* (Γ, S)›
have $x \notin upds\ S$ **by** (*auto simp add*: *a-consistent.simps elim!*: *heap-upds-okE*)
hence [*simp*]: $(- \{x\} \cap (domA\ \Gamma \cup upds\ S)) = (domA\ \Gamma - \{x\} \cup upds\ S)$ **by** *auto*

have *Astack* $S \sqsubseteq u$ **using** *assms(1)* ‹$a \sqsubseteq u$›
by (*auto elim*: *below-trans simp add*: *a-consistent.simps*)

from *Abinds-reorder1*[*OF* ‹*map-of* $\Gamma\ x = Some\ e$›] ‹$ae\ x = up\cdot u$›
have *ABinds* (*delete* $x\ \Gamma$)$\cdot ae \sqcup Aexp\ e\cdot u = ABinds\ \Gamma\cdot ae$ **by** (*auto intro*: *join-comm*)
moreover
have $(\ldots \sqcup AEstack\ as\ S)\ f|`\ (domA\ \Gamma \cup upds\ S) \sqsubseteq ae$

using *assms(1)* **by** (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
ultimately
have ((*ABinds* (*delete x* Γ))·*ae* ⊔ *Aexp e*·*u* ⊔ *AEstack as S*) *f*|' (*domA* Γ ∪ *upds S*) ⊑ *ae* **by** *simp*
hence ((*ABinds* (*delete x* Γ))·(*env-delete x ae*) ⊔ *Aexp e*·*u* ⊔ *AEstack as S*) *f*|' (*domA* Γ ∪ *upds S*) ⊑ *ae*
by (*auto simp add*: *join-below-iff env-restr-join elim*: *below-trans*[*OF env-restr-mono*[*OF monofun-cfun-arg*[*OF env-delete-below-arg*]]])
hence *env-delete x* (((*ABinds* (*delete x* Γ))·(*env-delete x ae*) ⊔ *Aexp e*·*u* ⊔ *AEstack as S*) *f*|' (*domA* Γ ∪ *upds S*)) ⊑ *env-delete x ae*
by (*rule env-delete-mono*)
hence (((*ABinds* (*delete x* Γ))·(*env-delete x ae*) ⊔ *Aexp e*·*u* ⊔ *AEstack as S*) *f*|' (*domA* (*delete x* Γ) ∪ *upds S*)) ⊑ *env-delete x ae*
by (*simp add*: *env-delete-restr*)
then
show *?thesis*
using ‹*ae x* = *up*·*u*› ‹*Astack S* ⊑ *u*› *assms(1)*
by (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps elim* : *below-trans*)
qed

lemma *a-consistent-lamvar*:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *Var x*, *S*)
assumes *map-of* Γ *x* = *Some e*
assumes [*simp*]: *ae x* = *up*·*u*
shows *a-consistent* (*ae*, *u*, *as*) ((*x*,*e*)# *delete x* Γ, *e*, *S*)
proof−
have [*simp*]: *x* ∈ *domA* Γ **by** (*metis assms(2) domI dom-map-of-conv-domA*)
have [*simp*]: *insert x* (*domA* Γ ∪ *upds S*) = (*domA* Γ ∪ *upds S*)
by *auto*

from *assms(1)* **have** *Aexp* (*Var x*)·*a f*|' (*domA* Γ ∪ *upds S*) ⊑ *ae* **by** (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
from *fun-belowD*[**where** *x* = *x*, *OF this*]
have (*Aexp* (*Var x*)·*a*) *x* ⊑ *up*·*u* **by** *simp*
from *below-trans*[*OF Aexp-Var this*]
have *a* ⊑ *u* **by** *simp*

have *Astack S* ⊑ *u* **using** *assms(1)* ‹*a* ⊑ *u*›
by (*auto elim*: *below-trans simp add*: *a-consistent.simps*)

from *Abinds-reorder1*[*OF* ‹*map-of* Γ *x* = *Some e*›] ‹*ae x* = *up*·*u*›
have *ABinds* ((*x*,*e*) # *delete x* Γ)·*ae* ⊔ *Aexp e*·*u* = *ABinds* Γ·*ae* **by** (*auto intro*: *join-comm*)
moreover
have (. . . ⊔ *AEstack as S*) *f*|' (*domA* Γ ∪ *upds S*) ⊑ *ae*
using *assms(1)* **by** (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
ultimately
have ((*ABinds* ((*x*,*e*) # *delete x* Γ))·*ae* ⊔ *Aexp e*·*u* ⊔ *AEstack as S*) *f*|' (*domA* Γ ∪ *upds S*) ⊑ *ae* **by** *simp*
then

show *?thesis*
  using ‹$ae\ x = up{\cdot}u$› ‹$Astack\ S \sqsubseteq u$› *assms*(*1*)
  by (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
qed

lemma
  assumes *a-consistent* (ae, a, as) (Γ, e, $Upd\ x\ \#\ S$)
  shows *a-consistent-var*$_2$: *a-consistent* (ae, a, as) ((x, e) # Γ, e, S)
  and *a-consistent-UpdD*: $ae\ x = up{\cdot}0 a = 0$
  using *assms*
  by (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*
      elim:*below-trans*[*OF env-restr-mono*[*OF ABinds-delete-below*]])

lemma *a-consistent-let*:
  assumes *a-consistent* (ae, a, as) (Γ, $Let\ \Delta\ e$, S)
  assumes *atom* ‘ $domA\ \Delta\ \sharp{*}\ \Gamma$
  assumes *atom* ‘ $domA\ \Delta\ \sharp{*}\ S$
  assumes $edom\ ae \cap domA\ \Delta = \{\}$
  shows *a-consistent* ($Aheap\ \Delta\ e{\cdot}a \sqcup ae$, a, as) (Δ @ Γ, e, S)
proof−

First some boring stuff about scope:

  have [*simp*]: $\bigwedge S.\ S \subseteq domA\ \Delta \Longrightarrow ae\ f|`\ S = \bot$ **using** *assms*(*4*) **by** *auto*
  have [*simp*]: $ABinds\ \Delta{\cdot}(Aheap\ \Delta\ e{\cdot}a \sqcup ae) = ABinds\ \Delta{\cdot}(Aheap\ \Delta\ e{\cdot}a)$
  by (*rule Abinds-env-restr-cong*) (*simp add*: *env-restr-join*)

  have [*simp*]: $Aheap\ \Delta\ e{\cdot}a\ f|`\ domA\ \Gamma = \bot$
  using *fresh-distinct*[*OF assms*(*2*)]
  by (*auto intro*: *env-restr-empty dest*!: *set-mp*[*OF edom-Aheap*])

  have [*simp*]: $ABinds\ \Gamma{\cdot}(Aheap\ \Delta\ e{\cdot}a \sqcup ae) = ABinds\ \Gamma{\cdot}ae$
  by (*rule Abinds-env-restr-cong*) (*simp add*: *env-restr-join*)

  have [*simp*]: $ABinds\ \Gamma{\cdot}ae\ f|`\ (domA\ \Delta \cup domA\ \Gamma \cup upds\ S) = ABinds\ \Gamma{\cdot}ae\ f|`\ (domA\ \Gamma \cup upds\ S)$
  using *fresh-distinct-fv*[*OF assms*(*2*)]
  by (*auto intro*: *env-restr-cong dest*!: *set-mp*[*OF edom-AnalBinds*])

  have [*simp*]: $AEstack\ as\ S\ f|`\ (domA\ \Delta \cup domA\ \Gamma \cup upds\ S) = AEstack\ as\ S\ f|`\ (domA\ \Gamma \cup upds\ S)$
  using *fresh-distinct-fv*[*OF assms*(*3*)]
  by (*auto intro*: *env-restr-cong dest*!: *set-mp*[*OF edom-AEstack*])

  have [*simp*]: $Aexp\ (Let\ \Delta\ e){\cdot}a\ f|`\ (domA\ \Delta \cup domA\ \Gamma \cup upds\ S) = Aexp\ (Terms.Let\ \Delta\ e){\cdot}a\ f|`\ (domA\ \Gamma \cup upds\ S)$
  by (*rule env-restr-cong*) (*auto dest*!: *set-mp*[*OF Aexp-edom*])

  have [*simp*]: $Aheap\ \Delta\ e{\cdot}a\ f|`\ (domA\ \Delta \cup domA\ \Gamma \cup upds\ S) = Aheap\ \Delta\ e{\cdot}a$
  by (*rule env-restr-useless*) (*auto dest*!: *set-mp*[*OF edom-Aheap*])

have $((ABinds\ \Gamma)\cdot ae \sqcup AEstack\ as\ S)\ f|\text{`}\ (domA\ \Gamma \cup upds\ S) \sqsubseteq ae$ **using** *assms*(*1*) **by** (*auto simp add*: *a-consistent.simps join-below-iff env-restr-join*)
moreover
have $Aexp\ (Let\ \Delta\ e)\cdot a\ f|\text{`}\ (domA\ \Gamma \cup upds\ S) \sqsubseteq ae$ **using** *assms*(*1*) **by** (*auto simp add*: *a-consistent.simps join-below-iff env-restr-join*)
moreover
have $ABinds\ \Delta\cdot(Aheap\ \Delta\ e\cdot a) \sqcup Aexp\ e\cdot a \sqsubseteq Aheap\ \Delta\ e\cdot a \sqcup Aexp\ (Let\ \Delta\ e)\cdot a$ **by** (*rule Aexp-Let*)
ultimately
have $(ABinds\ (\Delta\ @\ \Gamma)\cdot(Aheap\ \Delta\ e\cdot a \sqcup ae) \sqcup Aexp\ e\cdot a \sqcup AEstack\ as\ S)\ f|\text{`}\ (domA\ (\Delta @\Gamma) \cup upds\ S) \sqsubseteq Aheap\ \Delta\ e\cdot a \sqcup ae$
by (*auto 4 4 simp add*: *env-restr-join Abinds-append-disjoint*[*OF fresh-distinct*[*OF assms*(*2*)]] *join-below-iff*
simp del: *join-comm*
elim: *below-trans below-trans*[*OF env-restr-mono*])
moreover
note *fresh-distinct*[*OF assms*(*2*)]
moreover
from *fresh-distinct-fv*[*OF assms*(*3*)]
have $domA\ \Delta \cap upds\ S = \{\}$ **by** (*auto dest*!: *set-mp*[*OF ups-fv-subset*])
ultimately
show *?thesis* **using** *assms*(*1*)
by (*auto simp add*: *a-consistent.simps dest*!: *set-mp*[*OF edom-Aheap*] *intro*: *heap-upds-ok-append*)
qed

lemma *a-consistent-if*$_1$:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *scrut ? e1* : *e2*, *S*)
shows *a-consistent* (*ae*, *0*, *a#as*) (Γ, *scrut*, *Alts e1 e2 # S*)
proof−
from *assms*
have $Aexp\ (scrut\ ?\ e1 : e2)\cdot a\ f|\text{`}\ (domA\ \Gamma \cup upds\ S) \sqsubseteq ae$ **by** (*auto simp add*: *a-consistent.simps env-restr-join join-below-iff*)
hence $(Aexp\ scrut\cdot 0 \sqcup Aexp\ e1\cdot a \sqcup Aexp\ e2\cdot a)\ f|\text{`}\ (domA\ \Gamma \cup upds\ S) \sqsubseteq ae$
by (*rule below-trans*[*OF env-restr-mono*[*OF Aexp-IfThenElse*]])
thus *?thesis*
using *assms*
by (*auto simp add*: *a-consistent.simps join-below-iff env-restr-join*)
qed

lemma *a-consistent-if*$_2$:
assumes *a-consistent* (*ae*, *a*, *a'#as'*) (Γ, *Bool b*, *Alts e1 e2 # S*)
shows *a-consistent* (*ae*, *a'*, *as'*) (Γ, *if b then e1 else e2*, *S*)
using *assms* **by** (*auto simp add*: *a-consistent.simps join-below-iff env-restr-join*)

lemma *a-consistent-alts-on-stack*:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *Bool b*, *Alts e1 e2 # S*)
obtains *a' as'* **where** $as = a'\ \#\ as'\ a = 0$
using *assms* **by** (*auto simp add*: *a-consistent.simps*)

lemma *closed-a-consistent*:
fv e = ({}::*var set*) $\Longrightarrow$ *a-consistent* (⊥, *0*, []) ([], *e*, [])
by (*auto simp add*: *edom-empty-iff-bot a-consistent.simps dest*!: *set-mp*[*OF Aexp-edom*])

end

end

65 ArityTransformSafe.tex

theory *ArityTransformSafe*
imports *ArityTransform ArityConsistent ArityAnalysisSpec ArityEtaExpansionSafe Abstract-Transform ConstOn*
begin

locale *CardinalityArityTransformation* = *ArityAnalysisLetSafeNoCard*
begin
sublocale *AbstractTransformBoundSubst*
λ *a* . *inc*·*a*
λ *a* . *pred*·*a*
λ Δ *e a* . (*a*, *Aheap* Δ *e*·*a*)
fst
snd
λ -. *0*
Aeta-expand
snd
apply *standard*
apply (*simp add*: *Aheap-subst*)
apply (*rule subst-Aeta-expand*)
done

abbreviation *ccTransform* **where** *ccTransform* ≡ *transform*

lemma *supp-transform*: *supp* (*transform a e*) ⊆ *supp e*
by (*induction rule*: *transform.induct*)
(*auto simp add*: *exp-assn.supp Let-supp dest*!: *set-mp*[*OF supp-map-transform*] *set-mp*[*OF supp-map-transform-step*])
interpretation *supp-bounded-transform transform*
by *standard* (*auto simp add*: *fresh-def supp-transform*)

fun *transform-alts* :: *Arity list* ⇒ *stack* ⇒ *stack*
where
transform-alts - [] = []
| *transform-alts* (*a*#*as*) (*Alts e1 e2* # *S*) = (*Alts* (*ccTransform a e1*) (*ccTransform a e2*)) # *transform-alts as S*
| *transform-alts as* (*x* # *S*) = *x* # *transform-alts as S*

lemma *transform-alts-Nil*[*simp*]: *transform-alts* [] $S = S$
by (*induction* S) *auto*

lemma *Astack-transform-alts*[*simp*]:
Astack (*transform-alts as* S) = *Astack* S
by (*induction rule*: *transform-alts.induct*) *auto*

lemma *fresh-star-transform-alts*[*intro*]: $a \sharp* S \Longrightarrow a \sharp*$ *transform-alts as* S
by (*induction as* S *rule*: *transform-alts.induct*) (*auto simp add*: *fresh-star-Cons*)

fun *a-transform* :: *astate* ⇒ *conf* ⇒ *conf*
where *a-transform* (*ae*, *a*, *as*) (Γ, *e*, S) =
(*map-transform Aeta-expand ae* (*map-transform ccTransform ae* Γ),
ccTransform a e,
transform-alts as S)

fun *restr-conf* :: *var set* ⇒ *conf* ⇒ *conf*
where *restr-conf* V (Γ, *e*, S) = (*restrictA* V Γ, *e*, *restr-stack* V S)

inductive *consistent* :: *astate* ⇒ *conf* ⇒ *bool* **where**
consistentI[*intro*!]:
a-consistent (*ae*, *a*, *as*) (Γ, *e*, S)
$\Longrightarrow (\bigwedge x.\ x \in$ *thunks* $\Gamma \Longrightarrow$ *ae* x = *up*·*0*)
$\Longrightarrow$ *consistent* (*ae*, *a*, *as*) (Γ, *e*, S)
inductive-cases *consistentE*[*elim*!]: *consistent* (*ae*, *a*, *as*) (Γ, *e*, S)

lemma *closed-consistent*:
assumes *fv e* = ({}::*var set*)
shows *consistent* ($\bot$, *0*, []) ([], *e*, [])
by (*auto simp add*: *edom-empty-iff-bot closed-a-consistent*[*OF assms*])

lemma *arity-tranform-safe*:
fixes c c'
assumes $c \Rightarrow^* c'$ **and** ¬ *boring-step* c' **and** *heap-upds-ok-conf* c **and** *consistent* (*ae*,*a*,*as*) c
shows ∃ *ae*′ *a*′ *as*′. *consistent* (*ae*′,*a*′,*as*′) c' ∧ *a-transform* (*ae*,*a*,*as*) $c \Rightarrow^*$ *a-transform* (*ae*′,*a*′,*as*′) c'
using *assms*(*1*,*2*) *heap-upds-ok-invariant assms*(*3*−)
proof(*induction* c c' *arbitrary*: *ae a as rule*:*step-invariant-induction*)
case (app_1 Γ *e* *x* S)
from app_1 **have** *consistent* (*ae*, *inc*·*a*, *as*) (Γ, *e*, *Arg x* # S)
by (*auto intro*: *a-consistent-app*$_1$)
moreover
have *a-transform* (*ae*, *a*, *as*) (Γ, *App e x*, S) ⇒ *a-transform* (*ae*, *inc*·*a*, *as*) (Γ, *e*, *Arg x* # S)
by *simp rule*
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next

case (*app*$_2$ Γ *y e x S*)
have *consistent* (*ae*, *pred*·*a*, *as*) (Γ, *e*[*y*::=*x*], *S*) **using** *app*$_2$
by (*auto 4 3 intro*: *a-consistent-app*$_2$)
moreover
have *a-transform* (*ae*, *a*, *as*) (Γ, *Lam* [*y*]. *e*, *Arg x* # *S*) ⇒ *a-transform* (*ae*, *pred* · *a*, *as*) (Γ, *e*[*y*::=*x*], *S*) **by** (*simp add*: *subst-transform*[*symmetric*]) *rule*
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (*thunk* Γ *x e S*)
hence *x* ∈ *thunks* Γ **by** *auto*
hence [*simp*]: *x* ∈ *domA* Γ **by** (*rule set-mp*[*OF thunks-domA*])

from ⟨*heap-upds-ok-conf* (Γ, *Var x*, *S*)⟩
have *x* ∉ *upds S* **by** (*auto dest*!: *heap-upds-okE*)

have *x* ∈ *edom ae* **using** *thunk* **by** *auto*
have *ae x* = *up*·*0* **using** *thunk* ⟨*x* ∈ *thunks* Γ⟩ **by** (*auto*)

have *a-consistent* (*ae*, *0*, *as*) (*delete x* Γ, *e*, *Upd x* # *S*) **using** *thunk* ⟨*ae x* = *up*·*0*⟩
by (*auto intro*!: *a-consistent-thunk-0 simp del*: *restr-delete*)
hence *consistent* (*ae*, *0*, *as*) (*delete x* Γ, *e*, *Upd x* # *S*) **using** *thunk* ⟨*ae x* = *up*·*0*⟩
by (*auto simp add*: *restr-delete-twist*)
moreover

from ⟨*map-of* Γ *x* = *Some e*⟩ ⟨*ae x* = *up*·*0*⟩
have *map-of* (*map-transform Aeta-expand ae* (*map-transform ccTransform ae* Γ)) *x* = *Some* (*transform 0 e*)
by (*simp add*: *map-of-map-transform*)
with ⟨¬ *isVal e*⟩
have *a-transform* (*ae*, *a*, *as*) (Γ, *Var x*, *S*) ⇒ *a-transform* (*ae*, *0*, *as*) (*delete x* Γ, *e*, *Upd x* # *S*)
by (*auto simp add*: *map-transform-delete restr-delete-twist intro*!: *step.intros simp del*: *restr-delete*)
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (*lamvar* Γ *x e S*)
from *lamvar*(*1*) **have** [*simp*]: *x* ∈ *domA* Γ **by** (*metis domI dom-map-of-conv-domA*)

have *up*·*a* ⊑ (*Aexp* (*Var x*)·*a f*|‘ (*domA* Γ ∪ *upds S*)) *x*
by (*simp*) (*rule Aexp-Var*)
also from *lamvar* **have** *Aexp* (*Var x*)·*a f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *ae* **by** (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
finally
obtain *u* **where** *ae x* = *up*·*u* **by** (*cases ae x*) (*auto simp add*: *edom-def*)
hence *x* ∈ *edom ae* **by** (*auto simp add*: *edomIff*)

have *a-consistent* (*ae*, *u*, *as*) ((*x*,*e*) # *delete x* Γ, *e*, *S*) **using** *lamvar* ⟨*ae x* = *up*·*u*⟩

by (*auto intro*!: *a-consistent-lamvar simp del*: *restr-delete*)
hence *consistent* (*ae*, *u*, *as*) ((*x*, *e*) # *delete x* Γ, *e*, *S*)
using *lamvar* **by** (*auto simp add*: *thunks-Cons restr-delete-twist elim*: *below-trans*)
moreover

from ‹*a-consistent* - -›
have *Astack* (*transform-alts as S*) ⊑ *u* **by** (*auto elim*: *a-consistent-stackD*)

{
from ‹*isVal e*›
have *isVal* (*transform u e*) **by** *simp*
hence *isVal* (*Aeta-expand u* (*transform u e*)) **by** (*rule isVal-Aeta-expand*)
moreover
from ‹*map-of* Γ *x* = *Some e*› ‹*ae x* = *up* · *u*› ‹*isVal* (*transform u e*)›
have *map-of* (*map-transform Aeta-expand ae* (*map-transform transform ae* Γ)) *x* = *Some* (*Aeta-expand u* (*transform u e*))
by (*simp add*: *map-of-map-transform*)
ultimately
have *a-transform* (*ae*, *a*, *as*) (Γ, *Var x*, *S*) ⇒*
((*x*, *Aeta-expand u* (*transform u e*)) # *delete x* (*map-transform Aeta-expand ae* (*map-transform transform ae* Γ)), *Aeta-expand u* (*transform u e*), *transform-alts as S*)
by (*auto intro*: *lambda-var simp del*: *restr-delete*)
also have ... = ((*map-transform Aeta-expand ae* (*map-transform transform ae* ((*x*,*e*) # *delete x* Γ))), *Aeta-expand u* (*transform u e*), *transform-alts as S*)
using ‹*ae x* = *up* · *u*› ‹*isVal* (*transform u e*)›
by (*simp add*: *map-transform-Cons map-transform-delete del*: *restr-delete*)
also(*subst*[*rotated*]) **have** ... ⇒* *a-transform* (*ae*, *u*, *as*) ((*x*, *e*) # *delete x* Γ, *e*, *S*)
by (*simp add*: *restr-delete-twist*) (*rule Aeta-expand-safe*[*OF* ‹*Astack* - ⊑ *u*›])
finally(*rtranclp-trans*)
have *a-transform* (*ae*, *a*, *as*) (Γ, *Var x*, *S*) ⇒* *a-transform* (*ae*, *u*, *as*) ((*x*, *e*) # *delete x* Γ, *e*, *S*).
}
ultimately show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (var_2 Γ *x e S*)
from var_2
have *a-consistent* (*ae*, *a*, *as*) (Γ, *e*, *Upd x* # *S*) **by** *auto*
from *a-consistent-UpdD*[*OF this*]
have *ae x* = *up*·*0* **and** *a* = *0*.

have *a-consistent* (*ae*, *a*, *as*) ((*x*, *e*) # Γ, *e*, *S*)
using var_2 **by** (*auto intro*!: *a-consistent-var*$_2$)
hence *consistent* (*ae*, *0*, *as*) ((*x*, *e*) # Γ, *e*, *S*)
using var_2 ‹*a* = *0*›
by (*auto simp add*: *thunks-Cons elim*: *below-trans*)
moreover
have *a-transform* (*ae*, *a*, *as*) (Γ, *e*, *Upd x* # *S*) ⇒ *a-transform* (*ae*, *0*, *as*) ((*x*, *e*) # Γ, *e*, *S*)
using ‹*ae x* = *up*·*0*› ‹*a* = *0*› var_2

by (*auto intro*!: *step.intros simp add*: *map-transform-Cons*)
ultimately show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (let_1 Δ Γ *e* *S*)
let *?ae* = *Aheap* Δ *e*·*a*

have *domA* $\Delta \cap$ *upds S* = {} **using** *fresh-distinct-fv*[*OF* let_1(*2*)] **by** (*auto dest*: *set-mp*[*OF* *ups-fv-subset*])
hence ∗: $\bigwedge$ *x*. *x* $\in$ *upds S* $\Longrightarrow$ *x* $\notin$ *edom ?ae* **by** (*auto simp add*: *dest*!: *set-mp*[*OF* *edom-Aheap*])
have *restr-stack-simp2*: *restr-stack* (*edom* (*?ae* $\sqcup$ *ae*)) *S* = *restr-stack* (*edom ae*) *S*
by (*auto intro*: *restr-stack-cong dest*!: ∗)

have *edom ae* $\subseteq$ *domA* $\Gamma \cup$ *upds S* **using** let_1 **by** (*auto dest*!: *a-consistent-edom-subsetD*)
from *set-mp*[*OF this*] *fresh-distinct*[*OF* let_1(*1*)] *fresh-distinct-fv*[*OF* let_1(*2*)]
have *edom ae* $\cap$ *domA* Δ = {} **by** (*auto dest*: *set-mp*[*OF ups-fv-subset*])

{
{ fix *x* *e*′
assume *x* $\in$ *thunks* Γ
with let_1
have (*?ae* $\sqcup$ *ae*) *x* = *up*·*0* **by** *auto*
}
moreover
{ fix *x* *e*′
assume *x* $\in$ *thunks* Δ
hence (*?ae* $\sqcup$ *ae*) *x* = *up*·*0* **by** (*auto simp add*: *Aheap-heap3*)
}
moreover

have *a-consistent* (*ae*, *a*, *as*) (Γ, *Let* Δ *e*, *S*)
using let_1 **by** *auto*
hence *a-consistent* (*?ae* $\sqcup$ *ae*, *a*, *as*) (Δ @ Γ, *e*, *S*)
using let_1(*1*,*2*) ‹*edom ae* $\cap$ *domA* Δ = {}›
by (*auto intro*!: *a-consistent-let simp del*: *join-comm*)
ultimately
have *consistent* (*?ae* $\sqcup$ *ae*, *a*, *as*) (Δ @ Γ, *e*, *S*)
by *auto*
}
moreover
{
have $\bigwedge$ *x*. *x* $\in$ *domA* $\Gamma \Longrightarrow$ *x* $\notin$ *edom ?ae*
using *fresh-distinct*[*OF* let_1(*1*)]
by (*auto dest*!: *set-mp*[*OF edom-Aheap*])
hence *map-transform Aeta-expand* (*?ae* $\sqcup$ *ae*) (*map-transform transform* (*?ae* $\sqcup$ *ae*) Γ)
= *map-transform Aeta-expand ae* (*map-transform transform ae* Γ)
by (*auto intro*!: *map-transform-cong restrictA-cong simp add*: *edomIff*)
moreover

from ‹*edom ae* $\subseteq$ *domA* $\Gamma \cup$ *upds S*›
have $\bigwedge x.\ x \in domA\ \Delta \Longrightarrow x \notin edom\ ae$
using *fresh-distinct*[*OF let*$_1$(*1*)] *fresh-distinct-fv*[*OF let*$_1$(*2*)]
by (*auto dest*!: *set-mp*[*OF ups-fv-subset*])
hence *map-transform Aeta-expand* (*?ae* $\sqcup$ *ae*) (*map-transform transform* (*?ae* $\sqcup$ *ae*) Δ)
= *map-transform Aeta-expand ?ae* (*map-transform transform ?ae* Δ)
by (*auto intro*!: *map-transform-cong restrictA-cong simp add*: *edomIff*)
ultimately

have *a-transform* (*ae*, *a*, *as*) (Γ, *Let* Δ *e*, *S*) $\Rightarrow$ *a-transform* (*?ae* $\sqcup$ *ae*, *a*, *as*) (Δ @ Γ, *e*, *S*)
using *restr-stack-simp2 let*$_1$(*1*,*2*)
apply (*auto simp add*: *map-transform-append restrictA-append restr-stack-simp2*[*simplified*] *map-transform-restrA*)
apply (*rule step.let*$_1$)
apply (*auto dest*: *set-mp*[*OF edom-Aheap*])
done
}
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (*if*$_1$ Γ *scrut e1 e2 S*)
have *consistent* (*ae*, *0*, *a#as*) (Γ, *scrut*, *Alts e1 e2* # *S*)
using *if*$_1$ **by** (*auto dest*: *a-consistent-if*$_1$)
moreover
have *a-transform* (*ae*, *a*, *as*) (Γ, *scrut ? e1* : *e2*, *S*) $\Rightarrow$ *a-transform* (*ae*, *0*, *a#as*) (Γ, *scrut*, *Alts e1 e2* # *S*)
by (*auto intro*: *step.intros*)
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (*if*$_2$ Γ *b e1 e2 S*)
hence *a-consistent* (*ae*, *a*, *as*) (Γ, *Bool b*, *Alts e1 e2* # *S*) **by** *auto*
then obtain *a' as'* **where** [*simp*]: *as* = *a'* # *as' a* = *0*
by (*rule a-consistent-alts-on-stack*)

have *consistent* (*ae*, *a'*, *as'*) (Γ, *if b then e1 else e2*, *S*)
using *if*$_2$ **by** (*auto dest*!: *a-consistent-if*$_2$)
moreover
have *a-transform* (*ae*, *a*, *as*) (Γ, *Bool b*, *Alts e1 e2* # *S*) $\Rightarrow$ *a-transform* (*ae*, *a'*, *as'*) (Γ, *if b then e1 else e2*, *S*)
by (*auto intro*: *step.if*$_2$[**where** *b* = *True*, *simplified*] *step.if*$_2$[**where** *b* = *False*, *simplified*])
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case *refl* **thus** *?case* **by** *auto*
next
case (*trans c c' c''*)

from *trans(3)[OF trans(5)]*
obtain *ae′ a′ as′* **where** *consistent (ae′, a′, as′) c′* **and** $*$: *a-transform (ae, a, as) c* $\Rightarrow^*$ *a-transform (ae′, a′, as′) c′* **by** *blast*
from *trans(4)[OF this(1)]*
obtain *ae′′ a′′ as′′* **where** *consistent (ae′′, a′′, as′′) c′′* **and** $**$: *a-transform (ae′, a′, as′) c′* $\Rightarrow^*$ *a-transform (ae′′, a′′, as′′) c′′* **by** *blast*
from *this(1) rtranclp-trans[OF * **]*
show *?case* **by** *blast*
qed
end

end

66 Set-Cpo.tex

theory *Set−Cpo*
imports *~~/src/HOL/HOLCF/HOLCF*
begin

default-sort *type*

instantiation *set* :: (*type*) *below*
begin
definition *below-set* **where** *op* $\sqsubseteq$ = *op* $\subseteq$
instance..
end

instance *set* :: (*type*) *po*
by *standard (auto simp add: below-set-def)*

lemma *is-lub-set*:
$S <<| \bigcup S$
by(*auto simp add: is-lub-def below-set-def is-ub-def*)

lemma *lub-set*: *lub* $S = \bigcup S$
by (*metis is-lub-set lub-eqI*)

instance *set* :: (*type*) *cpo*
by *standard (rule exI, rule is-lub-set)*

lemma *minimal-set*: $\{\} \sqsubseteq S$
unfolding *below-set-def* **by** *simp*

instance *set* :: (*type*) *pcpo*
by *standard (rule+, rule minimal-set)*

lemma *set-contI*:
assumes $\bigwedge Y.\ chain\ Y \Longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = \bigcup\ (f\ `\ range\ Y)$

shows *cont f*
proof(*rule contI*)
fix Y :: *nat* $\Rightarrow$ *'a*
assume *chain Y*
hence $f\ (\bigsqcup\ i.\ Y\ i) = \bigcup\ (f\ `\ range\ Y)$ **by** (*rule assms*)
also have $\ldots\ = \bigcup\ (range\ (\lambda i.\ f\ (Y\ i)))$ **by** *simp*
finally
show $range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup\ i.\ Y\ i)$ **using** *is-lub-set* **by** *metis*
qed

lemma *set-set-contI*:
assumes $\bigwedge\ S.\ f\ (\bigcup S) = \bigcup\ (f\ `\ S)$
shows *cont f*
by (*metis set-set-contI assms is-lub-set lub-eqI*)

lemma *adm-subseteq*[*simp*]:
assumes *cont f*
shows $adm\ (\lambda a.\ f\ a \subseteq S)$
by (*rule admI*)(*auto simp add*: *cont2contlubE*[*OF assms*] *lub-set*)

lemma *adm-Ball*[*simp*]: $adm\ (\lambda S.\ \forall x{\in}S.\ P\ x)$
by (*auto intro*!: *admI simp add*: *lub-set*)

lemma *finite-subset-chain*:
fixes Y :: *nat* $\Rightarrow$ *'a set*
assumes *chain Y*
assumes $S \subseteq$ *UNION UNIV Y*
assumes *finite S*
shows $\exists\, i.\ S \subseteq Y\ i$
proof−
from *assms*(*2*)
have $\forall\, x \in S.\ \exists\ i.\ x \in Y\ i$ **by** *auto*
then obtain f **where** f: $\forall\ x{\in}\ S.\ x \in\ Y\ (f\ x)$ **by** *metis*

def $i \equiv Max\ (f\ `\ S)$
from ‹*finite S*›
have *finite* $(f\ `\ S)$ **by** *simp*
hence $\forall\ x{\in}S.\ f\ x \le i$ **unfolding** *i-def* **by** *auto*
with *chain-mono*[*OF* ‹*chain Y*›]
have $\forall\ x{\in}S.\ Y\ (f\ x) \subseteq\ Y\ i$ **by** (*auto simp add*: *below-set-def*)
with f
have $S \subseteq\ Y\ i$ **by** *auto*
thus *?thesis*..
qed

lemma *diff-cont*[*THEN cont-compose*, *simp*, *cont2cont*]:
fixes S' :: *'a set*
shows $cont\ (\lambda S.\ S - S')$
by (*rule set-set-contI*) *simp*

end

67 Env-Set-Cpo.tex

theory *Env−Set−Cpo*
imports *Env Set−Cpo*
begin

lemma *cont-edom*[*THEN cont-compose*, *simp*, *cont2cont*]:
  cont (λ *f* . *edom f*)
  apply (*rule set-contI*)
  apply (*auto simp add*: *edom-def*)
  apply (*metis ch2ch-fun lub-eq-bottom-iff lub-fun*)
  apply (*metis ch2ch-fun lub-eq-bottom-iff lub-fun*)
  done

end

68 CoCallGraph.tex

theory *CoCallGraph*
imports *Vars HOLCF−Join−Classes HOLCF−Utils Set−Cpo*
begin

default-sort *type*

typedef *CoCalls* = {*G* :: (*var* × *var*) *set*. *sym G*}
  morphisms *Rep-CoCall Abs-CoCall*
  by (*auto intro*: *exI*[**where** *x* = {}] *symI*)

setup-lifting *type-definition-CoCalls*

instantiation *CoCalls* :: *po*
begin
lift-definition *below-CoCalls* :: *CoCalls* ⇒ *CoCalls* ⇒ *bool* **is** *op* ⊆**.**
instance
  apply *standard*
  apply ((*transfer*, *auto*)+)
  done
end

lift-definition *coCallsLub* :: *CoCalls set* ⇒ *CoCalls* **is** λ *S*. $\bigcup$ *S*
  by (*auto intro*: *symI elim*: *symE*)

lemma *coCallsLub-is-lub*: *S* <<| *coCallsLub S*

proof (*rule is-lubI*)
  show $S <| coCallsLub\ S$
    by (*rule is-ubI*, *transfer*, *auto*)
next
  fix u
  assume $S <| u$
  hence $\forall x \in S.\ x \sqsubseteq u$ **by** (*auto dest*: *is-ubD*)
  thus $coCallsLub\ S \sqsubseteq u$ **by** *transfer auto*
qed

instance *CoCalls* :: *cpo*
proof
  fix $S :: nat \Rightarrow CoCalls$
  show $\exists x.\ range\ S <<| x$ **using** *coCallsLub-is-lub*..
qed

lemma *ccLubTransfer*[*transfer-rule*]: (*rel-set pcr-CoCalls* ===> *pcr-CoCalls*) *Union lub*
proof−
  have *lub* = *coCallsLub*
    apply (*rule*)
    apply (*rule lub-eqI*)
    apply (*rule coCallsLub-is-lub*)
    done
  with *coCallsLub.transfer*
  show *?thesis* **by** *metis*
qed

lift-definition *is-cc-lub* :: $CoCalls\ set \Rightarrow CoCalls \Rightarrow bool$ **is** $(\lambda\ S\ x\ .\ x = Union\ S)$**.**

lemma *ccis-lubTransfer*[*transfer-rule*]: (*rel-set pcr-CoCalls* ===> *pcr-CoCalls* ===> *op* =) $(\lambda\ S\ x\ .\ x = Union\ S)$ *op* <<|
proof−
  have $\bigwedge\ x\ xa\ .\ is\text{-}cc\text{-}lub\ x\ xa \longleftrightarrow xa = coCallsLub\ x$ **by** *transfer auto*
  hence *is-cc-lub* = *op* <<|
  apply −
  apply (*rule*, *rule*)
  by (*metis coCallsLub-is-lub is-lub-unique*)
  thus *?thesis* **using** *is-cc-lub.transfer* **by** *simp*
qed

lift-definition *coCallsJoin* :: $CoCalls \Rightarrow CoCalls\ \Rightarrow CoCalls$ **is** *op* $\cup$
    by (*rule sym-Un*)

lemma *ccJoinTransfer*[*transfer-rule*]: (*pcr-CoCalls* ===> *pcr-CoCalls* ===> *pcr-CoCalls*) *op* $\cup$ *op* $\sqcup$
proof−
  have *op* $\sqcup$ = *coCallsJoin*
    apply (*rule*)
    apply *rule*

apply (*rule lub-is-join*)
unfolding *is-lub-def is-ub-def*
apply *transfer*
apply *auto*
done
with *coCallsJoin.transfer*
show *?thesis* **by** *metis*
qed

lift-definition *ccEmpty* :: *CoCalls* **is** {} **by** (*auto intro*: *symI*)

lemma *ccEmpty-below*[*simp*]: *ccEmpty* ⊑ *G*
by *transfer auto*

instance *CoCalls* :: *pcpo*
proof
have ∀ *y* . *ccEmpty* ⊑ *y* **by** *transfer simp*
thus ∃ *x*. ∀ *y*. (*x*::*CoCalls*) ⊑ *y*..
qed

lemma *ccBotTransfer*[*transfer-rule*]: *pcr-CoCalls* {} ⊥
proof−
have ⋀*x*. *ccEmpty* ⊑ *x* **by** *transfer simp*
hence *ccEmpty* = ⊥ **by** (*rule bottomI*)
thus *?thesis* **using** *ccEmpty.transfer* **by** *simp*
qed

lemma *cc-lub-below-iff*:
fixes *G* :: *CoCalls*
shows *lub X* ⊑ *G* ⟷ (∀ *G'*∈*X*. *G'* ⊑ *G*)
by *transfer auto*

lift-definition *ccField* :: *CoCalls* ⇒ *var set* **is** *Field*.

lemma *ccField-nil*[*simp*]: *ccField* ⊥ = {}
by *transfer auto*

lift-definition
inCC :: *var* ⇒ *var* ⇒ *CoCalls* ⇒ *bool* (-−−-∈- [*1000*, *1000*, *900*] *900*)
is λ *x y s*. (*x*,*y*) ∈ *s*.

abbreviation
notInCC :: *var* ⇒ *var* ⇒ *CoCalls* ⇒ *bool* (-−−-∉- [*1000*, *1000*, *900*] *900*)
where *x*−−*y*∉*S* ≡ ¬ *x*−−*y*∈*S*

lemma *notInCC-bot*[*simp*]: *x*−−*y*∈⊥ ⟷ *False*
by *transfer auto*

lemma *below-CoCallsI*:

$(\bigwedge x\ y.\ x{-}{-}y{\in}G \Longrightarrow x{-}{-}y{\in}G') \Longrightarrow G \sqsubseteq G'$
by *transfer auto*

lemma *CoCalls-eqI*:
$(\bigwedge x\ y.\ x{-}{-}y{\in}G \longleftrightarrow x{-}{-}y{\in}G') \Longrightarrow G = G'$
by *transfer auto*

lemma *in-join*[*simp*]:
$x{-}{-}y \in (G \sqcup G') \longleftrightarrow x{-}{-}y{\in}G \vee x{-}{-}y{\in}G'$
by *transfer auto*

lemma *in-lub*[*simp*]: $x{-}{-}y{\in}(lub\ S) \longleftrightarrow (\exists\ G{\in}S.\ x{-}{-}y{\in}G)$
by *transfer auto*

lemma *in-CoCallsLubI*:
$x{-}{-}y{\in}G \Longrightarrow G \in S \Longrightarrow x{-}{-}y{\in}lub\ S$
by *transfer auto*

lemma *adm-not-in*[*simp*]:
assumes *cont t*
shows *adm* $(\lambda a.\ x{-}{-}y{\notin}t\ a)$
by (*rule admI*) (*auto simp add*: *cont2contlubE*[*OF assms*])

lift-definition *cc-delete* :: *var* ⇒ *CoCalls* ⇒ *CoCalls*
is $\lambda\ z.\ Set.filter\ (\lambda\ (x,y)\ .\ x \neq z \wedge y \neq z)$
by (*auto intro*!: *symI elim*: *symE*)

lemma *ccField-cc-delete*: *ccField* (*cc-delete x S*) $\subseteq$ *ccField S* $- \{x\}$
by *transfer* (*auto simp add*: *Field-def*)

lift-definition *ccProd* :: *var set* ⇒ *var set* ⇒ *CoCalls* (**infixr** *G*× *90*)
is $\lambda\ S1\ S2.\ S1 \times S2 \cup S2 \times S1$
by (*auto intro*!: *symI elim*: *symE*)

lemma *ccProd-empty*[*simp*]: $\{\}\ G{\times}\ S = \bot$ **by** *transfer auto*

lemma *ccProd-empty'*[*simp*]: $S\ G{\times}\ \{\} = \bot$ **by** *transfer auto*

lemma *ccProd-union2*[*simp*]: $S\ G{\times}\ (S' \cup S'') = S\ G{\times}\ S' \sqcup S\ G{\times}\ S''$
by *transfer auto*

lemma *ccProd-Union2*[*simp*]: $S\ G{\times} \bigcup S' = (\bigsqcup\ X{\in}S'.\ ccProd\ S\ X)$
by *transfer auto*

lemma *ccProd-Union2'*[*simp*]: $S\ G{\times}\ (\bigcup X{\in}S'.\ f\ X) = (\bigsqcup\ X{\in}S'.\ ccProd\ S\ (f\ X))$
by *transfer auto*

lemma *in-ccProd*[*simp*]: $x{-}{-}y{\in}(S\ G{\times}\ S') = (x \in S \wedge y \in S' \vee x \in S' \wedge y \in S)$
by *transfer auto*

lemma *ccProd-union1*[*simp*]: $(S' \cup S'')\ G\times\ S = S'\ G\times\ S \sqcup S''\ G\times\ S$
  by *transfer auto*

lemma *ccProd-insert2*: $S\ G\times\ \mathit{insert}\ x\ S' = S\ G\times\ \{x\} \sqcup S\ G\times\ S'$
  by *transfer auto*

lemma *ccProd-insert1*: $\mathit{insert}\ x\ S'\ G\times\ S = \{x\}\ G\times\ S \sqcup S'\ G\times\ S$
  by *transfer auto*

lemma *ccProd-mono1*: $S' \subseteq S'' \Longrightarrow S'\ G\times\ S \sqsubseteq S''\ G\times\ S$
  by *transfer auto*

lemma *ccProd-mono2*: $S' \subseteq S'' \Longrightarrow S\ G\times\ S' \sqsubseteq S\ G\times\ S''$
  by *transfer auto*

lemma *ccProd-mono*: $S \subseteq S' \Longrightarrow T \subseteq T' \Longrightarrow S\ G\times\ T \sqsubseteq S'\ G\times\ T'$
  by *transfer auto*

lemma *ccProd-comm*: $S\ G\times\ S' = S'\ G\times\ S$ **by** *transfer auto*

lemma *ccProd-belowI*:
  $(\bigwedge x\ y.\ x \in S \Longrightarrow y \in S' \Longrightarrow x{-}{-}y{\in}G) \Longrightarrow S\ G\times\ S' \sqsubseteq G$
  by *transfer* (*auto elim*: *symE*)

lift-definition *cc-restr* :: *var set* $\Rightarrow$ *CoCalls* $\Rightarrow$ *CoCalls*
  is $\lambda\ S.\ \mathit{Set.filter}\ (\lambda\ (x,y)\ .\ x \in S \wedge y \in S)$
  by (*auto intro*!: *symI elim*: *symE*)

abbreviation *cc-restr-sym* (**infixl** $G|^{\backprime}$ *110*) **where** $G\ G|^{\backprime}\ S \equiv \mathit{cc\text{-}restr}\ S\ G$

lemma *elem-cc-restr*[*simp*]: $x{-}{-}y{\in}(G\ G|^{\backprime}\ S) = (x{-}{-}y{\in}G \wedge x \in S \wedge y \in S)$
  by *transfer auto*

lemma *ccField-cc-restr*: $\mathit{ccField}\ (G\ G|^{\backprime}\ S) \subseteq \mathit{ccField}\ G \cap S$
  by *transfer* (*auto simp add*: *Field-def*)

lemma *cc-restr-empty*: $\mathit{ccField}\ G \subseteq -\ S \Longrightarrow G\ G|^{\backprime}\ S = \bot$
  apply *transfer*
  apply (*auto simp add*: *Field-def*)
  apply (*drule DomainI*)
  apply (*drule* (*1*) *set-mp*)
  apply *simp*
  done

lemma *cc-restr-empty-set*[*simp*]: *cc-restr* $\{\}$ $G = \bot$
  by *transfer auto*

lemma *cc-restr-noop*[*simp*]: *ccField* $G \subseteq S \Longrightarrow$ *cc-restr* S $G = G$
by *transfer* (*force simp add*: *Field-def dest*: *DomainI RangeI elim*: *set-mp*)

lemma *cc-restr-bot*[*simp*]: *cc-restr* S $\bot = \bot$
by *simp*

lemma *ccRestr-ccDelete*[*simp*]: *cc-restr* $(-\{x\})$ $G =$ *cc-delete* x G
by *transfer auto*

lemma *cc-restr-join*[*simp*]:
cc-restr S $(G \sqcup G')$ = *cc-restr* S $G \sqcup$ *cc-restr* S G'
by *transfer auto*

lemma *cont-cc-restr*: *cont* (*cc-restr* S)
apply (*rule contI*)
apply (*thin-tac chain* -)
apply *transfer*
apply *auto*
done

lemmas *cont-compose*[*OF cont-cc-restr*, *cont2cont*, *simp*]

lemma *cc-restr-mono1*:
$S \subseteq S' \Longrightarrow$ *cc-restr* S $G \sqsubseteq$ *cc-restr* S' G **by** *transfer auto*

lemma *cc-restr-mono2*:
$G \sqsubseteq G' \Longrightarrow$ *cc-restr* S $G \sqsubseteq$ *cc-restr* S G' **by** *transfer auto*

lemma *cc-restr-below-arg*:
cc-restr S $G \sqsubseteq G$ **by** *transfer auto*

lemma *cc-restr-lub*[*simp*]:
cc-restr S (*lub* X) = ($\bigsqcup$ $G{\in}X$. *cc-restr* S G) **by** *transfer auto*

lemma *elem-to-ccField*: $x{-}{-}y{\in}G \Longrightarrow x \in$ *ccField* $G \wedge y \in$ *ccField* G
by *transfer* (*auto simp add*: *Field-def*)

lemma *ccField-to-elem*: $x \in$ *ccField* $G \Longrightarrow \exists$ y. $x{-}{-}y{\in}G$
by *transfer* (*auto simp add*: *Field-def dest*: *symD*)

lemma *cc-restr-intersect*: *ccField* $G \cap ((S - S') \cup (S' - S)) = \{\} \Longrightarrow$ *cc-restr* S G = *cc-restr* S' G
by (*rule CoCalls-eqI*) (*auto dest*: *elem-to-ccField*)

lemma *cc-restr-cc-restr*[*simp*]: *cc-restr* S (*cc-restr* S' G) = *cc-restr* $(S \cap S')$ G
by *transfer auto*

lemma *cc-restr-twist*: *cc-restr* S (*cc-restr* S' G) = *cc-restr* S' (*cc-restr* S G)
by *transfer auto*

lemma *cc-restr-cc-delete-twist*: *cc-restr x* (*cc-delete S G*) = *cc-delete S* (*cc-restr x G*)
  by *transfer auto*

lemma *cc-restr-ccProd*[*simp*]:
  cc-restr S (*ccProd* S_1 S_2) = *ccProd* ($S_1 \cap S$) ($S_2 \cap S$)
  by *transfer auto*

lemma *ccProd-below-cc-restr*:
  ccProd S S′ $\sqsubseteq$ *cc-restr S″ G* $\longleftrightarrow$ *ccProd S S′* $\sqsubseteq$ *G* $\wedge$ ($S = \{\} \vee S' = \{\} \vee S \subseteq S'' \wedge S' \subseteq S''$)
  by *transfer auto*

lemma *cc-restr-eq-subset*: $S \subseteq S' \Longrightarrow$ *cc-restr S′ G* = *cc-restr S′ G2* $\Longrightarrow$ *cc-restr S G* = *cc-restr S G2*
  by *transfer′* (*auto simp add*: *Set.filter-def*)

definition *ccSquare* (-^{2} [*80*] *80*)
  where S^2 = *ccProd S S*

lemma *ccField-ccSquare*[*simp*]: *ccField* (S^2) = *S*
  unfolding *ccSquare-def* **by** *transfer* (*auto simp add*: *Field-def*)

lemma *below-ccSquare*[*iff*]: ($G \sqsubseteq S^2$) = (*ccField G* $\subseteq$ *S*)
  unfolding *ccSquare-def* **by** *transfer* (*auto simp add*: *Field-def*)

lemma *cc-restr-ccSquare*[*simp*]: (S'^2) *G*|‘ *S* = $(S' \cap S)^2$
  unfolding *ccSquare-def* **by** *auto*

lemma *ccSquare-empty*[*simp*]: $\{\}^2 = \bot$
  unfolding *ccSquare-def* **by** *simp*

lift-definition *ccNeighbors* :: *var* $\Rightarrow$ *CoCalls* $\Rightarrow$ *var set*
  is λ *x G*. {*y* .(*y*,*x*) $\in$ *G* $\vee$ (*x*,*y*) $\in$ *G*}**.**

lemma *ccNeighbors-bot*[*simp*]: *ccNeighbors x* $\bot$ = {} **by** *transfer auto*

lemma *cont-ccProd1*:
  cont (λ *S*. *ccProd S S′*)
  apply (*rule contI*)
  apply (*thin-tac chain* -)
  apply (*subst lub-set*)
  apply *transfer*
  apply *auto*
  done

lemma *cont-ccProd2*:
  cont (λ *S′*. *ccProd S S′*)
  apply (*rule contI*)

apply (*thin-tac chain* -)
apply (*subst lub-set*)
apply *transfer*
apply *auto*
done

lemmas *cont-compose2*[*OF cont-ccProd1 cont-ccProd2*, *simp*, *cont2cont*]

lemma *cont-ccNeighbors*[*THEN cont-compose*, *cont2cont*, *simp*]:
cont (λy. *ccNeighbors x y*)
apply (*rule set-contI*)
apply (*thin-tac chain* -)
apply *transfer*
apply *auto*
done

lemma *ccNeighbors-join*[*simp*]: *ccNeighbors x* $(G \sqcup G')$ = *ccNeighbors x G* $\cup$ *ccNeighbors x* G'
by *transfer auto*

lemma *ccNeighbors-ccProd*:
ccNeighbors x (*ccProd S* S') = (*if* $x \in S$ *then* S' *else* {}) $\cup$ (*if* $x \in S'$ *then S else* {})
by *transfer auto*

lemma *ccNeighbors-ccSquare*:
ccNeighbors x (*ccSquare S*) = (*if* $x \in S$ *then S else* {})
unfolding *ccSquare-def* **by** (*auto simp add*: *ccNeighbors-ccProd*)

lemma *ccNeighbors-cc-restr*[*simp*]:
ccNeighbors x (*cc-restr S G*) = (*if* $x \in S$ *then ccNeighbors x G* $\cap$ *S else* {})
by *transfer auto*

lemma *ccNeighbors-mono*:
$G \sqsubseteq G' \Longrightarrow$ *ccNeighbors x G* $\subseteq$ *ccNeighbors x* G'
by *transfer auto*

lemma *subset-ccNeighbors*:
$S \subseteq$ *ccNeighbors x G* $\longleftrightarrow$ *ccProd* {x} $S \sqsubseteq G$
by *transfer* (*auto simp add*: *sym-def*)

lemma *elem-ccNeighbors*[*simp*]:
$y \in$ *ccNeighbors x G* $\longleftrightarrow$ $(y{-}{-}x{\in}G)$
by *transfer* (*auto simp add*: *sym-def*)

lemma *ccNeighbors-ccField*:
ccNeighbors x G $\subseteq$ *ccField G* **by** *transfer* (*auto simp add*: *Field-def*)

lemma *ccNeighbors-disjoint-empty*[*simp*]:

ccNeighbors x G = {} ⟷ *x* ∉ *ccField G*
by *transfer* (*auto simp add*: *Field-def*)

instance *CoCalls* :: *Join-cpo*
by *standard* (*metis coCallsLub-is-lub*)

lemma *ccNeighbors-lub*[*simp*]: *ccNeighbors x* (*lub Gs*) = *lub* (*ccNeighbors x* ' *Gs*)
by *transfer* (*auto simp add*: *lub-set*)

inductive *list-pairs* :: *'a list* ⇒ (*'a* × *'a*) ⇒ *bool*
where *list-pairs xs p* ⟹ *list-pairs* (*x#xs*) *p*
| *y* ∈ *set xs* ⟹ *list-pairs* (*x#xs*) (*x*,*y*)

lift-definition *ccFromList* :: *var list* ⇒ *CoCalls* **is** λ *xs*. {(*x*,*y*). *list-pairs xs* (*x*,*y*) ∨ *list-pairs xs* (*y*,*x*)}
by (*auto intro*: *symI*)

lemma *ccFromList-Nil*[*simp*]: *ccFromList* [] = ⊥
by *transfer* (*auto elim*: *list-pairs.cases*)

lemma *ccFromList-Cons*[*simp*]: *ccFromList* (*x#xs*) = *ccProd* {*x*} (*set xs*) ⊔ *ccFromList xs*
by *transfer* (*auto elim*: *list-pairs.cases intro*: *list-pairs.intros*)

lemma *ccFromList-append*[*simp*]: *ccFromList* (*xs@ys*) = *ccFromList xs* ⊔ *ccFromList ys* ⊔ *ccProd* (*set xs*) (*set ys*)
by (*induction xs*) (*auto simp add*: *ccProd-insert1*[**where** *S'* = *set xs* **for** *xs*])

lemma *ccFromList-filter*[*simp*]:
ccFromList (*filter P xs*) = *cc-restr* {*x*. *P x*} (*ccFromList xs*)
by (*induction xs*) (*auto simp add*: *Collect-conj-eq*)

lemma *ccFromList-replicate*[*simp*]: *ccFromList* (*replicate n x*) = (*if n* ≤ *1 then* ⊥ *else ccProd* {*x*} {*x*})
by (*induction n*) *auto*

definition *ccLinear* :: *var set* ⇒ *CoCalls* ⇒ *bool*
where *ccLinear S G* = (∀ *x*∈*S*. ∀ *y*∈*S*. *x*−−*y*∉*G*)

lemma *ccLinear-bottom*[*simp*]:
ccLinear S ⊥
unfolding *ccLinear-def* **by** *simp*

lemma *ccLinear-empty*[*simp*]:
ccLinear {} *G*
unfolding *ccLinear-def* **by** *simp*

lemma *ccLinear-lub*[*simp*]:
ccLinear S (*lub X*) = (∀ *G*∈*X*. *ccLinear S G*)
unfolding *ccLinear-def* **by** *auto*

lemma *ccLinear-cc-restr*[*intro*]:
$ccLinear\ S\ G \Longrightarrow ccLinear\ S\ (cc\text{-}restr\ S'\ G)$
unfolding *ccLinear-def* **by** *transfer auto*

lemma *ccLinear-join*[*simp*]:
$ccLinear\ S\ (G \sqcup G') \longleftrightarrow ccLinear\ S\ G \wedge ccLinear\ S\ G'$
unfolding *ccLinear-def*
by *transfer auto*

lemma *ccLinear-ccProd*[*simp*]:
$ccLinear\ S\ (ccProd\ S_1\ S_2) \longleftrightarrow S_1 \cap S = \{\} \vee S_2 \cap S = \{\}$
unfolding *ccLinear-def*
by *transfer auto*

lemma *ccLinear-mono1*: $ccLinear\ S'\ G \Longrightarrow S \subseteq S' \Longrightarrow ccLinear\ S\ G$
unfolding *ccLinear-def*
by *transfer auto*

lemma *ccLinear-mono2*: $ccLinear\ S\ G' \Longrightarrow G \sqsubseteq G' \Longrightarrow ccLinear\ S\ G$
unfolding *ccLinear-def*
by *transfer auto*

lemma *ccField-join*[*simp*]:
$ccField\ (G \sqcup G') = ccField\ G \cup ccField\ G'$ **by** *transfer auto*

lemma *ccField-lub*[*simp*]:
$ccField\ (lub\ S) = \bigcup (ccField\ `\ S)$ **by** *transfer auto*

lemma *ccField-ccProd*:
$ccField\ (ccProd\ S\ S') = (if\ S = \{\}\ then\ \{\}\ else\ if\ S' = \{\}\ then\ \{\}\ else\ \ S \cup S')$
by *transfer* (*auto simp add*: *Field-def*)

lemma *ccField-ccProd-subset*:
$ccField\ (ccProd\ S\ S') \subseteq\ \ S \cup S'$
by (*simp add*: *ccField-ccProd*)

lemma *cont-ccField*[*THEN cont-compose*, *simp*, *cont2cont*]:
cont ccField
by (*rule set-contI*) *auto*

end

69 CoCallAnalysisSig.tex

theory *CoCallAnalysisSig*
imports *Terms Arity CoCallGraph*
begin

locale *CoCallAnalysis* =
  fixes *ccExp* :: *exp* ⇒ *Arity* → *CoCalls*
begin
  abbreviation *ccExp-syn* ($\mathcal{G}_{\_}$)
    where $\mathcal{G}_a \equiv (\lambda e.\ ccExp\ e \cdot a)$
  abbreviation *ccExp-bot-syn* ($\mathcal{G}^{\bot}{}_{\_}$)
    where $\mathcal{G}^{\bot}{}_a \equiv (\lambda e.\ fup \cdot (ccExp\ e) \cdot a)$
end

locale *CoCallAnalyisHeap* =
  fixes *ccHeap* :: *heap* ⇒ *exp* ⇒ *Arity* → *CoCalls*

end

70 AList-Utils-HOLCF.tex

theory *AList−Utils−HOLCF*
imports *HOLCF−Utils HOLCF−Join−Classes AList−Utils*
begin

syntax
  -BLubMap :: [*pttrn*, *pttrn*, *'a* ⇀ *'b*, *'b*] ⇒ *'b* ((*3*⨆/-/↦/-/∈/-/./ -) [*0*,*0*,*0*, *10*] *10*)

translations
  ⨆ *k*↦*v*∈*m*. *e* == *CONST lub* (*CONST mapCollect* (λ*k v* . *e*) *m*)

lemma *below-lubmapI*[*intro*]:
  $m\ k = Some\ v \Longrightarrow (e\ k\ v{::}'a{::}Join\text{-}cpo) \sqsubseteq (\bigsqcup\ k{\mapsto}v{\in}m.\ e\ k\ v)$
unfolding *mapCollect-def* **by** *auto*

lemma *lubmap-belowI*[*intro*]:
  $(\bigwedge\ k\ v\ .\ m\ k = Some\ v \Longrightarrow (e\ k\ v{::}'a{::}Join\text{-}cpo) \sqsubseteq u) \Longrightarrow (\bigsqcup\ k{\mapsto}v{\in}m.\ e\ k\ v) \sqsubseteq u$
unfolding *mapCollect-def* **by** *auto*

lemma *lubmap-const-bottom*[*simp*]:
  $(\bigsqcup\ k{\mapsto}v{\in}m.\ \bot) = (\bot{::}'a{::}Join\text{-}cpo)$
  by (*cases m* = *empty*) *auto*

lemma *lubmap-map-upd*[*simp*]:
  fixes *e* :: *'a* ⇒ *'b* ⇒ (*'c* :: *Join-cpo*)
  shows $(\bigsqcup k{\mapsto}v{\in}m(k' \mapsto v').\ e\ k\ v) = e\ k'\ v' \sqcup (\bigsqcup k{\mapsto}v{\in}m(k'{:=}None).\ e\ k\ v)$
  by *simp*

lemma *lubmap-below-cong*:
  assumes $\bigwedge$ *k v. m k = Some v* $\Longrightarrow$ *f1 k v* $\sqsubseteq$ (*f2 k v* :: *'a* :: *Join-cpo*)
  shows ($\bigsqcup$ *k*$\mapsto$*v*$\in$*m. f1 k v*) $\sqsubseteq$ ($\bigsqcup$ *k*$\mapsto$*v*$\in$*m. f2 k v*)
  apply (*rule lubmap-belowI*)
  apply (*rule below-trans*[*OF assms*], *assumption*)
  apply (*rule below-lubmapI*, *assumption*)
  done

lemma *cont2cont-lubmap*[*simp*, *cont2cont*]:
  assumes ($\bigwedge$*k v . cont* (*f k v*))
  shows *cont* (λ*x.* $\bigsqcup$ *k*$\mapsto$*v*$\in$*m.* (*f k v x*) :: *'a* :: *Join-cpo*)
proof (*rule contI2*)
  show *monofun* (λ*x.* $\bigsqcup$*k*$\mapsto$*v*$\in$*m. f k v x*)
    apply (*rule monofunI*)
    apply (*rule lubmap-below-cong*)
    apply (*erule cont2monofunE*[*OF assms*])
    done
next
  fix *Y* :: *nat* $\Rightarrow$ *'d*
  assume *chain Y*
  assume *chain* (λ*i.* $\bigsqcup$*k*$\mapsto$*v*$\in$*m. f k v* (*Y i*))

  show ($\bigsqcup$*k*$\mapsto$*v*$\in$*m. f k v* ($\bigsqcup$ *i. Y i*)) $\sqsubseteq$ ($\bigsqcup$ *i.* $\bigsqcup$*k*$\mapsto$*v*$\in$*m. f k v* (*Y i*))
    apply (*subst cont2contlubE*[*OF assms* ‹*chain Y*›])
    apply (*rule lubmap-belowI*)
    apply (*rule lub-mono*[*OF ch2ch-cont*[*OF assms* ‹*chain Y*›] ‹*chain* (λ*i.* $\bigsqcup$*k*$\mapsto$*v*$\in$*m. f k v* (*Y*
i))›])
    apply (*erule below-lubmapI*)
    done
qed

end

71 CoCallGraph-Nominal.tex

theory *CoCallGraph−Nominal*
imports *CoCallGraph Nominal−HOLCF*
begin

instantiation *CoCalls* :: *pt*
begin
  lift-definition *permute-CoCalls* :: *perm* $\Rightarrow$ *CoCalls* $\Rightarrow$ *CoCalls* **is** *permute*
    by (*auto intro*!: *symI elim*: *symE simp add*: *mem-permute-set*)
instance

apply *standard*
apply (*transfer*, *simp*)+
done
end

instance *CoCalls* :: *cont-pt*
apply *standard*
apply (*rule contI2*)
apply (*rule monofunI*)
apply *transfer*
apply (*metis* (*full-types*) *True-eqvt subset-eqvt*)
apply (*thin-tac chain -*)+
apply *transfer*
apply *simp*
done

lemmas *lub-eqvt*[*OF exists-lub*, *simp*, *eqvt*]

lemma *cc-restr-perm*:
fixes G :: *CoCalls*
assumes *supp p* $\sharp*$ S **and** [*simp*]: *finite S*
shows *cc-restr* S ($p \cdot G$) = *cc-restr* S G
using *assms*
apply $-$
apply *transfer*
apply (*auto simp add*: *mem-permute-set*)
apply (*subst* (*asm*) *perm-supp-eq*, *simp add*: *supp-minus-perm*, *metis* (*full-types*) *fresh-def fresh-star-def supp-set-elem-finite*)+
apply *assumption*
apply (*subst perm-supp-eq*, *simp add*: *supp-minus-perm*, *metis* (*full-types*) *fresh-def fresh-star-def supp-set-elem-finite*)+
apply *assumption*
done

lemma *inCC-eqvt*[*eqvt*]: $\pi \cdot (x{-}{-}y{\in}G) = (\pi{\cdot}x){-}{-}(\pi{\cdot}y){\in}(\pi{\cdot}G)$
by *transfer auto*
lemma *cc-restr-eqvt*[*eqvt*]: $\pi \cdot$ *cc-restr* S G = *cc-restr* ($\pi \cdot S$) ($\pi \cdot G$)
by *transfer* (*perm-simp*, *rule*)
lemma *ccProd-eqvt*[*eqvt*]: $\pi \cdot$ *ccProd* S S' = *ccProd* ($\pi \cdot S$) ($\pi \cdot S'$)
by *transfer* (*perm-simp*, *rule*)
lemma *ccSquare-eqvt*[*eqvt*]: $\pi \cdot$ *ccSquare* S = *ccSquare* ($\pi \cdot S$)
unfolding *ccSquare-def*
by *perm-simp rule*
lemma *ccNeighbors-eqvt*[*eqvt*]: $\pi \cdot$ *ccNeighbors* S G = *ccNeighbors* ($\pi \cdot S$) ($\pi \cdot G$)
by *transfer* (*perm-simp*, *rule*)

end

72 CoCallAnalysisBinds.tex

theory *CoCallAnalysisBinds*
imports *CoCallAnalysisSig AEnv AList−Utils−HOLCF Arity−Nominal CoCallGraph−Nominal*
begin

context *CoCallAnalysis*
begin
definition *ccBind* :: *var* ⇒ *exp* ⇒ ((*AEnv* × *CoCalls*) → *CoCalls*)
  where *ccBind v e* = (Λ (*ae*, *G*). *if* (*v*−−*v*∉*G*) ∨ ¬ *isVal e then cc-restr* (*fv e*) (*fup*·(*ccExp e*)·(*ae v*)) *else ccSquare* (*fv e*))

lemma *ccBind-eq*:
  ccBind v e·(*ae*, *G*) = (*if v*−−*v*∉*G* ∨ ¬ *isVal e then* $\mathcal{G}^{\perp}{}_{ae\ v}$ *e G*|‘ *fv e else* (*fv e*)^2)
  unfolding *ccBind-def*
  apply (*rule cfun-beta-Pair*)
  apply (*rule cont-if-else-above*)
  apply *simp*
  apply *simp*
  apply (*auto dest*: *set-mp*[*OF ccField-cc-restr*])[*1*]

  apply (*case-tac p*, *auto*, *transfer*, *auto*)[*1*]
  apply (*rule adm-subst*[*OF cont-snd*])
  apply (*rule admI*, *thin-tac chain -*, *transfer*, *auto*)
  done

lemma *ccBind-strict*[*simp*]: *ccBind v e* · ⊥ = ⊥
  by (*auto simp add*: *inst-prod-pcpo ccBind-eq simp del*: *Pair-strict*)

lemma *ccField-ccBind*: *ccField* (*ccBind v e*·(*ae*,*G*)) ⊆ *fv e*
  by (*auto simp add*: *ccBind-eq dest*: *set-mp*[*OF ccField-cc-restr*])

definition *ccBinds* :: *heap* ⇒ ((*AEnv* × *CoCalls*) → *CoCalls*)
  where *ccBinds* Γ = (Λ *i*. (⨆ *v*↦*e*∈*map-of* Γ. *ccBind v e*·*i*))

lemma *ccBinds-eq*:
  ccBinds Γ·*i* = (⨆ *v*↦*e*∈*map-of* Γ. *ccBind v e*·*i*)
  unfolding *ccBinds-def*
  by *simp*

lemma *ccBinds-strict*[*simp*]: *ccBinds* Γ·⊥=⊥
  unfolding *ccBinds-eq*
  by (*cases* Γ = []) *simp-all*

lemma *ccBinds-strict'*[*simp*]: *ccBinds* Γ·(⊥,⊥)=⊥

by (*metis CoCallAnalysis.ccBinds-strict Pair-bottom-iff*)

lemma *ccBinds-reorder1*:
assumes *map-of* Γ *v* = *Some e*
shows *ccBinds* Γ = *ccBind v e* $\sqcup$ *ccBinds* (*delete v* Γ)
proof−
from *assms*
have *map-of* Γ = *map-of* ((*v*,*e*) # *delete v* Γ) **by** (*metis map-of-delete-insert*)
thus *?thesis*
by (*auto intro*: *cfun-eqI simp add*: *ccBinds-eq delete-set-none*)
qed

lemma *ccBinds-Nil*[*simp*]:
ccBinds [] = $\bot$
unfolding *ccBinds-def* **by** *simp*

lemma *ccBinds-Cons*[*simp*]:
ccBinds ((*x*,*e*)#Γ) = *ccBind x e* $\sqcup$ *ccBinds* (*delete x* Γ)
by (*subst ccBinds-reorder1*[**where** *v* = *x* **and** *e* = *e*]) *auto*

lemma *ccBind-below-ccBinds*: *map-of* Γ *x* = *Some e* $\Longrightarrow$ *ccBind x e*$\cdot$*ae* $\sqsubseteq$ (*ccBinds* $\Gamma$$\cdot$*ae*)
by (*auto simp add*: *ccBinds-eq*)

lemma *ccField-ccBinds*: *ccField* (*ccBinds* $\Gamma$$\cdot$(*ae*,*G*)) $\subseteq$ *fv* Γ
by (*auto simp add*: *ccBinds-eq dest*: *set-mp*[*OF ccField-ccBind*] *intro*: *set-mp*[*OF map-of-Some-fv-subset*])

definition *ccBindsExtra* :: *heap* $\Rightarrow$ ((*AEnv* $\times$ *CoCalls*) $\rightarrow$ *CoCalls*)
where *ccBindsExtra* Γ = (Λ *i*. *snd i* $\sqcup$ *ccBinds* Γ $\cdot$ *i* $\sqcup$ ($\bigsqcup$*x*$\mapsto$*e*$\in$*map-of* Γ. *ccProd* (*fv e*) (*ccNeighbors x* (*snd i*))))

lemma *ccBindsExtra-simp*: *ccBindsExtra* Γ $\cdot$ *i* =*snd i* $\sqcup$ *ccBinds* Γ $\cdot$ *i* $\sqcup$ ($\bigsqcup$*x*$\mapsto$*e*$\in$*map-of* Γ. *ccProd* (*fv e*) (*ccNeighbors x* (*snd i*)))
unfolding *ccBindsExtra-def* **by** *simp*

lemma *ccBindsExtra-eq*: *ccBindsExtra* $\Gamma$$\cdot$(*ae*,*G*) =
G $\sqcup$ *ccBinds* $\Gamma$$\cdot$(*ae*,*G*) $\sqcup$ ($\bigsqcup$*x*$\mapsto$*e*$\in$*map-of* Γ. *fv e G*$\times$ *ccNeighbors x G*)
unfolding *ccBindsExtra-def* **by** *simp*

lemma *ccBindsExtra-strict*[*simp*]: *ccBindsExtra* Γ $\cdot$ $\bot$ = $\bot$
by (*auto simp add*: *ccBindsExtra-simp inst-prod-pcpo simp del*: *Pair-strict*)

lemma *ccField-ccBindsExtra*:
ccField (*ccBindsExtra* $\Gamma$$\cdot$(*ae*,*G*)) $\subseteq$ *fv* Γ $\cup$ *ccField G*
by (*auto simp add*: *ccBindsExtra-simp elem-to-ccField*
dest!: *set-mp*[*OF ccField-ccBinds*] *set-mp*[*OF ccField-ccProd-subset*] *map-of-Some-fv-subset*)

end

lemma *ccBind-eqvt*[*eqvt*]: π $\bullet$ (*CoCallAnalysis.ccBind cccExp x e*) = *CoCallAnalysis.ccBind* (π

