The Safety of Call Arity

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

November 7, 2026

We formalize the Call Arity analysis [Bre15a], as implemented in GHC, and prove both functional correctness and, more interestingly, safety (i.e. the transformation does not increase allocation). A highlevel overview of the work can be found in [Bre15b].

We use syntax and the denotational semantics from an earlier work [Bre13], where we formalized Launchbury's natural semantics for lazy evaluation [Lau93]. The functional correctness of Call Arity is proved with regard to that denotational semantics. The operational properties are shown with regard to a small-step semantics akin to Sestoft's mark 1 machine [Ses97], which we prove to be equivalent to Launchbury's semantics.

We use Christian Urban's Nominal2 package [UK12] to define our terms and make use of Brian Huffman's HOLCF package for the domain-theoretical aspects of the development [Huf12].

Artifact correspondence table

The following table connects the definitions and theorems from [Bre15b] with their corresponding Isabelle concept in this development.

Concept	corresponds to	in theory
Syntax	**nominal-datatype** *expr*	Terms in [Bre13]
Stack	**type-synonym** *stack*	SestoftConf
Configuration	**type-synonym** *conf*	SestoftConf
Semantics ($\Rightarrow$)	**inductive** *step*	Sestoft
Arity	**typedef** *Arity*	Arity
Eta-expansion	**lift-definition** *Aeta-expand*	ArityEtaExpansion

Lemma 1	**theorem** *Aeta-expand-safe*	ArityEtaExpansionSafe
$\mathcal{A}_\alpha(\Gamma, e)$	**locale** *ArityAnalysisHeap*	ArityAnalysisSig
$\mathsf{T}_\alpha(e)$	**sublocale** *AbstractTransformBound*	ArityTransform
$\mathcal{A}_\alpha(e)$	**locale** *ArityAnalysis*	ArityAnalysisSig
Definition 2	**locale** *ArityAnalysisLetSafe*	ArityAnalysisSpec
Definition 3	**locale** *ArityAnalysisLetSafeNoCard*	ArityAnalysisSpec
Definition 4	**inductive** *a-consistent*	ArityConsistent
Definition 5	**inductive** *consistent*	ArityTransformSafe
Lemma 2	**lemma** *arity-transform-safe*	ArityTransformSafe
Card	**type-synonym** *two*	Cardinality-Domain
$\mathcal{C}_\alpha(\Gamma, e)$	**locale** *CardinalityHeap*	CardinalityAnalysisSig
$\mathcal{C}_{(\bar{\alpha},\alpha,\dot{\alpha})}((\Gamma, e, S))$	**locale** *CardinalityPrognosis*	CardinalityAnalysisSig
Definition 6	**locale** *CardinalityPrognosisSafe*	CardinalityAnalysisSpec
Definition 7 ($\Rightarrow_\#$)	**inductive** *gc-step*	SestoftGC
Definition 8	**inductive** *consistent*	CardArityTransformSafe
Lemma 3	**lemma** *card-arity-transform-safe*	CardArityTransformSafe
Trace trees	**typedef** *'a ttree*	TTree
Function s	**lift-definition** *substitute*	TTree
$\mathcal{T}_\alpha(e)$	**locale** *TTreeAnalysis*	TTreeAnalysisSig
$\mathcal{T}_\alpha(\Gamma, e)$	**locale** *TTreeAnalysisCardinalityHeap*	TTreeAnalysisSpec
Definition 9	**locale** *TTreeAnalysisCardinalityHeap*	TTreeAnalysisSpec
Lemma 4	**sublocale** *CardinalityPrognosisSafe*	TTreeImplCardinalitySafe
Co-Call graphs	**typedef** *CoCalls*	CoCallGraph
Function g	**lift-definition** *ccApprox*	CoCallGraph-TTree
Function t	**lift-definition** *ccTTree*	CoCallGraph-TTree
$\mathcal{G}_\alpha(e)$	**locale** *CoCallAnalysis*	CoCallAnalysisSig
$\mathcal{G}_\alpha(\Gamma, e)$	**locale** *CoCallAnalysisHeap*	CoCallAnalysisSig
Definition 10	**locale** *CoCallAritySafe*	CoCallAnalysisSpec
Lemma 5	**sublocale** *TTreeAnalysisCardinalityHeap*	CoCallImplTTreeSafe
Call Arity	**nominal-function** *cCCexp*	CoCallAnalysisImpl
Theorem 1	**lemma** *end2end-closed*	CallArityEnd2EndSafe

References

[Bre13] Joachim Breitner, *The correctness of launchbury's natural semantics for lazy evaluation*, Archive of Formal Proofs (2013), http://isa-afp.org/entries/Launchbury.shtml, Formal proof development.

[Bre15a] ———, *Call Arity*, TFP'14, LNCS, vol. 8843, Springer, 2015, pp. 34–50.

[Bre15b] ———, *Formally proving a compiler transformation safe*, Haskell Symposium, 2015.

[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.

[UK12] Christian Urban and Cezary Kaliszyk, *General bindings and alpha-equivalence in nominal isabelle*, Logical Methods in Computer Science **8** (2012), no. 2.

Contents

1 Various Utilities **5**
- 1.1 ConstOn . . . 5
- 1.2 Set-Cpo . . . 6
- 1.3 Env-Set-Cpo . . . 8
- 1.4 AList-Utils-HOLCF . . . 8
- 1.5 List-Interleavings . . . 10

2 Small-step Semantics **13**
- 2.1 SestoftConf . . . 13
  - 2.1.1 Invariants of the semantics . . . 16
- 2.2 Sestoft . . . 19
  - 2.2.1 Equivariance . . . 22
  - 2.2.2 Invariants . . . 23
- 2.3 SestoftGC . . . 23
- 2.4 BalancedTraces . . . 31
- 2.5 SestoftCorrect . . . 36

3 Arity **42**
- 3.1 Arity . . . 42
- 3.2 AEnv . . . 45
- 3.3 Arity-Nominal . . . 45
- 3.4 ArityStack . . . 45

4 Eta-Expansion **46**
- 4.1 EtaExpansion . . . 46
- 4.2 EtaExpansionSafe . . . 48
- 4.3 TransformTools . . . 49
- 4.4 ArityEtaExpansion . . . 51

4.5 ArityEtaExpansionSafe . . . 52

5 Arity Analysis **53**
5.1 ArityAnalysisSig . . . 53
5.2 ArityAnalysisAbinds . . . 54
5.2.1 Lifting arity analysis to recursive groups . . . 54
5.3 ArityAnalysisSpec . . . 57
5.4 TrivialArityAnal . . . 60
5.5 ArityAnalysisStack . . . 62
5.6 ArityAnalysisFix . . . 62
5.7 ArityAnalysisFixProps . . . 68

6 Arity Transformation **68**
6.1 AbstractTransform . . . 68
6.2 ArityTransform . . . 74

7 Arity Analysis Safety (without Cardinality) **75**
7.1 ArityConsistent . . . 75
7.2 ArityTransformSafe . . . 81

8 Cardinality Analysis **87**
8.1 Cardinality-Domain . . . 87
8.2 CardinalityAnalysisSig . . . 89
8.3 CardinalityAnalysisSpec . . . 89
8.4 NoCardinalityAnalysis . . . 90
8.5 CardArityTransformSafe . . . 94

9 Trace Trees **106**
9.1 TTree . . . 106
9.1.1 Prefix-closed sets of lists . . . 106
9.1.2 The type of infinite labeled trees . . . 108
9.1.3 Deconstructors . . . 108
9.1.4 Trees as set of paths . . . 108
9.1.5 The carrier of a tree . . . 109
9.1.6 Repeatable trees . . . 110
9.1.7 Simple trees . . . 110
9.1.8 Intersection of two trees . . . 111
9.1.9 Disjoint union of trees . . . 112
9.1.10 Merging of trees . . . 112
9.1.11 Removing elements from a tree . . . 116
9.1.12 Multiple variables, each called at most once . . . 117
9.1.13 Substituting trees for every node . . . 119
9.2 TTree-HOLCF . . . 134

10 Trace Tree Cardinality Analysis **140**
10.1 AnalBinds . . . 140
10.2 TTreeAnalysisSig . . . 141
10.3 Cardinality-Domain-Lists . . . 141
10.4 TTreeAnalysisSpec . . . 144
10.5 TTreeImplCardinality . . . 145
10.6 TTreeImplCardinalitySafe . . . 145

11 Co-Call Graphs **154**
11.1 CoCallGraph . . . 154
11.2 CoCallGraph-Nominal . . . 163

12 Co-Call Cardinality Analysis **165**
12.1 CoCallAnalysisSig . . . 165
12.2 CoCallAnalysisBinds . . . 165
12.3 CoCallAritySig . . . 168
12.4 CoCallAnalysisSpec . . . 168
12.5 CoCallFix . . . 169
12.5.1 The non-recursive case . . . 173
12.5.2 Combining the cases . . . 175
12.6 CoCallGraph-TTree . . . 176
12.7 CoCallImplTTree . . . 194
12.8 CoCallImplTTreeSafe . . . 195

13 CoCall Cardinality Implementation **205**
13.1 CoCallAnalysisImpl . . . 205
13.2 CoCallImplSafe . . . 212

14 End-to-end Saftey Results and Example **223**
14.1 CallArityEnd2End . . . 223
14.2 CallArityEnd2EndSafe . . . 225

15 Functional Correctness of the Arity Analysis **228**
15.1 ArityAnalysisCorrDenotational . . . 228

1 Various Utilities

1.1 ConstOn

theory *ConstOn*
imports *Main*
begin

definition *const-on* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool$

where *const-on f S x* = (∀ *y* ∈ *S* . *f y* = *x*)

lemma *const-onI*[*intro*]: (⋀*y*. *y* ∈ *S* ⟹ *f y* = *x*) ⟹ *const-on f S x*
by (*simp add*: *const-on-def*)

lemma *const-onD*[*dest*]: *const-on f S x* ⟹ *y* ∈ *S* ⟹ *f y* = *x*
by (*simp add*: *const-on-def*)

lemma *const-on-insert*[*simp*]: *const-on f* (*insert x S*) *y* ⟷ *const-on f S y* ∧ *f x* = *y*
by *auto*

lemma *const-on-union*[*simp*]: *const-on f* (*S* ∪ *S'*) *y* ⟷ *const-on f S y* ∧ *const-on f S' y*
by *auto*

lemma *const-on-subset*[*elim*]: *const-on f S y* ⟹ *S'* ⊆ *S* ⟹ *const-on f S' y*
by *auto*

end

1.2 Set-Cpo

theory *Set−Cpo*
imports *HOLCF*
begin

default-sort *type*

instantiation *set* :: (*type*) *below*
begin
definition *below-set* **where** (⊑) = (⊆)
instance..
end

instance *set* :: (*type*) *po*
by *standard* (*auto simp add*: *below-set-def*)

lemma *is-lub-set*:
S <<| ⋃*S*
by(*auto simp add*: *is-lub-def below-set-def is-ub-def*)

lemma *lub-set*: *lub S* = ⋃*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-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 $S \subseteq \bigcup(Y\ `\ UNIV)$
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*

define i **where** $i = Max\ (f\ `\ S)$
from ‹*finite S*›
have *finite* $(f\ `\ S)$ **by** *simp*
hence $\forall\ x{\in}S.\ f\ x \leq i$ **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

1.3 Env-Set-Cpo

theory *Env−Set−Cpo*
imports *Launchbury.Env Set−Cpo*
begin

lemma *cont-edom*[*THEN cont-compose*, *simp*, *cont2cont*]:
cont $(\lambda\ f.\ edom\ f)$
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

1.4 AList-Utils-HOLCF

theory *AList−Utils−HOLCF*
imports *Launchbury.HOLCF−Utils Launchbury.HOLCF−Join−Classes Launchbury.AList−Utils*
begin

syntax
-BLubMap :: [*pttrn*, *pttrn*, $'a \rightharpoonup 'b$, $'b$] $\Rightarrow$ $'b$ ($\langle(3\bigsqcup$/-/$\mapsto$/-/$\in$/-/./ -)$\rangle$ [0,0,0, 10] 10)

syntax-consts
-BLubMap == *lub*

translations
$\bigsqcup\ k{\mapsto}v{\in}m.\ e$ == *CONST lub* (*CONST mapCollect* $(\lambda k\ v\ .\ e)\ m$)

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*]:

```
  (⋀ k v . m k = Some v ⟹ (e k v::'a::Join-cpo) ⊑ u) ⟹ (⨆ k↦v∈m. e k v) ⊑ u
unfolding mapCollect-def by auto

lemma lubmap-const-bottom[simp]:
  (⨆ k↦v∈m. ⊥) = (⊥::'a::Join-cpo)
  by (cases m = Map.empty) auto

lemma lubmap-map-upd[simp]:
  fixes e :: 'a ⇒ 'b ⇒ ('c :: Join-cpo)
  shows (⨆k↦v∈m(k' ↦ v'). e k v) = e k' v' ⊔ (⨆k↦v∈m(k':=None). e k v)
  by simp

lemma lubmap-below-cong:
  assumes ⋀ k v. m k = Some v ⟹ f1 k v ⊑ (f2 k v :: 'a :: Join-cpo)
  shows (⨆ k↦v∈m. f1 k v) ⊑ (⨆ k↦v∈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 (⋀k v . cont (f k v))
  shows cont (λx. ⨆ k↦v∈m. (f k v x) :: 'a :: Join-cpo)
proof (rule contI2)
  show monofun (λx. ⨆k↦v∈m. f k v x)
    apply (rule monofunI)
    apply (rule lubmap-below-cong)
    apply (erule cont2monofunE[OF assms])
    done
next
  fix Y :: nat ⇒ 'd
  assume chain Y
  assume chain (λi. ⨆k↦v∈m. f k v (Y i))

  show (⨆k↦v∈m. f k v (⨆ i. Y i)) ⊑ (⨆ i. ⨆k↦v∈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 (λi. ⨆k↦v∈m. f k v (Y
i))›])
    apply (erule below-lubmapI)
    done
qed




end
```

1.5 List-Interleavings

theory *List−Interleavings*
imports *Main*
begin

inductive *interleave′* :: *'a list* ⇒ *'a list* ⇒ *'a list* ⇒ *bool*
  where [*simp*]: *interleave′* [] [] []
  | *interleave′ xs ys zs* ⟹*interleave′* (*x*#*xs*) *ys* (*x*#*zs*)
  | *interleave′ xs ys zs* ⟹*interleave′ xs* (*x*#*ys*) (*x*#*zs*)

definition *interleave* :: *'a list* ⇒ *'a list* ⇒ *'a list set* (**infixr** ‹⊗› *64*)
  where *xs* ⊗ *ys* = *Collect* (*interleave′ xs ys*)
lemma *elim-interleave′*[*pred-set-conv*]: *interleave′ xs ys zs* ⟷ *zs* ∈ *xs* ⊗ *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*) ∈ *ys* ⊗ *zs*
inductive-cases *interleave-ConsConsE*[*elim*]: *xs* ∈ *y*#*ys* ⊗ *z*#*zs*
inductive-cases *interleave-ConsE2*[*elim*]: *xs* ∈ *x*#*ys* ⊗ *zs*
inductive-cases *interleave-ConsE3*[*elim*]: *xs* ∈ *ys* ⊗ *x*#*zs*

lemma *interleave-comm*: *xs* ∈ *ys* ⊗ *zs* ⟹ *xs* ∈ *zs* ⊗ *ys*
  by (*induction rule*: *interleave-induct*) (*auto intro*: *interleave-intros*)

lemma *interleave-Nil1*[*simp*]: [] ⊗ *xs* = {*xs*}
  by (*induction xs*) (*auto intro*: *interleave-intros elim*: *interleave-cases*)

lemma *interleave-Nil2*[*simp*]: *xs* ⊗ [] = {*xs*}
  by (*induction xs*) (*auto intro*: *interleave-intros elim*: *interleave-cases*)

lemma *interleave-nil-simp*[*simp*]: [] ∈ *xs* ⊗ *ys* ⟷ *xs* = [] ∧ *ys* = []
  by (*auto elim*: *interleave-cases*)

lemma *append-interleave*: *xs* @ *ys* ∈ *xs* ⊗ *ys*
  by (*induction xs*) (*auto intro*: *interleave-intros*)

lemma *interleave-assoc1*: *a* ∈ *xs* ⊗ *ys* ⟹ *b* ∈ *a* ⊗ *zs* ⟹ ∃ *c*. *c* ∈ *ys* ⊗ *zs* ∧ *b* ∈ *xs* ⊗ *c*
  by (*induction b arbitrary*: *a xs ys zs*)
    (*simp*, *fastforce del*: *interleave-ConsE elim*!: *interleave-ConsE intro*: *interleave-intros*)

lemma *interleave-assoc2*: *a* ∈ *ys* ⊗ *zs* ⟹ *b* ∈ *xs* ⊗ *a* ⟹ ∃ *c*. *c* ∈ *xs* ⊗ *ys* ∧ *b* ∈ *c* ⊗ *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*: $xs \in ys \otimes zs \Longrightarrow filter\ P\ xs \in filter\ P\ ys \otimes filter\ P\ zs$
by (*induction rule*: *interleave-induct*) (*auto intro*: *interleave-intros*)

lemma *interleave-filtered*: $xs \in interleave\ (filter\ P\ xs)\ (filter\ (\lambda x'.\ \neg\ P\ x')\ xs)$
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


end
```

2 Small-step Semantics

2.1 SestoftConf

```
theory SestoftConf
imports Launchbury.Terms Launchbury.Substitution
begin

datatype stack-elem = Alts exp exp | Arg var | Upd var | Dummy var

instantiation stack-elem :: pt
begin
definition  π · x = (case x of (Alts e1 e2) ⇒ Alts (π · e1) (π · e2) | (Arg v) ⇒ Arg (π · v) |
(Upd v) ⇒ Upd (π · v) | (Dummy v) ⇒ Dummy (π · v))
instance
  by standard (auto simp add: permute-stack-elem-def split:stack-elem.split)
end

lemma Alts-eqvt[eqvt]: π · (Alts e1 e2) = Alts (π · e1) (π · e2)
  and Arg-eqvt[eqvt]: π · (Arg v) = Arg (π · v)
  and Upd-eqvt[eqvt]: π · (Upd v) = Upd (π · v)
  and Dummy-eqvt[eqvt]: π · (Dummy v) = Dummy (π · 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*]: $a \sharp \mathit{Alts}\ e1\ e2 = (a \sharp e1 \wedge a \sharp e2)$ **unfolding** *fresh-def* **by** *auto*
lemma *fresh-star-Alts*[*simp*]: $a \sharp* \mathit{Alts}\ e1\ e2 = (a \sharp* e1 \wedge a \sharp* e2)$ **unfolding** *fresh-star-def* **by** *auto*
lemma *fresh-Arg*[*simp*]: $a \sharp \mathit{Arg}\ v = a \sharp v$ **unfolding** *fresh-def* **by** *auto*
lemma *fresh-Upd*[*simp*]: $a \sharp \mathit{Upd}\ v = a \sharp v$ **unfolding** *fresh-def* **by** *auto*
lemma *fresh-Dummy*[*simp*]: $a \sharp \mathit{Dummy}\ v = a \sharp v$ **unfolding** *fresh-def* **by** *auto*
lemma *fv-Alts*[*simp*]: $\mathit{fv}\ (\mathit{Alts}\ e1\ e2) = \mathit{fv}\ e1 \cup \mathit{fv}\ e2$ **unfolding** *fv-def* **by** *auto*
lemma *fv-Arg*[*simp*]: $\mathit{fv}\ (\mathit{Arg}\ v) = \mathit{fv}\ v$ **unfolding** *fv-def* **by** *auto*
lemma *fv-Upd*[*simp*]: $\mathit{fv}\ (\mathit{Upd}\ v) = \mathit{fv}\ v$ **unfolding** *fv-def* **by** *auto*
lemma *fv-Dummy*[*simp*]: $\mathit{fv}\ (\mathit{Dummy}\ v) = \mathit{fv}\ v$ **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 subsetD*[*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* ∈ *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* ∈ *V*
using *arg-cong*[**where** *f* = *upds*, *OF assms*]
by *auto*

lemma *restr-stack-noop*[*simp*]:
restr-stack V S = *S* ⟷ *upds S* ⊆ *V*
by (*induction V S rule*: *restr-stack.induct*)
(*auto dest*: *Upd-eq-restr-stackD2*)

2.1.1 Invariants of the semantics

inductive *invariant* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*a* ⇒ *bool*) ⇒ *bool*
where (⋀ *x y*. *rel x y* ⟹ *I x* ⟹ *I y*) ⟹ *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* (λ *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* $\Rightarrow$ *bool*
where *closed* (Γ, *e*, *S*) $\longleftrightarrow$ *fv* (Γ, *e*, *S*) $\subseteq$ *domA* Γ $\cup$ *upds S*

fun *heap-upds-ok* **where** *heap-upds-ok* (Γ,*S*) $\longleftrightarrow$ *domA* Γ $\cap$ *upds S* = {} $\wedge$ *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*: *subsetD*[*OF ups-fv-subset*])

lemma *heap-upds-ok-to-stack*:
  x ∈ *domA* Γ ⟹ *heap-upds-ok* (Γ, *S*) ⟹ *heap-upds-ok* (*delete x* Γ, *Upd x* #*S*)
  by (*auto*)

lemma *heap-upds-ok-to-stack′*:
  map-of Γ *x* = *Some e* ⟹ *heap-upds-ok* (Γ, *S*) ⟹ *heap-upds-ok* (*delete x* Γ, *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-upds-ok (Γ, *S*) ⟹ *heap-upds-ok* (*delete x* Γ, *S*)
  by *auto*

lemma *heap-upds-ok-restrictA*:
  heap-upds-ok (Γ, *S*) ⟹ *heap-upds-ok* (*restrictA V* Γ, *S*)
  by *auto*

lemma *heap-upds-ok-restr-stack*:
  heap-upds-ok (Γ, *S*) ⟹ *heap-upds-ok* (Γ, *restr-stack V S*)
  apply *auto*
  by (*induction V S rule*: *restr-stack.induct*) *auto*

lemma *heap-upds-ok-to-heap*:

heap-upds-ok (Γ, *Upd x # S*) ⟹ *heap-upds-ok* ((*x*,*e*) # Γ, *S*)
by *auto*

lemma *heap-upds-ok-reorder*:
x ∈ *domA* Γ ⟹ *heap-upds-ok* (Γ, *S*) ⟹ *heap-upds-ok* ((*x*,*e*) # *delete x* Γ, *S*)
by (*intro heap-upds-ok-to-heap heap-upds-ok-to-stack*)

lemma *heap-upds-ok-upd*:
heap-upds-ok (Γ, *Upd x # S*) ⟹ *x* ∉ *domA* Γ ∧ *x* ∉ *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

2.2 Sestoft

theory *Sestoft*
imports *SestoftConf*
begin

inductive *step* :: *conf* ⇒ *conf* ⇒ *bool* (**infix** ‹⇒› *50*) **where**
app$_1$: (Γ, *App e x*, *S*) ⇒ (Γ, *e* , *Arg x # S*)
| *app*$_2$: (Γ, *Lam* [*y*]. *e*, *Arg x # S*) ⇒ (Γ, *e*[*y* ::= *x*] , *S*)
| *var*$_1$: *map-of* Γ *x* = *Some e* ⟹ (Γ, *Var x*, *S*) ⇒ (*delete x* Γ, *e* , *Upd x # S*)
| *var*$_2$: *x* ∉ *domA* Γ ⟹ *isVal e* ⟹ (Γ, *e*, *Upd x # S*) ⇒ ((*x*,*e*)# Γ, *e* , *S*)
| *let*$_1$: *atom* ‘ *domA* Δ ♯* Γ ⟹ *atom* ‘ *domA* Δ ♯* *S*
⟹ (Γ, *Let* Δ *e*, *S*) ⇒ (Δ@Γ, *e* , *S*)
| *if*$_1$: (Γ, *scrut ? e1 : e2*, *S*) ⇒ (Γ, *scrut*, *Alts e1 e2 # S*)
| *if*$_2$: (Γ, *Bool b*, *Alts e1 e2 # S*) ⇒ (Γ, *if b then e1 else e2*, *S*)

abbreviation *steps* (**infix** ‹⇒*› *50*) **where** *steps* ≡ *step***

lemma *SmartLet-stepI*:
atom ‘ *domA* Δ ♯* Γ ⟹ *atom* ‘ *domA* Δ ♯* *S* ⟹ (Γ, *SmartLet* Δ *e*, *S*) ⇒* (Δ@Γ, *e* , *S*)
unfolding *SmartLet-def* **by** (*auto intro*: *let*$_1$)

lemma *lambda-var*: *map-of* Γ *x* = *Some e* ⟹ *isVal e* ⟹ (Γ, *Var x*, *S*) ⇒* ((*x*,*e*) # *delete x* Γ, *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* (Γ, *Let* Δ *e*, *S*)
assumes *domA* Δ ∩ *domA* Γ = {}
assumes *domA* Δ ∩ *upds S* = {}
shows (Γ, *Let* Δ *e*, *S*) ⇒ (Δ@Γ, *e* , *S*)
proof
from ‹*domA* Δ ∩ *domA* Γ = {}› **and** ‹*domA* Δ ∩ *upds S* = {}›
have *domA* Δ ∩ (*domA* Γ ∪ *upds S*) = {} **by** *auto*
with ‹*closed* -›
have *domA* Δ ∩ *fv* (Γ, *S*) = {} **by** *auto*
hence *atom* ‘ *domA* Δ ♯∗ (Γ, *S*)
by (*auto simp add*: *fresh-star-def fv-def fresh-def*)
thus *atom*‘ *domA* Δ ♯∗ Γ **and** *atom* ‘ *domA* Δ ♯∗ *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* ⇒∗ *c′*
assumes ¬ *boring-step c′*
assumes *invariant* (⇒) *I*
assumes *I c*
assumes *app$_1$*: ⋀ Γ *e x S* . *I* (Γ, *App e x*, *S*) ⟹ *P* (Γ, *App e x*, *S*) (Γ, *e* , *Arg x* # *S*)
assumes *app$_2$*: ⋀ Γ *y e x S* . *I* (Γ, *Lam* [*y*]. *e*, *Arg x* # *S*) ⟹ *P* (Γ, *Lam* [*y*]. *e*, *Arg x* # *S*) (Γ, *e*[*y* ::= *x*] , *S*)
assumes *thunk*: ⋀ Γ *x e S* . *map-of* Γ *x* = *Some e* ⟹ ¬ *isVal e* ⟹ *I* (Γ, *Var x*, *S*) ⟹ *P* (Γ, *Var x*, *S*) (*delete x* Γ, *e* , *Upd x* # *S*)
assumes *lamvar*: ⋀ Γ *x e S* . *map-of* Γ *x* = *Some e* ⟹ *isVal e* ⟹ *I* (Γ, *Var x*, *S*) ⟹ *P* (Γ, *Var x*, *S*) ((*x*,*e*) # *delete x* Γ, *e* , *S*)
assumes *var$_2$*: ⋀ Γ *x e S* . *x* ∉ *domA* Γ ⟹ *isVal e* ⟹ *I* (Γ, *e*, *Upd x* # *S*) ⟹ *P* (Γ, *e*, *Upd x* # *S*) ((*x*,*e*)# Γ, *e* , *S*)
assumes *let$_1$*: ⋀ Δ Γ *e S* . *atom* ‘ *domA* Δ ♯∗ Γ ⟹ *atom* ‘ *domA* Δ ♯∗ *S* ⟹ *I* (Γ, *Let* Δ *e*, *S*) ⟹ *P* (Γ, *Let* Δ *e*, *S*) (Δ@Γ, *e*, *S*)
assumes *if$_1$*: ⋀Γ *scrut e1 e2 S*. *I* (Γ, *scrut ? e1* : *e2*, *S*) ⟹ *P* (Γ, *scrut ? e1* : *e2*, *S*) (Γ, *scrut*, *Alts e1 e2* # *S*)
assumes *if$_2$*: ⋀Γ *b e1 e2 S*. *I* (Γ, *Bool b*, *Alts e1 e2* # *S*) ⟹ *P* (Γ, *Bool b*, *Alts e1 e2* # *S*) (Γ, *if b then e1 else e2*, *S*)
assumes *refl*: ⋀ *c*. *P c c*
assumes *trans*[*trans*]: ⋀ *c c′ c″*. *c* ⇒∗ *c′* ⟹ *c′* ⇒∗ *c″* ⟹ *P c c′* ⟹ *P c′ c″* ⟹ *P c c″*
shows *P c c′*
proof−
from *assms*(*1*,*3*,*4*)
have *P c c′* ∨ (*boring-step c′* ∧ (∀ *c″*. *c′* ⇒ *c″* ⟶ *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 ‹c ⇒* y› r-into-rtranclp[where r = step, OF ‹y ⇒ z›]]

    from ‹y ⇒ z›
    show ?thesis
    proof (cases)
      case app1 hence P y z using assms(5) ‹I y› by metis
      with ‹P c y› show ?thesis by (metis t)
    next
      case app2 hence P y z using assms(6) ‹I y› by metis
      with ‹P c y› show ?thesis by (metis t)
    next
      case (var1 Γ x e S)
      show ?thesis
      proof (cases isVal e)
        case False with var1 have P y z using assms(7) ‹I y› by metis
        with ‹P c y› show ?thesis by (metis t)
      next
        case True
        have ∗: y ⇒* ((x,e) # delete x Γ, e , S) using var1 True lambda-var by metis

        have boring-step (delete x Γ, e, Upd x # S) using True ..
        moreover
        have P y ((x,e) # delete x Γ, e , S) using var1 True assms(8) ‹I y› by metis
        with ‹P c y› have P c ((x,e) # delete x Γ, e , S) by (rule trans[OF ‹c ⇒* y› ∗])
        ultimately
        show ?thesis using var1(2,3) True by (auto elim!: step.cases)
      qed
    next
      case var2 hence P y z using assms(9) ‹I y› by metis
      with ‹P c y› show ?thesis by (metis t)
    next
      case let1 hence P y z using assms(10) ‹I y› by metis
      with ‹P c y› show ?thesis by (metis t)
    next
      case if1 hence P y z using assms(11) ‹I y› by metis
      with ‹P c y› show ?thesis by (metis t)
    next
      case if2 hence P y z using assms(12) ‹I y› by metis
      with ‹P c y› show ?thesis by (metis t)
    qed
  next
    assume boring-step y ∧ (∀ c''. y ⇒ c'' ⟶ P c c'')
    with ‹y ⇒ z›
```

have $P\ c\ z$ **by** *blast*
thus *?thesis*..
qed
qed
with *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 ¬ *boring-step* c'
assumes *app*$_1$: $\bigwedge \Gamma\ e\ x\ S\ .\ P\ (\Gamma,\ App\ e\ x,\ S)\ \ (\Gamma,\ e\ ,\ Arg\ x\ \#\ S)$
assumes *app*$_2$: $\bigwedge \Gamma\ y\ e\ x\ S\ .\ P\ (\Gamma,\ Lam\ [y].\ e,\ Arg\ x\ \#\ S)\ (\Gamma,\ e[y ::= x]\ ,\ S)$
assumes *thunk*: $\bigwedge \Gamma\ x\ e\ S\ .\ map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow \neg\ isVal\ e \Longrightarrow P\ (\Gamma,\ Var\ x,\ S)\ (delete\ x\ \Gamma,\ e\ ,\ Upd\ x\ \#\ S)$
assumes *lamvar*: $\bigwedge \Gamma\ x\ e\ S\ .\ map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow isVal\ e \Longrightarrow P\ (\Gamma,\ Var\ x,\ S)\ ((x,e)\ \#\ delete\ x\ \Gamma,\ e\ ,\ S)$
assumes *var*$_2$: $\bigwedge \Gamma\ x\ e\ S\ .\ x \notin domA\ \Gamma \Longrightarrow isVal\ e \Longrightarrow P\ (\Gamma,\ e,\ Upd\ x\ \#\ S)\ ((x,e)\#\ \Gamma,\ e\ ,\ S)$
assumes *let*$_1$: $\bigwedge \Delta\ \Gamma\ e\ S\ .\ atom\ `\ domA\ \Delta\ \sharp*\ \Gamma \Longrightarrow atom\ `\ domA\ \Delta\ \sharp*\ S \Longrightarrow P\ (\Gamma,\ Let\ \Delta\ e,\ S)\ (\Delta@\Gamma,\ e,\ S)$
assumes *if*$_1$: $\bigwedge\Gamma\ scrut\ e1\ e2\ S.\ P\ (\Gamma,\ scrut\ ?\ e1\ :\ e2,\ S)\ (\Gamma,\ scrut,\ Alts\ e1\ e2\ \#\ S)$
assumes *if*$_2$: $\bigwedge\Gamma\ b\ e1\ e2\ S.\ P\ (\Gamma,\ Bool\ b,\ Alts\ e1\ e2\ \#\ S)\ (\Gamma,\ if\ b\ then\ e1\ else\ e2,\ S)$
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 $P\ c\ c'$
by (*rule step-invariant-induction*[*OF - - invariant-True*, *simplified*, *OF assms*])

2.2.1 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

2.2.2 Invariants

lemma *closed-invariant*:
invariant step closed
proof
fix *c c′*
assume *c* ⇒ *c′* **and** *closed c*
thus *closed c′*
by (*induction rule*: *step.induct*) (*auto simp add*: *fv-subst-eq dest*!: *subsetD*[*OF fv-delete-subset*] *dest*: *subsetD*[*OF map-of-Some-fv-subset*])
qed

lemma *heap-upds-ok-invariant*:
invariant step heap-upds-ok-conf
proof
fix *c c′*
assume *c* ⇒ *c′* **and** *heap-upds-ok-conf c*
thus *heap-upds-ok-conf c′*
by (*induction rule*: *step.induct*) *auto*
qed

end

2.3 SestoftGC

theory *SestoftGC*
imports *Sestoft*
begin

inductive *gc-step* :: *conf* ⇒ *conf* ⇒ *bool* (**infix** ‹⇒$_G$› *50*) **where**
normal: *c* ⇒ *c′* ⟹ *c* ⇒$_G$ *c′*
| *dropUpd*: (Γ, *e*, *Upd x* # *S*) ⇒$_G$ (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** ‹⇒$_G^*$› *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*) ⇒$_G^*$ (*delete x* Γ, *e* , *S*@[*Dummy x*])
proof−
from *assms*
have (Γ, *Var x*, *S*) ⇒$_G$ (*delete x* Γ, *e* , *Upd x* # *S*)..
moreover
have ... ⇒$_G$ (*delete x* Γ, *e*, *S*@[*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* ⇒ *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)) ⟷ 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) ⇒* (Γ', e, S')
    to-gc-conf r (Γ, e, S) = to-gc-conf r (Γ', e, S')
    ¬ isVal e ∨ 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 ∈ set r
          by(cases s) (auto split: if-splits)

        from ‹heap-upds-ok (Γ, s # S)› ‹s = Upd x›
        have [simp]: x ∉ domA Γ and heap-upds-ok ((x,e) # Γ, S)
          by (auto dest: heap-upds-okE)

        have (Γ, e, s # S) ⇒* (Γ, e, Upd x # S) unfolding ‹s = -› ..
```

```
      also have ... ⇒ ((x,e) # Γ, e, S) by (rule step.var_2[OF ‹x ∉ 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) ⇒* (Γ', e, S')
         and res: to-gc-conf r ((x,e) # Γ, e, S) = to-gc-conf r (Γ', e, S')
                  (¬ isVal e ∨ safe-hd S' = safe-hd (restr-stack (− set r) S'))
        by blast
      note this(1)
      finally
      have (Γ, e, s # S) ⇒* (Γ', e, S').
      thus ?thesis  using res by auto
    qed
  qed
 qed
qed

lemma perm-exI-trivial:
  P x x ⟹ ∃ π. P (π ∙ 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∈V) (upds-list S)
by (induction S rule: restr-stack.induct) auto

lemma heap-upd-ok-to-gc-conf:
  heap-upds-ok (Γ, S) ⟹ to-gc-conf r (Γ, e, S) = (Γ'', e'', S'') ⟹ 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 ⇒_G d
  assumes heap-upds-ok-conf c
  assumes closed c
  and r-ok r c
  shows  ∃ r' c'. c ⇒* c' ∧ d = to-gc-conf r' c' ∧ 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
    (Γ, e, S) ⇒* (Γ', e, S')
    and ∗: to-gc-conf r (Γ, e, S) = to-gc-conf r (Γ', e, S')
    and disj: ¬ isVal e ∨ safe-hd S' = safe-hd (restr-stack (− set r) S')
    by metis
```

```
  from invariant-starE[OF ‹- ⇒* -› heap-upds-ok-invariant]  ‹heap-upds-ok (Γ,S)›
  have heap-upds-ok (Γ', S') by auto

  from invariant-starE[OF ‹- ⇒* -› closed-invariant  ‹closed (Γ, e, S)› ]
  have closed (Γ', e, S') by auto

  from invariant-starE[OF ‹- ⇒* -› subset-bound-invariant ‹r-ok r (Γ, e, S)› ]
  have r-ok r (Γ', e, S') by auto

  from assms(1)[unfolded ‹c =- › ∗]
  have ∃ r' Γ'' e'' S''. (Γ', e, S') ⇒* (Γ'', e'', S'') ∧ d = to-gc-conf r' (Γ'', e'', S'') ∧ r-ok r'
(Γ'', e'', S'')
  proof(cases rule: gc-step.cases)
    case normal
    hence ∃ Γ'' e'' S''. (Γ', e, S') ⇒ (Γ'', e'', S'') ∧ d = to-gc-conf r (Γ'', e'', S'')
    proof(cases rule: step.cases)
      case app1
      thus ?thesis
        apply auto
        apply (intro exI conjI)
        apply (rule  step.intros)
        apply auto
        done
    next
      case (app2 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 var1
      thus ?thesis
        apply auto
        apply (intro  exI conjI)
        apply (rule step.intros)
        apply (auto simp add: restr-delete-twist)
        done
    next
      case var2
      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 (let1 Δ'' Γ'' S'' e')

    from ‹closed (Γ', e, S')› let1
    have closed (Γ', Let Δ'' e', S') by simp

    from fresh-distinct[OF let1(3)] fresh-distinct-fv[OF let1(4)]
     have domA Δ'' ∩ domA Γ'' = {} and domA Δ'' ∩ upds S'' = {}  and domA Δ'' ∩
dummies S'' = {}
      by (auto dest: subsetD[OF ups-fv-subset] subsetD[OF dummies-fv-subset])
    moreover
    from let1(1)
    have domA Γ' ∪ upds S' ⊆ domA Γ'' ∪ upds S'' ∪ dummies S''
      by auto
    ultimately
    have disj: domA Δ'' ∩ domA Γ' = {} domA Δ'' ∩ upds S' = {}
      by auto

    from ‹domA Δ'' ∩ dummies S'' = {}› let1(1)
    have domA Δ'' ∩ set r = {} by auto
    hence [simp]: restrictA (− set r) Δ'' = Δ''
      by (auto intro: restrictA-noop)

    from let1(1−3)
    show ?thesis
      apply auto
      apply (intro  exI[where x = r] exI[where x = Δ'' @ Γ'] exI[where x = S'] conjI)
      apply (rule let1-closed[OF ‹closed (Γ', Let Δ'' e', S')› disj])
      apply (auto simp add: restrictA-append)
      done
  next
    case if1
    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 if2
    thus ?thesis
      using disj
      apply (cases S')
      apply auto
      apply (intro exI exI conjI)
     apply (rule step.if2[where b = True, simplified] step.if2[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 ∉ 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 ∉ domA Γ'' x ∉ 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 ∈ 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') ⇒* (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) ⇒* (Γ', e, S')› and ‹(Γ', e, S') ⇒* (Γ'', e'', S'')›
have (Γ, e, S) ⇒* (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 $\exists\ \Gamma''.$ ([], *e*, []) $\Rightarrow^*$ (Γ'', *e'*, []) $\wedge\ \Gamma =$ *restrictA* (− *set r*) $\Gamma'' \wedge$ *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 $\Gamma =$ *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*, []) ⇒* (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* ‹- ⇒* -› *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*′) ⇒* (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*, []) ⇒* (Γ′, *e*′, *S*′)› **and** ‹(Γ′, *e*′, *S*′) ⇒* (Γ′′, *e*′, *S*′′)› **and** ‹*S*′′ = []›
have ([], *e*, []) ⇒* (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*′) ⇒* (Γ′′, *e*′, *S*′′)› *subset-bound-invariant* ‹*r-ok r*′ (Γ′, *e*′, *S*′)›]
have *set r* ⊆ *domA* Γ′′ **using** ‹*S*′′ = []› **by** *auto*
ultimately
show *?thesis* **by** *blast*
qed

end

2.4 BalancedTraces

theory *BalancedTraces*
imports *Main*
begin

locale *traces* =
fixes *step* :: *'c* => *'c* => *bool* (**infix** ‹⇒› *50*)
begin

abbreviation *steps* (**infix** ‹⇒*› *50*) **where** *steps* ≡ *step*⁺⁺

inductive *trace* :: *'c* ⇒ *'c list* ⇒ *'c* ⇒ *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*$_2$ ∧ *trace conf* (*conf'* # *T*$_1$) *c*$_2$ ∧ (∀ *x*∈*set* (*conf'* # *T*$_1$). *P x*) ∧ *P c*$_2$ ∧ *c*$_2$ ⇒ *c*$_3$ ∧ ¬ *P c*$_3$ ∧ *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* ∧ (∀ *x*∈*set* []. *P x*) ∧ *P conf* ∧ *conf* ⇒ *conf'* ∧ ¬ *P conf'* ∧ *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' ⟹ *P c'* ⟹ (⋀ *c c'*. *c* ⇒ *c'* ⟹ *P c'* ⟹ *P c*) ⟹ (∀ *x*∈*set T*. *P x*) ∧ *P c*
by (*induction rule*:*trace.induct*) *auto*

lemma *trace-nil*[*simp*]: *trace c* [] *c'* ⟷ *c* = *c'*
by (*metis list.distinct*(*1*) *trace.cases traces.trace-nil*)

end

definition *extends* :: *'a list* ⇒ *'a list* ⇒ *bool* (**infix** ‹≲› *50*) **where**
S ≲ *S'* = (∃ *S''*. *S'* = *S''* @ *S*)

lemma *extends-refl*[*simp*]: *S* ≲ *S* **unfolding** *extends-def* **by** *auto*
lemma *extends-cons*[*simp*]: *S* ≲ *x* # *S* **unfolding** *extends-def* **by** *auto*
lemma *extends-append*[*simp*]: *S* ≲ *L* @ *S* **unfolding** *extends-def* **by** *auto*
lemma *extends-not-cons*[*simp*]: ¬ (*x* # *S*) ≲ *S* **unfolding** *extends-def* **by** *auto*
lemma *extends-trans*[*trans*]: *S* ≲ *S'* ⟹ *S'* ≲ *S''* ⟹ *S* ≲ *S''* **unfolding** *extends-def* **by** *auto*

locale *balance-trace* = *traces* +
fixes *stack* :: *'a* ⇒ *'s list*
assumes *one-step-only*: *c* ⇒ *c'* ⟹ (*stack c*) = (*stack c'*) ∨ (∃ *x*. *stack c'* = *x* # *stack c*) ∨ (∃ *x*. *stack c* = *x* # *stack c'*)
begin

inductive *bal* :: *'a* ⇒ *'a list* ⇒ *'a* ⇒ *bool* **where**
balI[*intro*]: *trace c T c'* ⟹ ∀ *c'*∈ *set T*. *stack c* ≲ *stack c'* ⟹ *stack c'* = *stack c* ⟹ *bal c T c'*

inductive-cases *balE*: *bal c T c'*

lemma *bal-nil*[*simp*]: *bal c* [] *c'* ⟷ *c* = *c'*
by (*auto elim*: *balE trace.cases*)

lemma *bal-stackD*: *bal c T c'* ⟹ *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* $c_2 \wedge$ *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

2.5 SestoftCorrect

theory *SestoftCorrect*
imports *BalancedTraces Launchbury.Launchbury Sestoft*
begin

lemma *lemma-2*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  and *fv* $(\Gamma, e, S) \subseteq$ *set* $L \cup$ *domA* Γ
  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 ‹*fv* $(\Gamma,$ *App* e $x,$ $S) \subseteq$ *set* $L \cup$ *domA* Γ›
  have *prem1*: *fv* $(\Gamma, e,$ *Arg* $x \# S) \subseteq$ *set* $L \cup$ *domA* Γ **by** *simp*

  from *prem1 reds-pres-closed*[*OF* ‹$\Gamma : e \Downarrow_L \Delta :$ *Lam* $[y].\ e'$›] *reds-doesnt-forget*[*OF* ‹$\Gamma : e \Downarrow_L \Delta :$ *Lam* $[y].\ e'$›]
  have *prem2*: *fv* $(\Delta, e'[y::=x], S) \subseteq$ *set* $L \cup$ *domA* Δ **by** (*auto simp add*: *fv-subst-eq*)

  have $(\Gamma,$ *App* e $x,$ $S) \Rightarrow (\Gamma, e,$ *Arg* $x \# S)$..
  also have ... $\Rightarrow^* (\Delta,$ *Lam* $[y].\ e',$ *Arg* $x \# S)$ **by** (*rule Application.IH*(*1*)[*OF prem1*])
  also have ... $\Rightarrow (\Delta, e'[y ::= x], S)$..
  also have ... $\Rightarrow^* (\Theta, z, S)$ **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 $x \notin$ *domA* Δ **by** (*rule reds-avoids-live*[*OF Variable*(*2*), **where** $x = x$]) *simp-all*

  from ‹*fv* $(\Gamma,$ *Var* $x,$ $S) \subseteq$ *set* $L \cup$ *domA* Γ›
  have *prem*: *fv* $($*delete* x $\Gamma,$ $e,$ *Upd* $x \# S) \subseteq$ *set* $(x\#L) \cup$ *domA* $($*delete* x $\Gamma)$
    by (*auto dest*: *subsetD*[*OF fv-delete-subset*] *subsetD*[*OF map-of-Some-fv-subset*[*OF* ‹*map-of* Γ $x =$ *Some* e›]])

  from ‹*map-of* Γ $x =$ *Some* e›
  have $(\Gamma,$ *Var* $x,$ $S) \Rightarrow ($*delete* x $\Gamma,$ $e,$ *Upd* $x \# S)$..
  also have ... $\Rightarrow^* (\Delta, z,$ *Upd* $x \# S)$ **by** (*rule Variable.IH*[*OF prem*])
  also have ... $\Rightarrow ((x,z)\#\Delta, z, S)$ **using** ‹$x \notin$ *domA* Δ› ‹*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*) ⇒ (Γ, *scrut*, *Alts* e_1 e_2 #*S*)**..**
also
from *IfThenElse.prems*
have *prem1*: *fv* (Γ, *scrut*, *Alts* e_1 e_2 #*S*) ⊆ *set L* ∪ *domA* Γ **by** *auto*
hence (Γ, *scrut*, *Alts* e_1 e_2 #*S*) ⇒* (Δ, *Bool b*, *Alts* e_1 e_2 #*S*)
by (*rule IfThenElse.IH*)
also
have (Δ, *Bool b*, *Alts* e_1 e_2 #*S*) ⇒ (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*) ⊆ *set L* ∪ *domA* Δ **by** *auto*
hence (Δ, *if b then* e_1 *else* e_2, *S*) ⇒* (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*) ⊆ *set L* ∪ *domA* (*as* @ Γ) **by** *auto*

from *Let*(*1*)
have *atom* ‘ *domA as* ♯* Γ **by** (*auto simp add*: *fresh-star-Pair*)
moreover
from *Let*(*1*)
have *domA as* ∩ *fv* (Γ, *L*) = {}
by (*rule fresh-distinct-fv*)
hence *domA as* ∩ (*set L* ∪ *domA* Γ) = {}
by (*auto dest*: *subsetD*[*OF domA-fv-subset*])
with *Let*(*4*)
have *domA as* ∩ *fv S* = {}
by *auto*
hence *atom* ‘ *domA as* ♯* *S*
by (*auto simp add*: *fresh-star-def fv-def fresh-def*)
ultimately
have (Γ, *Terms.Let as body*, *S*) ⇒ (*as*@Γ, *body*, *S*)**..**
also have ... ⇒* (Δ, *z*, *S*)
by (*rule Let.IH*[*OF prem*])
finally show *?case***.**
qed

type-synonym *trace* = *conf list*

fun *stack* :: *conf* ⇒ *stack* **where** *stack* (Γ, *e*, *S*) = *S*

interpretation *traces step***.**

abbreviation *trace-syn* (‹- ⇒*_ -› [*50*,*50*,*50*] *50*) **where** *trace-syn* ≡ *trace*

lemma *conf-trace-induct-final*[*consumes 1*, *case-names trace-nil trace-cons*]:

$(\Gamma,\ e,\ S) \Rightarrow^*{}_T\ \mathit{final} \Longrightarrow (\bigwedge \Gamma\ e\ S.\ \mathit{final} = (\Gamma,\ e,\ S) \Longrightarrow P\ \Gamma\ e\ S\ []\ (\Gamma,\ e,\ S)) \Longrightarrow (\bigwedge \Gamma\ e\ S\ T\ \Gamma'\ e'\ S'.\ (\Gamma',\ e',\ S') \Rightarrow^*{}_T\ \mathit{final} \Longrightarrow P\ \Gamma'\ e'\ S'\ T\ \mathit{final} \Longrightarrow (\Gamma,\ e,\ S) \Rightarrow (\Gamma',\ e',\ S') \Longrightarrow P\ \Gamma\ e\ S\ ((\Gamma',\ e',\ S')\ \#\ T)\ \mathit{final}) \Longrightarrow P\ \Gamma\ e\ S\ T\ \mathit{final}$
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*]:
$(\forall\, p{\in}\mathit{set}\ (\Gamma[y::h{=}x]).\ f\ p) \longleftrightarrow (\forall\, p{\in}\mathit{set}\ \Gamma.\ \mathit{case}\ p\ \mathit{of}\ (z,e) \Rightarrow f\ (z,\ e[y::{=}x]))$
by (*induction* Γ) *auto*

lemma *lemma-3*:
assumes $(\Gamma,\ e,\ S) \Rightarrow^{b*}{}_T\ (\Delta,\ z,\ S)$
assumes *isVal z*
shows $\Gamma : e \Downarrow_{\mathit{upds\text{-}list}\ S} \Delta : z$
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** $\forall\ c'{\in}\mathit{set}\ T.\ S \lesssim \mathit{stack}\ c'$ **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** ‹$\forall\ c'{\in}\mathit{set}\ T.\ S \lesssim \mathit{stack}\ c'$› **have** $S \lesssim \mathit{stack}\ \mathit{conf'}$ **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-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-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* ≲ *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* $x\ \Gamma : e \Downarrow_{x\ \#\ upds\text{-}list\ S} \Delta' : z$ **by** *simp*
with ‹*map-of* - - = -›
show *?thesis* **unfolding** $var_1(1)$ ‹$\Delta = $ -› **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*: (Γ, *scrut*, *Alts* $e_1\ e_2\ \#\ S$) $\Rightarrow^{b*}{}_{T_1}\ c_3$ **and** $c_3 \Rightarrow c_4$ **and** *prem2*: $c_4 \Rightarrow^{b*}{}_{T_2}$ (Δ, 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 $\Delta'\ b$ **where** $c_3 = (\Delta', Bool\ b, Alts\ e_1\ e_2\ \#\ S)$ **and** $c_4 = (\Delta', (if\ b\ then\ e_1\ else\ e_2), S)$
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 Γ : *scrut* $\Downarrow_{upds\text{-}list\ S} \Delta'$: *Bool* b **by** *simp*
moreover
from *less*(*1*)[*OF* ‹*length* T_2 < *length* T› *prem2*[*unfolded* ‹$c_4 = $ -›] ‹*isVal* z›]
have Δ' : ($if\ b\ then\ e_1\ else\ e_2$) $\Downarrow_{upds\text{-}list\ S} \Delta : z$.
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 (let1 as e)
      from ‹T = conf' # T'› have length T' < length T by auto
      moreover
      have (as @ Γ, e, S) ⇒^b*_T' (Δ, z, S)
        using trace-cons ‹conf' = -› ‹∀ c'∈set T. S ≲ stack c'› by fastforce
      moreover
      note ‹isVal z›
      ultimately
      have as @ Γ : e ⇓upds-list S Δ : z by (rule less)
      moreover
      from ‹atom ‘ domA as ♯* Γ› ‹atom ‘ domA as ♯* S›
      have atom ‘ domA as ♯* (Γ, upds-list S) by (auto simp add: fresh-star-Pair)
      ultimately
      show ?thesis unfolding let1 by (rule reds.Let[rotated])
    qed
  qed
qed

lemma dummy-stack-extended:
  set S ⊆ Dummy ‘ UNIV ⟹ x ∉ Dummy ‘ UNIV ⟹ (S ≲ x # S') ⟷ S ≲ S'
  apply (auto simp add: extends-def)
  apply (case-tac S'')
  apply auto
  done

lemma[simp]: Arg x ∉ range Dummy  Upd x ∉ range Dummy  Alts e1 e2 ∉ range Dummy by
auto

lemma dummy-stack-balanced:
  assumes set S ⊆ Dummy ‘ UNIV
  assumes (Γ, e, S) ⇒* (Δ, z, S)
  obtains T where (Γ, e, S) ⇒^b*_T (Δ, z, S)
proof−
  from build-trace[OF assms(2)]
  obtain T where (Γ, e, S) ⇒*_T (Δ, z, S)..
  moreover
  hence ∀ c'∈set T. stack (Γ, e, S) ≲ stack c'
    by (rule conjunct1[OF traces-list-all])
       (auto elim: step.cases simp add: dummy-stack-extended[OF ‹set S ⊆ Dummy ‘ UNIV›])
  ultimately
  have (Γ, e, S) ⇒^b*_T (Δ, z, S)
    by (rule ballI) simp
  thus ?thesis by (rule that)
qed

end
```

3 Arity

3.1 Arity

theory *Arity*
imports *Launchbury.HOLCF−Join−Classes*
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* ≤ *x***.**

instance
apply *standard*
apply ((*transfer*, *auto*)+)
done
end

instance *Arity* :: *chfin*
proof
  fix *S* :: *nat* ⇒ *Arity*
  assume *chain S*
  have (*ARG-MIN Rep-Arity x*. *x* ∈ *range S*) ∈ *range S*
    by (*rule arg-min-natI*) *auto*
  then obtain *n* **where** *n*: *S n* = (*ARG-MIN Rep-Arity x*. *x* ∈ *range S*) **by** *auto*
  have *max-in-chain n S*
  proof(*rule max-in-chainI*)
    fix *j*
    assume *n* ≤ *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*) *arg-min-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** (λ *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 = (Λ *x*. *inc-Arity x*)

definition *pred* :: *Arity* → *Arity* **where**
pred = (Λ *x*. *pred-Arity x*)

lemma *inc-inj*[*simp*]: *inc*·*n* = *inc*·*n*′ ⟷ *n* = *n*′
by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *pred-inc*[*simp*]: *pred*·(*inc*·*n*) = *n*
by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *inc-below-inc*[*simp*]: *inc*·*a* ⊑ *inc*·*b* ⟷ *a* ⊑ *b*
by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *inc-below-below-pred*[*elim*]:
inc·*a* ⊑ *b* ⟹ *a* ⊑ *pred* · *b*
by (*simp add*: *inc-def pred-def*, *transfer*, *simp*)

lemma *Rep-Arity-inc*[*simp*]: *Rep-Arity* (*inc*·*a*′) = *Suc* (*Rep-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*]: $(x :: Arity_\perp) \sqcup up{\cdot}0 = up{\cdot}0$
by (*cases x*) *auto*
lemma *Arity-up-zero-join2*[*simp*]: $up{\cdot}0 \sqcup (x :: Arity_\perp) = up{\cdot}0$
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*]: $up{\cdot}0 \sqsubseteq (a :: Arity_\perp) \longleftrightarrow a = up{\cdot}0$
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

3.2 AEnv

theory *AEnv*
imports *Arity Launchbury.Vars Launchbury.Env*
begin

type-synonym $AEnv = var \Rightarrow Arity_\perp$

end

3.3 Arity-Nominal

theory *Arity−Nominal*
imports *Arity Launchbury.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

3.4 ArityStack

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

4 Eta-Expansion

4.1 EtaExpansion

theory *EtaExpansion*
imports *Launchbury.Terms Launchbury.Substitution*
begin

definition *fresh-var* :: *exp* ⇒ *var* **where**
fresh-var e = (*SOME v*. *v* ∉ *fv e*)

lemma *fresh-var-not-free*:
fresh-var e ∉ *fv e*
proof−
obtain *v* :: *var* **where** *atom v* ♯ *e* **by** (*rule obtain-fresh*)
hence *v* ∉ *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*) ♯ *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*]:
 π ∙ (*eta-expand n e*) = *eta-expand* (π ∙ *n*) (π ∙ *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 ⟹ *isLam* (*eta-expand n e*) **and** *n* > *0* ⟹ *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

4.2 EtaExpansionSafe

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 (Γ, *eta-expand n e*, *S*) $\Rightarrow^*$ (Γ, *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

4.3 TransformTools

theory *TransformTools*
imports *Launchbury.Nominal−HOLCF Launchbury.Terms Launchbury.Substitution Launchbury.Env*
begin

default-sort *type*

fun *lift-transform* :: (′*a*::*cont-pt* ⇒ *exp* ⇒ *exp*) ⇒ (′$a_\perp$ ⇒ *exp* ⇒ *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 ⊥ *e* = *e*
lift-transform t (*up*·*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*]: π · *lift-transform t a e* = *lift-transform* (π · *t*) (π · *a*) (π · *e*)
by (*cases a*) *simp-all*

lemma *lift-transform-fun-cong*[*fundef-cong*]:
(⋀ *a*. *t1 a e1* = *t2 a e1*) ⟹ *a1* = *a2* ⟹ *e1* = *e2* ⟹ *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 ⋀ *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-pt* ⇒ *exp* ⇒ *exp*) ⇒ (*var* ⇒ ′$a_\perp$) ⇒ *heap* ⇒ *heap*
where *map-transform t ae* = *map-ran* (λ *x e* . *lift-transform t* (*ae x*) *e*)

lemma *map-transform-eqvt*[*eqvt*]: π · *map-transform t ae* = *map-transform* (π · *t*) (π · *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 ⋀ *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*!: *subsetD*[*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*!: *subsetD*[*OF supp-lift-transform*])

lemma *fresh-transform*[*intro*]: *a* ♯ *e* ⟹ *a* ♯ *trans n e*
by (*auto simp add*: *fresh-def*) (*auto dest*!: *subsetD*[*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

4.4 ArityEtaExpansion

theory *ArityEtaExpansion*

imports *EtaExpansion Arity−Nominal TransformTools*
begin

lift-definition *Aeta-expand* :: *Arity* ⇒ *exp* ⇒ *exp* **is** *eta-expand*.

lemma *Aeta-expand-eqvt*[*eqvt*]: π · *Aeta-expand a e* = *Aeta-expand* (π · *a*) (π · *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

4.5 ArityEtaExpansionSafe

theory *ArityEtaExpansionSafe*
imports *EtaExpansionSafe AbstractStack ArityEtaExpansion*
begin

lemma *Aeta-expand-safe*:
  assumes *Astack S* ⊑ *a*

shows $(\Gamma,\ \mathit{Aeta\text{-}expand}\ a\ e,\ S) \Rightarrow^* (\Gamma,\ e,\ S)$
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* ≤ *Rep-Arity* (*Astack S*) **by** (*metis below-Arity.rep-eq*)
finally
show *?thesis*
by *transfer* (*rule eta-expansion-safe′*)
qed

end

5 Arity Analysis

5.1 ArityAnalysisSig

theory *ArityAnalysisSig*
imports *Launchbury.Terms AEnv Arity*−*Nominal Launchbury.Nominal*−*HOLCF Launchbury.Substitution*
begin

locale *ArityAnalysis* =
fixes *Aexp* :: *exp* ⇒ *Arity* → *AEnv*
begin
abbreviation *Aexp-syn* (‹$\mathcal{A}$_›)**where** $\mathcal{A}_a\ e \equiv \mathit{Aexp}\ e{\cdot}a$
abbreviation *Aexp-bot-syn* (‹$\mathcal{A}^\perp$_›)
where $\mathcal{A}^\perp{}_a\ e \equiv \mathit{fup}{\cdot}(\mathit{Aexp}\ e){\cdot}a$

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*:*subsetD*[*OF Aexp-edom*])

lemma *Aexp-fresh-bot*[*simp*]: **assumes** *atom v* ♯ *e* **shows** $\mathcal{A}_a\ e\ v = \perp$
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*]: $\pi \cdot Aheap = Aheap$

end

5.2 ArityAnalysisAbinds

theory *ArityAnalysisAbinds*
imports *ArityAnalysisSig*
begin

context *ArityAnalysis*
begin

5.2.1 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 $\sqcup$ *ABinds* (*delete* v Δ) **by** (*rule Abinds-reorder1*)
ultimately
show *?case* **by** *auto*
qed

lemma *Abinds-env-cong*: ($\bigwedge$ x. $x \in domA\ \Delta \Longrightarrow ae\ x = ae'\ x$) $\Longrightarrow$ *ABinds* $\Delta \cdot ae$ = *ABinds* $\Delta \cdot ae'$
by (*induct* Δ *rule*: *ABinds.induct*) *auto*

lemma *Abinds-env-restr-cong*: $ae\ f|`\ domA\ \Delta = ae'\ f|`\ domA\ \Delta \Longrightarrow$ *ABinds* $\Delta \cdot ae$ = *ABinds* $\Delta \cdot ae'$
by (*rule Abinds-env-cong*) (*metis env-restr-eqD*)

lemma *ABinds-env-restr*[*simp*]: *ABinds* $\Delta \cdot (ae\ f|`\ domA\ \Delta)$ = *ABinds* $\Delta \cdot ae$
by (*rule Abinds-env-restr-cong*) *simp*

lemma *Abinds-join-fresh*: $ae'\ `\ (domA\ \Delta) \subseteq \{\bot\} \Longrightarrow$ *ABinds* $\Delta \cdot (ae \sqcup ae')$ = (*ABinds* $\Delta \cdot ae$)
by (*rule Abinds-env-cong*) *auto*

lemma *ABinds-delete-bot*: $ae\ x = \bot \Longrightarrow$ *ABinds* (*delete* x Γ)$\cdot ae$ = *ABinds* $\Gamma \cdot ae$
by (*induction* Γ *rule*: *ABinds.induct*) (*auto simp add*: *delete-twist*)

lemma *ABinds-restr-fresh*:
assumes *atom* ` S $\sharp*$ Γ
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*, *opaque-lifting*) *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* Γ[*x*::*h*=*y*]·*ae* *f*|‘ *S* = *ABinds* Γ·(*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*: *subsetD*[*OF set-delete-subset*])
done

lemma *Abinds-append-disjoint*: *domA* Δ ∩ *domA* Γ = {} ⟹ *ABinds* (Δ @ Γ)·*ae* = *ABinds* Δ·*ae* ⊔ *ABinds* Γ·*ae*
proof (*induct* Δ *rule*: *ABinds.induct*)
case *1* **thus** *?case* **by** *simp*
next
case (*2 v e* Δ)
from *2*(*2*)
have *v* ∉ *domA* Γ **and** *domA* (*delete v* Δ) ∩ *domA* Γ = {} **by** *auto*
from *2*(*1*)[*OF this*(*2*)]
have *ABinds* (*delete v* Δ @ Γ)·*ae* = *ABinds* (*delete v* Δ)·*ae* ⊔ *ABinds* Γ·*ae*.
moreover
have *delete v* Γ = Γ **by** (*metis* ‹*v* ∉ *domA* Γ› *delete-not-domA*)
ultimately
show *ABinds* (((*v*, *e*) # Δ) @ Γ)·*ae* = *ABinds* ((*v*, *e*) # Δ)·*ae* ⊔ *ABinds* Γ·*ae*
by *auto*
qed

lemma *ABinds-restr-subset*: *S* ⊆ *S*′ ⟹ *ABinds* (*restrictA S* Γ)·*ae* ⊑ *ABinds* (*restrictA S*′ Γ)·*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*) Γ)·*ae* = *ABinds* Γ·*ae*
by (*induct* Γ *rule*: *ABinds.induct*) (*auto simp add*: *edom-def restr-delete-twist*)

lemma *ABinds-restrict-below*: *ABinds* (*restrictA S* Γ)·*ae* ⊑ *ABinds* Γ·*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* Γ)·*ae* ⊑ *ABinds* Γ·*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*]: π ∙ (*ArityAnalysis.ABind Aexp v e*) = *ArityAnalysis.ABind* (π ∙ *Aexp*) (π ∙ *v*) (π ∙ *e*)
apply (*rule cfun-eqvtI*)
unfolding *ArityAnalysis.ABind-eq*

by *perm-simp rule*

lemma *ABinds-eqvt*[*eqvt*]: $\pi \cdot (\mathit{ArityAnalysis.ABinds}\ \mathit{Aexp}\ \Gamma) = \mathit{ArityAnalysis.ABinds}\ (\pi \cdot \mathit{Aexp})\ (\pi \cdot \Gamma)$
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*]:
$[\![(\bigwedge e.\ e \in \mathit{snd}\ `\ \mathit{set}\ \mathit{heap2} \Longrightarrow \mathit{aexp1}\ e = \mathit{aexp2}\ e)\ ;\ \mathit{heap1} = \mathit{heap2}\ ]\!]$
$\Longrightarrow \mathit{ArityAnalysis.ABinds}\ \mathit{aexp1}\ \mathit{heap1} = \mathit{ArityAnalysis.ABinds}\ \mathit{aexp2}\ \mathit{heap2}$
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*) $\subseteq$ *snd* ` *set as* **by** (*rule dom-delete-subset*)
also have ... $\subseteq$ *snd* `*set* ((*v*, *e*) # *as*) **by** *auto*
also note *prems*(*3*)
finally
have ($\bigwedge$*e*. *e* $\in$ *snd* ` *set* (*delete v as*) $\Longrightarrow$ *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* $\sharp$ *e* $\Longrightarrow$ (*fup*·(*Aexp e*)·*a*) *v* = $\bot$
by (*cases a*) *auto*

lemma *edom-AnalBinds*: *edom* (*ABinds* Γ·*ae*) $\subseteq$ *fv* Γ
by (*induction* Γ *rule*: *ABinds.induct*)
(*auto simp del*: *fun-meet-simp dest*: *subsetD*[*OF fup-Aexp-edom*] *dest*: *subsetD*[*OF fv-delete-subset*])
end

end

5.3 ArityAnalysisSpec

theory *ArityAnalysisSpec*
imports *ArityAnalysisAbinds*
begin

locale *SubstArityAnalysis* = *EdomArityAnalysis* +

assumes *Aexp-subst-restr*: $x \notin S \Longrightarrow y \notin S \Longrightarrow (Aexp\ e[x::=y] \cdot a)\ f|`S = (Aexp\ e{\cdot}a)\ f|`S$

locale *ArityAnalysisSafe* = *SubstArityAnalysis* +
assumes *Aexp-Var*: $up \cdot n \sqsubseteq (Aexp\ (Var\ x){\cdot}n)\ x$
assumes *Aexp-App*: $Aexp\ e\ {\cdot}(inc{\cdot}n) \sqcup esing\ x \cdot (up{\cdot}0) \sqsubseteq Aexp\ (App\ e\ x) \cdot n$
assumes *Aexp-Lam*: $env\text{-}delete\ y\ (Aexp\ e\ {\cdot}(pred{\cdot}n)) \sqsubseteq Aexp\ (Lam\ [y].\ e) \cdot n$
assumes *Aexp-IfThenElse*: $Aexp\ scrut{\cdot}0 \sqcup Aexp\ e1{\cdot}a \sqcup Aexp\ e2{\cdot}a \sqsubseteq Aexp\ (scrut\ ?\ e1 : e2){\cdot}a$

locale *ArityAnalysisHeapSafe* = *ArityAnalysisSafe* + *ArityAnalysisHeapEqvt* +
assumes *edom-Aheap*: $edom\ (Aheap\ \Gamma\ e{\cdot}\ a) \subseteq domA\ \Gamma$
assumes *Aheap-subst*: $x \notin domA\ \Gamma \Longrightarrow y \notin domA\ \Gamma \Longrightarrow Aheap\ \Gamma[x::h=y]\ e[x ::=y]\ = Aheap\ \Gamma\ e$

locale *ArityAnalysisLetSafe* = *ArityAnalysisHeapSafe* +
assumes *Aexp-Let*: $ABinds\ \Gamma{\cdot}(Aheap\ \Gamma\ e{\cdot}a) \sqcup Aexp\ e{\cdot}a \sqsubseteq Aheap\ \Gamma\ e{\cdot}a \sqcup Aexp\ (Let\ \Gamma\ e){\cdot}a$

locale *ArityAnalysisLetSafeNoCard* = *ArityAnalysisLetSafe* +
assumes *Aheap-heap3*: $x \in thunks\ \Gamma \Longrightarrow (Aheap\ \Gamma\ e{\cdot}a)\ x = up{\cdot}0$