$\cdot$ *cccExp*) ($\pi \cdot x$) ($\pi \cdot e$)
proof−
  {
  fix π *ae G*
  have $\pi \cdot$ ((*CoCallAnalysis.ccBind cccExp x e*) $\cdot$ (*ae*,*G*)) = *CoCallAnalysis.ccBind* ($\pi \cdot$ *cccExp*) ($\pi \cdot x$) ($\pi \cdot e$) $\cdot$ ($\pi \cdot$ *ae*, $\pi \cdot G$)
    unfolding *CoCallAnalysis.ccBind-eq*
    by *perm-simp* (*simp add*: *Abs-cfun-eqvt*)
  }
  thus *?thesis* **by** (*auto intro*: *cfun-eqvtI*)
qed

lemma *ccBinds-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallAnalysis.ccBinds cccExp* Γ) = *CoCallAnalysis.ccBinds* ($\pi \cdot$ *cccExp*) ($\pi \cdot \Gamma$)
  apply (*rule cfun-eqvtI*)
  unfolding *CoCallAnalysis.ccBinds-eq*
  apply (*perm-simp*)
  apply *rule*
  done

lemma *ccBindsExtra-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallAnalysis.ccBindsExtra cccExp* Γ) = *CoCallAnalysis.ccBindsExtra* ($\pi \cdot$ *cccExp*) ($\pi \cdot \Gamma$)
  by (*rule cfun-eqvtI*) (*simp add*: *CoCallAnalysis.ccBindsExtra-def*)

lemma *ccBind-cong*[*fundef-cong*]:
  cccexp1 e = *cccexp2 e* $\Longrightarrow$ *CoCallAnalysis.ccBind cccexp1 x e* = *CoCallAnalysis.ccBind cccexp2 x e*
  apply (*rule cfun-eqI*)
  apply (*case-tac xa*)
  apply (*auto simp add*: *CoCallAnalysis.ccBind-eq*)
  done

lemma *ccBinds-cong*[*fundef-cong*]:
  $[\![$ ($\bigwedge$ *e*. *e* $\in$ *snd* ‘ *set heap2* $\Longrightarrow$ *cccexp1 e* = *cccexp2 e*); *heap1* = *heap2* $]\!]$
    $\Longrightarrow$ *CoCallAnalysis.ccBinds cccexp1 heap1* = *CoCallAnalysis.ccBinds cccexp2 heap2*
  apply (*rule cfun-eqI*)
  unfolding *CoCallAnalysis.ccBinds-eq*
  apply (*rule arg-cong*[*OF mapCollect-cong*])
  apply (*rule arg-cong*[*OF ccBind-cong*])
  apply *auto*
  by (*metis imageI map-of-SomeD snd-conv*)

lemma *ccBindsExtra-cong*[*fundef-cong*]:
  $[\![$ ($\bigwedge$ *e*. *e* $\in$ *snd* ‘ *set heap2* $\Longrightarrow$ *cccexp1 e* = *cccexp2 e*); *heap1* = *heap2* $]\!]$
    $\Longrightarrow$ *CoCallAnalysis.ccBindsExtra cccexp1 heap1* = *CoCallAnalysis.ccBindsExtra cccexp2 heap2*
  apply (*rule cfun-eqI*)
  unfolding *CoCallAnalysis.ccBindsExtra-simp*
  apply (*rule arg-cong2*[*OF ccBinds-cong mapCollect-cong*])

apply *simp+*
done

end

73 ArityAnalysisFix.tex

theory *ArityAnalysisFix*
imports *ArityAnalysisSig ArityAnalysisAbinds*
begin

context *ArityAnalysis*
begin

definition *Afix* :: *heap* ⇒ (*AEnv* → *AEnv*)
where $Afix\ \Gamma = (\Lambda\ ae.\ (\mu\ \ ae'.\ ABinds\ \Gamma \cdot ae' \sqcup ae))$

lemma *Afix-eq*: $Afix\ \Gamma{\cdot}ae = (\mu\ ae'.\ (ABinds\ \Gamma{\cdot}ae') \sqcup ae)$
unfolding *Afix-def* **by** *simp*

lemma *Afix-strict*[*simp*]: $Afix\ \Gamma{\cdot}\bot = \bot$
unfolding *Afix-eq*
by (*rule fix-eqI*) *auto*

lemma *Afix-least-below*: $ABinds\ \Gamma \cdot ae' \sqsubseteq ae' \Longrightarrow ae \sqsubseteq ae' \Longrightarrow Afix\ \Gamma \cdot ae \sqsubseteq ae'$
unfolding *Afix-eq*
by (*auto intro*: *fix-least-below*)

lemma *Afix-unroll*: $Afix\ \Gamma{\cdot}ae = ABinds\ \Gamma \cdot (Afix\ \Gamma{\cdot}ae) \sqcup ae$
unfolding *Afix-eq*
apply (*subst fix-eq*)
by *simp*

lemma *Abinds-below-Afix*: $ABinds\ \Delta \sqsubseteq Afix\ \Delta$
apply (*rule cfun-belowI*)
apply (*simp add*: *Afix-eq*)
apply (*subst fix-eq*, *simp*)
apply (*rule below-trans*[*OF - join-above2*])
apply (*rule monofun-cfun-arg*)
apply (*subst fix-eq*, *simp*)
done

lemma *Afix-above-arg*: $ae \sqsubseteq Afix\ \Gamma \cdot ae$
by (*subst Afix-unroll*) *simp*

lemma *Abinds-Afix-below*[*simp*]: $ABinds\ \Gamma{\cdot}(Afix\ \Gamma{\cdot}ae) \sqsubseteq Afix\ \Gamma{\cdot}ae$
apply (*subst Afix-unroll*) **back**
apply *simp*

done

lemma *Afix-reorder*: *map-of* Γ = *map-of* Δ ⟹ *Afix* Γ = *Afix* Δ
by (*intro cfun-eqI*)(*simp add*: *Afix-eq cong*: *Abinds-reorder*)

lemma *Afix-repeat-singleton*: (μ *xa*. *Afix* Γ·(*esing x*·(*n* ⊔ *xa x*) ⊔ *ae*)) = *Afix* Γ·(*esing x*·*n* ⊔ *ae*)
apply (*rule below-antisym*)
defer
apply (*subst fix-eq*, *simp*)
apply (*intro monofun-cfun-arg join-mono below-refl join-above1*)

apply (*rule fix-least-below*, *simp*)
apply (*rule Afix-least-below*, *simp*)
apply (*intro join-below below-refl iffD2*[*OF esing-below-iff*] *below-trans*[*OF - fun-belowD*[*OF Afix-above-arg*]] *below-trans*[*OF - Afix-above-arg*] *join-above1*)
apply *simp*
done

lemma *Afix-join-fresh*: *ae*′ ‘ (*domA* Δ) ⊆ {⊥} ⟹ *Afix* Δ·(*ae* ⊔ *ae*′) = (*Afix* Δ·*ae*) ⊔ *ae*′
apply (*rule below-antisym*)
apply (*rule Afix-least-below*)
apply (*subst Abinds-join-fresh*, *simp*)
apply (*rule below-trans*[*OF Abinds-Afix-below join-above1*])
apply (*rule join-below*)
apply (*rule below-trans*[*OF Afix-above-arg join-above1*])
apply (*rule join-above2*)
apply (*rule join-below*[*OF monofun-cfun-arg* [*OF join-above1*]])
apply (*rule below-trans*[*OF join-above2 Afix-above-arg*])
done

lemma *Afix-restr-fresh*:
assumes *atom* ‘ *S* ♯* Γ
shows *Afix* Γ·*ae f*|‘ (− *S*) = *Afix* Γ·(*ae f*|‘ (− *S*)) *f*|‘ (− *S*)
unfolding *Afix-eq*
proof (*rule parallel-fix-ind*[**where** *P* = λ *x y* . *x f*|‘ (− *S*) = *y f*|‘ (− *S*)], *goal-cases*)
case *1*
show *?case* **by** *simp*
next
case *2*
show *?case* **..**
next
case *prems*: (*3 aeL aeR*)
have (*ABinds* Γ·*aeL* ⊔ *ae*) *f*|‘ (− *S*) = *ABinds* Γ·*aeL f*|‘ (− *S*) ⊔ *ae f*|‘ (− *S*) **by** (*simp add*: *env-restr-join*)

also have ... = *ABinds* $\Gamma \cdot (aeL\ \ f|^{`}\ (-\ S))\ f|^{`}\ (-\ S) \sqcup ae\ \ f|^{`}\ (-\ S)$ **by** (*rule arg-cong*[*OF ABinds-restr-fresh*[*OF assms*]])
also have ... = *ABinds* $\Gamma \cdot (aeR\ \ f|^{`}\ (-\ S))\ f|^{`}\ (-\ S) \sqcup ae\ \ f|^{`}\ (-\ S)$ **unfolding** *prems* **..**
also have ... = *ABinds* $\Gamma \cdot aeR\ f|^{`}\ (-\ S) \sqcup ae\ \ f|^{`}\ (-\ S)$ **by** (*rule arg-cong*[*OF ABinds-restr-fresh*[*OF assms*, *symmetric*]])
also have ... = (*ABinds* $\Gamma \cdot aeR \sqcup ae\ f|^{`}\ (-\ S))\ f|^{`}\ (-\ S)$ **by** (*simp add*: *env-restr-join*)
finally show *?case* **by** *simp*
qed

lemma *Afix-restr*:
assumes *domA* $\Gamma \subseteq S$
shows *Afix* $\Gamma \cdot ae\ f|^{`}\ S$ = *Afix* $\Gamma \cdot (ae\ \ f|^{`}\ S)\ f|^{`}\ S$
unfolding *Afix-eq*
apply (*rule parallel-fix-ind*[**where** $P = \lambda\ x\ y\ .\ x\ f|^{`}S = y\ \ f|^{`}\ S$])
apply *simp*
apply *rule*
apply (*auto simp add*: *env-restr-join*)
apply (*metis ABinds-restr*[*OF assms*, *symmetric*])
done

lemma *Afix-restr-subst'*:
assumes $\bigwedge\ x'\ e\ a.\ (x',e) \in set\ \Gamma \Longrightarrow Aexp\ e[x::=y] \cdot a\ f|^{`}\ S = Aexp\ e \cdot a\ f|^{`}\ S$
assumes $x \notin S$
assumes $y \notin S$
assumes *domA* $\Gamma \subseteq S$
shows *Afix* $\Gamma[x::h=y] \cdot ae\ f|^{`}\ S$ = *Afix* $\Gamma \cdot (ae\ f|^{`}\ S)\ f|^{`}\ S$
unfolding *Afix-eq*
apply (*rule parallel-fix-ind*[**where** $P = \lambda\ x\ y\ .\ x\ f|^{`}S = y\ \ f|^{`}\ S$])
apply *simp*
apply *rule*
apply (*auto simp add*: *env-restr-join*)
apply (*subst ABinds-restr-subst*[*OF assms*]) **apply** *assumption*
apply (*subst ABinds-restr*[*OF assms*(*4*)]) **back**
apply *simp*
done

lemma *Afix-subst-approx*:
assumes $\bigwedge\ v\ n.\ v \in domA\ \Gamma \Longrightarrow Aexp\ (the\ (map\text{-}of\ \Gamma\ v))[y::=x] \cdot n \sqsubseteq (Aexp\ (the\ (map\text{-}of\ \Gamma\ v)) \cdot n)(y := \bot,\ x := up \cdot 0)$
assumes $x \notin domA\ \Gamma$
assumes $y \notin domA\ \Gamma$
shows *Afix* $\Gamma[y::h= x] \cdot (ae(y := \bot,\ x := up \cdot 0)) \sqsubseteq (Afix\ \Gamma \cdot ae)(y := \bot,\ x := up \cdot 0)$
unfolding *Afix-eq*
proof (*rule parallel-fix-ind*[**where** $P = \lambda\ aeL\ aeR\ .\ aeL \sqsubseteq aeR(y := \bot,\ x := up \cdot 0)$], *goal-cases*)
case *1*
show *?case* **by** *simp*
next
case *2*

show *?case*..
next
case (*3 aeL aeR*)
hence *ABinds* Γ[*y*::*h*=*x*]·*aeL* ⊑ *ABinds* Γ[*y*::*h*=*x*]·(*aeR* (*y* := ⊥, *x* := *up*·*0*)) **by** (*rule monofun-cfun-arg*)
also have ... ⊑ (*ABinds* Γ·*aeR*)(*y* := ⊥, *x* := *up*·*0*)
using *assms*
proof (*induction rule*: *ABinds.induct*, *goal-cases*)
case *1*
thus *?case* **by** *simp*
next
case *prems*: (*2 v e* Γ)
have ⋀*n*. *Aexp e*[*y*::=*x*]·*n* ⊑ (*Aexp e*·*n*)(*y* := ⊥, *x* := *up*·*0*) **using** *prems*(*2*)[**where** *v* = *v*] **by** *auto*
hence *IH1*: ⋀ *n*. *fup*·(*Aexp e*[*y*::=*x*])·*n* ⊑ (*fup*·(*Aexp e*)·*n*)(*y* := ⊥, *x* := *up*·*0*) **by** (*case-tac n*) *auto*

have *ABinds* (*delete v* Γ)[*y*::*h*=*x*]·(*aeR*(*y* := ⊥, *x* := *up*·*0*)) ⊑ (*ABinds* (*delete v* Γ)·*aeR*)(*y* := ⊥, *x* := *up*·*0*)
apply (*rule prems*) **using** *prems*(*2*,*3*,*4*) **by** *fastforce*+
hence *IH2*: *ABinds* (*delete v* Γ[*y*::*h*=*x*])·(*aeR*(*y* := ⊥, *x* := *up*·*0*)) ⊑ (*ABinds* (*delete v* Γ)·*aeR*)(*y* := ⊥, *x* := *up*·*0*)
unfolding *subst-heap-delete*.

have [*simp*]: (*aeR*(*y* := ⊥, *x* := *up*·*0*)) *v* = *aeR v* **using** *prems*(*3*,*4*) **by** *auto*

show *?case* **by** (*simp del*: *fun-upd-apply join-comm*) (*rule join-mono*[*OF IH1 IH2*])
qed
finally have *ABinds* Γ[*y*::*h*=*x*]·*aeL* ⊑ (*ABinds* Γ·*aeR*)(*y* := ⊥, *x* := *up*·*0*)
by *this simp*
thus *?case*
by (*auto simp add*: *join-below-iff elim*: *below-trans*)
qed

end

lemma *Afix-eqvt*[*eqvt*]: π • (*ArityAnalysis.Afix Aexp* Γ) = *ArityAnalysis.Afix* (π • *Aexp*) (π • Γ)
unfolding *ArityAnalysis.Afix-def*
by *perm-simp* (*simp add*: *Abs-cfun-eqvt*)

lemma *Afix-cong*[*fundef-cong*]:
⟦ (⋀ *e*. *e* ∈ *snd* ‘ *set heap2* ⟹ *aexp1 e* = *aexp2 e*); *heap1* = *heap2* ⟧
⟹ *ArityAnalysis.Afix aexp1 heap1* = *ArityAnalysis.Afix aexp2 heap2*
unfolding *ArityAnalysis.Afix-def* **by** (*metis Abinds-cong*)

context *EdomArityAnalysis*

begin

lemma *Afix-edom*: *edom* (*Afix* Γ · *ae*) ⊆ *fv* Γ ∪ *edom ae*
  unfolding *Afix-eq*
  by (*rule fix-ind*[**where** $P = \lambda\ ae'\ .\ edom\ ae' \subseteq fv\ \Gamma \cup edom\ ae$])
    (*auto dest*: *set-mp*[*OF edom-AnalBinds*])

lemma *ABinds-lookup-fresh*:
  $atom\ v \sharp \Gamma \Longrightarrow (ABinds\ \Gamma \cdot ae)\ v = \bot$
by (*induct* Γ *rule*: *ABinds.induct*) (*auto simp add*: *fresh-Cons fresh-Pair fup-Aexp-lookup-fresh fresh-delete*)

lemma *Afix-lookup-fresh*:
  assumes $atom\ v \sharp \Gamma$
  shows $(Afix\ \Gamma \cdot ae)\ v = ae\ v$
  apply (*rule below-antisym*)
  apply (*subst Afix-eq*)
  apply (*rule fix-ind*[**where** $P = \lambda\ ae'.\ ae'\ v \sqsubseteq ae\ v$])
  apply (*auto simp add*: *ABinds-lookup-fresh*[*OF assms*] *fun-belowD*[*OF Afix-above-arg*])
  done

lemma *Afix-comp2join-fresh*:
  $atom$ ‘ $(domA\ \Delta) \sharp{*}\ \Gamma \Longrightarrow ABinds\ \Delta \cdot (Afix\ \Gamma \cdot ae) = ABinds\ \Delta \cdot ae$
proof (*induct* Δ *rule*: *ABinds.induct*)
  case *1* **show** *?case* **by** (*simp add*: *Afix-above-arg del*: *fun-meet-simp*)
next
  case (*2 v e* Δ)
  from *2*(*2*)
  have $atom\ v \sharp \Gamma$ **and** $atom$ ‘ $domA\ (delete\ v\ \Delta) \sharp{*}\ \Gamma$
    by (*auto simp add*: *fresh-star-def*)
  from *2*(*1*)[*OF this*(*2*)]
  show *?case* **by** (*simp del*: *fun-meet-simp add*: *Afix-lookup-fresh*[*OF* ‹$atom\ v \sharp \Gamma$›])
qed

lemma *Afix-append-fresh*:
  assumes $atom$ ‘ $domA\ \Delta \sharp{*}\ \Gamma$
  shows $Afix\ (\Delta\ @\ \Gamma) \cdot ae = Afix\ \Gamma \cdot (Afix\ \Delta \cdot ae)$
proof (*rule below-antisym*)
  show ∗: $Afix\ (\Delta\ @\ \Gamma) \cdot ae \sqsubseteq Afix\ \Gamma \cdot (Afix\ \Delta \cdot ae)$
  apply (*rule Afix-least-below*)
  apply (*simp add*: *Abinds-append-disjoint*[*OF fresh-distinct*[*OF assms*]] *Afix-comp2join-fresh*[*OF assms*])
  apply (*rule below-trans*[*OF join-mono*[*OF Abinds-Afix-below Abinds-Afix-below*]])
  apply (*simp-all add*: *Afix-above-arg below-trans*[*OF Afix-above-arg Afix-above-arg*])
  done
next
  show $Afix\ \Gamma \cdot (Afix\ \Delta \cdot ae) \sqsubseteq Afix\ (\Delta\ @\ \Gamma) \cdot ae$
  proof (*rule Afix-least-below*)
    show $ABinds\ \Gamma \cdot (Afix\ (\Delta\ @\ \Gamma) \cdot ae) \sqsubseteq Afix\ (\Delta\ @\ \Gamma) \cdot ae$

apply (*rule below-trans*[*OF* - *Abinds-Afix-below*])
apply (*subst Abinds-append-disjoint*[*OF fresh-distinct*[*OF assms*]])
apply *simp*
done
have *ABinds* $\Delta \cdot (Afix\ (\Delta\ @\ \Gamma) \cdot ae) \sqsubseteq Afix\ (\Delta\ @\ \Gamma) \cdot ae$
apply (*rule below-trans*[*OF* - *Abinds-Afix-below*])
apply (*subst Abinds-append-disjoint*[*OF fresh-distinct*[*OF assms*]])
apply *simp*
done
thus $Afix\ \Delta \cdot ae \sqsubseteq Afix\ (\Delta\ @\ \Gamma) \cdot ae$
apply (*rule Afix-least-below*)
apply (*rule Afix-above-arg*)
done
qed
qed

lemma *Afix-e-to-heap*:
$Afix\ (delete\ x\ \Gamma) \cdot (fup \cdot (Aexp\ e) \cdot n \sqcup ae) \sqsubseteq Afix\ ((x,\ e)\ \#\ delete\ x\ \Gamma) \cdot (esing\ x \cdot n \sqcup ae)$
apply (*simp add*: *Afix-eq*)
apply (*rule fix-least-below*, *simp*)
apply (*intro join-below*)
apply (*subst fix-eq*, *simp*)
apply (*subst fix-eq*, *simp*)

apply (*rule below-trans*[*OF* - *join-above2*])
apply (*rule below-trans*[*OF* - *join-above2*])
apply (*rule below-trans*[*OF* - *join-above2*])
apply (*rule monofun-cfun-arg*)
apply (*subst fix-eq*, *simp*)

apply (*subst fix-eq*, *simp*) **back apply** (*simp add*: *below-trans*[*OF* - *join-above2*])
done

lemma *Afix-e-to-heap'*:
$Afix\ (delete\ x\ \Gamma) \cdot (Aexp\ e \cdot n) \sqsubseteq Afix\ ((x,\ e)\ \#\ delete\ x\ \Gamma) \cdot (esing\ x \cdot (up \cdot n))$
using *Afix-e-to-heap*[**where** $ae = \bot$ **and** $n = up \cdot n$] **by** *simp*

end

end

74 CoCallFix.tex

theory *CoCallFix*
imports *CoCallAnalysisSig CoCallAnalysisBinds ArityAnalysisSig Env−Nominal ArityAnalysisFix*
begin

locale *CoCallArityAnalysis* =
  fixes *cccExp* :: *exp* ⇒ (*Arity* → *AEnv* × *CoCalls*)
begin

definition *Aexp* :: *exp* ⇒ (*Arity* → *AEnv*)
  where *Aexp e* = (Λ *a*. *fst* (*cccExp e* · *a*))

sublocale *ArityAnalysis Aexp*.

abbreviation *Aexp-syn'* ($\mathcal{A}_{-}$) **where** $\mathcal{A}_a \equiv (\lambda e.\ Aexp\ e \cdot a)$
abbreviation *Aexp-bot-syn'* ($\mathcal{A}^{\perp}{}_{-}$) **where** $\mathcal{A}^{\perp}{}_a \equiv (\lambda e.\ fup \cdot (Aexp\ e) \cdot a)$

lemma *Aexp-eq*:
  $\mathcal{A}_a\ e = fst\ (cccExp\ e \cdot a)$
  unfolding *Aexp-def* **by** (*rule beta-cfun*) (*intro cont2cont*)

lemma *fup-Aexp-eq*:
  $fup \cdot (Aexp\ e) \cdot a = fst\ (fup \cdot (cccExp\ e) \cdot a)$
  by (*cases a*)(*simp-all add*: *Aexp-eq*)

definition *CCexp* :: *exp* ⇒ (*Arity* → *CoCalls*) **where** *CCexp* Γ = (Λ *a*. *snd* (*cccExp* Γ·*a*))
lemma *CCexp-eq*:
  $CCexp\ e \cdot a = snd\ (cccExp\ e \cdot a)$
  unfolding *CCexp-def* **by** (*rule beta-cfun*) (*intro cont2cont*)

lemma *fup-CCexp-eq*:
  $fup \cdot (CCexp\ e) \cdot a = snd\ (fup \cdot (cccExp\ e) \cdot a)$
  by (*cases a*)(*simp-all add*: *CCexp-eq*)

sublocale *CoCallAnalysis CCexp*.

definition *CCfix* :: *heap* ⇒ (*AEnv* × *CoCalls*) → *CoCalls*
  where *CCfix* Γ = (Λ *aeG*. (μ *G'*. *ccBindsExtra* Γ·(*fst aeG* , *G'*) ⊔ *snd aeG*))

lemma *CCfix-eq*:
  CCfix Γ·(*ae*,*G*) = (μ *G'*. *ccBindsExtra* Γ·(*ae*, *G'*) ⊔ *G*)
  unfolding *CCfix-def*
  by *simp*

lemma *CCfix-unroll*: *CCfix* Γ·(*ae*,*G*) = *ccBindsExtra* Γ·(*ae*, *CCfix* Γ·(*ae*,*G*)) ⊔ *G*
  unfolding *CCfix-eq*
  apply (*subst fix-eq*)
  apply *simp*
  done

lemma *fup-ccExp-restr-subst'*:

assumes $\bigwedge$ *a*. *cc-restr S* (*CCexp e*[*x*::=*y*]·*a*) = *cc-restr S* (*CCexp e*·*a*)
shows *cc-restr S* (*fup*·(*CCexp e*[*x*::=*y*])·*a*) = *cc-restr S* (*fup*·(*CCexp e*)·*a*)
using *assms*
by (*cases a*) (*auto simp del*: *cc-restr-cc-restr simp add*: *cc-restr-cc-restr*[*symmetric*])

lemma *ccBindsExtra-restr-subst'*:
assumes $\bigwedge$ *x' e a*. (*x'*,*e*) ∈ *set* Γ ⟹ *cc-restr S* (*CCexp e*[*x*::=*y*]·*a*) = *cc-restr S* (*CCexp e*·*a*)
assumes *x* ∉ *S*
assumes *y* ∉ *S*
assumes *domA* Γ ⊆ *S*
shows *cc-restr S* (*ccBindsExtra* Γ[*x*::*h*=*y*]·(*ae*, *G*))
= *cc-restr S* (*ccBindsExtra* Γ·(*ae f*|‘ *S* , *cc-restr S G*))
apply (*simp add*: *ccBindsExtra-simp ccBinds-eq ccBind-eq Int-absorb2*[*OF assms*(*4*)] *fv-subst-int*[*OF assms*(*3*,*2*)])
apply (*intro arg-cong2*[**where** *f* = *op* ⊔] *refl arg-cong*[*OF mapCollect-cong*])
apply (*subgoal-tac k* ∈ *S*)
apply (*auto intro*: *fup-ccExp-restr-subst'*[*OF assms*(*1*)[*OF map-of-SomeD*]] *simp add*: *fv-subst-int*[*OF assms*(*3*,*2*)] *fv-subst-int2*[*OF assms*(*3*,*2*)] *ccSquare-def*)[*1*]
apply (*metis assms*(*4*) *contra-subsetD domI dom-map-of-conv-domA*)
apply (*subgoal-tac k* ∈ *S*)
apply (*auto intro*: *fup-ccExp-restr-subst'*[*OF assms*(*1*)[*OF map-of-SomeD*]]
simp add: *fv-subst-int*[*OF assms*(*3*,*2*)] *fv-subst-int2*[*OF assms*(*3*,*2*)] *ccSquare-def cc-restr-twist*[**where** *S* = *S*] *simp del*: *cc-restr-cc-restr*)[*1*]
apply (*subst fup-ccExp-restr-subst'*[*OF assms*(*1*)[*OF map-of-SomeD*]], *assumption*)
apply (*simp add*: *fv-subst-int*[*OF assms*(*3*,*2*)] *fv-subst-int2*[*OF assms*(*3*,*2*)])
apply (*subst fup-ccExp-restr-subst'*[*OF assms*(*1*)[*OF map-of-SomeD*]], *assumption*)
apply (*simp add*: *fv-subst-int*[*OF assms*(*3*,*2*)] *fv-subst-int2*[*OF assms*(*3*,*2*)])
apply (*metis assms*(*4*) *contra-subsetD domI dom-map-of-conv-domA*)
done

lemma *ccBindsExtra-restr*:
assumes *domA* Γ ⊆ *S*
shows *cc-restr S* (*ccBindsExtra* Γ·(*ae*, *G*)) = *cc-restr S* (*ccBindsExtra* Γ·(*ae f*|‘ *S*, *cc-restr S G*))
using *assms*
apply (*simp add*: *ccBindsExtra-simp ccBinds-eq ccBind-eq Int-absorb2*)
apply (*intro arg-cong2*[**where** *f* = *op* ⊔] *refl arg-cong*[*OF mapCollect-cong*])
apply (*subgoal-tac k* ∈ *S*)
apply *simp*
apply (*metis contra-subsetD domI dom-map-of-conv-domA*)
apply (*subgoal-tac k* ∈ *S*)
apply *simp*
apply (*metis contra-subsetD domI dom-map-of-conv-domA*)
done

lemma *CCfix-restr*:
assumes *domA* Γ ⊆ *S*
shows *cc-restr S* (*CCfix* Γ·(*ae*, *G*)) = *cc-restr S* (*CCfix* Γ·(*ae f*|‘ *S*, *cc-restr S G*))

unfolding *CCfix-def*
apply *simp*
apply (*rule parallel-fix-ind*[**where** $P = \lambda\ x\ y\ .\ cc\text{-}restr\ S\ x = cc\text{-}restr\ S\ y$])
apply *simp*
apply *rule*
apply *simp*
apply (*subst* (*1 2*) *ccBindsExtra-restr*[*OF assms*])
apply (*auto*)
done

lemma *ccField-CCfix*:
shows $ccField\ (CCfix\ \Gamma\cdot(ae,\ G)) \subseteq fv\ \Gamma \cup ccField\ G$
unfolding *CCfix-def*
apply *simp*
apply (*rule fix-ind*[**where** $P = \lambda\ x\ .\ ccField\ x \subseteq fv\ \Gamma \cup ccField\ G$])
apply (*auto dest*!: *set-mp*[*OF ccField-ccBindsExtra*])
done

lemma *CCfix-restr-subst'*:
assumes $\bigwedge\ x'\ e\ a.\ (x',e) \in set\ \Gamma \Longrightarrow cc\text{-}restr\ S\ (CCexp\ e[x::=y]\cdot a) = cc\text{-}restr\ S\ (CCexp\ e\cdot a)$
assumes $x \notin S$
assumes $y \notin S$
assumes $domA\ \Gamma \subseteq S$
shows $cc\text{-}restr\ S\ (CCfix\ \Gamma[x::h=y]\cdot(ae,\ G)) = cc\text{-}restr\ S\ (CCfix\ \Gamma\cdot(ae\ f|`\ S,\ cc\text{-}restr\ S\ G))$
unfolding *CCfix-def*
apply *simp*
apply (*rule parallel-fix-ind*[**where** $P = \lambda\ x\ y\ .\ cc\text{-}restr\ S\ x = cc\text{-}restr\ S\ y$])
apply *simp*
apply *rule*
apply *simp*
apply (*subst ccBindsExtra-restr-subst'*[*OF assms*], *assumption*)
apply (*subst ccBindsExtra-restr*[*OF assms*(*4*)]) **back**
apply (*auto*)
done

end

lemma *Aexp-eqvt*[*eqvt*]: $\pi \bullet (CoCallArityAnalysis.Aexp\ cccExp\ e) = CoCallArityAnalysis.Aexp\ (\pi \bullet cccExp)\ (\pi \bullet e)$
apply (*rule cfun-eqvtI*) **unfolding** *CoCallArityAnalysis.Aexp-eq* **by** *perm-simp rule*

lemma *CCexp-eqvt*[*eqvt*]: $\pi \bullet (CoCallArityAnalysis.CCexp\ cccExp\ e) = CoCallArityAnalysis.CCexp\ (\pi \bullet cccExp)\ (\pi \bullet e)$
apply (*rule cfun-eqvtI*) **unfolding** *CoCallArityAnalysis.CCexp-eq* **by** *perm-simp rule*

lemma *CCfix-eqvt*[*eqvt*]: $\pi \bullet (CoCallArityAnalysis.CCfix\ cccExp\ \Gamma) = CoCallArityAnalysis.CCfix$

$(\pi \cdot cccExp)$ $(\pi \cdot \Gamma)$
unfolding *CoCallArityAnalysis.CCfix-def* **by** *perm-simp* (*simp-all add*: *Abs-cfun-eqvt*)

lemma *ccFix-cong*[*fundef-cong*]:
$\llbracket$ ($\bigwedge$ *e*. *e* $\in$ *snd* ' *set heap2* $\Longrightarrow$ *cccexp1 e* = *cccexp2 e*); *heap1* = *heap2* $\rrbracket$
$\Longrightarrow$ *CoCallArityAnalysis.CCfix cccexp1 heap1* = *CoCallArityAnalysis.CCfix cccexp2 heap2*
unfolding *CoCallArityAnalysis.CCfix-def*
apply (*rule arg-cong*) **back**
apply (*rule ccBindsExtra-cong*)
apply (*auto simp add*: *CoCallArityAnalysis.CCexp-def*)
done

context *CoCallArityAnalysis*
begin
definition *cccFix* :: *heap* $\Rightarrow$ ((*AEnv* $\times$ *CoCalls*) $\to$ (*AEnv* $\times$ *CoCalls*))
where *cccFix* Γ = (Λ *i*. (*Afix* $\Gamma\cdot$(*fst i* $\sqcup$ (λ-.*up*$\cdot$*0*) *f*|' *thunks* Γ), *CCfix* $\Gamma\cdot$(*Afix* $\Gamma\cdot$(*fst i* $\sqcup$ (λ-.*up*$\cdot$*0*) *f*|' (*thunks* Γ)), *snd i*)))

lemma *cccFix-eq*:
cccFix $\Gamma\cdot i$ = (*Afix* $\Gamma\cdot$(*fst i* $\sqcup$ (λ-.*up*$\cdot$*0*) *f*|' *thunks* Γ), *CCfix* $\Gamma\cdot$(*Afix* $\Gamma\cdot$(*fst i* $\sqcup$ (λ-.*up*$\cdot$*0*) *f*|' (*thunks* Γ)), *snd i*))
unfolding *cccFix-def*
by (*rule beta-cfun*)(*intro cont2cont*)
end

lemma *cccFix-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.cccFix cccExp* Γ) = *CoCallArityAnalysis.cccFix* $(\pi \cdot cccExp)$ $(\pi \cdot \Gamma)$
apply (*rule cfun-eqvtI*) **unfolding** *CoCallArityAnalysis.cccFix-eq* **by** *perm-simp rule*

lemma *cccFix-cong*[*fundef-cong*]:
$\llbracket$ ($\bigwedge$ *e*. *e* $\in$ *snd* ' *set heap2* $\Longrightarrow$ *cccexp1 e* = *cccexp2 e*); *heap1* = *heap2* $\rrbracket$
$\Longrightarrow$ *CoCallArityAnalysis.cccFix cccexp1 heap1* = *CoCallArityAnalysis.cccFix cccexp2 heap2*
unfolding *CoCallArityAnalysis.cccFix-def*
apply (*rule cfun-eqI*)
apply *auto*
apply (*rule arg-cong*[*OF Afix-cong*], *auto simp add*: *CoCallArityAnalysis.Aexp-def*)[*1*]
apply (*rule arg-cong2*[*OF ccFix-cong Afix-cong*])
apply (*auto simp add*: *CoCallArityAnalysis.Aexp-def*)
done

74.1 The non-recursive case

definition *ABind-nonrec* :: *var* $\Rightarrow$ *exp* $\Rightarrow$ *AEnv* $\times$ *CoCalls* $\to$ *Arity*$_\bot$
where
ABind-nonrec x e = (Λ *i*. (*if isVal e* $\vee$ *x*$--$*x*$\notin$(*snd i*) *then fst i x else up*$\cdot$*0*))

lemma *ABind-nonrec-eq*:

ABind-nonrec x e·(*ae*,*G*) = (*if isVal e* ∨ *x*−−*x*∉*G then ae x else up*·*0*)
unfolding *ABind-nonrec-def*
apply (*subst beta-cfun*)
apply (*rule cont-if-else-above*)
apply *auto*
by (*metis in-join join-self-below*(*4*))

lemma *ABind-nonrec-eqvt*[*eqvt*]: π • (*ABind-nonrec x e*) = *ABind-nonrec* (π • *x*) (π • *e*)
apply (*rule cfun-eqvtI*)
apply (*case-tac xa*, *simp*)
unfolding *ABind-nonrec-eq*
by *perm-simp rule*

lemma *ABind-nonrec-above-arg*:
ae x ⊑ *ABind-nonrec x e* · (*ae*, *G*)
unfolding *ABind-nonrec-eq* **by** *auto*

definition *Aheap-nonrec* **where**
Aheap-nonrec x e = (Λ *i*. *esing x*·(*ABind-nonrec x e*·*i*))

lemma *Aheap-nonrec-simp*:
Aheap-nonrec x e·*i* = *esing x*·(*ABind-nonrec x e*·*i*)
unfolding *Aheap-nonrec-def* **by** *simp*

lemma *Aheap-nonrec-lookup*[*simp*]:
(*Aheap-nonrec x e*·*i*) *x* = *ABind-nonrec x e*·*i*
unfolding *Aheap-nonrec-simp* **by** *simp*

lemma *Aheap-nonrec-eqvt'*[*eqvt*]:
π • (*Aheap-nonrec x e*) = *Aheap-nonrec* (π • *x*) (π • *e*)
apply (*rule cfun-eqvtI*)
unfolding *Aheap-nonrec-simp*
by (*perm-simp*, *rule*)

context *CoCallArityAnalysis*
begin

definition *Afix-nonrec*
where *Afix-nonrec x e* = (Λ *i*. *fup*·(*Aexp e*)·(*ABind-nonrec x e* · *i*) ⊔ *fst i*)

lemma *Afix-nonrec-eq*[*simp*]:
Afix-nonrec x e · *i* = *fup*·(*Aexp e*)·(*ABind-nonrec x e* · *i*) ⊔ *fst i*
unfolding *Afix-nonrec-def*
by (*rule beta-cfun*) *simp*

definition *CCfix-nonrec*
where *CCfix-nonrec x e* = (Λ *i*. *ccBind x e* · (*Aheap-nonrec x e*·*i*, *snd i*) ⊔ *ccProd* (*fv e*) (*ccNeighbors x* (*snd i*) − (*if isVal e then* {} *else* {*x*})) ⊔ *snd i*)

lemma *CCfix-nonrec-eq*[*simp*]:
CCfix-nonrec x e · *i* = *ccBind x e*·(*Aheap-nonrec x e*·*i*, *snd i*) ⊔ *ccProd* (*fv e*) (*ccNeighbors x* (*snd i*) − (*if isVal e then* {} *else* {*x*})) ⊔ *snd i*
unfolding *CCfix-nonrec-def*
by (*rule beta-cfun*) (*intro cont2cont*)

definition *cccFix-nonrec* :: *var* ⇒ *exp* ⇒ ((*AEnv* × *CoCalls*) → (*AEnv* × *CoCalls*))
where *cccFix-nonrec x e* = (Λ *i*. (*Afix-nonrec x e* ·*i* , *CCfix-nonrec x e* ·*i*))

lemma *cccFix-nonrec-eq*[*simp*]:
cccFix-nonrec x e·*i* = (*Afix-nonrec x e* ·*i* , *CCfix-nonrec x e* ·*i*)
unfolding *cccFix-nonrec-def*
by (*rule beta-cfun*) (*intro cont2cont*)

end

lemma *AFix-nonrec-eqvt*[*eqvt*]: π · (*CoCallArityAnalysis.Afix-nonrec cccExp x e*) = *CoCallArityAnalysis.Afix-nonrec* (π · *cccExp*) (π · *x*) (π · *e*)
apply (*rule cfun-eqvtI*)
unfolding *CoCallArityAnalysis.Afix-nonrec-eq*
by *perm-simp rule*

lemma *CCFix-nonrec-eqvt*[*eqvt*]: π · (*CoCallArityAnalysis.CCfix-nonrec cccExp x e*) = *CoCallArityAnalysis.CCfix-nonrec* (π · *cccExp*) (π · *x*) (π · *e*)
apply (*rule cfun-eqvtI*)
unfolding *CoCallArityAnalysis.CCfix-nonrec-eq*
by *perm-simp rule*

lemma *cccFix-nonrec-eqvt*[*eqvt*]: π · (*CoCallArityAnalysis.cccFix-nonrec cccExp x e*) = *CoCallArityAnalysis.cccFix-nonrec* (π · *cccExp*) (π · *x*) (π · *e*)
apply (*rule cfun-eqvtI*)
unfolding *CoCallArityAnalysis.cccFix-nonrec-eq*
by *perm-simp rule*

74.2 Combining the cases

context *CoCallArityAnalysis*
begin
definition *cccFix-choose* :: *heap* ⇒ ((*AEnv* × *CoCalls*) → (*AEnv* × *CoCalls*))
where *cccFix-choose* Γ = (*if nonrec* Γ *then case-prod cccFix-nonrec* (*hd* Γ) *else cccFix* Γ)

lemma *cccFix-choose-simp1*[*simp*]:
¬ *nonrec* Γ ⟹ *cccFix-choose* Γ = *cccFix* Γ
unfolding *cccFix-choose-def* **by** *simp*

lemma *cccFix-choose-simp2*[*simp*]:

$x \notin fv\ e \Longrightarrow cccFix\text{-}choose\ [(x,e)] = cccFix\text{-}nonrec\ x\ e$
unfolding *cccFix-choose-def nonrec-def* **by** *auto*

end

lemma *cccFix-choose-eqvt*[*eqvt*]: $\pi \cdot (CoCallArityAnalysis.cccFix\text{-}choose\ cccExp\ \Gamma) = CoCallArityAnalysis.cccFix\text{-}choose\ (\pi \cdot cccExp)\ (\pi \cdot \Gamma)$
unfolding *CoCallArityAnalysis.cccFix-choose-def*
apply (*cases nonrec* π *rule*: *eqvt-cases*[**where** $x = \Gamma$])
apply (*perm-simp*, *rule*)
apply *simp*
apply (*erule nonrecE*)
apply (*simp*)

apply *simp*
done

lemma *cccFix-nonrec-cong*[*fundef-cong*]:
$cccexp1\ e = cccexp2\ e \Longrightarrow CoCallArityAnalysis.cccFix\text{-}nonrec\ cccexp1\ x\ e = CoCallArityAnalysis.cccFix\text{-}nonrec\ cccexp2\ x\ e$
apply (*rule cfun-eqI*)
unfolding *CoCallArityAnalysis.cccFix-nonrec-eq*
unfolding *CoCallArityAnalysis.Afix-nonrec-eq*
unfolding *CoCallArityAnalysis.CCfix-nonrec-eq*
unfolding *CoCallArityAnalysis.fup-Aexp-eq*
apply (*simp only*:)
apply (*rule arg-cong*[*OF ccBind-cong*])
apply *simp*
unfolding *CoCallArityAnalysis.CCexp-def*
apply *simp*
done

lemma *cccFix-choose-cong*[*fundef-cong*]:
$\llbracket (\bigwedge e.\ e \in snd\ `\ set\ heap2 \Longrightarrow cccexp1\ e = cccexp2\ e);\ heap1 = heap2\ \rrbracket$
$\Longrightarrow CoCallArityAnalysis.cccFix\text{-}choose\ cccexp1\ heap1 = CoCallArityAnalysis.cccFix\text{-}choose\ cccexp2\ heap2$
unfolding *CoCallArityAnalysis.cccFix-choose-def*
apply (*rule cfun-eqI*)
apply (*auto elim*!: *nonrecE*)
apply (*rule arg-cong*[*OF cccFix-nonrec-cong*], *auto*)
apply (*rule arg-cong*[*OF cccFix-cong*], *auto*)[*1*]
done

end

75 CoCallAnalysisImpl.tex

theory *CoCallAnalysisImpl*

imports *Arity−Nominal Nominal−HOLCF Env−Nominal Env−Set−Cpo Env−HOLCF CoCallFix*
begin

fun *combined-restrict* :: *var set* ⇒ (*AEnv* × *CoCalls*) ⇒ (*AEnv* × *CoCalls*)
  where *combined-restrict S* (*env*, *G*) = (*env f*|' *S*, *cc-restr S G*)

lemma *fst-combined-restrict*[*simp*]:
  fst (*combined-restrict S p*) = *fst p f*|' *S*
  by (*cases p*, *simp*)

lemma *snd-combined-restrict*[*simp*]:
  snd (*combined-restrict S p*) = *cc-restr S* (*snd p*)
  by (*cases p*, *simp*)

lemma *combined-restrict-eqvt*[*eqvt*]:
  shows π · *combined-restrict S p* = *combined-restrict* (π · *S*) (π · *p*)
  by (*cases p*) *auto*

lemma *combined-restrict-cont*:
  cont (λ*x*. *combined-restrict S x*)
proof−
  have *cont* (λ(*env*, *G*). *combined-restrict S* (*env*, *G*)) **by** *simp*
  then show *?thesis* **by** (*simp only*: *case-prod-eta*)
qed
lemmas *cont-compose*[*OF combined-restrict-cont*, *cont2cont*, *simp*]

lemma *combined-restrict-perm*:
  assumes *supp* π ♯* *S* **and** [*simp*]: *finite S*
  shows *combined-restrict S* (π · *p*) = *combined-restrict S p*
proof(*cases p*)
  fix *env* :: *AEnv* **and** *G* :: *CoCalls*
  assume *p* = (*env*, *G*)
  moreover
  from *assms*
  have *env-restr S* (π · *env*) = *env-restr S env* **by** (*rule env-restr-perm*)
  moreover
  from *assms*
  have *cc-restr S* (π · *G*) = *cc-restr S G* **by** (*rule cc-restr-perm*)
  ultimately
  show *?thesis* **by** *simp*
qed

definition *predCC* :: *var set* ⇒ (*Arity* → *CoCalls*) ⇒ (*Arity* → *CoCalls*)
  where *predCC S f* = (Λ *a*. *if a* ≠ *0 then cc-restr S* (*f*·(*pred*·*a*)) *else ccSquare S*)

lemma *predCC-eq*:
  shows *predCC S f* · *a* = (*if a* ≠ *0 then cc-restr S* (*f*·(*pred*·*a*)) *else ccSquare S*)
  unfolding *predCC-def*
  apply (*rule beta-cfun*)

apply (*rule cont-if-else-above*)
apply (*auto dest*: *set-mp*[*OF ccField-cc-restr*])
done

lemma *predCC-eqvt*[*eqvt*, *simp*]: $\pi \cdot (predCC\ S\ f) = predCC\ (\pi \cdot S)\ (\pi \cdot f)$
apply (*rule cfun-eqvtI*)
unfolding *predCC-eq*
by *perm-simp rule*

lemma *cc-restr-predCC*:
$cc\text{-}restr\ S\ (predCC\ S'\ f \cdot n) = (predCC\ (S' \cap S)\ (\Lambda\ n.\ cc\text{-}restr\ S\ (f \cdot n))) \cdot n$
unfolding *predCC-eq*
by (*auto simp add*: *inf-commute ccSquare-def*)

lemma *cc-restr-predCC'*[*simp*]:
$cc\text{-}restr\ S\ (predCC\ S\ f \cdot n) = predCC\ S\ f \cdot n$
unfolding *predCC-eq* **by** *simp*

nominal-function
cCCexp :: *exp* ⇒ (*Arity* → *AEnv* × *CoCalls*)
where
cCCexp (*Var x*) = (Λ *n* . (*esing x* · (*up* · *n*), ⊥))
| *cCCexp* (*Lam* [*x*]. *e*) = (Λ *n* . *combined-restrict* (*fv* (*Lam* [*x*]. *e*)) (*fst* (*cCCexp e*·(*pred*·*n*)), *predCC* (*fv* (*Lam* [*x*]. *e*)) (Λ *a*. *snd*(*cCCexp e*·*a*))·*n*))
| *cCCexp* (*App e x*) = (Λ *n* . (*fst* (*cCCexp e*·(*inc*·*n*)) ⊔ (*esing x* · (*up*·*0*)), *snd* (*cCCexp e*·(*inc*·*n*)) ⊔ *ccProd* {*x*} (*insert x* (*fv e*))))
| *cCCexp* (*Let* Γ *e*) = (Λ *n* . *combined-restrict* (*fv* (*Let* Γ *e*)) (*CoCallArityAnalysis.cccFix-choose cCCexp* Γ · (*cCCexp e*·*n*)))
| *cCCexp* (*Bool b*) = ⊥
| *cCCexp* (*scrut ? e1 : e2*) = (Λ *n*. (*fst* (*cCCexp scrut*·*0*) ⊔ *fst* (*cCCexp e1*·*n*) ⊔ *fst* (*cCCexp e2*·*n*),
snd (*cCCexp scrut*·*0*) ⊔ (*snd* (*cCCexp e1*·*n*) ⊔ *snd* (*cCCexp e2*·*n*)) ⊔ *ccProd* (*edom* (*fst* (*cCCexp scrut*·*0*))) (*edom* (*fst* (*cCCexp e1*·*n*)) ∪ *edom* (*fst* (*cCCexp e2*·*n*)))))
proof *goal-cases*
case *1*
show *?case*
unfolding *eqvt-def cCCexp-graph-aux-def*
apply *rule*
apply (*perm-simp*)
apply (*simp add*: *Abs-cfun-eqvt*)
done
next
case *3*
thus *?case* **by** (*metis Terms.exp-strong-exhaust*)
next
case *prems*: (*10 x e x' e'*)
from *prems*(*9*)
show *?case*

proof(*rule eqvt-lam-case*)
fix π :: *perm*
assume $*$: *supp* $(-\pi)$ $\sharp*$ (*fv* (*Lam* [*x*]. *e*) :: *var set*)
{
fix *n*
have *combined-restrict* (*fv* (*Lam* [*x*]. *e*)) (*fst* (*cCCexp-sumC* (π • *e*)·(*pred*·*n*)), *predCC* (*fv* (*Lam* [*x*]. *e*)) (Λ *a*. *snd*(*cCCexp-sumC* (π • *e*)·*a*))·*n*)
= *combined-restrict* (*fv* (*Lam* [*x*]. *e*)) ($-$ π • (*fst* (*cCCexp-sumC* (π • *e*)·(*pred*·*n*)), *predCC* (*fv* (*Lam* [*x*]. *e*)) (Λ *a*. *snd*(*cCCexp-sumC* (π • *e*)·*a*))·*n*))
by (*rule combined-restrict-perm*[*symmetric*, *OF* $*$]) *simp*
also have ... = *combined-restrict* (*fv* (*Lam* [*x*]. *e*)) (*fst* (*cCCexp-sumC e*·(*pred*·*n*)), *predCC* ($-$ π • *fv* (*Lam* [*x*]. *e*)) (Λ *a*. *snd*(*cCCexp-sumC e*·*a*))·*n*)
by (*perm-simp*, *simp add*: *eqvt-at-apply*[*OF prems*(*1*)] *pemute-minus-self Abs-cfun-eqvt*)
also have $-$ π • *fv* (*Lam* [*x*]. *e*) = (*fv* (*Lam* [*x*]. *e*) :: *var set*) **by** (*rule perm-supp-eq*[*OF* $*$])
also note *calculation*
}
thus (Λ *n*. *combined-restrict* (*fv* (*Lam* [*x*]. *e*)) (*fst* (*cCCexp-sumC* (π • *e*)·(*pred*·*n*)), *predCC* (*fv* (*Lam* [*x*]. *e*)) (Λ *a*. *snd*(*cCCexp-sumC* (π • *e*)·*a*))·*n*))
= (Λ *n*. *combined-restrict* (*fv* (*Lam* [*x*]. *e*)) (*fst* (*cCCexp-sumC e*·(*pred*·*n*)), *predCC* (*fv* (*Lam* [*x*]. *e*)) (Λ *a*. *snd*(*cCCexp-sumC e*·*a*))·*n*)) **by** *simp*
qed
next
case *prems*: (*19* Γ *body* Γ' *body'*)
from *prems*(*9*)
show *?case*
proof (*rule eqvt-let-case*)
fix π :: *perm*
assume $*$: *supp* $(-\pi)$ $\sharp*$ (*fv* (*Terms.Let* Γ *body*) :: *var set*)

{ fix *n*
have *combined-restrict* (*fv* (*Terms.Let* Γ *body*)) (*CoCallArityAnalysis.cccFix-choose cCCexp-sumC* (π • Γ)·(*cCCexp-sumC* (π • *body*)·*n*))
= *combined-restrict* (*fv* (*Terms.Let* Γ *body*)) ($-$ π • (*CoCallArityAnalysis.cccFix-choose cCCexp-sumC* (π • Γ)·(*cCCexp-sumC* (π • *body*)·*n*)))
by (*rule combined-restrict-perm*[*OF* $*$, *symmetric*]) *simp*
also have $-$ π • (*CoCallArityAnalysis.cccFix-choose cCCexp-sumC* (π • Γ)·(*cCCexp-sumC* (π • *body*)·*n*)) =
CoCallArityAnalysis.cccFix-choose ($-$ π • *cCCexp-sumC*) Γ·(($-$ π • *cCCexp-sumC*) *body*·*n*)
by (*simp add*: *pemute-minus-self*)
also have *CoCallArityAnalysis.cccFix-choose* ($-$ π • *cCCexp-sumC*) Γ = *CoCallArityAnalysis.cccFix-choose cCCexp-sumC* Γ
by (*rule cccFix-choose-cong*[*OF eqvt-at-apply*[*OF prems*(*1*)] *refl*])
also have ($-$ π • *cCCexp-sumC*) *body* = *cCCexp-sumC body*
by (*rule eqvt-at-apply*[*OF prems*(*2*)])
also note *calculation*
}
thus (Λ *n*. *combined-restrict* (*fv* (*Terms.Let* Γ *body*)) (*CoCallArityAnalysis.cccFix-choose*

cCCexp-sumC $(\pi \cdot \Gamma)\cdot(cCCexp\text{-}sumC\ (\pi \cdot body)\cdot n))) =$
$(\Lambda\ n.\ combined\text{-}restrict\ (fv\ (Terms.Let\ \Gamma\ body))\ (CoCallArityAnalysis.cccFix\text{-}choose$ *cCCexp-sumC* $\Gamma\cdot(cCCexp\text{-}sumC\ body\cdot n)))$ **by** (*simp only*:)
qed
qed *auto*

nominal-termination (*eqvt*) **by** *lexicographic-order*

locale *CoCallAnalysisImpl*
begin
sublocale *CoCallArityAnalysis cCCexp*.
sublocale *ArityAnalysis Aexp*.

abbreviation *Aexp-syn''* $(\mathcal{A}_{\_})$ **where** $\mathcal{A}_a\ e \equiv Aexp\ e\cdot a$
abbreviation *Aexp-bot-syn''* $(\mathcal{A}^{\perp}{}_{\_})$ **where** $\mathcal{A}^{\perp}{}_a\ e \equiv fup\cdot(Aexp\ e)\cdot a$

abbreviation *ccExp-syn''* $(\mathcal{G}_{\_})$ **where** $\mathcal{G}_a\ e \equiv CCexp\ e\cdot a$
abbreviation *ccExp-bot-syn''* $(\mathcal{G}^{\perp}{}_{\_})$ **where** $\mathcal{G}^{\perp}{}_a\ e \equiv fup\cdot(CCexp\ e)\cdot a$

lemma *cCCexp-eq*[*simp*]:
$cCCexp\ (Var\ x)\cdot n = \quad (esing\ x \cdot (up \cdot n), \quad \perp)$
$cCCexp\ (Lam\ [x].\ e)\cdot n = combined\text{-}restrict\ (fv\ (Lam\ [x].\ e))\ (fst\ (cCCexp\ e\cdot(pred\cdot n)), predCC\ (fv\ (Lam\ [x].\ e))\ (\Lambda\ a.\ snd(cCCexp\ e\cdot a))\cdot n)$
$cCCexp\ (App\ e\ x)\cdot n = \quad (fst\ (cCCexp\ e\cdot(inc\cdot n)) \sqcup (esing\ x \cdot (up\cdot 0)), \quad snd\ (cCCexp\ e\cdot(inc\cdot n)) \sqcup ccProd\ \{x\}\ (insert\ x\ (fv\ e)))$
$cCCexp\ (Let\ \Gamma\ e)\cdot n = \quad combined\text{-}restrict\ (fv\ (Let\ \Gamma\ e))\ (CoCallArityAnalysis.cccFix\text{-}choose\ cCCexp\ \Gamma \cdot (cCCexp\ e\cdot n))$
$cCCexp\ (Bool\ b)\cdot n = \perp$
$cCCexp\ (scrut\ ?\ e1\ :\ e2)\cdot n = (fst\ (cCCexp\ scrut\cdot 0) \sqcup fst\ (cCCexp\ e1\cdot n) \sqcup fst\ (cCCexp\ e2\cdot n),$
$snd\ (cCCexp\ scrut\cdot 0) \sqcup (snd\ (cCCexp\ e1\cdot n) \sqcup snd\ (cCCexp\ e2\cdot n)) \sqcup ccProd\ (edom\ (fst\ (cCCexp\ scrut\cdot 0)))\ (edom\ (fst\ (cCCexp\ e1\cdot n)) \cup edom\ (fst\ (cCCexp\ e2\cdot n))))$
by (*simp-all*)
declare *cCCexp.simps*[*simp del*]

lemma *Aexp-pre-simps*:
$\mathcal{A}_a\ (Var\ x) = esing\ x\cdot(up\cdot a)$
$\mathcal{A}_a\ (Lam\ [x].\ e) = Aexp\ e\cdot(pred\cdot a)\ f|`\ fv\ (Lam\ [x].\ e)$
$\mathcal{A}_a\ (App\ e\ x) = Aexp\ e\cdot(inc\cdot a) \sqcup esing\ x\cdot(up\cdot 0)$
$\neg\ nonrec\ \Gamma \Longrightarrow$
$\mathcal{A}_a\ (Let\ \Gamma\ e) = (Afix\ \Gamma\cdot(\mathcal{A}_a\ e \sqcup (\lambda\text{-}.up\cdot 0)\ f|`\ thunks\ \Gamma))\ f|`\ (fv\ (Let\ \Gamma\ e))$
$x \notin fv\ e \Longrightarrow$
$\mathcal{A}_a\ (let\ x\ be\ e\ in\ exp) =$
$(fup\cdot(Aexp\ e)\cdot(ABind\text{-}nonrec\ x\ e\cdot(\mathcal{A}_a\ exp,\ CCexp\ exp\cdot a)) \sqcup \mathcal{A}_a\ exp)$
$f|`\ (fv\ (let\ x\ be\ e\ in\ exp))$
$\mathcal{A}_a\ (Bool\ b) = \perp$
$\mathcal{A}_a\ (scrut\ ?\ e1\ :\ e2) = \mathcal{A}_0\ scrut \sqcup \mathcal{A}_a\ e1 \sqcup \mathcal{A}_a\ e2$

by (*simp add*: *cccFix-eq Aexp-eq fup-Aexp-eq CCexp-eq fup-CCexp-eq*)+

lemma *CCexp-pre-simps*:
$CCexp\ (Var\ x) \cdot n = \bot$
$CCexp\ (Lam\ [x].\ e) \cdot n = predCC\ (fv\ (Lam\ [x].\ e))\ (CCexp\ e) \cdot n$
$CCexp\ (App\ e\ x) \cdot n = CCexp\ e \cdot (inc \cdot n) \sqcup ccProd\ \{x\}\ (insert\ x\ (fv\ e))$
$\neg\ nonrec\ \Gamma \Longrightarrow$
$CCexp\ (Let\ \Gamma\ e) \cdot n = cc\text{-}restr\ (fv\ (Let\ \Gamma\ e))$
$(CCfix\ \Gamma \cdot (Afix\ \Gamma \cdot (Aexp\ e \cdot n \sqcup (\lambda\text{-}.up \cdot 0)\ f|`\ thunks\ \Gamma),\ CCexp\ e \cdot n))$
$x \notin fv\ e \Longrightarrow CCexp\ (let\ x\ be\ e\ in\ exp) \cdot n =$
$cc\text{-}restr\ (fv\ (let\ x\ be\ e\ in\ exp))$
$(ccBind\ x\ e\ \cdot (Aheap\text{-}nonrec\ x\ e \cdot (Aexp\ exp \cdot n,\ CCexp\ exp \cdot n),\ CCexp\ exp \cdot n)$
$\sqcup\ ccProd\ (fv\ e)\ (ccNeighbors\ x\ (CCexp\ exp \cdot n) - (if\ isVal\ e\ then\ \{\}\ else\ \{x\})) \sqcup CCexp$
$exp \cdot n)$
$CCexp\ (Bool\ b) \cdot n = \bot$
$CCexp\ (scrut\ ?\ e1\ :\ e2) \cdot n =$
$CCexp\ scrut \cdot 0\ \sqcup$
$(CCexp\ e1 \cdot n \sqcup CCexp\ e2 \cdot n)\ \sqcup$
$ccProd\ (edom\ (Aexp\ scrut \cdot 0))\ (edom\ (Aexp\ e1 \cdot n) \cup edom\ (Aexp\ e2 \cdot n))$
by (*simp add*: *cccFix-eq Aexp-eq fup-Aexp-eq CCexp-eq fup-CCexp-eq predCC-eq*)+

lemma
shows *ccField-CCexp*: $ccField\ (CCexp\ e \cdot a) \subseteq fv\ e$ **and** *Aexp-edom'*: $edom\ (\mathcal{A}_a\ e) \subseteq fv\ e$
apply (*induction e arbitrary*: *a rule*: *exp-induct-rec*)
apply (*auto simp add*: *CCexp-pre-simps predCC-eq Aexp-pre-simps dest*!: *set-mp*[*OF ccField-cc-restr*] *set-mp*[*OF ccField-ccProd-subset*])
apply *fastforce*+
done

lemma *cc-restr-CCexp*[*simp*]:
$cc\text{-}restr\ (fv\ e)\ (CCexp\ e \cdot a) = CCexp\ e \cdot a$
by (*rule cc-restr-noop*[*OF ccField-CCexp*])

lemma *ccField-fup-CCexp*:
$ccField\ (fup \cdot (CCexp\ e) \cdot n) \subseteq fv\ e$
by (*cases n*) (*auto dest*: *set-mp*[*OF ccField-CCexp*])

lemma *cc-restr-fup-ccExp-useless*[*simp*]: $cc\text{-}restr\ (fv\ e)\ (fup \cdot (CCexp\ e) \cdot n) = fup \cdot (CCexp\ e) \cdot n$
by (*rule cc-restr-noop*[*OF ccField-fup-CCexp*])

sublocale *EdomArityAnalysis Aexp* **by** *standard* (*rule Aexp-edom'*)

lemma *CCexp-simps*[*simp*]:
$\mathcal{G}_a(Var\ x) = \bot$
$\mathcal{G}_0(Lam\ [x].\ e) = (fv\ (Lam\ [x].\ e))^2$
$\mathcal{G}_{inc \cdot a}(Lam\ [x].\ e) = cc\text{-}delete\ x\ (\mathcal{G}_a\ e)$
$\mathcal{G}_a\ (App\ e\ x) = \mathcal{G}_{inc \cdot a}\ e \sqcup \{x\}\ G{\times}insert\ x\ (fv\ e)$
$\neg\ nonrec\ \Gamma \Longrightarrow \mathcal{G}_a\ (Let\ \Gamma\ e) =$

$(CCfix\ \Gamma\cdot(Afix\ \Gamma\cdot(\mathcal{A}_a\ e \sqcup (\lambda\text{-}.up\cdot 0)\ f|\text{`}\ thunks\ \Gamma), \mathcal{G}_a\ e))\ G|\text{`}\ (-\ domA\ \Gamma)$
$x \notin fv\ e' \Longrightarrow \mathcal{G}_a\ (let\ x\ be\ e'\ in\ e) =$
$cc\text{-}delete\ x$
$(ccBind\ x\ e'\ \cdot(Aheap\text{-}nonrec\ x\ e'\cdot(\mathcal{A}_a\ e, \mathcal{G}_a\ e), \mathcal{G}_a\ e)$
$\sqcup\ fv\ e'\ G\times\ (ccNeighbors\ x\ (\mathcal{G}_a\ e) - (if\ isVal\ e'\ then\ \{\}\ else\ \{x\})) \sqcup \mathcal{G}_a\ e)$
$\mathcal{G}_a\ (Bool\ b) = \bot$
$\mathcal{G}_a\ (scrut\ ?\ e1 : e2) =$
$\mathcal{G}_0\ scrut \sqcup (\mathcal{G}_a\ e1 \sqcup \mathcal{G}_a\ e2) \sqcup$
$edom\ (\mathcal{A}_0\ scrut)\ G\times\ (edom\ (\mathcal{A}_a\ e1) \cup edom\ (\mathcal{A}_a\ e2))$
by (*auto simp add*: *CCexp-pre-simps Diff-eq cc-restr-cc-restr*[*symmetric*] *predCC-eq*
simp del: *cc-restr-cc-restr cc-restr-join*
intro!: *cc-restr-noop*
dest!: *set-mp*[*OF ccField-cc-delete*] *set-mp*[*OF ccField-cc-restr*] *set-mp*[*OF ccField-CCexp*]
set-mp[*OF ccField-CCfix*] *set-mp*[*OF ccField-ccBind*] *set-mp*[*OF ccField-ccProd-subset*]
elem-to-ccField
)

definition *Aheap* **where**
$Aheap\ \Gamma\ e = (\Lambda\ a.\ if\ nonrec\ \Gamma\ then\ (case\text{-}prod\ Aheap\text{-}nonrec\ (hd\ \Gamma))\cdot(Aexp\ e\cdot a,\ CCexp\ e\cdot a)$
$else\ \ (Afix\ \Gamma \cdot (Aexp\ e\cdot a \sqcup (\lambda\text{-}.up\cdot 0)\ f|\text{`}\ thunks\ \Gamma))\ f|\text{`}\ domA\ \Gamma)$

lemma *Aheap-simp1*[*simp*]:
$\neg\ nonrec\ \Gamma \Longrightarrow Aheap\ \Gamma\ e\ \cdot a = (Afix\ \Gamma \cdot (Aexp\ e\cdot a \sqcup (\lambda\text{-}.up\cdot 0)\ f|\text{`}\ thunks\ \Gamma))\ f|\text{`}\ domA\ \Gamma$
unfolding *Aheap-def* **by** *simp*

lemma *Aheap-simp2*[*simp*]:
$x \notin fv\ e' \Longrightarrow Aheap\ [(x,e')]\ e\ \cdot a = Aheap\text{-}nonrec\ x\ e'\cdot(Aexp\ e\cdot a,\ CCexp\ e\cdot a)$
unfolding *Aheap-def* **by** (*simp add*: *nonrec-def*)

lemma *Aheap-eqvt'*[*eqvt*]:
$\pi \bullet (Aheap\ \Gamma\ e) = Aheap\ (\pi \bullet \Gamma)\ (\pi \bullet e)$
apply (*rule cfun-eqvtI*)
apply (*cases nonrec* π *rule*: *eqvt-cases*[**where** $x = \Gamma$])
apply *simp*
apply (*erule nonrecE*)
apply *simp*
apply (*erule nonrecE*)
apply *simp*
apply (*perm-simp*, *rule*)
apply *simp*
apply (*perm-simp*, *rule*)
done

sublocale *ArityAnalysisHeap Aheap*.

sublocale *ArityAnalysisHeapEqvt Aheap*
proof
fix π **show** $\pi \bullet Aheap = Aheap$
by *perm-simp rule*

qed

lemma *Aexp-lam-simp*: *Aexp* (*Lam* [*x*]. *e*) · *n* = *env-delete x* (*Aexp e* · (*pred* · *n*))
proof−
have *Aexp* (*Lam* [*x*]. *e*) · *n* = *Aexp e*·(*pred*·*n*) *f*|‘ (*fv e* − {*x*}) **by** (*simp add*: *Aexp-pre-simps*)
also have ... = *env-delete x* (*Aexp e*·(*pred*·*n*)) *f*|‘ (*fv e* − {*x*}) **by** *simp*
also have . . . = *env-delete x* (*Aexp e*·(*pred*·*n*))
by (*rule env-restr-useless*) (*auto dest*: *set-mp*[*OF Aexp-edom*])
finally show *?thesis***.**
qed

lemma *Aexp-Let-simp1*:
¬ *nonrec* Γ =⇒ $\mathcal{A}_a$ (*Let* Γ *e*) = (*Afix* Γ·($\mathcal{A}_a$ *e* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ)) *f*|‘ (− *domA* Γ)
unfolding *Aexp-pre-simps*
by (*rule env-restr-cong*) (*auto simp add*: *dest*!: *set-mp*[*OF Afix-edom*] *set-mp*[*OF Aexp-edom*] *set-mp*[*OF thunks-domA*])

lemma *Aexp-Let-simp2*:
x ∉ *fv e* =⇒ $\mathcal{A}_a$(*let x be e in exp*) = *env-delete x* ($\mathcal{A}^\perp$ *ABind-nonrec x e* · ($\mathcal{A}_a$ *exp*, *CCexp exp*·*a*) *e* ⊔ $\mathcal{A}_a$ *exp*)
unfolding *Aexp-pre-simps env-delete-restr*
by (*rule env-restr-cong*) (*auto dest*!: *set-mp*[*OF fup-Aexp-edom*] *set-mp*[*OF Aexp-edom*])

lemma *Aexp-simps*[*simp*]:
$\mathcal{A}_a$(*Var x*) = *esing x*·(*up*·*a*)
$\mathcal{A}_a$(*Lam* [*x*]. *e*) = *env-delete x* ($\mathcal{A}_{pred \cdot a}$ *e*)
$\mathcal{A}_a$(*App e x*) = *Aexp e*·(*inc*·*a*) ⊔ *esing x*·(*up*·*0*)
¬ *nonrec* Γ =⇒ $\mathcal{A}_a$(*Let* Γ *e*) =
(*Afix* Γ·($\mathcal{A}_a$ *e* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ)) *f*|‘ (− *domA* Γ)
x ∉ *fv e*′ =⇒ $\mathcal{A}_a$(*let x be e*′ *in e*) =
env-delete x ($\mathcal{A}^\perp$ *ABind-nonrec x e*′·($\mathcal{A}_a$ *e*, $\mathcal{G}_a$ *e*) *e*′ ⊔ $\mathcal{A}_a$ *e*)
$\mathcal{A}_a$(*Bool b*) = ⊥
$\mathcal{A}_a$(*scrut ? e1* : *e2*) = $\mathcal{A}_0$ *scrut* ⊔ $\mathcal{A}_a$ *e1* ⊔ $\mathcal{A}_a$ *e2*
by (*simp-all add*: *Aexp-lam-simp Aexp-Let-simp1 Aexp-Let-simp2*, *simp-all add*: *Aexp-pre-simps*)

end

end

76 CallArityEnd2End.tex

theory *CallArityEnd2End*
imports *ArityTransform CoCallAnalysisImpl*
begin

locale *CallArityEnd2End*
begin
sublocale *CoCallAnalysisImpl*.

lemma *fresh-var-eqE*[*elim-format*]: *fresh-var e* = *x* $\Longrightarrow$ *x* $\notin$ *fv e*
  by (*metis fresh-var-not-free*)

lemma *example1*:
  fixes *e* :: *exp*
  fixes *f g x y z* :: *var*
  assumes *Aexp-e*: $\bigwedge$*a*. *Aexp e*·*a* = *esing x*·(*up*·*a*) ⊔ *esing y*·(*up*·*a*)
  assumes *ccExp-e*: $\bigwedge$*a*. *CCexp e*·*a* = ⊥
  assumes [*simp*]: *transform 1 e* = *e*
  assumes *isVal e*
  assumes *disj*: *y* ≠ *f y* ≠ *g x* ≠ *y z* ≠ *f z* ≠ *g y* ≠ *x*
  assumes *fresh*: *atom z* ♯ *e*
  shows *transform 1* (*let y be App* (*Var f*) *g in* (*let x be e in* (*Var x*))) =
        let y be (*Lam* [*z*]. *App* (*App* (*Var f*) *g*) *z*) *in* (*let x be* (*Lam* [*z*]. *App e z*) *in* (*Var x*))
proof−
  from *arg-cong*[**where** *f* = *edom*, *OF Aexp-e*]
  have *x* ∈ *fv e* **by** *simp* (*metis Aexp-edom′ insert-subset*)
  hence [*simp*]: ¬ *nonrec* [(*x*,*e*)]
    by (*simp add*: *nonrec-def*)

  from ‹*isVal e*›
  have [*simp*]: *thunks* [(*x*, *e*)] = {}
    by (*simp add*: *thunks-Cons*)

  have [*simp*]: *CCfix* [(*x*, *e*)]·(*esing x*·(*up*·*1*) ⊔ *esing y*·(*up*·*1*), ⊥) = ⊥
    unfolding *CCfix-def*
    apply (*simp add*: *fix-bottom-iff ccBindsExtra-simp*)
    apply (*simp add*: *ccBind-eq disj ccExp-e*)
    done

  have [*simp*]: *Afix* [(*x*, *e*)]·(*esing x*·(*up*·*1*)) = *esing x*·(*up*·*1*) ⊔ *esing y*·(*up*·*1*)
    unfolding *Afix-def*
    apply *simp*
    apply (*rule fix-eqI*)
    apply (*simp add*: *disj Aexp-e*)
    apply (*case-tac z x*)
    apply (*auto simp add*: *disj Aexp-e*)
    done

  have [*simp*]: *Aheap* [(*y*, *App* (*Var f*) *g*)] (*let x be e in Var x*)·*1* = *esing y*·((*Aexp* (*let x be e in Var x*)·*1*) *y*)
    by (*auto simp add*: *Aheap-nonrec-simp ABind-nonrec-eq pure-fresh fresh-at-base disj*)

  have [*simp*]: (*Aexp* (*let x be e in Var x*)·*1*) = *esing y*·(*up*·*1*)
    by (*simp add*: *env-restr-join disj*)

have [*simp*]: *Aheap* [(*x*, *e*)] (*Var x*)·*1* = *esing x*·(*up*·*1*)
  by (*simp add*: *env-restr-join disj*)

have *1*: *1* = *inc*·*0* **apply** (*simp add*: *inc-def*) **apply** *transfer* **apply** *simp* **done**

have [*simp*]: *Aeta-expand 1* (*App* (*Var f*) *g*) = (*Lam* [*z*]. *App* (*App* (*Var f*) *g*) *z*)
  apply (*simp add*: *1 del*: *exp-assn.eq-iff*)
  apply (*subst change-Lam-Variable*[*of z fresh-var* (*App* (*Var f*) *g*)])
  apply (*auto simp add*: *fresh-Pair fresh-at-base pure-fresh disj intro*!: *flip-fresh-fresh elim*!: *fresh-var-eqE*)
  done

have [*simp*]: *Aeta-expand 1 e* = (*Lam* [*z*]. *App e z*)
  apply (*simp add*: *1 del*: *exp-assn.eq-iff*)
  apply (*subst change-Lam-Variable*[*of z fresh-var e*])
  apply (*auto simp add*: *fresh-Pair fresh-at-base pure-fresh disj fresh intro*!: *flip-fresh-fresh elim*!: *fresh-var-eqE*)
  done

show *?thesis*
  by (*simp del*: *Let-eq-iff add*: *map-transform-Cons map-transform-Nil disj*[*symmetric*])
qed

end
end

77 SestoftGC.tex

theory *SestoftGC*
imports *Sestoft*
begin

inductive *gc-step* :: *conf* ⇒ *conf* ⇒ *bool* (**infix** $\Rightarrow_G$ *50*) **where**
  normal: $c \Rightarrow c' \Longrightarrow c \Rightarrow_G c'$
| *dropUpd*: (Γ, *e*, *Upd x* # *S*) $\Rightarrow_G$ (Γ, *e*, *S* @ [*Dummy x*])

lemmas *gc-step-intros*[*intro*] =
  normal[*OF step.intros*(*1*)] *normal*[*OF step.intros*(*2*)] *normal*[*OF step.intros*(*3*)]
  normal[*OF step.intros*(*4*)] *normal*[*OF step.intros*(*5*)] *dropUpd*

abbreviation *gc-steps* (**infix** $\Rightarrow_G{}^*$ *50*) **where** *gc-steps* ≡ *gc-step***
lemmas *converse-rtranclp-into-rtranclp*[*of gc-step*, *OF - r-into-rtranclp*, *trans*]

lemma *var-onceI*:
  assumes *map-of* Γ *x* = *Some e*
  shows (Γ, *Var x*, *S*) $\Rightarrow_G{}^*$ (*delete x* Γ, *e* , *S*@[*Dummy x*])

proof−
  from *assms*
  have $(\Gamma,\ \mathit{Var}\ x,\ S) \Rightarrow_G (\mathit{delete}\ x\ \Gamma,\ e\ ,\ \mathit{Upd}\ x\ \#\ S)$..
  moreover
  have $\ldots \Rightarrow_G\ (\mathit{delete}\ x\ \Gamma,\ e,\ S@[\mathit{Dummy}\ x])$..
  ultimately
  show *?thesis* **by** (*rule converse-rtranclp-into-rtranclp*[*OF - r-into-rtranclp*])
qed

lemma *normal-trans*: $c \Rightarrow^* c' \Longrightarrow c \Rightarrow_G{}^* c'$
  by (*induction rule*:*rtranclp-induct*)
    (*simp*, *metis normal rtranclp.rtrancl-into-rtrancl*)

fun *to-gc-conf* :: *var list* ⇒ *conf* ⇒ *conf*
  where *to-gc-conf r* (Γ, *e*, *S*) = (*restrictA* (− *set r*) Γ, *e*, *restr-stack* (− *set r*) *S* @ (*map Dummy* (*rev r*)))

lemma *restr-stack-map-Dummy*[*simp*]: *restr-stack V* (*map Dummy l*) = *map Dummy l*
  by (*induction l*) *auto*

lemma *restr-stack-append*[*simp*]: *restr-stack V* ($l@l'$) = *restr-stack V l* @ *restr-stack V* l'
  by (*induction l rule*: *restr-stack.induct*) *auto*

lemma *to-gc-conf-append*[*simp*]:
  to-gc-conf ($r@r'$) *c* = *to-gc-conf r* (*to-gc-conf* r' *c*)
  by (*cases c*) *auto*

lemma *to-gc-conf-eqE*[*elim*!]:
  assumes *to-gc-conf r c* = (Γ, *e*, *S*)
  obtains Γ' S' **where** *c* = (Γ', *e*, S') **and** Γ = *restrictA* (− *set r*) Γ' **and** *S* = *restr-stack* (− *set r*) S' @ *map Dummy* (*rev r*)
  using *assms* **by** (*cases c*) *auto*

fun *safe-hd* :: *'a list* ⇒ *'a option*
  where *safe-hd* (*x*#-) = *Some x*
    | *safe-hd* [] = *None*

lemma *safe-hd-None*[*simp*]: *safe-hd xs* = *None* ⟷ *xs* = []
  by (*cases xs*) *auto*

abbreviation *r-ok* :: *var list* ⇒ *conf* ⇒ *bool*
  where *r-ok r c* ≡ *set r* ⊆ *domA* (*fst c*) ∪ *upds* (*snd* (*snd c*))

lemma *subset-bound-invariant*:
  invariant step (*r-ok r*)
proof
  fix *x y*

assume $x \Rightarrow y$ **and** *r-ok r x* **thus** *r-ok r y*
by (*induction*) *auto*
qed

lemma *safe-hd-restr-stack*[*simp*]:
Some a = *safe-hd* (*restr-stack V* (*a* # *S*)) $\longleftrightarrow$ *restr-stack V* (*a* # *S*) = *a* # *restr-stack V S*
apply (*cases a*)
apply (*auto split*: *if-splits*)
apply (*thin-tac P a* **for** *P*)
apply (*induction S rule*: *restr-stack.induct*)
apply (*auto split*: *if-splits*)
done

lemma *sestoftUnGCStack*:
assumes *heap-upds-ok* (Γ, *S*)
obtains Γ' S' **where**
(Γ, *e*, *S*) $\Rightarrow^*$ (Γ', *e*, S')
to-gc-conf r (Γ, *e*, *S*) = *to-gc-conf r* (Γ', *e*, S')
$\neg$ *isVal e* $\vee$ *safe-hd* S' = *safe-hd* (*restr-stack* (− *set r*) S')
proof−
show *?thesis*
proof(*cases isVal e*)
case *False*
thus *?thesis* **using** *assms* **by** −(*rule that*, *auto*)
next
case *True*
from *that assms*
show *?thesis*
apply (*atomize-elim*)
proof(*induction S arbitrary*: Γ)
case *Nil* **thus** *?case* **by** (*fastforce*)
next
case (*Cons s S*)
show *?case*
proof(*cases Some s* = *safe-hd* (*restr-stack* (− *set r*) (*s*#*S*)))
case *True*
thus *?thesis*
using ‹*isVal e*› ‹*heap-upds-ok* (Γ, *s* # *S*)›
apply *auto*
apply (*intro exI conjI*)
apply (*rule rtranclp.intros*(*1*))
apply *auto*
done
next
case *False*
then obtain *x* **where** [*simp*]: *s* = *Upd x* **and** [*simp*]: $x \in$ *set r*
by(*cases s*) (*auto split*: *if-splits*)

from ‹*heap-upds-ok* (Γ, *s* # *S*)› ‹*s* = *Upd x*›
have [*simp*]: $x \notin$ *domA* Γ **and** *heap-upds-ok* ((*x*,*e*) # Γ, *S*)
by (*auto dest*: *heap-upds-okE*)

have (Γ, *e*, *s* # *S*) $\Rightarrow^*$ (Γ, *e*, *Upd x* # *S*) **unfolding** ‹*s* = -› **..**
also have ... $\Rightarrow$ ((*x*,*e*) # Γ, *e*, *S*) **by** (*rule step.var*$_2$[*OF* ‹$x \notin$ *domA* Γ› ‹*isVal e*›])
also
from *Cons.IH*[*OF* ‹*heap-upds-ok* ((*x*,*e*) # Γ, *S*)›]
obtain Γ′ *S*′ **where** ((*x*,*e*) # Γ, *e*, *S*) $\Rightarrow^*$ (Γ′, *e*, *S*′)
and *res*: *to-gc-conf r* ((*x*,*e*) # Γ, *e*, *S*) = *to-gc-conf r* (Γ′, *e*, *S*′)
($\neg$ *isVal e* $\vee$ *safe-hd S*′ = *safe-hd* (*restr-stack* (− *set r*) *S*′))
by *blast*
note *this*(*1*)
finally
have (Γ, *e*, *s* # *S*) $\Rightarrow^*$ (Γ′, *e*, *S*′).
thus *?thesis* **using** *res* **by** *auto*
qed
qed
qed
qed

lemma *perm-exI-trivial*:
P x x $\Longrightarrow \exists\ \pi.$ *P* ($\pi \cdot x$) *x*
by (*rule exI*[**where** *x* = *0*::*perm*]) *auto*

lemma *upds-list-restr-stack*[*simp*]:
upds-list (*restr-stack V S*) = *filter* (λ *x*. *x*$\in$*V*) (*upds-list S*)
by (*induction S rule*: *restr-stack.induct*) *auto*

lemma *heap-upd-ok-to-gc-conf*:
heap-upds-ok (Γ, *S*) $\Longrightarrow$ *to-gc-conf r* (Γ, *e*, *S*) = (Γ″, *e*″, *S*″) $\Longrightarrow$ *heap-upds-ok* (Γ″, *S*″)
by (*auto simp add*: *heap-upds-ok.simps*)

lemma *delete-restrictA-conv*:
delete x Γ = *restrictA* (−{*x*}) Γ
by (*induction* Γ) *auto*

lemma *sestoftUnGCstep*:
assumes *to-gc-conf r c* $\Rightarrow_G$ *d*
assumes *heap-upds-ok-conf c*
assumes *closed c*
and *r-ok r c*
shows $\exists$ *r*′ *c*′. *c* $\Rightarrow^*$ *c*′ $\wedge$ *d* = *to-gc-conf r*′ *c*′ $\wedge$ *r-ok r*′ *c*′
proof−
obtain Γ *e S* **where** *c* = (Γ, *e*, *S*) **by** (*cases c*) *auto*
with *assms*
have *heap-upds-ok* (Γ,*S*) **and** *closed* (Γ, *e*, *S*) **and** *r-ok r* (Γ, *e*, *S*) **by** *auto*
from *sestoftUnGCStack*[*OF this*(*1*)]
obtain Γ′ *S*′ **where**

$(\Gamma, e, S) \Rightarrow^* (\Gamma', e, S')$
and $*$: *to-gc-conf* r (Γ, e, S) = *to-gc-conf* r (Γ', e, S')
and *disj*: $\neg$ *isVal* e $\vee$ *safe-hd* S' = *safe-hd* (*restr-stack* ($-$ *set* r) S')
by *metis*

from *invariant-starE*[*OF* ‹- $\Rightarrow^*$ -› *heap-upds-ok-invariant*] ‹*heap-upds-ok* (Γ,S)›
have *heap-upds-ok* (Γ', S') **by** *auto*

from *invariant-starE*[*OF* ‹- $\Rightarrow^*$ -› *closed-invariant* ‹*closed* (Γ, e, S)›]
have *closed* (Γ', e, S') **by** *auto*

from *invariant-starE*[*OF* ‹- $\Rightarrow^*$ -› *subset-bound-invariant* ‹*r-ok* r (Γ, e, S)›]
have *r-ok* r (Γ', e, S') **by** *auto*

from *assms*(*1*)[*unfolded* ‹*c* =- › $*$]
have $\exists$ r' Γ'' e'' S''. $(\Gamma', e, S') \Rightarrow^* (\Gamma'', e'', S'')$ $\wedge$ d = *to-gc-conf* r' (Γ'', e'', S'') $\wedge$ *r-ok* r' (Γ'', e'', S'')
proof(*cases rule*: *gc-step.cases*)
case *normal*
hence $\exists$ Γ'' e'' S''. $(\Gamma', e, S') \Rightarrow (\Gamma'', e'', S'')$ $\wedge$ d = *to-gc-conf* r (Γ'', e'', S'')
proof(*cases rule*: *step.cases*)
case app_1
thus *?thesis*
apply *auto*
apply (*intro exI conjI*)
apply (*rule step.intros*)
apply *auto*
done
next
case (app_2 Γ y *ea* x S)
thus *?thesis*
using *disj*
apply (*cases* S')
apply *auto*
apply (*intro exI conjI*)
apply (*rule step.intros*)
apply *auto*
done
next
case var_1
thus *?thesis*
apply *auto*
apply (*intro exI conjI*)
apply (*rule step.intros*)
apply (*auto simp add*: *restr-delete-twist*)
done
next
case var_2
thus *?thesis*

using *disj*
apply (*cases* S')
apply *auto*
apply (*intro exI conjI*)
apply (*rule step.intros*)
apply (*auto split*: *if-splits dest*: *Upd-eq-restr-stackD2*)
done
next
case (let_1 Δ'' Γ'' S'' e')

from ‹*closed* (Γ', e, S')› let_1
have *closed* (Γ', *Let* Δ'' e', S') **by** *simp*

from *fresh-distinct*[*OF* $let_1(3)$] *fresh-distinct-fv*[*OF* $let_1(4)$]
have *domA* $\Delta'' \cap$ *domA* $\Gamma'' = \{\}$ **and** *domA* $\Delta'' \cap$ *upds* $S'' = \{\}$ **and** *domA* $\Delta'' \cap$ *dummies* $S'' = \{\}$
by (*auto dest*: *set-mp*[*OF ups-fv-subset*] *set-mp*[*OF dummies-fv-subset*])
moreover
from $let_1(1)$
have *domA* $\Gamma' \cup$ *upds* $S' \subseteq$ *domA* $\Gamma'' \cup$ *upds* $S'' \cup$ *dummies* S''
by *auto*
ultimately
have *disj*: *domA* $\Delta'' \cap$ *domA* $\Gamma' = \{\}$ *domA* $\Delta'' \cap$ *upds* $S' = \{\}$
by *auto*

from ‹*domA* $\Delta'' \cap$ *dummies* $S'' = \{\}$› $let_1(1)$
have *domA* $\Delta'' \cap$ *set* $r = \{\}$ **by** *auto*
hence [*simp*]: *restrictA* ($-$ *set* r) $\Delta'' = \Delta''$
by (*auto intro*: *restrictA-noop*)

from $let_1(1{-}3)$
show *?thesis*
apply *auto*
apply (*intro* *exI*[**where** $x = r$] *exI*[**where** $x = \Delta''$ @ Γ'] *exI*[**where** $x = S'$] *conjI*)
apply (*rule* let_1*-closed*[*OF* ‹*closed* (Γ', *Let* Δ'' e', S')› *disj*])
apply (*auto simp add*: *restrictA-append*)
done
next
case if_1
thus *?thesis*
apply *auto*
apply (*intro exI*[**where** $x = 0$::*perm*] *exI conjI*)
unfolding *permute-zero*
apply (*rule step.intros*)
apply (*auto*)
done
next
case if_2
thus *?thesis*

using *disj*
apply (*cases* S')
apply *auto*
apply (*intro exI exI conjI*)
apply (*rule step.if*$_2$[**where** b = *True*, *simplified*] *step.if*$_2$[**where** b = *False*, *simplified*])
apply (*auto split*: *if-splits dest*: *Upd-eq-restr-stackD2*)
apply (*intro exI conjI*)
apply (*rule step.if*$_2$[**where** b = *True*, *simplified*] *step.if*$_2$[**where** b = *False*, *simplified*])
apply (*auto split*: *if-splits dest*: *Upd-eq-restr-stackD2*)
done
qed
with *invariantE*[*OF subset-bound-invariant* - ‹*r-ok* r (Γ', e, S')›]
show *?thesis* **by** *blast*
next
case (*dropUpd* Γ'' e'' x S'')
from ‹*to-gc-conf* r (Γ', e, S') = (Γ'', e'', *Upd* x # S'')›
have $x \notin$ *set* r **by** (*auto dest*!: *arg-cong*[**where** f = *upds*])

from ‹*heap-upds-ok* (Γ', S')› **and** ‹*to-gc-conf* r (Γ', e, S') = (Γ'', e'', *Upd* x # S'')›
have *heap-upds-ok* (Γ'', *Upd* x # S'') **by** (*rule heap-upd-ok-to-gc-conf*)
hence [*simp*]: $x \notin$ *domA* Γ'' $x \notin$ *upds* S'' **by** (*auto dest*: *heap-upds-ok-upd*)

have *to-gc-conf* (x # r) (Γ', e, S') = *to-gc-conf* ([x]@ r) (Γ', e, S') **by** *simp*
also have ... = *to-gc-conf* [x] (*to-gc-conf* r (Γ', e, S')) **by** (*rule to-gc-conf-append*)
also have ... = *to-gc-conf* [x] (Γ'', e'', *Upd* x # S'') **unfolding** ‹*to-gc-conf* r (Γ', e, S') =
-›..
also have ... = (Γ'', e'', S''@[*Dummy* x]) **by** (*auto intro*: *restrictA-noop*)
also have ... = d **using** ‹ d= -› **by** *simp*
finally have *to-gc-conf* (x # r) (Γ', e, S') = d.
moreover
from ‹*to-gc-conf* r (Γ', e, S') = (Γ'', e'', *Upd* x # S'')›
have $x \in$ *upds* S' **by** (*auto dest*!: *arg-cong*[**where** f = *upds*])
with ‹*r-ok* r (Γ', e, S')›
have *r-ok* (x # r) (Γ', e, S') **by** *auto*
moreover
note ‹*to-gc-conf* r (Γ', e, S') = (Γ'', e'', *Upd* x # S'')›
ultimately
show *?thesis* **by** *fastforce*

qed
then obtain r' Γ'' e'' S''
where (Γ', e, S') $\Rightarrow^*$ (Γ'', e'', S'')
and d = *to-gc-conf* r' (Γ'', e'', S'')
and *r-ok* r' (Γ'', e'', S'')
by *metis*

from ‹(Γ, e, S) $\Rightarrow^*$ (Γ', e, S')› **and** ‹(Γ', e, S') $\Rightarrow^*$ (Γ'', e'', S'')›
have (Γ, e, S) $\Rightarrow^*$ (Γ'', e'', S'') **by** (*rule rtranclp-trans*)
with ‹d = -› ‹*r-ok* r' -›

  show *?thesis* **unfolding** ‹c = -› **by** *auto*
qed

lemma *sestoftUnGC*:
  assumes (*to-gc-conf* r c) $\Rightarrow_G{}^*$ d **and** *heap-upds-ok-conf* c **and** *closed* c **and** *r-ok* r c
  shows $\exists$ r' c'. $c \Rightarrow^* c' \wedge d =$ *to-gc-conf* r' c' $\wedge$ *r-ok* r' c'
using *assms*
proof(*induction rule*: *rtranclp-induct*)
  case *base*
  thus *?case* **by** *blast*
next
  case (*step* d' d'')
  then obtain r' c' **where** $c \Rightarrow^*$ c' **and** $d' =$ *to-gc-conf* r' c' **and** *r-ok* r' c' **by** *auto*

  from *invariant-starE*[*OF* ‹- $\Rightarrow^*$ -› *heap-upds-ok-invariant*] ‹*heap-upds-ok* -›
  have *heap-upds-ok-conf* c'.

  from *invariant-starE*[*OF* ‹- $\Rightarrow^*$ -› *closed-invariant*] ‹*closed* -›
  have *closed* c'.

  from *step* ‹d' = *to-gc-conf* r' c'›
  have *to-gc-conf* r' c' $\Rightarrow_G$ d'' **by** *simp*
  from *this* ‹*heap-upds-ok-conf* c'› ‹*closed* c'› ‹*r-ok* r' c'›
  have $\exists$ r'' c''. $c' \Rightarrow^* c'' \wedge d'' =$ *to-gc-conf* r'' c'' $\wedge$ *r-ok* r'' c''
    by (*rule sestoftUnGCstep*)
  then obtain r'' c'' **where** $c' \Rightarrow^* c''$ **and** d'' = *to-gc-conf* r'' c'' **and** *r-ok* r'' c'' **by** *auto*

  from ‹$c' \Rightarrow^*$ c''› ‹$c \Rightarrow^*$ c'›
  have $c \Rightarrow^* c''$ **by** *auto*
  with ‹d'' = -› ‹*r-ok* r'' c''›
  show *?case* **by** *blast*
qed

lemma *dummies-unchanged-invariant*:
  invariant step (λ (Γ, e, S) . *dummies* S = V) (**is** *invariant* - *?I*)
proof
  fix c c'
  assume $c \Rightarrow c'$ **and** *?I* c
  thus *?I* c' **by** (*induction*) *auto*
qed

lemma *sestoftUnGC'*:
  assumes ([], e, []) $\Rightarrow_G{}^*$ (Γ, e', *map Dummy* r)
  assumes *isVal* e'
  assumes *fv* e = ({}::*var set*)
  shows $\exists$ Γ''. ([], e, []) $\Rightarrow^*$ (Γ'', e', []) $\wedge$ Γ = *restrictA* ($-$ *set* r) Γ'' $\wedge$ *set* $r \subseteq$ *domA* Γ''
proof$-$
 from *sestoftUnGC*[**where** r = [] **and** c = ([], e, []), *simplified*, *OF assms*(*1*,*3*)]

obtain r' Γ' S'
where ([], e, []) $\Rightarrow^*$ (Γ', e', S')
and Γ = *restrictA* (− *set* r') Γ'
and *map Dummy r* = *restr-stack* (− *set* r') S' @ *map Dummy* (*rev* r')
and *r-ok* r' (Γ', e', S')
by *auto*

from *invariant-starE*[*OF* ‹([], e, []) $\Rightarrow^*$ (Γ', e', S')› *dummies-unchanged-invariant*]
have *dummies* S' = {} **by** *auto*
with ‹*map Dummy r* = *restr-stack* (− *set* r') S' @ *map Dummy* (*rev* r')›
have *restr-stack* (− *set* r') S' = [] **and** [*simp*]: *r* = *rev* r'
by (*induction* S' *rule*: *restr-stack.induct*) (*auto split*: *if-splits*)

from *invariant-starE*[*OF* ‹- $\Rightarrow^*$ -› *heap-upds-ok-invariant*]
have *heap-upds-ok* (Γ', S') **by** *auto*

from ‹*isVal* e'› *sestoftUnGCStack*[**where** e = e', *OF* ‹*heap-upds-ok* (Γ', S')›]
obtain Γ'' S''
where (Γ', e', S') $\Rightarrow^*$ (Γ'', e', S'')
and *to-gc-conf r* (Γ', e', S') = *to-gc-conf r* (Γ'', e', S'')
and *safe-hd* S'' = *safe-hd* (*restr-stack* (− *set r*) S'')
by *metis*

from *this* (*2*,*3*) ‹*restr-stack* (− *set* r') S' = []›
have S'' = [] **by** *auto*

from ‹([], e, []) $\Rightarrow^*$ (Γ', e', S')› **and** ‹(Γ', e', S') $\Rightarrow^*$ (Γ'', e', S'')› **and** ‹S'' = []›
have ([], e, []) $\Rightarrow^*$ (Γ'', e', []) **by** *auto*
moreover
have Γ = *restrictA* (− *set r*) Γ'' **using** ‹*to-gc-conf r* - = -› ‹Γ = -› **by** *auto*
moreover
from *invariant-starE*[*OF* ‹(Γ', e', S') $\Rightarrow^*$ (Γ'', e', S'')› *subset-bound-invariant* ‹*r-ok* r' (Γ', e', S')›]
have *set r* $\subseteq$ *domA* Γ'' **using** ‹S'' = []› **by** *auto*
ultimately
show *?thesis* **by** *blast*
qed

end

78 CardArityTransformSafe.tex

theory *CardArityTransformSafe*
imports *ArityTransform CardinalityAnalysisSpec AbstractTransform Sestoft SestoftGC ArityEtaExpansionSafe ArityAnalysisStack ArityConsistent*
begin

context *CardinalityPrognosisSafe*

begin
  sublocale *AbstractTransformBoundSubst*
    λ *a* . *inc*·*a*
    λ *a* . *pred*·*a*
    λ Δ *e a* . (*a*, *Aheap* Δ *e*·*a*)
    fst
    snd
    λ -. *0*
    Aeta-expand
    snd
  apply *standard*
  apply (*simp add*: *Aheap-subst*)
  apply (*rule subst-Aeta-expand*)
  done

  abbreviation *ccTransform* **where** *ccTransform* $\equiv$ *transform*

  lemma *supp-transform*: *supp* (*transform a e*) $\subseteq$ *supp e*
    by (*induction rule*: *transform.induct*)
      (*auto simp add*: *exp-assn.supp Let-supp dest*!: *set-mp*[*OF supp-map-transform*] *set-mp*[*OF supp-map-transform-step*])
  interpretation *supp-bounded-transform transform*
    by *standard* (*auto simp add*: *fresh-def supp-transform*)

  type-synonym *tstate* = (*AEnv* $\times$ (*var* $\Rightarrow$ *two*) $\times$ *Arity* $\times$ *Arity list* $\times$ *var list*)

  fun *transform-alts* :: *Arity list* $\Rightarrow$ *stack* $\Rightarrow$ *stack*
    where
      transform-alts - [] = []
    | *transform-alts* (*a*#*as*) (*Alts e1 e2* # *S*) = (*Alts* (*ccTransform a e1*) (*ccTransform a e2*)) # *transform-alts as S*
    | *transform-alts as* (*x* # *S*) = *x* # *transform-alts as S*

  lemma *transform-alts-Nil*[*simp*]: *transform-alts* [] *S* = *S*
    by (*induction S*) *auto*

  lemma *Astack-transform-alts*[*simp*]:
    Astack (*transform-alts as S*) = *Astack S*
   by (*induction rule*: *transform-alts.induct*) *auto*

  lemma *fresh-star-transform-alts*[*intro*]: *a* $\sharp$* *S* $\Longrightarrow$ *a* $\sharp$* *transform-alts as S*
   by (*induction as S rule*: *transform-alts.induct*) (*auto simp add*: *fresh-star-Cons*)

  fun *a-transform* :: *astate* $\Rightarrow$ *conf* $\Rightarrow$ *conf*
  where *a-transform* (*ae*, *a*, *as*) (Γ, *e*, *S*) =
    (*map-transform Aeta-expand ae* (*map-transform ccTransform ae* Γ),
     ccTransform a e,
     transform-alts as S)

fun *restr-conf* :: *var set* ⇒ *conf* ⇒ *conf*
where *restr-conf* *V* (Γ, *e*, *S*) = (*restrictA* *V* Γ, *e*, *restr-stack* *V* *S*)

fun *add-dummies-conf* :: *var list* ⇒ *conf* ⇒ *conf*
where *add-dummies-conf* *l* (Γ, *e*, *S*) = (Γ, *e*, *S* @ *map Dummy* (*rev l*))

fun *conf-transform* :: *tstate* ⇒ *conf* ⇒ *conf*
where *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) *c* = *add-dummies-conf* *r* ((*a-transform* (*ae*, *a*, *as*) (*restr-conf* (− *set r*) *c*)))

inductive *consistent* :: *tstate* ⇒ *conf* ⇒ *bool* **where**
consistentI[*intro*!]:
a-consistent (*ae*, *a*, *as*) (*restr-conf* (− *set r*) (Γ, *e*, *S*))
⟹ *edom ae* = *edom ce*
⟹ *prognosis ae as a* (Γ, *e*, *S*) ⊑ *ce*
⟹ (⋀ *x*. *x* ∈ *thunks* Γ ⟹ *many* ⊑ *ce x* ⟹ *ae x* = *up*·*0*)
⟹ *set r* ⊆ (*domA* Γ ∪ *upds S*) − *edom ce*
⟹ *consistent* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *e*, *S*)
inductive-cases *consistentE*[*elim*!]: *consistent* (*ae*, *ce*, *a*, *as*) (Γ, *e*, *S*)

lemma *closed-consistent*:
assumes *fv e* = ({}::*var set*)
shows *consistent* (⊥, ⊥, *0*, [], []) ([], *e*, [])
proof−
from *assms*
have *edom* (*prognosis* ⊥ [] *0* ([], *e*, [])) = {}
by (*auto dest*!: *set-mp*[*OF edom-prognosis*])
thus *?thesis*
by (*auto simp add*: *edom-empty-iff-bot closed-a-consistent*[*OF assms*])
qed

lemma *card-arity-transform-safe*:
fixes *c c′*
assumes *c* ⇒* *c′* **and** ¬ *boring-step c′* **and** *heap-upds-ok-conf c* **and** *consistent* (*ae*,*ce*,*a*,*as*,*r*) *c*
shows ∃ *ae′ ce′ a′ as′ r′*. *consistent* (*ae′*,*ce′*,*a′*,*as′*,*r′*) *c′* ∧ *conf-transform* (*ae*,*ce*,*a*,*as*,*r*) *c* $\Rightarrow_G{}^*$ *conf-transform* (*ae′*,*ce′*,*a′*,*as′*,*r′*) *c′*
using *assms*(*1*,*2*) *heap-upds-ok-invariant assms*(*3*−)
proof(*induction c c′ arbitrary*: *ae ce a as r rule*:*step-invariant-induction*)
case (app_1 Γ *e x S*)
have *prognosis ae as* (*inc*·*a*) (Γ, *e*, *Arg x* # *S*) ⊑ *prognosis ae as a* (Γ, *App e x*, *S*) **by** (*rule prognosis-App*)
with app_1 **have** *consistent* (*ae*, *ce*, *inc*·*a*, *as*, *r*) (Γ, *e*, *Arg x* # *S*)
by (*auto intro*: *a-consistent-*app_1 *elim*: *below-trans*)
moreover
have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *App e x*, *S*) $\Rightarrow_G$ *conf-transform* (*ae*, *ce*, *inc*·*a*, *as*, *r*) (Γ, *e*, *Arg x* # *S*)
by *simp rule*
ultimately

show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (app_2 Γ *y e x S*)
have *prognosis ae as* (*pred*·*a*) (Γ, *e*[*y*::=*x*], *S*) $\sqsubseteq$ *prognosis ae as a* (Γ, (*Lam* [*y*]. *e*), *Arg x* # *S*)
by (*rule prognosis-subst-Lam*)
then
have *consistent* (*ae*, *ce*, *pred*·*a*, *as*, *r*) (Γ, *e*[*y*::=*x*], *S*) **using** app_2
by (*auto 4 3 intro*: *a-consistent-app*$_2$ *elim*: *below-trans*)
moreover
have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *Lam* [*y*]. *e*, *Arg x* # *S*) $\Rightarrow_G$ *conf-transform* (*ae*, *ce*, *pred* · *a*, *as*, *r*) (Γ, *e*[*y*::=*x*], *S*) **by** (*simp add*: *subst-transform*[*symmetric*]) *rule*
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (*thunk* Γ *x e S*)
hence $x \in$ *thunks* Γ **by** *auto*
hence [*simp*]: $x \in$ *domA* Γ **by** (*rule set-mp*[*OF thunks-domA*])

from *thunk* **have** *prognosis ae as a* (Γ, *Var x*, *S*) $\sqsubseteq$ *ce* **by** *auto*
from *below-trans*[*OF prognosis-called fun-belowD*[*OF this*]]
have [*simp*]: $x \in$ *edom ce* **by** (*auto simp add*: *edom-def*)
hence [*simp*]: $x \notin$ *set r* **using** *thunk* **by** *auto*

from ‹*heap-upds-ok-conf* (Γ, *Var x*, *S*)›
have $x \notin$ *upds S* **by** (*auto dest*!: *heap-upds-okE*)

have $x \in$ *edom ae* **using** *thunk* **by** *auto*
then obtain *u* **where** *ae x* = *up*·*u* **by** (*cases ae x*) (*auto simp add*: *edom-def*)

show *?case*
proof(*cases ce x rule*:*two-cases*)
case *none*
with ‹$x \in$ *edom ce*› **have** *False* **by** (*auto simp add*: *edom-def*)
thus *?thesis*..
next
case *once*

from ‹*prognosis ae as a* (Γ, *Var x*, *S*) $\sqsubseteq$ *ce*›
have *prognosis ae as a* (Γ, *Var x*, *S*) *x* $\sqsubseteq$ *once*
using *once* **by** (*metis* (*mono-tags*) *fun-belowD*)
hence $x \notin$ *ap S* **using** *prognosis-ap*[*of ae as a* Γ (*Var x*) *S*] **by** *auto*

from ‹*map-of* Γ *x* = *Some e*› ‹*ae x* = *up*·*u*› ‹$\neg$ *isVal e*›
have $*$: *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) $\sqsubseteq$ *record-call x* · (*prognosis ae as a* (Γ, *Var x*, *S*))
by (*rule prognosis-Var-thunk*)

from ‹*prognosis ae as a* (Γ, *Var x*, *S*) *x* ⊑ *once*›
have (*record-call x* · (*prognosis ae as a* (Γ, *Var x*, *S*))) *x* = *none*
by (*simp add*: *two-pred-none*)
hence ∗∗: *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) *x* = *none* **using** *fun-belowD*[*OF* ∗, **where** *x* = *x*] **by** *auto*

have *eq*: *prognosis* (*env-delete x ae*) *as u* (*delete x* Γ, *e*, *Upd x* # *S*) = *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*)
by (*rule prognosis-env-cong*) *simp*

have [*simp*]: *restr-stack* (− *set r* − {*x*}) *S* = *restr-stack* (− *set r*) *S*
using ‹*x* ∉ *upds S*› **by** (*auto intro*: *restr-stack-cong*)

have *prognosis* (*env-delete x ae*) *as u* (*delete x* Γ, *e*, *Upd x* # *S*) ⊑ *env-delete x ce*
unfolding *eq*
using ∗∗ *below-trans*[*OF below-trans*[*OF* ∗ *Cfun.monofun-cfun-arg*[*OF* ‹*prognosis ae as a* (Γ, *Var x*, *S*) ⊑ *ce*›]] *record-call-below-arg*]
by (*rule below-env-deleteI*)
moreover

have ∗: *a-consistent* (*env-delete x ae*, *u*, *as*) (*delete x* (*restrictA* (− *set r*) Γ), *e*, *restr-stack* (− *set r*) *S*)
using *thunk* ‹*ae x* = *up*·*u*›
by (*auto intro*!: *a-consistent-thunk-once simp del*: *restr-delete*)
ultimately

have *consistent* (*env-delete x ae*, *env-delete x ce*, *u*, *as*, *x* # *r*) (*delete x* Γ, *e*, *Upd x* # *S*) **using** *thunk*
by (*auto simp add*: *restr-delete-twist Compl-insert elim*:*below-trans*)
moreover

from ∗
have ∗∗: *Astack* (*transform-alts as* (*restr-stack* (− *set r*) *S*) @ *map Dummy* (*rev r*) @ [*Dummy x*]) ⊑ *u* **by** (*auto elim*: *a-consistent-stackD*)

{
from ‹*map-of* Γ *x* = *Some e*› ‹*ae x* = *up*·*u*› *once*
have *map-of* (*map-transform Aeta-expand ae* (*map-transform ccTransform ae* (*restrictA* (− *set r*) Γ))) *x* = *Some* (*Aeta-expand u* (*transform u e*))
by (*simp add*: *map-of-map-transform*)
hence *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *Var x*, *S*) $\Rightarrow_G$
add-dummies-conf r (*delete x* (*map-transform Aeta-expand ae* (*map-transform ccTransform ae* (*restrictA* (− *set r*) Γ))), *Aeta-expand u* (*ccTransform u e*), *Upd x* # *transform-alts as* (*restr-stack* (− *set r*) *S*))
by (*auto simp add*: *map-transform-delete delete-map-transform-env-delete insert-absorb restr-delete-twist simp del*: *restr-delete*)
also
have . . . $\Rightarrow_G{}^*$ *add-dummies-conf* (*x* # *r*) (*delete x* (*map-transform Aeta-expand ae*

(*map-transform ccTransform ae* (*restrictA* (− *set r*) Γ))), *Aeta-expand u* (*ccTransform u e*), *transform-alts as* (*restr-stack* (− *set r*) *S*))
apply (*rule r-into-rtranclp*)
apply (*simp add*: *append-assoc*[*symmetric*] *del*: *append-assoc*)
apply (*rule dropUpd*)
done
also
have ... $\Rightarrow_G{}^*$ *add-dummies-conf* (*x* # *r*) (*delete x* (*map-transform Aeta-expand ae* (*map-transform ccTransform ae* (*restrictA* (− *set r*) Γ))), *ccTransform u e*, *transform-alts as* (*restr-stack* (− *set r*) *S*))
by *simp* (*intro normal-trans Aeta-expand-safe* ∗∗)
also(*rtranclp-trans*)
have ... = *conf-transform* (*env-delete x ae*, *env-delete x ce*, *u*, *as*, *x* # *r*) (*delete x* Γ, *e*, *Upd x* # *S*)
by (*auto intro*!: *map-transform-cong simp add*: *map-transform-delete*[*symmetric*] *restr-delete-twist Compl-insert*)
finally(*back-subst*)
have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *Var x*, *S*) $\Rightarrow_G{}^*$ *conf-transform* (*env-delete x ae*, *env-delete x ce*, *u*, *as*, *x* # *r*) (*delete x* Γ, *e*, *Upd x* # *S*).
}
ultimately
show *?thesis* **by** (*blast del*: *consistentI consistentE*)

next
case *many*

from ⟨*map-of* Γ *x* = *Some e*⟩ ⟨*ae x* = *up*·*u*⟩ ⟨¬ *isVal e*⟩
have *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) ⊑ *record-call x* · (*prognosis ae as a* (Γ, *Var x*, *S*))
by (*rule prognosis-Var-thunk*)
also note *record-call-below-arg*
finally
have ∗: *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) ⊑ *prognosis ae as a* (Γ, *Var x*, *S*) **by** *this simp-all*

have *ae x* = *up*·*0* **using** *thunk many* ⟨*x* ∈ *thunks* Γ⟩ **by** (*auto*)
hence *u* = *0* **using** ⟨*ae x* = *up*·*u*⟩ **by** *simp*

have *prognosis ae as 0* (*delete x* Γ, *e*, *Upd x* # *S*) ⊑ *ce* **using** ∗[*unfolded* ⟨*u*=*0*⟩] *thunk* **by** (*auto elim*: *below-trans*)
moreover
have *a-consistent* (*ae*, *0*, *as*) (*delete x* (*restrictA* (− *set r*) Γ), *e*, *Upd x* # *restr-stack* (− *set r*) *S*) **using** *thunk* ⟨*ae x* = *up*·*0*⟩
by (*auto intro*!: *a-consistent-thunk-0 simp del*: *restr-delete*)
ultimately
have *consistent* (*ae*, *ce*, *0*, *as*, *r*) (*delete x* Γ, *e*, *Upd x* # *S*) **using** *thunk* ⟨*ae x* = *up*·*u*⟩ ⟨*u* = *0*⟩
by (*auto simp add*: *restr-delete-twist*)

moreover

from ⟨*map-of* Γ *x* = *Some e*⟩ ⟨*ae x* = *up·0*⟩ *many*
have *map-of* (*map-transform Aeta-expand ae* (*map-transform ccTransform ae* (*restrictA* (− *set r*) Γ))) *x* = *Some* (*transform 0 e*)
by (*simp add*: *map-of-map-transform*)
with ⟨¬ *isVal e*⟩
have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *Var x*, *S*) $\Rightarrow_G$ *conf-transform* (*ae*, *ce*, *0*, *as*, *r*) (*delete x* Γ, *e*, *Upd x* # *S*)
by (*auto intro*: *gc-step.intros simp add*: *map-transform-delete restr-delete-twist intro*!: *step.intros simp del*: *restr-delete*)
ultimately
show *?thesis* **by** (*blast del*: *consistentI consistentE*)
qed
next
case (*lamvar* Γ *x e S*)
from *lamvar*(*1*) **have** [*simp*]: *x* ∈ *domA* Γ **by** (*metis domI dom-map-of-conv-domA*)

from *lamvar* **have** *prognosis ae as a* (Γ, *Var x*, *S*) ⊑ *ce* **by** *auto*
from *below-trans*[*OF prognosis-called fun-belowD*[*OF this*]]
have [*simp*]: *x* ∈ *edom ce* **by** (*auto simp add*: *edom-def*)
then obtain *c* **where** *ce x* = *up·c* **by** (*cases ce x*) (*auto simp add*: *edom-def*)

from *lamvar*
have [*simp*]: *x* ∉ *set r* **by** *auto*

then have *x* ∈ *edom ae* **using** *lamvar* **by** *auto*
then obtain *u* **where** *ae x* = *up·u* **by** (*cases ae x*) (*auto simp add*: *edom-def*)

have *prognosis ae as u* ((*x*, *e*) # *delete x* Γ, *e*, *S*) = *prognosis ae as u* (Γ, *e*, *S*)
using ⟨*map-of* Γ *x* = *Some e*⟩ **by** (*auto intro*!: *prognosis-reorder*)
also have ... ⊑ *record-call x* · (*prognosis ae as a* (Γ, *Var x*, *S*))
using ⟨*map-of* Γ *x* = *Some e*⟩ ⟨*ae x* = *up·u*⟩ ⟨*isVal e*⟩ **by** (*rule prognosis-Var-lam*)
also have ... ⊑ *prognosis ae as a* (Γ, *Var x*, *S*) **by** (*rule record-call-below-arg*)
finally have ∗: *prognosis ae as u* ((*x*, *e*) # *delete x* Γ, *e*, *S*) ⊑ *prognosis ae as a* (Γ, *Var x*, *S*) **by** *this simp-all*
moreover
have *a-consistent* (*ae*, *u*, *as*) ((*x*,*e*) # *delete x* (*restrictA* (− *set r*) Γ), *e*, *restr-stack* (− *set r*) *S*) **using** *lamvar* ⟨*ae x* = *up·u*⟩
by (*auto intro*!: *a-consistent-lamvar simp del*: *restr-delete*)
ultimately
have *consistent* (*ae*, *ce*, *u*, *as*, *r*) ((*x*, *e*) # *delete x* Γ, *e*, *S*)
using *lamvar edom-mono*[*OF* ∗] **by** (*auto simp add*: *thunks-Cons restr-delete-twist elim*: *below-trans*)
moreover

from ⟨*a-consistent* - -⟩
have ∗∗: *Astack* (*transform-alts as* (*restr-stack* (− *set r*) *S*) @ *map Dummy* (*rev r*)) ⊑ *u*

by (*auto elim*: *a-consistent-stackD*)

{
from ‹*isVal e*›
have *isVal* (*transform u e*) **by** *simp*
hence *isVal* (*Aeta-expand u* (*transform u e*)) **by** (*rule isVal-Aeta-expand*)
moreover
from ‹*map-of* Γ *x* = *Some e*› ‹*ae x* = *up* · *u*› ‹*ce x* = *up*·*c*› ‹*isVal* (*transform u e*)›
have *map-of* (*map-transform Aeta-expand ae* (*map-transform transform ae* (*restrictA* (− *set r*) Γ))) *x* = *Some* (*Aeta-expand u* (*transform u e*))
by (*simp add*: *map-of-map-transform*)
ultimately
have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *Var x*, *S*) $\Rightarrow_G{}^*$
add-dummies-conf r ((*x*, *Aeta-expand u* (*transform u e*)) # *delete x* (*map-transform Aeta-expand ae* (*map-transform transform ae* (*restrictA* (− *set r*) Γ))), *Aeta-expand u* (*transform u e*), *transform-alts as* (*restr-stack* (− *set r*) *S*))
by (*auto intro*!: *normal-trans*[*OF lambda-var*] *simp add*: *map-transform-delete simp del*: *restr-delete*)
also have ... = *add-dummies-conf r* ((*map-transform Aeta-expand ae* (*map-transform transform ae* ((*x*,*e*) # *delete x* (*restrictA* (− *set r*) Γ)))), *Aeta-expand u* (*transform u e*), *transform-alts as* (*restr-stack* (− *set r*) *S*))
using ‹*ae x* = *up* · *u*› ‹*ce x* = *up*·*c*› ‹*isVal* (*transform u e*)›
by (*simp add*: *map-transform-Cons map-transform-delete restr-delete-twist del*: *restr-delete*)
also(*subst*[*rotated*]) **have** ... $\Rightarrow_G{}^*$ *conf-transform* (*ae*, *ce*, *u*, *as*, *r*) ((*x*, *e*) # *delete x* Γ, *e*, *S*)
by (*simp add*: *restr-delete-twist*) (*rule normal-trans*[*OF Aeta-expand-safe*[*OF* ∗∗]])
finally(*rtranclp-trans*)
have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *Var x*, *S*) $\Rightarrow_G{}^*$ *conf-transform* (*ae*, *ce*, *u*, *as*, *r*) ((*x*, *e*) # *delete x* Γ, *e*, *S*).
}
ultimately show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (*var*$_2$ Γ *x e S*)
show *?case*
proof(*cases x* ∈ *set r*)
case [*simp*]: *False*

from *var*$_2$
have *a-consistent* (*ae*, *a*, *as*) (*restrictA* (− *set r*) Γ, *e*, *Upd x* # *restr-stack* (−*set r*) *S*) **by** *auto*
from *a-consistent-UpdD*[*OF this*]
have *ae x* = *up*·*0* **and** *a* = *0*.