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 $x' = x \vee x' = y \vee x' \in (-\{x,y\})$ **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])·a ⊑ env-delete y ((Aexp e)·a) ⊔ esing x·(up·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 · (up·n) ⊑ Aexp (Var x) · n
 by (simp add: Aexp-Var)

lemma fup-Aexp-Var: esing x · n ⊑ fup·(Aexp (Var x))·n
 by (cases n) (simp-all add: Aexp-Var)
end


context ArityAnalysisLetSafe
begin
 lemma Aheap-nonrec:
  assumes nonrec Δ
  shows Aexp e·a f|` domA Δ ⊑ Aheap Δ e·a
 proof−
   have ABinds Δ·(Aheap Δ e·a) ⊔ Aexp e·a ⊑ Aheap Δ e·a ⊔ Aexp (Let Δ e)·a by (rule
Aexp-Let)
  note env-restr-mono[where S = domA Δ, OF this]
  moreover
  from assms
  have ABinds Δ·(Aheap Δ e·a) f|` domA Δ = ⊥
   by (rule nonrecE) (auto simp add: fv-def fresh-def dest!: subsetD[OF fup-Aexp-edom])
  moreover
  have Aheap Δ e·a f|` domA Δ = Aheap Δ e·a
   by (rule env-restr-useless[OF edom-Aheap])
  moreover
  have (Aexp (Let Δ e)·a) f|` domA Δ = ⊥
   by (auto dest!: subsetD[OF Aexp-edom])
  ultimately
  show Aexp e·a f|` domA Δ ⊑ Aheap Δ e·a
   by (simp add: env-restr-join)
 qed
end


end
```

5.4 TrivialArityAnal

theory *TrivialArityAnal*
imports *ArityAnalysisSpec Launchbury.Env−Nominal*
begin

definition *Trivial-Aexp* :: *exp* ⇒ *Arity* → *AEnv*
  where *Trivial-Aexp e* = (Λ *n*. (λ *x*. *up*·*0*) *f*|‘ *fv e*)

lemma *Trivial-Aexp-simp*: *Trivial-Aexp e* · *n* = (λ *x*. *up*·*0*) *f*|‘ *fv e*
  unfolding *Trivial-Aexp-def* **by** *simp*

lemma *edom-Trivial-Aexp*[*simp*]: *edom* (*Trivial-Aexp e* · *n*) = *fv e*
  by (*auto simp add*: *edom-def env-restr-def Trivial-Aexp-def*)

lemma *Trivial-Aexp-eq*[*iff*]: *Trivial-Aexp e* · *n* = *Trivial-Aexp e′* · *n′* ⟷ *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* ⊑ *Trivial-Aexp e* · *n*) ⟷ *edom ae* ⊆ *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*·*n* ⊑ (*Trivial-Aexp* (*Var x*)·*n*) *x*
    by (*simp add*: *Trivial-Aexp-simp*)
next
  fix *e x n*
  show *Trivial-Aexp e*·(*inc*·*n*) ⊔ *esing x*·(*up*·*0*) ⊑ *Trivial-Aexp* (*App e x*)·*n*
    by (*auto intro*: *fun-belowI simp add*: *Trivial-Aexp-def env-restr-def*)
next
  fix *y e n*
  show *env-delete y* (*Trivial-Aexp e*·(*pred*·*n*)) ⊑ *Trivial-Aexp* (*Lam* [*y*]. *e*)·*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* ∉ *S* **and** *y* ∉ *S*
  thus *Trivial-Aexp e*[*x*::=*y*]·*a f*|‘ *S* = *Trivial-Aexp e*·*a f*|‘ *S*
    by (*auto simp add*: *Trivial-Aexp-simp fv-subst-eq intro*!: *arg-cong*[**where** *f* = λ *S*. *env-restr*

S e **for** *e*])
next
fix *scrut e1 a e2*
show *Trivial-Aexp scrut·0 ⊔ Trivial-Aexp e1·a ⊔ Trivial-Aexp e2·a ⊑ Trivial-Aexp* (*scrut ? e1 : e2*)·*a*
by (*auto intro*: *env-restr-mono2 simp add*: *Trivial-Aexp-simp join-below-iff*)
qed

definition *Trivial-Aheap* :: *heap* ⇒ *exp* ⇒ *Arity* → *AEnv* **where**
Trivial-Aheap Γ *e* = (Λ *a*. (λ *x*. *up·0*) *f*|‘ *domA* Γ)

lemma *Trivial-Aheap-eqvt*[*eqvt*]: π ∙ (*Trivial-Aheap* Γ *e*) = *Trivial-Aheap* (π ∙ Γ) (π ∙ *e*)
unfolding *Trivial-Aheap-def*
apply *perm-simp*
apply (*simp add*: *Abs-cfun-eqvt*)
done

lemma *Trivial-Aheap-simp*: *Trivial-Aheap* Γ *e*· *a* = (λ *x*. *up·0*) *f*|‘ *domA* Γ
unfolding *Trivial-Aheap-def* **by** *simp*

lemma *Trivial-fup-Aexp-below-fv*: *fup·*(*Trivial-Aexp e*)·*a* ⊑ (λ *x* . *up·0*) *f*|‘ *fv e*
by (*cases a*)(*auto simp add*: *Trivial-Aexp-simp*)

lemma *Trivial-Abinds-below-fv*: *ABinds* Γ·*ae* ⊑ (λ *x* . *up·0*) *f*|‘ *fv* Γ
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*: *subsetD*[*OF fv-delete-subset*] *simp del*: *fun-meet-simp*)

interpretation *ArityAnalysisLetSafe Trivial-Aexp Trivial-Aheap*
proof
fix π
show π ∙ *Trivial-Aheap* = *Trivial-Aheap* **by** *perm-simp rule*
next
fix Γ *e ae* **show** *edom* (*Trivial-Aheap* Γ *e*·*ae*) ⊆ *domA* Γ
by (*simp add*: *Trivial-Aheap-simp*)
next
fix Γ :: *heap* **and** *e* **and** *a*
show *ABinds* Γ·(*Trivial-Aheap* Γ *e*·*a*) ⊔ *Trivial-Aexp e*·*a* ⊑ *Trivial-Aheap* Γ *e*·*a* ⊔ *Trivial-Aexp* (*Terms.Let* Γ *e*)·*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* ∉ *domA* Γ **and** *y* ∉ *domA* Γ
thus *Trivial-Aheap* Γ[*x*::*h*=*y*] *e*[*x*::=*y*] = *Trivial-Aheap* Γ *e*
by (*auto intro*: *cfun-eqI simp add*: *Trivial-Aheap-simp*)
qed

end

5.5 ArityAnalysisStack

theory *ArityAnalysisStack*
imports *SestoftConf ArityAnalysisSig*
begin

context *ArityAnalysis*
begin
fun *AEstack* :: *Arity list* ⇒ *stack* ⇒ *AEnv*
where
AEstack - [] = ⊥
| *AEstack* (*a*#*as*) (*Alts e1 e2* # *S*) = *Aexp e1*·*a* ⊔ *Aexp e2*·*a* ⊔ *AEstack as S*
| *AEstack as* (*Upd x* # *S*) = *esing x*·(*up*·*0*) ⊔ *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*!: *subsetD*[*OF Aexp-edom*])
end

end

5.6 ArityAnalysisFix

theory *ArityAnalysisFix*
imports *ArityAnalysisSig ArityAnalysisAbinds*
begin

context *ArityAnalysis*
begin

definition *Afix* :: *heap* ⇒ (*AEnv* → *AEnv*)
where *Afix* Γ = (Λ *ae*. (μ *ae*′. *ABinds* Γ · *ae*′ ⊔ *ae*))

lemma *Afix-eq*: *Afix* Γ·*ae* = (μ *ae*′. (*ABinds* Γ·*ae*′) ⊔ *ae*)
unfolding *Afix-def* **by** *simp*

lemma *Afix-strict*[*simp*]: *Afix* Γ·⊥ = ⊥
unfolding *Afix-eq*
by (*rule fix-eqI*) *auto*

lemma *Afix-least-below*: *ABinds* Γ · *ae*′ ⊑ *ae*′ ⟹ *ae* ⊑ *ae*′ ⟹ *Afix* Γ · *ae* ⊑ *ae*′
unfolding *Afix-eq*
by (*auto intro*: *fix-least-below*)

lemma *Afix-unroll*: *Afix* Γ·*ae* = *ABinds* Γ · (*Afix* Γ·*ae*) ⊔ *ae*
unfolding *Afix-eq*
apply (*subst fix-eq*)
by *simp*

lemma *Abinds-below-Afix*: *ABinds* Δ ⊑ *Afix* Δ
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* ⊑ *Afix* Γ · *ae*
by (*subst Afix-unroll*) *simp*

lemma *Abinds-Afix-below*[*simp*]: *ABinds* Γ·(*Afix* Γ·*ae*) ⊑ *Afix* Γ·*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* Γ·(*aeL f*|‘ (− *S*)) *f*|‘ (− *S*) ⊔ *ae f*|‘ (− *S*) **by** (*rule arg-cong*[*OF ABinds-restr-fresh*[*OF assms*]])
also have . . . = *ABinds* Γ·(*aeR f*|‘ (− *S*)) *f*|‘ (− *S*) ⊔ *ae f*|‘ (− *S*) **unfolding** *prems* **..**
also have . . . = *ABinds* Γ·*aeR f*|‘ (− *S*) ⊔ *ae f*|‘ (− *S*) **by** (*rule arg-cong*[*OF ABinds-restr-fresh*[*OF assms*, *symmetric*]])
also have . . . = (*ABinds* Γ·*aeR* ⊔ *ae f*|‘ (− *S*)) *f*|‘ (− *S*) **by** (*simp add*: *env-restr-join*)
finally show *?case* **by** *simp*
qed

lemma *Afix-restr*:
assumes *domA* Γ ⊆ *S*
shows *Afix* Γ·*ae f*|‘ *S* = *Afix* Γ·(*ae f*|‘ *S*) *f*|‘ *S*
unfolding *Afix-eq*
apply (*rule parallel-fix-ind*[**where** *P* = λ *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 ⋀ *x′ e a*. (*x′*,*e*) ∈ *set* Γ ⟹ *Aexp e*[*x*::=*y*]·*a f*|‘ *S* = *Aexp e*·*a f*|‘ *S*
assumes *x* ∉ *S*
assumes *y* ∉ *S*
assumes *domA* Γ ⊆ *S*
shows *Afix* Γ[*x*::*h*=*y*]·*ae f*|‘ *S* = *Afix* Γ·(*ae f*|‘ *S*) *f*|‘ *S*
unfolding *Afix-eq*
apply (*rule parallel-fix-ind*[**where** *P* = λ *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 $ABinds\ \Gamma[y::h=x]\cdot aeL \sqsubseteq ABinds\ \Gamma[y::h=x]\cdot(aeR\ (y := \bot,\ x := up\cdot 0))$ **by** (*rule monofun-cfun-arg*)
also have ... $\sqsubseteq (ABinds\ \Gamma\cdot aeR)(y := \bot,\ x := up\cdot 0)$
using *assms*
proof (*induction rule*: *ABinds.induct*, *goal-cases*)
case *1*
thus *?case* **by** *simp*
next
case *prems*: (*2 v e* Γ)
have $\bigwedge n.\ Aexp\ e[y::=x]\cdot n \sqsubseteq (Aexp\ e\cdot n)(y := \bot,\ x := up\cdot 0)$ **using** *prems*(*2*)[**where** $v = v$] **by** *auto*
hence *IH1*: $\bigwedge\ n.\ fup\cdot(Aexp\ e[y::=x])\cdot n \sqsubseteq (fup\cdot(Aexp\ e)\cdot n)(y := \bot,\ x := up\cdot 0)$ **by** (*case-tac n*) *auto*

have $ABinds\ (delete\ v\ \Gamma)[y::h=x]\cdot(aeR(y := \bot,\ x := up\cdot 0)) \sqsubseteq (ABinds\ (delete\ v\ \Gamma)\cdot aeR)(y := \bot,\ x := up\cdot 0)$
apply (*rule prems*) **using** *prems*(*2*,*3*,*4*) **by** *fastforce*+
hence *IH2*: $ABinds\ (delete\ v\ \Gamma[y::h=x])\cdot(aeR(y := \bot,\ x := up\cdot 0)) \sqsubseteq (ABinds\ (delete\ v\ \Gamma)\cdot aeR)(y := \bot,\ x := up\cdot 0)$
unfolding *subst-heap-delete*.

have [*simp*]: $(aeR(y := \bot,\ x := up\cdot 0))\ v = aeR\ v$ **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 $ABinds\ \Gamma[y::h=x]\cdot aeL \sqsubseteq (ABinds\ \Gamma\cdot aeR)(y := \bot,\ x := up\cdot 0)$

by *this simp*
thus *?case*
by (*auto simp add*: *join-below-iff elim*: *below-trans*)
qed

end

lemma *Afix-eqvt*[*eqvt*]: $\pi \cdot (\mathit{ArityAnalysis.Afix}\ \mathit{Aexp}\ \Gamma) = \mathit{ArityAnalysis.Afix}\ (\pi \cdot \mathit{Aexp})\ (\pi \cdot \Gamma)$
unfolding *ArityAnalysis.Afix-def*
by *perm-simp* (*simp add*: *Abs-cfun-eqvt*)

lemma *Afix-cong*[*fundef-cong*]:
$[\![(\bigwedge e.\ e \in \mathit{snd}\ `\ \mathit{set}\ \mathit{heap2} \Longrightarrow \mathit{aexp1}\ e = \mathit{aexp2}\ e);\ \mathit{heap1} = \mathit{heap2}\ ]\!]$
$\Longrightarrow \mathit{ArityAnalysis.Afix}\ \mathit{aexp1}\ \mathit{heap1} = \mathit{ArityAnalysis.Afix}\ \mathit{aexp2}\ \mathit{heap2}$
unfolding *ArityAnalysis.Afix-def* **by** (*metis Abinds-cong*)

context *EdomArityAnalysis*
begin

lemma *Afix-edom*: $\mathit{edom}\ (\mathit{Afix}\ \Gamma \cdot \mathit{ae}) \subseteq \mathit{fv}\ \Gamma \cup \mathit{edom}\ \mathit{ae}$
unfolding *Afix-eq*
by (*rule fix-ind*[**where** $P = \lambda\ \mathit{ae}'\ .\ \mathit{edom}\ \mathit{ae}' \subseteq \mathit{fv}\ \Gamma \cup \mathit{edom}\ \mathit{ae}$])
(*auto dest*: *subsetD*[*OF edom-AnalBinds*])

lemma *ABinds-lookup-fresh*:
$\mathit{atom}\ v \sharp \Gamma \Longrightarrow (\mathit{ABinds}\ \Gamma{\cdot}\mathit{ae})\ v = \bot$
by (*induct* Γ *rule*: *ABinds.induct*) (*auto simp add*: *fresh-Cons fresh-Pair fup-Aexp-lookup-fresh fresh-delete*)

lemma *Afix-lookup-fresh*:
assumes $\mathit{atom}\ v \sharp \Gamma$
shows $(\mathit{Afix}\ \Gamma{\cdot}\mathit{ae})\ v = \mathit{ae}\ v$
apply (*rule below-antisym*)
apply (*subst Afix-eq*)
apply (*rule fix-ind*[**where** $P = \lambda\ \mathit{ae}'.\ \mathit{ae}'\ v \sqsubseteq \mathit{ae}\ v$])
apply (*auto simp add*: *ABinds-lookup-fresh*[*OF assms*] *fun-belowD*[*OF Afix-above-arg*])
done

lemma *Afix-comp2join-fresh*:
$\mathit{atom}\ `\ (\mathit{domA}\ \Delta) \sharp* \Gamma \Longrightarrow \mathit{ABinds}\ \Delta{\cdot}(\mathit{Afix}\ \Gamma{\cdot}\mathit{ae}) = \mathit{ABinds}\ \Delta{\cdot}\mathit{ae}$
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 $\mathit{atom}\ v \sharp \Gamma$ **and** $\mathit{atom}\ `\ \mathit{domA}\ (\mathit{delete}\ v\ \Delta) \sharp* \Gamma$

```
    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 ♯ Γ›])
qed

lemma Afix-append-fresh:
  assumes atom ‘ domA Δ ♯* Γ
  shows Afix (Δ @ Γ)·ae = Afix Γ·(Afix Δ·ae)
proof (rule below-antisym)
  show ∗: Afix (Δ @ Γ)·ae ⊑ Afix Γ·(Afix Δ·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 Γ·(Afix Δ·ae) ⊑ Afix (Δ @ Γ)·ae
  proof (rule Afix-least-below)
    show ABinds Γ·(Afix (Δ @ Γ)·ae) ⊑ Afix (Δ @ Γ)·ae
      apply (rule below-trans[OF - Abinds-Afix-below])
      apply (subst Abinds-append-disjoint[OF fresh-distinct[OF assms]])
      apply simp
      done
    have ABinds Δ·(Afix (Δ @ Γ)·ae) ⊑ Afix (Δ @ Γ)·ae
      apply (rule below-trans[OF - Abinds-Afix-below])
      apply (subst Abinds-append-disjoint[OF fresh-distinct[OF assms]])
      apply simp
      done
    thus Afix Δ·ae ⊑ Afix (Δ @ Γ)·ae
      apply (rule Afix-least-below)
      apply (rule Afix-above-arg)
      done
  qed
qed


lemma Afix-e-to-heap:
   Afix (delete x Γ)·(fup·(Aexp e)·n ⊔ ae) ⊑ Afix ((x, e) # delete x Γ)·(esing x·n ⊔ 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* Γ)·(*Aexp e*·*n*) ⊑ *Afix* ((*x*, *e*) # *delete x* Γ)·(*esing x*·(*up*·*n*))
using *Afix-e-to-heap*[**where** *ae* = ⊥ **and** *n* = *up*·*n*] **by** *simp*

end

end

5.7 ArityAnalysisFixProps

theory *ArityAnalysisFixProps*
imports *ArityAnalysisFix ArityAnalysisSpec*
begin

context *SubstArityAnalysis*
begin

lemma *Afix-restr-subst*:
assumes *x* ∉ *S*
assumes *y* ∉ *S*
assumes *domA* Γ ⊆ *S*
shows *Afix* Γ[*x*::*h*=*y*]·*ae f*|‘ *S* = *Afix* Γ·(*ae f*|‘ *S*) *f*|‘ *S*
by (*rule Afix-restr-subst'*[*OF Aexp-subst-restr*[*OF assms*(*1*,*2*)] *assms*])
end

end

6 Arity Transformation

6.1 AbstractTransform

theory *AbstractTransform*
imports *Launchbury.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*$_\perp$
fixes *PropIfScrut* :: *'a ⇒ 'a*
assumes *PropApp-eqvt*: $\pi \cdot$ *PropApp* $\equiv$ *PropApp*
assumes *PropLam-eqvt*: $\pi \cdot$ *PropLam* $\equiv$ *PropLam*
assumes *AnalLet-eqvt*: $\pi \cdot$ *AnalLet* $\equiv$ *AnalLet*
assumes *PropLetBody-eqvt*: $\pi \cdot$ *PropLetBody* $\equiv$ *PropLetBody*
assumes *PropLetHeap-eqvt*: $\pi \cdot$ *PropLetHeap* $\equiv$ *PropLetHeap*
assumes *PropIfScrut-eqvt*: $\pi \cdot$ *PropIfScrut* $\equiv$ *PropIfScrut*

locale *AbstractAnalPropSubst* = *AbstractAnalProp* +
assumes *AnalLet-subst*: $x \notin$ *domA* $\Gamma \Longrightarrow y \notin$ *domA* $\Gamma \Longrightarrow$ *AnalLet* $(\Gamma[x::h=y])$ $(e[x::=y])$ *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*: $\pi \cdot$ *TransVar* = *TransVar*
assumes *TransApp-eqvt*: $\pi \cdot$ *TransApp* = *TransApp*
assumes *TransLam-eqvt*: $\pi \cdot$ *TransLam* = *TransLam*
assumes *TransLet-eqvt*: $\pi \cdot$ *TransLet* = *TransLet*
assumes *SuppTransLam*: *supp* (*TransLam a v e*) $\subseteq$ *supp e* − *supp v*
assumes *SuppTransLet*: *supp* (*TransLet b* Γ *e*) $\subseteq$ *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!: subsetD[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 (π ∙ x) (transform-sumC (PropLam a, π ∙ e))
       = TransLam a (π ∙ x) (transform-sumC (π ∙  (PropLam a, e)))
      by perm-simp rule
    also have ... = TransLam a (π ∙ x) (π ∙ transform-sumC (PropLam a, e))
      unfolding eqvt-at-apply'[OF prems(1)] ..
    also have ... = π ∙ (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 (π ∙ x) (transform-sumC (PropLam a, π ∙ 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 $\bigwedge x\ e\ a.\ (x, e) \in set\ as \Longrightarrow supp\ (transform\text{-}sumC\ (a, e)) \subseteq supp\ e$
by (*auto simp add*: *exp-assn.fsupp supp-Pair pure-supp*)
hence *supp-map*: $\bigwedge ae.\ supp\ (map\text{-}transform\ (\lambda x0\ x1.\ transform\text{-}sumC\ (x0, x1))\ ae\ as) \subseteq supp\ as$
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* (λ*x0 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!: *subsetD*[*OF supp-eqvt-at*[*OF prems*(*2*)], *rotated*] *subsetD*[*OF SuppTransLet*] *subsetD*[*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* (λ*x0 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* (λ*x0 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* (λ*x0 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* (λ*x0 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* (λ*x0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet* (π • *as*) (π • *body*) *a*)) (π • *as*)
= *map-transform* (λ*x 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* = (− π

$\cdot$ *a*) **for** *a*], **where** $p = \pi$, *symmetric*])
done
moreover
have ($\pi \cdot$ *transform-sumC*) (*PropLetBody* (*AnalLet* ($\pi \cdot$ *as*) ($\pi \cdot$ *body*) *a*), $\pi \cdot$ *body*) = *transform-sumC* (*PropLetBody* (*AnalLet* ($\pi \cdot$ *as*) ($\pi \cdot$ *body*) *a*), $\pi \cdot$ *body*)
using *eqvt-at-apply''*[*OF prems*(*2*), **where** $p = \pi$] **by** *perm-simp*
ultimately
show *TransLet* (*AnalLet* ($\pi \cdot$ *as*) ($\pi \cdot$ *body*) *a*) (*map-transform* (λ*x0 x1*. *transform-sumC* (*x0*, *x1*)) (*PropLetHeap* (*AnalLet* ($\pi \cdot$ *as*) ($\pi \cdot$ *body*) *a*)) ($\pi \cdot$ *as*)) (*transform-sumC* (*PropLetBody* (*AnalLet* ($\pi \cdot$ *as*) ($\pi \cdot$ *body*) *a*), $\pi \cdot$ *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*) $\subseteq$ *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*) $\subseteq$ *fv e*
unfolding *fv-def* **by** (*auto dest*: *subsetD*[*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* $\sharp$ (*x*,*y*) $\Longrightarrow$ (*TransLam a v e*)[*x* ::= *y*] = (*TransLam a v*[*x* ::*v*= *y*] *e*[*x* ::= *y*])
assumes *TransLet-subst*: *atom* ' *domA* Γ $\sharp*$ (*x*,*y*) $\Longrightarrow$ (*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* Δ *y* $\notin$ *domA* Δ **by** (*auto simp add*: *fresh-star-def fresh-at-base*)
hence *AnalLet* Δ[*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* ⇒ *'a*::*pure-cont-pt*
constrains *PropLetHeap* :: *'b*::*cont-pt* ⇒ *var* ⇒ *'a*$_\perp$
constrains *trans* :: *'c*::*cont-pt* ⇒ *exp* ⇒ *exp*
fixes *PropLetHeapTrans* :: *'b*⇒ *var* ⇒ *'c*$_\perp$
assumes *PropLetHeapTrans-eqvt*: *π* ∙ *PropLetHeapTrans* = *PropLetHeapTrans*
assumes *TransBound-eqvt*: *π* ∙ *trans* = *trans*
begin
sublocale *AbstractTransform PropApp PropLam AnalLet PropLetBody PropLetHeap PropIf-Scrut*
(*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*: *subsetD*[*OF supp-map-transform*])[*1*]
done
qed

lemma *isLam-transform*[*simp*]:
isLam (*transform a e*) ⟷ *isLam e*
by (*induction e rule*:*isLam.induct*) *auto*

lemma *isVal-transform*[*simp*]:
isVal (*transform a e*) ⟷ *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

6.2 ArityTransform

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* (‹$\mathcal{T}$_›) **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

7 Arity Analysis Safety (without Cardinality)

7.1 ArityConsistent

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* ∈ *domA* Γ **by** (*metis domI dom-map-of-conv-domA*)

from *assms*(*3*)
have [*simp*]: *x* ∈ *edom ae* **by** (*auto simp add*: *edom-def*)

have *x* ∈ *domA* Γ **by** (*metis assms*(*2*) *domI dom-map-of-conv-domA*)
hence [*simp*]: *insert x* (*domA* Γ − {*x*} ∪ *upds S*) = (*domA* Γ ∪ *upds S*)
by *auto*

from *Abinds-reorder1*[*OF* ‹*map-of* Γ *x* = *Some e*›] ‹*ae x* = *up·0*›
have *ABinds* (*delete x* Γ)·*ae* ⊔ *Aexp e·0* = *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* (*delete x* Γ))·*ae* ⊔ *Aexp e·0* ⊔ *AEstack as S*) *f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *ae* **by** *simp*
then
show *?thesis*
using ‹*ae x* = *up·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* Γ *x* = *Some e*
assumes [*simp*]: *ae x* = *up·u*
assumes *heap-upds-ok* (Γ, *S*)
shows *a-consistent* (*env-delete x ae*, *u*, *as*) (*delete x* Γ, *e*, *S*)
proof−
from *assms*(*2*)
have [*simp*]: *x* ∈ *domA* Γ **by** (*metis domI dom-map-of-conv-domA*)

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*

from ‹*heap-upds-ok* (Γ, *S*)›
have *x* ∉ *upds S* **by** (*auto simp add*: *a-consistent.simps elim*!: *heap-upds-okE*)
hence [*simp*]: (− {*x*} ∩ (*domA* Γ ∪ *upds S*)) = (*domA* Γ − {*x*} ∪ *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* Γ *x* = *Some e*›] ‹*ae x* = *up*·*u*›
have *ABinds* (*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*) $\sqsubseteq$ *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*) $\sqsubseteq$ *ae* **by** *simp*
hence ((*ABinds* (*delete x* Γ))·(*env-delete x ae*) ⊔ *Aexp e*·*u* ⊔ *AEstack as S*) *f*|‘ (*domA* Γ ∪ *upds S*) $\sqsubseteq$ *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*)) $\sqsubseteq$ *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*)) $\sqsubseteq$ *env-delete x ae*
by (*simp add*: *env-delete-restr*)
then
show *?thesis*
using ‹*ae x* = *up*·*u*› ‹*Astack S* $\sqsubseteq$ *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*) $\sqsubseteq$ *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*)·*a*) *x* $\sqsubseteq$ *up*·*u* **by** *simp*
from *below-trans*[*OF Aexp-Var this*]
have *a* $\sqsubseteq$ *u* **by** *simp*

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* 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 $(\ldots \sqcup AEstack\ as\ S)\ f|`(domA\ \Gamma \cup upds\ S) \sqsubseteq ae$
using *assms(1)* **by** (*auto simp add*: *join-below-iff env-restr-join a-consistent.simps*)
ultimately
have $((ABinds\ ((x,e)\ \#\ delete\ x\ \Gamma))\cdot ae \sqcup Aexp\ e\cdot u \sqcup AEstack\ as\ S)\ f|`(domA\ \Gamma \cup upds\ S) \sqsubseteq ae$ **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*·*0a* = *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* Δ *e*, *S*)
assumes *atom* ‘ *domA* Δ ♯∗ Γ
assumes *atom* ‘ *domA* Δ ♯∗ *S*
assumes *edom ae* ∩ *domA* Δ = {}
shows *a-consistent* (*Aheap* Δ *e*·*a* ⊔ *ae*, *a*, *as*) (Δ @ Γ, *e*, *S*)
proof−

First some boring stuff about scope:

have [*simp*]: ⋀ *S*. *S* ⊆ *domA* Δ ⟹ *ae f*|‘ *S* = ⊥ **using** *assms(4)* **by** *auto*
have [*simp*]: *ABinds* Δ·(*Aheap* Δ *e*·*a* ⊔ *ae*) = *ABinds* Δ·(*Aheap* Δ *e*·*a*)
by (*rule Abinds-env-restr-cong*) (*simp add*: *env-restr-join*)

have [*simp*]: *Aheap* Δ *e*·*a f*|‘ *domA* Γ = ⊥
using *fresh-distinct*[*OF assms(2)*]
by (*auto intro*: *env-restr-empty dest*!: *subsetD*[*OF edom-Aheap*])

have [*simp*]: *ABinds* Γ·(*Aheap* Δ *e*·*a* ⊔ *ae*) = *ABinds* Γ·*ae*
by (*rule Abinds-env-restr-cong*) (*simp add*: *env-restr-join*)

have [*simp*]: *ABinds* Γ·*ae f*|‘ (*domA* Δ ∪ *domA* Γ ∪ *upds S*) = *ABinds* Γ·*ae f*|‘ (*domA* Γ ∪ *upds S*)
using *fresh-distinct-fv*[*OF assms(2)*]
by (*auto intro*: *env-restr-cong dest*!: *subsetD*[*OF edom-AnalBinds*])

have [*simp*]: *AEstack as S f*|‘ (*domA* Δ ∪ *domA* Γ ∪ *upds S*) = *AEstack as S f*|‘ (*domA* Γ ∪ *upds S*)
using *fresh-distinct-fv*[*OF assms(3)*]
by (*auto intro*: *env-restr-cong dest*!: *subsetD*[*OF edom-AEstack*])

have [*simp*]: *Aexp* (*Let* Δ *e*)·*a f*|‘ (*domA* Δ ∪ *domA* Γ ∪ *upds S*) = *Aexp* (*Terms.Let* Δ *e*)·*a f*|‘ (*domA* Γ ∪ *upds S*)
by (*rule env-restr-cong*) (*auto dest*!: *subsetD*[*OF Aexp-edom*])

have [*simp*]: *Aheap* Δ *e*·*a f*|‘ (*domA* Δ ∪ *domA* Γ ∪ *upds S*) = *Aheap* Δ *e*·*a*
by (*rule env-restr-useless*) (*auto dest*!: *subsetD*[*OF edom-Aheap*])

have ((*ABinds* Γ)·*ae* ⊔ *AEstack as S*) *f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *ae* **using** *assms*(*1*) **by** (*auto simp add*: *a-consistent.simps join-below-iff env-restr-join*)
moreover
have *Aexp* (*Let* Δ *e*)·*a f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *ae* **using** *assms*(*1*) **by** (*auto simp add*: *a-consistent.simps join-below-iff env-restr-join*)
moreover
have *ABinds* Δ·(*Aheap* Δ *e*·*a*) ⊔ *Aexp e*·*a* ⊑ *Aheap* Δ *e*·*a* ⊔ *Aexp* (*Let* Δ *e*)·*a* **by** (*rule Aexp-Let*)
ultimately
have (*ABinds* (Δ @ Γ)·(*Aheap* Δ *e*·*a* ⊔ *ae*) ⊔ *Aexp e*·*a* ⊔ *AEstack as S*) *f*|‘ (*domA* (Δ@Γ) ∪ *upds S*) ⊑ *Aheap* Δ *e*·*a* ⊔ *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* Δ ∩ *upds S* = {} **by** (*auto dest*!: *subsetD*[*OF ups-fv-subset*])
ultimately
show *?thesis* **using** *assms*(*1*)
by (*auto simp add*: *a-consistent.simps dest*!: *subsetD*[*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*)·*a f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *ae* **by** (*auto simp add*: *a-consistent.simps env-restr-join join-below-iff*)
hence (*Aexp scrut*·*0* ⊔ *Aexp e1*·*a* ⊔ *Aexp e2*·*a*) *f*|‘ (*domA* Γ ∪ *upds S*) ⊑ *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*) ⟹ *a-consistent* (⊥, *0*, []) ([], *e*, [])
by (*auto simp add*: *edom-empty-iff-bot a-consistent.simps dest*!: *subsetD*[*OF Aexp-edom*])

end

end

7.2 ArityTransformSafe

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*!: *subsetD*[*OF supp-map-transform*] *subsetD*[*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* ♯∗ *S* ⟹ *a* ♯∗ *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*)
⟹ (⋀ *x*. *x* ∈ *thunks* Γ ⟹ *ae x* = *up*·*0*)
⟹ *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* (⊥, *0*, []) ([], *e*, [])
by (*auto simp add*: *edom-empty-iff-bot closed-a-consistent*[*OF assms*])

lemma *arity-tranform-safe*:
fixes *c c*′
assumes *c* ⇒* *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* ⇒* *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 subsetD*[*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 var2 ‹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› var2
    by (auto intro!: step.intros simp add: map-transform-Cons)
  ultimately show ?case by (blast del: consistentI consistentE)
 next
  case (let1 Δ Γ e S)
  let ?ae = Aheap Δ e·a

  have domA Δ ∩ upds S = {} using fresh-distinct-fv[OF let1(2)] by (auto dest: subsetD[OF
ups-fv-subset])
   hence *: ⋀ x. x ∈ upds S ⟹ x ∉ edom ?ae by (auto simp add:  dest!: subsetD[OF
edom-Aheap])
  have restr-stack-simp2: restr-stack (edom (?ae ⊔ ae)) S = restr-stack (edom ae) S
    by (auto intro: restr-stack-cong dest!: *)

  have edom ae ⊆ domA Γ ∪ upds S using let1 by (auto dest!: a-consistent-edom-subsetD)
  from subsetD[OF this] fresh-distinct[OF let1(1)] fresh-distinct-fv[OF let1(2)]
  have edom ae ∩ domA Δ = {} by (auto dest: subsetD[OF ups-fv-subset])

  {
  { fix x e'
    assume x ∈ thunks Γ
    with let1
    have (?ae ⊔ ae) x = up·0 by auto
  }
  moreover
  { fix x e'
    assume x ∈ thunks Δ
    hence (?ae ⊔ ae) x = up·0 by (auto simp add: Aheap-heap3)
  }
  moreover

  have a-consistent (ae, a, as) (Γ, Let Δ e, S)
    using let1 by auto
  hence a-consistent (?ae ⊔ ae, a, as) (Δ @ Γ, e, S)
    using let1(1,2) ‹edom ae ∩ domA Δ = {}›
    by (auto intro!:  a-consistent-let simp del: join-comm)
  ultimately
  have consistent (?ae ⊔ ae, a, as) (Δ @ Γ, e, S)
    by auto
  }
  moreover
  {
    have ⋀ x. x ∈ domA Γ ⟹ x ∉ edom ?ae
      using fresh-distinct[OF let1(1)]
```