from ‹*isVal e*› ‹*x* ∉ *domA* Γ›
have ∗: *prognosis ae as 0* ((*x*, *e*) # Γ, *e*, *S*) ⊑ *prognosis ae as 0* (Γ, *e*, *Upd x* # *S*) **by** (*rule prognosis-Var2*)
moreover
have *a-consistent* (*ae*, *a*, *as*) ((*x*, *e*) # *restrictA* (− *set r*) Γ, *e*, *restr-stack* (− *set r*) *S*)
using *var*$_2$ **by** (*auto intro*!: *a-consistent-var*$_2$)

ultimately
have *consistent* $(ae,\ ce,\ 0,\ as,\ r)\ ((x,\ e)\ \#\ \Gamma,\ e,\ S)$
using var_2 ‹$a = 0$›
by (*auto simp add*: *thunks-Cons elim*: *below-trans*)
moreover
have *conf-transform* $(ae,\ ce,\ a,\ as,\ r)\ (\Gamma,\ e,\ Upd\ x\ \#\ S) \Rightarrow_G$ *conf-transform* $(ae,\ ce,\ 0,\ as,\ r)\ ((x,\ e)\ \#\ \Gamma,\ e,\ S)$
using ‹$ae\ x = up\cdot 0$› ‹$a = 0$› var_2
by (*auto intro*: *gc-step.intros simp add*: *map-transform-Cons*)
ultimately show *?thesis* **by** (*blast del*: *consistentI consistentE*)
next
case *True*
hence $ce\ x = \bot$ **using** var_2 **by** (*auto simp add*: *edom-def*)
hence $x \notin edom\ ce$ **by** (*simp add*: *edomIff*)
hence $x \notin edom\ ae$ **using** var_2 **by** *auto*
hence [*simp*]: $ae\ x = \bot$ **by** (*auto simp add*: *edom-def*)

note ‹$x \in set\ r$›[*simp*]

have *prognosis ae as a* $((x,\ e)\ \#\ \Gamma,\ e,\ S) \sqsubseteq$ *prognosis ae as a* $((x,\ e)\ \#\ \Gamma,\ e,\ Upd\ x\ \#\ S)$ **by** (*rule prognosis-upd*)
also have $\ldots \sqsubseteq$ *prognosis ae as a* (*delete* x $((x,e)\ \#\ \Gamma),\ e,\ Upd\ x\ \#\ S)$
using ‹$ae\ x = \bot$› **by** (*rule prognosis-not-called*)
also have *delete* $x\ ((x,e)\#\Gamma) = \Gamma$ **using** ‹$x \notin domA\ \Gamma$› **by** *simp*
finally
have $*$: *prognosis ae as a* $((x,\ e)\ \#\ \Gamma,\ e,\ S) \sqsubseteq$ *prognosis ae as a* $(\Gamma,\ e,\ Upd\ x\ \#\ S)$ **by** *this simp*
then
have *consistent* $(ae,\ ce,\ a,\ as,\ r)\ ((x,\ e)\ \#\ \Gamma,\ e,\ S)$ **using** var_2
by (*auto simp add*: *thunks-Cons elim*:*below-trans a-consistent-var*$_2$)
moreover
have *conf-transform* $(ae,\ ce,\ a,\ as,\ r)\ (\Gamma,\ e,\ Upd\ x\ \#\ S) =$ *conf-transform* $(ae,\ ce,\ a,\ as,\ r)\ ((x,\ e)\ \#\ \Gamma,\ e,\ S)$
by (*auto simp add*: *map-transform-restrA*[*symmetric*])
ultimately show *?thesis*
by (*fastforce del*: *consistentI consistentE simp del*:*conf-transform.simps*)
qed
next
case $(let_1\ \Delta\ \Gamma\ e\ S)$
let $?ae = Aheap\ \Delta\ e\cdot a$
let $?ce = cHeap\ \Delta\ e\cdot a$

have $domA\ \Delta \cap upds\ S = \{\}$ **using** *fresh-distinct-fv*[*OF let*$_1$(*2*)] **by** (*auto dest*: *set-mp*[*OF ups-fv-subset*])
hence $*$: $\bigwedge x.\ x \in upds\ S \Longrightarrow x \notin edom\ ?ae$ **by** (*auto simp add*: *edom-cHeap dest*!: *set-mp*[*OF edom-Aheap*])
have *restr-stack-simp2*: *restr-stack* $(edom\ (?ae \sqcup ae))\ S =$ *restr-stack* $(edom\ ae)\ S$
by (*auto intro*: *restr-stack-cong dest*!: $*$)

have *edom ce* = *edom ae* **using** let_1 **by** *auto*

have *edom ae* ⊆ *domA* Γ ∪ *upds S* **using** let_1 **by** (*auto dest*!: *a-consistent-edom-subsetD*)
from *set-mp*[*OF this*] *fresh-distinct*[*OF* let_1(*1*)] *fresh-distinct-fv*[*OF* let_1(*2*)]
have *edom ae* ∩ *domA* Δ = {} **by** (*auto dest*: *set-mp*[*OF ups-fv-subset*])

from ‹*edom ae* ∩ *domA* Δ = {}›
have [*simp*]: *edom* (*Aheap* Δ *e*·*a*) ∩ *edom ae* = {} **by** (*auto dest*!: *set-mp*[*OF edom-Aheap*])

from *fresh-distinct*[*OF* let_1(*1*)]
have [*simp*]: *restrictA* (*edom ae* ∪ *edom* (*Aheap* Δ *e*·*a*)) Γ = *restrictA* (*edom ae*) Γ
by (*auto intro*: *restrictA-cong dest*!: *set-mp*[*OF edom-Aheap*])

have *set r* ⊆ *domA* Γ ∪ *upds S* **using** let_1 **by** *auto*
have [*simp*]: *restrictA* (− *set r*) Δ = Δ
apply (*rule restrictA-noop*)
apply *auto*
by (*metis IntI UnE* ‹*set r* ⊆ *domA* Γ ∪ *upds S*› ‹*domA* Δ ∩ *domA* Γ = {}› ‹*domA* Δ ∩ *upds S* = {}› *contra-subsetD empty-iff*)

{
have *edom* (*?ae* ⊔ *ae*) = *edom* (*?ce* ⊔ *ce*)
using let_1(*4*) **by** (*auto simp add*: *edom-cHeap*)
moreover
{ fix *x e*′
assume *x* ∈ *thunks* Γ
hence *x* ∉ *edom ?ce* **using** *fresh-distinct*[*OF* let_1(*1*)]
by (*auto simp add*: *edom-cHeap dest*: *set-mp*[*OF edom-Aheap*] *set-mp*[*OF thunks-domA*])
hence [*simp*]: *?ce x* = ⊥ **unfolding** *edomIff* **by** *auto*

assume *many* ⊑ (*?ce* ⊔ *ce*) *x*
with let_1 ‹*x* ∈ *thunks* Γ›
have (*?ae* ⊔ *ae*) *x* = *up* ·*0* **by** *auto*
}
moreover
{ fix *x e*′
assume *x* ∈ *thunks* Δ
hence *x* ∉ *domA* Γ **and** *x* ∉ *upds S*
using *fresh-distinct*[*OF* let_1(*1*)] *fresh-distinct-fv*[*OF* let_1(*2*)]
by (*auto dest*!: *set-mp*[*OF thunks-domA*] *set-mp*[*OF ups-fv-subset*])
hence *x* ∉ *edom ce* **using** ‹*edom ae* ⊆ *domA* Γ ∪ *upds S*› ‹*edom ce* = *edom ae*› **by** *auto*
hence [*simp*]: *ce x* = ⊥ **by** (*auto simp add*: *edomIff*)

assume *many* ⊑ (*?ce* ⊔ *ce*) *x* **with** ‹*x* ∈ *thunks* Δ›
have (*?ae* ⊔ *ae*) *x* = *up*·*0* **by** (*auto simp add*: *Aheap-heap3*)
}
moreover
{

from $let_1(1,2)$ ‹$edom\ ae \subseteq domA\ \Gamma \cup upds\ S$›
have $prognosis\ (?ae \sqcup ae)\ as\ a\ (\Delta\ @\ \Gamma,\ e,\ S) \sqsubseteq\ ?ce \sqcup prognosis\ ae\ as\ a\ (\Gamma,\ Let\ \Delta\ e,\ S)$ **by** (*rule prognosis-Let*)
also have $prognosis\ ae\ as\ a\ (\Gamma,\ Let\ \Delta\ e,\ S) \sqsubseteq ce$ **using** let_1 **by** *auto*
finally have $prognosis\ (?ae \sqcup ae)\ as\ a\ (\Delta\ @\ \Gamma,\ e,\ S) \sqsubseteq\ ?ce \sqcup ce$ **by** *this simp*
}
moreover

have $a\text{-}consistent\ (ae,\ a,\ as)\ (restrictA\ (-\ set\ r)\ \Gamma,\ Let\ \Delta\ e,\ restr\text{-}stack\ (-\ set\ r)\ S)$
using let_1 **by** *auto*
hence $a\text{-}consistent\ (?ae \sqcup ae,\ a,\ as)\ (\Delta\ @\ restrictA\ (-\ set\ r)\ \Gamma,\ e,\ restr\text{-}stack\ (-\ set\ r)\ S)$
using $let_1(1,2)$ ‹$edom\ ae \cap domA\ \Delta = \{\}$›
by (*auto intro*!: *a-consistent-let simp del*: *join-comm*)
hence $a\text{-}consistent\ (?ae \sqcup ae,\ a,\ as)\ (restrictA\ (-\ set\ r)\ (\Delta\ @\ \Gamma),\ e,\ restr\text{-}stack\ (-\ set\ r)\ S)$
by (*simp add*: *restrictA-append*)
moreover
have $set\ r \subseteq (domA\ \Gamma \cup upds\ S) - edom\ ce$ **using** let_1 **by** *auto*
hence $set\ r \subseteq (domA\ \Gamma \cup upds\ S) - edom\ (?ce \sqcup ce)$
apply (*rule order-trans*)
using ‹$domA\ \Delta \cap domA\ \Gamma = \{\}$› ‹$domA\ \Delta \cap upds\ S = \{\}$›
apply (*auto simp add*: *edom-cHeap dest*!: *set-mp*[*OF edom-Aheap*])
done
ultimately
have $consistent\ (?ae \sqcup ae,\ ?ce \sqcup ce,\ a,\ as,\ r)\ (\Delta\ @\ \Gamma,\ e,\ S)$ **by** *auto*
}
moreover
{
have $\bigwedge x.\ x \in domA\ \Gamma \Longrightarrow x \notin edom\ ?ae\ \bigwedge x.\ x \in domA\ \Gamma \Longrightarrow x \notin edom\ ?ce$
using *fresh-distinct*[*OF* $let_1(1)$]
by (*auto simp add*: *edom-cHeap dest*!: *set-mp*[*OF edom-Aheap*])
hence $map\text{-}transform\ Aeta\text{-}expand\ (?ae \sqcup ae)\ (map\text{-}transform\ transform\ (?ae \sqcup ae)\ (restrictA\ (-set\ r)\ \Gamma))$
$= map\text{-}transform\ Aeta\text{-}expand\ ae\ (map\text{-}transform\ transform\ ae\ (restrictA\ (-set\ r)\ \Gamma))$
by (*auto intro*!: *map-transform-cong restrictA-cong simp add*: *edomIff*)
moreover

from ‹$edom\ ae \subseteq domA\ \Gamma \cup upds\ S$› ‹$edom\ ce = edom\ ae$›
have $\bigwedge x.\ x \in domA\ \Delta \Longrightarrow x \notin edom\ ce$ **and** $\bigwedge x.\ x \in domA\ \Delta \Longrightarrow x \notin edom\ ae$
using *fresh-distinct*[*OF* $let_1(1)$] *fresh-distinct-ups*[*OF* $let_1(2)$] **by** *auto*
hence $map\text{-}transform\ Aeta\text{-}expand\ (?ae \sqcup ae)\ (map\text{-}transform\ transform\ (?ae \sqcup ae)\ (restrictA\ (-\ set\ r)\ \Delta))$
$= map\text{-}transform\ Aeta\text{-}expand\ ?ae\ (map\text{-}transform\ transform\ ?ae\ (restrictA\ (-\ set\ r)\ \Delta))$
by (*auto intro*!: *map-transform-cong restrictA-cong simp add*: *edomIff*)
moreover

from ‹$domA\ \Delta \cap domA\ \Gamma = \{\}$› ‹$domA\ \Delta \cap upds\ S = \{\}$›

have *atom ‘ domA* Δ $\sharp$* *set r*
by (*auto simp add*: *fresh-star-def fresh-at-base fresh-finite-set-at-base dest*!: *set-mp*[*OF* ‹*set r* $\subseteq$ *domA* Γ $\cup$ *upds S*›])
hence *atom ‘ domA* Δ $\sharp$* *map Dummy* (*rev r*)
apply −
apply (*rule eqvt-fresh-star-cong1*[**where** *f* = *map Dummy*], *perm-simp*, *rule*)
apply (*rule eqvt-fresh-star-cong1*[**where** *f* = *rev*], *perm-simp*, *rule*)
apply (*auto simp add*: *fresh-star-def fresh-set*)
done
ultimately

have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *Let* Δ *e*, *S*) $\Rightarrow_G$ *conf-transform* (*?ae* $\sqcup$ *ae*, *?ce* $\sqcup$ *ce*, *a*, *as*, *r*) (Δ @ Γ, *e*, *S*)
using *restr-stack-simp2* let_1(*1*,*2*) ‹*edom ce* = *edom ae*›
apply (*auto simp add*: *map-transform-append restrictA-append edom-cHeap restr-stack-simp2*[*simplified*])
apply (*rule normal*)
apply (*rule step*.let_1)
apply (*auto intro*: *normal step*.let_1 *dest*: *set-mp*[*OF edom-Aheap*] *simp add*: *fresh-star-list*)
done
}
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (if_1 Γ *scrut e1 e2 S*)
have *prognosis ae as a* (Γ, *scrut ? e1* : *e2*, *S*) $\sqsubseteq$ *ce* **using** if_1 **by** *auto*
hence *prognosis ae* (*a#as*) *0* (Γ, *scrut*, *Alts e1 e2* # *S*) $\sqsubseteq$ *ce*
by (*rule below-trans*[*OF prognosis-IfThenElse*])
hence *consistent* (*ae*, *ce*, *0*, *a#as*, *r*) (Γ, *scrut*, *Alts e1 e2* # *S*)
using if_1 **by** (*auto dest*: *a-consistent-*if_1)
moreover
have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *scrut ? e1* : *e2*, *S*) $\Rightarrow_G$ *conf-transform* (*ae*, *ce*, *0*, *a#as*, *r*) (Γ, *scrut*, *Alts e1 e2* # *S*)
by (*auto intro*: *normal step.intros*)
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case (if_2 Γ *b e1 e2 S*)
hence *a-consistent* (*ae*, *a*, *as*) (*restrictA* (− *set r*) Γ, *Bool b*, *Alts e1 e2* # *restr-stack* (−*set r*) *S*) **by** *auto*
then obtain *a' as'* **where** [*simp*]: *as* = *a'* # *as' a* = *0*
by (*rule a-consistent-alts-on-stack*)

{
have *prognosis ae* (*a'#as'*) *0* (Γ, *Bool b*, *Alts e1 e2* # *S*) $\sqsubseteq$ *ce* **using** if_2 **by** *auto*
hence *prognosis ae as' a'* (Γ, *if b then e1 else e2*, *S*) $\sqsubseteq$ *ce* **by** (*rule below-trans*[*OF prognosis-Alts*])
then

have *consistent* (ae, ce, a', as', r) $(\Gamma, \textit{if } b \textit{ then } e1 \textit{ else } e2, S)$
using if_2 **by** (*auto dest*!: *a-consistent-*if_2)
}
moreover
have *conf-transform* (ae, ce, a, as, r) $(\Gamma, \textit{Bool } b, \textit{Alts } e1\ e2\ \#\ S) \Rightarrow_G$ *conf-transform* (ae, ce, a', as', r) $(\Gamma, \textit{if } b \textit{ then } e1 \textit{ else } e2, S)$
by (*auto intro*: *normal step.*if_2[**where** b = *True*, *simplified*] *step.*if_2[**where** b = *False*, *simplified*])
ultimately
show *?case* **by** (*blast del*: *consistentI consistentE*)
next
case *refl* **thus** *?case* **by** *force*
next
case $(trans\ c\ c'\ c'')$
from *trans*(*3*)[*OF trans*(*5*)]
obtain $ae'\ ce'\ a'\ as'\ r'$
where *consistent* (ae', ce', a', as', r') c' **and** $*$: *conf-transform* (ae, ce, a, as, r) $c \Rightarrow_G{}^*$ *conf-transform* (ae', ce', a', as', r') c' **by** *blast*
from *trans*(*4*)[*OF this*(*1*)]
obtain $ae''\ ce''\ a''\ as''\ r''$
where *consistent* $(ae'', ce'', a'', as'', r'')$ c'' **and** $**$: *conf-transform* (ae', ce', a', as', r') $c' \Rightarrow_G{}^*$ *conf-transform* $(ae'', ce'', a'', as'', r'')$ c'' **by** *blast*
from *this*(*1*) *rtranclp-trans*[*OF* $*$ $**$]
show *?case* **by** *blast*
qed
end

end

79 CoCallAritySig.tex

theory *CoCallAritySig*
imports *ArityAnalysisSig CoCallAnalysisSig*
begin

locale *CoCallArity* = *CoCallAnalysis* + *ArityAnalysis*

end

80 CoCallAnalysisSpec.tex

theory *CoCallAnalysisSpec*
imports *CoCallAritySig ArityAnalysisSpec*
begin

hide-const *Multiset.single*

locale *CoCallArityEdom* = *CoCallArity* + *EdomArityAnalysis*

locale *CoCallAritySafe* = *CoCallArity* + *CoCallAnalyisHeap* + *ArityAnalysisLetSafe* +
assumes *ccExp-App*: $ccExp\ e\cdot(inc\cdot a) \sqcup ccProd\ \{x\}\ (insert\ x\ (fv\ e)) \sqsubseteq ccExp\ (App\ e\ x)\cdot a$
assumes *ccExp-Lam*: $cc\text{-}restr\ (fv\ (Lam\ [y].\ e))\ (ccExp\ e\cdot(pred\cdot n)) \sqsubseteq ccExp\ (Lam\ [y].\ e)\cdot n$
assumes *ccExp-subst*: $x \notin S \Longrightarrow y \notin S \Longrightarrow cc\text{-}restr\ S\ (ccExp\ e[y::=x]\cdot a) \sqsubseteq cc\text{-}restr\ S\ (ccExp\ e\cdot a)$
assumes *ccExp-pap*: $isVal\ e \Longrightarrow ccExp\ e\cdot 0 = ccSquare\ (fv\ e)$
assumes *ccExp-Let*: $cc\text{-}restr\ (-domA\ \Gamma)\ (ccHeap\ \Gamma\ e\cdot a) \sqsubseteq ccExp\ (Let\ \Gamma\ e)\cdot a$
assumes *ccExp-IfThenElse*: $ccExp\ scrut\cdot 0 \sqcup (ccExp\ e1\cdot a \sqcup ccExp\ e2\cdot a) \sqcup ccProd\ (edom\ (Aexp\ scrut\cdot 0))\ (edom\ (Aexp\ e1\cdot a) \cup edom\ (Aexp\ e2\cdot a)) \sqsubseteq ccExp\ (scrut\ ?\ e1\ :\ e2)\cdot a$

assumes *ccHeap-Exp*: $ccExp\ e\cdot a \sqsubseteq ccHeap\ \Delta\ e\cdot a$
assumes *ccHeap-Heap*: $map\text{-}of\ \Delta\ x = Some\ e' \Longrightarrow (Aheap\ \Delta\ e\cdot a)\ x= up\cdot a' \Longrightarrow ccExp\ e'\cdot a' \sqsubseteq ccHeap\ \Delta\ e\cdot a$
assumes *ccHeap-Extra-Edges*:
$map\text{-}of\ \Delta\ x = Some\ e' \Longrightarrow (Aheap\ \Delta\ e\cdot a)\ x = up\cdot a' \Longrightarrow ccProd\ (fv\ e')\ (ccNeighbors\ x\ (ccHeap\ \Delta\ e\cdot a) - \{x\} \cap thunks\ \Delta) \sqsubseteq ccHeap\ \Delta\ e\cdot a$

assumes *aHeap-thunks-rec*: $\neg\ nonrec\ \Gamma \Longrightarrow x \in thunks\ \Gamma \Longrightarrow x \in edom\ (Aheap\ \Gamma\ e\cdot a) \Longrightarrow (Aheap\ \Gamma\ e\cdot a)\ x = up\cdot 0$
assumes *aHeap-thunks-nonrec*: $nonrec\ \Gamma \Longrightarrow x \in thunks\ \Gamma \Longrightarrow x{-}{-}x \in ccExp\ e\cdot a \Longrightarrow (Aheap\ \Gamma\ e\cdot a)\ x = up\cdot 0$

end

81 ArityAnalysisFixProps.tex

theory *ArityAnalysisFixProps*
imports *ArityAnalysisFix ArityAnalysisSpec*
begin

context *SubstArityAnalysis*
begin

lemma *Afix-restr-subst*:
assumes $x \notin S$
assumes $y \notin S$
assumes $domA\ \Gamma \subseteq S$
shows $Afix\ \Gamma[x::h=y]\cdot ae\ f|`\ S = Afix\ \Gamma\cdot(ae\ f|`\ S)\ f|`\ S$
by (*rule Afix-restr-subst'*[*OF Aexp-subst-restr*[*OF assms*(*1*,*2*)] *assms*])
end

end

82 CoCallImplSafe.tex

theory *CoCallImplSafe*
imports *CoCallAnalysisImpl CoCallAnalysisSpec ArityAnalysisFixProps*
begin

locale *CoCallImplSafe*
begin
sublocale *CoCallAnalysisImpl***.**

lemma *ccNeighbors-Int-ccrestr*: (*ccNeighbors x G* $\cap$ *S*) = *ccNeighbors x* (*cc-restr* (*insert x S*) *G*) $\cap$ *S*
  by *transfer auto*

lemma
  assumes $x \notin S$ **and** $y \notin S$
  shows *CCexp-subst*: *cc-restr S* (*CCexp e*[*y*::=*x*]·*a*) = *cc-restr S* (*CCexp e*·*a*)
    and *Aexp-restr-subst*: (*Aexp e*[*y*::=*x*]·*a*) *f*|‘ *S* = (*Aexp e*·*a*) *f*|‘ *S*
using *assms*
proof (*nominal-induct e avoiding*: *x y arbitrary*: *a S rule*: *exp-strong-induct-rec-set*)
  case (*Var b v*)
  case *1* **show** *?case* **by** *auto*
  case *2* **thus** *?case* **by** *auto*
next
  case (*App e v*)
  case *1*
    with *App* **show** *?case*
    by (*auto simp add*: *Int-insert-left fv-subst-int simp del*: *join-comm intro*: *join-mono*)
  case *2*
    with *App* **show** *?case*
     by (*auto simp add*: *env-restr-join simp del*: *fun-meet-simp*)
next
  case (*Lam v e*)
  case *1*
    with *Lam*
    show *?case*
     by (*auto simp add*: *CCexp-pre-simps cc-restr-predCC Diff-Int-distrib2 fv-subst-int env-restr-join env-delete-env-restr-swap*[*symmetric*] *simp del*: *CCexp-simps*)
  case *2*
    with *Lam*
    show *?case*
     by (*auto simp add*: *env-restr-join env-delete-env-restr-swap*[*symmetric*] *simp del*: *fun-meet-simp*)
next
  case (*Let* Γ *e x y*)
  hence [*simp*]: $x \notin$ *domA* Γ $y \notin$ *domA* Γ
   by (*metis* (*erased*, *hide-lams*) *bn-subst domA-not-fresh fresh-def fresh-star-at-base fresh-star-def obtain-fresh subst-is-fresh*(*2*))+

  note *Let*(*1*,*2*)[*simp*]

from *Let*(*3*)
have ¬ *nonrec* (Γ[*y*::*h*=*x*]) **by** (*simp add*: *nonrec-subst*)

case [*simp*]: *1*
have *cc-restr* (*S* ∪ *domA* Γ) (*CCfix* Γ[*y*::*h*=*x*]·(*Afix* Γ[*y*::*h*=*x*]·(*Aexp* *e*[*y*::=*x*]·*a* ⊔ (λ-. *up*·*0*) *f*|' *thunks* Γ), *CCexp* *e*[*y*::=*x*]·*a*)) =
cc-restr (*S* ∪ *domA* Γ) (*CCfix* Γ· (*Afix* Γ· (*Aexp* *e*· *a* ⊔ (λ-. *up*·*0*) *f*|' *thunks* Γ), *CCexp* *e*· *a*))
apply (*subst CCfix-restr-subst'*)
apply (*erule Let*(*4*))
apply *auto*[*5*]
apply (*subst CCfix-restr*) **back**
apply *simp*
apply (*subst Afix-restr-subst'*)
apply (*erule Let*(*5*))
apply *auto*[*5*]
apply (*subst Afix-restr*) **back**
apply *simp*
apply (*simp only*: *env-restr-join*)
apply (*subst Let*(*7*))
apply *auto*[*2*]
apply (*subst Let*(*6*))
apply *auto*[*2*]
apply *rule*
done
thus *?case* **using** *Let*(*1*,*2*) ‹¬ *nonrec* Γ› ‹¬ *nonrec* (Γ[*y*::*h*=*x*])›
by (*auto simp add*: *fresh-star-Pair* *elim*: *cc-restr-eq-subset*[*rotated*])

case [*simp*]: *2*
have *Afix* Γ[*y*::*h*=*x*]·(*Aexp* *e*[*y*::=*x*]·*a* ⊔ (λ-. *up*·*0*) *f*|' (*thunks* Γ)) *f*|' (*S* ∪ *domA* Γ) = *Afix* Γ·(*Aexp* *e*·*a* ⊔ (λ-. *up*·*0*) *f*|' (*thunks* Γ)) *f*|' (*S* ∪ *domA* Γ)
apply (*subst Afix-restr-subst'*)
apply (*erule Let*(*5*))
apply *auto*[*5*]
apply (*subst Afix-restr*) **back**
apply *auto*[*1*]
apply (*simp only*: *env-restr-join*)
apply (*subst Let*(*7*))
apply *auto*[*2*]
apply *rule*
done
thus *?case* **using** *Let*(*1*,*2*)
using ‹¬ *nonrec* Γ› ‹¬ *nonrec* (Γ[*y*::*h*=*x*])›
by (*auto simp add*: *fresh-star-Pair elim*:*env-restr-eq-subset*[*rotated*])
next
case (*Let-nonrec x' e exp x y*)

from *Let-nonrec*(*1*,*2*)

have $x \neq x'$ $y \neq x'$ **by** (*simp-all add*: *fresh-at-base*)

note *Let-nonrec*(*1*,*2*)[*simp*]

from ‹$x' \notin fv\ e$› ‹$y \neq x'$› ‹$x \neq x'$›
have [*simp*]: $x' \notin fv\ (e[y::=x])$
  by (*auto simp add*: *fv-subst-eq*)

note ‹$x' \notin fv\ e$›[*simp*] ‹$y \neq x'$› [*simp*]‹$x \neq x'$› [*simp*]

case [*simp*]: *1*

have $\bigwedge a.\ cc\text{-}restr\ \{x'\}\ (CCexp\ exp[y::=x]\cdot a) = cc\text{-}restr\ \{x'\}\ (CCexp\ exp\cdot a)$
 by (*rule Let-nonrec*(*6*)) *auto*
from *arg-cong*[**where** $f = \lambda x.\ x'--x' \in x$, *OF this*]
have [*simp*]: $x'--x' \in CCexp\ exp[y::=x]\cdot a \longleftrightarrow x'--x' \in CCexp\ exp\cdot a$ **by** *auto*

have [*simp*]: $\bigwedge a.\ Aexp\ e[y::=x]\cdot a\ f|`\ S = Aexp\ e\cdot a\ f|`\ S$
  by (*rule Let-nonrec*(*5*)) *auto*

have [*simp*]: $\bigwedge a.\ fup\cdot(Aexp\ e[y::=x])\cdot a\ f|`\ S = fup\cdot(Aexp\ e)\cdot a\ f|`\ S$
  by (*case-tac a*) *auto*

have [*simp*]: $Aexp\ exp[y::=x]\cdot a\ f|`\ S = Aexp\ exp\cdot a\ f|`\ S$
  by (*rule Let-nonrec*(*7*)) *auto*

have $Aexp\ exp[y::=x]\cdot a\ f|`\ \{x'\} = Aexp\ exp\cdot a\ f|`\ \{x'\}$
  by (*rule Let-nonrec*(*7*)) *auto*
from *fun-cong*[*OF this*, **where** $x = x'$]
have [*simp*]: $(Aexp\ exp[y::=x]\cdot a)\ x' = (Aexp\ exp\cdot a)\ x'$ **by** *auto*

have [*simp*]: $\bigwedge a.\ cc\text{-}restr\ S\ (CCexp\ exp[y::=x]\cdot a) = cc\text{-}restr\ S\ (CCexp\ exp\cdot a)$
  by (*rule Let-nonrec*(*6*)) *auto*

have [*simp*]: $\bigwedge a.\ cc\text{-}restr\ S\ (CCexp\ e[y::=x]\cdot a) = cc\text{-}restr\ S\ (CCexp\ e\cdot a)$
  by (*rule Let-nonrec*(*4*)) *auto*

have [*simp*]: $\bigwedge a.\ cc\text{-}restr\ S\ (fup\cdot(CCexp\ e[y::=x])\cdot a) = cc\text{-}restr\ S\ (fup\cdot(CCexp\ e)\cdot a)$
  by (*rule fup-ccExp-restr-subst'*) *simp*

have [*simp*]: $fv\ e[y::=x] \cap S = fv\ e \cap S$
  by (*auto simp add*: *fv-subst-eq*)

have [*simp*]:
  $ccNeighbors\ x'\ (CCexp\ exp[y::=x]\cdot a) \cap -\ \{x'\} \cap S = ccNeighbors\ x'\ (CCexp\ exp\cdot a) \cap -\ \{x'\} \cap S$
  apply (*simp only*: *Int-assoc*)
  apply (*subst* (*1 2*) *ccNeighbors-Int-ccrestr*)
  apply (*subst Let-nonrec*(*6*))

apply *auto*[*2*]
apply *rule*
done

have [*simp*]:
ccNeighbors x′ (*CCexp exp*[*y*::=*x*]·*a*) ∩ *S* = *ccNeighbors x′* (*CCexp exp*·*a*) ∩ *S*
apply (*subst* (*1 2*) *ccNeighbors-Int-ccrestr*)
apply (*subst Let-nonrec*(*6*))
apply *auto*[*2*]
apply *rule*
done

show *cc-restr S* (*CCexp* (*let x′ be e in exp*)[*y*::=*x*]·*a*) = *cc-restr S* (*CCexp* (*let x′ be e in exp*)·*a*)
apply (*subst subst-let-be*)
apply *auto*[*2*]
apply (*subst* (*1 2*) *CCexp-simps*(*6*))
apply *fact*+
apply (*simp only*: *cc-restr-cc-delete-twist*)
apply (*rule arg-cong*) **back**
apply (*simp add*: *Diff-eq ccBind-eq ABind-nonrec-eq*)
done

show *Aexp* (*let x′ be e in exp*)[*y*::=*x*]·*a f*|‘ *S* = *Aexp* (*let x′ be e in exp*)·*a f*|‘ *S*
by (*simp add*: *env-restr-join env-delete-env-restr-swap*[*symmetric*] *ABind-nonrec-eq*)
next
case (*IfThenElse scrut e1 e2*)
case [*simp*]: *2*
from *IfThenElse*
show *cc-restr S* (*CCexp* (*scrut ? e1 : e2*)[*y*::=*x*]·*a*) = *cc-restr S* (*CCexp* (*scrut ? e1 : e2*)·*a*)
by (*auto simp del*: *edom-env env-restr-empty env-restr-empty-iff simp add*: *edom-env*[*symmetric*])

from *IfThenElse*(*2*,*4*,*6*)
show *Aexp* (*scrut ? e1 : e2*)[*y*::=*x*]·*a f*|‘ *S* = *Aexp* (*scrut ? e1 : e2*)·*a f*|‘ *S*
by (*auto simp add*: *env-restr-join simp del*: *fun-meet-simp*)
qed *auto*

sublocale *ArityAnalysisSafe Aexp*
by *standard* (*simp-all add*:*Aexp-restr-subst*)

sublocale *ArityAnalysisLetSafe Aexp Aheap*
proof
fix Γ *e a*
show *edom* (*Aheap* Γ *e*·*a*) ⊆ *domA* Γ
by (*cases nonrec* Γ)
(*auto simp add*: *Aheap-nonrec-simp dest*: *set-mp*[*OF edom-esing-subset*] *elim*!: *nonrecE*)
next

fix $x\ y$ :: *var* **and** Γ :: *heap* **and** e :: *exp*
assume *assms*: $x \notin domA\ \Gamma \quad y \notin domA\ \Gamma$

from *Aexp-restr-subst*[*OF assms*(*2*,*1*)]
have $**$: $\bigwedge a.\ Aexp\ e[x::=y] \cdot a\ f|`\ domA\ \Gamma = Aexp\ e \cdot a\ f|`\ domA\ \Gamma$**.**

show $Aheap\ \Gamma[x::h=y]\ e[x::=y] = Aheap\ \Gamma\ e$
proof(*cases nonrec* Γ)
  case [*simp*]: *False*

  from *assms*
  have $atom\ `\ domA\ \Gamma\ \sharp*\ x$ **and** $atom\ `\ domA\ \Gamma\ \sharp*\ y$
    by (*auto simp add*: *fresh-star-at-base image-iff*)
  hence [*simp*]: $\neg\ nonrec\ (\Gamma[x::h=y])$
    by (*simp add*: *nonrec-subst*)

  show *?thesis*
  apply (*rule cfun-eqI*)
  apply *simp*
  apply (*subst Afix-restr-subst*[*OF assms subset-refl*])
  apply (*subst Afix-restr*[*OF subset-refl*]) **back**
  apply (*simp add*: *env-restr-join*)
  apply (*subst* $**$)
  apply *simp*
  done
next
  case *True*

  from *assms*
  have $atom\ `\ domA\ \Gamma\ \sharp*\ x$ **and** $atom\ `\ domA\ \Gamma\ \sharp*\ y$
    by (*auto simp add*: *fresh-star-at-base image-iff*)
  with *True*
  have $*$: $nonrec\ (\Gamma[x::h=y])$ **by** (*simp add*: *nonrec-subst*)

  from *True*
  obtain $x'\ e'$ **where** [*simp*]: $\Gamma = [(x',e')]\ \ x' \notin fv\ e'$ **by** (*auto elim*: *nonrecE*)

  from $*$ **have** [*simp*]: $x' \notin fv\ (e'[x::=y])$
    by (*auto simp add*: *nonrec-def*)

  from *fun-cong*[*OF* $**$, **where** $x = x'$]
  have [*simp*]: $\bigwedge a.\ (Aexp\ e[x::=y] \cdot a)\ x' = (Aexp\ e \cdot a)\ x'$ **by** *simp*

  from *CCexp-subst*[*OF assms*(*2*,*1*)]
  have $\bigwedge a.\ \ cc\text{-}restr\ \{x'\}\ (CCexp\ e[x::=y] \cdot a) = cc\text{-}restr\ \{x'\}\ (CCexp\ e \cdot a)$ **by** *simp*
  from *arg-cong*[**where** $f = \lambda x.\ \ x'\text{--}x' \in x$, *OF this*]
  have [*simp*]: $\bigwedge a.\ x'\text{--}x' \in (CCexp\ e[x::=y] \cdot a) \longleftrightarrow x'\text{--}x' \in (CCexp\ e \cdot a)$ **by** *simp*

  show *?thesis*

apply −
apply (*rule cfun-eqI*)
apply (*auto simp add*: *Aheap-nonrec-simp ABind-nonrec-eq*)
done
qed
next
fix Γ *e a*
show *ABinds* Γ·(*Aheap* Γ *e*·*a*) ⊔ *Aexp e*·*a* ⊑ *Aheap* Γ *e*·*a* ⊔ *Aexp* (*Let* Γ *e*)·*a*
proof(*cases nonrec* Γ)
case *False*
thus *?thesis*
by (*auto simp add*: *Aheap-def join-below-iff env-restr-join2 Compl-partition intro*: *below-trans*[*OF* - *Afix-above-arg*])
next
case *True*
then obtain *x e′* **where** [*simp*]: Γ = [(*x*,*e′*)] *x* ∉ *fv e′* **by** (*auto elim*: *nonrecE*)

hence $\bigwedge$ *a*. *x* ∉ *edom* (*fup*·(*Aexp e′*)·*a*)
by (*auto dest*:*set-mp*[*OF fup-Aexp-edom*])
hence [*simp*]: $\bigwedge$ *a*. (*fup*·(*Aexp e′*)·*a*) *x* = ⊥ **by** (*simp add*: *edomIff*)

show *?thesis*
apply (*rule env-restr-below-split*[**where** *S* = {*x*}])
apply (*rule env-restr-belowI2*)
apply (*auto simp add*: *Aheap-nonrec-simp join-below-iff env-restr-join env-delete-restr*)
apply (*rule ABind-nonrec-above-arg*)
apply (*rule below-trans*[*OF* - *join-above2*])
apply (*rule below-trans*[*OF* - *join-above2*])
apply (*rule below-refl*)
done
qed
qed

definition *ccHeap-nonrec*
where *ccHeap-nonrec x e exp* = (Λ *n*. *CCfix-nonrec x e*·(*Aexp exp*·*n*, *CCexp exp*·*n*))

lemma *ccHeap-nonrec-eq*:
ccHeap-nonrec x e exp·*n* = *CCfix-nonrec x e*·(*Aexp exp*·*n*, *CCexp exp*·*n*)
unfolding *ccHeap-nonrec-def* **by** (*rule beta-cfun*) (*intro cont2cont*)

definition *ccHeap-rec* :: *heap* ⇒ *exp* ⇒ *Arity* → *CoCalls*
where *ccHeap-rec* Γ *e* = (Λ *a*. *CCfix* Γ·(*Afix* Γ·(*Aexp e*·*a* ⊔ (λ-.*up*·*0*) *f*|‘ (*thunks* Γ)), *CCexp e*·*a*))

lemma *ccHeap-rec-eq*:
ccHeap-rec Γ *e*·*a* = *CCfix* Γ·(*Afix* Γ·(*Aexp e*·*a* ⊔ (λ-.*up*·*0*) *f*|‘ (*thunks* Γ)), *CCexp e*·*a*)
unfolding *ccHeap-rec-def* **by** *simp*

definition *ccHeap* :: *heap* ⇒ *exp* ⇒ *Arity* → *CoCalls*

where *ccHeap* Γ = (*if nonrec* Γ *then case-prod ccHeap-nonrec* (*hd* Γ) *else ccHeap-rec* Γ)

lemma *ccHeap-simp1*:
$\neg$ *nonrec* $\Gamma \Longrightarrow$ *ccHeap* Γ $e \cdot a$ = *CCfix* $\Gamma \cdot$(*Afix* $\Gamma \cdot$(*Aexp* $e \cdot a \sqcup (\lambda$-.*up*$\cdot 0)$ *f*|‘ (*thunks* Γ)), *CCexp* $e \cdot a$)
by (*simp add*: *ccHeap-def ccHeap-rec-eq*)

lemma *ccHeap-simp2*:
$x \notin fv\ e \Longrightarrow$ *ccHeap* [(*x*,*e*)] *exp*$\cdot n$ = *CCfix-nonrec* x $e \cdot$(*Aexp exp*$\cdot n$, *CCexp exp*$\cdot n$)
by (*simp add*: *ccHeap-def ccHeap-nonrec-eq nonrec-def*)

sublocale *CoCallAritySafe CCexp Aexp ccHeap Aheap*
proof
  fix *e a x*
  show *CCexp* $e \cdot (inc \cdot a) \sqcup$ *ccProd* $\{x\}$ (*insert* x (*fv* e)) $\sqsubseteq$ *CCexp* (*App e x*)$\cdot a$
    by *simp*
next
  fix *y e n*
  show *cc-restr* (*fv* (*Lam* [*y*]. *e*)) (*CCexp* $e \cdot (pred \cdot n)$) $\sqsubseteq$ *CCexp* (*Lam* [*y*]. *e*)$\cdot n$
    by (*auto simp add*: *CCexp-pre-simps predCC-eq dest*!: *set-mp*[*OF ccField-cc-restr*] *simp del*: *CCexp-simps*)
next
  fix *x y* :: *var* **and** *S e a*
  assume $x \notin S$ **and** $y \notin S$
  thus *cc-restr S* (*CCexp* *e*[*y*::=*x*]$\cdot a$) $\sqsubseteq$ *cc-restr S* (*CCexp* $e \cdot a$)
    by (*rule eq-imp-below*[*OF CCexp-subst*])
next
  fix *e*
  assume *isVal e*
  thus *CCexp* $e \cdot 0$ = *ccSquare* (*fv e*)
    by (*induction e rule*: *isVal.induct*) (*auto simp add*: *predCC-eq*)
next
  fix Γ *e a*
  show *cc-restr* ($-$ *domA* Γ) (*ccHeap* Γ $e \cdot a$) $\sqsubseteq$ *CCexp* (*Let* Γ *e*)$\cdot a$
  proof(*cases nonrec* Γ)
    case *False*
    thus *cc-restr* ($-$ *domA* Γ) (*ccHeap* Γ $e \cdot a$) $\sqsubseteq$ *CCexp* (*Let* Γ *e*)$\cdot a$
      by (*simp add*: *ccHeap-simp1*[*OF False*, *symmetric*] *del*: *cc-restr-join*)
  next
    case *True*
    thus *?thesis*
      by (*auto simp add*: *ccHeap-simp2 Diff-eq elim*!: *nonrecE simp del*: *cc-restr-join*)
  qed
next
  fix Δ :: *heap* **and** *e a*

  show *CCexp* $e \cdot a \sqsubseteq$ *ccHeap* Δ $e \cdot a$
    by (*cases nonrec* Δ)

(*auto simp add*: *ccHeap-simp1 ccHeap-simp2 arg-cong*[*OF CCfix-unroll*, **where** f = *op* $\sqsubseteq$ x **for** x] *elim*!: *nonrecE*)

fix x e' a'
assume *map-of* Δ x = *Some* e'
hence [*simp*]: $x \in$ *domA* Δ **by** (*metis domI dom-map-of-conv-domA*)
assume (*Aheap* Δ $e{\cdot}a$) x = *up*$\cdot a'$
show *CCexp* $e'{\cdot}a'$ $\sqsubseteq$ *ccHeap* Δ $e{\cdot}a$
proof(*cases nonrec* Δ)
case *False*

from ‹(*Aheap* Δ $e{\cdot}a$) x = *up*$\cdot a'$› *False*
have (*Afix* $\Delta\cdot$(*Aexp* $e{\cdot}a$ $\sqcup$ (λ-.*up*$\cdot$*0*)$f|$‘ (*thunks* Δ))) x = *up*$\cdot a'$
by (*simp add*: *Aheap-def*)
hence *CCexp* $e'{\cdot}a'$ $\sqsubseteq$ *ccBind* x $e'\cdot$(*Afix* $\Delta\cdot$(*Aexp* $e{\cdot}a$ $\sqcup$ (λ-.*up*$\cdot$*0*)$f|$‘ (*thunks* Δ)), *CCfix* $\Delta\cdot$(*Afix* $\Delta\cdot$(*Aexp* $e{\cdot}a$ $\sqcup$ (λ-.*up*$\cdot$*0*)$f|$‘ (*thunks* Δ)), *CCexp* $e{\cdot}a$))
by (*auto simp add*: *ccBind-eq dest*: *set-mp*[*OF ccField-CCexp*])
also
have *ccBind* x $e'\cdot$(*Afix* $\Delta\cdot$(*Aexp* $e{\cdot}a$ $\sqcup$ (λ-.*up*$\cdot$*0*)$f|$‘ (*thunks* Δ)), *CCfix* $\Delta\cdot$(*Afix* $\Delta\cdot$(*Aexp* $e{\cdot}a$ $\sqcup$ (λ-.*up*$\cdot$*0*)$f|$‘ (*thunks* Δ)), *CCexp* $e{\cdot}a$)) $\sqsubseteq$ *ccHeap* Δ $e{\cdot}a$
using ‹*map-of* Δ x = *Some* e'› *False*
by (*fastforce simp add*: *ccHeap-simp1 ccHeap-rec-eq ccBindsExtra-simp ccBinds-eq arg-cong*[*OF CCfix-unroll*, **where** f = *op* $\sqsubseteq$ x **for** x]
intro: *below-trans*[*OF - join-above2*])
finally
show *CCexp* $e'{\cdot}a'$ $\sqsubseteq$ *ccHeap* Δ $e{\cdot}a$ **by** *this simp-all*
next
case *True*
with ‹*map-of* Δ x = *Some* e'›
have [*simp*]: Δ = [(x,e')] $x \notin$ *fv* e' **by** (*auto elim*!: *nonrecE split*: *if-splits*)

show *?thesis*
proof(*cases* x−−$x\notin$*CCexp* $e{\cdot}a$ $\vee$ *isVal* e')
case *True*
with ‹(*Aheap* Δ $e{\cdot}a$) x = *up*$\cdot a'$›
have [*simp*]: (*CoCallArityAnalysis.Aexp cCCexp* $e{\cdot}a$) x = *up*$\cdot a'$
by (*auto simp add*: *Aheap-nonrec-simp ABind-nonrec-eq split*: *if-splits*)

have *CCexp* $e'{\cdot}a'$ $\sqsubseteq$ *ccSquare* (*fv* e')
unfolding *below-ccSquare*
by (*rule ccField-CCexp*)
then
show *?thesis* **using** *True*
by (*auto simp add*: *ccHeap-simp2 ccBind-eq Aheap-nonrec-simp ABind-nonrec-eq below-trans*[*OF - join-above2*] *simp del*: *below-ccSquare*)
next
case *False*

from ‹(*Aheap* Δ $e{\cdot}a$) x = *up*$\cdot a'$›

have [*simp*]: $a' = 0$ **using** *False*
by (*auto simp add*: *Aheap-nonrec-simp ABind-nonrec-eq split*: *if-splits*)

show *?thesis* **using** *False*
by (*auto simp add*: *ccHeap-simp2 ccBind-eq Aheap-nonrec-simp ABind-nonrec-eq simp del*: *below-ccSquare*)
qed
qed

show *ccProd* (*fv* e') (*ccNeighbors* x (*ccHeap* Δ $e{\cdot}a$) $-$ $\{x\}$ $\cap$ *thunks* Δ) $\sqsubseteq$ *ccHeap* Δ $e{\cdot}a$
proof (*cases nonrec* Δ)
case [*simp*]: *False*

have *ccProd* (*fv* e') (*ccNeighbors* x (*ccHeap* Δ $e{\cdot}a$) $-$ $\{x\}$ $\cap$ *thunks* Δ) $\sqsubseteq$ *ccProd* (*fv* e') (*ccNeighbors* x (*ccHeap* Δ $e{\cdot}a$))
by (*rule ccProd-mono2*) *auto*
also have ... $\sqsubseteq$ ($\bigsqcup x{\mapsto}e'{\in}$*map-of* Δ. *ccProd* (*fv* e') (*ccNeighbors* x (*ccHeap* Δ $e{\cdot}a$)))
using ‹*map-of* Δ x = *Some* e'› **by** (*rule below-lubmapI*)
also have ... $\sqsubseteq$ *ccBindsExtra* $\Delta{\cdot}$(*Afix* $\Delta{\cdot}$(*Aexp* $e{\cdot}a$ $\sqcup$ (λ-.*up*$\cdot$*0*)$f|$‘ (*thunks* Δ)), *ccHeap* Δ $e{\cdot}a$)
by (*simp add*: *ccBindsExtra-simp below-trans*[*OF* - *join-above2*])
also have ... $\sqsubseteq$ *ccHeap* Δ $e{\cdot}a$
by (*simp add*: *ccHeap-simp1 arg-cong*[*OF CCfix-unroll*, **where** f = *op* $\sqsubseteq$ x **for** x])
finally
show *?thesis* **by** *this simp-all*
next
case *True*
with ‹*map-of* Δ x = *Some* e'›
have [*simp*]: Δ = [(x,e')] $x \notin$ *fv* e' **by** (*auto elim*!: *nonrecE split*: *if-splits*)

have [*simp*]: (*ccNeighbors* x (*ccBind* x $e'{\cdot}$(*Aexp* $e{\cdot}a$, *CCexp* $e{\cdot}a$))) = {}
by (*auto simp add*: *ccBind-eq dest*!: *set-mp*[*OF ccField-cc-restr*] *set-mp*[*OF ccField-fup-CCexp*])

show *?thesis*
proof(*cases isVal* e' $\wedge$ x−−$x{\in}$*CCexp* $e{\cdot}a$)
case *True*

have *ccNeighbors* x (*ccHeap* Δ $e{\cdot}a$) =
ccNeighbors x (*ccBind* x $e'{\cdot}$(*Aheap-nonrec* x $e'{\cdot}$(*Aexp* $e{\cdot}a$, *CCexp* $e{\cdot}a$), *CCexp* $e{\cdot}a$)) $\cup$
ccNeighbors x (*ccProd* (*fv* e') (*ccNeighbors* x (*CCexp* $e{\cdot}a$) $-$ (*if isVal* e' *then* {} *else* $\{x\}$))) $\cup$
ccNeighbors x (*CCexp* $e{\cdot}a$) **by** (*auto simp add*: *ccHeap-simp2*)
also have *ccNeighbors* x (*ccBind* x $e'{\cdot}$(*Aheap-nonrec* x $e'{\cdot}$(*Aexp* $e{\cdot}a$, *CCexp* $e{\cdot}a$), *CCexp* $e{\cdot}a$)) = {}
by (*auto simp add*: *ccBind-eq dest*!: *set-mp*[*OF ccField-cc-restr*] *set-mp*[*OF ccField-fup-CCexp*])
also have *ccNeighbors* x (*ccProd* (*fv* e') (*ccNeighbors* x (*CCexp* $e{\cdot}a$) $-$ (*if isVal* e' *then* {} *else* $\{x\}$)))
$\subseteq$ *ccNeighbors* x (*ccProd* (*fv* e') (*ccNeighbors* x (*CCexp* $e{\cdot}a$))) **by** (*simp add*: *ccNeighbors-ccProd*)
also have ... $\subseteq$ *fv* e' **by** (*simp add*: *ccNeighbors-ccProd*)

finally
have $ccNeighbors\ x\ (ccHeap\ \Delta\ e{\cdot}a) - \{x\} \cap thunks\ \Delta \subseteq ccNeighbors\ x\ (CCexp\ e{\cdot}a) \cup fv\ e'$ **by** *auto*
hence $ccProd\ (fv\ e')\ (ccNeighbors\ x\ (ccHeap\ \Delta\ e{\cdot}a) - \{x\} \cap thunks\ \Delta) \sqsubseteq ccProd\ (fv\ e')\ (ccNeighbors\ x\ (CCexp\ e{\cdot}a) \cup fv\ e')$ **by** (*rule ccProd-mono2*)
also have $\ldots \sqsubseteq ccProd\ (fv\ e')\ (ccNeighbors\ x\ (CCexp\ e{\cdot}a)) \sqcup ccProd\ (fv\ e')\ (fv\ e')$ **by** *simp*
also have $ccProd\ (fv\ e')\ (ccNeighbors\ x\ (CCexp\ e{\cdot}a)) \sqsubseteq ccHeap\ \Delta\ e{\cdot}a$
using ‹$map\text{-}of\ \Delta\ x = Some\ e'$› ‹$(Aheap\ \Delta\ e{\cdot}a)\ x = up{\cdot}a'$› *True*
by (*auto simp add*: *ccHeap-simp2 below-trans*[*OF - join-above2*])
also have $ccProd\ (fv\ e')\ (fv\ e') = ccSquare\ (fv\ e')$ **by** (*simp add*: *ccSquare-def*)
also have $\ldots \sqsubseteq ccHeap\ \Delta\ e{\cdot}a$
using ‹$map\text{-}of\ \Delta\ x = Some\ e'$› ‹$(Aheap\ \Delta\ e{\cdot}a)\ x = up{\cdot}a'$› *True*
by (*auto simp add*: *ccHeap-simp2 ccBind-eq below-trans*[*OF - join-above2*])
also note *join-self*
finally show *?thesis* **by** *this simp-all*
next
case *False*
have $ccNeighbors\ x\ (ccHeap\ \Delta\ e{\cdot}a) =$
$ccNeighbors\ x\ (ccBind\ x\ e'{\cdot}(Aheap\text{-}nonrec\ x\ e'{\cdot}(Aexp\ e{\cdot}a,\ CCexp\ e{\cdot}a),\ CCexp\ e{\cdot}a)) \cup$
$ccNeighbors\ x\ (ccProd\ (fv\ e')\ (ccNeighbors\ x\ (CCexp\ e{\cdot}a) - (\text{if } isVal\ e' \text{ then } \{\} \text{ else } \{x\}))) \cup$
$ccNeighbors\ x\ (CCexp\ e{\cdot}a)$ **by** (*auto simp add*: *ccHeap-simp2*)
also have $ccNeighbors\ x\ (ccBind\ x\ e'{\cdot}(Aheap\text{-}nonrec\ x\ e'{\cdot}(Aexp\ e{\cdot}a,\ CCexp\ e{\cdot}a),\ CCexp\ e{\cdot}a)) = \{\}$
by (*auto simp add*: *ccBind-eq dest*!: *set-mp*[*OF ccField-cc-restr*] *set-mp*[*OF ccField-fup-CCexp*])
also have $ccNeighbors\ x\ (ccProd\ (fv\ e')\ (ccNeighbors\ x\ (CCexp\ e{\cdot}a) - (\text{if } isVal\ e' \text{ then } \{\} \text{ else } \{x\})\ ))$
$= \{\}$ **using** *False* **by** (*auto simp add*: *ccNeighbors-ccProd*)
finally
have $ccNeighbors\ x\ (ccHeap\ \Delta\ e{\cdot}a) \subseteq ccNeighbors\ x\ (CCexp\ e{\cdot}a)$ **by** *auto*
hence$ccNeighbors\ x\ (ccHeap\ \Delta\ e{\cdot}a)\ - \{x\} \cap thunks\ \Delta \subseteq ccNeighbors\ x\ (CCexp\ e{\cdot}a)\ - \{x\} \cap thunks\ \Delta$ **by** *auto*
hence $ccProd\ (fv\ e')\ (ccNeighbors\ x\ (ccHeap\ \Delta\ e{\cdot}a) - \{x\} \cap thunks\ \Delta\ ) \sqsubseteq ccProd\ (fv\ e')\ (ccNeighbors\ x\ (CCexp\ e{\cdot}a)\ - \{x\} \cap thunks\ \Delta\ )$ **by** (*rule ccProd-mono2*)
also have $\ldots \sqsubseteq ccHeap\ \Delta\ e{\cdot}a$
using ‹$map\text{-}of\ \Delta\ x = Some\ e'$› ‹$(Aheap\ \Delta\ e{\cdot}a)\ x = up{\cdot}a'$› *False*
by (*auto simp add*: *ccHeap-simp2 thunks-Cons below-trans*[*OF - join-above2*])
finally show *?thesis* **by** *this simp-all*
qed
qed

next
fix $x\ \Gamma\ e\ a$
assume [*simp*]: $\neg\ nonrec\ \Gamma$
assume $x \in thunks\ \Gamma$
hence [*simp*]: $x \in domA\ \Gamma$ **by** (*rule set-mp*[*OF thunks-domA*])
assume $x \in edom\ (Aheap\ \Gamma\ e{\cdot}a)$

from ‹$x \in$ *thunks* Γ›
have (*Afix* $\Gamma \cdot$(*Aexp* $e \cdot a \sqcup (\lambda\text{-}.up \cdot 0) f|$‘ (*thunks* Γ))) $x = up \cdot 0$
by (*subst Afix-unroll*) *simp*

thus (*Aheap* Γ $e \cdot a$) $x = up \cdot 0$ **by** *simp*
next
fix x Γ e a
assume *nonrec* Γ
then obtain x' e' **where** [*simp*]: $\Gamma = [(x', e')]$ $x' \notin fv\ e'$ **by** (*auto elim*: *nonrecE*)
assume $x \in$ *thunks* Γ
hence [*simp*]: $x = x'$ $\neg$ *isVal* e' **by** (*auto simp add*: *thunks-Cons split*: *if-splits*)

assume $x{-}{-}x \in$ *CCexp* $e \cdot a$
hence [*simp*]: $x'{-}{-}x' \in$ *CCexp* $e \cdot a$ **by** *simp*

from ‹$x \in$ *thunks* Γ›
have (*Afix* $\Gamma \cdot$(*Aexp* $e \cdot a \sqcup (\lambda\text{-}.up \cdot 0) f|$‘ (*thunks* Γ))) $x = up \cdot 0$
by (*subst Afix-unroll*) *simp*

show (*Aheap* Γ $e \cdot a$) $x = up \cdot 0$ **by** (*auto simp add*: *Aheap-nonrec-simp ABind-nonrec-eq*)
next
fix *scrut e1 a e2*
show *CCexp scrut*$\cdot 0 \sqcup$ (*CCexp e1*$\cdot a \sqcup$ *CCexp e2*$\cdot a$) $\sqcup$ *ccProd* (*edom* (*Aexp scrut*$\cdot 0$)) (*edom* (*Aexp e1*$\cdot a$) $\cup$ *edom* (*Aexp e2*$\cdot a$)) $\sqsubseteq$ *CCexp* (*scrut ? e1 : e2*)$\cdot a$
by *simp*
qed
end

end

83 List-Interleavings.tex

theory *List−Interleavings*
imports *Main*
begin

inductive *interleave'* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *bool*
where [*simp*]: *interleave'* [] [] []
| *interleave' xs ys zs* $\Longrightarrow$ *interleave'* (*x#xs*) *ys* (*x#zs*)
| *interleave' xs ys zs* $\Longrightarrow$ *interleave' xs* (*x#ys*) (*x#zs*)

definition *interleave* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list set* (**infixr** $\otimes$ *64*)
where *xs* $\otimes$ *ys* = *Collect* (*interleave' xs ys*)
lemma *elim-interleave'*[*pred-set-conv*]: *interleave' xs ys zs* $\longleftrightarrow$ *zs* $\in$ *xs* $\otimes$ *ys* **unfolding** *interleave-def*
by *simp*

lemmas *interleave-intros*[*intro?*] = *interleave'.intros*[*to-set*]
lemmas *interleave-intros*(*1*)[*simp*]

lemmas *interleave-induct*[*consumes 1*, *induct set*: *interleave*, *case-names Nil left right*] = *interleave'.induct*[*to-set*]
lemmas *interleave-cases*[*consumes 1*, *cases set*: *interleave*] = *interleave'.cases*[*to-set*]
lemmas *interleave-simps* = *interleave'.simps*[*to-set*]

inductive-cases *interleave-ConsE*[*elim*]: $(x\#xs) \in ys \otimes zs$
inductive-cases *interleave-ConsConsE*[*elim*]: $xs \in y\#ys \otimes z\#zs$
inductive-cases *interleave-ConsE2*[*elim*]: $xs \in x\#ys \otimes zs$
inductive-cases *interleave-ConsE3*[*elim*]: $xs \in ys \otimes x\#zs$

lemma *interleave-comm*: $xs \in ys \otimes zs \Longrightarrow xs \in zs \otimes ys$
  by (*induction rule*: *interleave-induct*) (*auto intro*: *interleave-intros*)

lemma *interleave-Nil1*[*simp*]: $[] \otimes xs = \{xs\}$
  by (*induction xs*) (*auto intro*: *interleave-intros elim*: *interleave-cases*)

lemma *interleave-Nil2*[*simp*]: $xs \otimes [] = \{xs\}$
  by (*induction xs*) (*auto intro*: *interleave-intros elim*: *interleave-cases*)

lemma *interleave-nil-simp*[*simp*]: $[] \in xs \otimes ys \longleftrightarrow xs = [] \wedge ys = []$
  by (*auto elim*: *interleave-cases*)

lemma *append-interleave*: $xs\ @\ ys \in xs \otimes ys$
  by (*induction xs*) (*auto intro*: *interleave-intros*)

lemma *interleave-assoc1*: $a \in xs \otimes ys \Longrightarrow b \in a \otimes zs \Longrightarrow \exists\ c.\ c \in ys \otimes zs \wedge\ b \in xs\ \otimes c$
  by (*induction b arbitrary*: *a xs ys zs*)
    (*simp*, *fastforce del*: *interleave-ConsE elim*!: *interleave-ConsE intro*: *interleave-intros*)

lemma *interleave-assoc2*: $a \in ys \otimes zs \Longrightarrow b \in xs \otimes a \Longrightarrow \exists\ c.\ c \in xs \otimes ys \wedge\ b \in c \otimes zs$
  by (*induction b arbitrary*: *a xs ys zs*)
    (*simp*, *fastforce del*: *interleave-ConsE elim*!: *interleave-ConsE intro*: *interleave-intros*)

lemma *interleave-set*: $zs \in xs \otimes ys \Longrightarrow set\ zs = set\ xs \cup set\ ys$
  by(*induction rule*:*interleave-induct*) *auto*

lemma *interleave-tl*: $xs \in ys \otimes zs \Longrightarrow tl\ xs \in tl\ ys \otimes zs \vee tl\ xs \in ys \otimes (tl\ zs)$
  by(*induction rule*:*interleave-induct*) *auto*

lemma *interleave-butlast*: $xs \in ys \otimes zs \Longrightarrow butlast\ xs \in butlast\ ys \otimes zs \vee butlast\ xs \in ys \otimes (butlast\ zs)$
  by (*induction rule*:*interleave-induct*) (*auto intro*: *interleave-intros*)

lemma *interleave-take*: $zs \in xs \otimes ys \Longrightarrow \exists\ n_1\ n_2.\ n = n_1 + n_2 \wedge\ take\ n\ zs \in take\ n_1\ xs \otimes take\ n_2\ ys$
  apply(*induction arbitrary*: *n rule*:*interleave-induct*)
  apply *auto*
  apply *arith*

apply (*case-tac n*, *simp*)
apply (*drule-tac x* = *nat* **in** *meta-spec*)
apply *auto*
apply (*rule-tac x* = *Suc* n_1 **in** *exI*)
apply (*rule-tac x* = n_2 **in** *exI*)
apply (*auto intro*: *interleave-intros*)[*1*]

apply (*case-tac n*, *simp*)
apply (*drule-tac x* = *nat* **in** *meta-spec*)
apply *auto*
apply (*rule-tac x* = n_1 **in** *exI*)
apply (*rule-tac x* = *Suc* n_2 **in** *exI*)
apply (*auto intro*: *interleave-intros*)[*1*]
done

lemma *filter-interleave*: *xs* ∈ *ys* ⊗ *zs* ⟹ *filter P xs* ∈ *filter P ys* ⊗ *filter P zs*
by (*induction rule*: *interleave-induct*) (*auto intro*: *interleave-intros*)

lemma *interleave-filtered*: *xs* ∈ *interleave* (*filter P xs*) (*filter* (λ*x'*. ¬ *P x'*) *xs*)
by (*induction xs*) (*auto intro*: *interleave-intros*)

function *foo* **where**
foo [] [] = *undefined*
| *foo xs* [] = *undefined*
| *foo* [] *ys* = *undefined*
| *foo* (*x*#*xs*) (*y*#*ys*) = *undefined* (*foo xs* (*y*#*ys*)) (*foo* (*x*#*xs*) *ys*)
by *pat-completeness auto*
termination by *lexicographic-order*
lemmas *list-induct2''* = *foo.induct*[*case-names NilNil ConsNil NilCons ConsCons*]

lemma *interleave-filter*:
assumes *xs* ∈ *filter P ys* ⊗ *filter P zs*
obtains *xs'* **where** *xs'* ∈ *ys* ⊗ *zs* **and** *xs* = *filter P xs'*
using *assms*
apply *atomize-elim*
proof(*induction ys zs arbitrary*: *xs rule*: *list-induct2''*)
case *NilNil*
thus *?case* **by** *simp*
next
case (*ConsNil ys xs*)
thus *?case* **by** *auto*
next
case (*NilCons zs xs*)
thus *?case* **by** *auto*
next
case (*ConsCons y ys z zs xs*)
show *?case*
proof(*cases P y*)

```
  case False
    with ConsCons.prems(1)
    have xs ∈ filter P ys ⊗ filter P (z#zs) by simp
    from ConsCons.IH(1)[OF this]
    obtain xs' where xs' ∈ ys ⊗ (z # zs) xs = filter P xs' by auto
    hence y#xs' ∈ y#ys ⊗ z#zs and xs = filter P (y#xs')
      using False by (auto intro: interleave-intros)
    thus ?thesis by blast
  next
 case True
    show ?thesis
    proof(cases P z)
    case False
      with ConsCons.prems(1)
      have xs ∈ filter P (y#ys) ⊗ filter P zs by simp
      from ConsCons.IH(2)[OF this]
      obtain xs' where xs' ∈ y#ys ⊗ zs xs = filter P xs' by auto
      hence z#xs' ∈ y#ys ⊗ z#zs and xs = filter P (z#xs')
        using False by (auto intro: interleave-intros)
      thus ?thesis by blast
    next
    case True
      from ConsCons.prems(1) ‹P y› ‹P z›
      have xs ∈ y # filter P ys  ⊗ z # filter P zs  by simp
      thus ?thesis
      proof(rule interleave-ConsConsE)
        fix xs'
        assume xs = y # xs' and xs' ∈ interleave (filter P ys) (z # filter P zs)
        hence xs' ∈ filter P ys  ⊗ filter P (z#zs) using ‹P z› by simp
        from ConsCons.IH(1)[OF this]
        obtain xs'' where xs'' ∈ ys ⊗ (z # zs) and xs' = filter P xs'' by auto
        hence y#xs'' ∈ y#ys  ⊗ z#zs and y#xs' = filter P (y#xs'')
          using ‹P y› by (auto intro: interleave-intros)
        thus ?thesis using ‹xs = -› by blast
      next
        fix xs'
        assume xs = z # xs' and xs' ∈ y # filter P ys  ⊗ filter P zs
        hence xs' ∈ filter P (y#ys) ⊗ filter P zs using ‹P y› by simp
        from ConsCons.IH(2)[OF this]
        obtain xs'' where xs'' ∈ y#ys ⊗ zs and xs' = filter P xs'' by auto
        hence z#xs'' ∈ y#ys ⊗ z#zs and z#xs' = filter P (z#xs'')
          using ‹P z› by (auto intro: interleave-intros)
        thus ?thesis using ‹xs = -› by blast
      qed
    qed
  qed
qed
```

end

84 TTree.tex

theory *TTree*
imports *Main ConstOn List−Interleavings*
begin

84.1 Prefix-closed sets of lists

definition *downset* :: *'a list set* ⇒ *bool* **where**
downset xss = (∀ *x n*. *x* ∈ *xss* ⟶ *take n x* ∈ *xss*)

lemma *downsetE*[*elim*]:
downset xss ⟹ *xs* ∈ *xss* ⟹ *butlast xs* ∈ *xss*
by (*auto simp add*: *downset-def butlast-conv-take*)

lemma *downset-appendE*[*elim*]:
downset xss ⟹ *xs*@*ys* ∈ *xss* ⟹ *xs* ∈ *xss*
by (*auto simp add*: *downset-def*) (*metis append-eq-conv-conj*)

lemma *downset-hdE*[*elim*]:
downset xss ⟹ *xs* ∈ *xss* ⟹ *xs* ≠ [] ⟹ [*hd xs*] ∈ *xss*
by (*auto simp add*: *downset-def*) (*metis take-0 take-Suc*)

lemma *downsetI*[*intro*]:
assumes ⋀ *xs*. *xs* ∈ *xss* ⟹ *xs* ≠ [] ⟹ *butlast xs* ∈ *xss*
shows *downset xss*
unfolding *downset-def*
proof(*intro impI allI*)
from *assms*
have *butlast*: ⋀ *xs*. *xs* ∈ *xss* ⟹ *butlast xs* ∈ *xss*
by (*metis butlast.simps*(*1*))

fix *xs n*
assume *xs* ∈ *xss*
show *take n xs* ∈ *xss*
proof(*cases n* ≤ *length xs*)
case *True*
from *this*
show *?thesis*
proof(*induction rule*: *inc-induct*)
case *base* **with** ‹*xs* ∈ *xss*› **show** *?case* **by** *simp*
next
case (*step n'*)
from *butlast*[*OF step.IH*] *step*(*2*)
show *?case* **by** (*simp add*: *butlast-take*)

    qed
  next
  case *False* **with** ‹*xs* ∈ *xss*› **show** *?thesis* **by** *simp*
  qed
qed

lemma [*simp*]: *downset* {[]} **by** *auto*

lemma *downset-mapI*: *downset xss* ⟹ *downset* (*map f* ‘ *xss*)
  by (*fastforce simp add*: *map-butlast*[*symmetric*])

lemma *downset-filter*:
  assumes *downset xss*
  shows *downset* (*filter P* ‘ *xss*)
proof(*rule*, *elim imageE*, *clarsimp*)
  fix *xs*
  assume *xs* ∈ *xss*
  thus *butlast* (*filter P xs*) ∈ *filter P* ‘ *xss*
  proof (*induction xs rule*: *rev-induct*)
    case *Nil* **thus** *?case* **by** *force*
  next
    case *snoc*
    thus *?case* **using** ‹*downset xss*› **by** (*auto intro*: *snoc.IH*)
  qed
qed

lemma *downset-set-subset*:
  downset ({*xs*. *set xs* ⊆ *S*})
by (*auto dest*: *in-set-butlastD*)