```
      by (auto dest!: subsetD[OF edom-Aheap])
    hence map-transform Aeta-expand (?ae ⊔ ae) (map-transform transform (?ae ⊔ 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 ⊆ domA Γ ∪ upds S›
    have ⋀ x. x ∈ domA Δ ⟹ x ∉ edom ae
      using fresh-distinct[OF let1(1)] fresh-distinct-fv[OF let1(2)]
      by (auto dest!: subsetD[OF ups-fv-subset])
    hence map-transform Aeta-expand (?ae ⊔ ae) (map-transform transform (?ae ⊔ 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) ⇒ a-transform (?ae ⊔ ae, a, as) (Δ @ Γ, e,
S)
      using restr-stack-simp2 let1(1,2)
    apply (auto simp add: map-transform-append restrictA-append restr-stack-simp2[simplified]
map-transform-restrA)
      apply (rule step.let1)
      apply (auto dest: subsetD[OF edom-Aheap])
      done
  }
  ultimately
  show ?case by (blast del: consistentI consistentE)
 next
  case (if1 Γ scrut e1 e2 S)
  have consistent (ae, 0, a#as) (Γ, scrut, Alts e1 e2 # S)
    using if1 by (auto dest: a-consistent-if1)
  moreover
  have a-transform (ae, a, as) (Γ, scrut ? e1 : e2, S) ⇒ 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 (if2 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 if2 by (auto dest!: a-consistent-if2)
  moreover
  have a-transform (ae, a, as) (Γ, Bool b, Alts e1 e2 # S) ⇒ a-transform (ae, a', as') (Γ, if
b then e1 else e2, S)
   by (auto intro: step.if2[where b = True, simplified] step.if2[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* ⇒* *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'* ⇒* *a-transform* (*ae''*, *a''*, *as''*) *c''* **by** *blast*
from *this*(*1*) *rtranclp-trans*[*OF* ∗ ∗∗]
show *?case* **by** *blast*
qed
end

end

8 Cardinality Analysis

8.1 Cardinality-Domain

theory *Cardinality−Domain*
imports *Launchbury.HOLCF−Utils*
begin

type-synonym *oneShot* = *one*
abbreviation *notOneShot* :: *oneShot* **where** *notOneShot* ≡ *ONE*
abbreviation *oneShot* :: *oneShot* **where** *oneShot* ≡ ⊥

type-synonym *two* = *oneShot*$_\perp$
abbreviation *many* :: *two* **where** *many* ≡ *up*·*notOneShot*
abbreviation *once* :: *two* **where** *once* ≡ *up*·*oneShot*
abbreviation *none* :: *two* **where** *none* ≡ ⊥

lemma *many-max*[*simp*]: *a* ⊑ *many* **by** (*cases a*) *auto*

lemma *two-conj*: *c* = *many* ∨ *c* = *once* ∨ *c* = *none* **by** (*metis Exh-Up one-neq-iffs*(*1*))

lemma *two-cases*[*case-names many once none*]:
obtains *c* = *many* | *c* = *once* | *c* = *none* **using** *two-conj* **by** *metis*

definition *two-pred* **where** *two-pred* = (Λ *x*. *if x* ⊑ *once then* ⊥ *else x*)

lemma *two-pred-simp*: *two-pred*·*c* = (*if c* ⊑ *once then* ⊥ *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-pred·*many* = *many*
two-pred·*once* = *none*
two-pred·*none* = *none*
by (*simp-all add*: *two-pred-simp*)

lemma *two-pred-below-arg*: *two-pred* · *f* ⊑ *f*
by (*auto simp add*: *two-pred-simp*)

lemma *two-pred-none*: *two-pred*·*c* = *none* ⟷ *c* ⊑ *once*
by (*auto simp add*: *two-pred-simp*)

definition *record-call* **where** *record-call x* = (Λ *ce*. (λ *y*. *if x* = *y then two-pred*·(*ce y*) *else ce y*))

lemma *record-call-simp*: (*record-call x* · *f*) *x*′ = (*if x* = *x*′ *then two-pred* · (*f x*′) *else f x*′)
unfolding *record-call-def* **by** *auto*

lemma *record-call*[*simp*]: (*record-call x* · *f*) *x* = *two-pred* · (*f x*)
unfolding *record-call-simp* **by** *auto*

lemma *record-call-other*[*simp*]: *x*′ ≠ *x* ⟹ (*record-call x* · *f*) *x*′ = *f x*′
unfolding *record-call-simp* **by** *auto*

lemma *record-call-below-arg*: *record-call x* · *f* ⊑ *f*
unfolding *record-call-def*
by (*auto intro*!: *fun-belowI two-pred-below-arg*)

definition *two-add* :: *two* → *two* → *two*
where *two-add* = (Λ *x*. (Λ *y*. *if x* ⊑ ⊥ *then y else* (*if y* ⊑ ⊥ *then x else many*)))

lemma *two-add-simp*: *two-add*·*x*·*y* = (*if x* ⊑ ⊥ *then y else* (*if y* ⊑ ⊥ *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*·(*two-add*·*once*·*c*)
by (*cases c rule*: *two-cases*) (*auto simp add*: *two-add-simp*)

end

8.2 CardinalityAnalysisSig

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

8.3 CardinalityAnalysisSpec

theory *CardinalityAnalysisSpec*
imports *ArityAnalysisSpec CardinalityAnalysisSig ConstOn*
begin

locale *CardinalityPrognosisEdom* = *CardinalityPrognosis* +
assumes *edom-prognosis*:
edom (*prognosis ae as a* (Γ, *e*, *S*)) ⊆ *fv* Γ ∪ *fv e* ∪ *fv S*

locale *CardinalityPrognosisShape* = *CardinalityPrognosis* +
assumes *prognosis-env-cong*: *ae* $f|`$ *domA* Γ = *ae′* $f|`$ *domA* Γ ⟹ *prognosis ae as u* (Γ, *e*, *S*) = *prognosis ae′ as u* (Γ, *e*, *S*)
assumes *prognosis-reorder*: *map-of* Γ = *map-of* Δ ⟹ *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*) ⊑ *prognosis ae as u* (Γ, *e*, *Upd x* # *S*)
assumes *prognosis-not-called*: *ae x* = ⊥ ⟹ *prognosis ae as a* (Γ, *e*, *S*) ⊑ *prognosis ae as a* (*delete x* Γ, *e*, *S*)
assumes *prognosis-called*: *once* ⊑ *prognosis ae as a* (Γ, *Var x*, *S*) *x*

locale *CardinalityPrognosisApp* = *CardinalityPrognosis* +
assumes *prognosis-App*: *prognosis ae as* (*inc*·*a*) (Γ, *e*, *Arg x* # *S*) ⊑ *prognosis ae as a* (Γ, *App e x*, *S*)

locale *CardinalityPrognosisLam* = *CardinalityPrognosis* +

assumes *prognosis-subst-Lam*: *prognosis ae as* (*pred*·*a*) (Γ, *e*[*y*::=*x*], *S*) ⊑ *prognosis ae as a* (Γ, *Lam* [*y*]. *e*, *Arg x* # *S*)

locale *CardinalityPrognosisVar* = *CardinalityPrognosis* +
assumes *prognosis-Var-lam*: *map-of* Γ *x* = *Some e* ⟹ *ae x* = *up*·*u* ⟹ *isVal e* ⟹ *prognosis ae as u* (Γ, *e*, *S*) ⊑ *record-call x* · (*prognosis ae as a* (Γ, *Var x*, *S*))
assumes *prognosis-Var-thunk*: *map-of* Γ *x* = *Some e* ⟹ *ae x* = *up*·*u* ⟹ ¬ *isVal e* ⟹ *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) ⊑ *record-call x* · (*prognosis ae as a* (Γ, *Var x*, *S*))
assumes *prognosis-Var2*: *isVal e* ⟹ *x* ∉ *domA* Γ ⟹ *prognosis ae as 0* ((*x*, *e*) # Γ, *e*, *S*) ⊑ *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*) ⊑ *prognosis ae as a* (Γ, *scrut ? e1* : *e2*, *S*)
assumes *prognosis-Alts*: *prognosis ae as a* (Γ, *if b then e1 else e2*, *S*) ⊑ *prognosis ae* (*a*#*as*) *0* (Γ, *Bool b*, *Alts e1 e2* # *S*)

locale *CardinalityPrognosisLet* = *CardinalityPrognosis* + *CardinalityHeap* + *ArityAnalysisHeap* +
assumes *prognosis-Let*:
atom ‘ *domA* Δ ♯∗ Γ ⟹ *atom* ‘ *domA* Δ ♯∗ *S* ⟹ *edom ae* ⊆ *domA* Γ ∪ *upds S* ⟹ *prognosis* (*Aheap* Δ *e*·*a* ⊔ *ae*) *as a* (Δ @ Γ, *e*, *S*) ⊑ *cHeap* Δ *e*·*a* ⊔ *prognosis ae as a* (Γ, *Terms.Let* Δ *e*, *S*)

locale *CardinalityHeapSafe* = *CardinalityHeap* + *ArityAnalysisHeap* +
assumes *Aheap-heap3*: *x* ∈ *thunks* Γ ⟹ *many* ⊑ (*cHeap* Γ *e*·*a*) *x* ⟹ (*Aheap* Γ *e*·*a*) *x* = *up*·*0*
assumes *edom-cHeap*: *edom* (*cHeap* Δ *e*·*a*) = *edom* (*Aheap* Δ *e*·*a*)

locale *CardinalityPrognosisSafe* =
CardinalityPrognosisEdom +
CardinalityPrognosisShape +
CardinalityPrognosisApp +
CardinalityPrognosisLam +
CardinalityPrognosisVar +
CardinalityPrognosisLet +
CardinalityPrognosisIfThenElse +
CardinalityHeapSafe +
ArityAnalysisLetSafe

end

8.4 NoCardinalityAnalysis

theory *NoCardinalityAnalysis*
imports *CardinalityAnalysisSpec ArityAnalysisStack*
begin

locale *NoCardinalityAnalysis* = *ArityAnalysisLetSafe* +

assumes *Aheap-thunk*: $x \in \mathit{thunks}\ \Gamma \Longrightarrow (\mathit{Aheap}\ \Gamma\ e{\cdot}a)\ x = \mathit{up}{\cdot}0$
begin

definition *a2c* :: $\mathit{Arity}_\perp \to \mathit{two}$ **where** $\mathit{a2c} = (\Lambda\ a.\ \mathit{if}\ a \sqsubseteq \perp\ \mathit{then}\ \perp\ \mathit{else}\ \mathit{many})$
lemma *a2c-simp*: $\mathit{a2c}{\cdot}a = (\mathit{if}\ a \sqsubseteq \perp\ \mathit{then}\ \perp\ \mathit{else}\ \mathit{many})$
unfolding *a2c-def* **by** (*rule beta-cfun*[*OF cont-if-else-above*]) *auto*

lemma *a2c-eqvt*[*eqvt*]: $\pi \cdot \mathit{a2c} = \mathit{a2c}$
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** $\mathit{ae2ce}\ \mathit{ae}\ x = \mathit{a2c}{\cdot}(\mathit{ae}\ x)$

lemma *ae2ce-cont*: *cont ae2ce*
by (*auto simp add*: *ae2ce-def*)
lemmas *cont-compose*[*OF ae2ce-cont*, *cont2cont*, *simp*]

lemma *ae2ce-eqvt*[*eqvt*]: $\pi \cdot \mathit{ae2ce}\ \mathit{ae}\ x = \mathit{ae2ce}\ (\pi \cdot \mathit{ae})\ (\pi \cdot x)$
unfolding *ae2ce-def* **by** *perm-simp rule*

lemma *ae2ce-to-env-restr*: $\mathit{ae2ce}\ \mathit{ae} = (\lambda\text{-}\ .\ \mathit{many})\ f|`\ \mathit{edom}\ \mathit{ae}$
by (*auto simp add*: *ae2ce-def lookup-env-restr-eq edom-def a2c-simp*)

lemma *edom-ae2ce*[*simp*]: $\mathit{edom}\ (\mathit{ae2ce}\ \mathit{ae}) = \mathit{edom}\ \mathit{ae}$
unfolding *edom-def*
by (*auto simp add*: *ae2ce-def a2c-simp*)

definition *cHeap* :: *heap* ⇒ *exp* ⇒ *Arity* → (*var* ⇒ *two*)
where $\mathit{cHeap}\ \Gamma\ e = (\Lambda\ a.\ \mathit{ae2ce}\ (\mathit{Aheap}\ \Gamma\ e{\cdot}a))$
lemma *cHeap-simp*[*simp*]: $\mathit{cHeap}\ \Gamma\ e{\cdot}a = \mathit{ae2ce}\ (\mathit{Aheap}\ \Gamma\ e{\cdot}a)$
unfolding *cHeap-def* **by** *simp*

sublocale *CardinalityHeap cHeap*.

sublocale *CardinalityHeapSafe cHeap Aheap*
apply *standard*
apply (*erule Aheap-thunk*)
apply *simp*
done

fun *prognosis* **where**
$\mathit{prognosis}\ \mathit{ae}\ \mathit{as}\ a\ (\Gamma,\ e,\ S) = ((\lambda\text{-}.\ \mathit{many})\ f|`\ (\mathit{edom}\ (\mathit{ABinds}\ \Gamma{\cdot}\mathit{ae}) \cup \mathit{edom}\ (\mathit{Aexp}\ e{\cdot}a) \cup \mathit{edom}\ (\mathit{AEstack}\ \mathit{as}\ S)))$

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*: *subsetD*[*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*: *subsetD*[*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*: *subsetD*[*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 *subsetD*[*OF prems*(*3*)] *fresh-distinct*[*OF prems*(*1*)] *fresh-distinct-fv*[*OF prems*(*2*)]
  have *ae f*|‘ *domA* Δ = ⊥
    by (*auto dest*: *subsetD*[*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*!: *subsetD*[*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*: *subsetD*[*OF Aexp-edom*] *subsetD*[*OF ap-fv-subset*] *subsetD*[*OF edom-AnalBinds*] *subsetD*[*OF edom-AEstack*])

sublocale *CardinalityPrognosisSafe prognosis cHeap Aheap Aexp*..
end

end

8.5 CardArityTransformSafe

theory *CardArityTransformSafe*
imports *ArityTransform CardinalityAnalysisSpec AbstractTransform Sestoft SestoftGC ArityEta-ExpansionSafe 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* ≡ *transform*

lemma *supp-transform*: *supp* (*transform a e*) ⊆ *supp e*
by (*induction rule*: *transform.induct*)
(*auto simp add*: *exp-assn.supp Let-supp dest*!: *subsetD*[*OF supp-map-transform*] *subsetD*[*OF supp-map-transform-step*])
interpretation *supp-bounded-transform transform*
by *standard* (*auto simp add*: *fresh-def supp-transform*)

type-synonym *tstate* = (*AEnv* × (*var* ⇒ *two*) × *Arity* × *Arity list* × *var list*)

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* ♯∗ *S* ⟹ *a* ♯∗ *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*)

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*!: *subsetD*[*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* ⇒$_G$* *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*) ⇒$_G$ *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*) ⊑ *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* ∈ *thunks* Γ **by** *auto*
hence [*simp*]: *x* ∈ *domA* Γ **by** (*rule subsetD*[*OF thunks-domA*])

from *thunk* **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*)
hence [*simp*]: *x* ∉ *set r* **using** *thunk* **by** *auto*

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*
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* ∈ *edom ce*› **have** *False* **by** (*auto simp add*: *edom-def*)
thus *?thesis*..
next
case *once*

from ‹*prognosis ae as a* (Γ, *Var x*, *S*) ⊑ *ce*›
have *prognosis ae as a* (Γ, *Var x*, *S*) *x* ⊑ *once*
using *once* **by** (*metis* (*mono-tags*) *fun-belowD*)
hence *x* ∉ *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*› ‹¬ *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*)

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*) ⇒$_G$
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 . . . ⇒$_G^*$ *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 . . . ⇒$_G^*$ *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*) ⇒$_G^*$ *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*) ⇒$_G$ *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*]: $x \in domA\ \Gamma$ **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*]: $x \in edom\ ce$ **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*]: $x \notin set\ r$ **by** *auto*

then have $x \in edom\ ae$ **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*) # Γ, *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*) (Γ, *e*, *Upd x* # *S*) $\Rightarrow_G$ *conf-transform* (*ae*, *ce*, *0*, *as*, *r*) ((*x*, *e*) # Γ, *e*, *S*)
using ‹*ae x* = *up*·*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* = ⊥ **using** *var*$_2$ **by** (*auto simp add*: *edom-def*)
hence *x* ∉ *edom ce* **by** (*simp add*: *edomIff*)
hence *x* ∉ *edom ae* **using** *var*$_2$ **by** *auto*
hence [*simp*]: *ae x* = ⊥ **by** (*auto simp add*: *edom-def*)

note ‹*x* ∈ *set r*›[*simp*]

have *prognosis ae as a* ((*x*, *e*) # Γ, *e*, *S*) ⊑ *prognosis ae as a* ((*x*, *e*) # Γ, *e*, *Upd x* # *S*) **by** (*rule prognosis-upd*)
also have ... ⊑ *prognosis ae as a* (*delete x* ((*x*,*e*) # Γ), *e*, *Upd x* # *S*)
using ‹*ae x* = ⊥› **by** (*rule prognosis-not-called*)
also have *delete x* ((*x*,*e*)#Γ) = Γ **using** ‹*x* ∉ *domA* Γ› **by** *simp*
finally
have ∗: *prognosis ae as a* ((*x*, *e*) # Γ, *e*, *S*) ⊑ *prognosis ae as a* (Γ, *e*, *Upd x* # *S*) **by** *this simp*
then
have *consistent* (*ae*, *ce*, *a*, *as*, *r*) ((*x*, *e*) # Γ, *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*) (Γ, *e*, *Upd x* # *S*) = *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) ((*x*, *e*) # Γ, *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*$_1$ Δ Γ *e S*)
let *?ae* = *Aheap* Δ *e*·*a*
let *?ce* = *cHeap* Δ *e*·*a*

have *domA* Δ ∩ *upds S* = {} **using** *fresh-distinct-fv*[*OF let*$_1$(*2*)] **by** (*auto dest*: *subsetD*[*OF ups-fv-subset*])
hence ∗: ⋀ *x*. *x* ∈ *upds S* ⟹ *x* ∉ *edom ?ae* **by** (*auto simp add*: *edom-cHeap dest*!: *subsetD*[*OF edom-Aheap*])
have *restr-stack-simp2*: *restr-stack* (*edom* (*?ae* ⊔ *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 *subsetD*[*OF this*] *fresh-distinct*[*OF let*$_1$(*1*)] *fresh-distinct-fv*[*OF let*$_1$(*2*)]
have *edom ae* ∩ *domA* Δ = {} **by** (*auto dest*: *subsetD*[*OF ups-fv-subset*])

from ‹*edom ae* ∩ *domA* Δ = {}›
have [*simp*]: *edom* (*Aheap* Δ *e*·*a*) ∩ *edom ae* = {} **by** (*auto dest*!: *subsetD*[*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*!: *subsetD*[*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*: *subsetD*[*OF edom-Aheap*] *subsetD*[*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*!: *subsetD*[*OF thunks-domA*] *subsetD*[*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* ⊆ *domA* Γ ∪ *upds S*›
have *prognosis* (*?ae* ⊔ *ae*) *as a* (Δ @ Γ, *e*, *S*) ⊑ *?ce* ⊔ *prognosis ae as a* (Γ, *Let* Δ *e*, *S*) **by** (*rule prognosis-Let*)
also have *prognosis ae as a* (Γ, *Let* Δ *e*, *S*) ⊑ *ce* **using** let_1 **by** *auto*
finally have *prognosis* (*?ae* ⊔ *ae*) *as a* (Δ @ Γ, *e*, *S*) ⊑ *?ce* ⊔ *ce* **by** *this simp*
}
moreover

have *a-consistent* (*ae*, *a*, *as*) (*restrictA* (− *set r*) Γ, *Let* Δ *e*, *restr-stack* (− *set r*) *S*)

using let_1 **by** *auto*
hence *a-consistent* (*?ae* ⊔ *ae*, *a*, *as*) (Δ @ *restrictA* (− *set r*) Γ, *e*, *restr-stack* (− *set r*) *S*)
using let_1(*1*,*2*) ‹*edom ae* ∩ *domA* Δ = {}›
by (*auto intro*!: *a-consistent-let simp del*: *join-comm*)
hence *a-consistent* (*?ae* ⊔ *ae*, *a*, *as*) (*restrictA* (− *set r*) (Δ @ Γ), *e*, *restr-stack* (− *set r*) *S*)
by (*simp add*: *restrictA-append*)
moreover
have *set r* ⊆ (*domA* Γ ∪ *upds S*) − *edom ce* **using** let_1 **by** *auto*
hence *set r* ⊆ (*domA* Γ ∪ *upds S*) − *edom* (*?ce* ⊔ *ce*)
apply (*rule order-trans*)
using ‹*domA* Δ ∩ *domA* Γ = {}› ‹*domA* Δ ∩ *upds S* = {}›
apply (*auto simp add*: *edom-cHeap dest*!: *subsetD*[*OF edom-Aheap*])
done
ultimately
have *consistent* (*?ae* ⊔ *ae*, *?ce* ⊔ *ce*, *a*, *as*, *r*) (Δ @ Γ, *e*, *S*) **by** *auto*
}
moreover
{
have ⋀ *x*. *x* ∈ *domA* Γ ⟹ *x* ∉ *edom ?ae* ⋀ *x*. *x* ∈ *domA* Γ ⟹ *x* ∉ *edom ?ce*
using *fresh-distinct*[*OF* let_1(*1*)]
by (*auto simp add*: *edom-cHeap dest*!: *subsetD*[*OF edom-Aheap*])
hence *map-transform Aeta-expand* (*?ae* ⊔ *ae*) (*map-transform transform* (*?ae* ⊔ *ae*) (*restrictA* (−*set r*) Γ))
= *map-transform Aeta-expand ae* (*map-transform transform ae* (*restrictA* (−*set r*) Γ))
by (*auto intro*!: *map-transform-cong restrictA-cong simp add*: *edomIff*)
moreover

from ‹*edom ae* ⊆ *domA* Γ ∪ *upds S*› ‹*edom ce* = *edom ae*›
have ⋀ *x*. *x* ∈ *domA* Δ ⟹ *x* ∉ *edom ce* **and** ⋀ *x*. *x* ∈ *domA* Δ ⟹ *x* ∉ *edom ae*
using *fresh-distinct*[*OF* let_1(*1*)] *fresh-distinct-ups*[*OF* let_1(*2*)] **by** *auto*
hence *map-transform Aeta-expand* (*?ae* ⊔ *ae*) (*map-transform transform* (*?ae* ⊔ *ae*) (*restrictA* (− *set r*) Δ))
= *map-transform Aeta-expand ?ae* (*map-transform transform ?ae* (*restrictA* (− *set r*) Δ))
by (*auto intro*!: *map-transform-cong restrictA-cong simp add*: *edomIff*)
moreover

from ‹*domA* Δ ∩ *domA* Γ = {}› ‹*domA* Δ ∩ *upds S* = {}›
have *atom* ‘ *domA* Δ ♯* *set r*
by (*auto simp add*: *fresh-star-def fresh-at-base fresh-finite-set-at-base dest*!: *subsetD*[*OF* ‹*set r* ⊆ *domA* Γ ∪ *upds S*›])
hence *atom* ‘ *domA* Δ ♯* *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* ⊔ *ae*, *?ce* ⊔ *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*: *subsetD*[*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*) ⊑ *ce* **using** if_1 **by** *auto*
hence *prognosis ae* (*a*#*as*) *0* (Γ, *scrut*, *Alts e1 e2* # *S*) ⊑ *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*) ⊑ *ce* **using** if_2 **by** *auto*
hence *prognosis ae as′ a′* (Γ, *if b then e1 else e2*, *S*) ⊑ *ce* **by** (*rule below-trans*[*OF prognosis-Alts*])
then
have *consistent* (*ae*, *ce*, *a′*, *as′*, *r*) (Γ, *if b then e1 else e2*, *S*)
using if_2 **by** (*auto dest*!: *a-consistent-if*$_2$)
}
moreover
have *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ, *Bool b*, *Alts e1 e2* # *S*) $\Rightarrow_G$ *conf-transform* (*ae*, *ce*, *a′*, *as′*, *r*) (Γ, *if b then e1 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

9 Trace Trees

9.1 TTree

theory *TTree*
imports *Main ConstOn List−Interleavings*
begin

9.1.1 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)
```

9.1.2 The type of infinite labeled trees

typedef *'a ttree* = {*xss* :: *'a list set* . [] ∈ *xss* ∧ *downset xss*} **by** *auto*

setup-lifting *type-definition-ttree*

9.1.3 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*)

9.1.4 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'* ⟷ *t* = *t'* **by** *transfer auto*

lemma *paths-Nil*[*simp*]: [] ∈ *paths t* **by** *transfer simp*

lemma *paths-not-empty*[*simp*]: (*paths t* = {}) ⟷ *False* **by** *transfer auto*

lemma *paths-Cons-nxt*:
possible t x ⟹ *xs* ∈ *paths* (*nxt t x*) ⟹ (*x*#*xs*) ∈ *paths t*
by *transfer auto*

lemma *paths-Cons-nxt-iff*:
possible t x ⟹ *xs* ∈ *paths* (*nxt t x*) ⟷ (*x*#*xs*) ∈ *paths t*
by *transfer auto*

lemma *possible-mono*:
paths t ⊆ *paths t'* ⟹ *possible t x* ⟹ *possible t' x*
by *transfer auto*

lemma *nxt-mono*:
paths t ⊆ *paths t'* ⟹ *paths* (*nxt t x*) ⊆ *paths* (*nxt t' x*)
by *transfer auto*

lemma *ttree-eqI*: (⋀ *x xs*. *x*#*xs* ∈ *paths t* ⟷ *x*#*xs* ∈ *paths t'*) ⟹ *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 $x\#xs \in paths\ t \mid xs = []$
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 $xs \in paths\ t \longleftrightarrow xs \in paths\ t'$
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

9.1.5 The carrier of a tree

lift-definition *carrier* :: *'a ttree* ⇒ *'a set* **is** λ *xss*. $\bigcup$(*set* ‘ *xss*)**.**

lemma *carrier-mono*: $paths\ t \subseteq paths\ t' \Longrightarrow carrier\ t \subseteq carrier\ t'$ **by** *transfer auto*

lemma *carrier-possible*:
possible t x $\Longrightarrow$ *x* $\in$ *carrier t* **by** *transfer force*

lemma *carrier-possible-subset*:
carrier t $\subseteq$ *A* $\Longrightarrow$ *possible t x* $\Longrightarrow$ *x* $\in$ *A* **by** *transfer force*

lemma *carrier-nxt-subset*:
carrier (*nxt t x*) $\subseteq$ *carrier t*
by *transfer auto*

lemma *Union-paths-carrier*: ($\bigcup$*x*$\in$*paths t. set x*) = *carrier t*
by *transfer auto*

9.1.6 Repeatable trees

definition *repeatable* **where** *repeatable t* = ($\forall$ *x . possible t x* $\longrightarrow$ *nxt t x* = *t*)

lemma *nxt-repeatable*[*simp*]: *repeatable t* $\Longrightarrow$ *possible t x* $\Longrightarrow$ *nxt t x* = *t*
unfolding *repeatable-def* **by** *auto*

9.1.7 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* ⇒ *'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'* ⟷ *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'* ⟷ *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*))

9.1.8 Intersection of two trees

lift-definition *intersect* :: *'a ttree* ⇒ *'a ttree* ⇒ *'a ttree* (**infixl** ‹∩∩› *80*)
is (∩)
by (*auto simp add*: *downset-def*)

lemma *paths-intersect*[*simp*]: *paths* (*t* ∩∩ *t'*) = *paths t* ∩ *paths t'*
by *transfer auto*

lemma *carrier-intersect*: *carrier* $(t \cap\cap t') \subseteq$ *carrier* $t \cap$ *carrier* t'
unfolding *Union-paths-carrier*[*symmetric*]
by *auto*

9.1.9 Disjoint union of trees

lift-definition *either* :: $'a$ *ttree* $\Rightarrow$ $'a$ *ttree* $\Rightarrow$ $'a$ *ttree* (**infixl** ‹$\oplus\oplus$› *80*)
is $(\cup)$
by (*auto simp add*: *downset-def*)

lemma *either-empty1*[*simp*]: *empty* $\oplus\oplus$ $t = t$
by *transfer auto*
lemma *either-empty2*[*simp*]: $t \oplus\oplus$ *empty* $= t$
by *transfer auto*
lemma *either-sym*[*simp*]: $t \oplus\oplus$ *t2* $=$ *t2* $\oplus\oplus$ t
by *transfer auto*
lemma *either-idem*[*simp*]: $t \oplus\oplus$ $t = t$
by *transfer auto*

lemma *possible-either*[*simp*]: *possible* $(t \oplus\oplus t')$ $x \longleftrightarrow$ *possible* t $x \vee$ *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** λ S. *insert* [] $(\bigcup S)$
by (*auto simp add*: *downset-def*)

lemma *paths-Either*: *paths* (*Either ts*) $=$ *insert* [] $(\bigcup$(*paths* ‘ *ts*))
by *transfer auto*

9.1.10 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* ⇒ *'a ttree* ⇒ *'a ttree* (**infixl** ‹⊗⊗› *86*)
is λ *xss yss* . ⋃ {*xs* ⊗ *ys* | *xs ys*. *xs* ∈ *xss* ∧ *ys* ∈ *yss*}
by (*force simp*: *ex-ex-eq-hint dest*: *interleave-butlast*)

lemma *both-assoc*[*simp*]: *t* ⊗⊗ (*t'* ⊗⊗ *t''*) = (*t* ⊗⊗ *t'*) ⊗⊗ *t''*
apply *transfer*
apply *auto*
apply (*metis interleave-assoc2*)
apply (*metis interleave-assoc1*)
done

lemma *both-comm*: *t* ⊗⊗ *t'* = *t'* ⊗⊗ *t*
by *transfer* (*auto*, (*metis interleave-comm*)+)

lemma *both-empty1*[*simp*]: *empty* ⊗⊗ *t* = *t*
by *transfer auto*

lemma *both-empty2*[*simp*]: *t* ⊗⊗ *empty* = *t*
by *transfer auto*

lemma *paths-both*: *xs* ∈ *paths* (*t* ⊗⊗ *t'*) ⟷ (∃ *ys* ∈ *paths t*. ∃ *zs* ∈ *paths t'*. *xs* ∈ *ys* ⊗ *zs*)
by *transfer fastforce*

lemma *both-contains-arg1*: *paths t* ⊆ *paths* (*t* ⊗⊗ *t'*)
by *transfer fastforce*

lemma *both-contains-arg2*: *paths t'* ⊆ *paths* (*t* ⊗⊗ *t'*)
by *transfer fastforce*

lemma *both-mono1*:
paths t ⊆ *paths t'* ⟹ *paths* (*t* ⊗⊗ *t''*) ⊆ *paths* (*t'* ⊗⊗ *t''*)
by *transfer auto*

lemma *both-mono2*:
paths t ⊆ *paths t'* ⟹ *paths* (*t''* ⊗⊗ *t*) ⊆ *paths* (*t''* ⊗⊗ *t'*)
by *transfer auto*

lemma *possible-both*[*simp*]: *possible* (*t* ⊗⊗ *t'*) *x* ⟷ *possible t x* ∨ *possible t' x*
proof
assume *possible* (*t* ⊗⊗ *t'*) *x*
then obtain *xs* **where** *x*#*xs* ∈ *paths* (*t* ⊗⊗ *t'*)
by *transfer auto*

from ‹*x*#*xs* ∈ *paths* (*t* ⊗⊗ *t'*)›
obtain *ys zs* **where** *ys* ∈ *paths t* **and** *zs* ∈ *paths t'* **and** *x*#*xs* ∈ *ys* ⊗ *zs*
by *transfer auto*

from ‹*x*#*xs* ∈ *ys* ⊗ *zs*›

have $ys \neq [] \wedge hd\ ys = x \ \vee zs \neq [] \wedge hd\ zs = x$
by (*auto elim*: *interleave-cases*)
thus *possible* $t\ x \vee$ *possible* $t'\ x$
using ‹$ys \in$ *paths* t› ‹$zs \in$ *paths* t'›
by *transfer auto*
next
assume *possible* $t\ x \vee$ *possible* $t'\ x$
then obtain xs **where** $x\#xs \in$ *paths* $t \vee x\#xs \in$ *paths* t'
by *transfer auto*
from *this* **have** $x\#xs \in$ *paths* $(t \otimes\otimes t')$ **by** (*auto dest*: *subsetD*[*OF both-contains-arg1*] *subsetD*[*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* $t'\ x \wedge$ *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*:
$x \#\ xs \in$ *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-calls x* ⊗⊗ *t*) *x* = *many-calls x* ⊗⊗ (*t* ⊕⊕ *nxt t x*)
by (*simp add*: *repeatable-many-calls*)

lemma *repeatable-both-self*[*simp*]:
assumes [*simp*]: *repeatable t*
shows *t* ⊗⊗ *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* ⊗⊗ *t'* ⊗⊗ *t* = *t* ⊗⊗ *t'*
by (*metis repeatable-both-self*[*OF assms*] *both-assoc both-comm*)

lemma *repeatable-both-both2*[*simp*]:
assumes *repeatable t*
shows *t'* ⊗⊗ *t* ⊗⊗ *t* = *t'* ⊗⊗ *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'* ⊗⊗ *t* = *t'*
shows *nxt t' x* ⊗⊗ *t* = *nxt t' x*
proof(*rule classical*)
assume *nxt t' x* ⊗⊗ *t* ≠ *nxt t' x*
hence (*nxt t' x* ⊕⊕ *t'*) ⊗⊗ *t* ≠ *nxt t' x* **by** (*metis* (*no-types*) *assms*(*1*) *both-assoc repeatable-both-self*)
thus *nxt t' x* ⊗⊗ *t* = *nxt t' x* **by** (*metis* (*no-types*) *assms either-both-distr2 nxt-both nxt-repeatable*)
qed

lemma *repeatable-both-both-nxt*:
assumes *t'* ⊗⊗ *t* = *t'*
shows *t'* ⊗⊗ *t''* ⊗⊗ *t* = *t'* ⊗⊗ *t''*
by (*metis assms both-assoc both-comm*)

lemma *carrier-both*[*simp*]:
carrier (*t* ⊗⊗ *t'*) = *carrier t* ∪ *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* ∪ *set zs* **by** (*rule interleave-set*)
with ‹*ys* ∈ -› ‹*zs* ∈ -› ‹*x* ∈ *set xs*›
have $x \in$ *carrier* $t \cup$ *carrier* t' **by** *transfer auto*
}
moreover
note *subsetD*[*OF carrier-mono*[*OF both-contains-arg1*[**where** *t*=*t* **and** $t' = t'$]]]
subsetD[*OF carrier-mono*[*OF both-contains-arg2*[**where** *t*=*t* **and** $t' = t'$]]]
ultimately
show *?thesis* **by** *auto*
qed

9.1.11 Removing elements from a tree

lift-definition *without* :: $'a \Rightarrow$ *'a ttree* $\Rightarrow$ *'a ttree*
is λ *x xss*. *filter* $(\lambda\ x'.\ x' \neq x)$ ‘ *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* ∈ *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* ∈ *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** λ *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-restr S t*) **by** *transfer auto*

lemma *filter-paths-conv-free-restr2*:
filter $(\lambda\ x'\ .\ x' \notin S)$ ‘ *paths t* = *paths* (*ttree-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* ∩ *S* = {} ⟹ *ttree-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-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-restr S (*ttree-restr* S' t) = *ttree-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*

9.1.12 Multiple variables, each called at most once

lift-definition *singles* :: *'a set* ⇒ *'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 (⋀ *x*. *P x* ⟹ *Q x*)
shows *length* (*filter P xs*) ≤ *length* (*filter Q xs*)
by (*induction xs*) (*auto dest*: *assms*)

lemma *nxt-singles*[*simp*]: *nxt* (*singles S*) *x'* = (*if x'* ∈ *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 ⊆ *S'* ⟹ *paths* (*singles S'*) ⊆ *paths* (*singles S*)
by *transfer auto*

lemma *paths-many-calls-subset*:

paths t ⊆ *paths* (*many-calls x* ⊗⊗ *without x t*)
proof
fix *xs*
assume *xs* ∈ *paths t*

have *filter* (λ*x'*. *x'* = *x*) *xs* = *replicate* (*length* (*filter* (λ*x'*. *x'* = *x*) *xs*)) *x*
by (*induction xs*) *auto*
hence *filter* (λ*x'*. *x'* = *x*) *xs* ∈ *paths* (*many-calls x*) **by** *transfer auto*
moreover
from ‹*xs* ∈ *paths t*›
have *filter* (λ*x'*. *x'* ≠ *x*) *xs* ∈ *paths* (*without x t*) **by** *transfer auto*
moreover
have *xs* ∈ *interleave* (*filter* (λ*x'*. *x'* = *x*) *xs*) (*filter* (λ*x'*. *x'* ≠ *x*) *xs*) **by** (*rule interleave-filtered*)
ultimately show *xs* ∈ *paths* (*many-calls x* ⊗⊗ *without x t*) **by** *transfer auto*
qed

9.1.13 Substituting trees for every node

definition *f-nxt* :: ('*a* ⇒ '*a ttree*) ⇒ '*a set* ⇒ '*a* ⇒ ('*a* ⇒ '*a ttree*)
where *f-nxt f T x* = (*if x* ∈ *T then f*(*x*:=*empty*) *else f*)

fun *substitute'* :: ('*a* ⇒ '*a ttree*) ⇒ '*a set* ⇒ '*a ttree* ⇒ '*a list* ⇒ *bool* **where**
substitute'-Nil: *substitute' f T t* [] ⟷ *True*
| *substitute'-Cons*: *substitute' f T t* (*x*#*xs*) ⟷
possible t x ∧ *substitute'* (*f-nxt f T x*) *T* (*nxt t x* ⊗⊗ *f x*) *xs*

lemma *f-nxt-mono1*: (⋀ *x*. *paths* (*f x*) ⊆ *paths* (*f' x*)) ⟹ *paths* (*f-nxt f T x x'*) ⊆ *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* ⇒ '*a ttree*) ⇒ '*a set* ⇒ '*a ttree* ⇒ '*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* ⟷ *xs* ∈ *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* ⊆ *paths t'*
shows *paths* (*substitute f T t*) ⊆ *paths* (*substitute f T t'*)
proof
fix *xs*
assume *xs* ∈ *paths* (*substitute f T t*)
thus *xs* ∈ *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 ⋀*x*. *paths* (*f x*) ⊆ *paths* (*f' x*)
shows *paths* (*substitute f T t*) ⊆ *paths* (*substitute f' T t*)
proof
fix *xs*
assume *xs* ∈ *paths* (*substitute f T t*)
from *this assms*
show *xs* ∈ *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*: *subsetD dest*!: *subsetD*[*OF f-nxt-mono1*[*OF Cons.prems*(*2*)]]
subsetD[*OF substitute-mono2*[*OF both-mono2*[*OF Cons.prems*(*2*)]]])
qed
qed

lemma *substitute-monoT*:
assumes *T* ⊆ *T'*
shows *paths* (*substitute f T' t*) ⊆ *paths* (*substitute f T t*)
proof
fix *xs*
assume *xs* ∈ *paths* (*substitute f T' t*)
thus *xs* ∈ *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* ∈ *paths* (*substitute f T′ t*)›
have [*simp*]: *possible t x* **and** *xs* ∈ *paths* (*substitute* (*f-nxt f T′ x*) *T′* (*nxt t x* ⊗⊗ *f x*)) **by** *auto*
from *Cons.IH*[*OF this*(*2*) *Cons.prems*(*2*)]
have *xs* ∈ *paths* (*substitute* (*f-nxt f T′ x*) *T* (*nxt t x* ⊗⊗ *f x*)).
hence *xs* ∈ *paths* (*substitute* (*f-nxt f T x*) *T* (*nxt t x* ⊗⊗ *f x*))
by (*rule subsetD*[*OF substitute-mono1*, *rotated*])
(*auto simp add*: *f-nxt-def subsetD*[*OF Cons.prems*(*2*)])
thus *?case* **by** *auto*
qed
qed

lemma *substitute-contains-arg*: *paths t* ⊆ *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 subsetD*[*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*]: $\bigwedge\ t\ t'\ x\ .\ (nxt\ t\ x \oplus\oplus nxt\ t'\ x) \otimes\otimes f\ x = nxt\ t\ x \otimes\otimes f\ x \oplus\oplus nxt\ t'\ x \otimes\otimes f\ x$ **by** (*metis both-comm either-both-distr*)
{
fix *xs*
have $xs \in paths\ (substitute\ f\ T\ (t \oplus\oplus t')) \longleftrightarrow xs \in paths\ (substitute\ f\ T\ t) \vee xs \in paths\ (substitute\ f\ T\ t')$
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* $\otimes\otimes$ *f x* **and** *t'* = *nxt t x* $\otimes\otimes$ *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-nxt f* (*T* − {*x*}) *x'* = *f-nxt f T x'*
using *assms*
by (*auto simp add*: *f-nxt-def*)

lemma *f-nxt-empty*[*simp*]:
assumes *f x* = *empty*
shows *f-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-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* ∈ *paths* (*substitute f* (*T* − {*x*}) *t*) ⟷ *xs* ∈ *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* ∈ *paths* (*substitute f T t*) ⟷ *xs* ∈ *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* ⟹ *f-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* ⟹ (*xs* ∈ *paths* (*substitute f T* (*nxt t x*))) = (*xs* ∈ *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* ⟹ *substitute f T* (*t* ⊗⊗ *t'*) = *substitute f T t* ⊗⊗ *t'*
proof (*intro paths-inj set-eqI*)
fix *xs*
assume *const-on f* (*carrier t'*) *TTree.empty*
thus (*xs* ∈ *paths* (*substitute f T* (*t* ⊗⊗ *t'*))) = (*xs* ∈ *paths* (*substitute f T t* ⊗⊗ *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 ∈ 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 ⊗⊗ t' ⊗⊗ f x = nxt t x ⊗⊗ f x ⊗⊗ 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 ⊗⊗ 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-nxt f T x)(x' := empty)
 by (auto simp add: f-nxt-def)

lemma repeatable-f-nxt-upd[simp]:
 repeatable (f x) ⟹ repeatable (f-nxt f T x' x)
 by (auto simp add: f-nxt-def)

lemma substitute-remove-anyways-aux:
 assumes repeatable (f x)
 assumes xs ∈ paths (substitute f T t)
 assumes t ⊗⊗ f x = t
 shows xs ∈ 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-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-nxt f T x) x = f x ∨ (f-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 ⊗⊗ t') = substitute (f(x := empty)) T (t ⊗⊗ t')
proof (rule paths-inj, rule, rule subsetI)
  fix xs
  have repeatable (f x) using assms by simp
  moreover
  assume xs ∈ paths (substitute f T (t ⊗⊗ t'))
  moreover
  have t ⊗⊗ t' ⊗⊗ f x = t ⊗⊗ t'
    by (metis assms both-assoc both-comm repeatable-both-self)
  ultimately
  show xs ∈ paths (substitute (f(x := empty)) T (t ⊗⊗ t'))
     by (rule substitute-remove-anyways-aux)
next
  show paths (substitute (f(x := empty)) T (t ⊗⊗ t')) ⊆ paths (substitute f T (t ⊗⊗ t'))
    by (rule substitute-mono1) auto
qed

lemma carrier-f-nxt: carrier (f-nxt f T x x') ⊆ carrier (f x')
  by (simp add: f-nxt-def)

lemma f-nxt-cong: f x' = f' x' ⟹ f-nxt f T x x' = f-nxt f' T x x'
  by (simp add: f-nxt-def)

lemma substitute-cong':
  assumes xs ∈ paths (substitute f T t)
  assumes ⋀ x n. x ∈ A ⟹ carrier (f x) ⊆ A
  assumes carrier t ⊆ A
  assumes ⋀ x. x ∈ A ⟹ f x = f' x
```

shows *xs* ∈ *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* ∈ *carrier t* **by** (*metis carrier-possible*)
hence *x* ∈ *A* **using** *Cons.prems*(*3*) **by** *auto*
with *Cons.prems* **have** [*simp*]: *f′ x* = *f x* **by** *auto*
have *carrier* (*f x*) ⊆ *A* **using** ‹*x* ∈ *A*› **by** (*rule Cons.prems*(*2*))

from *Cons.prems*(*1*,*2*) *Cons.prems*(*4*)[*symmetric*]
show *?case*
by (*auto elim*!: *Cons.IH*
dest!: *subsetD*[*OF carrier-f-nxt*] *subsetD*[*OF carrier-nxt-subset*] *subsetD*[*OF Cons.prems*(*3*)]
subsetD[*OF* ‹*carrier* (*f x*) ⊆ *A*›]
intro: *f-nxt-cong*
)
qed

lemma *substitute-cong-induct*:
assumes ⋀ *x*. *x* ∈ *A* ⟹ *carrier* (*f x*) ⊆ *A*
assumes *carrier t* ⊆ *A*
assumes ⋀ *x*. *x* ∈ *A* ⟹ *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* ∈ *paths* (*substitute f T t*)
assumes *carrier t* ⊆ *A*
assumes ⋀ *x*. *x* ∈ *A* ⟹ *carrier* (*f x*) ⊆ *A*
shows *set xs* ⊆ *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*:
$f\text{-}nxt\ f\ T\ x\ x' = empty \longleftrightarrow f\ x' = empty \vee (x' = x \wedge x \in T)$
by (*auto simp add*: *f-nxt-def*)

lemma *substitute-T-cong'*:
assumes $xs \in paths\ (substitute\ f\ T\ t)$
assumes $\bigwedge x.\ (x \in T \longleftrightarrow x \in T') \vee f\ x = empty$
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-nxt-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-nxt-eq-empty-iff*)
from *Cons.prems*(*1*) *Cons.IH*[*OF - this*]
show *?case*
by *auto*
qed

lemma *substitute-cong-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-inj*)
apply (*rule set-eqI*)
apply (*rule iffI*)
apply (*erule substitute-T-cong'*[*OF - assms*])
apply (*erule substitute-T-cong'*)
apply (*metis assms*)
done

lemma *carrier-substitute1*: $carrier\ t \subseteq carrier\ (substitute\ f\ T\ t)$
by (*rule carrier-mono*) (*rule substitute-contains-arg*)

lemma *substitute-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-cong-induct*[*OF - - assms*])
show *carrier t ⊆ carrier* (*substitute f T t*)
by (*rule carrier-substitute1*)
next
fix *x*
assume *x ∈ carrier* (*substitute f T t*)
then obtain *xs* **where** *xs ∈ paths* (*substitute f T t*) **and** *x ∈ set xs* **by** *transfer auto*
thus *carrier* (*f x*) *⊆ 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 ‹*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*

from ‹*x ∈ set* (*x' # xs*)›
have *x = x' ∨* (*x ≠ x' ∧ x ∈ set xs*) **by** *auto*
hence *carrier* (*f x*) *⊆ carrier* (*substitute* (*f-nxt f T x'*) *T* (*nxt t x' ⊗⊗ f x'*))
proof(*elim conjE disjE*)
assume *x = x'*
have *carrier* (*f x*) *⊆ carrier* (*nxt t x ⊗⊗ f x*) **by** *simp*
also have *... ⊆ carrier* (*substitute* (*f-nxt f T x'*) *T* (*nxt t x ⊗⊗ f x*)) **by** (*rule carrier-substitute1*)
finally show *?thesis* **unfolding** ‹*x = x'*›**.**
next
assume *x ≠ x'*
hence [*simp*]: (*f-nxt f T x' x*) = *f x* **by** (*simp add*: *f-nxt-def*)

assume *x ∈ set xs*
from *Cons.IH*[*OF* ‹*xs ∈ -* › *this*]
show *carrier* (*f x*) *⊆ carrier* (*substitute* (*f-nxt f T x'*) *T* (*nxt t x' ⊗⊗ f x'*)) **by** *simp*
qed
also
from ‹*possible t x'*›
have *carrier* (*substitute* (*f-nxt f T x'*) *T* (*nxt t x' ⊗⊗ f x'*)) *⊆ 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 ⋀ x. const-on f' (carrier (f x)) empty
  shows substitute f T (substitute f' T t) = substitute (λ x. f x ⊗⊗ f' x) T t
proof (rule paths-inj, rule set-eqI)
  fix xs
  have [simp]: ⋀ f f' x'. f-nxt (λx. f x ⊗⊗ f' x) T x' = (λx. f-nxt f T x' x ⊗⊗ f-nxt f' T x' x)
    by (auto simp add: f-nxt-def)

  from assms
  show xs ∈ paths (substitute f T (substitute f' T t)) ⟷ xs ∈ paths (substitute (λx. f x ⊗⊗ 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': ⋀ x'. const-on (f-nxt f' T x) (carrier (f x')) empty
        by (force simp add: f-nxt-def)
      hence ⋀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 ⊗⊗ f x ⊗⊗ f' x = nxt t x ⊗⊗ f' x ⊗⊗ 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] , sym-
metric])
    next
    case False
      thus ?thesis by simp
    qed
  qed
qed


lemma ttree-rest-substitute:
  assumes ⋀ 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* ⊗⊗ *f x*)) **by** *auto*

from *Cons.prems*(*2*)
have (⋀*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*) ⊗⊗ *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*: *subsetD*[*OF ttree-restr-nxt-subset2*] *subsetD*[*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 subsetD*[*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 ∃ xs' ∈ paths t. substitute'' f T xs' xs
  then obtain xs' where substitute'' f T xs' xs and xs' ∈ paths t by auto
  thus xs ∈ 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) = ⋃((λ xs . Collect (substitute'' f T xs)) ‘ paths t)
  by (auto simp add: substitute-substitute'')

lemma ttree-rest-substitute2:
  assumes ⋀ x. carrier (f x) ⊆ 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 ∈ 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 (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 ∈ paths (substitute (f-nxt f T x) T (nxt t x ⊗⊗ f x)) by auto
    note this(2)
    moreover
    from  Cons.prems(2)
    have ⋀ x'. carrier (f-nxt f T x x') ⊆ 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' ∈ S] ∈ paths (substitute (f-nxt f T x) T (ttree-restr S (nxt t x ⊗⊗ f x)))
by (rule Cons.IH)
    hence ∗: [x'←xs . x' ∈ S] ∈ paths (substitute (f-nxt f T x) T (ttree-restr S (nxt t x ⊗⊗ f
x))) by (simp add: ttree-restr-both)
    show ?case
    proof (cases x ∈ 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: subsetD[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:  subsetD[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* ∈ *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''* ∈ *interleave* [*x'*←*xs'* . *x'* ∈ *S*] *zs*›
have *xs''* ∈ *interleave* [*x'*←*xs'* . *x'* ∈ *S*] [*x'*←*zs* . *x'* ∈ *S*] **by** *simp*
then obtain *xs'''* **where** *xs''* = [*x'*←*xs'''* . *x'* ∈ *S*] **and** *xs'''* ∈ *interleave xs' zs* **by** (*rule interleave-filter*)

from ‹*xs'''* ∈ *interleave xs' zs*›
have *l*: ⋀ *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'*←*xs'''* . *x'* ∈ *S*] *ys'* **by** *simp*
moreover
from *less.prems*(*2*)
have ⋀*xa*. *carrier* (*f-nxt f T x xa*) ⊆ *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 ∃ *ys''*. *ys'* = [*x'*←*ys''* . *x'* ∈ *S*] ∧ *substitute''* (*f-nxt f T x*) *T xs''' ys''*
by (*rule less.hyps*[*rotated*])
(*auto simp add*: ‹*ys* = - › ‹*xs* =-› ‹*x* ∈ *S*› ‹*xs''* = -›[*symmetric*] *l*)[*1*]
then obtain *ys''* **where** *ys'* = [*x'*←*ys''* . *x'* ∈ *S*] **and** *substitute''* (*f-nxt f T x*) *T xs'''* *ys''* **by** *blast*
hence *ys* = [*x'*←*x*#*ys''* . *x'* ∈ *S*] **using** ‹*x* ∈ *S*› ‹*ys* = -› **by** *simp*
moreover
from ‹*zs* ∈ *paths* (*f x*)› ‹*xs'''* ∈ *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'←xs' . x' ∈ S]) ys by simp
    hence ∃ ys'. ys = [x'←ys' . x' ∈ S] ∧ substitute'' f T xs' ys'
        by (rule less.hyps[OF - - less.prems(2,3), rotated]) (auto simp add: ‹xs =-› ‹x ∉
S›)
    then obtain ys' where ys = [x'←ys' . x' ∈ S] and substitute'' f T xs' ys' by auto

    from this(1)
    have ys = [x'←x#ys' . x' ∈ S] using ‹x ∉ S› ‹ys = -› by simp
    moreover
    have [simp]: f x = empty using ‹x ∉ 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
 then obtain xs'' where xs = filter (λ x'. x'∈S) xs'' and substitute'' f T xs' xs'' by auto
 from this(2) ‹xs' ∈ paths t›
 have xs'' ∈ paths (substitute f T t) by (auto simp add: substitute-substitute'')
 with ‹xs = -›
 show xs ∈ paths (ttree-restr S (substitute f T t))
  by (auto simp add: filter-paths-conv-free-restr[symmetric])
qed

end
```

9.2 TTree-HOLCF

```
theory TTree−HOLCF
imports TTree Launchbury.HOLCF−Utils Set−Cpo Launchbury.HOLCF−Join−Classes
begin

instantiation ttree :: (type) below
begin
 lift-definition below-ttree :: 'a ttree ⇒ 'a ttree ⇒ bool is (⊆).
instance..
end

lemma paths-mono: t ⊑ t' ⟹ paths t ⊑ paths t'
 by transfer (auto simp add: below-set-def)
```

lemma *paths-mono-iff*: *paths* $t \sqsubseteq$ *paths* $t' \longleftrightarrow t \sqsubseteq t'$
by *transfer* (*auto simp add*: *below-set-def*)

lemma *ttree-belowI*: $(\bigwedge xs.\ xs \in$ *paths* $t \Longrightarrow xs \in$ *paths* $t') \Longrightarrow t \sqsubseteq t'$
by *transfer auto*

lemma *paths-belowI*: $(\bigwedge x\ xs.\ x\#xs \in$ *paths* $t \Longrightarrow x\#xs \in$ *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* $\ldots = f\ (Either\ (range\ Y))$ **unfolding** *assms*(*1*)**..**

```
  also have Either (range Y) = lub (range Y) unfolding lub-is-either by simp
  finally
  show range (λi. f (Y i)) <<| f (⨆ i. Y i) by (metis is-lub-ttree)
qed

lemma ttree-contI2:
  assumes ⋀ x. paths (f x) = ⋃(t ‘ paths x)
  assumes [] ∈ t []
  shows cont f
proof(rule contI)
  fix Y :: nat ⇒ 'a ttree
  have paths (Either (range (λi. f (Y i)))) = insert [] (⋃x. paths (f (Y x)))
    by (simp add: paths-Either)
  also have ... = insert [] (⋃x. ⋃(t ‘ paths (Y x)))
    by (simp add: assms(1))
  also have ... = ⋃(t ‘ insert [] (⋃x. paths (Y x)))
    using assms(2) by (auto cong add: SUP-cong-simp)
  also have ... = ⋃(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 (⨆ 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* = λ*xs* . {*zs* . ∃ *ys*∈*paths y*. *zs* ∈ *xs* ⊗ *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* = λ*ys* . {*zs* . ∃ *xs*∈*paths y*. *zs* ∈ *xs* ⊗ *ys*}])
apply (*rule set-eqI*)
by (*auto intro*: *simp add*: *paths-both*)

lemma *cont-both*[*cont2cont*,*simp*]: *cont f* ⟹ *cont g* ⟹ *cont* (λ *x*. *f x* ⊗⊗ *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* = λ*xs* . (*if xs* ∈ *paths y then* {*xs*} *else* {})])
(*auto split*: *if-splits*)

lemma *cont-intersect2*:
cont (λ *x*. *intersect y x*)
by (*rule ttree-contI2* [**where** *t* = λ*xs* . (*if xs* ∈ *paths y then* {*xs*} *else* {})])
(*auto split*: *if-splits*)

lemma *cont-intersect*[*cont2cont*,*simp*]: *cont f* ⟹ *cont g* ⟹ *cont* (λ *x*. *f x* ∩∩ *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* = λ *xs*.{*filter* (λ *x'*. *x'* ≠ *x*) *xs*}])
(*transfer*, *auto*)

lemma *paths-many-calls-subset*:
t ⊑ *many-calls x* ⊗⊗ *without x t*
by (*metis* (*full-types*) *below-set-def paths-many-calls-subset paths-mono-iff*)

lemma *single-below*:
[*x*] ∈ *paths t* ⟹ *single x* ⊑ *t* **by** *transfer auto*

lemma *cont-ttree-restr*[*THEN cont-compose*, *cont2cont*,*simp*]: *cont* (*ttree-restr S*)
by (*rule ttree-contI2*[**where** $t = \lambda\ xs.\{filter\ (\lambda\ x'.\ x' \in S)\ xs\}$])
(*transfer*, *auto*)

lemmas *ttree-restr-mono* = *cont2monofunE*[*OF cont-ttree-restr*[*OF cont-id*]]

lemma *range-filter*[*simp*]: *range* (*filter P*) = {*xs*. *set xs* ⊆ *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* (=)) (*rel-fun* (*pcr-ttree* (=)) (*pcr-ttree* (=))) (∪) (⊔)
proof−
have (⊔) = ((⊕⊕) :: *'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 ⊔ 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′ ∈ carrier (fup·f·u) ⟷ (∃ u′. u = up·u′ ∧ x′ ∈ carrier (f·u′))
by (*cases u*) *auto*

end

10 Trace Tree Cardinality Analysis

10.1 AnalBinds

theory *AnalBinds*
imports *Launchbury.Terms Launchbury.HOLCF−Utils Launchbury.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* $\Gamma$ = *ae'* $f|`$ *domA* Γ
shows *AnalBinds* $\Gamma\cdot ae$ = *AnalBinds* $\Gamma\cdot ae'$
using *env-restr-eqD*[*OF assms*]
by (*induction* Γ *rule*: *AnalBinds.induct*) (*auto split*: *if-splits*)

lemma *AnalBinds-lookup*: (*AnalBinds* $\Gamma\cdot ae$) x = (*case map-of* Γ x *of Some e* $\Rightarrow$ *fup*$\cdot$(*exp e*)$\cdot$(*ae* x) | *None* $\Rightarrow$ $\bot$)
by (*induction* Γ *rule*: *AnalBinds.induct*) *auto*

lemma *AnalBinds-delete-bot*: *ae* $x = \bot \Longrightarrow$ *AnalBinds* (*delete* x Γ)$\cdot ae$ = *AnalBinds* $\Gamma\cdot ae$
by (*auto simp add*: *AnalBinds-lookup split*:*option.split simp add*: *delete-conv*)

lemma *AnalBinds-delete-below*: *AnalBinds* (*delete* x Γ)$\cdot ae \sqsubseteq$ *AnalBinds* $\Gamma\cdot ae$
by (*auto intro*: *fun-belowI simp add*: *AnalBinds-lookup split*:*option.split*)

lemma *AnalBinds-delete-lookup*[*simp*]: (*AnalBinds* (*delete* x Γ)$\cdot ae$) $x = \bot$
by (*auto simp add*: *AnalBinds-lookup split*:*option.split*)

lemma *AnalBinds-delete-to-fun-upd*: *AnalBinds* (*delete* x Γ)$\cdot ae$ = (*AnalBinds* $\Gamma\cdot ae$)($x := \bot$)
by (*auto simp add*: *AnalBinds-lookup split*:*option.split*)

lemma *edom-AnalBinds*: *edom* (*AnalBinds* $\Gamma\cdot ae$) $\subseteq$ *domA* $\Gamma \cap$ *edom ae*
by (*induction* Γ *rule*: *AnalBinds.induct*) (*auto simp add*: *edom-def*)

end

end

10.2 TTreeAnalysisSig

theory *TTreeAnalysisSig*
imports *Arity TTree−HOLCF AnalBinds*
begin

locale *TTreeAnalysis* =
fixes *Texp* :: *exp* $\Rightarrow$ *Arity* $\rightarrow$ *var ttree*
begin
sublocale *Texp*: *ExpAnalysis Texp*.
abbreviation *FBinds* == *Texp.AnalBinds*
end

end

10.3 Cardinality-Domain-Lists

theory *Cardinality−Domain−Lists*

imports *Launchbury.Vars Launchbury.Nominal−HOLCF Launchbury.Env Cardinality−Domain Set−Cpo Env−Set−Cpo*
begin

fun *no-call-in-path* **where**
  no-call-in-path x [] ⟷ *True*
| *no-call-in-path x* (*y*#*xs*) ⟷ *y* ≠ *x* ∧ *no-call-in-path x xs*

fun *one-call-in-path* **where**
  one-call-in-path x [] ⟷ *True*
| *one-call-in-path x* (*y*#*xs*) ⟷ (*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 ⟷ *x* ∉ *set p*
  by(*induction p*) *auto*

lemma *one-call-in-path-filter-conv*:
  one-call-in-path x p ⟷ *length* (*filter* (λ *x'*. *x'* = *x*) *p*) ≤ *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*) ⟷ (*no-call-in-path x p* ∨ *one-call-in-path x p* ∧ *hd p* = *x*)
  by(*induction p*) *auto*

lemma *no-imp-one*: *no-call-in-path x p* ⟹ *one-call-in-path x p*
  by (*induction p*) *auto*

lemma *one-imp-one-tail*: *one-call-in-path x p* ⟹ *one-call-in-path x* (*tl p*)
  by (*induction p*) (*auto split*: *if-splits intro*: *no-imp-one*)

lemma *more-than-one-setD*:
  ¬ *one-call-in-path x p* ⟹ *x* ∈ *set p*
  by (*induction p*) (*auto split*: *if-splits*)

lemma *no-call-in-path*[*eqvt*]: *no-call-in-path p x* ⟹ *no-call-in-path* (π ∙ *p*) (π ∙ *x*)
  by (*induction p x rule*: *no-call-in-path.induct*) *auto*

lemma *one-call-in-path*[*eqvt*]: *one-call-in-path p x* ⟹ *one-call-in-path* (π ∙ *p*) (π ∙ *x*)
  by (*induction p x rule*: *one-call-in-path.induct*) (*auto dest*: *no-call-in-path*)

definition *pathCard* :: *var list* ⇒ (*var* ⇒ *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* [] = ⊥
  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* ∈ *S*] ⟷ *no-call-in-path x xs* ∨ *x* ∉ *S*
by (*induction xs*) *auto*

lemma *one-call-in-path-filter*[*simp*]: *one-call-in-path x* [*x*←*xs* . *x* ∈ *S*] ⟷ *one-call-in-path x xs* ∨ *x* ∉ *S*
by (*induction xs*) *auto*

definition *pathsCard* :: *var list set* ⇒ (*var* ⇒ *two*)
where *pathsCard ps x* = (*if* (∀ *p* ∈ *ps*. *no-call-in-path x p*) *then none else* (*if* (∀ *p*∈*ps*. *one-call-in-path x p*) *then once else many*))

lemma *paths-Card-above*:
p ∈ *ps* ⟹ *pathCard p* ⊑ *pathsCard ps*
by (*rule fun-belowI*) (*auto simp add*: *pathsCard-def pathCard-def*)

lemma *pathsCard-below*:
assumes ⋀ *p*. *p* ∈ *ps* ⟹ *pathCard p* ⊑ *ce*
shows *pathsCard ps* ⊑ *ce*
proof(*rule fun-belowI*)
fix *x*
show *pathsCard ps x* ⊑ *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 ⊆ *ps'* ⟹ *pathsCard ps* ⊑ *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

10.4 TTreeAnalysisSpec

theory *TTreeAnalysisSpec*
imports *TTreeAnalysisSig AbstractAnalysisSpec Cardinality−Domain−Lists*
begin

locale *TTreeAnalysisCarrier = TTreeAnalysis + EdomAbstractAnalysis +*
assumes *carrier-Fexp*: *carrier (Texp e·a) = edom (Aexp e·a)*

locale *TTreeAnalysisSafe = TTreeAnalysisCarrier +*
assumes *Texp-App*: *many-calls x ⊗⊗ (Texp e)·(inc·a) ⊑ Texp (App e x)·a*
assumes *Texp-Lam*: *without y (Texp e·(pred·n)) ⊑ Texp (Lam [y]. e) · n*
assumes *Texp-subst*: *Texp (e[y::=x])·a ⊑ many-calls x ⊗⊗ without y ((Texp e)·a)*
assumes *Texp-Var*: *single v ⊑ Texp (Var v)·a*
assumes *Fun-repeatable*: *isVal e ⟹ repeatable (Texp e·0)*
assumes *Texp-IfThenElse*: *Texp scrut·0 ⊗⊗ (Texp e1·a ⊕⊕ Texp e2·a) ⊑ Texp (scrut ? e1 : e2)·a*

locale *TTreeAnalysisCardinalityHeap =*
TTreeAnalysisSafe + AbstractAnalysisLetSafe +
fixes *Theap :: heap ⇒ exp ⇒ Arity → var ttree*
assumes *carrier-Fheap*: *carrier (Theap Γ e·a) = edom (Aheap Γ e·a)*

assumes *Theap-thunk*: $x \in thunks\ \Gamma \Longrightarrow p \in paths\ (Theap\ \Gamma\ e{\cdot}a) \Longrightarrow \neg\ one\text{-}call\text{-}in\text{-}path\ x\ p \Longrightarrow (Aheap\ \Gamma\ e{\cdot}a)\ x = up{\cdot}0$
assumes *Theap-substitute*: $ttree\text{-}restr\ (domA\ \Delta)\ (substitute\ (FBinds\ \Delta{\cdot}(Aheap\ \Delta\ e{\cdot}a))\ (thunks\ \Delta)\ (Texp\ e{\cdot}a)) \sqsubseteq Theap\ \Delta\ e{\cdot}a$
assumes *Texp-Let*: $ttree\text{-}restr\ (-\ domA\ \Delta)\ (substitute\ (FBinds\ \Delta{\cdot}(Aheap\ \Delta\ e{\cdot}a))\ (thunks\ \Delta)\ (Texp\ e{\cdot}a)) \sqsubseteq Texp\ (Terms.Let\ \Delta\ e){\cdot}a$

end

10.5 TTreeImplCardinality

theory *TTreeImplCardinality*
imports *TTreeAnalysisSig CardinalityAnalysisSig Cardinality−Domain−Lists*
begin

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* - [] = ⊥
  | $Fstack\ (a\#as)\ (Alts\ e1\ e2\ \#\ S) = (Texp\ e1{\cdot}a \oplus\oplus Texp\ e2{\cdot}a) \otimes\otimes Fstack\ as\ S$
  | $Fstack\ as\ (Arg\ x\ \#\ S) = many\text{-}calls\ x \otimes\otimes Fstack\ as\ S$
  | $Fstack\ as\ (-\ \#\ S) = Fstack\ as\ S$

fun *prognosis* :: *AEnv* ⇒ *Arity list* ⇒ *Arity* ⇒ *conf* ⇒ *var* ⇒ *two*
  where $prognosis\ ae\ as\ a\ (\Gamma,\ e,\ S) = pathsCard\ (paths\ (substitute\ (FBinds\ \Gamma{\cdot}ae)\ (thunks\ \Gamma)\ (Texp\ e{\cdot}a \otimes\otimes Fstack\ as\ S)))$
end

end

10.6 TTreeImplCardinalitySafe

theory *TTreeImplCardinalitySafe*
imports *TTreeImplCardinality TTreeAnalysisSpec CardinalityAnalysisSpec*
begin

lemma *pathsCard-paths-nxt*: *pathsCard* (*paths* (*nxt f x*)) ⊑ *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* ⟹ *x* ∉ *carrier t*
by *transfer* (*auto dest*: *pathCards-noneD*)

lemma *const-on-edom-disj*: *const-on f S empty* ⟷ *edom f* ∩ *S* = {}
by (*auto simp add*: *empty-is-bottom edom-def*)

context *TTreeAnalysisCarrier*
begin
lemma *carrier-Fstack*: *carrier* (*Fstack as S*) ⊆ *fv S*
by (*induction S rule*: *Fstack.induct*)
(*auto simp add*: *empty-is-bottom*[*symmetric*] *carrier-Fexp dest*!: *subsetD*[*OF Aexp-edom*])

lemma *carrier-FBinds*: *carrier* ((*FBinds* Γ·*ae*) *x*) ⊆ *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*!: *subsetD*[*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* ∈ *ap S*
hence [*x*,*x*] ∈ *paths* (*Fstack as S*)
by (*induction S rule*: *Fstack.induct*)
(*auto 4 4 intro*: *subsetD*[*OF both-contains-arg1*] *subsetD*[*OF both-contains-arg2*] *paths-Cons-nxt*)

hence $[x,x] \in$ *paths* (*Texp e·a* $\otimes\otimes$ *Fstack as S*)
by (*rule subsetD*[*OF both-contains-arg2*])
hence $[x,x] \in$ *paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp e·a* $\otimes\otimes$ *Fstack as S*))
by (*rule subsetD*[*OF substitute-contains-arg*])
hence *pathCard* $[x,x]$ $x \sqsubseteq$ *pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp e·a* $\otimes\otimes$ *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*) $\sqsubseteq$ *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* = $\bot$

hence *FBinds* (*delete x* Γ)·*ae* = *FBinds* Γ·*ae* **by** (*rule Texp.AnalBinds-delete-bot*)
moreover
hence ((*FBinds* Γ·*ae*) *x*) = $\bot$ **by** (*metis Texp.AnalBinds-delete-lookup*)
ultimately
show *prognosis ae as a* (Γ, *e*, *S*) $\sqsubseteq$ *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* $\sqsubseteq$ (*pathCard* [*x*]) *x* **by** (*simp add*: *two-add-simp*)
also have *pathCard* [*x*] $\sqsubseteq$ *pathsCard* ({[],[*x*]})
by (*rule paths-Card-above*) *simp*
also have ... = *pathsCard* (*paths* (*single x*)) **by** *simp*
also have *single x* $\sqsubseteq$ (*Texp* (*Var x*)·*a*) **by** (*rule Texp-Var*)
also have ... $\sqsubseteq$ *Texp* (*Var x*)·*a* $\otimes\otimes$ *Fstack as S* **by** (*rule both-above-arg1*)
also have ... $\sqsubseteq$ *substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp* (*Var x*)·*a* $\otimes\otimes$ *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* $\sqsubseteq$ *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·(inc·a)* ⊗⊗ *many-calls x* ⊗⊗ *Fstack as S* = *many-calls x* ⊗⊗ (*Texp e*)·(*inc·a*) ⊗⊗ *Fstack as S*
by (*metis both-assoc both-comm*)
thus *prognosis ae as* (*inc·a*) (Γ, *e*, *Arg x* # *S*) ⊑ *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*]·(*pred·a*) ⊑ *many-calls x* ⊗⊗ *Texp* (*Lam* [*y*]. *e*)·*a*
by (*rule below-trans*[*OF Texp-subst both-mono2′*[*OF Texp-Lam*]])
moreover have *Texp* (*Lam* [*y*]. *e*)·*a* ⊗⊗ *many-calls x* ⊗⊗ *Fstack as S* = *many-calls x* ⊗⊗ *Texp* (*Lam* [*y*]. *e*)·*a* ⊗⊗ *Fstack as S*
by (*metis both-assoc both-comm*)
ultimately
show *prognosis ae as* (*pred·a*) (Γ, *e*[*y*::=*x*], *S*) ⊑ *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·u*

assume *isVal e*
hence *x* ∉ *thunks* Γ **using** ‹*map-of* Γ *x* = *Some e*› **by** (*metis thunksE*)
hence [*simp*]: *f-nxt* (*FBinds* Γ·*ae*) (*thunks* Γ) *x* = *FBinds* Γ·*ae* **by** (*auto simp add*: *f-nxt-def*)

have *prognosis ae as u* (Γ, *e*, *S*) = *pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*Texp e·u* ⊗⊗ *Fstack as S*)))
by *simp*
also have ... = *pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*nxt* (*single x*) *x* ⊗⊗ *Texp e·u* ⊗⊗ *Fstack as S*)))
by *simp*
also have ... = *pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) ((*nxt* (*single x*) *x* ⊗⊗ *Fstack as S*) ⊗⊗ *Texp e·u*)))
by (*metis both-assoc both-comm*)
also have ... ⊑ *pathsCard* (*paths* (*substitute* (*FBinds* 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 ... ⊑ *record-call x* ·(*pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*single x* ⊗⊗ *Fstack as S*))))

by (*rule pathsCard-paths-nxt*)
also have ... ⊑ *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*) ⊑ *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 ... ⊑ *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 ... ⊑ *record-call x ·*(*pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*single x* ⊗⊗ *Fstack as S*))))
by (*rule pathsCard-paths-nxt*)
also have ... ⊑ *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*) ⊑ *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*]: $x \notin$ *domA* Γ

have [*simp*]: *thunks* ((*x*, *e*) # Γ) = *thunks* Γ
using ‹*isVal e*›
by (*auto simp add*: *thunks-Cons dest*: *subsetD*[*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*: *subsetD*[*OF Aexp-edom*] *subsetD*[*OF carrier-Fstack*]
subsetD[*OF ap-fv-subset*] *subsetD*[*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* $\Delta \sharp*$ Γ›] *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* $\Delta \sharp*$ *S*›]
‹*edom ae* $\subseteq$ *domA* $\Gamma \cup$ *upds S*› *ups-fv-subset*[*of S*]
by *auto*

have *const-on1*: $\bigwedge$ *x*. *const-on* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*carrier* ((*FBinds* Γ·*ae*) *x*)) *empty*
unfolding *const-on-edom-disj* **using** *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* $\Delta \sharp*$ Γ›]
by (*auto dest*!: *subsetD*[*OF carrier-FBinds*] *subsetD*[*OF Texp.edom-AnalBinds*])
have *const-on2*: *const-on* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*carrier* (*Fstack as S*)) *empty*
unfolding *const-on-edom-disj* **using** *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* $\Delta \sharp*$ *S*›]
by (*auto dest*!: *subsetD*[*OF carrier-FBinds*] *subsetD*[*OF carrier-Fstack*] *subsetD*[*OF Texp.edom-AnalBinds*] *subsetD*[*OF ap-fv-subset*])
have *const-on3*: *const-on* (*FBinds* Γ·*ae*) (− (− *domA* Δ)) *TTree.empty*
and *const-on4*: *const-on* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*domA* Γ) *TTree.empty*
unfolding *const-on-edom-disj* **using** *fresh-distinct*[*OF* ‹*atom* ‘ *domA* $\Delta \sharp*$ Γ›]
by (*auto dest*!: *subsetD*[*OF Texp.edom-AnalBinds*])

have *disj1*: $\bigwedge$ *x*. *carrier* ((*FBinds* Γ·*ae*) *x*) $\cap$ *domA* $\Delta = \{\}$
using *fresh-distinct-fv*[*OF* ‹*atom* ‘ *domA* $\Delta \sharp*$ Γ›]
by (*auto dest*: *subsetD*[*OF carrier-FBinds*])
hence *disj1′*: $\bigwedge$ *x*. *carrier* ((*FBinds* Γ·*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* $\Delta \sharp*$ *S*›] **by** (*auto dest*!: *subsetD*[*OF carrier-Fstack*])
hence *disj2′*: *carrier* (*Fstack as S*) $\subseteq$ − *domA* Δ **by** *auto*