84.2 The type of infinite labeled trees

typedef *'a ttree* = {*xss* :: *'a list set* . [] ∈ *xss* ∧ *downset xss*} **by** *auto*

setup-lifting *type-definition-ttree*

84.3 Deconstructors

lift-definition *possible* ::*'a ttree* ⇒ *'a* ⇒ *bool*
  is λ *xss x*. ∃ *xs*. *x#xs* ∈ *xss*.

lift-definition *nxt* ::*'a ttree* ⇒ *'a* ⇒ *'a ttree*
  is λ *xss x*. *insert* [] {*xs* | *xs*. *x#xs* ∈ *xss*}
  by (*auto simp add*: *downset-def take-Suc-Cons*[*symmetric*] *simp del*: *take-Suc-Cons*)

84.4 Trees as set of paths

lift-definition *paths* :: *'a ttree* ⇒ *'a list set* **is** (λ *x*. *x*).

lemma *paths-inj*: *paths t* = *paths t'* ⟹ *t* = *t'* **by** *transfer auto*

lemma *paths-injs-simps*[*simp*]: *paths* t = *paths* t' $\longleftrightarrow$ $t = t'$ **by** *transfer auto*

lemma *paths-Nil*[*simp*]: [] $\in$ *paths* t **by** *transfer simp*

lemma *paths-not-empty*[*simp*]: (*paths* t = {}) $\longleftrightarrow$ *False* **by** *transfer auto*

lemma *paths-Cons-nxt*:
possible t x $\Longrightarrow$ *xs* $\in$ *paths* (*nxt* t x) $\Longrightarrow$ (x#*xs*) $\in$ *paths* t
by *transfer auto*

lemma *paths-Cons-nxt-iff*:
possible t x $\Longrightarrow$ *xs* $\in$ *paths* (*nxt* t x) $\longleftrightarrow$ (x#*xs*) $\in$ *paths* t
by *transfer auto*

lemma *possible-mono*:
paths $t \subseteq$ *paths* t' $\Longrightarrow$ *possible* t x $\Longrightarrow$ *possible* t' x
by *transfer auto*

lemma *nxt-mono*:
paths $t \subseteq$ *paths* t' $\Longrightarrow$ *paths* (*nxt* t x) $\subseteq$ *paths* (*nxt* t' x)
by *transfer auto*

lemma *ttree-eqI*: ($\bigwedge$ x *xs*. x#*xs* $\in$ *paths* t $\longleftrightarrow$ x#*xs* $\in$ *paths* t') $\Longrightarrow$ $t = t'$
apply (*rule paths-inj*)
apply (*rule set-eqI*)
apply (*case-tac* x)
apply *auto*
done

lemma *paths-nxt*[*elim*]:
assumes *xs* $\in$ *paths* (*nxt* t x)
obtains x#*xs* $\in$ *paths* t | *xs* = []
using *assms* **by** *transfer auto*

lemma *Cons-path*: x # *xs* $\in$ *paths* t $\longleftrightarrow$ *possible* t x $\wedge$ *xs* $\in$ *paths* (*nxt* t x)
by *transfer auto*

lemma *Cons-pathI*[*intro*]:
assumes *possible* t x $\longleftrightarrow$ *possible* t' x
assumes *possible* t x $\Longrightarrow$ *possible* t' x $\Longrightarrow$ *xs* $\in$ *paths* (*nxt* t x) $\longleftrightarrow$ *xs* $\in$ *paths* (*nxt* t' x)
shows x # *xs* $\in$ *paths* t $\longleftrightarrow$ x # *xs* $\in$ *paths* t'
using *assms* **by** (*auto simp add*: *Cons-path*)

lemma *paths-nxt-eq*: *xs* $\in$ *paths* (*nxt* t x) $\longleftrightarrow$ *xs* = [] $\vee$ x#*xs* $\in$ *paths* t
by *transfer auto*

lemma *ttree-coinduct*:
assumes P t t'

assumes $\bigwedge$ *t t' x* . *P t t'* $\Longrightarrow$ *possible t x* $\longleftrightarrow$ *possible t' x*
assumes $\bigwedge$ *t t' x* . *P t t'* $\Longrightarrow$ *possible t x* $\Longrightarrow$ *possible t' x* $\Longrightarrow$ *P* (*nxt t x*) (*nxt t' x*)
shows *t* = *t'*
proof(*rule paths-inj*, *rule set-eqI*)
fix *xs*
from *assms*(*1*)
show *xs* ∈ *paths t* $\longleftrightarrow$ *xs* ∈ *paths t'*
proof (*induction xs arbitrary*: *t t'*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs t t'*)
show *?case*
proof (*rule Cons-pathI*)
from ‹*P t t'*›
show *possible t x* $\longleftrightarrow$ *possible t' x* **by** (*rule assms*(*2*))
next
assume *possible t x* **and** *possible t' x*
with ‹*P t t'*›
have *P* (*nxt t x*) (*nxt t' x*) **by** (*rule assms*(*3*))
thus *xs* ∈ *paths* (*nxt t x*) $\longleftrightarrow$ *xs* ∈ *paths* (*nxt t' x*) **by** (*rule Cons.IH*)
qed
qed
qed

84.5 The carrier of a tree

lift-definition *carrier* :: *'a ttree* ⇒ *'a set* **is** λ *xss*. ⋃(*set* ‘ *xss*).

lemma *carrier-mono*: *paths t* ⊆ *paths t'* $\Longrightarrow$ *carrier t* ⊆ *carrier t'* **by** *transfer auto*

lemma *carrier-possible*:
possible t x $\Longrightarrow$ *x* ∈ *carrier t* **by** *transfer force*

lemma *carrier-possible-subset*:
carrier t ⊆ *A* $\Longrightarrow$ *possible t x* $\Longrightarrow$ *x* ∈ *A* **by** *transfer force*

lemma *carrier-nxt-subset*:
carrier (*nxt t x*) ⊆ *carrier t*
by *transfer auto*

lemma *Union-paths-carrier*: (⋃*x*∈*paths t*. *set x*) = *carrier t*
by *transfer auto*

84.6 Repeatable trees

definition *repeatable* **where** *repeatable t* = (∀ *x* . *possible t x* ⟶ *nxt t x* = *t*)

lemma *nxt-repeatable*[*simp*]: *repeatable t* $\Longrightarrow$ *possible t x* $\Longrightarrow$ *nxt t x* = *t*
unfolding *repeatable-def* **by** *auto*

84.7 Simple trees

lift-definition *empty* :: *'a ttree* **is** $\{[]\}$ **by** *auto*

lemma *possible-empty*[*simp*]: *possible empty* $x' \longleftrightarrow$ *False*
  by *transfer auto*

lemma *nxt-not-possible*[*simp*]: $\neg$ *possible t x* $\Longrightarrow$ *nxt t x* = *empty*
  by *transfer auto*

lemma *paths-empty*[*simp*]: *paths empty* = $\{[]\}$ **by** *transfer auto*

lemma *carrier-empty*[*simp*]: *carrier empty* = $\{\}$ **by** *transfer auto*

lemma *repeatable-empty*[*simp*]: *repeatable empty* **unfolding** *repeatable-def* **by** *transfer auto*

lift-definition *single* :: *'a* $\Rightarrow$ *'a ttree* **is** λ *x*. $\{[], [x]\}$
  by *auto*

lemma *possible-single*[*simp*]: *possible* (*single x*) $x' \longleftrightarrow x = x'$
  by *transfer auto*

lemma *nxt-single*[*simp*]: *nxt* (*single x*) x' = *empty*
  by *transfer auto*

lemma *carrier-single*[*simp*]: *carrier* (*single y*) = $\{y\}$
  by *transfer auto*

lemma *paths-single*[*simp*]: *paths* (*single x*) = $\{[], [x]\}$
  by *transfer auto*

lift-definition *many-calls* :: *'a* $\Rightarrow$ *'a ttree* **is** λ *x*. *range* (λ *n*. *replicate n x*)
  by (*auto simp add*: *downset-def*)

lemma *possible-many-calls*[*simp*]: *possible* (*many-calls x*) $x' \longleftrightarrow x = x'$
  by *transfer* (*force simp add*: *Cons-replicate-eq*)

lemma *nxt-many-calls*[*simp*]: *nxt* (*many-calls x*) x' = (*if* $x' = x$ *then many-calls x else empty*)
  by *transfer* (*force simp add*: *Cons-replicate-eq*)

lemma *repeatable-many-calls*: *repeatable* (*many-calls x*)
  unfolding *repeatable-def* **by** *auto*

lemma *carrier-many-calls*[*simp*]: *carrier* (*many-calls x*) = $\{x\}$ **by** *transfer auto*

lift-definition *anything* :: *'a ttree* **is** *UNIV*
  by *auto*

lemma *possible-anything*[*simp*]: *possible anything* $x' \longleftrightarrow$ *True*

by *transfer auto*

lemma *nxt-anything*[*simp*]: *nxt anything x* = *anything*
by *transfer auto*

lemma *paths-anything*[*simp*]:
paths anything = *UNIV* **by** *transfer auto*

lemma *carrier-anything*[*simp*]:
carrier anything = *UNIV*
apply (*auto simp add*: *Union-paths-carrier*[*symmetric*])
apply (*rule-tac x* = [*x*] **in** *exI*)
apply *simp*
done

lift-definition *many-among* :: $'a$ *set* ⇒ $'a$ *ttree* **is** λ *S*. {*xs* . *set xs* ⊆ *S*}
by (*auto intro*: *downset-set-subset*)

lemma *carrier-many-among*[*simp*]: *carrier* (*many-among S*) = *S*
by *transfer* (*auto*, *metis List.set-insert bot.extremum insertCI insert-subset list.set*(*1*))

84.8 Intersection of two trees

lift-definition *intersect* :: $'a$ *ttree* ⇒ $'a$ *ttree* ⇒ $'a$ *ttree* (**infixl** ∩∩ *80*)
is *op* ∩
by (*auto simp add*: *downset-def*)

lemma *paths-intersect*[*simp*]: *paths* (*t* ∩∩ *t'*) = *paths t* ∩ *paths t'*
by *transfer auto*

lemma *carrier-intersect*: *carrier* (*t* ∩∩ *t'*) ⊆ *carrier t* ∩ *carrier t'*
unfolding *Union-paths-carrier*[*symmetric*]
by *auto*

84.9 Disjoint union of trees

lift-definition *either* :: $'a$ *ttree* ⇒ $'a$ *ttree* ⇒ $'a$ *ttree* (**infixl** ⊕⊕ *80*)
is *op* ∪
by (*auto simp add*: *downset-def*)

lemma *either-empty1*[*simp*]: *empty* ⊕⊕ *t* = *t*
by *transfer auto*
lemma *either-empty2*[*simp*]: *t* ⊕⊕ *empty* = *t*
by *transfer auto*
lemma *either-sym*[*simp*]: *t* ⊕⊕ *t2* = *t2* ⊕⊕ *t*
by *transfer auto*
lemma *either-idem*[*simp*]: *t* ⊕⊕ *t* = *t*
by *transfer auto*

lemma *possible-either*[*simp*]: *possible* (*t* ⊕⊕ *t'*) *x* ⟷ *possible t x* ∨ *possible t' x*

by *transfer auto*

lemma *nxt-either*[*simp*]: $nxt\ (t \oplus\oplus\ t')\ x = nxt\ t\ x \oplus\oplus\ nxt\ t'\ x$
by *transfer auto*

lemma *paths-either*[*simp*]: $paths\ (t \oplus\oplus\ t') = paths\ t \cup paths\ t'$
by *transfer simp*

lemma *carrier-either*[*simp*]:
$carrier\ (t \oplus\oplus\ t') = carrier\ t \cup carrier\ t'$
by *transfer simp*

lemma *either-contains-arg1*: $paths\ t \subseteq paths\ (t \oplus\oplus\ t')$
by *transfer fastforce*

lemma *either-contains-arg2*: $paths\ t' \subseteq paths\ (t \oplus\oplus\ t')$
by *transfer fastforce*

lift-definition *Either* :: $'a\ ttree\ set \Rightarrow\ 'a\ ttree$ **is** $\lambda\ S.\ insert\ []\ (\bigcup S)$
by (*auto simp add*: *downset-def*)

lemma *paths-Either*: $paths\ (Either\ ts) = insert\ []\ (\bigcup (paths\ `\ ts))$
by *transfer auto*

84.10 Merging of trees

lemma *ex-ex-eq-hint*: $(\exists\,x.\ (\exists\,xs\ ys.\ x = f\ xs\ ys \wedge P\ xs\ ys) \wedge Q\ x) \longleftrightarrow (\exists\,xs\ ys.\ Q\ (f\ xs\ ys) \wedge P\ xs\ ys)$
by *auto*

lift-definition *both* :: $'a\ ttree \Rightarrow\ 'a\ ttree \Rightarrow\ 'a\ ttree$ (**infixl** $\otimes\otimes$ *86*)
is $\lambda\ xss\ yss\ .\ \bigcup\ \{xs \otimes ys\ |\ xs\ ys.\ xs \in xss \wedge ys \in yss\}$
by (*force simp*: *ex-ex-eq-hint dest*: *interleave-butlast*)

lemma *both-assoc*[*simp*]: $t \otimes\otimes\ (t' \otimes\otimes\ t'') = (t \otimes\otimes\ t') \otimes\otimes\ t''$
apply *transfer*
apply *auto*
apply (*metis interleave-assoc2*)
apply (*metis interleave-assoc1*)
done

lemma *both-comm*: $t \otimes\otimes\ t' = t' \otimes\otimes\ t$
by *transfer* (*auto*, (*metis interleave-comm*)+)

lemma *both-empty1*[*simp*]: $empty \otimes\otimes\ t = t$
by *transfer auto*

lemma *both-empty2*[*simp*]: $t \otimes\otimes\ empty = t$
by *transfer auto*

lemma *paths-both*: $xs \in paths\ (t \otimes\otimes t') \longleftrightarrow (\exists\ ys \in paths\ t.\ \exists\ zs \in paths\ t'.\ xs \in ys \otimes zs)$
  by *transfer fastforce*

lemma *both-contains-arg1*: $paths\ t \subseteq paths\ (t \otimes\otimes t')$
  by *transfer fastforce*

lemma *both-contains-arg2*: $paths\ t' \subseteq paths\ (t \otimes\otimes t')$
  by *transfer fastforce*

lemma *both-mono1*:
  $paths\ t \subseteq paths\ t' \Longrightarrow paths\ (t \otimes\otimes t'') \subseteq paths\ (t' \otimes\otimes t'')$
  by *transfer auto*

lemma *both-mono2*:
  $paths\ t \subseteq paths\ t' \Longrightarrow paths\ (t'' \otimes\otimes t) \subseteq paths\ (t'' \otimes\otimes t')$
  by *transfer auto*

lemma *possible-both*[*simp*]: $possible\ (t \otimes\otimes t')\ x \longleftrightarrow possible\ t\ x \lor possible\ t'\ x$
proof
  assume $possible\ (t \otimes\otimes t')\ x$
  then obtain *xs* **where** $x\#xs \in paths\ (t \otimes\otimes t')$
    by *transfer auto*

  from ‹$x\#xs \in paths\ (t \otimes\otimes t')$›
  obtain *ys zs* **where** $ys \in paths\ t$ **and** $zs \in paths\ t'$ **and** $x\#xs \in ys \otimes zs$
    by *transfer auto*

  from ‹$x\#xs \in ys \otimes zs$›
  have $ys \neq [] \land hd\ ys = x\ \lor zs \neq [] \land hd\ zs = x$
    by (*auto elim*: *interleave-cases*)
  thus $possible\ t\ x \lor possible\ t'\ x$
    using ‹$ys \in paths\ t$› ‹$zs \in paths\ t'$›
    by *transfer auto*
next
  assume $possible\ t\ x \lor possible\ t'\ x$
  then obtain *xs* **where** $x\#xs\in paths\ t \lor x\#xs\in paths\ t'$
    by *transfer auto*
   from *this* **have** $x\#xs \in paths\ (t \otimes\otimes t')$ **by** (*auto dest*: *set-mp*[*OF both-contains-arg1*] *set-mp*[*OF both-contains-arg2*])
  thus $possible\ (t \otimes\otimes t')\ x$ **by** *transfer auto*
qed

lemma *nxt-both*:
  $nxt\ (t' \otimes\otimes t)\ x = (if\ possible\ t'\ x \land possible\ t\ x\ then\ nxt\ t'\ x \otimes\otimes t \oplus\oplus t' \otimes\otimes nxt\ t\ x\ else$
  $if\ possible\ t'\ x\ then\ nxt\ t'\ x \otimes\otimes t\ else$
  $if\ possible\ t\ x\ then\ t' \otimes\otimes nxt\ t\ x\ else$
  $empty)$
  by (*transfer*, *auto 4 4 intro*: *interleave-intros*)

lemma *Cons-both*:
$x \mathbin{\#} xs \in paths\ (t' \otimes\otimes\ t) \longleftrightarrow (if\ possible\ t'\ x \wedge possible\ t\ x\ then\ xs \in paths\ (nxt\ t'\ x \otimes\otimes\ t) \vee xs \in paths\ (t' \otimes\otimes\ nxt\ t\ x)\ else$
$if\ possible\ t'\ x\ then\ xs \in paths\ (nxt\ t'\ x \otimes\otimes\ t)\ else$
$if\ possible\ t\ x\ then\ xs \in paths\ (t' \otimes\otimes\ nxt\ t\ x)\ else$
$False)$
apply (*auto simp add*: *paths-Cons-nxt-iff*[*symmetric*] *nxt-both*)
by (*metis paths.rep-eq possible.rep-eq possible-both*)

lemma *Cons-both-possible-leftE*: $possible\ t\ x \Longrightarrow xs \in paths\ (nxt\ t\ x \otimes\otimes\ t') \Longrightarrow x\#xs \in paths\ (t \otimes\otimes\ t')$
by (*auto simp add*: *Cons-both*)
lemma *Cons-both-possible-rightE*: $possible\ t'\ x \Longrightarrow xs \in paths\ (t \otimes\otimes\ nxt\ t'\ x) \Longrightarrow x\#xs \in paths\ (t \otimes\otimes\ t')$
by (*auto simp add*: *Cons-both*)

lemma *either-both-distr*[*simp*]:
$t' \otimes\otimes\ t \oplus\oplus\ t' \otimes\otimes\ t'' = t' \otimes\otimes\ (t \oplus\oplus\ t'')$
by *transfer auto*

lemma *either-both-distr2*[*simp*]:
$t' \otimes\otimes\ t \oplus\oplus\ t'' \otimes\otimes\ t = (t' \oplus\oplus\ t'') \otimes\otimes\ t$
by *transfer auto*

lemma *nxt-both-repeatable*[*simp*]:
assumes [*simp*]: *repeatable* t'
assumes [*simp*]: *possible* $t'\ x$
shows $nxt\ (t' \otimes\otimes\ t)\ x = t' \otimes\otimes\ (t \oplus\oplus\ nxt\ t\ x)$
by (*auto simp add*: *nxt-both*)

lemma *nxt-both-many-calls*[*simp*]: $nxt\ (many\text{-}calls\ x \otimes\otimes\ t)\ x = many\text{-}calls\ x \otimes\otimes\ (t\ \oplus\oplus\ nxt\ t\ x)$
by (*simp add*: *repeatable-many-calls*)

lemma *repeatable-both-self*[*simp*]:
assumes [*simp*]: *repeatable t*
shows $t \otimes\otimes\ t = t$
apply (*intro paths-inj set-eqI*)
apply (*induct-tac x*)
apply (*auto simp add*: *Cons-both paths-Cons-nxt-iff*[*symmetric*])
apply (*metis Cons-both both-empty1 possible-empty*)+
done

lemma *repeatable-both-both*[*simp*]:
assumes *repeatable t*
shows $t \otimes\otimes\ t' \otimes\otimes\ t = t \otimes\otimes\ t'$
by (*metis repeatable-both-self*[*OF assms*] *both-assoc both-comm*)

lemma *repeatable-both-both2*[*simp*]:
  assumes *repeatable t*
  shows $t' \otimes\otimes\ t \otimes\otimes\ t = t' \otimes\otimes\ t$
  by (*metis repeatable-both-self*[*OF assms*] *both-assoc both-comm*)

lemma *repeatable-both-nxt*:
  assumes *repeatable t*
  assumes *possible* t' x
  assumes $t' \otimes\otimes\ t = t'$
  shows *nxt* t' $x \otimes\otimes\ t =$ *nxt* t' x
proof(*rule classical*)
  assume *nxt* t' $x \otimes\otimes\ t \neq$ *nxt* t' x
  hence (*nxt* t' $x \oplus\oplus\ t'$) $\otimes\otimes\ t \neq$ *nxt* t' x **by** (*metis* (*no-types*) *assms*(*1*) *both-assoc repeatable-both-self*)
  thus *nxt* t' $x \otimes\otimes\ t =$ *nxt* t' x **by** (*metis* (*no-types*) *assms either-both-distr2 nxt-both nxt-repeatable*)
qed

lemma *repeatable-both-both-nxt*:
  assumes $t' \otimes\otimes\ t = t'$
  shows $t' \otimes\otimes\ t'' \otimes\otimes\ t = t' \otimes\otimes\ t''$
  by (*metis assms both-assoc both-comm*)

lemma *carrier-both*[*simp*]:
  carrier ($t \otimes\otimes\ t'$) = *carrier* t $\cup$ *carrier* t'
proof−
  {
  fix x
  assume $x \in$ *carrier* ($t \otimes\otimes\ t'$)
  then obtain *xs* **where** *xs* $\in$ *paths* ($t \otimes\otimes\ t'$) **and** $x \in$ *set xs* **by** *transfer auto*
  then obtain *ys zs* **where** *ys* $\in$ *paths* t **and** *zs* $\in$ *paths* t' **and** *xs* $\in$ *interleave ys zs*
    by (*auto simp add*: *paths-both*)
  from *this*(*3*) **have** *set xs* = *set ys* $\cup$ *set zs* **by** (*rule interleave-set*)
  with ‹*ys* ∈ -› ‹*zs* ∈ -› ‹$x \in$ *set xs*›
  have $x \in$ *carrier* t $\cup$ *carrier* t' **by** *transfer auto*
  }
  moreover
  note *set-mp*[*OF carrier-mono*[*OF both-contains-arg1*[**where** t=t **and** $t' = t'$]]]
      set-mp[*OF carrier-mono*[*OF both-contains-arg2*[**where** t=t **and** $t' = t'$]]]
  ultimately
  show *?thesis* **by** *auto*
qed

84.11 Removing elements from a tree

lift-definition *without* :: $'a \Rightarrow 'a$ *ttree* $\Rightarrow 'a$ *ttree*
  is λ *x xss*. *filter* (λ x'. $x' \neq x$) ‘ *xss*
  by (*auto intro*: *downset-filter*)(*metis filter.simps*(*1*) *imageI*)

lemma *paths-withoutI*:
  assumes $xs \in paths\ t$
  assumes $x \notin set\ xs$
  shows $xs \in paths\ (without\ x\ t)$
proof−
  from *assms*(*2*)
  have $filter\ (\lambda\ x'.\ x' \neq x)\ xs = xs$ **by** (*auto simp add*: *filter-id-conv*)
  with *assms*(*1*)
  have $xs \in filter\ (\lambda\ x'.\ x' \neq x)`\ paths\ t$ **by** (*metis imageI*)
  thus *?thesis* **by** *transfer*
qed

lemma *carrier-without*[*simp*]: $carrier\ (without\ x\ t) = carrier\ t - \{x\}$
  by *transfer auto*

lift-definition *ttree-restr* :: $'a\ set \Rightarrow 'a\ ttree \Rightarrow 'a\ ttree$ **is** $\lambda\ S\ xss.\ filter\ (\lambda\ x'.\ x' \in S)\ `\ xss$
  by (*auto intro*: *downset-filter*)(*metis filter.simps*(*1*) *imageI*)

lemma *filter-paths-conv-free-restr*:
  $filter\ (\lambda\ x'\ .\ x' \in S)\ `\ paths\ t = paths\ (ttree\text{-}restr\ S\ t)$ **by** *transfer auto*

lemma *filter-paths-conv-free-restr2*:
  $filter\ (\lambda\ x'\ .\ x' \notin S)\ `\ paths\ t = paths\ (ttree\text{-}restr\ (-\ S)\ t)$ **by** *transfer auto*

lemma *filter-paths-conv-free-without*:
  $filter\ (\lambda\ x'\ .\ x' \neq y)\ `\ paths\ t = paths\ (without\ y\ t)$ **by** *transfer auto*

lemma *ttree-restr-is-empty*: $carrier\ t \cap S = \{\} \Longrightarrow ttree\text{-}restr\ S\ t = empty$
  apply *transfer*
  apply (*auto del*: *iffI*)
  apply (*metis SUP-bot-conv*(*2*) *SUP-inf inf-commute inter-set-filter set-empty*)
  apply *force*
  done

lemma *ttree-restr-noop*: $carrier\ t \subseteq S \Longrightarrow ttree\text{-}restr\ S\ t = t$
  apply *transfer*
  apply (*auto simp add*: *image-iff*)
  apply (*metis SUP-le-iff contra-subsetD filter-True*)
  apply (*rule-tac* $x = x$ **in** *bexI*)
  apply (*metis SUP-upper contra-subsetD filter-True*)
  apply *assumption*
  done

lemma *ttree-restr-tree-restr*[*simp*]:
  $ttree\text{-}restr\ S\ (ttree\text{-}restr\ S'\ t) = ttree\text{-}restr\ (S' \cap S)\ t$
  by *transfer* (*simp add*: *image-comp comp-def*)

lemma *ttree-restr-both*:

ttree-restr S $(t \otimes\otimes t') =$ *ttree-restr S t* $\otimes\otimes$ *ttree-restr S* t'
by (*force simp add*: *paths-both filter-paths-conv-free-restr*[*symmetric*] *intro*: *paths-inj filter-interleave elim*: *interleave-filter*)

lemma *ttree-restr-nxt-subset*: $x \in S \Longrightarrow$ *paths* (*ttree-restr S* (*nxt t x*)) $\subseteq$ *paths* (*nxt* (*ttree-restr S t*) *x*)
by *transfer* (*force simp add*: *image-iff*)

lemma *ttree-restr-nxt-subset2*: $x \notin S \Longrightarrow$ *paths* (*ttree-restr S* (*nxt t x*)) $\subseteq$ *paths* (*ttree-restr S t*)
apply *transfer*
apply *auto*
apply *force*
by (*metis filter.simps*(*2*) *imageI*)

lemma *ttree-restr-possible*: $x \in S \Longrightarrow$ *possible t x* $\Longrightarrow$ *possible* (*ttree-restr S t*) *x*
by *transfer force*

lemma *ttree-restr-possible2*: *possible* (*ttree-restr S* t') $x \Longrightarrow x \in S$
by *transfer* (*auto*, *metis filter-eq-Cons-iff*)

lemma *carrier-ttree-restr*[*simp*]:
carrier (*ttree-restr S t*) $= S \cap$ *carrier t*
by *transfer auto*

84.12 Multiple variables, each called at most once

lift-definition *singles* :: *'a set* $\Rightarrow$ *'a ttree* **is** λ S. $\{xs. \forall\ x \in S.$ *length* (*filter* (λ x'. $x' = x$) *xs*) $\leq 1\}$
apply *auto*
apply (*rule downsetI*)
apply *auto*
apply (*subst* (*asm*) *append-butlast-last-id*[*symmetric*]) **back**
apply *simp*
apply (*subst* (*asm*) *filter-append*)
apply *auto*
done

lemma *possible-singles*[*simp*]: *possible* (*singles S*) *x*
apply *transfer'*
apply (*rule-tac* $x = []$ **in** *exI*)
apply *auto*
done

lemma *length-filter-mono*[*intro*]:
assumes ($\bigwedge$ x. $P\ x \Longrightarrow Q\ x$)
shows *length* (*filter P xs*) $\leq$ *length* (*filter Q xs*)
by (*induction xs*) (*auto dest*: *assms*)

lemma *nxt-singles*[*simp*]: *nxt* (*singles* S) x' = (*if* $x' \in S$ *then without* x' (*singles* S) *else singles* S)
  apply *transfer'*
  apply *auto*
  apply (*rule rev-image-eqI*[**where** $x = []$], *auto*)[*1*]
  apply (*rule-tac* $x = x$ **in** *rev-image-eqI*)
  apply (*simp*, *rule ballI*, *erule-tac* $x = xa$ **in** *ballE*, *auto*)[*1*]
  apply (*rule sym*)
  apply (*simp add*: *filter-id-conv filter-empty-conv*)[*1*]
  apply (*erule-tac* $x = xb$ **in** *ballE*)
  apply (*erule order-trans*[*rotated*])
  apply (*rule length-filter-mono*)
  apply *auto*
  done

lemma *carrier-singles*[*simp*]:
  carrier (*singles* S) = *UNIV*
  apply *transfer*
  apply *auto*
  apply (*rule-tac* $x = [x]$ **in** *exI*)
  apply *auto*
  done

lemma *singles-mono*:
  $S \subseteq S' \Longrightarrow$ *paths* (*singles* S') $\subseteq$ *paths* (*singles* S)
by *transfer auto*

lemma *paths-many-calls-subset*:
  paths $t \subseteq$ *paths* (*many-calls* x $\otimes\otimes$ *without* x t)
proof
  fix *xs*
  assume $xs \in$ *paths* t

  have *filter* ($\lambda x'. x' = x$) *xs* = *replicate* (*length* (*filter* ($\lambda x'. x' = x$) *xs*)) x
    by (*induction xs*) *auto*
  hence *filter* ($\lambda x'. x' = x$) $xs \in$ *paths* (*many-calls* x) **by** *transfer auto*
  moreover
  from ‹$xs \in$ *paths* t›
  have *filter* ($\lambda x'. x' \neq x$) $xs \in$ *paths* (*without* x t) **by** *transfer auto*
  moreover
  have $xs \in$ *interleave* (*filter* ($\lambda x'. x' = x$) *xs*) (*filter* ($\lambda x'. x' \neq x$) *xs*) **by** (*rule interleave-filtered*)
  ultimately show $xs \in$ *paths* (*many-calls* x $\otimes\otimes$ *without* x t) **by** *transfer auto*
qed

84.13 Substituting trees for every node

definition *f-nxt* :: ($'a \Rightarrow 'a$ *ttree*) $\Rightarrow 'a$ *set* $\Rightarrow 'a \Rightarrow$ ($'a \Rightarrow 'a$ *ttree*)
  where *f-nxt* f T x = (*if* $x \in T$ *then* $f(x{:=}empty)$ *else* f)

fun *substitute′* :: ($'a \Rightarrow 'a$ *ttree*) $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *ttree* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *bool* **where**
substitute′-Nil: *substitute′ f T t* [] $\longleftrightarrow$ *True*
| *substitute′-Cons*: *substitute′ f T t* ($x\#xs$) $\longleftrightarrow$
possible t x $\wedge$ *substitute′* (*f-nxt f T x*) *T* (*nxt t x* $\otimes\otimes$ *f x*) *xs*

lemma *f-nxt-mono1*: ($\bigwedge$ *x*. *paths* (*f x*) $\subseteq$ *paths* (f' *x*)) $\Longrightarrow$ *paths* (*f-nxt f T x x′*) $\subseteq$ *paths* (*f-nxt f′ T x x′*)
unfolding *f-nxt-def* **by** *auto*

lemma *f-nxt-empty-set*[*simp*]: *f-nxt f* {} *x* = *f* **by** (*simp add*: *f-nxt-def*)

lemma *downset-substitute*: *downset* (*Collect* (*substitute′ f T t*))
apply (*rule*) **unfolding** *mem-Collect-eq*
proof−
fix *x*
assume *substitute′ f T t x*
thus *substitute′ f T t* (*butlast x*) **by**(*induction t x rule*: *substitute′.induct*) (*auto*)
qed

lift-definition *substitute* :: ($'a \Rightarrow 'a$ *ttree*) $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *ttree* $\Rightarrow$ $'a$ *ttree*
is λ *f T t*. *Collect* (*substitute′ f T t*)
by (*simp add*: *downset-substitute*)

lemma *elim-substitute′*[*pred-set-conv*]: *substitute′ f T t xs* $\longleftrightarrow$ *xs* $\in$ *paths* (*substitute f T t*) **by** *transfer auto*

lemmas *substitute-induct*[*case-names Nil Cons*] = *substitute′.induct*
lemmas *substitute-simps*[*simp*] = *substitute′.simps*[*unfolded elim-substitute′*]

lemma *substitute-mono2*:
assumes *paths t* $\subseteq$ *paths t′*
shows *paths* (*substitute f T t*) $\subseteq$ *paths* (*substitute f T t′*)
proof
fix *xs*
assume *xs* $\in$ *paths* (*substitute f T t*)
thus *xs* $\in$ *paths* (*substitute f T t′*)
using *assms*
proof(*induction xs arbitrary*:*f t t′*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons.prems*
show *?case*
by (*auto dest*: *possible-mono elim*: *Cons.IH intro*!: *both-mono1 nxt-mono*)
qed
qed

lemma *substitute-mono1*:
assumes $\bigwedge x$. *paths* $(f\ x) \subseteq$ *paths* $(f'\ x)$
shows *paths* (*substitute* $f\ T\ t$) $\subseteq$ *paths* (*substitute* $f'\ T\ t$)
proof
fix *xs*
assume *xs* $\in$ *paths* (*substitute* $f\ T\ t$)
from *this assms*
show *xs* $\in$ *paths* (*substitute* $f'\ T\ t$)
proof (*induction xs arbitrary*: $f\ f'\ t$)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons.prems*
show *?case*
by (*auto elim*!: *Cons.IH dest*: *set-mp dest*!: *set-mp*[*OF f-nxt-mono1*[*OF Cons.prems*(*2*)]]
set-mp[*OF substitute-mono2*[*OF both-mono2*[*OF Cons.prems*(*2*)]]])
qed
qed

lemma *substitute-monoT*:
assumes $T \subseteq T'$
shows *paths* (*substitute* $f\ T'\ t$) $\subseteq$ *paths* (*substitute* $f\ T\ t$)
proof
fix *xs*
assume *xs* $\in$ *paths* (*substitute* $f\ T'\ t$)
thus *xs* $\in$ *paths* (*substitute* $f\ T\ t$)
using *assms*
proof(*induction* $f\ T'\ t$ *xs arbitrary*: T *rule*: *substitute-induct*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons* $f\ T'\ t\ x$ *xs* T)
from ‹x # *xs* $\in$ *paths* (*substitute* $f\ T'\ t$)›
have [*simp*]: *possible* $t\ x$ **and** *xs* $\in$ *paths* (*substitute* (*f-nxt* $f\ T'\ x$) T' (*nxt* $t\ x \otimes\otimes f\ x$)) **by**
auto
from *Cons.IH*[*OF this*(*2*) *Cons.prems*(*2*)]
have *xs* $\in$ *paths* (*substitute* (*f-nxt* $f\ T'\ x$) T (*nxt* $t\ x \otimes\otimes f\ x$)).
hence *xs* $\in$ *paths* (*substitute* (*f-nxt* $f\ T\ x$) T (*nxt* $t\ x \otimes\otimes f\ x$))
by (*rule set-mp*[*OF substitute-mono1*, *rotated*])
(*auto simp add*: *f-nxt-def set-mp*[*OF Cons.prems*(*2*)])
thus *?case* **by** *auto*
qed
qed

lemma *substitute-contains-arg*: *paths* $t \subseteq$ *paths* (*substitute* $f\ T\ t$)
proof
fix *xs*

show *xs* ∈ *paths t* ⟹ *xs* ∈ *paths* (*substitute f T t*)
proof (*induction xs arbitrary*: *f t*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
from ‹*x* # *xs* ∈ *paths t*›
have *possible t x* **by** *transfer auto*
moreover
from ‹*x* # *xs* ∈ *paths t*› **have** *xs* ∈ *paths* (*nxt t x*)
by (*auto simp add*: *paths-nxt-eq*)
hence *xs* ∈ *paths* (*nxt t x* ⊗⊗ *f x*) **by** (*rule set-mp*[*OF both-contains-arg1*])
note *Cons.IH*[*OF this*]
ultimately
show *?case* **by** *simp*
qed
qed

lemma *possible-substitute*[*simp*]: *possible* (*substitute f T t*) *x* ⟷ *possible t x*
by (*metis Cons-both both-empty2 paths-Nil substitute-simps*(*2*))

lemma *nxt-substitute*[*simp*]: *possible t x* ⟹ *nxt* (*substitute f T t*) *x* = *substitute* (*f-nxt f T x*) *T* (*nxt t x* ⊗⊗ *f x*)
by (*rule ttree-eqI*) (*simp add*: *paths-nxt-eq*)

lemma *substitute-either*: *substitute f T* (*t* ⊕⊕ *t'*) = *substitute f T t* ⊕⊕ *substitute f T t'*
proof−
have [*simp*]: ⋀ *t t' x* . (*nxt t x* ⊕⊕ *nxt t' x*) ⊗⊗ *f x* = *nxt t x* ⊗⊗ *f x* ⊕⊕ *nxt t' x* ⊗⊗ *f x* **by** (*metis both-comm either-both-distr*)
{
fix *xs*
have *xs* ∈ *paths* (*substitute f T* (*t* ⊕⊕ *t'*)) ⟷ *xs* ∈ *paths* (*substitute f T t*) ∨ *xs* ∈ *paths* (*substitute f T t'*)
proof (*induction xs arbitrary*: *f t t'*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
note *IH* = *Cons.IH*[**where** *f* = *f-nxt f T x* **and** *t* = *nxt t' x* ⊗⊗ *f x* **and** *t'* = *nxt t x* ⊗⊗ *f x*]
show *?case*
apply (*auto simp del*: *either-both-distr2 simp add*: *either-both-distr2*[*symmetric*] *IH*)
apply (*metis IH both-comm either-both-distr either-empty2 nxt-not-possible*)
apply (*metis IH both-comm both-empty1 either-both-distr either-empty1 nxt-not-possible*)
done
qed
}
thus *?thesis* **by** (*auto intro*: *paths-inj*)
qed

lemma *f-nxt-T-delete*:
  assumes $f\ x = empty$
  shows $f\text{-}nxt\ f\ (T - \{x\})\ x' = f\text{-}nxt\ f\ T\ x'$
using *assms*
by (*auto simp add*: *f-nxt-def*)

lemma *f-nxt-empty*[*simp*]:
  assumes $f\ x = empty$
  shows $f\text{-}nxt\ f\ T\ x'\ x = empty$
using *assms*
by (*auto simp add*: *f-nxt-def*)

lemma *f-nxt-empty'*[*simp*]:
  assumes $f\ x = empty$
  shows $f\text{-}nxt\ f\ T\ x = f$
using *assms*
by (*auto simp add*: *f-nxt-def*)

lemma *substitute-T-delete*:
  assumes $f\ x = empty$
  shows $substitute\ f\ (T - \{x\})\ t = substitute\ f\ T\ t$
proof (*intro paths-inj set-eqI*)
  fix *xs*
  from *assms*
  show $xs \in paths\ (substitute\ f\ (T - \{x\})\ t) \longleftrightarrow xs \in paths\ (substitute\ f\ T\ t)$
  by (*induction xs arbitrary*: *f t*) (*auto simp add*: *f-nxt-T-delete*)
qed

lemma *substitute-only-empty*:
  assumes *const-on f* (*carrier t*) *empty*
  shows $substitute\ f\ T\ t = t$
proof (*intro paths-inj set-eqI*)
  fix *xs*
  from *assms*
  show $xs \in paths\ (substitute\ f\ T\ t) \longleftrightarrow xs \in paths\ t$
  proof (*induction xs arbitrary*: *f t*)
  case *Nil* **thus** *?case* **by** *simp*
  case (*Cons x xs f t*)

    note *const-onD*[*OF Cons.prems carrier-possible*, **where** $y = x$, *simp*]

    have [*simp*]: $possible\ t\ x \Longrightarrow f\text{-}nxt\ f\ T\ x = f$
      by (*rule f-nxt-empty'*, *rule const-onD*[*OF Cons.prems carrier-possible*, **where** $y = x$])

from *Cons.prems carrier-nxt-subset*
have *const-on f* (*carrier* (*nxt t x*)) *empty*
by (*rule const-on-subset*)
hence *const-on* (*f-nxt f T x*) (*carrier* (*nxt t x*)) *empty*
by (*auto simp add*: *const-on-def f-nxt-def*)
note *Cons.IH*[*OF this*]
hence [*simp*]: *possible t x* $\Longrightarrow$ (*xs* $\in$ *paths* (*substitute f T* (*nxt t x*))) = (*xs* $\in$ *paths* (*nxt t x*))
by *simp*

show *?case* **by** (*auto simp add*: *Cons-path*)
qed
qed

lemma *substitute-only-empty-both*: *const-on f* (*carrier t'*) *empty* $\Longrightarrow$ *substitute f T* (*t* $\otimes\otimes$ *t'*) = *substitute f T t* $\otimes\otimes$ *t'*
proof (*intro paths-inj set-eqI*)
fix *xs*
assume *const-on f* (*carrier t'*) *TTree.empty*
thus (*xs* $\in$ *paths* (*substitute f T* (*t* $\otimes\otimes$ *t'*))) = (*xs* $\in$ *paths* (*substitute f T t* $\otimes\otimes$ *t'*))
proof (*induction xs arbitrary*: *f t t'*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof(*cases possible t' x*)
case *True*
hence *x* $\in$ *carrier t'* **by** (*metis carrier-possible*)
with *Cons.prems* **have** [*simp*]: *f x* = *empty* **by** *auto*
hence [*simp*]: *f-nxt f T x* = *f* **by** (*auto simp add*: *f-nxt-def*)

note *Cons.IH*[*OF Cons.prems*, **where** *t* = *nxt t x*, *simp*]

from *Cons.prems*
have *const-on f* (*carrier* (*nxt t' x*)) *empty* **by** (*metis carrier-nxt-subset const-on-subset*)
note *Cons.IH*[*OF this*, **where** *t* = *t*, *simp*]

show *?thesis* **using** *True*
by (*auto simp add*: *Cons-both nxt-both substitute-either*)
next
case *False*

have [*simp*]: *nxt t x* $\otimes\otimes$ *t'* $\otimes\otimes$ *f x* = *nxt t x* $\otimes\otimes$ *f x* $\otimes\otimes$ *t'*
by (*metis both-assoc both-comm*)

from *Cons.prems*
have *const-on* (*f-nxt f T x*) (*carrier t'*) *empty*
by (*force simp add*: *f-nxt-def*)

note *Cons.IH*[*OF this*, **where** $t = nxt\ t\ x \otimes\otimes f\ x$, *simp*]

show *?thesis* **using** *False*
by (*auto simp add*: *Cons-both nxt-both substitute-either*)
qed
qed
qed

lemma *f-nxt-upd-empty*[*simp*]:
f-nxt $(f(x' := empty))\ T\ x = (f\text{-}nxt\ f\ T\ x)(x' := empty)$
by (*auto simp add*: *f-nxt-def*)

lemma *repeatable-f-nxt-upd*[*simp*]:
repeatable $(f\ x) \Longrightarrow$ *repeatable* $(f\text{-}nxt\ f\ T\ x'\ x)$
by (*auto simp add*: *f-nxt-def*)

lemma *substitute-remove-anyways-aux*:
assumes *repeatable* $(f\ x)$
assumes $xs \in paths\ (substitute\ f\ T\ t)$
assumes $t \otimes\otimes f\ x = t$
shows $xs \in paths\ (substitute\ (f(x := empty))\ T\ t)$
using *assms*(*2*,*3*) *assms*(*1*)
proof (*induction f T t xs rule*: *substitute-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons f T t x' xs*)
show *?case*
proof(*cases* $x' = x$)
case *False*
hence [*simp*]: $(f(x := TTree.empty))\ x' = f\ x'$ **by** *simp*
have [*simp*]: $f\text{-}nxt\ f\ T\ x'\ x = f\ x$ **using** *False* **by** (*auto simp add*: *f-nxt-def*)
show *?thesis* **using** *Cons* **by** (*auto simp add*: *repeatable-both-nxt repeatable-both-both-nxt simp del*: *fun-upd-apply*)
next
case *True*
hence [*simp*]: $(f(x := TTree.empty))\ x = empty$ **by** *simp*

have $*$: $(f\text{-}nxt\ f\ T\ x)\ x = f\ x \vee (f\text{-}nxt\ f\ T\ x)\ x = empty$ **by** (*simp add*: *f-nxt-def*)
thus *?thesis*
using *Cons True*
by (*auto simp add*: *repeatable-both-nxt repeatable-both-both-nxt simp del*: *fun-upd-apply*)
qed
qed

lemma *substitute-remove-anyways*:
assumes *repeatable t*
assumes $f\ x = t$

shows *substitute f T* $(t \otimes\otimes t')$ = *substitute* $(f(x := \textit{empty}))$ T $(t \otimes\otimes t')$
proof (*rule paths-inj*, *rule*, *rule subsetI*)
fix *xs*
have *repeatable* $(f\, x)$ **using** *assms* **by** *simp*
moreover
assume $xs \in$ *paths* (*substitute f T* $(t \otimes\otimes t')$)
moreover
have $t \otimes\otimes t' \otimes\otimes f\, x = t \otimes\otimes t'$
by (*metis assms both-assoc both-comm repeatable-both-self*)
ultimately
show $xs \in$ *paths* (*substitute* $(f(x := \textit{empty}))$ T $(t \otimes\otimes t')$)
by (*rule substitute-remove-anyways-aux*)
next
show *paths* (*substitute* $(f(x := \textit{empty}))$ T $(t \otimes\otimes t')$) $\subseteq$ *paths* (*substitute f T* $(t \otimes\otimes t')$)
by (*rule substitute-mono1*) *auto*
qed

lemma *carrier-f-nxt*: *carrier* (*f-nxt f T x x'*) $\subseteq$ *carrier* $(f\, x')$
by (*simp add*: *f-nxt-def*)

lemma *f-nxt-cong*: $f\, x' = f'\, x' \Longrightarrow$ *f-nxt f T x x'* = *f-nxt f' T x x'*
by (*simp add*: *f-nxt-def*)

lemma *substitute-cong'*:
assumes $xs \in$ *paths* (*substitute f T t*)
assumes $\bigwedge x\ n.\ x \in A \Longrightarrow$ *carrier* $(f\, x) \subseteq A$
assumes *carrier* $t \subseteq A$
assumes $\bigwedge x.\ x \in A \Longrightarrow f\, x = f'\, x$
shows $xs \in$ *paths* (*substitute f' T t*)
using *assms*
proof (*induction f T t xs arbitrary*: *f' rule*: *substitute-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons f T t x xs*)
hence *possible t x* **by** *auto*
hence $x \in$ *carrier t* **by** (*metis carrier-possible*)
hence $x \in A$ **using** *Cons.prems*(*3*) **by** *auto*
with *Cons.prems* **have** [*simp*]: $f'\, x = f\, x$ **by** *auto*
have *carrier* $(f\, x) \subseteq A$ **using** ‹$x \in A$› **by** (*rule Cons.prems*(*2*))

from *Cons.prems*(*1*,*2*) *Cons.prems*(*4*)[*symmetric*]
show *?case*
by (*auto elim*!: *Cons.IH*
dest!: *set-mp*[*OF carrier-f-nxt*] *set-mp*[*OF carrier-nxt-subset*] *set-mp*[*OF Cons.prems*(*3*)]
set-mp[*OF* ‹*carrier* $(f\, x) \subseteq A$›]
intro: *f-nxt-cong*
)
qed

lemma *substitute-cong-induct*:
  assumes $\bigwedge x.\ x \in A \Longrightarrow carrier\ (f\ x) \subseteq A$
  assumes $carrier\ t \subseteq A$
  assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x = f'\ x$
  shows $substitute\ f\ T\ t = substitute\ f'\ T\ t$
  apply (*rule paths-inj*)
  apply (*rule set-eqI*)
  apply (*rule iffI*)
  apply (*erule* (*2*) *substitute-cong'*[*OF - assms*])
  apply (*erule substitute-cong'*[*OF - - assms*(*2*)])
  apply (*metis assms*(*1*,*3*))
  apply (*metis assms*(*3*))
  done

lemma *carrier-substitute-aux*:
  assumes $xs \in paths\ (substitute\ f\ T\ t)$
  assumes $carrier\ t \subseteq A$
  assumes $\bigwedge x.\ x \in A \Longrightarrow carrier\ (f\ x) \subseteq A$
  shows $set\ xs \subseteq A$
  using *assms*
  apply(*induction f T t xs rule*: *substitute-induct*)
  apply *auto*
  apply (*metis carrier-possible-subset*)
  apply (*metis carrier-f-nxt carrier-nxt-subset carrier-possible-subset contra-subsetD order-trans*)
  done

lemma *carrier-substitute-below*:
  assumes $\bigwedge x.\ x \in A \Longrightarrow carrier\ (f\ x) \subseteq A$
  assumes $carrier\ t \subseteq A$
  shows $carrier\ (substitute\ f\ T\ t) \subseteq A$
proof−
  have $\bigwedge xs.\ xs \in paths\ (substitute\ f\ T\ t) \Longrightarrow set\ xs \subseteq A$ **by** (*rule carrier-substitute-aux*[*OF - assms*(*2*,*1*)])
  thus *?thesis* **by** (*auto simp add*: *Union-paths-carrier*[*symmetric*])
qed

lemma *f-nxt-eq-empty-iff*:
  $f\text{-}nxt\ f\ T\ x\ x' = empty \longleftrightarrow f\ x' = empty \lor (x' = x \land x \in T)$
  by (*auto simp add*: *f-nxt-def*)

lemma *substitute-T-cong'*:
  assumes $xs \in paths\ (substitute\ f\ T\ t)$
  assumes $\bigwedge x.\ (x \in T \longleftrightarrow x \in T') \lor f\ x = empty$
  shows $xs \in paths\ (substitute\ f\ T'\ t)$
  using *assms*
proof (*induction f T t xs rule*: *substitute-induct*)
  case *Nil* **thus** *?case* **by** *simp*

next
  case $(Cons\ f\ T\ t\ x\ xs)$
  from $Cons.prems(2)$[**where** $x = x$]
  have $[simp]$: $f\text{-}nxt\ f\ T\ x = f\text{-}nxt\ f\ T'\ x$
    by $(auto\ simp\ add$: $f\text{-}nxt\text{-}def)$

  from $Cons.prems(2)$
  have $(\bigwedge x'.\ (x' \in T) = (x' \in T') \vee f\text{-}nxt\ f\ T\ x\ x' = TTree.empty)$
    by $(auto\ simp\ add$: $f\text{-}nxt\text{-}eq\text{-}empty\text{-}iff)$
  from $Cons.prems(1)\ Cons.IH[OF\ \text{-}\ this]$
  show $?case$
    by $auto$
qed

lemma $substitute\text{-}cong\text{-}T$:
  assumes $\bigwedge x.\ (x \in T \longleftrightarrow x \in T') \vee f\ x = empty$
  shows $substitute\ f\ T = substitute\ f\ T'$
  apply $rule$
  apply $(rule\ paths\text{-}inj)$
  apply $(rule\ set\text{-}eqI)$
  apply $(rule\ iffI)$
  apply $(erule\ substitute\text{-}T\text{-}cong'[OF\ \text{-}\ assms])$
  apply $(erule\ substitute\text{-}T\text{-}cong')$
  apply $(metis\ assms)$
  done

lemma $carrier\text{-}substitute1$: $carrier\ t \subseteq carrier\ (substitute\ f\ T\ t)$
    by $(rule\ carrier\text{-}mono)\ (rule\ substitute\text{-}contains\text{-}arg)$

lemma $substitute\text{-}cong$:
  assumes $\bigwedge x.\ x \in carrier\ (substitute\ f\ T\ t) \Longrightarrow f\ x = f'\ x$
  shows $substitute\ f\ T\ t = substitute\ f'\ T\ t$
proof$(rule\ substitute\text{-}cong\text{-}induct[OF\ \text{-}\ \text{-}\ assms])$
  show $carrier\ t \subseteq carrier\ (substitute\ f\ T\ t)$
    by $(rule\ carrier\text{-}substitute1)$
next
  fix x
  assume $x \in carrier\ (substitute\ f\ T\ t)$
  then obtain xs **where** $xs \in paths\ (substitute\ f\ T\ t)$ **and** $x \in set\ xs$ **by** $transfer\ auto$
  thus $carrier\ (f\ x) \subseteq carrier\ (substitute\ f\ T\ t)$
  proof $(induction\ xs\ arbitrary$: $f\ t)$
  case Nil **thus** $?case$ **by** $simp$
  next
  case $(Cons\ x'\ xs\ f\ t)$
    from $\langle x' \# xs \in paths\ (substitute\ f\ T\ t)\rangle$
    have $possible\ t\ x'$ **and** $xs \in paths\ (substitute\ (f\text{-}nxt\ f\ T\ x')\ T\ (nxt\ t\ x' \otimes\otimes\ f\ x'))$ **by** $auto$

    from $\langle x \in set\ (x' \# xs)\rangle$
    have $x = x' \vee (x \neq x' \wedge x \in set\ xs)$ **by** $auto$

hence *carrier* $(f\, x) \subseteq$ *carrier* (*substitute* (*f-nxt* f T x') T (*nxt* t x' $\otimes\otimes$ f x'))
proof(*elim conjE disjE*)
assume $x = x'$
have *carrier* $(f\, x) \subseteq$ *carrier* (*nxt* t x $\otimes\otimes$ f x) **by** *simp*
also have $\ldots \subseteq$ *carrier* (*substitute* (*f-nxt* f T x') T (*nxt* t x $\otimes\otimes$ f x)) **by** (*rule carrier-substitute1*)
finally show *?thesis* **unfolding** ‹$x = x'$›**.**
next
assume $x \neq x'$
hence [*simp*]: (*f-nxt* f T x' x) $= f\, x$ **by** (*simp add*: *f-nxt-def*)

assume $x \in$ *set xs*
from *Cons.IH*[*OF* ‹*xs* $\in$ - › *this*]
show *carrier* $(f\, x) \subseteq$ *carrier* (*substitute* (*f-nxt* f T x') T (*nxt* t x' $\otimes\otimes$ f x')) **by** *simp*
qed
also
from ‹*possible* t x'›
have *carrier* (*substitute* (*f-nxt* f T x') T (*nxt* t x' $\otimes\otimes$ f x')) $\subseteq$ *carrier* (*substitute* f T t)
apply *transfer*
apply *auto*
apply (*rule-tac* $x = x'\#xa$ **in** *exI*)
apply *auto*
done
finally show *?case***.**
qed
qed

lemma *substitute-substitute*:
assumes $\bigwedge$ x. *const-on* f' (*carrier* $(f\, x)$) *empty*
shows *substitute* f T (*substitute* f' T t) $=$ *substitute* (λ x. $f\, x$ $\otimes\otimes$ f' x) T t
proof (*rule paths-inj*, *rule set-eqI*)
fix *xs*
have [*simp*]: $\bigwedge$ f f' x'. *f-nxt* (λx. $f\, x$ $\otimes\otimes$ f' x) T x' $=$ (λx. *f-nxt* f T x' x $\otimes\otimes$ *f-nxt* f' T x' x)
by (*auto simp add*: *f-nxt-def*)

from *assms*
show *xs* $\in$ *paths* (*substitute* f T (*substitute* f' T t)) $\longleftrightarrow$ *xs* $\in$ *paths* (*substitute* (λx. $f\, x$ $\otimes\otimes$ f' x) T t)
proof (*induction xs arbitrary*: f f' t)
case *Nil* **thus** *?case* **by** *simp*
case (*Cons x xs*)
thus *?case*
proof (*cases possible t x*)
case *True*

from *Cons.prems*
have *prem'*: $\bigwedge$ x'. *const-on* (*f-nxt* f' T x) (*carrier* $(f\, x')$) *empty*
by (*force simp add*: *f-nxt-def*)

hence $\bigwedge x'$. *const-on* (*f-nxt* f' T x) (*carrier* ((*f-nxt* f T x) x')) *empty*
by (*metis carrier-empty const-onI emptyE f-nxt-def fun-upd-apply*)
note *Cons.IH*[**where** f = *f-nxt* f T x **and** f' = *f-nxt* f' T x, *OF this*, *simp*]

have [*simp*]: *nxt* t x $\otimes\otimes$ f x $\otimes\otimes$ f' x = *nxt* t x $\otimes\otimes$ f' x $\otimes\otimes$ f x
by (*metis both-comm both-assoc*)

show *?thesis* **using** *True*
by (*auto del*: *iffI simp add*: *substitute-only-empty-both*[*OF prem'*[**where** x' = x] , *symmetric*])
next
case *False*
thus *?thesis* **by** *simp*
qed
qed
qed

lemma *ttree-rest-substitute*:
assumes $\bigwedge$ x. *carrier* (f x) ∩ S = {}
shows *ttree-restr* S (*substitute* f T t) = *ttree-restr* S t
proof(*rule paths-inj*, *rule set-eqI*, *rule iffI*)
fix *xs*
assume *xs* ∈ *paths* (*ttree-restr* S (*substitute* f T t))
then
obtain *xs'* **where** [*simp*]: *xs* = *filter* (λ x'. x' ∈ S) *xs'* **and** *xs'* ∈ *paths* (*substitute* f T t)
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*])
from *this*(*2*) *assms*
have *filter* (λ x'. x' ∈ S) *xs'* ∈ *paths* (*ttree-restr* S t)
proof (*induction xs' arbitrary*: f t)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* x *xs* f t)
from *Cons.prems*
have *possible* t x **and** *xs* ∈ *paths* (*substitute* (*f-nxt* f T x) T (*nxt* t x $\otimes\otimes$ f x)) **by** *auto*

from *Cons.prems*(*2*)
have ($\bigwedge x'$. *carrier* (*f-nxt* f T x x') ∩ S = {}) **by** (*auto simp add*: *f-nxt-def*)

from *Cons.IH*[*OF* ‹*xs* ∈ -› *this*]
have [x'←*xs* . x' ∈ S] ∈ *paths* (*ttree-restr* S (*nxt* t x) $\otimes\otimes$ *ttree-restr* S (f x)) **by** (*simp add*: *ttree-restr-both*)
hence [x'←*xs* . x' ∈ S] ∈ *paths* (*ttree-restr* S (*nxt* t x)) **by** (*simp add*: *ttree-restr-is-empty*[*OF Cons.prems*(*2*)])
with ‹*possible* t x›
show [x'←x#*xs* . x' ∈ S] ∈ *paths* (*ttree-restr* S t)
by (*cases* x ∈ S) (*auto simp add*: *Cons-path ttree-restr-possible dest*: *set-mp*[*OF ttree-restr-nxt-subset2*] *set-mp*[*OF ttree-restr-nxt-subset*])
qed

thus *xs* ∈ *paths* (*ttree-restr S t*) **by** *simp*
next
fix *xs*
assume *xs* ∈ *paths* (*ttree-restr S t*)
then obtain *xs'* **where** [*simp*]:*xs* = *filter* (λ *x'*. *x'* ∈ *S*) *xs'* **and** *xs'* ∈ *paths t*
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*])
from *this*(*2*)
have *xs'* ∈ *paths* (*substitute f T t*) **by** (*rule set-mp*[*OF substitute-contains-arg*])
thus *xs* ∈ *paths* (*ttree-restr S* (*substitute f T t*))
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*])
qed