{
fix *x*
have (*FBinds* (Δ @ Γ)·(*ae* ⊔ *Aheap* Δ *e*·*a*)) *x* = (*FBinds* Γ·*ae*) *x* ⊗⊗ (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) *x*
proof (*cases x* ∈ *domA* Δ)
case *True*
have *map-of* Γ *x* = *None* **using** *True fresh-distinct*[*OF* ‹*atom* ‘ *domA* $\Delta \sharp*$ Γ›] **by** (*metis disjoint-iff-not-equal domA-def map-of-eq-None-iff*)
moreover
have *ae x* = ⊥ **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*·*a*) *x* = ⊥ **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)·(*Aheap* Δ *e*·*a* ⊔ *ae*)) (*thunks* (Δ @ Γ)) (*Texp e*·*a* ⊗⊗ *Fstack as S*)))
= *pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* (Δ @ Γ)) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* (Δ @ Γ)) (*Texp e*·*a* ⊗⊗ *Fstack as S*))))
by (*simp add*: *substitute-substitute*[*OF const-on1*] *FBinds*)
also have *substitute* (*FBinds* Γ·*ae*) (*thunks* (Δ @ Γ)) = *substitute* (*FBinds* Γ·*ae*) (*thunks* Γ)
apply (*rule substitute-cong-T*)
using *const-on3*
by (*auto dest*: *subsetD*[*OF thunks-domA*])
also have *substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* (Δ @ Γ)) = *substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ)
apply (*rule substitute-cong-T*)
using *const-on4*
by (*auto dest*: *subsetD*[*OF thunks-domA*])
also have *substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a* ⊗⊗ *Fstack as S*) = *substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*) ⊗⊗ *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* Γ·*ae*) (*thunks* Γ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*) ⊗⊗ *Fstack as S*))) *f*|‘ *domA* Δ
= *pathsCard* (*paths* (*ttree-restr* (*domA* Δ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*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* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)) ⊑ *Theap* Δ *e*·*a* **by** (*rule Theap-substitute*)
also
have *pathsCard* (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*) ⊗⊗ *Fstack as S*))) *f*|‘ (− *domA* Δ) =
pathsCard (*paths* (*substitute* (*FBinds* Γ·*ae*) (*thunks* Γ) (*ttree-restr* (− *domA* Δ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)) ⊗⊗ *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* Δ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp* *e*·*a*)) ⊑ *Texp* (*Terms.Let* Δ *e*)·*a* **by** (*rule Texp-Let*)
finally
show *prognosis* (*Aheap* Δ *e*·*a* ⊔ *ae*) *as a* (Δ @ Γ, *e*, *S*) ⊑ *cHeap* Δ *e*·*a* ⊔ *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

11 Co-Call Graphs

11.1 CoCallGraph

theory *CoCallGraph*
imports *Launchbury.Vars Launchbury.HOLCF−Join−Classes Launchbury.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** (⊆).
instance
apply *standard*
apply ((*transfer*, *auto*)+)
done
end

lift-definition *coCallsLub* :: *CoCalls set* ⇒ *CoCalls* **is** λ *S*. ⋃ *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 ∀ x ∈ S. x ⊑ u by (auto dest: is-ubD)
  thus coCallsLub S ⊑ u by transfer auto
qed

instance CoCalls :: cpo
proof
  fix S :: nat ⇒ CoCalls
  show ∃ 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 ⇒ CoCalls ⇒ bool is (λ S x . x = Union S).

lemma ccis-lubTransfer[transfer-rule]: (rel-set pcr-CoCalls ===> pcr-CoCalls ===> (=)) (λ
S x . x = Union S) (<<|)
proof−
  have ⋀ x xa . is-cc-lub x xa ⟷ xa = coCallsLub x by transfer auto
  hence is-cc-lub = (<<|)
  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 ⇒ CoCalls ⇒ CoCalls is (∪)
   by (rule sym-Un)

lemma ccJoinTransfer[transfer-rule]: (pcr-CoCalls ===> pcr-CoCalls ===> pcr-CoCalls)
(∪) (⊔)
proof−
  have (⊔) = 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*:
(⋀ *x y*. *x*−−*y*∈*G* ⟹ *x*−−*y*∈*G′*) ⟹ *G* ⊑ *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*) ⊆ *ccField S* − {*x*}
by *transfer* (*auto simp add*: *Field-def*)

lift-definition *ccProd* :: *var set* ⇒ *var set* ⇒ *CoCalls* (**infixr** ‹*G*×› *90*)
is λ *S1 S2*. *S1* × *S2* ∪ *S2* × *S1*
by (*auto intro*!: *symI elim*: *symE*)

lemma *ccProd-empty*[*simp*]: {} *G*× *S* = ⊥ **by** *transfer auto*

lemma *ccProd-empty'*[*simp*]: *S G*× {} = ⊥ **by** *transfer auto*

lemma *ccProd-union2*[*simp*]: *S G*× (*S'* ∪ *S''*) = *S G*× *S'* ⊔ *S G*× *S''*
by *transfer auto*

lemma *ccProd-Union2*[*simp*]: *S G*× ⋃*S'* = (⨆ *X*∈*S'*. *ccProd S X*)
by *transfer auto*

lemma *ccProd-Union2'*[*simp*]: *S G*× (⋃*X*∈*S'*. *f X*) = (⨆ *X*∈*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'* ∪ *S''*) *G*× *S* = *S' G*× *S* ⊔ *S'' G*× *S*
by *transfer auto*

lemma *ccProd-insert2*: $S\ G\times\ insert\ x\ S' = S\ G\times \{x\} \sqcup S\ G\times\ S'$
by *transfer auto*

lemma *ccProd-insert1*: $insert\ x\ S'\ G\times\ S = \{x\}\ G\times\ S \sqcup S'\ G\times\ S$
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* ⇒ *CoCalls* ⇒ *CoCalls*
is $\lambda\ S.\ Set.filter\ (\lambda\ (x,y)\ .\ x \in S \wedge y \in S)$
by (*auto intro*!: *symI elim*: *symE*)

abbreviation *cc-restr-sym* (**infixl** ‹$G|`$›  *110*) **where** $G\ G|`\ S \equiv$ *cc-restr S G*

lemma *elem-cc-restr*[*simp*]: $x{-}{-}y{\in}(G\ G|`\ S) = (x{-}{-}y{\in}G \wedge x \in S \wedge y \in S)$
by *transfer auto*

lemma *ccField-cc-restr*: $ccField\ (G\ G|`\ S) \subseteq ccField\ G \cap S$
by *transfer* (*auto simp add*: *Field-def*)

lemma *cc-restr-empty*: $ccField\ G \subseteq -\ S \Longrightarrow G\ G|`\ S = \bot$
apply *transfer*
apply (*auto simp add*: *Field-def*)
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*: *subsetD*)

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* ⊔ *G'*) = *cc-restr S G* ⊔ *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 ⊆ *S'* ⟹ *cc-restr S G* ⊑ *cc-restr S' G* **by** *transfer auto*

lemma *cc-restr-mono2*:
G ⊑ *G'* ⟹ *cc-restr S G* ⊑ *cc-restr S G'* **by** *transfer auto*

lemma *cc-restr-below-arg*:
cc-restr S G ⊑ *G* **by** *transfer auto*

lemma *cc-restr-lub*[*simp*]:
cc-restr S (*lub X*) = (⨆ *G*∈*X*. *cc-restr S G*) **by** *transfer auto*

lemma *elem-to-ccField*: *x*−−*y*∈*G* ⟹ *x* ∈ *ccField G* ∧ *y* ∈ *ccField G*
by *transfer* (*auto simp add*: *Field-def*)

lemma *ccField-to-elem*: *x* ∈ *ccField G* ⟹ ∃ *y*. *x*−−*y*∈*G*
by *transfer* (*auto simp add*: *Field-def dest*: *symD*)

lemma *cc-restr-intersect*: *ccField G* ∩ ((*S* − *S'*) ∪ (*S'* − *S*)) = {} ⟹ *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* ∩ *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*$_1$ ∩ *S*) (*S*$_2$ ∩ *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*:)

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 (λ*y*. *ccNeighbors x y*)
apply (*rule set-contI*)
apply (*thin-tac chain* -)
apply *transfer*
apply *auto*
done

lemma *ccNeighbors-join*[*simp*]: *ccNeighbors x* (*G* ⊔ *G'*) = *ccNeighbors x G* ∪ *ccNeighbors x G'*
by *transfer auto*

lemma *ccNeighbors-ccProd*:
ccNeighbors x (*ccProd S S'*) = (*if x* ∈ *S then S' else* {}) ∪ (*if x* ∈ *S' then S else* {})
by *transfer auto*

lemma *ccNeighbors-ccSquare*:
ccNeighbors x (*ccSquare S*) = (*if x* ∈ *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* ∈ *S then ccNeighbors x G* ∩ *S else* {})
by *transfer auto*

lemma *ccNeighbors-mono*:
G ⊑ *G'* ⟹ *ccNeighbors x G* ⊆ *ccNeighbors x G'*
by *transfer auto*

lemma *subset-ccNeighbors*:
S ⊆ *ccNeighbors x G* ⟷ *ccProd* {*x*} *S* ⊑ *G*
by *transfer* (*auto simp add*: *sym-def*)

lemma *elem-ccNeighbors*[*simp*]:
y ∈ *ccNeighbors x G* ⟷ (*y*−−*x*∈*G*)
by *transfer* (*auto simp add*: *sym-def*)

lemma *ccNeighbors-ccField*:
ccNeighbors x G ⊆ *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 ⟹ *ccLinear S* (*cc-restr S'* *G*)
unfolding *ccLinear-def* **by** *transfer auto*

lemma *ccLinear-join*[*simp*]:
ccLinear S (*G* ⊔ *G'*) ⟷ *ccLinear S G* ∧ *ccLinear S G'*
unfolding *ccLinear-def*
by *transfer auto*

lemma *ccLinear-ccProd*[*simp*]:
ccLinear S (*ccProd S*$_1$ *S*$_2$) ⟷ *S*$_1$ ∩ *S* = {} ∨ *S*$_2$ ∩ *S* = {}
unfolding *ccLinear-def*
by *transfer auto*

lemma *ccLinear-mono1*: *ccLinear S' G* ⟹ *S* ⊆ *S'* ⟹ *ccLinear S G*
unfolding *ccLinear-def*
by *transfer auto*

lemma *ccLinear-mono2*: *ccLinear S G'* ⟹ *G* ⊑ *G'* ⟹ *ccLinear S G*
unfolding *ccLinear-def*
by *transfer auto*

lemma *ccField-join*[*simp*]:
ccField (*G* ⊔ *G'*) = *ccField G* ∪ *ccField G'* **by** *transfer auto*

lemma *ccField-lub*[*simp*]:
ccField (*lub S*) = ⋃(*ccField* ‘ *S*) **by** *transfer auto*

lemma *ccField-ccProd*:
ccField (*ccProd S S'*) = (*if S* = {} *then* {} *else if S'* = {} *then* {} *else* *S* ∪ *S'*)
by *transfer* (*auto simp add*: *Field-def*)

lemma *ccField-ccProd-subset*:
ccField (*ccProd S S'*) ⊆ *S* ∪ *S'*
by (*simp add*: *ccField-ccProd*)

lemma *cont-ccField*[*THEN cont-compose*, *simp*, *cont2cont*]:
cont ccField
by (*rule set-contI*) *auto*

end

11.2 CoCallGraph-Nominal

theory *CoCallGraph−Nominal*
imports *CoCallGraph Launchbury.Nominal−HOLCF*
begin

instantiation *CoCalls* :: *pt*
begin
lift-definition *permute-CoCalls* :: *perm* ⇒ *CoCalls* ⇒ *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* ♯∗ *S* **and** [*simp*]: *finite S*
shows *cc-restr S* (*p* · *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*]: π · (*x*−−*y*∈*G*) = (π·*x*)−−(π·*y*)∈(π·*G*)
by *transfer auto*
lemma *cc-restr-eqvt*[*eqvt*]: π · *cc-restr S G* = *cc-restr* (π · *S*) (π · *G*)
by *transfer* (*perm-simp*, *rule*)
lemma *ccProd-eqvt*[*eqvt*]: π · *ccProd S S*′ = *ccProd* (π · *S*) (π · *S*′)
by *transfer* (*perm-simp*, *rule*)
lemma *ccSquare-eqvt*[*eqvt*]: π · *ccSquare S* = *ccSquare* (π · *S*)
unfolding *ccSquare-def*
by *perm-simp rule*
lemma *ccNeighbors-eqvt*[*eqvt*]: π · *ccNeighbors S G* = *ccNeighbors* (π · *S*) (π · *G*)
by *transfer* (*perm-simp*, *rule*)

end

12 Co-Call Cardinality Analysis

12.1 CoCallAnalysisSig

theory *CoCallAnalysisSig*
imports *Launchbury.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* (‹$\mathcal{G}^\perp$_›)
    where ${\mathcal{G}^\perp}_a \equiv (\lambda e.\ fup{\cdot}(ccExp\ e){\cdot}a)$
end

locale *CoCallAnalyisHeap* =
  fixes *ccHeap* :: *heap* ⇒ *exp* ⇒ *Arity* → *CoCalls*

end

12.2 CoCallAnalysisBinds

theory *CoCallAnalysisBinds*
imports *CoCallAnalysisSig AEnv ListUtilsHOLCF ArityNominal CoCallGraphNominal*
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*: *subsetD*[*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*: *subsetD*[*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* ⊔ *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 [] = ⊥
unfolding *ccBinds-def* **by** *simp*

lemma *ccBinds-Cons*[*simp*]:
ccBinds ((*x*,*e*)#Γ) = *ccBind x e* ⊔ *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* ⟹ *ccBind x e*·*ae* ⊑ (*ccBinds* Γ·*ae*)
by (*auto simp add*: *ccBinds-eq*)

lemma *ccField-ccBinds*: *ccField* (*ccBinds* Γ·(*ae*,*G*)) ⊆ *fv* Γ

by (*auto simp add*: *ccBinds-eq dest*: *subsetD*[*OF ccField-ccBind*] *intro*: *subsetD*[*OF map-of-Some-fv-subset*])

definition *ccBindsExtra* :: *heap* ⇒ ((*AEnv* × *CoCalls*) → *CoCalls*)
where *ccBindsExtra* Γ = (Λ *i*. *snd i* ⊔ *ccBinds* Γ · *i* ⊔ (⨆*x*↦*e*∈*map-of* Γ. *ccProd* (*fv e*) (*ccNeighbors x* (*snd i*))))

lemma *ccBindsExtra-simp*: *ccBindsExtra* Γ · *i* =*snd i* ⊔ *ccBinds* Γ · *i* ⊔ (⨆*x*↦*e*∈*map-of* Γ. *ccProd* (*fv e*) (*ccNeighbors x* (*snd i*)))
unfolding *ccBindsExtra-def* **by** *simp*

lemma *ccBindsExtra-eq*: *ccBindsExtra* Γ·(*ae*,*G*) =
G ⊔ *ccBinds* Γ·(*ae*,*G*) ⊔ (⨆*x*↦*e*∈*map-of* Γ. *fv e* *G*× *ccNeighbors x G*)
unfolding *ccBindsExtra-def* **by** *simp*

lemma *ccBindsExtra-strict*[*simp*]: *ccBindsExtra* Γ · ⊥ = ⊥
by (*auto simp add*: *ccBindsExtra-simp inst-prod-pcpo simp del*: *Pair-strict*)

lemma *ccField-ccBindsExtra*:
ccField (*ccBindsExtra* Γ·(*ae*,*G*)) ⊆ *fv* Γ ∪ *ccField G*
by (*auto simp add*: *ccBindsExtra-simp elem-to-ccField*
dest!: *subsetD*[*OF ccField-ccBinds*] *subsetD*[*OF ccField-ccProd-subset*] *map-of-Some-fv-subset*)

end

lemma *ccBind-eqvt*[*eqvt*]: π · (*CoCallAnalysis.ccBind cccExp x e*) = *CoCallAnalysis.ccBind* (π · *cccExp*) (π · *x*) (π · *e*)
proof−
{
fix π *ae G*
have π · ((*CoCallAnalysis.ccBind cccExp x e*) · (*ae*,*G*)) = *CoCallAnalysis.ccBind* (π · *cccExp*) (π · *x*) (π · *e*) · (π · *ae*, π · *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*]: π · (*CoCallAnalysis.ccBinds cccExp* Γ) = *CoCallAnalysis.ccBinds* (π · *cccExp*) (π · Γ)
apply (*rule cfun-eqvtI*)
unfolding *CoCallAnalysis.ccBinds-eq*
apply (*perm-simp*)
apply *rule*
done

lemma *ccBindsExtra-eqvt*[*eqvt*]: π · (*CoCallAnalysis.ccBindsExtra cccExp* Γ) = *CoCallAnalysis.ccBindsExtra* (π · *cccExp*) (π · Γ)
by (*rule cfun-eqvtI*) (*simp add*: *CoCallAnalysis.ccBindsExtra-def*)

lemma *ccBind-cong*[*fundef-cong*]:
cccexp1 e = cccexp2 e ⟹ 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*]:
⟦ (⋀ *e*. *e* ∈ *snd* ' *set heap2* ⟹ *cccexp1 e = cccexp2 e*); *heap1 = heap2* ⟧
⟹ *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*]:
⟦ (⋀ *e*. *e* ∈ *snd* ' *set heap2* ⟹ *cccexp1 e = cccexp2 e*); *heap1 = heap2* ⟧
⟹ *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

12.3 CoCallAritySig

theory *CoCallAritySig*
imports *ArityAnalysisSig CoCallAnalysisSig*
begin

locale *CoCallArity = CoCallAnalysis + ArityAnalysis*

end

12.4 CoCallAnalysisSpec

theory *CoCallAnalysisSpec*
imports *CoCallAritySig ArityAnalysisSpec*
begin

locale *CoCallArityEdom = CoCallArity + EdomArityAnalysis*

locale *CoCallAritySafe* = *CoCallArity* + *CoCallAnalyisHeap* + *ArityAnalysisLetSafe* +
  assumes *ccExp-App*: *ccExp e·(inc·a)* ⊔ *ccProd* {*x*} (*insert x* (*fv e*)) ⊑ *ccExp* (*App e x*)·*a*
  assumes *ccExp-Lam*: *cc-restr* (*fv* (*Lam* [*y*]. *e*)) (*ccExp e·*(*pred·n*)) ⊑ *ccExp* (*Lam* [*y*]. *e*)·*n*
  assumes *ccExp-subst*: *x* ∉ *S* ⟹ *y* ∉ *S* ⟹ *cc-restr S* (*ccExp e*[*y*::=*x*]·*a*) ⊑ *cc-restr S* (*ccExp e·a*)
  assumes *ccExp-pap*: *isVal e* ⟹ *ccExp e·0* = *ccSquare* (*fv e*)
  assumes *ccExp-Let*: *cc-restr* (−*domA* Γ) (*ccHeap* Γ *e·a*) ⊑ *ccExp* (*Let* Γ *e*)·*a*
  assumes *ccExp-IfThenElse*: *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*

  assumes *ccHeap-Exp*: *ccExp e·a* ⊑ *ccHeap* Δ *e·a*
  assumes *ccHeap-Heap*: *map-of* Δ *x* = *Some e′* ⟹ (*Aheap* Δ *e·a*) *x*= *up·a′* ⟹ *ccExp e′·a′* ⊑ *ccHeap* Δ *e·a*
  assumes *ccHeap-Extra-Edges*:
    map-of Δ *x* = *Some e′* ⟹ (*Aheap* Δ *e·a*) *x* = *up·a′* ⟹ *ccProd* (*fv e′*) (*ccNeighbors x* (*ccHeap* Δ *e·a*) − {*x*} ∩ *thunks* Δ) ⊑ *ccHeap* Δ *e·a*

  assumes *aHeap-thunks-rec*: ¬ *nonrec* Γ ⟹ *x* ∈ *thunks* Γ ⟹ *x* ∈ *edom* (*Aheap* Γ *e·a*) ⟹ (*Aheap* Γ *e·a*) *x* = *up·0*
  assumes *aHeap-thunks-nonrec*: *nonrec* Γ ⟹ *x* ∈ *thunks* Γ ⟹ *x*−−*x* ∈ *ccExp e·a* ⟹ (*Aheap* Γ *e·a*) *x* = *up·0*

end

12.5 CoCallFix

theory *CoCallFix*
imports *CoCallAnalysisSig CoCallAnalysisBinds ArityAnalysisSig Launchbury.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** $\mathcal{A}_a$ ≡ (λ*e*. *Aexp e·a*)
abbreviation *Aexp-bot-syn′* (‹$\mathcal{A}^\perp$_›) **where** $\mathcal{A}^\perp{}_a$ ≡ (λ*e*. *fup*·(*Aexp e*)·*a*)

lemma *Aexp-eq*:
  $\mathcal{A}_a$ *e* = *fst* (*cccExp e* · *a*)
  unfolding *Aexp-def* **by** (*rule beta-cfun*) (*intro cont2cont*)

lemma *fup-Aexp-eq*:
fup·(*Aexp e*)·*a* = *fst* (*fup*·(*cccExp e*)·*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·*a* = *snd* (*cccExp e* · *a*)
unfolding *CCexp-def* **by** (*rule beta-cfun*) (*intro cont2cont*)

lemma *fup-CCexp-eq*:
fup·(*CCexp e*)·*a* = *snd* (*fup*·(*cccExp e*)·*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 ⋀ *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 ⋀ *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* = (⊔)] *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* $k \in S$)
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* $\Gamma \subseteq S$
shows *cc-restr S* (*ccBindsExtra* $\Gamma\cdot$(*ae*, *G*)) = *cc-restr S* (*ccBindsExtra* $\Gamma\cdot$(*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** $f = (\sqcup)$] *refl arg-cong*[*OF mapCollect-cong*])
apply (*subgoal-tac* $k \in S$)
apply *simp*
apply (*metis contra-subsetD domI dom-map-of-conv-domA*)
apply (*subgoal-tac* $k \in S$)
apply *simp*
apply (*metis contra-subsetD domI dom-map-of-conv-domA*)
done

lemma *CCfix-restr*:
assumes *domA* $\Gamma \subseteq S$
shows *cc-restr S* (*CCfix* $\Gamma\cdot$(*ae*, *G*)) = *cc-restr S* (*CCfix* $\Gamma\cdot$(*ae f*|‘ *S*, *cc-restr S G*))
unfolding *CCfix-def*
apply *simp*
apply (*rule parallel-fix-ind*[**where** $P = \lambda\ x\ y\ .$ *cc-restr S x* = *cc-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*!: *subsetD*[*OF ccField-ccBindsExtra*])
done

lemma *CCfix-restr-subst'*:
assumes $\bigwedge x'\ e\ a.\ (x',e) \in set\ \Gamma \Longrightarrow$ *cc-restr S* (*CCexp* $e[x::=y]\cdot a$) = *cc-restr S* (*CCexp* $e\cdot a$)
assumes $x \notin S$
assumes $y \notin S$
assumes *domA* $\Gamma \subseteq S$
shows *cc-restr S* (*CCfix* $\Gamma[x::h=y]\cdot(ae,\ G)$) = *cc-restr S* (*CCfix* $\Gamma\cdot(ae\ f|`\ S,$ *cc-restr S G*))
unfolding *CCfix-def*
apply *simp*
apply (*rule parallel-fix-ind*[**where** $P = \lambda\ x\ y\ .$ *cc-restr S x* = *cc-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*]: $\pi \cdot$ (*CoCallArityAnalysis.Aexp cccExp e*) = *CoCallArityAnalysis.Aexp* ($\pi \cdot$ *cccExp*) ($\pi \cdot$ *e*)
apply (*rule cfun-eqvtI*) **unfolding** *CoCallArityAnalysis.Aexp-eq* **by** *perm-simp rule*

lemma *CCexp-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.CCexp cccExp e*) = *CoCallArityAnalysis.CCexp* ($\pi \cdot$ *cccExp*) ($\pi \cdot$ *e*)
apply (*rule cfun-eqvtI*) **unfolding** *CoCallArityAnalysis.CCexp-eq* **by** *perm-simp rule*

lemma *CCfix-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.CCfix cccExp* Γ) = *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*]:
$[\![(\bigwedge e.\ e \in snd\ `\ set\ heap2 \Longrightarrow cccexp1\ e = cccexp2\ e);\ heap1 = heap2\ ]\!]$
$\Longrightarrow$ *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*) $\rightarrow$ (*AEnv* $\times$ *CoCalls*))
where *cccFix* Γ = ($\Lambda\ i.$ (*Afix* $\Gamma\cdot$(*fst i* $\sqcup$ (λ-.*up*·*0*) *f*|` *thunks* Γ), *CCfix* $\Gamma\cdot$(*Afix* $\Gamma\cdot$(*fst i* $\sqcup$

$(\lambda\text{-}.up{\cdot}0)\ f|`(thunks\ \Gamma)),\ snd\ i)))$

lemma *cccFix-eq*:
cccFix $\Gamma{\cdot}i$ = (*Afix* $\Gamma{\cdot}$(*fst i* ⊔ $(\lambda\text{-}.up{\cdot}0)\ f|`thunks\ \Gamma)$, *CCfix* $\Gamma{\cdot}$(*Afix* $\Gamma{\cdot}$(*fst i* ⊔ $(\lambda\text{-}.up{\cdot}0)\ f|`(thunks\ \Gamma))$, *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*]:
⟦ (⋀ *e*. *e* ∈ *snd* ` *set heap2* ⟹ *cccexp1 e* = *cccexp2 e*); *heap1* = *heap2* ⟧
⟹ *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

12.5.1 The non-recursive case

definition *ABind-nonrec* :: *var* ⇒ *exp* ⇒ *AEnv* × *CoCalls* → $Arity_\perp$
where
ABind-nonrec x e = (Λ *i*. (*if isVal e* ∨ *x*−−*x*∉(*snd i*) *then fst i x else up*·*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*]: $\pi \cdot$ (*ABind-nonrec x e*) = *ABind-nonrec* ($\pi \cdot x$) ($\pi \cdot e$)
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*]: $\pi \cdot (CoCallArityAnalysis.Afix\text{-}nonrec\ cccExp\ x\ e) = CoCallArityAnalysis.Afix\text{-}nonrec\ (\pi \cdot cccExp)\ (\pi \cdot x)\ (\pi \cdot e)$
apply (*rule cfun-eqvtI*)
unfolding *CoCallArityAnalysis.Afix-nonrec-eq*
by *perm-simp rule*

lemma *CCFix-nonrec-eqvt*[*eqvt*]: $\pi \cdot (CoCallArityAnalysis.CCfix\text{-}nonrec\ cccExp\ x\ e) = CoCallArityAnalysis.CCfix\text{-}nonrec\ (\pi \cdot cccExp)\ (\pi \cdot x)\ (\pi \cdot e)$
apply (*rule cfun-eqvtI*)
unfolding *CoCallArityAnalysis.CCfix-nonrec-eq*
by *perm-simp rule*

lemma *cccFix-nonrec-eqvt*[*eqvt*]: $\pi \cdot (CoCallArityAnalysis.cccFix\text{-}nonrec\ cccExp\ x\ e) = CoCallArityAnalysis.cccFix\text{-}nonrec\ (\pi \cdot cccExp)\ (\pi \cdot x)\ (\pi \cdot e)$
apply (*rule cfun-eqvtI*)
unfolding *CoCallArityAnalysis.cccFix-nonrec-eq*
by *perm-simp rule*

12.5.2 Combining the cases

context *CoCallArityAnalysis*
begin
definition *cccFix-choose* :: $heap \Rightarrow ((AEnv \times CoCalls) \to (AEnv \times CoCalls))$
where $cccFix\text{-}choose\ \Gamma = (if\ nonrec\ \Gamma\ then\ case\text{-}prod\ cccFix\text{-}nonrec\ (hd\ \Gamma)\ else\ cccFix\ \Gamma)$

lemma *cccFix-choose-simp1*[*simp*]:
$\neg\ nonrec\ \Gamma \Longrightarrow cccFix\text{-}choose\ \Gamma = cccFix\ \Gamma$
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 ⟹ CoCallArityAnalysis.cccFix-nonrec cccexp1 x e = CoCallArityAnalysis.cccFix-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*]:
⟦ (⋀ *e*. *e* ∈ *snd* ‘ *set heap2* ⟹ *cccexp1 e = cccexp2 e*); *heap1 = heap2* ⟧
⟹ *CoCallArityAnalysis.cccFix-choose cccexp1 heap1 = CoCallArityAnalysis.cccFix-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