An alternative characterization of substitution

inductive *substitute''* :: ('*a* ⇒ '*a ttree*) ⇒ '*a set* ⇒ '*a list* ⇒ '*a list* ⇒ *bool* **where**
substitute''-Nil: *substitute'' f T* [] []
| *substitute''-Cons*:
zs ∈ *paths* (*f x*) ⟹ *xs'* ∈ *interleave xs zs* ⟹ *substitute''* (*f-nxt f T x*) *T xs' ys*
⟹ *substitute'' f T* (*x#xs*) (*x#ys*)
inductive-cases *substitute''-NilE*[*elim*]: *substitute'' f T xs* [] *substitute'' f T* [] *xs*
inductive-cases *substitute''-ConsE*[*elim*]: *substitute'' f T* (*x#xs*) *ys*

lemma *substitute-substitute''*:
xs ∈ *paths* (*substitute f T t*) ⟷ (∃ *xs'* ∈ *paths t*. *substitute'' f T xs' xs*)
proof
assume *xs* ∈ *paths* (*substitute f T t*)
thus ∃ *xs'* ∈ *paths t*. *substitute'' f T xs' xs*
proof(*induction xs arbitrary*: *f t*)
case *Nil*
have *substitute'' f T* [] []..
thus *?case* **by** *auto*
next
case (*Cons x xs f t*)
from ‹*x* # *xs* ∈ *paths* (*substitute f T t*)›
have *possible t x* **and** *xs* ∈ *paths* (*substitute* (*f-nxt f T x*) *T* (*nxt t x* ⊗⊗ *f x*)) **by** (*auto simp add*: *Cons-path*)
from *Cons.IH*[*OF this*(*2*)]
obtain *xs'* **where** *xs'* ∈ *paths* (*nxt t x* ⊗⊗ *f x*) **and** *substitute''* (*f-nxt f T x*) *T xs' xs* **by** *auto*
from *this*(*1*)
obtain *ys' zs'* **where** *ys'* ∈ *paths* (*nxt t x*) **and** *zs'* ∈ *paths* (*f x*) **and** *xs'* ∈ *interleave ys' zs'*
by (*auto simp add*: *paths-both*)

from *this*(*2*,*3*) ‹*substitute''* (*f-nxt f T x*) *T xs' xs*›
have *substitute'' f T* (*x* # *ys'*) (*x* # *xs*)..
moreover
from ‹*ys'* ∈ *paths* (*nxt t x*)› ‹*possible t x*›
have *x* # *ys'* ∈ *paths t* **by** (*simp add*: *Cons-path*)
ultimately

show *?case* **by** *auto*
qed
next
assume $\exists$ $xs' \in$ *paths t*. *substitute''* *f T xs' xs*
then obtain *xs'* **where** *substitute'' f T xs' xs* **and** $xs' \in$ *paths t* **by** *auto*
thus $xs \in$ *paths* (*substitute f T t*)
proof(*induction arbitrary*: *t rule*: *substitute''.induct*[*case-names Nil Cons*])
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons zs x xs' xs ys t*)
from *Cons.prems Cons.hyps*
show *?case* **by** (*force simp add*: *Cons-path paths-both intro*!: *Cons.IH*)
qed
qed

lemma *paths-substitute-substitute''*:
paths (*substitute f T t*) $= \bigcup((\lambda$ *xs* . *Collect* (*substitute'' f T xs*)) ' *paths t*)
by (*auto simp add*: *substitute-substitute''*)

lemma *ttree-rest-substitute2*:
assumes $\bigwedge$ *x*. *carrier* (*f x*) $\subseteq S$
assumes *const-on f* ($-S$) *empty*
shows *ttree-restr S* (*substitute f T t*) = *substitute f T* (*ttree-restr S t*)
proof(*rule paths-inj*, *rule set-eqI*, *rule iffI*)
fix *xs*
assume $xs \in$ *paths* (*ttree-restr S* (*substitute f T t*))
then
obtain *xs'* **where** [*simp*]: *xs* = *filter* (λ x'. $x' \in S$) *xs'* **and** $xs' \in$ *paths* (*substitute f T t*)
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*])
from *this*(*2*) *assms*
have *filter* (λ x'. $x' \in S$) $xs' \in$ *paths* (*substitute f T* (*ttree-restr S t*))
proof (*induction f T t xs' rule*: *substitute-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons f T t x xs*)
from *Cons.prems*(*1*)
have *possible t x* **and** $xs \in$ *paths* (*substitute* (*f-nxt f T x*) *T* (*nxt t x* $\otimes\otimes$ *f x*)) **by** *auto*
note *this*(*2*)
moreover
from *Cons.prems*(*2*)
have $\bigwedge$ x'. *carrier* (*f-nxt f T x x'*) $\subseteq S$ **by** (*auto simp add*: *f-nxt-def*)
moreover
from *Cons.prems*(*3*)
have *const-on* (*f-nxt f T x*) ($-S$) *empty* **by** (*force simp add*: *f-nxt-def*)
ultimately
have [x'←*xs* . $x' \in S$] $\in$ *paths* (*substitute* (*f-nxt f T x*) *T* (*ttree-restr S* (*nxt t x* $\otimes\otimes$ *f x*)))
by (*rule Cons.IH*)
hence $*$: [x'←*xs* . $x' \in S$] $\in$ *paths* (*substitute* (*f-nxt f T x*) *T* (*ttree-restr S* (*nxt t x* $\otimes\otimes$ *f x*))) **by** (*simp add*: *ttree-restr-both*)

show *?case*
proof (*cases* $x \in S$)
case *True*
show *?thesis*
using ‹*possible t x*› *Cons.prems*(*3*) ∗ *True*
by (*auto simp add*: *ttree-restr-both ttree-restr-noop*[*OF Cons.prems*(*2*)] *intro*: *ttree-restr-possible*
dest: *set-mp*[*OF substitute-mono2*[*OF both-mono1*[*OF ttree-restr-nxt-subset*]]])
next
case *False*
with ‹*const-on f* (− *S*) *TTree.empty*› **have** [*simp*]: *f x* = *empty* **by** *auto*
hence [*simp*]: *f-nxt f T x* = *f* **by** (*auto simp add*: *f-nxt-def*)
show *?thesis*
using ∗ *False*
by (*auto dest*: *set-mp*[*OF substitute-mono2*[*OF ttree-restr-nxt-subset2*]])
qed
qed
thus *xs* ∈ *paths* (*substitute f T* (*ttree-restr S t*)) **by** *simp*
next
fix *xs*
assume *xs* ∈ *paths* (*substitute f T* (*ttree-restr S t*))
then obtain *xs′* **where** *xs′* ∈ *paths t* **and** *substitute′′ f T* (*filter* (λ *x′*. *x′*∈*S*) *xs′*) *xs*
unfolding *substitute-substitute′′*
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*])

from *this*(*2*) *assms*
have ∃ *xs′′*. *xs* = *filter* (λ *x′*. *x′*∈*S*) *xs′′* ∧ *substitute′′ f T xs′ xs′′*
proof(*induction* (*xs′*,*xs*) *arbitrary*: *f xs′ xs rule*: *measure-induct-rule*[**where** *f* = λ (*xs*,*ys*). *length* (*filter* (λ *x′*. *x′* ∉ *S*) *xs*) + *length ys*])
case (*less xs ys*)
note ‹*substitute′′ f T* [*x′*←*xs* . *x′* ∈ *S*] *ys*›

show *?case*
proof(*cases xs*)
case *Nil* **with** *less.prems* **have** *ys* = [] **by** *auto*
thus *?thesis* **using** *Nil* **by** (*auto*, *metis filter.simps*(*1*) *substitute′′-Nil*)
next
case (*Cons x xs′*)
show *?thesis*
proof (*cases* $x \in S$)
case *True* **with** *Cons less.prems*
have *substitute′′ f T* (*x*# [*x′*←*xs′* . *x′* ∈ *S*]) *ys* **by** *simp*
from *substitute′′-ConsE*[*OF this*]
obtain *zs xs′′ ys′* **where** *ys* = *x* # *ys′* **and** *zs* ∈ *paths* (*f x*) **and** *xs′′* ∈ *interleave* [*x′*←*xs′* . *x′* ∈ *S*] *zs* **and** *substitute′′* (*f-nxt f T x*) *T xs′′ ys′*.
from ‹*zs* ∈ *paths* (*f x*)› *less.prems*(*2*)
have *set zs* ⊆ *S* **by** (*auto simp add*: *Union-paths-carrier*[*symmetric*])
hence [*simp*]: [*x′*←*zs* . *x′* ∈ *S*] = *zs* [*x′*←*zs* . *x′* ∉ *S*] = []
by (*metis UnCI Un-subset-iff eq-iff filter-True*,
metis ‹*set zs* ⊆ *S*› *filter-False insert-absorb insert-subset*)

from ‹$xs'' \in$ *interleave* $[x'{\leftarrow}xs'\ .\ x' \in S]$ *zs*›
have $xs'' \in$ *interleave* $[x'{\leftarrow}xs'\ .\ x' \in S]$ $[x'{\leftarrow}zs\ .\ x' \in S]$ **by** *simp*
then obtain xs''' **where** $xs'' = [x'{\leftarrow}xs'''\ .\ x' \in S]$ **and** $xs''' \in$ *interleave* xs' *zs* **by** (*rule interleave-filter*)

from ‹$xs''' \in$ *interleave* xs' *zs*›
have *l*: $\bigwedge P$. *length* (*filter* P xs''') = *length* (*filter* P xs') + *length* (*filter* P *zs*)
by (*induction*) *auto*

from ‹*substitute''* (*f-nxt f T x*) *T* xs'' ys'› ‹xs'' = -›
have *substitute''* (*f-nxt f T x*) *T* $[x'{\leftarrow}xs'''\ .\ x' \in S]$ ys' **by** *simp*
moreover
from *less.prems*(*2*)
have $\bigwedge xa$. *carrier* (*f-nxt f T x xa*) $\subseteq S$
by (*auto simp add*: *f-nxt-def*)
moreover
from *less.prems*(*3*)
have *const-on* (*f-nxt f T x*) $(- S)$ *TTree.empty* **by** (*force simp add*: *f-nxt-def*)
ultimately
have $\exists\, ys''$. $ys' = [x'{\leftarrow}ys''\ .\ x' \in S] \wedge$ *substitute''* (*f-nxt f T x*) *T* xs''' ys''
by (*rule less.hyps*[*rotated*])
(*auto simp add*: ‹*ys* = - › ‹*xs* =-› ‹$x \in S$› ‹xs'' = -›[*symmetric*] *l*)[*1*]
then obtain ys'' **where** $ys' = [x'{\leftarrow}ys''\ .\ x' \in S]$ **and** *substitute''* (*f-nxt f T x*) *T* xs''' ys'' **by** *blast*
hence $ys = [x'{\leftarrow}x\#ys''\ .\ x' \in S]$ **using** ‹$x \in S$› ‹*ys* = -› **by** *simp*
moreover
from ‹$zs \in$ *paths* (*f x*)› ‹$xs''' \in$ *interleave* xs' *zs*› ‹*substitute''* (*f-nxt f T x*) *T* xs''' ys''›
have *substitute''* *f T* $(x\#xs')$ $(x\#ys'')$
by *rule*
ultimately
show *?thesis* **unfolding** *Cons* **by** *blast*
next
case *False* **with** *Cons less.prems*
have *substitute''* *f T* ($[x'{\leftarrow}xs'\ .\ x' \in S]$) *ys* **by** *simp*
hence $\exists\, ys'$. $ys = [x'{\leftarrow}ys'\ .\ x' \in S] \wedge$ *substitute''* *f T* xs' ys'
by (*rule less.hyps*[*OF - - less.prems*(*2*,*3*), *rotated*]) (*auto simp add*: ‹*xs* =-› ‹$x \notin S$›)
then obtain ys' **where** $ys = [x'{\leftarrow}ys'\ .\ x' \in S]$ **and** *substitute''* *f T* xs' ys' **by** *auto*

from *this*(*1*)
have $ys = [x'{\leftarrow}x\#ys'\ .\ x' \in S]$ **using** ‹$x \notin S$› ‹*ys* = -› **by** *simp*
moreover
have [*simp*]: *f x* = *empty* **using** ‹$x \notin S$› *less.prems*(*3*) **by** *force*
hence *f-nxt f T x* = *f* **by** (*auto simp add*: *f-nxt-def*)
with ‹*substitute''* *f T* xs' ys'›
have *substitute''* *f T* $(x\#xs')$ $(x\#ys')$
by (*auto intro*: *substitute''.intros*)
ultimately

show *?thesis* **unfolding** *Cons* **by** *blast*
qed
qed
qed
then obtain xs'' **where** $xs = \mathit{filter}\ (\lambda\ x'.\ x' \in S)\ xs''$ **and** $\mathit{substitute}''\ f\ T\ xs'\ xs''$ **by** *auto*
from $\mathit{this}(2)$ ‹$xs' \in \mathit{paths}\ t$›
have $xs'' \in \mathit{paths}\ (\mathit{substitute}\ f\ T\ t)$ **by** (*auto simp add*: *substitute-substitute''*)
with ‹$xs = -$›
show $xs \in \mathit{paths}\ (\mathit{ttree\text{-}restr}\ S\ (\mathit{substitute}\ f\ T\ t))$
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*])
qed

end

85 TTree-HOLCF.tex

theory *TTree−HOLCF*
imports *TTree HOLCF−Utils Set−Cpo HOLCF−Join−Classes*
begin

instantiation *ttree* :: (*type*) *below*
begin
lift-definition *below-ttree* :: $'a\ \mathit{ttree} \Rightarrow 'a\ \mathit{ttree} \Rightarrow \mathit{bool}$ **is** *op* $\subseteq$**.**
instance..
end

lemma *paths-mono*: $t \sqsubseteq t' \Longrightarrow \mathit{paths}\ t \sqsubseteq \mathit{paths}\ t'$
by *transfer* (*auto simp add*: *below-set-def*)

lemma *paths-mono-iff*: $\mathit{paths}\ t \sqsubseteq \mathit{paths}\ t' \longleftrightarrow t \sqsubseteq t'$
by *transfer* (*auto simp add*: *below-set-def*)

lemma *ttree-belowI*: $(\bigwedge\ xs.\ xs \in \mathit{paths}\ t \Longrightarrow xs \in \mathit{paths}\ t') \Longrightarrow t \sqsubseteq t'$
by *transfer auto*

lemma *paths-belowI*: $(\bigwedge\ x\ xs.\ x\#xs \in \mathit{paths}\ t \Longrightarrow x\#xs \in \mathit{paths}\ t') \Longrightarrow t \sqsubseteq t'$
apply (*rule ttree-belowI*)
apply (*case-tac xs*)
apply *auto*
done

instance *ttree* :: (*type*) *po*
by *standard* (*transfer*, *simp*)+

lemma *is-lub-ttree*:
S <<| *Either S*
unfolding *is-lub-def is-ub-def*
by *transfer auto*

lemma *lub-is-either*: *lub S* = *Either S*
  using *is-lub-ttree* **by** (*rule lub-eqI*)

instance *ttree* :: (*type*) *cpo*
  by *standard* (*rule exI*, *rule is-lub-ttree*)

lemma *minimal-ttree*[*simp*, *intro*!]: *empty* $\sqsubseteq$ *S*
  by *transfer simp*

instance *ttree* :: (*type*) *pcpo*
  by *standard* (*rule*+)

lemma *empty-is-bottom*: *empty* = $\bot$
  by (*metis below-bottom-iff minimal-ttree*)

lemma *carrier-bottom*[*simp*]: *carrier* $\bot$ = {}
  unfolding *empty-is-bottom*[*symmetric*] **by** *simp*

lemma *below-anything*[*simp*]:
  $t \sqsubseteq$ *anything*
by *transfer auto*

lemma *carrier-mono*: $t \sqsubseteq t' \Longrightarrow$ *carrier* $t \subseteq$ *carrier* t'
  by *transfer auto*

lemma *nxt-mono*: $t \sqsubseteq t' \Longrightarrow$ *nxt* t $x \sqsubseteq$ *nxt* t' x
  by *transfer auto*

lemma *either-above-arg1*: $t \sqsubseteq t \oplus\oplus t'$
  by *transfer fastforce*

lemma *either-above-arg2*: $t' \sqsubseteq t \oplus\oplus t'$
  by *transfer fastforce*

lemma *either-belowI*: $t \sqsubseteq t'' \Longrightarrow t' \sqsubseteq t'' \Longrightarrow t \oplus\oplus t' \sqsubseteq t''$
  by *transfer auto*

lemma *both-above-arg1*: $t \sqsubseteq t \otimes\otimes t'$
  by *transfer fastforce*

lemma *both-above-arg2*: $t' \sqsubseteq t \otimes\otimes t'$
  by *transfer fastforce*

lemma *both-mono1'*:
  $t \sqsubseteq t' \Longrightarrow t \otimes\otimes t'' \sqsubseteq t' \otimes\otimes t''$
  using *both-mono1*[*folded below-set-def*, *unfolded paths-mono-iff*].

lemma *both-mono2'*:

$t \sqsubseteq t' \Longrightarrow t'' \otimes\otimes\ t \sqsubseteq t'' \otimes\otimes\ t'$
using *both-mono2*[*folded below-set-def*, *unfolded paths-mono-iff*].

lemma *nxt-both-left*:
possible $t\ x \Longrightarrow$ *nxt* $t\ x \otimes\otimes\ t' \sqsubseteq$ *nxt* $(t \otimes\otimes\ t')\ x$
by (*auto simp add*: *nxt-both either-above-arg2*)

lemma *nxt-both-right*:
possible $t'\ x \Longrightarrow t \otimes\otimes$ *nxt* $t'\ x \sqsubseteq$ *nxt* $(t \otimes\otimes\ t')\ x$
by (*auto simp add*: *nxt-both either-above-arg1*)

lemma *substitute-mono1'*: $f \sqsubseteq f' \Longrightarrow$ *substitute* $f\ T\ t \sqsubseteq$ *substitute* $f'\ T\ t$
using *substitute-mono1*[*folded below-set-def*, *unfolded paths-mono-iff*] *fun-belowD*
by *metis*

lemma *substitute-mono2'*: $t \sqsubseteq t' \Longrightarrow$ *substitute* $f\ T\ t \sqsubseteq$ *substitute* $f\ T\ t'$
using *substitute-mono2*[*folded below-set-def*, *unfolded paths-mono-iff*].

lemma *substitute-above-arg*: $t \sqsubseteq$ *substitute* $f\ T\ t$
using *substitute-contains-arg*[*folded below-set-def*, *unfolded paths-mono-iff*].

lemma *ttree-contI*:
assumes $\bigwedge S.\ f$ (*Either* S) = *Either* (f ‘ S)
shows *cont f*
proof(*rule contI*)
fix Y :: *nat* $\Rightarrow$ *'a ttree*
have *range* $(\lambda i.\ f\ (Y\ i)) = f$ ‘ *range* Y **by** *auto*
also have *Either* ... = f (*Either* (*range* Y)) **unfolding** *assms*(*1*)..
also have *Either* (*range* Y) = *lub* (*range* Y) **unfolding** *lub-is-either* **by** *simp*
finally
show *range* $(\lambda i.\ f\ (Y\ i))$ <<| f $(\bigsqcup\ i.\ Y\ i)$ **by** (*metis is-lub-ttree*)
qed

lemma *ttree-contI2*:
assumes $\bigwedge x.$ *paths* $(f\ x) = \bigcup(t$ ‘ *paths* $x)$
assumes $[] \in t\ []$
shows *cont f*
proof(*rule contI*)
fix Y :: *nat* $\Rightarrow$ *'a ttree*
have *paths* (*Either* (*range* $(\lambda i.\ f\ (Y\ i))$)) = *insert* [] $(\bigcup x.$ *paths* $(f\ (Y\ x)))$
by (*simp add*: *paths-Either*)
also have ... = *insert* [] $(\bigcup x. \bigcup(t$ ‘ *paths* $(Y\ x)))$
by (*simp add*: *assms*(*1*))
also have ... = $\bigcup(t$ ‘ *insert* [] $(\bigcup x.$ *paths* $(Y\ x)))$
using *assms*(*2*) **by** (*auto 0 4*)
also have ... = $\bigcup(t$ ‘ *paths* (*Either* (*range* Y)))
by (*auto simp add*: *paths-Either*)

also have ... = *paths* (*f* (*Either* (*range Y*)))
  by (*simp add*: *assms*(*1*))
also have ... = *paths* (*f* (*lub* (*range Y*))) **unfolding** *lub-is-either* **by** *simp*
finally
show *range* (λi. *f* (*Y i*)) <<| *f* ($\bigsqcup$ *i*. *Y i*) **by** (*metis is-lub-ttree paths-inj*)
qed

lemma *cont-paths*[*THEN cont-compose*, *cont2cont*, *simp*]:
cont paths
apply (*rule set-contI*)
apply (*thin-tac* -)
unfolding *lub-is-either*
apply *transfer*
apply *auto*
done

lemma *ttree-contI3*:
assumes *cont* (λ *x*. *paths* (*f x*))
shows *cont f*
apply (*rule contI2*)
apply (*rule monofunI*)
apply (*subst paths-mono-iff*[*symmetric*])
apply (*erule cont2monofunE*[*OF assms*])

apply (*subst paths-mono-iff*[*symmetric*])
apply (*subst cont2contlubE*[*OF cont-paths*[*OF cont-id*]], *assumption*)
apply (*subst cont2contlubE*[*OF assms*], *assumption*)
apply *rule*
done

lemma *cont-substitute*[*THEN cont-compose*, *cont2cont*, *simp*]:
cont (*substitute f T*)
apply (*rule ttree-contI2*)
apply (*rule paths-substitute-substitute''*)
apply (*auto intro*: *substitute''.intros*)
done

lemma *cont-both1*:
cont (λ *x*. *both x y*)
apply (*rule ttree-contI2*[**where** $t = \lambda xs$. {*zs* . $\exists$ *ys*$\in$*paths y*. *zs* $\in$ *xs* $\otimes$ *ys*}])
apply (*rule set-eqI*)
by (*auto intro*: *simp add*: *paths-both*)

lemma *cont-both2*:
cont (λ *x*. *both y x*)
apply (*rule ttree-contI2*[**where** $t = \lambda ys$. {*zs* . $\exists$ *xs*$\in$*paths y*. *zs* $\in$ *xs* $\otimes$ *ys*}])

apply (*rule set-eqI*)
by (*auto intro*: *simp add*: *paths-both*)

lemma *cont-both*[*cont2cont*,*simp*]: *cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *cont* (λ *x*. *f x* $\otimes\otimes$ *g x*)
by (*rule cont-compose2*[*OF cont-both1 cont-both2*])

lemma *cont-intersect1*:
cont (λ *x*. *intersect x y*)
by (*rule ttree-contI2* [**where** $t = \lambda xs$. (*if xs* $\in$ *paths y then* {*xs*} *else* {})])
(*auto split*: *if-splits*)

lemma *cont-intersect2*:
cont (λ *x*. *intersect y x*)
by (*rule ttree-contI2* [**where** $t = \lambda xs$. (*if xs* $\in$ *paths y then* {*xs*} *else* {})])
(*auto split*: *if-splits*)

lemma *cont-intersect*[*cont2cont*,*simp*]: *cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *cont* (λ *x*. *f x* $\cap\cap$ *g x*)
by (*rule cont-compose2*[*OF cont-intersect1 cont-intersect2*])

lemma *cont-without*[*THEN cont-compose*, *cont2cont*,*simp*]: *cont* (*without x*)
by (*rule ttree-contI2*[**where** $t = \lambda$ *xs*.{*filter* (λ x'. $x' \neq x$) *xs*}])
(*transfer*, *auto*)

lemma *paths-many-calls-subset*:
$t \sqsubseteq$ *many-calls x* $\otimes\otimes$ *without x t*
by (*metis* (*full-types*) *below-set-def paths-many-calls-subset paths-mono-iff*)

lemma *single-below*:
[*x*] $\in$ *paths t* $\Longrightarrow$ *single x* $\sqsubseteq$ *t* **by** *transfer auto*

lemma *cont-ttree-restr*[*THEN cont-compose*, *cont2cont*,*simp*]: *cont* (*ttree-restr S*)
by (*rule ttree-contI2*[**where** $t = \lambda$ *xs*.{*filter* (λ x'. $x' \in S$) *xs*}])
(*transfer*, *auto*)

lemmas *ttree-restr-mono* = *cont2monofunE*[*OF cont-ttree-restr*[*OF cont-id*]]

lemma *range-filter*[*simp*]: *range* (*filter P*) = {*xs*. *set xs* $\subseteq$ *Collect P*}
apply *auto*
apply (*rule-tac x* = *x* **in** *rev-image-eqI*)
apply *simp*
apply (*rule sym*)
apply (*auto simp add*: *filter-id-conv*)
done

lemma *ttree-restr-anything-cont*[*THEN cont-compose*, *simp*, *cont2cont*]:
cont (λ *S*. *ttree-restr S anything*)
apply (*rule ttree-contI3*)
apply (*rule set-contI*)

apply (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*] *lub-set*)
apply (*rule finite-subset-chain*)
apply *auto*
done

instance *ttree* :: (*type*) *Finite-Join-cpo*
proof
fix *x y* :: *'a ttree*
show *compatible x y*
unfolding *compatible-def*
apply (*rule exI*)
apply (*rule is-lub-ttree*)
done
qed

lemma *ttree-join-is-either*:
$t \sqcup t' = t \oplus\oplus t'$
proof−
have $t \oplus\oplus t' = Either \{t, t'\}$ **by** *transfer auto*
thus $t \sqcup t' = t \oplus\oplus t'$ **by** (*metis lub-is-join is-lub-ttree*)
qed

lemma *ttree-join-transfer*[*transfer-rule*]: *rel-fun* (*pcr-ttree op* =) (*rel-fun* (*pcr-ttree op* =) (*pcr-ttree op* =)) *op* ∪ *op* ⊔
proof−
have *op* ⊔ = (*op* ⊕⊕ :: *'a ttree* ⇒ *'a ttree* ⇒ *'a ttree*) **using** *ttree-join-is-either* **by** *blast*
thus *?thesis* **using** *either.transfer* **by** *metis*
qed

lemma *ttree-restr-join*[*simp*]:
ttree-restr S $(t \sqcup t')$ = *ttree-restr S t* ⊔ *ttree-restr S t'*
by *transfer auto*

lemma *nxt-singles-below-singles*:
nxt (*singles S*) *x* ⊑ *singles S*
apply *auto*
apply *transfer*
apply *auto*
apply (*erule-tac x = xc* **in** *ballE*)
apply (*erule order-trans*[*rotated*])
apply (*rule length-filter-mono*)
apply *simp*
apply *simp*
done

lemma *in-carrier-fup*[*simp*]:
$x' \in carrier\ (fup \cdot f \cdot u) \longleftrightarrow (\exists\ u'.\ u = up \cdot u' \wedge x' \in carrier\ (f \cdot u'))$
by (*cases u*) *auto*

end

86 AnalBinds.tex

theory *AnalBinds*
imports *Terms HOLCF−Utils Env*
begin

locale *ExpAnalysis* =
  fixes *exp* :: *exp* ⇒ *'a*::*cpo* → *'b*::*pcpo*
begin

fun *AnalBinds* :: *heap* ⇒ (*var* ⇒ $'a_\perp$) → (*var* ⇒ *'b*)
  where *AnalBinds* [] = (Λ *ae*. ⊥)
    | *AnalBinds* ((*x*,*e*)#Γ) = (Λ *ae*. (*AnalBinds* Γ·*ae*)(*x* := *fup*·(*exp e*)·(*ae x*)))

lemma *AnalBinds-Nil-simp*[*simp*]: *AnalBinds* []·*ae* = ⊥ **by** *simp*

lemma *AnalBinds-Cons*[*simp*]:
  AnalBinds ((*x*,*e*)#Γ)·*ae* = (*AnalBinds* Γ·*ae*)(*x* := *fup*·(*exp e*)·(*ae x*))
  by *simp*

lemmas *AnalBinds.simps*[*simp del*]

lemma *AnalBinds-not-there*: *x* ∉ *domA* Γ ⟹ (*AnalBinds* Γ·*ae*) *x* = ⊥
  by (*induction* Γ *rule*: *AnalBinds.induct*) *auto*

lemma *AnalBinds-cong*:
  assumes *ae f*|' *domA* Γ = *ae' f*|' *domA* Γ
  shows *AnalBinds* Γ·*ae* = *AnalBinds* Γ·*ae'*
using *env-restr-eqD*[*OF assms*]
by (*induction* Γ *rule*: *AnalBinds.induct*) (*auto split*: *if-splits*)

lemma *AnalBinds-lookup*: (*AnalBinds* Γ·*ae*) *x* = (*case map-of* Γ *x of Some e* ⇒ *fup*·(*exp e*)·(*ae x*) | *None* ⇒ ⊥)
  by (*induction* Γ *rule*: *AnalBinds.induct*) *auto*

lemma *AnalBinds-delete-bot*: *ae x* = ⊥ ⟹ *AnalBinds* (*delete x* Γ)·*ae* = *AnalBinds* Γ·*ae*
  by (*auto simp add*: *AnalBinds-lookup split*:*option.split simp add*: *delete-conv*)

lemma *AnalBinds-delete-below*: *AnalBinds* (*delete x* Γ)·*ae* ⊑ *AnalBinds* Γ·*ae*
  by (*auto intro*: *fun-belowI simp add*: *AnalBinds-lookup split*:*option.split*)

lemma *AnalBinds-delete-lookup*[*simp*]: (*AnalBinds* (*delete x* Γ)·*ae*) *x* = ⊥
  by (*auto simp add*: *AnalBinds-lookup split*:*option.split*)

lemma *AnalBinds-delete-to-fun-upd*: *AnalBinds* (*delete x* Γ)·*ae* = (*AnalBinds* Γ·*ae*)(*x* := ⊥)
  by (*auto simp add*: *AnalBinds-lookup split*:*option.split*)

lemma *edom-AnalBinds*: *edom* (*AnalBinds* Γ·*ae*) ⊆ *domA* Γ ∩ *edom ae*
by (*induction* Γ *rule*: *AnalBinds.induct*) (*auto simp add*: *edom-def*)

end

end

87 TTreeAnalysisSig.tex

theory *TTreeAnalysisSig*
imports *Arity TTree−HOLCF AnalBinds*
begin

locale *TTreeAnalysis* =
fixes *Texp* :: *exp* ⇒ *Arity* → *var ttree*
begin
sublocale *Texp*: *ExpAnalysis Texp*.
abbreviation *FBinds* == *Texp.AnalBinds*
end

end

88 CoCallGraph-TTree.tex

theory *CoCallGraph−TTree*
imports *CoCallGraph TTree−HOLCF*
begin

lemma *interleave-ccFromList*:
xs ∈ *interleave ys zs* ⟹ *ccFromList xs* = *ccFromList ys* ⊔ *ccFromList zs* ⊔ *ccProd* (*set ys*) (*set zs*)
by (*induction rule*: *interleave-induct*)
(*auto simp add*: *interleave-set ccProd-comm ccProd-insert2*[**where** *S′* = *set xs* **for** *xs*] *ccProd-insert1*[**where** *S′* = *set xs* **for** *xs*])

lift-definition *ccApprox* :: *var ttree* ⇒ *CoCalls*
is λ *xss* . *lub* (*ccFromList* ‘ *xss*).

lemma *ccApprox-paths*: *ccApprox t* = *lub* (*ccFromList* ‘ (*paths t*)) **by** *transfer simp*

lemma *ccApprox-strict*[*simp*]: *ccApprox* ⊥ = ⊥
by (*simp add*: *ccApprox-paths empty-is-bottom*[*symmetric*])

lemma *in-ccApprox*: (*x*−−*y*∈(*ccApprox t*)) ⟷ (∃ *xs* ∈ *paths t*. (*x*−−*y*∈(*ccFromList xs*)))
unfolding *ccApprox-paths*

by *transfer auto*

lemma *ccApprox-mono*: *paths* $t \subseteq$ *paths* $t' \Longrightarrow$ *ccApprox* $t \sqsubseteq$ *ccApprox* t'
by (*rule below-CoCallsI*) (*auto simp add*: *in-ccApprox*)

lemma *ccApprox-mono'*: $t \sqsubseteq t' \Longrightarrow$ *ccApprox* $t \sqsubseteq$ *ccApprox* t'
by (*metis below-set-def ccApprox-mono paths-mono-iff*)

lemma *ccApprox-belowI*: $(\bigwedge$ *xs*. *xs* $\in$ *paths* $t \Longrightarrow$ *ccFromList xs* $\sqsubseteq G) \Longrightarrow$ *ccApprox* $t \sqsubseteq G$
unfolding *ccApprox-paths*
by *transfer auto*

lemma *ccApprox-below-iff*: *ccApprox* $t \sqsubseteq G \longleftrightarrow (\forall$ *xs* $\in$ *paths* t. *ccFromList xs* $\sqsubseteq G)$
unfolding *ccApprox-paths* **by** *transfer auto*

lemma *cc-restr-ccApprox-below-iff*: *cc-restr* S (*ccApprox* t) $\sqsubseteq G \longleftrightarrow (\forall$ *xs* $\in$ *paths* t. *cc-restr* S (*ccFromList xs*) $\sqsubseteq G)$
unfolding *ccApprox-paths cc-restr-lub*
by *transfer auto*

lemma *ccFromList-below-ccApprox*:
xs $\in$ *paths* $t \Longrightarrow$ *ccFromList xs* $\sqsubseteq$ *ccApprox* t
by (*rule below-CoCallsI*)(*auto simp add*: *in-ccApprox*)

lemma *ccApprox-nxt-below*:
ccApprox (*nxt* t x) $\sqsubseteq$ *ccApprox* t
by (*rule below-CoCallsI*)(*auto simp add*: *in-ccApprox paths-nxt-eq elim*!: *bexI*[*rotated*])

lemma *ccApprox-ttree-restr-nxt-below*:
ccApprox (*ttree-restr* S (*nxt* t x)) $\sqsubseteq$ *ccApprox* (*ttree-restr* S t)
by (*rule below-CoCallsI*)
(*auto simp add*: *in-ccApprox filter-paths-conv-free-restr*[*symmetric*] *paths-nxt-eq elim*!: *bexI*[*rotated*])

lemma *ccApprox-ttree-restr*[*simp*]: *ccApprox* (*ttree-restr* S t) = *cc-restr* S (*ccApprox* t)
by (*rule CoCalls-eqI*) (*auto simp add*: *in-ccApprox filter-paths-conv-free-restr*[*symmetric*])

lemma *ccApprox-both*: *ccApprox* ($t \otimes\otimes t'$) = *ccApprox* $t \sqcup$ *ccApprox* $t' \sqcup$ *ccProd* (*carrier* t) (*carrier* t')
proof (*rule below-antisym*)
show *ccApprox* ($t \otimes\otimes t'$) $\sqsubseteq$ *ccApprox* $t \sqcup$ *ccApprox* $t' \sqcup$ *ccProd* (*carrier* t) (*carrier* t')
by (*rule below-CoCallsI*)
(*auto 4 4 simp add*: *in-ccApprox paths-both Union-paths-carrier*[*symmetric*] *interleave-ccFromList*)
next
have *ccApprox* $t \sqsubseteq$ *ccApprox* ($t \otimes\otimes t'$)
by (*rule ccApprox-mono*[*OF both-contains-arg1*])
moreover
have *ccApprox* $t' \sqsubseteq$ *ccApprox* ($t \otimes\otimes t'$)
by (*rule ccApprox-mono*[*OF both-contains-arg2*])
moreover

have *ccProd* (*carrier t*) (*carrier t'*) $\sqsubseteq$ *ccApprox* ($t \otimes\otimes t'$)
proof(*rule ccProd-belowI*)
fix *x y*
assume $x \in$ *carrier t* **and** $y \in$ *carrier t'*
then obtain *xs ys* **where** $x \in$ *set xs* **and** $y \in$ *set ys*
and $xs \in$ *paths t* **and** $ys \in$ *paths t'* **by** (*auto simp add*: *Union-paths-carrier*[*symmetric*])
hence *xs* @ *ys* $\in$ *paths* ($t \otimes\otimes t'$) **by** (*metis paths-both append-interleave*)
moreover
from ‹$x \in$ *set xs*› ‹$y \in$ *set ys*›
have $x{-}{-}y{\in}$(*ccFromList* (*xs*@*ys*)) **by** *simp*
ultimately
show $x{-}{-}y{\in}$(*ccApprox* ($t \otimes\otimes t'$)) **by** (*auto simp add*: *in-ccApprox simp del*: *ccFromList-append*)
qed
ultimately
show *ccApprox t* $\sqcup$ *ccApprox t'* $\sqcup$ *ccProd* (*carrier t*) (*carrier t'*) $\sqsubseteq$ *ccApprox* ($t \otimes\otimes t'$)
by (*simp add*: *join-below-iff*)
qed

lemma *ccApprox-many-calls*[*simp*]:
ccApprox (*many-calls x*) = *ccProd* $\{x\}$ $\{x\}$
by *transfer'* (*rule CoCalls-eqI*, *auto*)

lemma *ccApprox-single*[*simp*]:
ccApprox (*TTree.single y*) = $\bot$
by *transfer' auto*

lemma *ccApprox-either*[*simp*]: *ccApprox* ($t \oplus\oplus t'$) = *ccApprox t* $\sqcup$ *ccApprox t'*
by *transfer'* (*rule CoCalls-eqI*, *auto*)

lemma *wild-recursion*:
assumes *ccApprox t* $\sqsubseteq$ *G*
assumes $\bigwedge x.\ x \notin S \Longrightarrow f\ x = empty$
assumes $\bigwedge x.\ x \in S \Longrightarrow ccApprox\ (f\ x) \sqsubseteq G$
assumes $\bigwedge x.\ x \in S \Longrightarrow ccProd\ (ccNeighbors\ x\ G)\ (carrier\ (f\ x)) \sqsubseteq G$
shows *ccApprox* (*ttree-restr* ($-S$) (*substitute f T t*)) $\sqsubseteq$ *G*
proof(*rule ccApprox-belowI*)
fix *xs*
def *seen* $\equiv$ {} :: *var set*

assume $xs \in$ *paths* (*ttree-restr* ($-$ *S*) (*substitute f T t*))
then obtain *xs' xs''* **where** $xs = [x{\leftarrow}xs'\ .\ x \notin S]$ **and** *substitute'' f T xs'' xs'* **and** $xs'' \in$ *paths t*
by (*auto simp add*: *filter-paths-conv-free-restr2*[*symmetric*] *substitute-substitute''*)

note *this*(*2*)
moreover
from ‹*ccApprox t* $\sqsubseteq$ *G*› **and** ‹$xs'' \in$ *paths t*›

have *ccFromList xs*$''$ $\sqsubseteq$ *G*
by (*auto simp add*: *ccApprox-below-iff*)
moreover
note *assms*(*2*)
moreover
from *assms*(*3*,*4*)
have $\bigwedge$ *x ys*. $x \in S \Longrightarrow ys \in paths\ (f\ x) \Longrightarrow$ *ccFromList ys* $\sqsubseteq G$
and $\bigwedge$ *x ys*. $x \in S \Longrightarrow ys \in paths\ (f\ x) \Longrightarrow$ *ccProd* (*ccNeighbors x G*) (*set ys*) $\sqsubseteq G$
by (*auto simp add*: *ccApprox-below-iff Union-paths-carrier*[*symmetric*] *cc-lub-below-iff*)
moreover
have *ccProd seen* (*set xs*$''$) $\sqsubseteq G$ **unfolding** *seen-def* **by** *simp*
ultimately
have *ccFromList* [$x \leftarrow xs'$. $x \notin S$] $\sqsubseteq G \wedge$ *ccProd* (*seen*) (*set xs*$'$) $\sqsubseteq G$
proof(*induction f T xs*$''$ *xs*$'$ *arbitrary*: *seen rule*: *substitute*$''$.*induct*[*case-names Nil Cons*])
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons zs f x xs*$'$ *xs T ys*)

have *seen-x*: *ccProd seen* $\{x\}$ $\sqsubseteq G$
using ‹*ccProd seen* (*set* (x # *xs*)) $\sqsubseteq G$›
by (*auto simp add*: *ccProd-insert2*[**where** S' = *set xs* **for** *xs*] *join-below-iff*)

show *?case*
proof(*cases* $x \in S$)
case *True*

from ‹*ccFromList* (x # *xs*) $\sqsubseteq G$›
have *ccProd* $\{x\}$ (*set xs*) $\sqsubseteq G$ **by** (*auto simp add*: *join-below-iff*)
hence *subset1*: *set xs* $\subseteq$ *ccNeighbors x G* **by** *transfer auto*

from ‹*ccProd seen* (*set* (x # *xs*)) $\sqsubseteq G$›
have *subset2*: *seen* $\subseteq$ *ccNeighbors x G*
by (*auto simp add*: *subset-ccNeighbors ccProd-insert2*[**where** S' = *set xs* **for** *xs*] *join-below-iff ccProd-comm*)

from *subset1* **and** *subset2*
have *seen* $\cup$ *set xs* $\subseteq$ *ccNeighbors x G* **by** *auto*
hence *ccProd* (*seen* $\cup$ *set xs*) (*set zs*) $\sqsubseteq$ *ccProd* (*ccNeighbors x G*) (*set zs*)
by (*rule ccProd-mono1*)
also
from ‹$x \in S$› ‹*zs* $\in$ *paths* (*f x*)›
have ... $\sqsubseteq G$
by (*rule Cons.prems*(*4*))
finally
have *ccProd* (*seen* $\cup$ *set xs*) (*set zs*) $\sqsubseteq G$ **by** *this simp*

with ‹$x \in S$› *Cons.prems Cons.hyps*
have *ccFromList* [$x \leftarrow ys$. $x \notin S$] $\sqsubseteq G \wedge$ *ccProd* (*seen*) (*set ys*) $\sqsubseteq G$
apply −

apply (*rule Cons.IH*)
apply (*auto simp add*: *f-nxt-def join-below-iff interleave-ccFromList interleave-set ccProd-insert2*[**where** $S' = set\ xs$ **for** *xs*]
split: *if-splits*)
done
with ‹$x \in S$› *seen-x*
show *ccFromList* [$x \leftarrow x \# ys\ .\ x \notin S$] $\sqsubseteq G \wedge$ *ccProd seen* (*set* ($x\#ys$)) $\sqsubseteq G$
by (*auto simp add*: *ccProd-insert2*[**where** $S' = set\ xs$ **for** *xs*] *join-below-iff*)
next
case *False*

from *False Cons.prems Cons.hyps*
have *ccFromList* [$x \leftarrow ys\ .\ x \notin S$] $\sqsubseteq G \wedge$ *ccProd* ((*insert x seen*)) (*set ys*) $\sqsubseteq G$
apply −
apply (*rule Cons.IH*[**where** *seen* = *insert x seen*])
apply (*auto simp add*: *ccApprox-both join-below-iff ttree-restr-both interleave-ccFromList insert-Diff-if*
simp add: *ccProd-insert2*[**where** $S' = set\ xs$ **for** *xs*]
simp add: *ccProd-insert1*[**where** $S' = seen$])
done
moreover
from *False this*
have *ccProd* $\{x\}$ (*set ys*) $\sqsubseteq G$
by (*auto simp add*: *insert-Diff-if ccProd-insert1*[**where** $S' = seen$] *join-below-iff*)
hence *ccProd* $\{x\}$ $\{x \in set\ ys.\ x \notin S\} \sqsubseteq G$
by (*rule below-trans*[*rotated*, *OF* - *ccProd-mono2*]) *auto*
moreover
note *False seen-x*
ultimately
show *ccFromList* [$x \leftarrow x \# ys\ .\ x \notin S$] $\sqsubseteq G \wedge$ *ccProd* (*seen*) (*set* ($x \# ys$)) $\sqsubseteq G$
by (*auto simp add*: *join-below-iff simp add*: *insert-Diff-if ccProd-insert2*[**where** $S' = set\ xs$ **for** *xs*] *ccProd-insert1*[**where** $S' = seen$])
qed
qed
with ‹$xs = \text{-}$›
show *ccFromList xs* $\sqsubseteq G$ **by** *simp*
qed

lemma *wild-recursion-thunked*:
assumes *ccApprox* $t \sqsubseteq G$
assumes $\bigwedge x.\ x \notin S \Longrightarrow f\ x = empty$
assumes $\bigwedge x.\ x \in S \Longrightarrow$ *ccApprox* $(f\ x) \sqsubseteq G$
assumes $\bigwedge x.\ x \in S \Longrightarrow$ *ccProd* (*ccNeighbors x G* $- \{x\} \cap T$) (*carrier* $(f\ x)$) $\sqsubseteq G$
shows *ccApprox* (*ttree-restr* $(-S)$ (*substitute f T t*)) $\sqsubseteq G$
proof(*rule ccApprox-belowI*)
fix *xs*

def *seen* $\equiv \{\}$ :: *var set*
def *seen-T* $\equiv \{\}$ :: *var set*

assume $xs \in paths$ $(ttree\text{-}restr$ $(- S)$ $(substitute$ f T $t))$
then obtain xs' xs'' **where** $xs = [x{\leftarrow}xs' . x \notin S]$ **and** $substitute''$ f T xs'' xs' **and** $xs'' \in paths$ t
by $(auto$ $simp$ add: $filter\text{-}paths\text{-}conv\text{-}free\text{-}restr2[symmetric]$ $substitute\text{-}substitute'')$

note $this(2)$
moreover
from ‹$ccApprox$ $t \sqsubseteq G$› **and** ‹$xs'' \in paths$ t›
have $ccFromList$ $xs'' \sqsubseteq G$
by $(auto$ $simp$ add: $ccApprox\text{-}below\text{-}iff)$
hence $ccFromList$ xs'' $G|`$ $(- seen\text{-}T) \sqsubseteq G$
by $(rule$ $rev\text{-}below\text{-}trans[OF$ - $cc\text{-}restr\text{-}below\text{-}arg])$
moreover
note $assms(2)$
moreover
from $assms(3,4)$
have $\bigwedge$ x ys. $x \in S \Longrightarrow ys \in paths$ $(f$ $x) \Longrightarrow ccFromList$ $ys \sqsubseteq G$
and $\bigwedge$ x ys. $x \in S \Longrightarrow ys \in paths$ $(f$ $x) \Longrightarrow ccProd$ $(ccNeighbors$ x $G - \{x\} \cap T)$ $(set$ $ys)$ $\sqsubseteq G$
by $(auto$ $simp$ add: $ccApprox\text{-}below\text{-}iff$ $seen\text{-}T\text{-}def$ $Union\text{-}paths\text{-}carrier[symmetric]$ $cc\text{-}lub\text{-}below\text{-}iff)$
moreover
have $ccProd$ $seen$ $(set$ $xs'' - seen\text{-}T) \sqsubseteq G$ **unfolding** $seen\text{-}def$ $seen\text{-}T\text{-}def$ **by** $simp$
moreover
have $seen \cap S = \{\}$ **unfolding** $seen\text{-}def$ **by** $simp$
moreover
have $seen\text{-}T \subseteq S$ **unfolding** $seen\text{-}T\text{-}def$ **by** $simp$
moreover
have $\bigwedge$ x. $x \in seen\text{-}T \Longrightarrow f$ $x = empty$ **unfolding** $seen\text{-}T\text{-}def$ **by** $simp$
ultimately
have $ccFromList$ $[x{\leftarrow}xs' . x \notin S] \sqsubseteq G \wedge ccProd$ $(seen)$ $(set$ $xs' - seen\text{-}T) \sqsubseteq G$
proof$(induction$ f T xs'' xs' $arbitrary$: $seen$ $seen\text{-}T$ $rule$: $substitute''.induct[case\text{-}names$ Nil $Cons])$
case Nil **thus** $?case$ **by** $simp$
next
case $(Cons$ zs f x xs' xs T $ys)$

let $?seen\text{-}T = if$ $x \in T$ $then$ $insert$ x $seen\text{-}T$ $else$ $seen\text{-}T$
have $subset$: $- insert$ x $seen\text{-}T \subseteq - seen\text{-}T$ **by** $auto$
have $subset2$: set $xs \cap - insert$ x $seen\text{-}T \subseteq insert$ x $(set$ $xs) \cap - seen\text{-}T$ **by** $auto$
have $subset3$: set $zs \cap - insert$ x $seen\text{-}T \subseteq set$ zs **by** $auto$
have $subset4$: set $xs \cap - seen\text{-}T \subseteq insert$ x $(set$ $xs) \cap - seen\text{-}T$ **by** $auto$
have $subset5$: set $zs \cap - seen\text{-}T \subseteq set$ zs **by** $auto$
have $subset6$: set $ys - seen\text{-}T \subseteq (set$ $ys - ?seen\text{-}T) \cup \{x\}$ **by** $auto$

show $?case$
proof$(cases$ $x \in seen\text{-}T)$
assume $x \in seen\text{-}T$

have [*simp*]: *f x* = *empty* **using** ‹$x \in$ *seen-T*› *Cons.prems* **by** *auto*
have [*simp*]: *f-nxt f T x* = *f* **by** (*auto simp add*: *f-nxt-def split*:*if-splits*)
have [*simp*]: *zs* = [] **using** ‹*zs* ∈ *paths* (*f x*)› **by** *simp*
have [*simp*]: *xs'* = *xs* **using** ‹*xs'* ∈ *xs* ⊗ *zs*› **by** *simp*
have [*simp*]: *x* ∈ *S* **using** ‹*x* ∈ *seen-T*› *Cons.prems* **by** *auto*

from *Cons.hyps Cons.prems*
have *ccFromList* [*x*←*ys* . *x* ∉ *S*] ⊑ *G* ∧ *ccProd seen* (*set ys* − *seen-T*) ⊑ *G*
apply −
apply (*rule Cons.IH*[**where** *seen-T* = *seen-T*])
apply (*auto simp add*: *join-below-iff Diff-eq*)
apply (*erule below-trans*[*OF ccProd-mono*[*OF order-refl subset4*]])
done
thus *?thesis* **using** ‹*x* ∈ *seen-T*› **by** *simp*
next
assume *x* ∉ *seen-T*

have *seen-x*: *ccProd seen* {*x*} ⊑ *G*
using ‹*ccProd seen* (*set* (*x* # *xs*) − *seen-T*) ⊑ *G*› ‹*x* ∉ *seen-T*›
by (*auto simp add*: *insert-Diff-if ccProd-insert2*[**where** *S'* = *set xs* − *seen-T* **for** *xs*] *join-below-iff*)

show *?case*
proof(*cases x* ∈ *S*)
case *True*

from ‹*cc-restr* (− *seen-T*) (*ccFromList* (*x* # *xs*)) ⊑ *G*›
have *ccProd* {*x*} (*set xs* − *seen-T*) ⊑ *G* **using** ‹*x* ∉ *seen-T*› **by** (*auto simp add*: *join-below-iff Diff-eq*)
hence *set xs* − *seen-T* ⊆ *ccNeighbors x G* **by** *transfer auto*
moreover

from *seen-x*
have *seen* ⊆ *ccNeighbors x G* **by** (*simp add*: *subset-ccNeighbors ccProd-comm*)
moreover
have *x* ∉ *seen* **using** *True* ‹*seen* ∩ *S* = {}› **by** *auto*

ultimately
have *seen* ∪ (*set xs* ∩ − *?seen-T*) ⊆ *ccNeighbors x G* − {*x*}∩*T* **by** *auto*
hence *ccProd* (*seen* ∪ (*set xs* ∩ − *?seen-T*)) (*set zs*) ⊑ *ccProd* (*ccNeighbors x G* − {*x*}∩*T*) (*set zs*)
by (*rule ccProd-mono1*)
also
from ‹*x* ∈ *S*› ‹*zs* ∈ *paths* (*f x*)›
have ... ⊑ *G*
by (*rule Cons.prems*(*4*))
finally
have *ccProd* (*seen* ∪ (*set xs* ∩ − *?seen-T*)) (*set zs*) ⊑ *G* **by** *this simp*

with ‹$x \in S$› *Cons.prems Cons.hyps*(*1*,*2*)
have *ccFromList* [$x \leftarrow ys$. $x \notin S$] $\sqsubseteq$ G $\wedge$ *ccProd* (*seen*) (*set ys* $-$ *?seen-T*) $\sqsubseteq$ G
apply $-$
apply (*rule Cons.IH*[**where** *seen-T* = *?seen-T*])
apply (*auto simp add*: *Un-Diff Int-Un-distrib2 Diff-eq f-nxt-def join-below-iff interleave-ccFromList interleave-set ccProd-insert2*[**where** S' = *set xs* **for** *xs*]
split: *if-splits*)
apply (*erule below-trans*[*OF cc-restr-mono1*[*OF subset*]])
apply (*rule below-trans*[*OF cc-restr-below-arg*], *simp*)
apply (*erule below-trans*[*OF ccProd-mono*[*OF order-refl Int-lower1*]])
apply (*rule below-trans*[*OF cc-restr-below-arg*], *simp*)
apply (*erule below-trans*[*OF ccProd-mono*[*OF order-refl Int-lower1*]])
apply (*erule below-trans*[*OF ccProd-mono*[*OF order-refl subset2*]])
apply (*erule below-trans*[*OF ccProd-mono*[*OF order-refl subset3*]])
apply (*erule below-trans*[*OF ccProd-mono*[*OF order-refl subset4*]])
apply (*erule below-trans*[*OF ccProd-mono*[*OF order-refl subset5*]])
done
with ‹$x \in S$› *seen-x* ‹$x \notin$ *seen-T*›
show *ccFromList* [$x \leftarrow x$ # ys . $x \notin S$] $\sqsubseteq$ G $\wedge$ *ccProd seen* (*set* (x#ys) $-$ *seen-T*) $\sqsubseteq$ G

apply (*auto simp add*: *insert-Diff-if ccProd-insert2*[**where** S' = *set ys* $-$ *seen-T* **for** *xs*] *join-below-iff*)
apply (*rule below-trans*[*OF ccProd-mono*[*OF order-refl subset6*]])
apply (*subst ccProd-union2*)
apply (*auto simp add*: *join-below-iff*)
done
next
case *False*

from *False Cons.prems Cons.hyps*
have *ccFromList* [$x \leftarrow ys$. $x \notin S$] $\sqsubseteq$ G $\wedge$ *ccProd* ((*insert x seen*)) (*set ys* $-$ *seen-T*) $\sqsubseteq$ G
apply $-$
apply (*rule Cons.IH*[**where** *seen* = *insert x seen* **and** *seen-T* = *seen-T*])
apply (*auto simp add*: ‹$x \notin$ *seen-T*› *Diff-eq ccApprox-both join-below-iff ttree-restr-both interleave-ccFromList insert-Diff-if*
simp add: *ccProd-insert2*[**where** S' = *set xs* $\cap$ $-$ *seen-T* **for** *xs*]
simp add: *ccProd-insert1*[**where** S' = *seen*])
done
moreover
{
from *False this*
have *ccProd* $\{x\}$ (*set ys* $-$ *seen-T*) $\sqsubseteq$ G
by (*auto simp add*: *insert-Diff-if ccProd-insert1*[**where** S' = *seen*] *join-below-iff*)
hence *ccProd* $\{x\}$ $\{x \in$ *set ys* $-$ *seen-T*. $x \notin S\}$ $\sqsubseteq$ G
by (*rule below-trans*[*rotated*, *OF* - *ccProd-mono2*]) *auto*
also have $\{x \in$ *set ys* $-$ *seen-T*. $x \notin S\}$ = $\{x \in$ *set ys*. $x \notin S\}$
using ‹*seen-T* $\subseteq$ S› **by** *auto*
finally

have $ccProd$ $\{x\}$ $\{x \in set\ ys.\ x \notin S\} \sqsubseteq G$**.**
}
moreover
note *False seen-x*
ultimately
show $ccFromList$ $[x \leftarrow x \# ys\ .\ x \notin S] \sqsubseteq G \wedge ccProd\ (seen)\ (set\ (x \# ys) - seen\text{-}T) \sqsubseteq G$
by (*auto simp add*: *join-below-iff simp add*: *insert-Diff-if ccProd-insert2*[**where** $S' = set\ ys - seen\text{-}T$ **for** *xs*] *ccProd-insert1*[**where** $S' = seen$])
qed
qed
qed
with ‹$xs = -$›
show $ccFromList\ xs \sqsubseteq G$ **by** *simp*
qed

inductive-set *valid-lists* :: *var set* $\Rightarrow$ *CoCalls* $\Rightarrow$ *var list set*
for *S G*
where $[] \in valid\text{-}lists\ S\ G$
$|$ $set\ xs \subseteq ccNeighbors\ x\ G \Longrightarrow xs \in valid\text{-}lists\ S\ G \Longrightarrow x \in S \Longrightarrow x\#xs \in valid\text{-}lists\ S\ G$

inductive-simps *valid-lists-simps*[*simp*]: $[] \in valid\text{-}lists\ S\ G$ $(x\#xs) \in valid\text{-}lists\ S\ G$
inductive-cases *vald-lists-ConsE*: $(x\#xs) \in valid\text{-}lists\ S\ G$

lemma *valid-lists-downset-aux*:
$xs \in valid\text{-}lists\ S\ CoCalls \Longrightarrow butlast\ xs \in valid\text{-}lists\ S\ CoCalls$
by (*induction xs*) (*auto dest*: *in-set-butlastD*)

lemma *valid-lists-subset*: $xs \in valid\text{-}lists\ S\ G \Longrightarrow set\ xs \subseteq S$
by (*induction rule*: *valid-lists.induct*) *auto*

lemma *valid-lists-mono1*:
assumes $S \subseteq S'$
shows $valid\text{-}lists\ S\ G \subseteq valid\text{-}lists\ S'\ G$
proof
fix *xs*
assume $xs \in valid\text{-}lists\ S\ G$
thus $xs \in valid\text{-}lists\ S'\ G$
by (*induction rule*: *valid-lists.induct*) (*auto dest*: *set-mp*[*OF assms*])
qed

lemma *valid-lists-chain1*:
assumes *chain Y*
assumes $xs \in valid\text{-}lists\ (UNION\ UNIV\ Y)\ G$
shows $\exists\ i.\ xs \in valid\text{-}lists\ (Y\ i)\ G$
proof−
note ‹*chain Y*›
moreover

```
  from assms(2)
  have set xs ⊆ UNION UNIV Y by (rule valid-lists-subset)
  moreover
  have finite (set xs) by simp
  ultimately
  have ∃ i. set xs ⊆ Y i by (rule finite-subset-chain)
  then obtain i where set xs ⊆ Y i..

  from assms(2) this
  have xs ∈ valid-lists (Y i) G by (induction rule:valid-lists.induct) auto
  thus ?thesis..
qed

lemma valid-lists-chain2:
   assumes chain Y
   assumes xs ∈ valid-lists S (⨆ i. Y i)
   shows ∃ i. xs ∈ valid-lists S (Y i)
using assms(2)
proof(induction rule:valid-lists.induct[case-names Nil Cons])
  case Nil thus ?case by simp
next
  case (Cons xs x)

  from ‹chain Y›
  have chain (λ i. ccNeighbors x (Y i))
    apply (rule ch2ch-monofun[OF monofunI, rotated])
    unfolding below-set-def
    by (rule ccNeighbors-mono)
  moreover
  from ‹set xs ⊆ ccNeighbors x (⨆ i. Y i)›
  have set xs ⊆ (⋃ i. ccNeighbors x (Y i))
    by (simp add: lub-set)
  moreover
  have finite (set xs) by simp
  ultimately
  have ∃ i. set xs ⊆ ccNeighbors x (Y i) by (rule finite-subset-chain)
  then obtain i where i: set xs ⊆ ccNeighbors x (Y i)..

  from Cons.IH
  obtain j where j: xs ∈ valid-lists S (Y j)..

  from i
  have set xs ⊆ ccNeighbors x (Y (max i j))
   by (rule order-trans[OF - ccNeighbors-mono[OF chain-mono[OF ‹chain Y› max.cobounded1]]])
  moreover
  from j
  have xs ∈ valid-lists S (Y (max i j))
    by (induction rule: valid-lists.induct)
      (auto del: subsetI elim: order-trans[OF - ccNeighbors-mono[OF chain-mono[OF ‹chain Y›
```