12.6 CoCallGraph-TTree

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*: $(x --y \in (ccApprox\ t)) \longleftrightarrow (\exists\ xs \in paths\ t.\ (x --y \in (ccFromList\ xs)))$
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*: $cc\text{-}restr\ S\ (ccApprox\ t) \sqsubseteq G \longleftrightarrow (\forall\ xs \in paths\ t.\ cc\text{-}restr\ S\ (ccFromList\ xs) \sqsubseteq G)$
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*:
$ccApprox\ (ttree\text{-}restr\ S\ (nxt\ t\ x)) \sqsubseteq ccApprox\ (ttree\text{-}restr\ S\ t)$
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*]: $ccApprox\ (ttree\text{-}restr\ S\ t) = cc\text{-}restr\ S\ (ccApprox\ t)$
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'*) ⊑ *ccApprox* (*t* ⊗⊗ *t'*)
proof(*rule ccProd-belowI*)
fix *x y*
assume *x* ∈ *carrier t* **and** *y* ∈ *carrier t'*
then obtain *xs ys* **where** *x* ∈ *set xs* **and** *y* ∈ *set ys*
and *xs* ∈ *paths t* **and** *ys* ∈ *paths t'* **by** (*auto simp add*: *Union-paths-carrier*[*symmetric*])
hence *xs* @ *ys* ∈ *paths* (*t* ⊗⊗ *t'*) **by** (*metis paths-both append-interleave*)
moreover
from ‹*x* ∈ *set xs*› ‹*y* ∈ *set ys*›
have *x*−−*y*∈(*ccFromList* (*xs*@*ys*)) **by** *simp*
ultimately
show *x*−−*y*∈(*ccApprox* (*t* ⊗⊗ *t'*)) **by** (*auto simp add*: *in-ccApprox simp del*: *ccFrom-List-append*)
qed
ultimately
show *ccApprox t* ⊔ *ccApprox t'* ⊔ *ccProd* (*carrier t*) (*carrier t'*) ⊑ *ccApprox* (*t* ⊗⊗ *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*) = ⊥
by *transfer' auto*

lemma *ccApprox-either*[*simp*]: *ccApprox* (*t* ⊕⊕ *t'*) = *ccApprox t* ⊔ *ccApprox t'*
by *transfer'* (*rule CoCalls-eqI*, *auto*)

lemma *wild-recursion*:
assumes *ccApprox t* ⊑ *G*
assumes ⋀ *x*. *x* ∉ *S* ⟹ *f x* = *empty*
assumes ⋀ *x*. *x* ∈ *S* ⟹ *ccApprox* (*f x*) ⊑ *G*
assumes ⋀ *x*. *x* ∈ *S* ⟹ *ccProd* (*ccNeighbors x G*) (*carrier* (*f x*)) ⊑ *G*
shows *ccApprox* (*ttree-restr* (−*S*) (*substitute f T t*)) ⊑ *G*
proof(*rule ccApprox-belowI*)
fix *xs*
define *seen* :: *var set* **where** *seen* = {}

assume *xs* ∈ *paths* (*ttree-restr* (− *S*) (*substitute f T t*))
then obtain *xs' xs''* **where** *xs* = [*x*←*xs'* . *x* ∉ *S*] **and** *substitute'' f T xs'' xs'* **and** *xs''* ∈ *paths t*
by (*auto simp add*: *filter-paths-conv-free-restr2*[*symmetric*] *substitute-substitute''*)

note *this*(*2*)
moreover
from ‹*ccApprox t* ⊑ *G*› **and** ‹*xs''* ∈ *paths t*›
have *ccFromList xs''* ⊑ *G*
by (*auto simp add*: *ccApprox-below-iff*)
moreover
note *assms*(*2*)
moreover
from *assms*(*3*,*4*)
have ⋀ *x ys*. *x* ∈ *S* ⟹ *ys* ∈ *paths* (*f x*) ⟹ *ccFromList ys* ⊑ *G*
and ⋀ *x ys*. *x* ∈ *S* ⟹ *ys* ∈ *paths* (*f x*) ⟹ *ccProd* (*ccNeighbors x G*) (*set ys*) ⊑ *G*
by (*auto simp add*: *ccApprox-below-iff Union-paths-carrier*[*symmetric*] *cc-lub-below-iff*)
moreover
have *ccProd seen* (*set xs''*) ⊑ *G* **unfolding** *seen-def* **by** *simp*
ultimately
have *ccFromList* [*x*←*xs'* . *x* ∉ *S*] ⊑ *G* ∧ *ccProd* (*seen*) (*set xs'*) ⊑ *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*} ⊑ *G*
using ‹*ccProd seen* (*set* (*x* # *xs*)) ⊑ *G*›
by (*auto simp add*: *ccProd-insert2*[**where** *S'* = *set xs* **for** *xs*] *join-below-iff*)

show *?case*
proof(*cases x* ∈ *S*)
case *True*

from ‹*ccFromList* (*x* # *xs*) ⊑ *G*›
have *ccProd* {*x*} (*set xs*) ⊑ *G* **by** (*auto simp add*: *join-below-iff*)
hence *subset1*: *set xs* ⊆ *ccNeighbors x G* **by** *transfer auto*

from ‹*ccProd seen* (*set* (*x* # *xs*)) ⊑ *G*›
have *subset2*: *seen* ⊆ *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* ∪ *set xs* ⊆ *ccNeighbors x G* **by** *auto*
hence *ccProd* (*seen* ∪ *set xs*) (*set zs*) ⊑ *ccProd* (*ccNeighbors x G*) (*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*) (*set zs*) ⊑ *G* **by** *this simp*

```
    with ‹x ∈ S› Cons.prems Cons.hyps
    have ccFromList [x←ys . x ∉ S] ⊑ G ∧ ccProd (seen) (set ys) ⊑ 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 ∈ S›  seen-x
    show ccFromList [x←x # ys . x ∉ S] ⊑ G  ∧ ccProd seen (set (x#ys)) ⊑ 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←ys . x ∉ S] ⊑ G ∧ ccProd ((insert x seen)) (set ys) ⊑ 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 ∗
    have ccProd {x} (set ys) ⊑  G
     by (auto simp add: insert-Diff-if ccProd-insert1[where S′ = seen] join-below-iff)
    hence ccProd {x} {x ∈ set ys. x ∉ S} ⊑ G
     by (rule below-trans[rotated, OF - ccProd-mono2]) auto
    moreover
    note False seen-x
    ultimately
    show ccFromList [x←x # ys . x ∉ S] ⊑ G ∧ ccProd (seen) (set (x # ys)) ⊑ 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 = -›
 show ccFromList xs ⊑ G by simp
qed

lemma wild-recursion-thunked:
 assumes ccApprox  t ⊑ G
 assumes ⋀ x. x ∉ S ⟹ f x = empty
 assumes ⋀ x. x ∈ S ⟹ ccApprox (f x) ⊑ G
 assumes ⋀ x. x ∈ S ⟹ ccProd (ccNeighbors x G − {x} ∩ T) (carrier (f x)) ⊑ G
 shows ccApprox (ttree-restr (−S) (substitute f T t)) ⊑ G
proof(rule ccApprox-belowI)
 fix xs
```

define *seen* :: *var set* **where** *seen* = {}
define *seen-T* :: *var set* **where** *seen-T* = {}

assume *xs* ∈ *paths* (*ttree-restr* (− *S*) (*substitute f T t*))
then obtain *xs′ xs″* **where** *xs* = [*x*←*xs′* . *x* ∉ *S*] **and** *substitute″ f T xs″ xs′* **and** *xs″* ∈ *paths t*
by (*auto simp add*: *filter-paths-conv-free-restr2*[*symmetric*] *substitute-substitute″*)

note *this*(*2*)
moreover
from ‹*ccApprox t* ⊑ *G*› **and** ‹*xs″* ∈ *paths t*›
have *ccFromList xs″* ⊑ *G*
by (*auto simp add*: *ccApprox-below-iff*)
hence *ccFromList xs″ G*|‘ (− *seen-T*) ⊑ *G*
by (*rule rev-below-trans*[*OF - cc-restr-below-arg*])
moreover
note *assms*(*2*)
moreover
from *assms*(*3*,*4*)
have ⋀ *x ys*. *x* ∈ *S* ⟹ *ys* ∈ *paths* (*f x*) ⟹ *ccFromList ys* ⊑ *G*
and ⋀ *x ys*. *x* ∈ *S* ⟹ *ys* ∈ *paths* (*f x*) ⟹ *ccProd* (*ccNeighbors x G* − {*x*} ∩ *T*) (*set ys*) ⊑ *G*
by (*auto simp add*: *ccApprox-below-iff seen-T-def Union-paths-carrier*[*symmetric*] *cc-lub-below-iff*)
moreover
have *ccProd seen* (*set xs″* − *seen-T*) ⊑ *G* **unfolding** *seen-def seen-T-def* **by** *simp*
moreover
have *seen* ∩ *S* = {} **unfolding** *seen-def* **by** *simp*
moreover
have *seen-T* ⊆ *S* **unfolding** *seen-T-def* **by** *simp*
moreover
have ⋀ *x*. *x* ∈ *seen-T* ⟹ *f x* = *empty* **unfolding** *seen-T-def* **by** *simp*
ultimately
have *ccFromList* [*x*←*xs′* . *x* ∉ *S*] ⊑ *G* ∧ *ccProd* (*seen*) (*set xs′* − *seen-T*) ⊑ *G*
proof(*induction f T xs″ xs′ arbitrary*: *seen seen-T rule*: *substitute″.induct*[*case-names Nil Cons*])
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons zs f x xs′ xs T ys*)

let *?seen-T* = *if x* ∈ *T then insert x seen-T else seen-T*
have *subset*: − *insert x seen-T* ⊆ − *seen-T* **by** *auto*
have *subset2*: *set xs* ∩ − *insert x seen-T* ⊆ *insert x* (*set xs*) ∩ − *seen-T* **by** *auto*
have *subset3*: *set zs* ∩ − *insert x seen-T* ⊆ *set zs* **by** *auto*
have *subset4*: *set xs* ∩ − *seen-T* ⊆ *insert x* (*set xs*) ∩ − *seen-T* **by** *auto*
have *subset5*: *set zs* ∩ − *seen-T* ⊆ *set zs* **by** *auto*
have *subset6*: *set ys* − *seen-T* ⊆ (*set ys* − *?seen-T*) ∪ {*x*} **by** *auto*

show *?case*

```
  proof(cases x ∈ seen-T)
    assume x ∈ seen-T

    have [simp]: f x = empty using ‹x ∈ 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* ∈ *S*› *Cons.prems Cons.hyps*(*1*,*2*)
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*: *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* ∈ *S*› *seen-x* ‹*x* ∉ *seen-T*›
show *ccFromList* [*x*←*x* # *ys* . *x* ∉ *S*] ⊑ *G* ∧ *ccProd seen* (*set* (*x*#*ys*) − *seen-T*) ⊑ *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*←*ys* . *x* ∉ *S*] ⊑ *G* ∧ *ccProd* ((*insert x seen*)) (*set ys* − *seen-T*) ⊑ *G*
apply −
apply (*rule Cons.IH*[**where** *seen* = *insert x seen* **and** *seen-T* = *seen-T*])
apply (*auto simp add*: ‹*x* ∉ *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* ∩ − *seen-T* **for** *xs*]
simp add: *ccProd-insert1*[**where** *S′* = *seen*])
done
moreover
{
from *False* ∗
have *ccProd* {*x*} (*set ys* − *seen-T*) ⊑ *G*
by (*auto simp add*: *insert-Diff-if ccProd-insert1*[**where** *S′* = *seen*] *join-below-iff*)
hence *ccProd* {*x*} {*x* ∈ *set ys* − *seen-T*. *x* ∉ *S*} ⊑ *G*
by (*rule below-trans*[*rotated*, *OF* - *ccProd-mono2*]) *auto*
also have {*x* ∈ *set ys* − *seen-T*. *x* ∉ *S*} = {*x* ∈ *set ys*. *x* ∉ *S*}

```
      using ‹seen-T ⊆ S› by auto
    finally
    have ccProd {x} {x ∈ set ys. x ∉ S} ⊑ G.
    }
    moreover
    note False seen-x
    ultimately
    show ccFromList [x←x # ys . x ∉ S] ⊑ G ∧ ccProd (seen) (set (x # ys) − seen-T) ⊑
G
      by (auto simp add: join-below-iff  simp add: insert-Diff-if  ccProd-insert2[where S′ =
set ys − seen-T for xs]   ccProd-insert1[where S′ = seen])
   qed
  qed
 qed
 with ‹xs = -›
 show ccFromList xs ⊑ G by simp
qed


inductive-set valid-lists :: var set ⇒ CoCalls ⇒ var list set
 for S G
 where  [] ∈ valid-lists S G
 | set xs ⊆ ccNeighbors x G ⟹ xs ∈ valid-lists S G ⟹ x ∈ S ⟹ x#xs ∈ valid-lists S G

inductive-simps valid-lists-simps[simp]: [] ∈ valid-lists S G (x#xs) ∈ valid-lists S G
inductive-cases vald-lists-ConsE: (x#xs) ∈ valid-lists S G

lemma  valid-lists-downset-aux:
 xs ∈ valid-lists S CoCalls ⟹ butlast xs ∈ valid-lists S CoCalls
 by (induction xs) (auto dest: in-set-butlastD)

lemma valid-lists-subset: xs ∈ valid-lists S G ⟹ set xs ⊆ S
 by (induction rule: valid-lists.induct) auto

lemma valid-lists-mono1:
 assumes S ⊆ S′
 shows valid-lists S G ⊆ valid-lists S′ G
proof
 fix xs
 assume xs ∈ valid-lists S G
 thus xs ∈ valid-lists S′ G
   by (induction rule: valid-lists.induct) (auto dest: subsetD[OF assms])
qed

lemma valid-lists-chain1:
  assumes chain Y
  assumes xs ∈ valid-lists (⋃(Y ‘ UNIV)) G
  shows ∃ i. xs ∈ valid-lists (Y i) G
proof−
```

```
  note ‹chain Y›
  moreover
  from assms(2)
  have set xs ⊆ ⋃(Y ` UNIV) 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 ∈ S›
  ultimately
  have x # xs ∈ 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 ∈ valid-lists S G) = (xs ∈ valid-lists S (cc-restr S G))
    by (induction xs) (auto dest: subsetD[OF valid-lists-subset])
qed

lemma interleave-valid-list:
  xs ∈ ys ⊗ zs ⟹ ys ∈ valid-lists S G ⟹ zs ∈ valid-lists S' G' ⟹ xs ∈ valid-lists (S ∪ S')
(G ⊔ (G' ⊔ ccProd S S'))
proof (induction rule:interleave-induct)
  case Nil
  show ?case by simp
next
  case (left ys zs xs x)

  from ‹x # ys ∈ valid-lists S G›
  have x ∈ S and set ys ⊆ ccNeighbors x G and ys ∈ valid-lists S G
    by auto

  from ‹xs ∈ ys ⊗ zs›
  have set xs = set ys ∪ set zs by (rule interleave-set)
  with ‹set ys ⊆ ccNeighbors x G› valid-lists-subset[OF ‹zs ∈ valid-lists S' G'›]
  have set xs ⊆ ccNeighbors x (G ⊔ (G' ⊔ ccProd S S'))
    by (auto simp add: ccNeighbors-ccProd ‹x ∈ S›)
  moreover
  from ‹ys ∈ valid-lists S G› ‹zs ∈ valid-lists S' G'›
  have xs ∈ valid-lists (S ∪ S') (G ⊔ (G' ⊔ ccProd S S'))
    by (rule left.IH)
  moreover
  from ‹x ∈ S›
  have x ∈ S ∪ S' by simp
  ultimately
  show ?case..
next
  case (right ys zs xs x)

  from ‹x # zs ∈ valid-lists S' G'›
  have x ∈ S' and set zs ⊆ ccNeighbors x G' and zs ∈ valid-lists S' G'
```

```
    by auto

  from ‹xs ∈ ys ⊗ zs›
  have set xs = set ys ∪ set zs by (rule interleave-set)
  with ‹set zs ⊆ ccNeighbors x G'› valid-lists-subset[OF ‹ys ∈ valid-lists S G›]
  have set xs ⊆ ccNeighbors x (G ⊔ (G' ⊔ ccProd S S'))
    by (auto simp add: ccNeighbors-ccProd ‹x ∈ S'›)
  moreover
  from ‹ys ∈ valid-lists S G› ‹zs ∈ valid-lists S' G'›
  have xs ∈ valid-lists (S ∪ S') (G ⊔ (G' ⊔ ccProd S S'))
    by (rule right.IH)
  moreover
  from ‹x ∈ S'›
  have x ∈ S ∪ S' by simp
  ultimately
  show ?case..
qed

lemma interleave-valid-list':
  xs ∈ valid-lists (S ∪ S') G ⟹ ∃ ys zs. xs ∈ ys ⊗ zs ∧ ys ∈ valid-lists S G ∧ zs ∈ valid-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 ∈ ys ⊗ zs ys ∈ valid-lists S G zs ∈ valid-lists S' G by auto

    from ‹xs ∈ ys ⊗ zs› have set xs = set ys ∪ set zs by (rule interleave-set)
    with ‹set xs ⊆ ccNeighbors x G›
    have set ys ⊆ ccNeighbors x G and set zs ⊆ ccNeighbors x G  by auto

  from ‹x ∈ S ∪ S'›
  show ?case
  proof
    assume x ∈ S
    with ‹set ys ⊆ ccNeighbors x G› ‹ys ∈ valid-lists S G›
    have x # ys ∈ valid-lists S G
      by rule
    moreover
    from ‹xs ∈ ys ⊗ zs›
    have x#xs ∈ x#ys ⊗ zs..
    ultimately
    show ?thesis using ‹zs ∈ valid-lists S' G› by blast
  next
    assume x ∈ S'
    with ‹set zs ⊆ ccNeighbors x G› ‹zs ∈ valid-lists S' G›
    have x # zs ∈ valid-lists S' G
      by rule
    moreover
```

```
    from ‹xs ∈ ys ⊗ zs›
    have x#xs ∈ ys ⊗ 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: subsetD[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 ∈ paths (ccTTree S G) by (metis paths-ccTTree)
qed

lemma ccTTree-mono1:
  S ⊆ S' ⟹ ccTTree S G ⊑ 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 ⊑ G' ⟹ ccTTree S G ⊑ 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 ⊆ S' ⟹ G ⊑ G' ⟹ ccTTree S G ⊑ 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* ∈ *valid-lists S G* **and** $x \in S'$
from *this assms*
have *length* [x'←*xs* . $x' = x$] ≤ *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'* ⟹ *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'* ⟹ *ccTTree S G* = *ccTTree S G'*
by (*metis ccTTree-cc-restr*)

lemma *either-ccTTree*:
ccTTree S G ⊕⊕ *ccTTree S' G'* $\sqsubseteq$ *ccTTree* (*S* ∪ *S'*) (*G* ⊔ *G'*)
by (*auto intro*!: *either-belowI ccTTree-mono*)

lemma *interleave-ccTTree*:
ccTTree S G ⊗⊗ *ccTTree S' G'* $\sqsubseteq$ *ccTTree* (*S* ∪ *S'*) (*G* ⊔ *G'* ⊔ *ccProd S S'*)
by *transfer'* (*auto*, *erule* (*2*) *interleave-valid-list*)

lemma *interleave-ccTTree'*:
ccTTree (*S* ∪ *S'*) *G* $\sqsubseteq$ *ccTTree S G* ⊗⊗ *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 ∈ *valid-lists* {*x′* ∈ *S*. *P x′*} *G* ⟹ *xs* ∈ *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* ⊆ *ccNeighbors x G* **and** *xs* ∈ *valid-lists* {*x′* ∈ *S*. *P x′*} *G* **and** *x* ∈ *S* **and** *P x* **by** *auto*

from *this*(*2*) **have** *set xs* ⊆ {*x′* ∈ *S*. *P x′*} **by** (*rule valid-lists-subset*)
hence ∀ *x* ∈ *set xs*. *P x* **by** *auto*
hence [*simp*]: *filter P xs* = *xs* **by** (*rule filter-True*)

from *Cons.IH*[*OF* ‹*xs* ∈ -›]
have *xs* ∈ *filter P* ‘ *valid-lists S G***.**

from ‹*xs* ∈ *valid-lists* {*x′* ∈ *S*. *P x′*} *G*›
have *xs* ∈ *valid-lists S G* **by** (*rule subsetD*[*OF valid-lists-mono1*, *rotated*]) *auto*

from ‹*set xs* ⊆ *ccNeighbors x G*› *this* ‹*x* ∈ *S*›
have *x* # *xs* ∈ *valid-lists S G* **by** *rule*

hence *filter P* (*x* # *xs*) ∈ *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* = (λ *x′*. *x′*≠ *x*)] *simp add*: *set-diff-eq*)

lemma *ttree-restr-ccTTree*[*simp*]:
ttree-restr S′ (*ccTTree S G*) = *ccTTree* (*S* ∩ *S′*) *G*
by (*transfer′ fixing*: *S′*) (*auto dest*: *filter-valid-lists′ filter-valid-lists*[**where** *P* = (λ *x′*. *x′* ∈ *S′*)] *simp add*:*Int-def*)

lemma *repeatable-ccTTree-ccSquare*: *S* ⊆ *S′* ⟹ *repeatable* (*ccTTree S* (*ccSquare S′*))
unfolding *repeatable-def*
by *transfer* (*auto simp add*:*ccNeighbors-ccSquare dest*: *subsetD*[*OF valid-lists-subset*])

An alternative definition

inductive *valid-lists′* :: *var set* ⇒ *CoCalls* ⇒ *var set* ⇒ *var list* ⇒ *bool*
for *S G*
where *valid-lists′ S G prefix* []
| *prefix* ⊆ *ccNeighbors x G* ⟹ *valid-lists′ S G* (*insert x prefix*) *xs* ⟹ *x* ∈ *S* ⟹ *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 ∈ *valid-lists S G* ⟹ *ccProd prefix* (*set xs*) ⊑ *G* ⟹ *valid-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-lists' S G prefix xs ⟹ *x* ∈ *prefix* ⟹ *ccProd* (*set xs*) {*x*} ⊑ *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-lists' S G prefix xs ⟹ *xs* ∈ *valid-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 ∈ *valid-lists S G* ⟷ *set xs* ⊆ *S* ∧ (∀ *n*. *ccProd* (*set* (*take n xs*)) (*set* (*drop n xs*)) ⊑ *G*)
proof(*safe*)
fix *x*
assume *xs* ∈ *valid-lists S G*
from *valid-lists-subset*[*OF this*]
show *x* ∈ *set xs* ⟹ *x* ∈ *S* **by** *auto*
next

```
  fix n
  assume xs ∈ valid-lists S G
  thus ccProd (set (take n xs)) (set (drop n xs)) ⊑ 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 sub-
set-ccNeighbors)
      by (metis dual-order.trans set-drop-subset subset-ccNeighbors)
    qed
  qed
next
  assume set xs ⊆ S
  and ∀ n. ccProd (set (take n xs)) (set (drop n xs)) ⊑ G
  thus xs ∈ valid-lists S G
  proof (induction xs)
    case Nil thus ?case by simp
  next
    case (Cons x xs)
    from ‹∀ n. ccProd (set (take n (x # xs))) (set (drop n (x # xs))) ⊑ G›
    have ∀ n. ccProd (set (take n xs)) (set (drop n xs)) ⊑ 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 ∈ valid-lists S G by auto
    with Cons.prems(1)  spec[OF ‹∀ n. ccProd (set (take n (x # xs))) (set (drop n (x # xs)))
⊑ G›, where x = 1]
    show ?case by (simp add: subset-ccNeighbors)
  qed
qed

end
```

12.7 CoCallImplTTree

```
theory CoCallImplTTree
imports TTreeAnalysisSig Env−Set−Cpo CoCallAritySig CoCallGraph−TTree
begin

context CoCallArity
begin
```

```
  definition Texp :: exp ⇒ Arity → var ttree
    where Texp e = (Λ a. ccTTree (edom (Aexp e ·a)) (ccExp e·a))

  lemma Texp-simp: Texp e·a = ccTTree (edom (Aexp e ·a)) (ccExp e·a)
    unfolding Texp-def
    by simp

  sublocale TTreeAnalysis Texp.
end



end
```

12.8 CoCallImplTTreeSafe

```
theory CoCallImplTTreeSafe
imports CoCallImplTTree CoCallAnalysisSpec TTreeAnalysisSpec
begin

lemma valid-lists-many-calls:
  assumes ¬ one-call-in-path x p
  assumes p ∈ valid-lists S G
  shows x−−x ∈ 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 ∈ set xs
    proof(rule ccontr)
      assume x ∉ 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 ⊆ ccNeighbors x' G›
    have x ∈ ccNeighbors x G by auto
    thus ?thesis by simp
```

```
  qed
qed

context CoCallArityEdom
begin
 lemma carrier-Fexp': carrier (Texp e·a) ⊆ fv e
   unfolding Texp-simp carrier-ccTTree
   by (rule Aexp-edom)

end


context CoCallAritySafe
begin

lemma carrier-AnalBinds-below:
  carrier ((Texp.AnalBinds  Δ·(Aheap Δ e·a)) x) ⊆ edom ((ABinds Δ)·(Aheap Δ e·a))
by (auto simp add: Texp.AnalBinds-lookup Texp-def split: option.splits
        elim!: subsetD[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} ∪ edom (Aexp e·(inc·a)) ⊆ edom (Aexp (App e x)·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*: *subsetD*[*OF Aexp-edom*]
intro!: *below-ccTTreeI below-trans*[*OF - ccExp-Lam*] *cc-restr-mono1 subsetD*[*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*] *subsetD*[*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 $e{\cdot}a$) = *edom* (*Aheap* Γ $e{\cdot}a$)
by (*rule carrier-Fheap'*)
next
fix x Γ p e a
assume $x \in$ *thunks* Γ

assume $\neg$ *one-call-in-path* x p
hence $x \in$ *set* p **by** (*rule more-than-one-setD*)

assume $p \in$ *paths* (*Theap* Γ $e{\cdot}a$) **with** ‹$x \in$ *set* p›
have $x \in$ *carrier* (*Theap* Γ $e{\cdot}a$) **by** (*auto simp add*: *Union-paths-carrier*[*symmetric*])
hence $x \in$ *edom* (*Aheap* Γ $e{\cdot}a$)
unfolding *Theap-simp* **by** (*auto split*: *if-splits*)

show (*Aheap* Γ $e{\cdot}a$) x = *up*·*0*
proof(*cases nonrec* Γ)
case *False*
from *False* ‹$x \in$ *thunks* Γ› ‹$x \in$ *edom* (*Aheap* Γ $e{\cdot}a$)›
show *?thesis* **by** (*rule aHeap-thunks-rec*)
next
case *True*
with ‹$p \in$ *paths* (*Theap* Γ $e{\cdot}a$)›
have $p \in$ *valid-lists* (*edom* (*Aheap* Γ $e{\cdot}a$)) (*ccExp* $e{\cdot}a$) **by** (*simp add*: *Theap-simp*)

with ‹$\neg$ *one-call-in-path* x p›
have $x{--}x{\in}$ (*ccExp* $e{\cdot}a$) **by** (*rule valid-lists-many-calls*)

from *True* ‹$x \in$ *thunks* Γ› *this*
show *?thesis* **by** (*rule aHeap-thunks-nonrec*)
qed
next
fix Δ e a

have *carrier*: *carrier* (*substitute* (*Texp.AnalBinds* $\Delta{\cdot}$(*Aheap* Δ $e{\cdot}a$)) (*thunks* Δ) (*Texp* $e{\cdot}a$)) $\subseteq$ *edom* (*Aheap* Δ $e{\cdot}a$) $\cup$ *edom* (*Aexp* (*Let* Δ e)·a)
proof(*rule carrier-substitute-below*)
from *edom-mono*[*OF Aexp-Let*[*of* Δ e a]]
show *carrier* (*Texp* $e{\cdot}a$) $\subseteq$ *edom* (*Aheap* Δ $e{\cdot}a$) $\cup$ *edom* (*Aexp* (*Let* Δ e)·a) **by** (*simp add*: *Texp-def*)
next
fix x
assume $x \in$ *edom* (*Aheap* Δ $e{\cdot}a$) $\cup$ *edom* (*Aexp* (*Let* Δ e)·a)
hence $x \in$ *edom* (*Aheap* Δ $e{\cdot}a$) $\vee$ x : (*edom* (*Aexp* (*Let* Δ e)·a)) **by** *simp*
thus *carrier* ((*Texp.AnalBinds* $\Delta{\cdot}$(*Aheap* Δ $e{\cdot}a$)) x) $\subseteq$ *edom* (*Aheap* Δ $e{\cdot}a$) $\cup$ *edom* (*Aexp* (*Let* Δ e)·a)
proof
assume $x \in$ *edom* (*Aheap* Δ $e{\cdot}a$)

```
    have carrier ((Texp.AnalBinds  Δ·(Aheap Δ e·a)) x) ⊆ edom (ABinds Δ·(Aheap Δ e·a))
      by (rule carrier-AnalBinds-below)
    also have ... ⊆ edom (Aheap Δ e·a ⊔ 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 ∈ edom (Aexp (Terms.Let Δ e)·a)
    hence x ∉ domA Δ by (auto  dest: subsetD[OF Aexp-edom])
    hence (Texp.AnalBinds  Δ·(Aheap Δ e·a)) x = ⊥
      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)) ⊑ 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) ∪ edom (Aexp (Terms.Let Δ e)·a) − domA Δ = edom (Aexp
(Let Δ e)·a)
     by (auto dest: subsetD[OF edom-Aheap] subsetD[OF Aexp-edom])
   finally
  show carrier (ttree-restr (− domA Δ) (substitute (Texp.AnalBinds Δ·(Aheap Δ e·a)) (thunks
Δ)(Texp e·a)))
       ⊆ 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*!: *subsetD*[*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*: *subsetD*[*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*!: *subsetD*[*OF edom-Aheap*] *subsetD*[*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* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)) ⊑ *Theap* Δ *e*·*a*.

qed
end

lemma *paths-singles*: *xs* ∈ *paths* (*singles S*) ⟷ (∀ *x* ∈ *S*. *one-call-in-path x xs*)
by *transfer* (*auto simp add*: *one-call-in-path-filter-conv*)

lemma *paths-singles′*: *xs* ∈ *paths* (*singles S*) ⟷ (∀ *x* ∈ (*set xs* ∩ *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* ∈ *paths* ($t \otimes\otimes t'$)
then obtain *ys zs* **where** *ys* ∈ *paths t* **and** *zs* ∈ *paths* t' **and** *xs* ∈ *ys* ⊗ *zs* **by** (*auto simp add*: *paths-both*)
with *assms*
have *ys* ∈ *paths* (*singles S*) **and** *set zs* ∩ *S* = {}
by (*metis below-ttree.rep-eq contra-subsetD paths.rep-eq*, *auto simp add*: *Union-paths-carrier*[*symmetric*])
with ‹*xs* ∈ *ys* ⊗ *zs*›
show *xs* ∈ *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* ∈ *paths* (*ttree-restr* S' (*singles S*)) ⟷ *set xs* ⊆ S' ∧ (∀ *x* ∈ *S*. *one-call-in-path x xs*)
proof
show *xs* ∈ *paths* (*ttree-restr* S' (*singles S*)) ⟹ *set xs* ⊆ S' ∧ (∀ *x* ∈ *S*. *one-call-in-path x xs*)
by (*auto simp add*: *filter-paths-conv-free-restr*[*symmetric*] *paths-singles*)
next
assume ∗: *set xs* ⊆ S' ∧ (∀ *x*∈*S*. *one-call-in-path x xs*)
hence *set xs* ⊆ S' **by** *auto*
hence [*simp*]: *filter* (λ x'. $x' \in S'$) *xs* = *xs* **by** (*auto simp add*: *filter-id-conv*)

from ∗
have *xs* ∈ *paths* (*singles S*)
by (*auto simp add*: *paths-singles'*)
hence *filter* (λ x'. $x' \in S'$) *xs* ∈ *filter* (λx'. $x' \in S'$) ‘ *paths* (*singles S*)
by (*rule imageI*)
thus *xs* ∈ *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 $\bigwedge x'.\ x \notin \mathit{carrier}\ (f\ x')$
shows $x \notin \ \mathit{carrier}\ (\mathit{substitute}\ f\ T\ t)$
proof−
have *ttree-restr* $(\{x\})$ $(\mathit{substitute}\ f\ T\ t) = \mathit{ttree\text{-}restr}\ (\{x\})\ t$
proof(*rule ttree-rest-substitute*)
fix x'
from ‹$x \notin \mathit{carrier}\ (f\ x')$›
show $\mathit{carrier}\ (f\ x') \cap \{x\} = \{\}$ **by** *auto*
qed
hence $x \notin \mathit{carrier}\ (\mathit{ttree\text{-}restr}\ (\{x\})\ (\mathit{substitute}\ f\ T\ t)) \longleftrightarrow x \notin \mathit{carrier}\ (\mathit{ttree\text{-}restr}\ (\{x\})\ t)$
by *metis*
with *assms*(*1*)
show *?thesis* **by** *simp*
qed

lemma *substitute-below-singlesI*:
assumes $t \sqsubseteq \mathit{singles}\ S$
assumes $\bigwedge x.\ \mathit{carrier}\ (f\ x) \cap S = \{\}$
shows $\mathit{substitute}\ f\ T\ t \sqsubseteq \mathit{singles}\ S$
proof(*rule ttree-belowI*)
fix *xs*
assume $\mathit{xs} \in \mathit{paths}\ (\mathit{substitute}\ f\ T\ t)$
thus $\mathit{xs} \in \mathit{paths}\ (\mathit{singles}\ S)$
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 ‹$x\#\mathit{xs} \in$ -›
have *xs*: $\mathit{xs} \in \mathit{paths}\ (\mathit{substitute}\ (\mathit{f\text{-}nxt}\ f\ T\ x)\ T\ (\mathit{nxt}\ t\ x \otimes\otimes f\ x))$ **by** *auto*
moreover

from ‹$t \sqsubseteq \mathit{singles}\ S$›
have $\mathit{nxt}\ t\ x \sqsubseteq \mathit{singles}\ S$
by (*metis TTree*−*HOLCF.nxt-mono below-trans nxt-singles-below-singles*)
from *this* ‹$\mathit{carrier}\ (f\ x) \cap S = \{\}$›
have $\mathit{nxt}\ t\ x \otimes\otimes f\ x \sqsubseteq \mathit{singles}\ S$
by (*rule both-below-singles1*)
moreover
{ fix x'
from ‹$\mathit{carrier}\ (f\ x') \cap S = \{\}$›
have $\mathit{carrier}\ (\mathit{f\text{-}nxt}\ f\ T\ x\ x') \cap S = \{\}$
by (*auto simp add*: *f-nxt-def*)
}
ultimately
have *IH*: $\mathit{xs} \in \mathit{paths}\ (\mathit{singles}\ S)$

```
    by (rule Cons.IH)

  show ?case
    proof(cases x ∈ S)
      case True
      with ‹carrier (f x) ∩ S = {}›
      have x ∉ carrier (f x) by auto
      moreover
      from ‹t ⊑ singles S›
      have nxt t x ⊑ nxt (singles S) x by (rule nxt-mono)
      hence carrier (nxt t x) ⊆ carrier (nxt (singles S) x) by (rule carrier-mono)
      from subsetD[OF this] True
      have x ∉ carrier (nxt t x) by auto
      ultimately
      have x ∉ carrier (nxt t x ⊗⊗ f x) by simp
      hence x ∉ carrier (substitute (f-nxt f T x) T (nxt t x ⊗⊗ f x))
      proof(rule substitute-not-carrier)
        fix x'
        from ‹carrier (f x') ∩ S = {}› ‹x ∈ S›
        show x ∉ carrier (f-nxt f T x x') by (auto simp add: f-nxt-def)
      qed
      with xs
      have x ∉ set xs by (auto simp add: Union-paths-carrier[symmetric])
      with IH
      have xs ∈ 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
```