$max.cobounded2$]]])
  moreover
  note ‹$x \in S$›
  ultimately
  have x # $xs \in$ *valid-lists* S (Y (*max* i j)) **by** *rule*
  thus *?case*..
qed

lemma *valid-lists-cc-restr*: *valid-lists* S G = *valid-lists* S (*cc-restr* S G)
proof(*rule set-eqI*)
  fix *xs*
  show (*xs* $\in$ *valid-lists* S G) = (*xs* $\in$ *valid-lists* S (*cc-restr* S G))
    by (*induction xs*) (*auto dest*: *set-mp*[*OF valid-lists-subset*])
qed

lemma *interleave-valid-list*:
  $xs \in ys \otimes zs \Longrightarrow ys \in$ *valid-lists* S $G \Longrightarrow zs \in$ *valid-lists* S' $G' \Longrightarrow xs \in$ *valid-lists* $(S \cup S')$ $(G \sqcup (G' \sqcup$ *ccProd* S $S'))$
proof (*induction rule*:*interleave-induct*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*left ys zs xs x*)

  from ‹x # $ys \in$ *valid-lists* S G›
  have $x \in S$ **and** *set ys* $\subseteq$ *ccNeighbors* x G **and** $ys \in$ *valid-lists* S G
    by *auto*

  from ‹$xs \in ys \otimes zs$›
  have *set xs* = *set ys* $\cup$ *set zs* **by** (*rule interleave-set*)
  with ‹*set ys* $\subseteq$ *ccNeighbors* x G› *valid-lists-subset*[*OF* ‹$zs \in$ *valid-lists* S' G'›]
  have *set xs* $\subseteq$ *ccNeighbors* x $(G \sqcup (G' \sqcup$ *ccProd* S $S'))$
    by (*auto simp add*: *ccNeighbors-ccProd* ‹$x \in S$›)
  moreover
  from ‹$ys \in$ *valid-lists* S G› ‹$zs \in$ *valid-lists* S' G'›
  have $xs \in$ *valid-lists* $(S \cup S')$ $(G \sqcup (G' \sqcup$ *ccProd* S $S'))$
    by (*rule left.IH*)
  moreover
  from ‹$x \in S$›
  have $x \in S \cup S'$ **by** *simp*
  ultimately
  show *?case*..
next
  case (*right ys zs xs x*)

  from ‹x # $zs \in$ *valid-lists* S' G'›
  have $x \in S'$ **and** *set zs* $\subseteq$ *ccNeighbors* x G' **and** $zs \in$ *valid-lists* S' G'
    by *auto*

from ‹$xs \in ys \otimes zs$›
have $set\ xs = set\ ys \cup set\ zs$ **by** (*rule interleave-set*)
with ‹$set\ zs \subseteq ccNeighbors\ x\ G'$› *valid-lists-subset*[*OF* ‹$ys \in valid\text{-}lists\ S\ G$›]
have $set\ xs \subseteq ccNeighbors\ x\ (G \sqcup (G' \sqcup ccProd\ S\ S'))$
by (*auto simp add*: *ccNeighbors-ccProd* ‹$x \in S'$›)
moreover
from ‹$ys \in valid\text{-}lists\ S\ G$› ‹$zs \in valid\text{-}lists\ S'\ G'$›
have $xs \in valid\text{-}lists\ (S \cup S')\ (G \sqcup (G' \sqcup ccProd\ S\ S'))$
by (*rule right.IH*)
moreover
from ‹$x \in S'$›
have $x \in S \cup S'$ **by** *simp*
ultimately
show *?case*..
qed

lemma *interleave-valid-list'*:
$xs \in valid\text{-}lists\ (S \cup S')\ G \Longrightarrow \exists\ ys\ zs.\ xs \in ys \otimes zs \wedge ys \in valid\text{-}lists\ S\ G \wedge zs \in valid\text{-}lists\ S'\ G$
proof(*induction rule*: *valid-lists.induct*[*case-names Nil Cons*])
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons xs x*)
then obtain *ys zs* **where** $xs \in ys \otimes zs$ $ys \in valid\text{-}lists\ S\ G$ $zs \in valid\text{-}lists\ S'\ G$ **by** *auto*

from ‹$xs \in ys \otimes zs$› **have** $set\ xs = set\ ys \cup set\ zs$ **by** (*rule interleave-set*)
with ‹$set\ xs \subseteq ccNeighbors\ x\ G$›
have $set\ ys \subseteq ccNeighbors\ x\ G$ **and** $set\ zs \subseteq ccNeighbors\ x\ G$ **by** *auto*

from ‹$x \in S \cup S'$›
show *?case*
proof
assume $x \in S$
with ‹$set\ ys \subseteq ccNeighbors\ x\ G$› ‹$ys \in valid\text{-}lists\ S\ G$›
have $x\ \#\ ys \in valid\text{-}lists\ S\ G$
by *rule*
moreover
from ‹$xs \in ys \otimes zs$›
have $x\#xs \in x\#ys \otimes zs$..
ultimately
show *?thesis* **using** ‹$zs \in valid\text{-}lists\ S'\ G$› **by** *blast*
next
assume $x \in S'$
with ‹$set\ zs \subseteq ccNeighbors\ x\ G$› ‹$zs \in valid\text{-}lists\ S'\ G$›
have $x\ \#\ zs \in valid\text{-}lists\ S'\ G$
by *rule*
moreover
from ‹$xs \in ys \otimes zs$›
have $x\#xs \in ys \otimes x\#zs$..

```
    ultimately
    show ?thesis using ‹ys ∈ valid-lists S G› by blast
  qed
qed

lemma many-calls-valid-list:
  xs ∈ valid-lists {x} (ccProd {x} {x}) ⟹ xs ∈ range (λn. replicate n x)
  by (induction rule: valid-lists.induct) (auto, metis UNIV-I image-iff replicate-Suc)

lemma filter-valid-lists:
  xs ∈ valid-lists S G ⟹ filter P xs ∈ valid-lists {a ∈ S. P a} G
by (induction rule:valid-lists.induct) auto

lift-definition ccTTree :: var set ⇒ CoCalls ⇒ var ttree is λ S G. valid-lists S G
  by (auto intro: valid-lists-downset-aux)

lemma paths-ccTTree[simp]: paths (ccTTree S G) = valid-lists S G by transfer auto

lemma carrier-ccTTree[simp]: carrier (ccTTree S G) = S
  apply transfer
  apply (auto dest: valid-lists-subset)
  apply (rule-tac x = [x] in bexI)
  apply auto
  done

lemma valid-lists-ccFromList:
  xs ∈ valid-lists S G ⟹ ccFromList xs ⊑ cc-restr S G
by (induction rule:valid-lists.induct)
  (auto simp add: join-below-iff subset-ccNeighbors ccProd-below-cc-restr elim: set-mp[OF valid-lists-subset])

lemma ccApprox-ccTTree[simp]: ccApprox (ccTTree S G) = cc-restr S G
proof (transfer' fixing: S G, rule below-antisym)
  show lub (ccFromList ‘ valid-lists S G) ⊑ cc-restr S G
    apply (rule is-lub-thelub-ex)
    apply (metis coCallsLub-is-lub)
    apply (rule is-ubI)
    apply clarify
    apply (erule valid-lists-ccFromList)
    done
next
  show cc-restr S G ⊑ lub (ccFromList ‘ valid-lists S G)
  proof (rule below-CoCallsI)
    fix x y
    have x−−y∈(ccFromList [y,x]) by simp
    moreover
    assume x−−y∈(cc-restr S G)
    hence [y,x] ∈ valid-lists S G  by (auto simp add: elem-ccNeighbors)
    ultimately
    show x−−y∈(lub (ccFromList ‘ valid-lists S G))
```

```
      by (rule in-CoCallsLubI[OF - imageI])
  qed
qed

lemma below-ccTTreeI:
  assumes carrier t ⊆ S and ccApprox t ⊑ G
  shows t ⊑ ccTTree S G
unfolding paths-mono-iff[symmetric] below-set-def
proof
  fix xs
  assume xs ∈ paths t
  with assms
  have xs ∈ valid-lists S G
  proof(induction xs arbitrary : t)
  case Nil thus ?case by simp
  next
  case (Cons x xs)
    from ‹x # xs ∈ paths t›
    have possible t x and xs ∈ paths (nxt t x) by (auto simp add: Cons-path)

    have ccProd {x} (set xs) ⊑ ccFromList (x # xs) by simp
    also
    from ‹x # xs ∈ paths t›
    have ... ⊑ ccApprox t
      by (rule ccFromList-below-ccApprox)
    also
    note ‹ccApprox t ⊑ G›
    finally
    have ccProd {x} (set xs) ⊑ G by this simp-all
    hence set xs ⊆ ccNeighbors x G unfolding subset-ccNeighbors.
    moreover
    have xs ∈ valid-lists S G
    proof(rule Cons.IH)
      show xs ∈ paths (nxt t x) by fact
    next
      from ‹carrier t ⊆ S›
      show carrier (nxt t x) ⊆ S
        by (rule order-trans[OF carrier-nxt-subset])
    next
      from ‹ccApprox t ⊑ G›
      show ccApprox (nxt t x) ⊑ G
        by (rule below-trans[OF ccApprox-nxt-below])
    qed
    moreover
    from ‹carrier t ⊆ S› and ‹possible t x›
    have x ∈ S by (rule carrier-possible-subset)
    ultimately
    show ?case..
  qed
```

thus *xs* $\in$ *paths* (*ccTTree S G*) **by** (*metis paths-ccTTree*)
qed

lemma *ccTTree-mono1*:
$S \subseteq S' \Longrightarrow$ *ccTTree S G* $\sqsubseteq$ *ccTTree S' G*
by (*rule below-ccTTreeI*) (*auto simp add: cc-restr-below-arg*)

lemma *cont-ccTTree1*:
cont (λ *S*. *ccTTree S G*)
apply (*rule contI2*)
apply (*rule monofunI*)
apply (*erule ccTTree-mono1*[*folded below-set-def*])

apply (*rule ttree-belowI*)
apply (*simp add*: *paths-Either lub-set lub-is-either*)
apply (*drule* (*1*) *valid-lists-chain1*[*rotated*])
apply *simp*
done

lemma *ccTTree-mono2*:
$G \sqsubseteq G' \Longrightarrow$ *ccTTree S G* $\sqsubseteq$ *ccTTree S G'*
apply (*rule ttree-belowI*)
apply *simp*
apply (*induct-tac rule*:*valid-lists.induct*) **apply** *assumption*
apply *simp*
apply *simp*
apply (*erule* (*1*) *order-trans*[*OF - ccNeighbors-mono*])
done

lemma *ccTTree-mono*:
$S \subseteq S' \Longrightarrow G \sqsubseteq G' \Longrightarrow$ *ccTTree S G* $\sqsubseteq$ *ccTTree S' G'*
by (*metis below-trans*[*OF ccTTree-mono1 ccTTree-mono2*])

lemma *cont-ccTTree2*:
cont (*ccTTree S*)
apply (*rule contI2*)
apply (*rule monofunI*)
apply (*erule ccTTree-mono2*)

apply (*rule ttree-belowI*)
apply (*simp add*: *paths-Either lub-set lub-is-either*)
apply (*drule* (*1*) *valid-lists-chain2*)
apply *simp*
done

lemmas *cont-ccTTree* = *cont-compose2*[**where** *c* = *ccTTree*, *OF cont-ccTTree1 cont-ccTTree2*, *simp*, *cont2cont*]

lemma *ccTTree-below-singleI*:
  assumes $S \cap S' = \{\}$
  shows *ccTTree S G* $\sqsubseteq$ *singles S'*
proof−
  {
  fix *xs x*
  assume *xs* $\in$ *valid-lists S G* **and** $x \in S'$
  from *this assms*
  have *length* $[x' \leftarrow xs\ .\ x' = x] \leq$ *Suc 0*
  by(*induction rule*: *valid-lists.induct*[*case-names Nil Cons*]) *auto*
  }
  thus *?thesis* **by** *transfer auto*
qed

lemma *ccTTree-cc-restr*: *ccTTree S G* = *ccTTree S* (*cc-restr S G*)
  by *transfer'* (*rule valid-lists-cc-restr*)

lemma *ccTTree-cong-below*: *cc-restr S G* $\sqsubseteq$ *cc-restr S G'* $\Longrightarrow$ *ccTTree S G* $\sqsubseteq$ *ccTTree S G'*
  by (*metis ccTTree-mono2 ccTTree-cc-restr*)

lemma *ccTTree-cong*: *cc-restr S G* = *cc-restr S G'* $\Longrightarrow$ *ccTTree S G* = *ccTTree S G'*
  by (*metis ccTTree-cc-restr*)

lemma *either-ccTTree*:
  ccTTree S G $\oplus\oplus$ *ccTTree S' G'* $\sqsubseteq$ *ccTTree* $(S \cup S')$ $(G \sqcup G')$
  by (*auto intro*!: *either-belowI ccTTree-mono*)

lemma *interleave-ccTTree*:
   ccTTree S G $\otimes\otimes$ *ccTTree S' G'* $\sqsubseteq$ *ccTTree* $(S \cup S')$ $(G \sqcup G' \sqcup$ *ccProd S S'*$)$
   by *transfer'* (*auto*, *erule* (*2*) *interleave-valid-list*)

lemma *interleave-ccTTree'*:
   ccTTree $(S \cup S')$ *G* $\sqsubseteq$ *ccTTree S G* $\otimes\otimes$ *ccTTree S' G*
   by *transfer'* (*auto dest*!: *interleave-valid-list'*)

lemma *many-calls-ccTTree*:
  shows *many-calls x* = *ccTTree* $\{x\}$ (*ccProd* $\{x\}$ $\{x\}$)
  apply(*transfer'*)
  apply (*auto intro*: *many-calls-valid-list*)
  apply (*induct-tac n*)
  apply (*auto simp add*: *ccNeighbors-ccProd*)
  done

lemma *filter-valid-lists'*:
  xs $\in$ *valid-lists* $\{x' \in S.\ P\ x'\}$ *G* $\Longrightarrow$ *xs* $\in$ *filter P* ' *valid-lists S G*
proof (*induction xs*)

case *Nil* **thus** *?case* **by** *auto* (*metis filter.simps*(*1*) *image-iff valid-lists-simps*(*1*))
next
case (*Cons x xs*)
from *Cons.prems*
have *set xs* $\subseteq$ *ccNeighbors x G* **and** *xs* $\in$ *valid-lists* $\{x' \in S.\ P\ x'\}$ *G* **and** $x \in S$ **and** *P x* **by** *auto*

from *this*(*2*) **have** *set xs* $\subseteq \{x' \in S.\ P\ x'\}$ **by** (*rule valid-lists-subset*)
hence $\forall\, x \in$ *set xs*. *P x* **by** *auto*
hence [*simp*]: *filter P xs* = *xs* **by** (*rule filter-True*)

from *Cons.IH*[*OF* ‹*xs* $\in$ -›]
have *xs* $\in$ *filter P* ‘ *valid-lists S G*.

from ‹*xs* $\in$ *valid-lists* $\{x' \in S.\ P\ x'\}$ *G*›
have *xs* $\in$ *valid-lists S G* **by** (*rule set-mp*[*OF valid-lists-mono1*, *rotated*]) *auto*

from ‹*set xs* $\subseteq$ *ccNeighbors x G*› *this* ‹$x \in S$›
have *x* # *xs* $\in$ *valid-lists S G* **by** *rule*

hence *filter P* (*x* # *xs*) $\in$ *filter P* ‘ *valid-lists S G* **by** (*rule imageI*)
thus *?case* **using** ‹*P x*› ‹*filter P xs* =*xs*› **by** *simp*
qed

lemma *without-ccTTree*[*simp*]:
without x (*ccTTree S G*) = *ccTTree* ($S - \{x\}$) *G*
by (*transfer' fixing*: *x*) (*auto dest*: *filter-valid-lists' filter-valid-lists*[**where** *P* = ($\lambda\ x'.\ x' \neq x$)] *simp add*: *set-diff-eq*)

lemma *ttree-restr-ccTTree*[*simp*]:
ttree-restr S' (*ccTTree S G*) = *ccTTree* ($S \cap S'$) *G*
by (*transfer' fixing*: *S'*) (*auto dest*: *filter-valid-lists' filter-valid-lists*[**where** *P* = ($\lambda\ x'.\ x' \in S'$)] *simp add*:*Int-def*)

lemma *repeatable-ccTTree-ccSquare*: $S \subseteq S' \Longrightarrow$ *repeatable* (*ccTTree S* (*ccSquare S'*))
unfolding *repeatable-def*
by *transfer* (*auto simp add*:*ccNeighbors-ccSquare dest*: *set-mp*[*OF valid-lists-subset*])

An alternative definition

inductive *valid-lists'* :: *var set* $\Rightarrow$ *CoCalls* $\Rightarrow$ *var set* $\Rightarrow$ *var list* $\Rightarrow$ *bool*
for *S G*
where *valid-lists' S G prefix* []
| *prefix* $\subseteq$ *ccNeighbors x G* $\Longrightarrow$ *valid-lists' S G* (*insert x prefix*) *xs* $\Longrightarrow x \in S \Longrightarrow$ *valid-lists' S G prefix* (*x*#*xs*)

inductive-simps *valid-lists'-simps*[*simp*]: *valid-lists' S G prefix* [] *valid-lists' S G prefix* (*x*#*xs*)
inductive-cases *vald-lists'-ConsE*: *valid-lists' S G prefix* (*x*#*xs*)

lemma *valid-lists-valid-lists'*:

$xs \in valid\text{-}lists\ S\ G \Longrightarrow ccProd\ prefix\ (set\ xs) \sqsubseteq G \Longrightarrow valid\text{-}lists'\ S\ G\ prefix\ xs$
proof(*induction arbitrary*: *prefix rule*: *valid-lists.induct*[*case-names Nil Cons*])
 case *Nil* **thus** *?case* **by** *simp*
next
 case (*Cons xs x*)

 from *Cons.prems Cons.hyps Cons.IH*[**where** *prefix* = *insert x prefix*]
 show *?case*
 by (*auto simp add*: *insert-is-Un*[**where** *A* = *set xs*] *insert-is-Un*[**where** *A* = *prefix*]
 join-below-iff subset-ccNeighbors elem-ccNeighbors ccProd-comm simp del:
Un-insert-left)
qed

lemma *valid-lists'-valid-lists-aux*:
 $valid\text{-}lists'\ S\ G\ prefix\ xs \Longrightarrow x \in prefix \Longrightarrow ccProd\ (set\ xs)\ \{x\} \sqsubseteq G$
proof(*induction rule*: *valid-lists'.induct*[*case-names Nil Cons*])
 case *Nil* **thus** *?case* **by** *simp*
next
 case (*Cons prefix x xs*)
 thus *?case*
 apply (*auto simp add*: *ccProd-insert2*[**where** *S'* = *prefix*] *ccProd-insert1*[**where** *S'* = *set*
xs] *join-below-iff subset-ccNeighbors*)
 by (*metis Cons.hyps*(*1*) *dual-order.trans empty-subsetI insert-subset subset-ccNeighbors*)
qed

lemma *valid-lists'-valid-lists*:
 $valid\text{-}lists'\ S\ G\ prefix\ xs \Longrightarrow xs \in valid\text{-}lists\ S\ G$
proof(*induction rule*: *valid-lists'.induct*[*case-names Nil Cons*])
 case *Nil* **thus** *?case* **by** *simp*
next
 case (*Cons prefix x xs*)
 thus *?case*
 by (*auto simp add*: *insert-is-Un*[**where** *A* = *set xs*] *insert-is-Un*[**where** *A* = *prefix*]
 join-below-iff subset-ccNeighbors elem-ccNeighbors ccProd-comm simp del:
Un-insert-left
 intro: *valid-lists'-valid-lists-aux*)
qed

Yet another definition

lemma *valid-lists-characterization*:
 $xs \in valid\text{-}lists\ S\ G \longleftrightarrow set\ xs \subseteq S \wedge (\forall n.\ ccProd\ (set\ (take\ n\ xs))\ (set\ (drop\ n\ xs)) \sqsubseteq G)$
proof(*safe*)
 fix *x*
 assume $xs \in valid\text{-}lists\ S\ G$
 from *valid-lists-subset*[*OF this*]
 show $x \in set\ xs \Longrightarrow x \in S$ **by** *auto*
next
 fix *n*
 assume $xs \in valid\text{-}lists\ S\ G$

thus *ccProd* (*set* (*take n xs*)) (*set* (*drop n xs*)) $\sqsubseteq$ *G*
proof(*induction arbitrary*: *n rule*: *valid-lists.induct*[*case-names Nil Cons*])
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons xs x*)
show *?case*
proof(*cases n*)
case *0* **thus** *?thesis* **by** *simp*
next
case (*Suc n*)
with *Cons.hyps Cons.IH*[**where** $n = n$]
show *?thesis*
apply (*auto simp add*: *ccProd-insert1*[**where** $S' = set\ xs$ **for** *xs*] *join-below-iff subset-ccNeighbors*)
by (*metis dual-order.trans set-drop-subset subset-ccNeighbors*)
qed
qed
next
assume *set xs* $\subseteq$ *S*
and $\forall$ *n*. *ccProd* (*set* (*take n xs*)) (*set* (*drop n xs*)) $\sqsubseteq$ *G*
thus *xs* $\in$ *valid-lists S G*
proof (*induction xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
from ‹$\forall$ *n*. *ccProd* (*set* (*take n* (*x* # *xs*))) (*set* (*drop n* (*x* # *xs*))) $\sqsubseteq$ *G*›
have $\forall$ *n*. *ccProd* (*set* (*take n xs*)) (*set* (*drop n xs*)) $\sqsubseteq$ *G*
by −(*rule*, *erule-tac* $x = Suc\ n$ **in** *allE*, *auto simp add*: *ccProd-insert1*[**where** $S' = set\ xs$ **for** *xs*] *join-below-iff*)
from *Cons.prems Cons.IH*[*OF* - *this*]
have *xs* $\in$ *valid-lists S G* **by** *auto*
with *Cons.prems*(*1*) *spec*[*OF* ‹$\forall$ *n*. *ccProd* (*set* (*take n* (*x* # *xs*))) (*set* (*drop n* (*x* # *xs*))) $\sqsubseteq$ *G*›, **where** $x = 1$]
show *?case* **by** (*simp add*: *subset-ccNeighbors*)
qed
qed

end

89 CoCallImplTTree.tex

theory *CoCallImplTTree*
imports *TTreeAnalysisSig Env−Set−Cpo CoCallAritySig CoCallGraph−TTree*
begin

context *CoCallArity*
begin
definition *Texp* :: *exp* $\Rightarrow$ *Arity* $\rightarrow$ *var ttree*
where *Texp e* = (Λ *a*. *ccTTree* (*edom* (*Aexp e* ·*a*)) (*ccExp e*·*a*))

lemma *Texp-simp*: *Texp* $e \cdot a$ = *ccTTree* (*edom* (*Aexp* $e \cdot a$)) (*ccExp* $e \cdot a$)
unfolding *Texp-def*
by *simp*

sublocale *TTreeAnalysis Texp*.
end

end

90 Cardinality-Domain-Lists.tex

theory *Cardinality−Domain−Lists*
imports *Vars Nominal−HOLCF Env Cardinality−Domain Set−Cpo Env−Set−Cpo*
begin

fun *no-call-in-path* **where**
no-call-in-path x [] $\longleftrightarrow$ *True*
| *no-call-in-path* x ($y\#xs$) $\longleftrightarrow$ $y \neq x \wedge$ *no-call-in-path* x xs

fun *one-call-in-path* **where**
one-call-in-path x [] $\longleftrightarrow$ *True*
| *one-call-in-path* x ($y\#xs$) $\longleftrightarrow$ (*if* $x = y$ *then* *no-call-in-path* x xs *else* *one-call-in-path* x xs)

lemma *no-call-in-path-set-conv*:
no-call-in-path x p $\longleftrightarrow$ $x \notin$ *set* p
by(*induction* p) *auto*

lemma *one-call-in-path-filter-conv*:
one-call-in-path x p $\longleftrightarrow$ *length* (*filter* (λ x'. $x' = x$) p) $\leq$ *1*
by(*induction* p) (*auto simp add*: *no-call-in-path-set-conv filter-empty-conv*)

lemma *no-call-in-tail*: *no-call-in-path* x (*tl* p) $\longleftrightarrow$ (*no-call-in-path* x p $\vee$ *one-call-in-path* x p $\wedge$ *hd* $p = x$)
by(*induction* p) *auto*

lemma *no-imp-one*: *no-call-in-path* x p $\Longrightarrow$ *one-call-in-path* x p
by (*induction* p) *auto*

lemma *one-imp-one-tail*: *one-call-in-path* x p $\Longrightarrow$ *one-call-in-path* x (*tl* p)
by (*induction* p) (*auto split*: *if-splits intro*: *no-imp-one*)

lemma *more-than-one-setD*:
$\neg$ *one-call-in-path* x p $\Longrightarrow$ $x \in$ *set* p
by (*induction* p) (*auto split*: *if-splits*)

lemma *no-call-in-path*[*eqvt*]: *no-call-in-path p x* $\Longrightarrow$ *no-call-in-path* $(\pi \cdot p)$ $(\pi \cdot x)$
by (*induction p x rule*: *no-call-in-path.induct*) *auto*

lemma *one-call-in-path*[*eqvt*]: *one-call-in-path p x* $\Longrightarrow$ *one-call-in-path* $(\pi \cdot p)$ $(\pi \cdot x)$
by (*induction p x rule*: *one-call-in-path.induct*) (*auto dest*: *no-call-in-path*)

definition *pathCard* :: *var list* $\Rightarrow$ (*var* $\Rightarrow$ *two*)
where *pathCard p x* = (*if no-call-in-path x p then none else* (*if one-call-in-path x p then once else many*))

lemma *pathCard-Nil*[*simp*]: *pathCard* [] = $\bot$
by *rule* (*simp add*: *pathCard-def*)

lemma *pathCard-Cons*[*simp*]: *pathCard* (*x#xs*) *x* = *two-add*·*once*·(*pathCard xs x*)
unfolding *pathCard-def*
by (*auto simp add*: *two-add-simp*)

lemma *pathCard-Cons-other*[*simp*]: $x' \neq x$ $\Longrightarrow$ *pathCard* (*x#xs*) x' = *pathCard xs* x'
unfolding *pathCard-def* **by** *auto*

lemma *no-call-in-path-filter*[*simp*]: *no-call-in-path x* [*x*←*xs* . $x \in S$] $\longleftrightarrow$ *no-call-in-path x xs* $\vee$ $x \notin S$
by (*induction xs*) *auto*

lemma *one-call-in-path-filter*[*simp*]: *one-call-in-path x* [*x*←*xs* . $x \in S$] $\longleftrightarrow$ *one-call-in-path x xs* $\vee$ $x \notin S$
by (*induction xs*) *auto*

definition *pathsCard* :: *var list set* $\Rightarrow$ (*var* $\Rightarrow$ *two*)
where *pathsCard ps x* = (*if* ($\forall$ $p \in ps$. *no-call-in-path x p*) *then none else* (*if* ($\forall$ $p{\in}ps$. *one-call-in-path x p*) *then once else many*))

lemma *paths-Card-above*:
$p \in ps$ $\Longrightarrow$ *pathCard p* $\sqsubseteq$ *pathsCard ps*
by (*rule fun-belowI*) (*auto simp add*: *pathsCard-def pathCard-def*)

lemma *pathsCard-below*:
assumes $\bigwedge$ *p*. $p \in ps$ $\Longrightarrow$ *pathCard p* $\sqsubseteq$ *ce*
shows *pathsCard ps* $\sqsubseteq$ *ce*
proof(*rule fun-belowI*)
fix *x*
show *pathsCard ps x* $\sqsubseteq$ *ce x*
by (*auto simp add*: *pathsCard-def pathCard-def split*: *if-splits dest*!: *fun-belowD*[*OF assms*, **where** *x* = *x*] *elim*: *below-trans*[*rotated*] *dest*: *no-imp-one*)
qed

lemma *pathsCard-mono*:
$ps \subseteq ps'$ $\Longrightarrow$ *pathsCard ps* $\sqsubseteq$ *pathsCard* ps'
by (*auto intro*: *pathsCard-below paths-Card-above*)

lemmas *pathsCard-mono'* = *pathsCard-mono*[*folded below-set-def*]

lemma *record-call-pathsCard*:
  pathsCard ({ *tl p* | *p* . *p* ∈ *fs* ∧ *hd p* = *x*}) ⊑ *record-call x*·(*pathsCard fs*)
proof (*rule pathsCard-below*)
  fix *p'*
  assume *p'* ∈ {*tl p* |*p*. *p* ∈ *fs* ∧ *hd p* = *x*}
  then obtain *p* **where** *p'* = *tl p* **and** *p* ∈ *fs* **and** *hd p* = *x* **by** *auto*

  have *pathCard* (*tl p*) ⊑ *record-call x*·(*pathCard p*)
    apply (*rule fun-belowI*)
      using ‹*hd p* = *x*› **by** (*auto simp add*: *pathCard-def record-call-simp no-call-in-tail dest*: *one-imp-one-tail*)

  hence *pathCard* (*tl p*) ⊑ *record-call x*·(*pathsCard fs*)
    by (*rule below-trans*[*OF - monofun-cfun-arg*[*OF paths-Card-above*[*OF* ‹*p* ∈ *fs*›]]])
  thus *pathCard p'* ⊑ *record-call x*·(*pathsCard fs*) **using** ‹*p'* = -› **by** *simp*
qed

lemma *pathCards-noneD*:
  pathsCard ps x = *none* ⟹ *x* ∉ ⋃(*set* ‘ *ps*)
  by (*auto simp add*: *pathsCard-def no-call-in-path-set-conv split*:*if-splits*)

lemma *cont-pathsCard*[*THEN cont-compose*, *cont2cont*, *simp*]:
  cont pathsCard
 by(*fastforce intro*!: *cont2cont-lambda cont-if-else-above simp add*: *pathsCard-def below-set-def*)

lemma *pathsCard-eqvt*[*eqvt*]: π • *pathsCard ps x* = *pathsCard* (π • *ps*) (π • *x*)
  unfolding *pathsCard-def* **by** *perm-simp rule*

lemma *edom-pathsCard*[*simp*]: *edom* (*pathsCard ps*) = ⋃(*set* ‘ *ps*)
  unfolding *edom-def pathsCard-def*
  by (*auto simp add*: *no-call-in-path-set-conv*)

lemma *env-restr-pathsCard*[*simp*]: *pathsCard ps f*|‘ *S* = *pathsCard* (*filter* (λ *x*. *x* ∈ *S*) ‘ *ps*)
  by (*auto simp add*: *pathsCard-def lookup-env-restr-eq*)

end

91 TTreeAnalysisSpec.tex

theory *TTreeAnalysisSpec*
imports *TTreeAnalysisSig ArityAnalysisSpec Cardinality−Domain−Lists*
begin

hide-const *Multiset.single*

locale *TTreeAnalysisCarrier* = *TTreeAnalysis* + *EdomArityAnalysis* +
  assumes *carrier-Fexp*: *carrier* (*Texp e·a*) = *edom* (*Aexp e·a*)

locale *TTreeAnalysisSafe* = *TTreeAnalysisCarrier* +
  assumes *Texp-App*: *many-calls x* $\otimes\otimes$ (*Texp e*)·(*inc·a*) $\sqsubseteq$ *Texp* (*App e x*)·*a*
  assumes *Texp-Lam*: *without y* (*Texp e*·(*pred·n*)) $\sqsubseteq$ *Texp* (*Lam* [*y*]. *e*) · *n*
  assumes *Texp-subst*: *Texp* (*e*[*y*::=*x*])·*a* $\sqsubseteq$ *many-calls x* $\otimes\otimes$ *without y* ((*Texp e*)·*a*)
  assumes *Texp-Var*: *single v* $\sqsubseteq$ *Texp* (*Var v*)·*a*
  assumes *Fun-repeatable*: *isVal e* $\Longrightarrow$ *repeatable* (*Texp e·0*)
  assumes *Texp-IfThenElse*: *Texp scrut·0* $\otimes\otimes$ (*Texp e1·a* $\oplus\oplus$ *Texp e2·a*) $\sqsubseteq$ *Texp* (*scrut ? e1* : *e2*)·*a*

locale *TTreeAnalysisCardinalityHeap* =
  TTreeAnalysisSafe + *ArityAnalysisLetSafe* +
  fixes *Theap* :: *heap* $\Rightarrow$ *exp* $\Rightarrow$ *Arity* $\rightarrow$ *var ttree*
  assumes *carrier-Fheap*: *carrier* (*Theap* Γ *e·a*) = *edom* (*Aheap* Γ *e·a*)
  assumes *Theap-thunk*: *x* $\in$ *thunks* Γ $\Longrightarrow$ *p* $\in$ *paths* (*Theap* Γ *e·a*) $\Longrightarrow$ $\neg$ *one-call-in-path x p* $\Longrightarrow$ (*Aheap* Γ *e·a*) *x* = *up·0*
  assumes *Theap-substitute*: *ttree-restr* (*domA* Δ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e·a*)) (*thunks* Δ) (*Texp e·a*)) $\sqsubseteq$ *Theap* Δ *e·a*
  assumes *Texp-Let*: *ttree-restr* (− *domA* Δ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e·a*)) (*thunks* Δ) (*Texp e·a*)) $\sqsubseteq$ *Texp* (*Terms.Let* Δ *e*)·*a*

end

92 CoCallImplTTreeSafe.tex

theory *CoCallImplTTreeSafe*
imports *CoCallImplTTree CoCallAnalysisSpec TTreeAnalysisSpec*
begin

hide-const *Multiset.single*

lemma *valid-lists-many-calls*:
  assumes $\neg$ *one-call-in-path x p*
  assumes *p* $\in$ *valid-lists S G*
  shows *x*−−*x* $\in$ *G*
using *assms*(*2*,*1*)
proof(*induction rule*:*valid-lists.induct*[*case-names Nil Cons*])
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons xs x'*)
  show *?case*
  proof(*cases one-call-in-path x xs*)
    case *False*
    from *Cons.IH*[*OF this*]

show *?thesis*.
next
case *True*
with ‹¬ *one-call-in-path* x (x' # *xs*)›
have [*simp*]: $x' = x$ **by** (*auto split*: *if-splits*)

have $x \in$ *set xs*
proof(*rule ccontr*)
assume $x \notin$ *set xs*
hence *no-call-in-path x xs* **by** (*metis no-call-in-path-set-conv*)
hence *one-call-in-path* x (x # *xs*) **by** *simp*
with *Cons* **show** *False* **by** *simp*
qed
with ‹*set xs* $\subseteq$ *ccNeighbors* x' *G*›
have $x \in$ *ccNeighbors x G* **by** *auto*
thus *?thesis* **by** *simp*
qed
qed

context *CoCallArityEdom*
begin
lemma *carrier-Fexp'*: *carrier* (*Texp* $e \cdot a$) $\subseteq$ *fv e*
unfolding *Texp-simp carrier-ccTTree*
by (*rule Aexp-edom*)

end

context *CoCallAritySafe*
begin

lemma *carrier-AnalBinds-below*:
carrier ((*Texp.AnalBinds* $\Delta \cdot$(*Aheap* Δ $e \cdot a$)) x) $\subseteq$ *edom* ((*ABinds* Δ)$\cdot$(*Aheap* Δ $e \cdot a$))
by (*auto simp add*: *Texp.AnalBinds-lookup Texp-def split*: *option.splits*
elim!: *set-mp*[*OF edom-mono*[*OF monofun-cfun-fun*[*OF ABind-below-ABinds*]]])

sublocale *TTreeAnalysisCarrier Texp*
apply *standard*
unfolding *Texp-simp carrier-ccTTree*
apply *standard*
done

sublocale *TTreeAnalysisSafe Texp*
proof
fix *x e a*

from *edom-mono*[*OF Aexp-App*]
have $\{x\} \cup$ *edom* (*Aexp* $e \cdot$(*inc*$\cdot a$)) $\subseteq$ *edom* (*Aexp* (*App e x*)$\cdot a$) **by** *auto*
moreover

{
have *ccApprox* (*many-calls* x ⊗⊗ *ccTTree* (*edom* (*Aexp* e·(*inc*·a))) (*ccExp* e·(*inc*·a)))
= *cc-restr* (*edom* (*Aexp* e·(*inc*·a))) (*ccExp* e·(*inc*·a)) ⊔ *ccProd* {x} (*insert* x (*edom* (*Aexp* e·(*inc*·a))))
by (*simp add*: *ccApprox-both ccProd-insert2*[**where** S' = *edom* e **for** e])
also
have *edom* (*Aexp* e·(*inc*·a)) ⊆ *fv* e
by (*rule Aexp-edom*)
also(*below-trans*[*OF eq-imp-below join-mono*[*OF below-refl ccProd-mono2*[*OF insert-mono*]]])
have *cc-restr* (*edom* (*Aexp* e·(*inc*·a))) (*ccExp* e·(*inc*·a)) ⊑ *ccExp* e·(*inc*·a)
by (*rule cc-restr-below-arg*)
also
have *ccExp* e·(*inc*·a) ⊔ *ccProd* {x} (*insert* x (*fv* e)) ⊑ *ccExp* (*App* e x)·a
by (*rule ccExp-App*)
finally
have *ccApprox* (*many-calls* x ⊗⊗ *ccTTree* (*edom* (*Aexp* e·(*inc*·a))) (*ccExp* e·(*inc*·a))) ⊑ *ccExp* (*App* e x)·a **by** *this simp-all*
}
ultimately
show *many-calls* x ⊗⊗ *Texp* e·(*inc*·a) ⊑ *Texp* (*App* e x)·a
unfolding *Texp-simp* **by** (*auto intro*!: *below-ccTTreeI*)
next
fix y e n
show *without* y (*Texp* e·(*pred*·n)) ⊑ *Texp* (*Lam* [y]. e)·n
unfolding *Texp-simp*
by (*auto dest*: *set-mp*[*OF Aexp-edom*]
intro!: *below-ccTTreeI below-trans*[*OF - ccExp-Lam*] *cc-restr-mono1 set-mp*[*OF edom-mono*[*OF Aexp-Lam*]])
next
fix e y x a

from *edom-mono*[*OF Aexp-subst*]
have ∗: *edom* (*Aexp* e[y::=x]·a) ⊆ *insert* x (*edom* (*Aexp* e·a) − {y}) **by** *simp*

have *Texp* e[y::=x]·a = *ccTTree* (*edom* (*Aexp* e[y::=x]·a)) (*ccExp* e[y::=x]·a)
unfolding *Texp-simp*..
also have ... ⊑ *ccTTree* (*insert* x (*edom* (*Aexp* e·a) − {y})) (*ccExp* e[y::=x]·a)
by (*rule ccTTree-mono1*[*OF* ∗])
also have ... ⊑ *many-calls* x ⊗⊗ *without* x (...)
by (*rule paths-many-calls-subset*)
also have *without* x (*ccTTree* (*insert* x (*edom* (*Aexp* e·a) − {y})) (*ccExp* e[y::=x]·a))
= *ccTTree* (*edom* (*Aexp* e·a) − {y} − {x}) (*ccExp* e[y::=x]·a)
by *simp*
also have ... ⊑ *ccTTree* (*edom* (*Aexp* e·a) − {y} − {x}) (*ccExp* e·a)
by (*rule ccTTree-cong-below*[*OF ccExp-subst*]) *auto*
also have ... = *without* y (*ccTTree* (*edom* (*Aexp* e·a) − {x}) (*ccExp* e·a))
by *simp* (*metis Diff-insert Diff-insert2*)
also have *ccTTree* (*edom* (*Aexp* e·a) − {x}) (*ccExp* e·a) ⊑ *ccTTree* (*edom* (*Aexp* e·a)) (*ccExp*

e·a)
by (rule ccTTree-mono1) auto
also have ... = Texp e·a
unfolding Texp-simp**..**
finally
show Texp e[y::=x]·a ⊑ many-calls x ⊗⊗ without y (Texp e·a)
by this simp-all
next
fix v a
have up·a ⊑ (Aexp (Var v)·a) v **by** (rule Aexp-Var)
hence v ∈ edom (Aexp (Var v)·a) **by** (auto simp add: edom-def)
thus single v ⊑ Texp (Var v)·a
unfolding Texp-simp
by (auto intro: below-ccTTreeI)
next
fix scrut e1 a e2
have ccTTree (edom (Aexp e1·a)) (ccExp e1·a) ⊕⊕ ccTTree (edom (Aexp e2·a)) (ccExp e2·a)
⊑ ccTTree (edom (Aexp e1·a) ∪ edom (Aexp e2·a)) (ccExp e1·a ⊔ ccExp e2·a)
by (rule either-ccTTree)
note both-mono2'[OF this]
also
have ccTTree (edom (Aexp scrut·0)) (ccExp scrut·0) ⊗⊗ ccTTree (edom (Aexp e1·a) ∪ edom (Aexp e2·a)) (ccExp e1·a ⊔ ccExp e2·a)
⊑ ccTTree (edom (Aexp scrut·0) ∪ (edom (Aexp e1·a) ∪ edom (Aexp e2·a))) (ccExp scrut·0 ⊔ (ccExp e1·a ⊔ ccExp e2·a) ⊔ ccProd (edom (Aexp scrut·0)) (edom (Aexp e1·a) ∪ edom (Aexp e2·a)))
by (rule interleave-ccTTree)
also
have edom (Aexp scrut·0) ∪ (edom (Aexp e1·a) ∪ edom (Aexp e2·a)) = edom (Aexp scrut·0 ⊔ Aexp e1·a ⊔ Aexp e2·a) **by** auto
also
have Aexp scrut·0 ⊔ Aexp e1·a ⊔ Aexp e2·a ⊑ Aexp (scrut ? e1 : e2)·a
by (rule Aexp-IfThenElse)
also
have ccExp scrut·0 ⊔ (ccExp e1·a ⊔ ccExp e2·a) ⊔ ccProd (edom (Aexp scrut·0)) (edom (Aexp e1·a) ∪ edom (Aexp e2·a)) ⊑
ccExp (scrut ? e1 : e2)·a
by (rule ccExp-IfThenElse)

show Texp scrut·0 ⊗⊗ (Texp e1·a ⊕⊕ Texp e2·a) ⊑ Texp (scrut ? e1 : e2)·a
unfolding Texp-simp
by (auto simp add: ccApprox-both join-below-iff below-trans[OF - join-above2]
intro!: below-ccTTreeI below-trans[OF cc-restr-below-arg]
below-trans[OF - ccExp-IfThenElse] set-mp[OF edom-mono[OF Aexp-IfThenElse]])
next
fix e
assume isVal e
hence [simp]: ccExp e·0 = ccSquare (fv e) **by** (rule ccExp-pap)

thus *repeatable* (*Texp e·0*)
  unfolding *Texp-simp* **by** (*auto intro*: *repeatable-ccTTree-ccSquare*[*OF Aexp-edom*])
qed

definition *Theap* :: *heap* ⇒ *exp* ⇒ *Arity* → *var ttree*
  where *Theap* Γ *e* = (Λ *a*. *if nonrec* Γ *then ccTTree* (*edom* (*Aheap* Γ *e·a*)) (*ccExp e·a*) *else ttree-restr* (*edom* (*Aheap* Γ *e·a*)) *anything*)

lemma *Theap-simp*: *Theap* Γ *e·a* = (*if nonrec* Γ *then ccTTree* (*edom* (*Aheap* Γ *e·a*)) (*ccExp e·a*) *else ttree-restr* (*edom* (*Aheap* Γ *e·a*)) *anything*)
  unfolding *Theap-def* **by** *simp*

lemma *carrier-Fheap′*:*carrier* (*Theap* Γ *e·a*) = *edom* (*Aheap* Γ *e·a*)
  unfolding *Theap-simp carrier-ccTTree* **by** *simp*

sublocale *TTreeAnalysisCardinalityHeap Texp Aexp Aheap Theap*
proof
  fix Γ *e a*
  show *carrier* (*Theap* Γ *e·a*) = *edom* (*Aheap* Γ *e·a*)
    by (*rule carrier-Fheap′*)
next
  fix *x* Γ *p e a*
  assume *x* ∈ *thunks* Γ

  assume ¬ *one-call-in-path x p*
  hence *x* ∈ *set p* **by** (*rule more-than-one-setD*)

  assume *p* ∈ *paths* (*Theap* Γ *e·a*) **with** ‹*x* ∈ *set p*›
  have *x* ∈ *carrier* (*Theap* Γ *e·a*) **by** (*auto simp add*: *Union-paths-carrier*[*symmetric*])
  hence *x* ∈ *edom* (*Aheap* Γ *e·a*)
    unfolding *Theap-simp* **by** (*auto split*: *if-splits*)

  show (*Aheap* Γ *e·a*) *x* = *up·0*
  proof(*cases nonrec* Γ)
    case *False*
    from *False* ‹*x* ∈ *thunks* Γ› ‹*x* ∈ *edom* (*Aheap* Γ *e·a*)›
    show *?thesis* **by** (*rule aHeap-thunks-rec*)
  next
    case *True*
    with ‹*p* ∈ *paths* (*Theap* Γ *e·a*)›
    have *p* ∈ *valid-lists* (*edom* (*Aheap* Γ *e·a*)) (*ccExp e·a*) **by** (*simp add*: *Theap-simp*)

    with ‹¬ *one-call-in-path x p*›
    have *x*−−*x*∈ (*ccExp e·a*) **by** (*rule valid-lists-many-calls*)

    from *True* ‹*x* ∈ *thunks* Γ› *this*
    show *?thesis* **by** (*rule aHeap-thunks-nonrec*)
  qed
next

fix Δ *e a*

have *carrier*: *carrier* (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)) $\subseteq$ *edom* (*Aheap* Δ *e*·*a*) $\cup$ *edom* (*Aexp* (*Let* Δ *e*)·*a*)
proof(*rule carrier-substitute-below*)
from *edom-mono*[*OF Aexp-Let*[*of* Δ *e a*]]
show *carrier* (*Texp e*·*a*) $\subseteq$ *edom* (*Aheap* Δ *e*·*a*) $\cup$ *edom* (*Aexp* (*Let* Δ *e*)·*a*) **by** (*simp add*: *Texp-def*)
next
fix *x*
assume $x \in$ *edom* (*Aheap* Δ *e*·*a*) $\cup$ *edom* (*Aexp* (*Let* Δ *e*)·*a*)
hence $x \in$ *edom* (*Aheap* Δ *e*·*a*) $\vee$ *x* : (*edom* (*Aexp* (*Let* Δ *e*)·*a*)) **by** *simp*
thus *carrier* ((*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) *x*) $\subseteq$ *edom* (*Aheap* Δ *e*·*a*) $\cup$ *edom* (*Aexp* (*Let* Δ *e*)·*a*)
proof
assume $x \in$ *edom* (*Aheap* Δ *e*·*a*)

have *carrier* ((*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) *x*) $\subseteq$ *edom* (*ABinds* Δ·(*Aheap* Δ *e*·*a*))
by (*rule carrier-AnalBinds-below*)
also have ... $\subseteq$ *edom* (*Aheap* Δ *e*·*a* $\sqcup$ *Aexp* (*Terms.Let* Δ *e*)·*a*)
using *edom-mono*[*OF Aexp-Let*[*of* Δ *e a*]] **by** *simp*
finally show *?thesis* **by** *simp*
next
assume $x \in$ *edom* (*Aexp* (*Terms.Let* Δ *e*)·*a*)
hence $x \notin$ *domA* Δ **by** (*auto dest*: *set-mp*[*OF Aexp-edom*])
hence (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) *x* = $\bot$
by (*rule Texp.AnalBinds-not-there*)
thus *?thesis* **by** *simp*
qed
qed

show *ttree-restr* (− *domA* Δ) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)) $\sqsubseteq$ *Texp* (*Let* Δ *e*)·*a*
proof (*rule below-trans*[*OF - eq-imp-below*[*OF Texp-simp*[*symmetric*]]], *rule below-ccTTreeI*)
have *carrier* (*ttree-restr* (− *domA* Δ) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)))
= *carrier* (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)) − *domA* Δ **by** *auto*
also note *carrier*
also have *edom* (*Aheap* Δ *e*·*a*) $\cup$ *edom* (*Aexp* (*Terms.Let* Δ *e*)·*a*) − *domA* Δ = *edom* (*Aexp* (*Let* Δ *e*)·*a*)
by (*auto dest*: *set-mp*[*OF edom-Aheap*] *set-mp*[*OF Aexp-edom*])
finally
show *carrier* (*ttree-restr* (− *domA* Δ) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ)(*Texp e*·*a*)))
$\subseteq$ *edom* (*Aexp* (*Terms.Let* Δ *e*)·*a*) **by** *this auto*
next
let *?x* = *ccApprox* (*ttree-restr* (− *domA* Δ) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)))

have *?x* = *cc-restr* (− *domA* Δ) *?x* **by** *simp*
also have ... ⊑ *cc-restr* (− *domA* Δ) (*ccHeap* Δ *e*·*a*)
proof(*rule cc-restr-mono2*[*OF wild-recursion-thunked*])
  have *ccExp e*·*a* ⊑ *ccHeap* Δ *e*·*a* **by** (*rule ccHeap-Exp*)
  thus *ccApprox* (*Texp e*·*a*) ⊑ *ccHeap* Δ *e*·*a*
    by (*auto simp add*: *Texp-simp intro*: *below-trans*[*OF cc-restr-below-arg*])
next
  fix *x*
  assume *x* ∉ *domA* Δ
  thus (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) *x* = *empty*
    by (*metis Texp.AnalBinds-not-there empty-is-bottom*)
next
  fix *x*
  assume *x* ∈ *domA* Δ
  then obtain *e′* **where** *e′*: *map-of* Δ *x* = *Some e′* **by** (*metis domA-map-of-Some-the*)

  show *ccApprox* ((*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) *x*) ⊑ *ccHeap* Δ *e*·*a*
  proof(*cases* (*Aheap* Δ *e*·*a*) *x*)
    case *bottom* **thus** *?thesis* **using** *e′* **by** (*simp add*: *Texp.AnalBinds-lookup*)
  next
    case (*up a′*)
    with *e′*
    have *ccExp e′*·*a′* ⊑ *ccHeap* Δ *e*·*a* **by** (*rule ccHeap-Heap*)
    thus *?thesis* **using** *up e′*
   by (*auto simp add*: *Texp.AnalBinds-lookup Texp-simp intro*: *below-trans*[*OF cc-restr-below-arg*])
  qed

  show *ccProd* (*ccNeighbors x* (*ccHeap* Δ *e*·*a*)− {*x*} ∩ *thunks* Δ) (*carrier* ((*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) *x*)) ⊑ *ccHeap* Δ *e*·*a*
  proof(*cases* (*Aheap* Δ *e*·*a*) *x*)
    case *bottom* **thus** *?thesis* **using** *e′* **by** (*simp add*: *Texp.AnalBinds-lookup*)
  next
    case (*up a′*)
    have *subset*: (*carrier* (*fup*·(*Texp e′*)·((*Aheap* Δ *e*·*a*) *x*))) ⊆ *fv e′*
      using *up e′* **by** (*auto simp add*: *Texp.AnalBinds-lookup carrier-Fexp dest*!: *set-mp*[*OF Aexp-edom*])

    from *e′ up*
   have *ccProd* (*fv e′*) (*ccNeighbors x* (*ccHeap* Δ *e*·*a*) − {*x*} ∩ *thunks* Δ) ⊑ *ccHeap* Δ *e*·*a*
     by (*rule ccHeap-Extra-Edges*)
    then
    show *?thesis* **using** *e′*
        by (*simp add*: *Texp.AnalBinds-lookup Texp-simp ccProd-comm below-trans*[*OF ccProd-mono2*[*OF subset*]])
  qed
qed
also have ... ⊑ *ccExp* (*Let* Δ *e*)·*a*
  by (*rule ccExp-Let*)

finally
show *ccApprox* (*ttree-restr* (− *domA* Δ) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp* *e*·*a*)))
⊑ *ccExp* (*Terms.Let* Δ *e*)·*a* **by** *this simp-all*
qed

note *carrier*
hence *carrier* (*substitute* (*ExpAnalysis.AnalBinds Texp* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp* *e*·*a*)) ⊆ *edom* (*Aheap* Δ *e*·*a*) ∪ − *domA* Δ
by (*rule order-trans*) (*auto dest*: *set-mp*[*OF Aexp-edom*])
hence *ttree-restr* (*domA* Δ) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp* *e*·*a*))
= *ttree-restr* (*edom* (*Aheap* Δ *e*·*a*)) (*ttree-restr* (*domA* Δ) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp* *e*·*a*)))
by −(*rule ttree-restr-noop*[*symmetric*], *auto*)
also
have ... = *ttree-restr* (*edom* (*Aheap* Δ *e*·*a*)) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp* *e*·*a*))
by (*simp add*: *inf.absorb2*[*OF edom-Aheap*])
also
have ... ⊑ *Theap* Δ *e* ·*a*
proof(*cases nonrec* Δ)
case *False*
have *ttree-restr* (*edom* (*Aheap* Δ *e*·*a*)) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp* *e*·*a*))
⊑ *ttree-restr* (*edom* (*Aheap* Δ *e*·*a*)) *anything*
by (*rule ttree-restr-mono*) *simp*
also have ... = *Theap* Δ *e*·*a*
by (*simp add*: *Theap-simp False*)
finally show *?thesis*.
next
case [*simp*]: *True*

from *True*
have *ttree-restr* (*edom* (*Aheap* Δ *e*·*a*)) (*substitute* (*Texp.AnalBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp* *e*·*a*))
= *ttree-restr* (*edom* (*Aheap* Δ *e*·*a*)) (*Texp* *e*·*a*)
by (*rule nonrecE*) (*rule ttree-rest-substitute*, *auto simp add*: *carrier-Fexp fv-def fresh-def dest*!: *set-mp*[*OF edom-Aheap*] *set-mp*[*OF Aexp-edom*])
also have ... = *ccTTree* (*edom* (*Aexp* *e*·*a*) ∩ *edom* (*Aheap* Δ *e*·*a*)) (*ccExp* *e*·*a*)
by (*simp add*: *Texp-simp*)
also have ... ⊑ *ccTTree* (*edom* (*Aexp* *e*·*a*) ∩ *domA* Δ) (*ccExp* *e*·*a*)
by (*rule ccTTree-mono1*[*OF Int-mono*[*OF order-refl edom-Aheap*]])
also have ... ⊑ *ccTTree* (*edom* (*Aheap* Δ *e*·*a*)) (*ccExp* *e*·*a*)
by (*rule ccTTree-mono1*[*OF edom-mono*[*OF Aheap-nonrec*[*OF True*], *simplified*]])
also have ... ⊑ *Theap* Δ *e*·*a*
by (*simp add*: *Theap-simp*)
finally
show *?thesis* **by** *this simp-all*

qed
finally
show *ttree-restr* (*domA* Δ) (*substitute* (*ExpAnalysis.AnalBinds Texp* $\Delta\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ) (*Texp e*$\cdot$*a*)) $\sqsubseteq$ *Theap* Δ *e*$\cdot$*a*.

qed
end

lemma *paths-singles*: *xs* $\in$ *paths* (*singles S*) $\longleftrightarrow$ ($\forall$ *x* $\in$ *S*. *one-call-in-path x xs*)
by *transfer* (*auto simp add*: *one-call-in-path-filter-conv*)

lemma *paths-singles'*: *xs* $\in$ *paths* (*singles S*) $\longleftrightarrow$ ($\forall$ *x* $\in$ (*set xs* $\cap$ *S*). *one-call-in-path x xs*)
apply *transfer*
apply (*auto simp add*: *one-call-in-path-filter-conv*)
apply (*erule-tac x* = *x* **in** *ballE*)
apply *auto*
by (*metis* (*poly-guards-query*) *filter-empty-conv le0 length-0-conv*)

lemma *both-below-singles1*:
assumes *t* $\sqsubseteq$ *singles S*
assumes *carrier t'* $\cap$ *S* = {}
shows *t* $\otimes\otimes$ *t'* $\sqsubseteq$ *singles S*
proof (*rule ttree-belowI*)
fix *xs*
assume *xs* $\in$ *paths* (*t* $\otimes\otimes$ *t'*)
then obtain *ys zs* **where** *ys* $\in$ *paths t* **and** *zs* $\in$ *paths t'* **and** *xs* $\in$ *ys* $\otimes$ *zs* **by** (*auto simp add*: *paths-both*)
with *assms*
have *ys* $\in$ *paths* (*singles S*) **and** *set zs* $\cap$ *S* = {}
by (*metis below-ttree.rep-eq contra-subsetD paths.rep-eq*, *auto simp add*: *Union-paths-carrier*[*symmetric*])
with ‹*xs* $\in$ *ys* $\otimes$ *zs*›
show *xs* $\in$ *paths* (*singles S*)
by (*induction*) (*auto simp add*: *paths-singles no-call-in-path-set-conv interleave-set dest*: *more-than-one-setD split*: *if-splits*)
qed

lemma *paths-ttree-restr-singles*: *xs* $\in$ *paths* (*ttree-restr S'* (*singles S*)) $\longleftrightarrow$ *set xs* $\subseteq$ *S'* $\wedge$ ($\forall$ *x* $\in$ *S*. *one-call-in-path x xs*)
proof
show *xs* $\in$ *paths* (*ttree-restr S'* (*singles S*)) $\Longrightarrow$ *set xs* $\subseteq$ *S'* $\wedge$ ($\forall$ *x* $\in$ *S*. *one-call-in-path x xs*)
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*] *paths-singles*)
next
assume $*$: *set xs* $\subseteq$ *S'* $\wedge$ ($\forall$ *x*$\in$*S*. *one-call-in-path x xs*)
hence *set xs* $\subseteq$ *S'* **by** *auto*

hence [*simp*]: *filter* (λ x'. $x' \in S'$) *xs* = *xs* **by** (*auto simp add*: *filter-id-conv*)

from $*$
have *xs* $\in$ *paths* (*singles* S)
by (*auto simp add*: *paths-singles*$'$)
hence *filter* (λ x'. $x' \in S'$) *xs* $\in$ *filter* ($\lambda x'$. $x' \in S'$) ' *paths* (*singles* S)
by (*rule imageI*)
thus *xs* $\in$ *paths* (*ttree-restr* S' (*singles* S))
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*])
qed

lemma *substitute-not-carrier*:
assumes $x \notin$ *carrier* t
assumes $\bigwedge$ x'. $x \notin$ *carrier* (f x')
shows $x \notin$ *carrier* (*substitute* f T t)
proof$-$
have *ttree-restr* ($\{x\}$) (*substitute* f T t) = *ttree-restr* ($\{x\}$) t
proof(*rule ttree-rest-substitute*)
fix x'
from ‹$x \notin$ *carrier* (f x')›
show *carrier* (f x') $\cap$ $\{x\}$ = {} **by** *auto*
qed
hence $x \notin$ *carrier* (*ttree-restr* ($\{x\}$) (*substitute* f T t)) $\longleftrightarrow$ $x \notin$ *carrier* (*ttree-restr* ($\{x\}$) t)
by *metis*
with *assms*(*1*)
show *?thesis* **by** *simp*
qed

lemma *substitute-below-singlesI*:
assumes $t \sqsubseteq$ *singles* S
assumes $\bigwedge$ x. *carrier* (f x) $\cap$ S = {}
shows *substitute* f T $t \sqsubseteq$ *singles* S
proof(*rule ttree-belowI*)
fix *xs*
assume *xs* $\in$ *paths* (*substitute* f T t)
thus *xs* $\in$ *paths* (*singles* S)
using *assms*
proof(*induction* f T t *xs arbitrary*: S *rule*: *substitute-induct*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons* f T t x *xs*)

from ‹$x\#xs \in$ -›
have *xs*: *xs* $\in$ *paths* (*substitute* (*f-nxt* f T x) T (*nxt* t x $\otimes\otimes$ f x)) **by** *auto*

moreover

from ‹$t \sqsubseteq$ *singles* S›
have *nxt t x* $\sqsubseteq$ *singles* S
  by (*metis TTree–HOLCF.nxt-mono below-trans nxt-singles-below-singles*)
from *this* ‹*carrier* $(f\ x) \cap S = \{\}$›
have *nxt t x* $\otimes\otimes$ *f x* $\sqsubseteq$ *singles* S
  by (*rule both-below-singles1*)
moreover
{ fix x'
  from ‹*carrier* $(f\ x') \cap S = \{\}$›
  have *carrier* (*f-nxt f T x x'*) $\cap S = \{\}$
    by (*auto simp add*: *f-nxt-def*)
}
ultimately
have *IH*: *xs* $\in$ *paths* (*singles* S)
  by (*rule Cons.IH*)

show *?case*
  proof(*cases* $x \in S$)
    case *True*
    with ‹*carrier* $(f\ x) \cap S = \{\}$›
    have $x \notin$ *carrier* $(f\ x)$ **by** *auto*
    moreover
    from ‹$t \sqsubseteq$ *singles* S›
    have *nxt t x* $\sqsubseteq$ *nxt* (*singles* S) x **by** (*rule nxt-mono*)
    hence *carrier* (*nxt t x*) $\subseteq$ *carrier* (*nxt* (*singles* S) x) **by** (*rule carrier-mono*)
    from *set-mp*[*OF this*] *True*
    have $x \notin$ *carrier* (*nxt t x*) **by** *auto*
    ultimately
    have $x \notin$ *carrier* (*nxt t x* $\otimes\otimes$ *f x*) **by** *simp*
    hence $x \notin$ *carrier* (*substitute* (*f-nxt f T x*) T (*nxt t x* $\otimes\otimes$ *f x*))
    proof(*rule substitute-not-carrier*)
      fix x'
      from ‹*carrier* $(f\ x') \cap S = \{\}$› ‹$x \in S$›
      show $x \notin$ *carrier* (*f-nxt f T x x'*) **by** (*auto simp add*: *f-nxt-def*)
    qed
    with *xs*
    have $x \notin$ *set xs* **by** (*auto simp add*: *Union-paths-carrier*[*symmetric*])
    with *IH*
    have *xs* $\in$ *paths* (*without x* (*singles* S)) **by** (*rule paths-withoutI*)
    thus *?thesis* **using** *True* **by** (*simp add*: *Cons-path*)
  next
    case *False*
    with *IH*
    show *?thesis* **by** (*simp add*: *Cons-path*)
  qed
qed
qed

end

93 TTreeImplCardinality.tex

theory *TTreeImplCardinality*
imports *TTreeAnalysisSig CardinalityAnalysisSig Cardinality−Domain−Lists*
begin

hide-const *Multiset.single*

context *TTreeAnalysis*
begin

fun *unstack* :: *stack* ⇒ *exp* ⇒ *exp* **where**
unstack [] *e* = *e*
| *unstack* (*Alts e1 e2* # *S*) *e* = *unstack S e*
| *unstack* (*Upd x* # *S*) *e* = *unstack S e*
| *unstack* (*Arg x* # *S*) *e* = *unstack S* (*App e x*)
| *unstack* (*Dummy x* # *S*) *e* = *unstack S e*

fun *Fstack* :: *Arity list* ⇒ *stack* ⇒ *var ttree*
where *Fstack* - [] = ⊥
| *Fstack* (*a*#*as*) (*Alts e1 e2* # *S*) = (*Texp e1*·*a* ⊕⊕ *Texp e2*·*a*) ⊗⊗ *Fstack as S*
| *Fstack as* (*Arg x* # *S*) = *many-calls x* ⊗⊗ *Fstack as S*
| *Fstack as* (- # *S*) = *Fstack as S*

fun *prognosis* :: *AEnv* ⇒ *Arity list* ⇒ *Arity* ⇒ *conf* ⇒ *var* ⇒ *two*
where *prognosis ae as a* (Γ, *e*, *S*) = *pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp e*·*a* ⊗⊗ *Fstack as S*)))
end

end

94 TTreeImplCardinalitySafe.tex

theory *TTreeImplCardinalitySafe*
imports *TTreeImplCardinality TTreeAnalysisSpec CardinalityAnalysisSpec*
begin

hide-const *Multiset.single*

lemma *pathsCard-paths-nxt*: *pathsCard* (*paths* (*nxt f x*)) $\sqsubseteq$ *record-call x*·(*pathsCard* (*paths f*))
apply *transfer*
apply (*rule pathsCard-below*)
apply *auto*
apply (*erule below-trans*[*OF - monofun-cfun-arg*[*OF paths-Card-above*], *rotated*]) **back**
apply (*auto intro*: *fun-belowI simp add*: *record-call-simp two-pred-two-add-once*)
done

lemma *pathsCards-none*: *pathsCard* (*paths t*) x = *none* $\Longrightarrow$ $x \notin$ *carrier t*
by *transfer* (*auto dest*: *pathCards-noneD*)

lemma *const-on-edom-disj*: *const-on f S empty* $\longleftrightarrow$ *edom f* $\cap$ S = {}
by (*auto simp add*: *empty-is-bottom edom-def*)

context *TTreeAnalysisCarrier*
begin
lemma *carrier-Fstack*: *carrier* (*Fstack as S*) $\subseteq$ *fv S*
by (*induction S rule*: *Fstack.induct*)
(*auto simp add*: *empty-is-bottom*[*symmetric*] *carrier-Fexp dest*!: *set-mp*[*OF Aexp-edom*])

lemma *carrier-FBinds*: *carrier* ((*FBinds* Γ·*ae*) *x*) $\subseteq$ *fv* Γ
apply (*simp add*: *Texp.AnalBinds-lookup*)
apply (*auto split*: *option.split simp add*: *empty-is-bottom*[*symmetric*])
apply (*case-tac ae x*)
apply (*auto simp add*: *empty-is-bottom*[*symmetric*] *carrier-Fexp dest*!: *set-mp*[*OF Aexp-edom*])
by (*metis* (*poly-guards-query*) *contra-subsetD domA-from-set map-of-fv-subset map-of-SomeD option.sel*)
end

context *TTreeAnalysisSafe*
begin

sublocale *CardinalityPrognosisShape prognosis*
proof
fix Γ :: *heap* **and** *ae ae'* :: *AEnv* **and** *u e S as*
assume *ae f*|' *domA* Γ = *ae' f*|' *domA* Γ
from *Texp.AnalBinds-cong*[*OF this*]
show *prognosis ae as u* (Γ, *e*, *S*) = *prognosis ae' as u* (Γ, *e*, *S*) **by** *simp*
next
fix *ae as a* Γ *e S*
show *const-on* (*prognosis ae as a* (Γ, *e*, *S*)) (*ap S*) *many*
proof
fix *x*
assume $x \in$ *ap S*
hence [*x*,*x*] $\in$ *paths* (*Fstack as S*)
by (*induction S rule*: *Fstack.induct*)
(*auto 4 4 intro*: *set-mp*[*OF both-contains-arg1*] *set-mp*[*OF both-contains-arg2*] *paths-Cons-nxt*)
hence [*x*,*x*] $\in$ *paths* (*Texp e*·*a* ⊗⊗ *Fstack as S*)

by (*rule set-mp*[*OF both-contains-arg2*])
hence [*x*,*x*] ∈ *paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp e*·*a* ⊗⊗ *Fstack as S*))
by (*rule set-mp*[*OF substitute-contains-arg*])
hence *pathCard* [*x*,*x*] *x* ⊑ *pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp e*·*a* ⊗⊗ *Fstack as S*))) *x*
by (*metis fun-belowD paths-Card-above*)
also have *pathCard* [*x*,*x*] *x* = *many* **by** (*auto simp add*: *pathCard-def*)
finally
show *prognosis ae as a* (Γ, *e*, *S*) *x* = *many*
by (*auto intro*: *below-antisym*)
qed
next
fix Γ Δ :: *heap* **and** *e* :: *exp* **and** *ae* :: *AEnv* **and** *as u S*
assume *map-of* Γ = *map-of* Δ
hence *FBinds* Γ = *FBinds* Δ **and** *thunks* Γ = *thunks* Δ **by** (*auto intro*!: *cfun-eqI thunks-cong simp add*: *Texp*.*AnalBinds-lookup*)
thus *prognosis ae as u* (Γ, *e*, *S*) = *prognosis ae as u* (Δ, *e*, *S*) **by** *simp*
next
fix Γ :: *heap* **and** *e* :: *exp* **and** *ae* :: *AEnv* **and** *as u S x*
show *prognosis ae as u* (Γ, *e*, *S*) ⊑ *prognosis ae as u* (Γ, *e*, *Upd x* # *S*) **by** *simp*
next
fix Γ :: *heap* **and** *e* :: *exp* **and** *ae* :: *AEnv* **and** *as a S x*
assume *ae x* = ⊥

hence *FBinds* (*delete x* Γ)·*ae* = *FBinds* Γ·*ae* **by** (*rule Texp*.*AnalBinds-delete-bot*)
moreover
hence ((*FBinds* Γ·*ae*) *x*) = ⊥ **by** (*metis Texp*.*AnalBinds-delete-lookup*)
ultimately
show *prognosis ae as a* (Γ, *e*, *S*) ⊑ *prognosis ae as a* (*delete x* Γ, *e*, *S*)
by (*simp add*: *substitute-T-delete empty-is-bottom*)
next
fix *ae as a* Γ *x S*
have *once* ⊑ (*pathCard* [*x*]) *x* **by** (*simp add*: *two-add-simp*)
also have *pathCard* [*x*] ⊑ *pathsCard* ({[],[*x*]})
by (*rule paths-Card-above*) *simp*
also have . . . = *pathsCard* (*paths* (*single x*)) **by** *simp*
also have *single x* ⊑ (*Texp* (*Var x*)·*a*) **by** (*rule Texp-Var*)
also have . . . ⊑ *Texp* (*Var x*)·*a* ⊗⊗ *Fstack as S* **by** (*rule both-above-arg1*)
also have . . . ⊑ *substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp* (*Var x*)·*a* ⊗⊗ *Fstack as S*) **by** (*rule substitute-above-arg*)
also have *pathsCard* (*paths* . . .) *x* = *prognosis ae as a* (Γ, *Var x*, *S*) *x* **by** *simp*
finally
show *once* ⊑ *prognosis ae as a* (Γ, *Var x*, *S*) *x*
by *this* (*rule cont2cont-fun*, *intro cont2cont*)+
qed

sublocale *CardinalityPrognosisApp prognosis*
proof
fix *ae as a* Γ *e x S*

have *Texp e*$\cdot$(*inc*$\cdot$*a*) $\otimes\otimes$ *many-calls x* $\otimes\otimes$ *Fstack as S* = *many-calls x* $\otimes\otimes$ (*Texp e*)$\cdot$(*inc*$\cdot$*a*) $\otimes\otimes$ *Fstack as S*
by (*metis both-assoc both-comm*)
thus *prognosis ae as* (*inc*$\cdot$*a*) (Γ, *e*, *Arg x* # *S*) $\sqsubseteq$ *prognosis ae as a* (Γ, *App e x*, *S*)
by *simp* (*intro pathsCard-mono' paths-mono substitute-mono2' both-mono1' Texp-App*)
qed

sublocale *CardinalityPrognosisLam prognosis*
proof
fix *ae as a* Γ *e y x S*
have *Texp e*[*y*::=*x*]$\cdot$(*pred*$\cdot$*a*) $\sqsubseteq$ *many-calls x* $\otimes\otimes$ *Texp* (*Lam* [*y*]. *e*)$\cdot$*a*
by (*rule below-trans*[*OF Texp-subst both-mono2'*[*OF Texp-Lam*]])
moreover have *Texp* (*Lam* [*y*]. *e*)$\cdot$*a* $\otimes\otimes$ *many-calls x* $\otimes\otimes$ *Fstack as S* = *many-calls x* $\otimes\otimes$ *Texp* (*Lam* [*y*]. *e*)$\cdot$*a* $\otimes\otimes$ *Fstack as S*
by (*metis both-assoc both-comm*)
ultimately
show *prognosis ae as* (*pred*$\cdot$*a*) (Γ, *e*[*y*::=*x*], *S*) $\sqsubseteq$ *prognosis ae as a* (Γ, *Lam* [*y*]. *e*, *Arg x* # *S*)
by *simp* (*intro pathsCard-mono' paths-mono substitute-mono2' both-mono1'*)
qed

sublocale *CardinalityPrognosisVar prognosis*
proof
fix Γ :: *heap* **and** *e* :: *exp* **and** *x* :: *var* **and** *ae* :: *AEnv* **and** *as u a S*
assume *map-of* Γ *x* = *Some e*
assume *ae x* = *up*$\cdot$*u*

assume *isVal e*
hence *x* $\notin$ *thunks* Γ **using** ‹*map-of* Γ *x* = *Some e*› **by** (*metis thunksE*)
hence [*simp*]: *f-nxt* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) *x* = *FBinds* $\Gamma$$\cdot$*ae* **by** (*auto simp add*: *f-nxt-def*)

have *prognosis ae as u* (Γ, *e*, *S*) = *pathsCard* (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) (*Texp e*$\cdot$*u* $\otimes\otimes$ *Fstack as S*)))
by *simp*
also have ... = *pathsCard* (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) (*nxt* (*single x*) *x* $\otimes\otimes$ *Texp e*$\cdot$*u* $\otimes\otimes$ *Fstack as S*)))
by *simp*
also have ... = *pathsCard* (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) ((*nxt* (*single x*) *x* $\otimes\otimes$ *Fstack as S*) $\otimes\otimes$ *Texp e*$\cdot$*u*)))
by (*metis both-assoc both-comm*)
also have ... $\sqsubseteq$ *pathsCard* (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) (*nxt* (*single x* $\otimes\otimes$ *Fstack as S*) *x* $\otimes\otimes$ *Texp e*$\cdot$*u*)))
by (*intro pathsCard-mono' paths-mono substitute-mono2' both-mono1' nxt-both-left*) *simp*
also have ... = *pathsCard* (*paths* (*nxt* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) (*single x* $\otimes\otimes$ *Fstack as S*)) *x*))
using ‹*map-of* Γ *x* = *Some e*› ‹*ae x* = *up*$\cdot$*u*› **by** (*simp add*: *Texp.AnalBinds-lookup*)
also have ... $\sqsubseteq$ *record-call x* $\cdot$(*pathsCard* (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) (*single x* $\otimes\otimes$ *Fstack as S*))))

by (*rule pathsCard-paths-nxt*)
also have ... $\sqsubseteq$ *record-call x* ·(*pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) ((*Texp* (*Var x*)·*a*) ⊗⊗ *Fstack as S*))))
by (*intro monofun-cfun-arg pathsCard-mono′ paths-mono substitute-mono2′ both-mono1′ Texp-Var*)
also have ... = *record-call x* ·(*prognosis ae as a* (Γ, *Var x*, *S*))
by *simp*
finally
show *prognosis ae as u* (Γ, *e*, *S*) $\sqsubseteq$ *record-call x*·(*prognosis ae as a* (Γ, *Var x*, *S*)) **by** *this simp-all*
next
fix Γ :: *heap* **and** *e* :: *exp* **and** *x* :: *var* **and** *ae* :: *AEnv* **and** *as u a S*
assume *map-of* Γ *x* = *Some e*
assume *ae x* = *up*·*u*
assume ¬ *isVal e*
hence *x* ∈ *thunks* Γ **using** ‹*map-of* Γ *x* = *Some e*› **by** (*metis thunksI*)
hence [*simp*]: *f-nxt* (*FBinds* Γ·*ae*) (*thunks* Γ) *x* = *FBinds* (*delete x* Γ)·*ae*
by (*auto simp add*: *f-nxt-def Texp.AnalBinds-delete-to-fun-upd empty-is-bottom*)

have *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) = *pathsCard* (*paths* (*substitute* (*FBinds* (*delete x* Γ)·*ae*) (*thunks* (*delete x* Γ)) (*Texp e*·*u* ⊗⊗ *Fstack as S*)))
by *simp*
also have ... = *pathsCard* (*paths* (*substitute* (*FBinds* (*delete x* Γ)·*ae*) (*thunks* Γ) (*Texp e*·*u* ⊗⊗ *Fstack as S*)))
by (*rule arg-cong*[*OF substitute-cong-T*]) (*auto simp add*: *empty-is-bottom*)
also have ... = *pathsCard* (*paths* (*substitute* (*FBinds* (*delete x* Γ)·*ae*) (*thunks* Γ) (*nxt* (*single x*) *x* ⊗⊗ *Texp e*·*u* ⊗⊗ *Fstack as S*)))
by *simp*
also have ... = *pathsCard* (*paths* (*substitute* (*FBinds* (*delete x* Γ)·*ae*) (*thunks* Γ) ((*nxt* (*single x*) *x* ⊗⊗ *Fstack as S*) ⊗⊗ *Texp e*·*u*)))
by (*metis both-assoc both-comm*)
also have ... $\sqsubseteq$ *pathsCard* (*paths* (*substitute* (*FBinds* (*delete x* Γ)·*ae*) (*thunks* Γ) (*nxt* (*single x* ⊗⊗ *Fstack as S*) *x* ⊗⊗ *Texp e*·*u*)))
by (*intro pathsCard-mono′ paths-mono substitute-mono2′ both-mono1′ nxt-both-left*) *simp*
also have ... = *pathsCard* (*paths* (*nxt* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*single x* ⊗⊗ *Fstack as S*)) *x*))
using ‹*map-of* Γ *x* = *Some e*› ‹*ae x* = *up*·*u*› **by** (*simp add*: *Texp.AnalBinds-lookup*)
also have ... $\sqsubseteq$ *record-call x* ·(*pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*single x* ⊗⊗ *Fstack as S*))))
by (*rule pathsCard-paths-nxt*)
also have ... $\sqsubseteq$ *record-call x* ·(*pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) ((*Texp* (*Var x*)·*a*) ⊗⊗ *Fstack as S*))))
by (*intro monofun-cfun-arg pathsCard-mono′ paths-mono substitute-mono2′ both-mono1′ Texp-Var*)
also have ... = *record-call x* ·(*prognosis ae as a* (Γ, *Var x*, *S*))
by *simp*
finally
show *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) $\sqsubseteq$ *record-call x*·(*prognosis ae as a* (Γ, *Var x*, *S*)) **by** *this simp-all*

next
  fix Γ :: *heap* **and** *e* :: *exp* **and** *ae* :: *AEnv* **and** *x* :: *var* **and** *as S*
  assume *isVal e*
  hence *repeatable* (*Texp e·0*) **by** (*rule Fun-repeatable*)

  assume [*simp*]: *x* ∉ *domA* Γ

  have [*simp*]: *thunks* ((*x*, *e*) # Γ) = *thunks* Γ
    using ‹*isVal e*›
    by (*auto simp add*: *thunks-Cons dest*: *set-mp*[*OF thunks-domA*])

  have *fup*·(*Texp e*)·(*ae x*) ⊑ *Texp e·0* **by** (*metis fup2 monofun-cfun-arg up-zero-top*)
  hence *substitute* ((*FBinds* Γ·*ae*)(*x* := *fup*·(*Texp e*)·(*ae x*))) (*thunks* Γ) (*Texp e·0* ⊗⊗ *Fstack as S*) ⊑ *substitute* ((*FBinds* Γ·*ae*)(*x* := *Texp e·0*)) (*thunks* Γ) (*Texp e·0* ⊗⊗ *Fstack as S*)
    by (*intro substitute-mono1′ fun-upd-mono below-refl monofun-cfun-arg*)
  also have ... = *substitute* (((*FBinds* Γ·*ae*)(*x* := *Texp e·0*))(*x* := *empty*)) (*thunks* Γ) (*Texp e·0* ⊗⊗ *Fstack as S*)
    using ‹*repeatable* (*Texp e·0*)› **by** (*rule substitute-remove-anyways*, *simp*)
  also have ((*FBinds* Γ·*ae*)(*x* := *Texp e·0*))(*x* := *empty*) = *FBinds* Γ·*ae*
    by (*simp add*: *fun-upd-idem Texp.AnalBinds-not-there empty-is-bottom*)
  finally
  show *prognosis ae as 0* ((*x*, *e*) # Γ, *e*, *S*) ⊑ *prognosis ae as 0* (Γ, *e*, *Upd x* # *S*)
    by (*simp*, *intro pathsCard-mono′ paths-mono*)
qed

sublocale *CardinalityPrognosisIfThenElse prognosis*
proof
  fix *ae as* Γ *scrut e1 e2 S a*
  have *Texp scrut·0* ⊗⊗ (*Texp e1·a* ⊕⊕ *Texp e2·a*) ⊑ *Texp* (*scrut ? e1* : *e2*)·*a*
    by (*rule Texp-IfThenElse*)
  hence *substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp scrut·0* ⊗⊗ (*Texp e1·a* ⊕⊕ *Texp e2·a*) ⊗⊗ *Fstack as S*) ⊑ *substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp* (*scrut ? e1* : *e2*)·*a* ⊗⊗ *Fstack as S*)
    by (*rule substitute-mono2′*[*OF both-mono1′*])
  thus *prognosis ae* (*a*#*as*) *0* (Γ, *scrut*, *Alts e1 e2* # *S*) ⊑ *prognosis ae as a* (Γ, *scrut ? e1* : *e2*, *S*)
    by (*simp*, *intro pathsCard-mono′ paths-mono*)
next
  fix *ae as a* Γ *b e1 e2 S*
  have *Texp* (*if b then e1 else e2*)·*a* ⊑ *Texp e1·a* ⊕⊕ *Texp e2·a*
    by (*auto simp add*: *either-above-arg1 either-above-arg2*)
  hence *substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp* (*if b then e1 else e2*)·*a* ⊗⊗ *Fstack as S*) ⊑ *substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp* (*Bool b*)·*0* ⊗⊗ (*Texp e1·a* ⊕⊕ *Texp e2·a*) ⊗⊗ *Fstack as S*)
    by (*rule substitute-mono2′*[*OF both-mono1′*[*OF below-trans*[*OF - both-above-arg2*]]])
  thus *prognosis ae as a* (Γ, *if b then e1 else e2*, *S*) ⊑ *prognosis ae* (*a*#*as*) *0* (Γ, *Bool b*, *Alts e1 e2* # *S*)
    by (*auto intro*!: *pathsCard-mono′ paths-mono*)
qed

```
end

context TTreeAnalysisCardinalityHeap
begin

  definition cHeap where
    cHeap Γ e = (Λ a. pathsCard (paths (Theap Γ e·a)))

  lemma cHeap-simp: (cHeap Γ e)·a = pathsCard (paths (Theap Γ e·a))
    unfolding cHeap-def  by (rule beta-cfun) (intro cont2cont)

  sublocale CardinalityHeap cHeap.

  sublocale CardinalityHeapSafe cHeap Aheap
  proof
    fix x Γ e a
    assume x ∈ thunks Γ
    moreover
    assume many ⊑ (cHeap Γ e·a) x
    hence many ⊑ pathsCard (paths (Theap Γ e ·a)) x unfolding cHeap-def by simp
    hence ∃ p∈ (paths (Theap Γ e·a)). ¬ (one-call-in-path x p) unfolding pathsCard-def
      by (auto split: if-splits)
    ultimately
    show (Aheap Γ e·a) x = up·0
      by (metis Theap-thunk)
  next
    fix Γ e a
    show edom (cHeap Γ e·a) = edom (Aheap Γ e·a)
    by (simp add: cHeap-def Union-paths-carrier carrier-Fheap)
  qed

  sublocale CardinalityPrognosisEdom prognosis
  proof
    fix ae as a Γ e S
    show edom (prognosis ae as a (Γ, e, S)) ⊆ fv Γ ∪ fv e ∪ fv S
      apply (simp add: Union-paths-carrier)
      apply (rule carrier-substitute-below)
     apply (auto simp add: carrier-Fexp dest: set-mp[OF Aexp-edom] set-mp[OF carrier-Fstack]
set-mp[OF ap-fv-subset] set-mp[OF carrier-FBinds])
      done
  qed

  sublocale CardinalityPrognosisLet prognosis cHeap
  proof
    fix Δ Γ :: heap and e :: exp and S :: stack and  ae :: AEnv and a :: Arity and as
    assume atom ‘ domA Δ ♯* Γ
    assume atom ‘ domA Δ ♯* S
    assume edom ae ⊆ domA Γ ∪ upds S
```

have *domA* $\Delta \cap$ *edom ae* $= \{\}$
using *fresh-distinct*[*OF* ‹*atom* ‘ *domA* Δ $\sharp*$ Γ›] *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* Δ $\sharp*$ *S*›]
‹*edom ae* $\subseteq$ *domA* $\Gamma \cup$ *upds S*› *ups-fv-subset*[*of S*]
by *auto*

have *const-on1*: $\bigwedge$ *x*. *const-on* (*FBinds* $\Delta\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*carrier* ((*FBinds* $\Gamma\cdot$*ae*) *x*)) *empty*
unfolding *const-on-edom-disj* **using** *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* Δ $\sharp*$ Γ›]
by (*auto dest*!: *set-mp*[*OF carrier-FBinds*] *set-mp*[*OF Texp.edom-AnalBinds*])
have *const-on2*: *const-on* (*FBinds* $\Delta\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*carrier* (*Fstack as S*)) *empty*
unfolding *const-on-edom-disj* **using** *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* Δ $\sharp*$ *S*›]
by (*auto dest*!: *set-mp*[*OF carrier-FBinds*] *set-mp*[*OF carrier-Fstack*] *set-mp*[*OF Texp.edom-AnalBinds*] *set-mp*[*OF ap-fv-subset*])
have *const-on3*: *const-on* (*FBinds* $\Gamma\cdot$*ae*) ($-$ ($-$ *domA* Δ)) *TTree.empty*
and *const-on4*: *const-on* (*FBinds* $\Delta\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*domA* Γ) *TTree.empty*
unfolding *const-on-edom-disj* **using** *fresh-distinct*[*OF* ‹*atom* ‘ *domA* Δ $\sharp*$ Γ›]
by (*auto dest*!: *set-mp*[*OF Texp.edom-AnalBinds*])

have *disj1*: $\bigwedge$ *x*. *carrier* ((*FBinds* $\Gamma\cdot$*ae*) *x*) $\cap$ *domA* $\Delta = \{\}$
using *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* Δ $\sharp*$ Γ›]
by (*auto dest*: *set-mp*[*OF carrier-FBinds*])
hence *disj1'*: $\bigwedge$ *x*. *carrier* ((*FBinds* $\Gamma\cdot$*ae*) *x*) $\subseteq -$ *domA* Δ **by** *auto*
have *disj2*: $\bigwedge$ *x*. *carrier* (*Fstack as S*) $\cap$ *domA* $\Delta = \{\}$
using *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* Δ $\sharp*$ *S*›] **by** (*auto dest*!: *set-mp*[*OF carrier-Fstack*])
hence *disj2'*: *carrier* (*Fstack as S*) $\subseteq -$ *domA* Δ **by** *auto*

{
fix *x*
have (*FBinds* (Δ @ Γ)$\cdot$(*ae* $\sqcup$ *Aheap* Δ *e*$\cdot$*a*)) *x* = (*FBinds* $\Gamma\cdot$*ae*) *x* $\otimes\otimes$ (*FBinds* $\Delta\cdot$(*Aheap* Δ *e*$\cdot$*a*)) *x*
proof (*cases x* $\in$ *domA* Δ)
case *True*
have *map-of* Γ *x* = *None* **using** *True fresh-distinct*[*OF* ‹*atom* ‘ *domA* Δ $\sharp*$ Γ›] **by** (*metis disjoint-iff-not-equal domA-def map-of-eq-None-iff*)
moreover
have *ae x* $= \bot$ **using** *True* ‹*domA* $\Delta \cap$ *edom ae* $= \{\}$› **by** *auto*
ultimately
show *?thesis* **using** *True*
by (*auto simp add*: *Texp.AnalBinds-lookup empty-is-bottom*[*symmetric*] *cong*: *option.case-cong*)
next
case *False*
have *map-of* Δ *x* = *None* **using** *False* **by** (*metis domA-def map-of-eq-None-iff*)
moreover
have (*Aheap* Δ *e*$\cdot$*a*) *x* $= \bot$ **using** *False* **using** *edom-Aheap* **by** (*metis contra-subsetD edomIff*)

ultimately
show *?thesis* **using** *False*
by (*auto simp add*: *Texp.AnalBinds-lookup empty-is-bottom*[*symmetric*] *cong*: *option.case-cong*)
qed
}
note *FBinds* = *ext*[*OF this*]

{
have *pathsCard* (*paths* (*substitute* (*FBinds* (Δ @ Γ)$\cdot$(*Aheap* Δ *e*$\cdot$*a* $\sqcup$ *ae*)) (*thunks* (Δ @ Γ)) (*Texp e*$\cdot$*a* $\otimes\otimes$ *Fstack as S*)))
= *pathsCard* (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* ($\Delta$ @ Γ)) (*substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* (Δ @ Γ)) (*Texp e*$\cdot$*a* $\otimes\otimes$ *Fstack as S*))))
by (*simp add*: *substitute-substitute*[*OF const-on1*] *FBinds*)
also have *substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* ($\Delta$ @ Γ)) = *substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ)
apply (*rule substitute-cong-T*)
using *const-on3*
by (*auto dest*: *set-mp*[*OF thunks-domA*])
also have *substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* (Δ @ Γ)) = *substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ)
apply (*rule substitute-cong-T*)
using *const-on4*
by (*auto dest*: *set-mp*[*OF thunks-domA*])
also have *substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ) (*Texp e*$\cdot$*a* $\otimes\otimes$ *Fstack as S*) = *substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ) (*Texp e*$\cdot$*a*) $\otimes\otimes$ *Fstack as S*
by (*rule substitute-only-empty-both*[*OF const-on2*])
also note *calculation*
}
note *eq-imp-below*[*OF this*]
also
note *env-restr-split*[**where** *S* = *domA* Δ]
also
have *pathsCard* (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) (*substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ) (*Texp e*$\cdot$*a*) $\otimes\otimes$ *Fstack as S*))) *f*|' *domA* Δ
= *pathsCard* (*paths* (*ttree-restr* (*domA* Δ) (*substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ) (*Texp e*$\cdot$*a*))))
by (*simp add*: *filter-paths-conv-free-restr ttree-restr-both ttree-rest-substitute*[*OF disj1*] *ttree-restr-is-empty*[*OF disj2*])
also
have *ttree-restr* (*domA* Δ) (*substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ) (*Texp e*$\cdot$*a*)) $\sqsubseteq$ *Theap* Δ *e*$\cdot$*a* **by** (*rule Theap-substitute*)
also
have *pathsCard* (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) (*substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ) (*Texp e*$\cdot$*a*) $\otimes\otimes$ *Fstack as S*))) *f*|' (− *domA* Δ) =
pathsCard (*paths* (*substitute* (*FBinds* $\Gamma$$\cdot$*ae*) (*thunks* Γ) (*ttree-restr* (− *domA* Δ) (*substitute* (*FBinds* $\Delta$$\cdot$(*Aheap* Δ *e*$\cdot$*a*)) (*thunks* Δ) (*Texp e*$\cdot$*a*)) $\otimes\otimes$ *Fstack as S*)))
by (*simp add*: *filter-paths-conv-free-restr2 ttree-rest-substitute2*[*OF disj1' const-on3*] *ttree-restr-both ttree-restr-noop*[*OF disj2'*])

also have *ttree-restr* $(-\ domA\ \Delta)$ (*substitute* (*FBinds* $\Delta\cdot$(*Aheap* Δ $e\cdot a$)) (*thunks* Δ) (*Texp* $e\cdot a$)) $\sqsubseteq$ *Texp* (*Terms.Let* Δ e)$\cdot a$ **by** (*rule Texp-Let*)
finally
show *prognosis* (*Aheap* Δ $e\cdot a \sqcup ae$) *as a* (Δ @ Γ, e, S) $\sqsubseteq$ *cHeap* Δ $e\cdot a \sqcup$ *prognosis ae as a* (Γ, *Terms.Let* Δ e, S)
by (*simp add*: *cHeap-def del*: *fun-meet-simp*)
qed

sublocale *CardinalityPrognosisSafe prognosis cHeap Aheap Aexp* **..**
end

end

95 CallArityEnd2EndSafe.tex

theory *CallArityEnd2EndSafe*
imports *CallArityEnd2End CardArityTransformSafe CoCallImplSafe CoCallImplTTreeSafe TTreeImplCardinalitySa*
begin

locale *CallArityEnd2EndSafe*
begin
sublocale *CoCallImplSafe*.
sublocale *CallArityEnd2End*.

abbreviation *transform-syn'* ($\mathcal{T}_{-}$) **where** $\mathcal{T}_a \equiv$ *transform a*

lemma *end2end*:
$c \Rightarrow^* c' \Longrightarrow$
$\neg$ *boring-step* $c' \Longrightarrow$
heap-upds-ok-conf $c \Longrightarrow$
consistent (ae, ce, a, as, r) $c \Longrightarrow$
$\exists\, ae'\ ce'\ a'\ as'\ r'.$ *consistent* (ae', ce', a', as', r') $c' \wedge$ *conf-transform* (ae, ce, a, as, r) $c \Rightarrow_G{}^*$ *conf-transform* (ae', ce', a', as', r') c'
by (*rule card-arity-transform-safe*)

theorem *end2end-closed*:
assumes *closed*: *fv* $e = (\{\} :: var\ set)$
assumes $([], e, []) \Rightarrow^* (\Gamma,v,[])$ **and** *isVal* v
obtains Γ' **and** v'
where $([], \mathcal{T}_0\ e, []) \Rightarrow^* (\Gamma',v',[])$ **and** *isVal* v'
and *card* (*domA* Γ') $\leq$ *card* (*domA* Γ)
proof−
note *assms*(*2*)
moreover
have $\neg$ *boring-step* $(\Gamma,v,[])$ **by** (*simp add*: *boring-step.simps*)
moreover
have *heap-upds-ok-conf* $([], e, [])$ **by** *simp*

moreover
have *consistent* $(\bot,\bot,0,[],[])$ $([], e, [])$ **using** *closed* **by** (*rule closed-consistent*)
ultimately
obtain *ae ce a as r* **where**
$*$: *consistent* (ae, ce, a, as, r) $(\Gamma,v,[])$ **and**
$**$: *conf-transform* $(\bot, \bot, 0, [], [])$ $([],e,[])$ $\Rightarrow_G{}^*$ *conf-transform* (ae, ce, a, as, r) $(\Gamma,v,[])$
by (*metis end2end*)

let $?\Gamma$ = *map-transform Aeta-expand ae* (*map-transform transform ae* (*restrictA* ($-$*set r*) Γ))
let $?v$ = *transform a v*

from $*$ **have** *set r* $\subseteq$ *domA* Γ **by** *auto*

have *conf-transform* $(\bot, \bot, 0, [], [])$ $([],e,[])$ = $([],$*transform 0 e*$,[])$ **by** *simp*
with $**$
have $([],$ *transform 0 e*, $[])$ $\Rightarrow_G{}^*$ $(?\Gamma, ?v,$ *map Dummy* (*rev r*)) **by** *simp*

have *isVal ?v* **using** ‹*isVal v*› **by** *simp*

have *fv* (*transform 0 e*) = ({} :: *var set*) **using** *closed*
by (*auto dest*: *set-mp*[*OF fv-transform*])

note *sestoftUnGC'*[*OF* ‹$([],$ *transform 0 e*, $[])$ $\Rightarrow_G{}^*$ $(?\Gamma, ?v,$ *map Dummy* (*rev r*))› ‹*isVal ?v*› ‹*fv* (*transform 0 e*) = {}›]
then obtain Γ'
where $([],$ *transform 0 e*, $[])$ $\Rightarrow^*$ $(\Gamma', ?v, [])$
and $?\Gamma$ = *restrictA* ($-$ *set r*) Γ'
and *set r* $\subseteq$ *domA* Γ'
by *auto*

have *card* (*domA* Γ) = *card* (*domA* $?\Gamma$ $\cup$ (*set r* $\cap$ *domA* Γ))
by (*rule arg-cong*[**where** *f* = *card*]) *auto*
also have ... = *card* (*domA* $?\Gamma$) + *card* (*set r* $\cap$ *domA* Γ)
by (*rule card-Un-disjoint*) *auto*
also note ‹$?\Gamma$ = *restrictA* ($-$ *set r*) Γ'›
also have *set r* $\cap$ *domA* Γ = *set r* $\cap$ *domA* Γ'
using ‹*set r* $\subseteq$ *domA* Γ› ‹*set r* $\subseteq$ *domA* Γ'› **by** *auto*
also have *card* (*domA* (*restrictA* ($-$ *set r*) Γ')) + *card* (*set r* $\cap$ *domA* Γ') = *card* (*domA* Γ')
by (*subst card-Un-disjoint*[*symmetric*]) (*auto intro*: *arg-cong*[**where** *f* = *card*])
finally
have *card* (*domA* Γ') $\leq$ *card* (*domA* Γ) **by** *simp*
with ‹$([],$ *transform 0 e*, $[])$ $\Rightarrow^*$ $(\Gamma', ?v, [])$› ‹*isVal ?v*›
show *thesis* **using** *that* **by** *blast*
qed

lemma *fresh-var-eqE*[*elim-format*]: *fresh-var e* = $x \Longrightarrow x \notin$ *fv e*
by (*metis fresh-var-not-free*)

lemma *example1*:
  fixes *e* :: *exp*
  fixes *f g x y z* :: *var*
  assumes *Aexp-e*: $\bigwedge a$. *Aexp e*·*a* = *esing x*·(*up*·*a*) ⊔ *esing y*·(*up*·*a*)
  assumes *ccExp-e*: $\bigwedge a$. *CCexp e*·*a* = ⊥
  assumes [*simp*]: *transform 1 e* = *e*
  assumes *isVal e*
  assumes *disj*: *y* ≠ *f y* ≠ *g x* ≠ *y z* ≠ *f z* ≠ *g y* ≠ *x*
  assumes *fresh*: *atom z* ♯ *e*
  shows *transform 1* (*let y be App* (*Var f*) *g in* (*let x be e in* (*Var x*))) =
      let y be (*Lam* [*z*]. *App* (*App* (*Var f*) *g*) *z*) *in* (*let x be* (*Lam* [*z*]. *App e z*) *in* (*Var x*))
proof−
  from *arg-cong*[**where** *f* = *edom*, *OF Aexp-e*]
  have *x* ∈ *fv e* **by** *simp* (*metis Aexp-edom′ insert-subset*)
  hence [*simp*]: ¬ *nonrec* [(*x*,*e*)]
    by (*simp add*: *nonrec-def*)

  from ‹*isVal e*›
  have [*simp*]: *thunks* [(*x*, *e*)] = {}
    by (*simp add*: *thunks-Cons*)

  have [*simp*]: *CCfix* [(*x*, *e*)]·(*esing x*·(*up*·*1*) ⊔ *esing y*·(*up*·*1*), ⊥) = ⊥
    unfolding *CCfix-def*
    apply (*simp add*: *fix-bottom-iff ccBindsExtra-simp*)
    apply (*simp add*: *ccBind-eq disj ccExp-e*)
    done

  have [*simp*]: *Afix* [(*x*, *e*)]·(*esing x*·(*up*·*1*)) = *esing x*·(*up*·*1*) ⊔ *esing y*·(*up*·*1*)
    unfolding *Afix-def*
    apply *simp*
    apply (*rule fix-eqI*)
    apply (*simp add*: *disj Aexp-e*)
    apply (*case-tac z x*)
    apply (*auto simp add*: *disj Aexp-e*)
    done

  have [*simp*]: *Aheap* [(*y*, *App* (*Var f*) *g*)] (*let x be e in Var x*)·*1* = *esing y*·((*Aexp* (*let x be e in Var x*)·*1*) *y*)
    by (*auto simp add*: *Aheap-nonrec-simp ABind-nonrec-eq pure-fresh fresh-at-base disj*)

  have [*simp*]: (*Aexp* (*let x be e in Var x*)·*1*) = *esing y*·(*up*·*1*)
    by (*simp add*: *env-restr-join disj*)

  have [*simp*]: *Aheap* [(*x*, *e*)] (*Var x*)·*1* = *esing x*·(*up*·*1*)
    by (*simp add*: *env-restr-join disj*)

  have [*simp*]: *Aeta-expand 1* (*App* (*Var f*) *g*) = (*Lam* [*z*]. *App* (*App* (*Var f*) *g*) *z*)
    apply (*simp add*: *one-is-inc-zero del*: *exp-assn.eq-iff*)

apply (*subst change-Lam-Variable*[*of z fresh-var* (*App* (*Var f*) *g*)])
apply (*auto simp add*: *fresh-Pair fresh-at-base pure-fresh disj intro*!: *flip-fresh-fresh elim*!: *fresh-var-eqE*)
done

have [*simp*]: *Aeta-expand 1 e* = (*Lam* [*z*]. *App e z*)
apply (*simp add*: *one-is-inc-zero del*: *exp-assn.eq-iff*)
apply (*subst change-Lam-Variable*[*of z fresh-var e*])
apply (*auto simp add*: *fresh-Pair fresh-at-base pure-fresh disj fresh intro*!: *flip-fresh-fresh elim*!: *fresh-var-eqE*)
done

show *?thesis*
by (*simp del*: *Let-eq-iff add*: *map-transform-Cons disj*[*symmetric*])
qed

end
end

96 ArityAnalysisCorrDenotational.tex

theory *ArityAnalysisCorrDenotational*
imports *ArityAnalysisSpec Denotational ArityTransform*
begin

context *ArityAnalysisLetSafe*
begin

inductive *eq* :: *Arity* ⇒ *Value* ⇒ *Value* ⇒ *bool* **where**
eq 0 v v
| ($\bigwedge$ *v*. *eq n* (*v1* ↓*Fn v*) (*v2* ↓*Fn v*)) ⟹ *eq* (*inc*·*n*) *v1 v2*

lemma [*simp*]: *eq 0 v v′* ⟷ *v* = *v′*
by (*auto elim*: *eq.cases intro*: *eq.intros*)

lemma *eq-inc-simp*:
eq (*inc*·*n*) *v1 v2* ⟷ (∀ *v* . *eq n* (*v1* ↓*Fn v*) (*v2* ↓*Fn v*))
by (*auto elim*: *eq.cases intro*: *eq.intros*)

lemma *eq-FnI*:
($\bigwedge$ *v*. *eq* (*pred*·*n*) (*f1*·*v*) (*f2*·*v*)) ⟹ *eq n* (*Fn*·*f1*) (*Fn*·*f2*)
by (*induction n rule*: *Arity-ind*) (*auto intro*: *eq.intros cfun-eqI*)

lemma *eq-refl*[*simp*]: *eq a v v*
by (*induction a arbitrary*: *v rule*: *Arity-ind*) (*auto intro*!: *eq.intros*)

lemma *eq-trans*[*trans*]: *eq a v1 v2* ⟹ *eq a v2 v3* ⟹ *eq a v1 v3*

```
  apply (induction a arbitrary: v1 v2 v3 rule: Arity-ind)
  apply (auto elim!: eq.cases intro!: eq.intros)
  apply blast
  done

lemma eq-Fn: eq a v1 v2 ⟹ eq (pred·a) (v1 ↓Fn v) (v2 ↓Fn v)
apply (induction a  rule: Arity-ind[case-names 0 inc])
apply (auto simp add: eq-inc-simp)
done

lemma eq-inc-same: eq a v1 v2 ⟹ eq (inc·a) v1 v2
by (induction a  arbitrary: v1 v2 rule: Arity-ind[case-names 0 inc])  (auto simp add: eq-inc-simp)

lemma eq-mono: a ⊑ a′ ⟹ eq a′ v1 v2 ⟹ eq a v1 v2
proof (induction a  rule: Arity-ind[case-names 0 inc])
  case 0 thus ?case by auto
next
  case (inc a)
  show eq (inc·a) v1 v2
  proof (cases inc·a = a′)
    case True with inc show ?thesis by simp
  next
    case False with ‹inc·a ⊑ a′› have a ⊑ a′
      by (simp add: inc-def)(transfer, simp)
    from this inc.prems(2)
    have eq a v1 v2 by (rule inc.IH)
    thus ?thesis by (rule eq-inc-same)
  qed
qed

lemma eq-join[simp]: eq (a ⊔ a′) v1 v2 ⟷ eq a v1 v2 ∧ eq a′ v1 v2
  using Arity-total[of a a′]
  apply (auto elim!: eq-mono[OF join-above1] eq-mono[OF join-above2])
  apply (metis join-self-below(2))
  apply (metis join-self-below(1))
  done

lemma eq-adm: cont f ⟹ cont g ⟹ adm (λ x. eq a (f x) (g x))
proof (induction a arbitrary: f g rule: Arity-ind[case-names 0 inc])
  case 0 thus ?case by simp
next
  case inc
  show ?case
  apply (subst eq-inc-simp)
  apply (rule adm-all)
  apply (rule inc)
  apply (intro cont2cont inc(2,3))+
  done
qed
```

inductive *eqϱ* :: *AEnv* ⇒ (*var* ⇒ *Value*) ⇒ (*var* ⇒ *Value*) ⇒ *bool* **where**
eqϱI: (⋀ *x a*. *ae x* = *up*·*a* ⟹ *eq a* (*ϱ1 x*) (*ϱ2 x*)) ⟹ *eqϱ ae ϱ1 ϱ2*

lemma *eqϱE*: *eqϱ ae ϱ1 ϱ2* ⟹ *ae x* = *up*·*a* ⟹ *eq a* (*ϱ1 x*) (*ϱ2 x*)
by (*auto simp add*: *eqϱ.simps*)

lemma *eqϱ-refl*[*simp*]: *eqϱ ae ϱ ϱ*
by (*simp add*: *eqϱ.simps*)

lemma *eq-esing-up*[*simp*]: *eqϱ* (*esing x*·(*up*·*a*)) *ϱ1 ϱ2* ⟷ *eq a* (*ϱ1 x*) (*ϱ2 x*)
by (*auto simp add*: *eqϱ.simps*)

lemma *eqϱ-mono*:
assumes *ae* ⊑ *ae'*
assumes *eqϱ ae' ϱ1 ϱ2*
shows *eqϱ ae ϱ1 ϱ2*
proof (*rule eqϱI*)
fix *x a*
assume *ae x* = *up*·*a*
with ‹*ae* ⊑ *ae'*› **have** *up*·*a* ⊑ *ae' x* **by** (*metis fun-belowD*)
then obtain *a'* **where** *ae' x* = *up*·*a'* **by** (*metis Exh-Up below-antisym minimal*)
with ‹*eqϱ ae' ϱ1 ϱ2*›
have *eq a'* (*ϱ1 x*) (*ϱ2 x*) **by** (*auto simp add*: *eqϱ.simps*)
with ‹*up*·*a* ⊑ *ae' x*› **and** ‹*ae' x* = *up*·*a'*›
show *eq a* (*ϱ1 x*) (*ϱ2 x*) **by** (*metis eq-mono up-below*)
qed

lemma *eqϱ-adm*: *cont f* ⟹ *cont g* ⟹ *adm* (λ *x*. *eqϱ a* (*f x*) (*g x*))
apply (*simp add*: *eqϱ.simps*)
apply (*intro adm-lemmas eq-adm*)
apply (*erule cont2cont-fun*)+
done

lemma *up-join-eq-up*[*simp*]: *up*·(*n*::'*a*::*Finite-Join-cpo*) ⊔ *up*·*n'* = *up*·(*n* ⊔ *n'*)
apply (*rule lub-is-join*)
apply (*auto simp add*: *is-lub-def*)
apply (*case-tac u*)
apply *auto*
done

lemma *eqϱ-join*[*simp*]: *eqϱ* (*ae* ⊔ *ae'*) *ϱ1 ϱ2* ⟷ *eqϱ ae ϱ1 ϱ2* ∧ *eqϱ ae' ϱ1 ϱ2*
apply (*auto elim*!: *eqϱ-mono*[*OF join-above1*] *eqϱ-mono*[*OF join-above2*])
apply (*auto intro*!: *eqϱI*)
apply (*case-tac ae x*, *auto elim*: *eqϱE*)
apply (*case-tac ae' x*, *auto elim*: *eqϱE*)
done

lemma *eqϱ-override*[*simp*]:

eqϱ ae $(\varrho 1 ++_S \varrho 2)$ $(\varrho 1' ++_S \varrho 2')$ $\longleftrightarrow$ *eqϱ ae* $(\varrho 1\ f|`\ (-\ S))$ $(\varrho 1'\ f|`\ (-\ S))$ $\wedge$ *eqϱ ae* $(\varrho 2\ f|`\ S)$ $(\varrho 2'\ f|`\ S)$
by (*auto simp add*: *lookup-env-restr-eq eqϱ.simps lookup-override-on-eq*)

lemma *Aexp-heap-below-Aheap*:
assumes $(Aheap\ \Gamma\ e \cdot a)\ x = up \cdot a'$
assumes *map-of* $\Gamma\ x = Some\ e'$
shows $Aexp\ e' \cdot a' \sqsubseteq Aheap\ \Gamma\ e \cdot a \sqcup Aexp\ (Let\ \Gamma\ e) \cdot a$
proof−
from *assms*(*1*)
have $Aexp\ e' \cdot a' = ABind\ x\ e' \cdot (Aheap\ \Gamma\ e \cdot a)$
by (*simp del*: *join-comm fun-meet-simp*)
also have $\ldots \sqsubseteq ABinds\ \Gamma \cdot (Aheap\ \Gamma\ e \cdot a)$
by (*rule monofun-cfun-fun*[*OF ABind-below-ABinds*[*OF* ‹*map-of* - - = -›]])
also have $\ldots \sqsubseteq ABinds\ \Gamma \cdot (Aheap\ \Gamma\ e \cdot a) \sqcup\ Aexp\ e \cdot a$
by *simp*
also note *Aexp-Let*
finally
show *?thesis* **by** *this simp-all*
qed

lemma *Aexp-body-below-Aheap*:
shows $Aexp\ e \cdot a \sqsubseteq Aheap\ \Gamma\ e \cdot a \sqcup Aexp\ (Let\ \Gamma\ e) \cdot a$
by (*rule below-trans*[*OF join-above2 Aexp-Let*])

lemma *Aexp-correct*: $eq\varrho\ (Aexp\ e \cdot a)\ \varrho 1\ \varrho 2 \Longrightarrow eq\ a\ (\llbracket e \rrbracket_{\varrho 1})\ (\llbracket e \rrbracket_{\varrho 2})$
proof(*induction a e arbitrary*: *ϱ1 ϱ2 rule*: *transform.induct*[*case-names App Lam Var Let Bool IfThenElse*])
case (*Var a x*)
from ‹$eq\varrho\ (Aexp\ (Var\ x) \cdot a)\ \varrho 1\ \varrho 2$›
have $eq\varrho\ (esing\ x \cdot (up \cdot a))\ \varrho 1\ \varrho 2$ **by** (*rule eqϱ-mono*[*OF Aexp-Var-singleton*])
thus *?case* **by** *simp*
next
case (*App a e x*)
from ‹$eq\varrho\ (Aexp\ (App\ e\ x) \cdot a)\ \varrho 1\ \varrho 2$›
have $eq\varrho\ (Aexp\ e \cdot (inc \cdot a) \sqcup esing\ x \cdot (up \cdot 0))\ \varrho 1\ \varrho 2$ **by** (*rule eqϱ-mono*[*OF Aexp-App*])
hence $eq\varrho\ (Aexp\ e \cdot (inc \cdot a))\ \varrho 1\ \varrho 2$ **and** $\varrho 1\ x = \varrho 2\ x$ **by** *simp-all*
from *App*(*1*)[*OF this*(*1*)] *this*(*2*)
show *?case* **by** (*auto elim*: *eq.cases*)
next
case (*Lam a x e*)
from ‹$eq\varrho\ (Aexp\ (Lam\ [x].\ e) \cdot a)\ \varrho 1\ \varrho 2$›
have $eq\varrho\ (env\text{-}delete\ x\ (Aexp\ e \cdot (pred \cdot a)))\ \varrho 1\ \varrho 2$ **by** (*rule eqϱ-mono*[*OF Aexp-Lam*])
hence $\bigwedge v.\ eq\varrho\ (Aexp\ e \cdot (pred \cdot a))\ (\varrho 1 (x := v))\ (\varrho 2 (x := v))$ **by** (*auto intro*!: *eqϱI elim*!: *eqϱE*)
from *Lam*(*1*)[*OF this*]
show *?case* **by** (*auto intro*: *eq-FnI simp del*: *fun-upd-apply*)
next

case (*Bool b*)
show *?case* **by** *simp*
next
case (*IfThenElse a scrut* e_1 e_2)
from ‹$eq\varrho$ (*Aexp* (*scrut* ? e_1 : e_2)$\cdot$*a*) $\varrho 1$ $\varrho 2$›
have $eq\varrho$ (*Aexp scrut*$\cdot 0$ $\sqcup$ *Aexp* $e_1 \cdot a$ $\sqcup$ *Aexp* $e_2 \cdot a$) $\varrho 1$ $\varrho 2$ **by** (*rule* $eq\varrho$*-mono*[*OF Aexp-IfThenElse*])
hence $eq\varrho$ (*Aexp scrut*$\cdot 0$) $\varrho 1$ $\varrho 2$
and $eq\varrho$ (*Aexp* $e_1 \cdot a$) $\varrho 1$ $\varrho 2$
and $eq\varrho$ (*Aexp* $e_2 \cdot a$) $\varrho 1$ $\varrho 2$ **by** *simp-all*
from *IfThenElse*(*1*)[*OF this*(*1*)] *IfThenElse*(*2*)[*OF this*(*2*)] *IfThenElse*(*3*)[*OF this*(*3*)]
show *?case*
by (*cases* $[\![$ *scrut* $]\!]_{\varrho 2}$) *auto*
next
case (*Let a* Γ *e*)

have $eq\varrho$ (*Aheap* Γ $e \cdot a$ $\sqcup$ *Aexp* (*Let* Γ *e*)$\cdot a$) ($\{\!|\Gamma|\!\}\varrho 1$) ($\{\!|\Gamma|\!\}\varrho 2$)
proof(*induction rule*: *parallel-HSem-ind*[*case-names adm bottom step*])
case *adm* **thus** *?case* **by** (*intro* $eq\varrho$*-adm cont2cont*)
next
case *bottom* **show** *?case* **by** *simp*
next
case (*step* $\varrho 1'$ $\varrho 2'$)
show *?case*
proof (*rule* $eq\varrho I$)
fix *x* a'
assume *ass*: (*Aheap* Γ $e \cdot a$ $\sqcup$ *Aexp* (*Let* Γ *e*)$\cdot a$) x = *up*$\cdot a'$
show *eq* a' (($\varrho 1$ $++_{domA\ \Gamma}$ $[\![\ \Gamma\ ]\!]_{\varrho 1'}$) *x*) (($\varrho 2$ $++_{domA\ \Gamma}$ $[\![\ \Gamma\ ]\!]_{\varrho 2'}$) *x*)
proof(*cases* $x \in$ *domA* Γ)
case [*simp*]: *True*
then obtain e' **where** [*simp*]: *map-of* Γ x = *Some* e' **by** (*metis domA-map-of-Some-the*)
have (*Aheap* Γ $e \cdot a$) x = *up*$\cdot a'$ **using** *ass* **by** *simp*
hence *Aexp* $e' \cdot a'$ $\sqsubseteq$ *Aheap* Γ $e \cdot a$ $\sqcup$ *Aexp* (*Let* Γ *e*)$\cdot a$ **using** ‹*map-of* - - = -› **by** (*rule Aexp-heap-below-Aheap*)
hence $eq\varrho$ (*Aexp* $e' \cdot a'$) $\varrho 1'$ $\varrho 2'$ **using** *step*(*1*) **by** (*rule* $eq\varrho$*-mono*)
hence *eq* a' ($[\![\ e'\ ]\!]_{\varrho 1'}$) ($[\![\ e'\ ]\!]_{\varrho 2'}$)
by (*rule Let*(*1*)[*OF map-of-SomeD*[*OF* ‹*map-of* - - = -›]])
thus *?thesis* **by** (*simp add*: *lookupEvalHeap'*)
next
case [*simp*]: *False*
with *edom-Aheap* **have** $x \notin$ *edom* (*Aheap* Γ $e \cdot a$) **by** *blast*
hence (*Aexp* (*Let* Γ *e*)$\cdot a$) x = *up*$\cdot a'$ **using** *ass* **by** (*simp add*: *edomIff*)
with ‹$eq\varrho$ (*Aexp* (*Let* Γ *e*)$\cdot a$) $\varrho 1$ $\varrho 2$›
have *eq* a' ($\varrho 1$ *x*) ($\varrho 2$ *x*) **by** (*auto elim*: $eq\varrho E$)
thus *?thesis* **by** *simp*
qed
qed
qed
hence $eq\varrho$ (*Aexp* $e \cdot a$) ($\{\!|\Gamma|\!\}\varrho 1$) ($\{\!|\Gamma|\!\}\varrho 2$) **by** (*rule* $eq\varrho$*-mono*[*OF Aexp-body-below-Aheap*])
hence *eq a* ($[\![\ e\ ]\!]_{\{\!|\Gamma|\!\}\varrho 1}$) ($[\![\ e\ ]\!]_{\{\!|\Gamma|\!\}\varrho 2}$) **by** (*rule Let*(*2*)[*simplified*])

thus *?case* **by** *simp*
qed

lemma *ESem-ignores-fresh*[*simp*]: $⟦ e ⟧_{\varrho(fresh\text{-}var\ e\ :=\ v)} = ⟦ e ⟧_{\varrho}$
by (*metis ESem-fresh-cong env-restr-fun-upd-other fresh-var-not-free*)

lemma *eq-Aeta-expand*: *eq a* ($⟦$ *Aeta-expand a e* $⟧_{\varrho}$) ($⟦e⟧_{\varrho}$)
apply (*induction a arbitrary*: *e* ϱ *rule*: *Arity-ind*[*case-names 0 inc*])
apply *simp*
apply (*fastforce simp add*: *eq-inc-simp elim*: *eq-trans*)
done

lemma *Arity-transformation-correct*: *eq a* ($⟦ \mathcal{T}_a\ e ⟧_{\varrho}$) ($⟦ e ⟧_{\varrho}$)
proof(*induction a e arbitrary*: ϱ *rule*: *transform.induct*[*case-names App Lam Var Let Bool IfThenElse*])
case *Var*
show *?case* **by** *simp*
next
case (*App a e x*)
from *this*[**where** $\varrho = \varrho$]
show *?case*
by (*auto elim*: *eq.cases*)
next
case (*Lam x e*)
thus *?case*
by (*auto intro*: *eq-FnI*)
next
case (*Bool b*)
show *?case* **by** *simp*
next
case (*IfThenElse a e e_1 e_2*)
thus *?case* **by** (*cases* $⟦ e ⟧_{\varrho}$) *auto*
next
case (*Let a Γ e*)

have *eq a* ($⟦$ *transform a* (*Let Γ e*) $⟧_{\varrho}$) ($⟦$ *transform a e* $⟧_{⦃map\text{-}transform\ Aeta\text{-}expand\ (Aheap\ Γ\ e·a)\ (map\text{-}transform\ tra$
by *simp*
also have *eq a* ... ($⟦ e ⟧_{⦃map\text{-}transform\ Aeta\text{-}expand\ (Aheap\ Γ\ e·a)\ (map\text{-}transform\ transform\ (Aheap\ Γ\ e·a)\ Γ)⦄\varrho}$)
using *Let*(*2*) **by** *simp*
also have *eq a* ... ($⟦ e ⟧_{⦃Γ⦄\varrho}$)
proof (*rule Aexp-correct*)
have *eq*ϱ (*Aheap Γ e·a* ⊔ *Aexp* (*Let Γ e*)·*a*) (⦃*map-transform Aeta-expand* (*Aheap Γ e·a*) (*map-transform transform* (*Aheap Γ e·a*) Γ)⦄ϱ) (⦃Γ⦄ϱ)
proof(*induction rule*: *parallel-HSem-ind*[*case-names adm bottom step*])
case *adm* **thus** *?case* **by** (*intro eq*ϱ*-adm cont2cont*)
next
case *bottom* **show** *?case* **by** *simp*
next
case (*step* ϱ*1* ϱ*2*)

have $eq\varrho$ $(Aheap\ \Gamma\ e{\cdot}a \sqcup Aexp\ (Let\ \Gamma\ e){\cdot}a)$ $(\llbracket$ *map-transform Aeta-expand* $(Aheap\ \Gamma\ e{\cdot}a)$ $($*map-transform transform* $(Aheap\ \Gamma\ e{\cdot}a)\ \Gamma)\ \rrbracket_{\varrho 1})$ $(\llbracket\Gamma\rrbracket_{\varrho 2})$
proof(*rule* $eq\varrho I$)
fix $x\ a'$
assume *ass*: $(Aheap\ \Gamma\ e{\cdot}a \sqcup Aexp\ (Let\ \Gamma\ e){\cdot}a)\ x = up{\cdot}a'$
show $eq\ a'$ $((\llbracket$ *map-transform Aeta-expand* $(Aheap\ \Gamma\ e{\cdot}a)$ $($*map-transform transform* $(Aheap\ \Gamma\ e{\cdot}a)\ \Gamma)\ \rrbracket_{\varrho 1})\ x)$ $((\llbracket\Gamma\rrbracket_{\varrho 2})\ x)$
proof(*cases* $x \in domA\ \Gamma$)
case [*simp*]: *True*
then obtain e' **where** [*simp*]: $map\text{-}of\ \Gamma\ x = Some\ e'$ **by** (*metis domA-map-of-Some-the*)
from *ass* **have** ass': $(Aheap\ \Gamma\ e{\cdot}a)\ x = up{\cdot}a'$ **by** *simp*

have $(\llbracket$ *map-transform Aeta-expand* $(Aheap\ \Gamma\ e{\cdot}a)$ $($*map-transform transform* $(Aheap\ \Gamma\ e{\cdot}a)\ \Gamma)\ \rrbracket_{\varrho 1})\ x =$
$\llbracket Aeta\text{-}expand\ a'\ (transform\ a'\ e')\rrbracket_{\varrho 1}$
by (*simp add*: *lookupEvalHeap′ map-of-map-transform ass′*)
also have $eq\ a'\ \ldots\ (\llbracket transform\ a'\ e'\rrbracket_{\varrho 1})$
by (*rule eq-Aeta-expand*)
also have $eq\ a'\ \ldots\ (\llbracket e'\rrbracket_{\varrho 1})$
by (*rule Let*(*1*)[*OF map-of-SomeD*[*OF* ‹*map-of - - = -*›]])
also have $eq\ a'\ \ldots\ (\llbracket e'\rrbracket_{\varrho 2})$
proof (*rule Aexp-correct*)
from ass' ‹*map-of - - = -*›
have $Aexp\ e'{\cdot}a' \sqsubseteq Aheap\ \Gamma\ e{\cdot}a \sqcup Aexp\ (Let\ \Gamma\ e){\cdot}a$ **by** (*rule Aexp-heap-below-Aheap*)
thus $eq\varrho\ (Aexp\ e'{\cdot}a')\ \varrho 1\ \varrho 2$ **using** *step* **by** (*rule* $eq\varrho$*-mono*)
qed
also have $\ldots = (\llbracket\Gamma\rrbracket_{\varrho 2})\ x$
by (*simp add*: *lookupEvalHeap′*)
finally
show *?thesis*.
next
case *False* **thus** *?thesis* **by** *simp*
qed
qed
thus *?case*
by (*simp add*: *env-restr-useless order-trans*[*OF edom-evalHeap-subset*] *del*: *fun-meet-simp* $eq\varrho$*-join*)
qed
thus $eq\varrho$ $(Aexp\ e{\cdot}a)$ $(\{\!\!\{$*map-transform Aeta-expand* $(Aheap\ \Gamma\ e{\cdot}a)$ $($*map-transform transform* $(Aheap\ \Gamma\ e{\cdot}a)\ \Gamma)\}\!\!\}\varrho)$ $(\{\!\!\{\Gamma\}\!\!\}\varrho)$
by (*rule* $eq\varrho$*-mono*[*OF Aexp-body-below-Aheap*])
qed
also have $\ldots = \llbracket\ Let\ \Gamma\ e\ \rrbracket_{\varrho}$
by *simp*
finally show *?case*.
qed

corollary *Arity-transformation-correct′*:
$\llbracket\ \mathcal{T}_0\ e\ \rrbracket_{\varrho} = \llbracket\ e\ \rrbracket_{\varrho}$

using *Arity-transformation-correct*[**where** *a* = *0*] **by** *simp*

end
end

References

[AO93] Samson Abramsky and Chih-Hao Luke Ong, *Full abstraction in the lazy lambda calculus*, Information and Computation **105** (1993), no. 2, 159 – 267.

[Huf12] Brian Huffman, *HOLCF '11: A definitional domain theory for verifying functional programs*, Ph.D. thesis, Portland State University, 2012.

[Lau93] John Launchbury, *A natural semantics for lazy evaluation*, POPL '93, 1993, pp. 144–154.

[Ses97] Peter Sestoft, *Deriving a lazy abstract machine*, Journal of Functional Programming **7** (1997), 231–264.

[SGHHOM11] Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero, and Yolanda Ortega-Mallén, *Relating function spaces to resourced function spaces*, SAC, 2011, pp. 1301–1308.