13 CoCall Cardinality Implementation

13.1 CoCallAnalysisImpl

```
theory CoCallAnalysisImpl
imports Arity−Nominal Launchbury.Nominal−HOLCF Launchbury.Env−Nominal Env−Set−Cpo
Launchbury.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*: *subsetD*[*OF ccField-cc-restr*])
  done

lemma *predCC-eqvt*[*eqvt*, *simp*]: π · (*predCC S f*) = *predCC* (π · *S*) (π · *f*)
  apply (*rule cfun-eqvtI*)

unfolding *predCC-eq*
by *perm-simp rule*

lemma *cc-restr-predCC*:
cc-restr S (*predCC S′ f*·*n*) = (*predCC* (*S′* ∩ *S*) (Λ *n*. *cc-restr S* (*f*·*n*)))·*n*
unfolding *predCC-eq*
by (*auto simp add*: *inf-commute ccSquare-def*)

lemma *cc-restr-predCC′*[*simp*]:
cc-restr S (*predCC S f*·*n*) = *predCC S f*·*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* (−π) ♯∗ (*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 (−π) ♯∗ (fv (Terms.Let Γ body) :: var set)

  { fix n
   have combined-restrict (fv (Terms.Let Γ body)) (CoCallArityAnalysis.cccFix-choose cCC-
exp-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 = CoCallArityAnal-
ysis.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 (π • Γ)·(cCCexp-sumC (π • body)·n))) =
        (Λ n. combined-restrict (fv (Terms.Let Γ body)) (CoCallArityAnalysis.cccFix-choose
cCCexp-sumC Γ·(cCCexp-sumC body·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* ≡ *Aexp e·a*
abbreviation *Aexp-bot-syn''* (‹$\mathcal{A}^{\perp}$_›) **where** $\mathcal{A}^{\perp}{}_a$ *e* ≡ *fup·(Aexp e)·a*

abbreviation *ccExp-syn''* (‹$\mathcal{G}$_›) **where** $\mathcal{G}_a$ *e* ≡ *CCexp e·a*
abbreviation *ccExp-bot-syn''* (‹$\mathcal{G}^{\perp}$_›) **where** $\mathcal{G}^{\perp}{}_a$ *e* ≡ *fup·(CCexp e)·a*

lemma *cCCexp-eq*[*simp*]:
cCCexp (Var x)·n = (esing x · (up · n), ⊥)
cCCexp (Lam [x]. e)·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)·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)·n = combined-restrict (fv (Let Γ e)) (CoCallArityAnalysis.cccFix-choose cCCexp Γ · (cCCexp e·n))
cCCexp (Bool b)·n = ⊥
cCCexp (scrut ? e1 : e2)·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))))
by (*simp-all*)
declare *cCCexp.simps*[*simp del*]

lemma *Aexp-pre-simps*:
$\mathcal{A}_a$ (Var x) = esing x·(up·a)
$\mathcal{A}_a$ (Lam [x]. e) = Aexp e·(pred·a) f|‘ fv (Lam [x]. 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|‘ (fv (Let Γ e))
x ∉ fv e ⟹
$\mathcal{A}_a$ (let x be e in exp) =
(fup·(Aexp e)·(ABind-nonrec x e·($\mathcal{A}_a$ exp, CCexp exp·a)) ⊔ $\mathcal{A}_a$ exp)
f|‘ (fv (let x be e in exp))
$\mathcal{A}_a$ (Bool b) = ⊥
$\mathcal{A}_a$ (scrut ? e1 : e2) = $\mathcal{A}_0$ scrut ⊔ $\mathcal{A}_a$ e1 ⊔ $\mathcal{A}_a$ e2
by (*simp add*: *cccFix-eq Aexp-eq fup-Aexp-eq CCexp-eq fup-CCexp-eq*)+

lemma *CCexp-pre-simps*:
CCexp (Var x)·n = ⊥
CCexp (Lam [x]. e)·n = predCC (fv (Lam [x]. e)) (CCexp e)·n

CCexp (*App e x*)·*n* = *CCexp e*·(*inc*·*n*) ⊔ *ccProd* {*x*} (*insert x* (*fv e*))
¬ *nonrec* Γ ⟹
CCexp (*Let* Γ *e*)·*n* = *cc-restr* (*fv* (*Let* Γ *e*))
(*CCfix* Γ·(*Afix* Γ·(*Aexp e*·*n* ⊔ (λ-.*up*·0) *f*|‘ *thunks* Γ), *CCexp e*·*n*))
x ∉ *fv e* ⟹ *CCexp* (*let x be e in exp*)·*n* =
cc-restr (*fv* (*let x be e in exp*))
(*ccBind x e* ·(*Aheap-nonrec x e*·(*Aexp exp*·*n*, *CCexp exp*·*n*), *CCexp exp*·*n*)
⊔ *ccProd* (*fv e*) (*ccNeighbors x* (*CCexp exp*·*n*) − (*if isVal e then* {} *else* {*x*})) ⊔ *CCexp exp*·*n*)
CCexp (*Bool b*)·*n* = ⊥
CCexp (*scrut ? e1 : e2*)·*n* =
CCexp scrut·*0* ⊔
(*CCexp e1*·*n* ⊔ *CCexp e2*·*n*) ⊔
ccProd (*edom* (*Aexp scrut*·*0*)) (*edom* (*Aexp e1*·*n*) ∪ *edom* (*Aexp e2*·*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*·*a*) ⊆ *fv e* **and** *Aexp-edom′*: *edom* (A*a e*) ⊆ *fv e*
apply (*induction e arbitrary*: *a rule*: *exp-induct-rec*)
apply (*auto simp add*: *CCexp-pre-simps predCC-eq Aexp-pre-simps dest*!: *subsetD*[*OF cc-Field-cc-restr*] *subsetD*[*OF ccField-ccProd-subset*])
apply *fastforce*+
done

lemma *cc-restr-CCexp*[*simp*]:
cc-restr (*fv e*) (*CCexp e*·*a*) = *CCexp e*·*a*
by (*rule cc-restr-noop*[*OF ccField-CCexp*])

lemma *ccField-fup-CCexp*:
ccField (*fup*·(*CCexp e*)·*n*) ⊆ *fv e*
by (*cases n*) (*auto dest*: *subsetD*[*OF ccField-CCexp*])

lemma *cc-restr-fup-ccExp-useless*[*simp*]: *cc-restr* (*fv e*) (*fup*·(*CCexp e*)·*n*) = *fup*·(*CCexp e*)·*n*
by (*rule cc-restr-noop*[*OF ccField-fup-CCexp*])

sublocale *EdomArityAnalysis Aexp* **by** *standard* (*rule Aexp-edom′*)

lemma *CCexp-simps*[*simp*]:
G*a*(*Var x*) = ⊥
G*0*(*Lam* [*x*]. *e*) = (*fv* (*Lam* [*x*]. *e*))²
G*inc*·*a*(*Lam* [*x*]. *e*) = *cc-delete x* (G*a e*)
G*a* (*App e x*) = G*inc*·*a e* ⊔ {*x*} *G*×*insert x* (*fv e*)
¬ *nonrec* Γ ⟹ G*a* (*Let* Γ *e*) =
(*CCfix* Γ·(*Afix* Γ·(A*a e* ⊔ (λ-.*up*·0) *f*|‘ *thunks* Γ), G*a e*)) *G*|‘ (− *domA* Γ)
x ∉ *fv e′* ⟹ G*a* (*let x be e′ in e*) =
cc-delete x
(*ccBind x e′* ·(*Aheap-nonrec x e′*·(A*a e*, G*a e*), G*a e*)
⊔ *fv e′ G*× (*ccNeighbors x* (G*a e*) − (*if isVal e′ then* {} *else* {*x*})) ⊔ G*a e*)
G*a* (*Bool b*) = ⊥

$\mathcal{G}_a$ (*scrut ? e1 : e2*) =
$\mathcal{G}_0$ *scrut* ⊔ ($\mathcal{G}_a$ *e1* ⊔ $\mathcal{G}_a$ *e2*) ⊔
edom ($\mathcal{A}_0$ *scrut*) *G*× (*edom* ($\mathcal{A}_a$ *e1*) ∪ *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!: *subsetD*[*OF ccField-cc-delete*] *subsetD*[*OF ccField-cc-restr*] *subsetD*[*OF cc-Field-CCexp*]
subsetD[*OF ccField-CCfix*] *subsetD*[*OF ccField-ccBind*] *subsetD*[*OF cc-Field-ccProd-subset*] *elem-to-ccField*
)

definition *Aheap* **where**
Aheap Γ *e* = (Λ *a*. *if nonrec* Γ *then* (*case-prod Aheap-nonrec* (*hd* Γ))·(*Aexp e*·*a*, *CCexp e*·*a*) *else* (*Afix* Γ · (*Aexp e*·*a* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ)) *f*|‘ *domA* Γ)

lemma *Aheap-simp1*[*simp*]:
¬ *nonrec* Γ ⟹ *Aheap* Γ *e* ·*a* = (*Afix* Γ · (*Aexp e*·*a* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ)) *f*|‘ *domA* Γ
unfolding *Aheap-def* **by** *simp*

lemma *Aheap-simp2*[*simp*]:
x ∉ *fv e*′ ⟹ *Aheap* [(*x*,*e*′)] *e* ·*a* = *Aheap-nonrec x e*′·(*Aexp e*·*a*, *CCexp e*·*a*)
unfolding *Aheap-def* **by** (*simp add*: *nonrec-def*)

lemma *Aheap-eqvt*′[*eqvt*]:
π · (*Aheap* Γ *e*) = *Aheap* (π · Γ) (π · *e*)
apply (*rule cfun-eqvtI*)
apply (*cases nonrec* π *rule*: *eqvt-cases*[**where** *x* = Γ])
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** π · *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*: *subsetD*[*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*!: *subsetD*[*OF Afix-edom*] *subsetD*[*OF Aexp-edom*] *subsetD*[*OF thunks-domA*])

lemma *Aexp-Let-simp2*:
x ∉ *fv e* ⟹ $\mathcal{A}_a$(*let x be e in exp*) = *env-delete x* ($\mathcal{A}^{\perp}{}_{ABind\text{-}nonrec\ x\ e}$ · ($\mathcal{A}_a$ *exp*, *CCexp exp*·*a*) *e* ⊔ $\mathcal{A}_a$ *exp*)
unfolding *Aexp-pre-simps env-delete-restr*
by (*rule env-restr-cong*) (*auto dest*!: *subsetD*[*OF fup-Aexp-edom*] *subsetD*[*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 ($\mathcal{A}^{\perp}{}_{ABind\text{-}nonrec\ x\ e'}$·($\mathcal{A}_a$ *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

13.2 CoCallImplSafe

theory *CoCallImplSafe*
imports *CoCallAnalysisImpl CoCallAnalysisSpec ArityAnalysisFixProps*
begin

locale *CoCallImplSafe*
begin
sublocale *CoCallAnalysisImpl*.

lemma *ccNeighbors-Int-ccrestr*: (*ccNeighbors x G* ∩ *S*) = *ccNeighbors x* (*cc-restr* (*insert x S*) *G*) ∩ *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*, *opaque-lifting*) *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 ≠ x' y ≠ x' by (simp-all add: fresh-at-base)

  note Let-nonrec(1,2)[simp]

  from ‹x' ∉ fv e› ‹y ≠ x'› ‹x ≠ x'›
  have [simp]: x' ∉ fv (e[y::=x])
    by (auto simp add: fv-subst-eq)

  note ‹x' ∉ fv e›[simp] ‹y ≠ x'› [simp]‹x ≠ x'›  [simp]

  case [simp]: 1
```

have $\bigwedge$ *a. cc-restr* $\{x'\}$ (*CCexp exp*[*y*::=*x*]·*a*) = *cc-restr* $\{x'\}$ (*CCexp exp*·*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*]·*a* $\longleftrightarrow$ $x'--x' \in$ *CCexp exp*·*a* **by** *auto*

have [*simp*]: $\bigwedge$ *a. Aexp e*[*y*::=*x*]·*a f*|‘ *S* = *Aexp e*·*a f*|‘ *S*
by (*rule Let-nonrec*(*5*)) *auto*

have [*simp*]: $\bigwedge$ *a. fup*·(*Aexp e*[*y*::=*x*])·*a f*|‘ *S* = *fup*·(*Aexp e*)·*a f*|‘ *S*
by (*case-tac a*) *auto*

have [*simp*]: *Aexp exp*[*y*::=*x*]·*a f*|‘ *S* = *Aexp exp*·*a f*|‘ *S*
by (*rule Let-nonrec*(*7*)) *auto*

have *Aexp exp*[*y*::=*x*]·*a f*|‘ $\{x'\}$ = *Aexp exp*·*a f*|‘ $\{x'\}$
by (*rule Let-nonrec*(*7*)) *auto*
from *fun-cong*[*OF this*, **where** $x = x'$]
have [*simp*]: (*Aexp exp*[*y*::=*x*]·*a*) x' = (*Aexp exp*·*a*) x' **by** *auto*

have [*simp*]: $\bigwedge$ *a. cc-restr S* (*CCexp exp*[*y*::=*x*]·*a*) = *cc-restr S* (*CCexp exp*·*a*)
by (*rule Let-nonrec*(*6*)) *auto*

have [*simp*]: $\bigwedge$ *a. cc-restr S* (*CCexp e*[*y*::=*x*]·*a*) = *cc-restr S* (*CCexp e*·*a*)
by (*rule Let-nonrec*(*4*)) *auto*

have [*simp*]: $\bigwedge$ *a. cc-restr S* (*fup*·(*CCexp e*[*y*::=*x*])·*a*) = *cc-restr S* (*fup*·(*CCexp e*)·*a*)
by (*rule fup-ccExp-restr-subst'*) *simp*

have [*simp*]: *fv e*[*y*::=*x*] ∩ *S* = *fv e* ∩ *S*
by (*auto simp add*: *fv-subst-eq*)

have [*simp*]:
ccNeighbors x' (*CCexp exp*[*y*::=*x*]·*a*) ∩ − $\{x'\}$ ∩ *S* = *ccNeighbors x'* (*CCexp exp*·*a*) ∩ − $\{x'\}$ ∩ *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: subsetD[OF edom-esing-subset] elim!: nonrecE)
next
  fix x y :: var and Γ :: heap and e :: exp
  assume assms: x ∉ domA Γ  y ∉ domA Γ

  from Aexp-restr-subst[OF assms(2,1)]
  have **: ⋀ a. Aexp e[x::=y]·a f|‘ domA Γ = Aexp e·a f|‘ domA Γ.

  show Aheap Γ[x::h=y] e[x::=y] = Aheap Γ e
  proof(cases nonrec Γ)
    case [simp]: False

    from assms
    have atom ‘ domA Γ ♯* x and atom ‘ domA Γ ♯* y
      by (auto simp add: fresh-star-at-base image-iff)
```

```
    hence [simp]: ¬ nonrec (Γ[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 Γ ♯* x and atom ‘ domA Γ ♯* y
      by (auto simp add: fresh-star-at-base image-iff)
    with True
    have *: nonrec (Γ[x::h=y]) by (simp add: nonrec-subst)

    from True
    obtain x' e' where [simp]: Γ = [(x',e')] x' ∉ fv e' by (auto elim: nonrecE)

    from * have [simp]: x' ∉ fv (e'[x::=y])
      by (auto simp add: nonrec-def)

    from fun-cong[OF **, where x = x']
    have [simp]: ⋀ a. (Aexp e[x::=y]·a) x' = (Aexp e·a) x' by simp

    from CCexp-subst[OF assms(2,1)]
    have ⋀ a.  cc-restr {x'} (CCexp e[x::=y]·a) = cc-restr {x'} (CCexp e·a) by simp
    from arg-cong[where f = λx.  x'--x'∈x, OF this]
    have [simp]: ⋀ a. x'--x'∈(CCexp e[x::=y]·a) ⟷ x'--x'∈(CCexp e·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*]: $\Gamma = [(x,e')]\ x \notin fv\ e'$ **by** (*auto elim*: *nonrecE*)

hence $\bigwedge a.\ x \notin edom\ (fup{\cdot}(Aexp\ e'){\cdot}a)$
by (*auto dest*:*subsetD*[*OF fup-Aexp-edom*])
hence [*simp*]: $\bigwedge a.\ (fup{\cdot}(Aexp\ e'){\cdot}a)\ x = \bot$ **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*:
¬ *nonrec* Γ ⟹ *ccHeap* Γ *e*·*a* = *CCfix* Γ·(*Afix* Γ·(*Aexp e*·*a* ⊔ (λ-.*up*·*0*) *f*|‘ (*thunks* Γ)), *CCexp e*·*a*)
by (*simp add*: *ccHeap-def ccHeap-rec-eq*)

lemma *ccHeap-simp2*:
x ∉ *fv e* ⟹ *ccHeap* [(*x*,*e*)] *exp*·*n* = *CCfix-nonrec x e*·(*Aexp exp*·*n*, *CCexp exp*·*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·(inc·a) ⊔ ccProd {x} (insert x (fv e)) ⊑ CCexp (App e x)·a*
by *simp*
next
fix *y e n*
show *cc-restr (fv (Lam [y]. e)) (CCexp e·(pred·n)) ⊑ CCexp (Lam [y]. e)·n*
by (*auto simp add*: *CCexp-pre-simps predCC-eq dest*!: *subsetD*[*OF ccField-cc-restr*] *simp del*: *CCexp-simps*)
next
fix *x y* :: *var* **and** *S e a*
assume *x* ∉ *S* **and** *y* ∉ *S*
thus *cc-restr S (CCexp e[y::=x]·a) ⊑ cc-restr S (CCexp e·a)*
by (*rule eq-imp-below*[*OF CCexp-subst*])
next
fix *e*
assume *isVal e*
thus *CCexp e·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·a) ⊑ CCexp (Let* Γ *e)·a*
proof(*cases nonrec* Γ)
case *False*
thus *cc-restr (− domA* Γ*) (ccHeap* Γ *e·a) ⊑ CCexp (Let* Γ *e)·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·a ⊑ ccHeap* Δ *e·a*
by (*cases nonrec* Δ)
(*auto simp add*: *ccHeap-simp1 ccHeap-simp2 arg-cong*[*OF CCfix-unroll*, **where** *f* = (⊑) *x* **for** *x*] *elim*!: *nonrecE*)

fix *x e′ a′*
assume *map-of* Δ *x = Some e′*
hence [*simp*]: *x* ∈ *domA* Δ **by** (*metis domI dom-map-of-conv-domA*)
assume (*Aheap* Δ *e·a*) *x = up·a′*
show *CCexp e′·a′ ⊑ ccHeap* Δ *e·a*
proof(*cases nonrec* Δ)
case *False*

from ‹(*Aheap* Δ *e·a*) *x = up·a′*› *False*
have (*Afix* Δ·(*Aexp e·a ⊔* (λ-.*up·0*)*f*|‘ (*thunks* Δ))) *x = up·a′*

```
    by (simp add: Aheap-def)
  hence CCexp e'·a' ⊑ ccBind x e'·(Afix Δ·(Aexp e·a ⊔ (λ-.up·0)f|‘ (thunks Δ)), CCfix Δ·(Afix
Δ·(Aexp e·a ⊔ (λ-.up·0)f|‘ (thunks Δ)), CCexp e·a))
    by (auto simp add: ccBind-eq dest: subsetD[OF ccField-CCexp])
   also
   have ccBind x e'·(Afix Δ·(Aexp e·a ⊔ (λ-.up·0)f|‘ (thunks Δ)), CCfix Δ·(Afix Δ·(Aexp e·a
⊔ (λ-.up·0)f|‘ (thunks Δ)), CCexp e·a)) ⊑  ccHeap Δ e·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 = (⊑) x for x ]
                intro: below-trans[OF - join-above2])
   finally
   show CCexp e'·a' ⊑ ccHeap Δ e·a by this simp-all
 next
   case True
   with ‹map-of Δ x = Some e'›
   have [simp]: Δ = [(x,e')] x ∉ fv e' by (auto elim!: nonrecE split: if-splits)

   show ?thesis
   proof(cases x−−x∉CCexp e·a ∨ isVal e')
     case True
     with ‹(Aheap Δ e·a) x = up·a'›
     have [simp]: (CoCallArityAnalysis.Aexp cCCexp e·a) x = up·a'
       by (auto simp add: Aheap-nonrec-simp ABind-nonrec-eq split: if-splits)

     have CCexp e'·a' ⊑ 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 be-
low-trans[OF - join-above2] simp del: below-ccSquare )
   next
     case False

     from ‹(Aheap Δ e·a) x = up·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·a) − {x} ∩ thunks Δ) ⊑ ccHeap Δ e·a
 proof (cases nonrec Δ)
   case [simp]: False
```

have *ccProd (fv e′) (ccNeighbors x (ccHeap Δ e·a) − {x} ∩ thunks Δ) ⊑ ccProd (fv e′) (ccNeighbors x (ccHeap Δ e·a))*
by (*rule ccProd-mono2*) *auto*
also have ... ⊑ (⨆ *x↦e′∈map-of Δ. ccProd (fv e′) (ccNeighbors x (ccHeap Δ e·a))*)
using ‹*map-of Δ x = Some e′*› **by** (*rule below-lubmapI*)
also have ... ⊑ *ccBindsExtra Δ·(Afix Δ·(Aexp e·a ⊔ (λ-.up·0)f|‘ (thunks Δ)), ccHeap Δ e·a*)
by (*simp add*: *ccBindsExtra-simp below-trans*[*OF - join-above2*])
also have ... ⊑ *ccHeap Δ e·a*
by (*simp add*: *ccHeap-simp1 arg-cong*[*OF CCfix-unroll*, **where** *f* = (⊑) *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 ∉ fv e′* **by** (*auto elim*!: *nonrecE split*: *if-splits*)

have [*simp*]: (*ccNeighbors x (ccBind x e′·(Aexp e·a, CCexp e·a))*) = {}
by (*auto simp add*: *ccBind-eq dest*!: *subsetD*[*OF ccField-cc-restr*] *subsetD*[*OF ccField-fup-CCexp*])

show *?thesis*
proof(*cases isVal e′ ∧ x−−x∈CCexp e·a*)
case *True*

have *ccNeighbors x (ccHeap Δ e·a) =*
ccNeighbors x (ccBind x e′·(Aheap-nonrec x e′·(Aexp e·a, CCexp e·a), CCexp e·a)) ∪
ccNeighbors x (ccProd (fv e′) (ccNeighbors x (CCexp e·a) − (if isVal e′ then {} else {x})))
∪
ccNeighbors x (CCexp e·a) **by** (*auto simp add*: *ccHeap-simp2*)
also have *ccNeighbors x (ccBind x e′·(Aheap-nonrec x e′·(Aexp e·a, CCexp e·a), CCexp e·a)) = {}*
by (*auto simp add*: *ccBind-eq dest*!: *subsetD*[*OF ccField-cc-restr*] *subsetD*[*OF ccField-fup-CCexp*])
also have *ccNeighbors x (ccProd (fv e′) (ccNeighbors x (CCexp e·a) − (if isVal e′ then {} else {x})))*
⊆ *ccNeighbors x (ccProd (fv e′) (ccNeighbors x (CCexp e·a)))* **by** (*simp add*: *ccNeighbors-ccProd*)
also have ... ⊆ *fv e′* **by** (*simp add*: *ccNeighbors-ccProd*)
finally
have *ccNeighbors x (ccHeap Δ e·a) − {x} ∩ thunks Δ ⊆ ccNeighbors x (CCexp e·a) ∪ fv e′* **by** *auto*
hence *ccProd (fv e′) (ccNeighbors x (ccHeap Δ e·a) − {x} ∩ thunks Δ) ⊑ ccProd (fv e′) (ccNeighbors x (CCexp e·a) ∪ fv e′)* **by** (*rule ccProd-mono2*)
also have ... ⊑ *ccProd (fv e′) (ccNeighbors x (CCexp e·a)) ⊔ ccProd (fv e′) (fv e′)* **by** *simp*
also have *ccProd (fv e′) (ccNeighbors x (CCexp e·a)) ⊑ ccHeap Δ e·a*
using ‹*map-of Δ x = Some e′*› ‹(*Aheap Δ e·a*) *x = up·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 ... ⊑ *ccHeap Δ e·a*

using ‹*map-of* Δ *x* = *Some e'*› ‹(*Aheap* Δ *e*·*a*) *x* = *up*·*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* Δ *e*·*a*) =
ccNeighbors x (*ccBind x e'*·(*Aheap-nonrec x e'*·(*Aexp e*·*a*, *CCexp e*·*a*), *CCexp e*·*a*)) ∪
ccNeighbors x (*ccProd* (*fv e'*) (*ccNeighbors x* (*CCexp e*·*a*) − (*if isVal e' then* {} *else* {*x*})))
∪
ccNeighbors x (*CCexp e*·*a*) **by** (*auto simp add*: *ccHeap-simp2*)
also have *ccNeighbors x* (*ccBind x e'*·(*Aheap-nonrec x e'*·(*Aexp e*·*a*, *CCexp e*·*a*), *CCexp e*·*a*)) = {}
by (*auto simp add*: *ccBind-eq dest*!: *subsetD*[*OF ccField-cc-restr*] *subsetD*[*OF ccField-fup-CCexp*])
also have *ccNeighbors x* (*ccProd* (*fv e'*) (*ccNeighbors x* (*CCexp e*·*a*) − (*if isVal e' then* {} *else* {*x*})))
= {} **using** *False* **by** (*auto simp add*: *ccNeighbors-ccProd*)
finally
have *ccNeighbors x* (*ccHeap* Δ *e*·*a*) ⊆ *ccNeighbors x* (*CCexp e*·*a*) **by** *auto*
hence*ccNeighbors x* (*ccHeap* Δ *e*·*a*) − {*x*} ∩ *thunks* Δ ⊆ *ccNeighbors x* (*CCexp e*·*a*) − {*x*} ∩ *thunks* Δ **by** *auto*
hence *ccProd* (*fv e'*) (*ccNeighbors x* (*ccHeap* Δ *e*·*a*) − {*x*} ∩ *thunks* Δ) ⊑ *ccProd* (*fv e'*) (*ccNeighbors x* (*CCexp e*·*a*) − {*x*} ∩ *thunks* Δ) **by** (*rule ccProd-mono2*)
also have ... ⊑ *ccHeap* Δ *e*·*a*
using ‹*map-of* Δ *x* = *Some e'*› ‹(*Aheap* Δ *e*·*a*) *x* = *up*·*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* Γ *e a*
assume [*simp*]: ¬ *nonrec* Γ
assume *x* ∈ *thunks* Γ
hence [*simp*]: *x* ∈ *domA* Γ **by** (*rule subsetD*[*OF thunks-domA*])
assume *x* ∈ *edom* (*Aheap* Γ *e*·*a*)

from ‹*x* ∈ *thunks* Γ›
have (*Afix* Γ·(*Aexp e*·*a* ⊔ (λ-.*up*·*0*)*f*|‘ (*thunks* Γ))) *x* = *up*·*0*
by (*subst Afix-unroll*) *simp*

thus (*Aheap* Γ *e*·*a*) *x* = *up*·*0* **by** *simp*
next
fix *x* Γ *e a*
assume *nonrec* Γ
then obtain *x' e'* **where** [*simp*]: Γ = [(*x'*,*e'*)] *x'* ∉ *fv e'* **by** (*auto elim*: *nonrecE*)
assume *x* ∈ *thunks* Γ
hence [*simp*]: *x* = *x'* ¬ *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 ‹$x \in thunks\ \Gamma$›
have $(Afix\ \Gamma{\cdot}(Aexp\ e{\cdot}a \sqcup (\lambda\text{-}.up{\cdot}0)f|`(thunks\ \Gamma)))\ x = up{\cdot}0$
by (*subst Afix-unroll*) *simp*

show $(Aheap\ \Gamma\ e{\cdot}a)\ x = up{\cdot}0$ **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

14 End-to-end Saftey Results and Example

14.1 CallArityEnd2End

theory *CallArityEnd2End*
imports *ArityTransform CoCallAnalysisImpl*
begin

locale *CallArityEnd2End*
begin
sublocale *CoCallAnalysisImpl*.

lemma *fresh-var-eqE*[*elim-format*]: $fresh\text{-}var\ e = x \Longrightarrow x \notin fv\ e$
by (*metis fresh-var-not-free*)

lemma *example1*:
fixes e :: *exp*
fixes $f\ g\ x\ y\ z$ :: *var*
assumes *Aexp-e*: $\bigwedge a.\ Aexp\ e{\cdot}a = esing\ x{\cdot}(up{\cdot}a) \sqcup esing\ y{\cdot}(up{\cdot}a)$
assumes *ccExp-e*: $\bigwedge a.\ CCexp\ e{\cdot}a = \bot$
assumes [*simp*]: $transform\ 1\ e = e$
assumes $isVal\ e$
assumes *disj*: $y \neq f\ y \neq g\ x \neq y\ z \neq f\ z \neq g\ y \neq x$
assumes *fresh*: $atom\ z \sharp e$
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 $x \in fv\ e$ **by** *simp* (*metis Aexp-edom' insert-subset*)
hence [*simp*]: $\neg\ nonrec\ [(x,e)]$

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

14.2 CallArityEnd2EndSafe

theory *CallArityEnd2EndSafe*
imports *CallArityEnd2End CardArityTransformSafe CoCallImplSafe CoCallImplTTreeSafe TTreeImplCardinalitySafe*
begin

locale *CallArityEnd2EndSafe*
begin
sublocale *CoCallImplSafe*.
sublocale *CallArityEnd2End*.

abbreviation *transform-syn'* ($\langle\mathcal{T}_\rangle$) **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*, []) $\Rightarrow^*$ (Γ,*v*,[]) **and** *isVal v*
obtains Γ' **and** v'
where ([], $\mathcal{T}_0$ *e*, []) $\Rightarrow^*$ (Γ',*v'*,[]) **and** *isVal v'*
and *card* (*domA* Γ') $\leq$ *card* (*domA* Γ)
proof−
note *assms*(*2*)
moreover
have $\neg$ *boring-step* (Γ,*v*,[]) **by** (*simp add*: *boring-step.simps*)
moreover
have *heap-upds-ok-conf* ([], *e*, []) **by** *simp*
moreover
have *consistent* (⊥,⊥,*0*,[],[]) ([], *e*, []) **using** *closed* **by** (*rule closed-consistent*)
ultimately
obtain *ae ce a as r* **where**
∗: *consistent* (*ae*, *ce*, *a*, *as*, *r*) (Γ,*v*,[]) **and**
∗∗: *conf-transform* (⊥, ⊥, *0*, [], []) ([],*e*,[]) $\Rightarrow_{G^*}$ *conf-transform* (*ae*, *ce*, *a*, *as*, *r*) (Γ,*v*,[])
by (*metis end2end*)

let *?*Γ = *map-transform Aeta-expand ae* (*map-transform transform ae* (*restrictA* (−*set r*) Γ))

let *?v* = *transform a v*

from ∗ **have** *set r* ⊆ *domA* Γ **by** *auto*

have *conf-transform* (⊥, ⊥, *0*, [], []) ([],*e*,[]) = ([],*transform 0 e*,[]) **by** *simp*
with ∗∗
have ([], *transform 0 e*, []) ⇒$_G$* (*?*Γ, *?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*: *subsetD*[*OF fv-transform*])

note *sestoftUnGC'*[*OF* ‹([], *transform 0 e*, []) ⇒$_G$* (*?*Γ, *?v*, *map Dummy* (*rev r*))› ‹*isVal ?v*›
‹*fv* (*transform 0 e*) = {}›]
then obtain Γ′
where ([], *transform 0 e*, []) ⇒* (Γ′, *?v*, [])
and *?*Γ = *restrictA* (− *set r*) Γ′
and *set r* ⊆ *domA* Γ′
by *auto*

have *card* (*domA* Γ) = *card* (*domA ?*Γ ∪ (*set r* ∩ *domA* Γ))
by (*rule arg-cong*[**where** *f* = *card*]) *auto*
also have ... = *card* (*domA ?*Γ) + *card* (*set r* ∩ *domA* Γ)
by (*rule card-Un-disjoint*) *auto*
also note ‹*?*Γ = *restrictA* (− *set r*) Γ′›
also have *set r* ∩ *domA* Γ = *set r* ∩ *domA* Γ′
using ‹*set r* ⊆ *domA* Γ› ‹*set r* ⊆ *domA* Γ′› **by** *auto*
also have *card* (*domA* (*restrictA* (− *set r*) Γ′)) + *card* (*set r* ∩ *domA* Γ′) = *card* (*domA* Γ′)
by (*subst card-Un-disjoint*[*symmetric*]) (*auto intro*: *arg-cong*[**where** *f* = *card*])
finally
have *card* (*domA* Γ′) ≤ *card* (*domA* Γ) **by** *simp*
with ‹([], *transform 0 e*, []) ⇒* (Γ′, *?v*, [])› ‹*isVal ?v*›
show *thesis* **using** *that* **by** *blast*
qed

lemma *fresh-var-eqE*[*elim-format*]: *fresh-var e* = *x* ⟹ *x* ∉ *fv e*
by (*metis fresh-var-not-free*)

lemma *example1*:
fixes *e* :: *exp*
fixes *f g x y z* :: *var*
assumes *Aexp-e*: ⋀*a*. *Aexp e*·*a* = *esing x*·(*up*·*a*) ⊔ *esing y*·(*up*·*a*)
assumes *ccExp-e*: ⋀*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

15 Functional Correctness of the Arity Analysis

15.1 ArityAnalysisCorrDenotational

theory *ArityAnalysisCorrDenotational*
imports *ArityAnalysisSpec Launchbury.Denotational AirityTransform*
begin

context *ArityAnalysisLetSafe*
begin

inductive *eq* :: *Arity* ⇒ *Value* ⇒ *Value* ⇒ *bool* **where**
eq 0 v v
| (⋀ *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*:
(⋀ *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* $\Longrightarrow$ *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*: $a \sqsubseteq a' \Longrightarrow$ *eq a' v1 v2* $\Longrightarrow$ *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* $\sqsubseteq$ *a'*› **have** *a* $\sqsubseteq$ *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* $\sqcup$ *a'*) *v1 v2* $\longleftrightarrow$ *eq a v1 v2* $\wedge$ *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* $\Longrightarrow$ *cont g* $\Longrightarrow$ *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* $\Rightarrow$ (*var* $\Rightarrow$ *Value*) $\Rightarrow$ (*var* $\Rightarrow$ *Value*) $\Rightarrow$ *bool* **where**
 eqϱI: ($\bigwedge$ *x a*. *ae x* = *up*·*a* $\Longrightarrow$ *eq a* (*ϱ1 x*) (*ϱ2 x*)) $\Longrightarrow$ *eqϱ ae ϱ1 ϱ2*

lemma *eqϱE*: *eqϱ ae ϱ1 ϱ2* $\Longrightarrow$ *ae x* = *up*·*a* $\Longrightarrow$ *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 (*ϱ1* +$+_S$ *ϱ2*) (*ϱ1′* +$+_S$*ϱ2′*) ⟷ *eqϱ ae* (*ϱ1 f*|‘ (− *S*)) (*ϱ1′ f*|‘ (− *S*)) ∧ *eqϱ ae* (*ϱ2 f*|‘ *S*) (*ϱ2′ f*|‘ *S*)
by (*auto simp add*: *lookup-env-restr-eq eqϱ.simps lookup-override-on-eq*)

lemma *Aexp-heap-below-Aheap*:
assumes (*Aheap* Γ *e·a*) *x* = *up·a′*
assumes *map-of* Γ *x* = *Some e′*
shows *Aexp e′·a′* ⊑ *Aheap* Γ *e·a* ⊔ *Aexp* (*Let* Γ *e*)*·a*
proof−

from *assms*(*1*)
have *Aexp e′·a′ = ABind x e′·*(*Aheap* Γ *e·a*)
by (*simp del*: *join-comm fun-meet-simp*)
also have ... ⊑ *ABinds* Γ·(*Aheap* Γ *e·a*)
by (*rule monofun-cfun-fun*[*OF ABind-below-ABinds*[*OF* ‹*map-of* - - = -›]])
also have ... ⊑ *ABinds* Γ·(*Aheap* Γ *e·a*) ⊔ *Aexp e·a*
by *simp*
also note *Aexp-Let*
finally
show *?thesis* **by** *this simp-all*
qed

lemma *Aexp-body-below-Aheap*:
shows *Aexp e·a* ⊑ *Aheap* Γ *e·a* ⊔ *Aexp* (*Let* Γ *e*)·*a*
by (*rule below-trans*[*OF join-above2 Aexp-Let*])

lemma *Aexp-correct*: *eqϱ* (*Aexp e·a*) *ϱ1 ϱ2* ⟹ *eq a* (⟦*e*⟧*ϱ1*) (⟦*e*⟧*ϱ2*)
proof(*induction a e arbitrary*: *ϱ1 ϱ2 rule*: *transform.induct*[*case-names App Lam Var Let Bool IfThenElse*])
case (*Var a x*)
from ‹*eqϱ* (*Aexp* (*Var x*)·*a*) *ϱ1 ϱ2*›
have *eqϱ* (*esing x*·(*up*·*a*)) *ϱ1 ϱ2* **by** (*rule eqϱ-mono*[*OF Aexp-Var-singleton*])
thus *?case* **by** *simp*
next
case (*App a e x*)
from ‹*eqϱ* (*Aexp* (*App e x*)·*a*) *ϱ1 ϱ2*›
have *eqϱ* (*Aexp e*·(*inc*·*a*) ⊔ *esing x*·(*up*·*0*)) *ϱ1 ϱ2* **by** (*rule eqϱ-mono*[*OF Aexp-App*])
hence *eqϱ* (*Aexp e*·(*inc*·*a*)) *ϱ1 ϱ2* **and** *ϱ1 x* = *ϱ2 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 ‹*eqϱ* (*Aexp* (*Lam* [*x*]. *e*)·*a*) *ϱ1 ϱ2*›
have *eqϱ* (*env-delete x* (*Aexp e*·(*pred*·*a*))) *ϱ1 ϱ2* **by** (*rule eqϱ-mono*[*OF Aexp-Lam*])
hence ⋀ *v*. *eqϱ* (*Aexp e*·(*pred*·*a*)) (*ϱ1*(*x* := *v*)) (*ϱ2*(*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 ‹*eqϱ* (*Aexp* (*scrut ?* e_1 : e_2)·*a*) *ϱ1 ϱ2*›
have *eqϱ* (*Aexp scrut*·*0* ⊔ *Aexp* e_1·*a* ⊔ *Aexp* e_2·*a*) *ϱ1 ϱ2* **by** (*rule eqϱ-mono*[*OF Aexp-IfThenElse*])
hence *eqϱ* (*Aexp scrut*·*0*) *ϱ1 ϱ2*
and *eqϱ* (*Aexp* e_1·*a*) *ϱ1 ϱ2*
and *eqϱ* (*Aexp* e_2·*a*) *ϱ1 ϱ2* **by** *simp-all*

```
  from IfThenElse(1)[OF this(1)] IfThenElse(2)[OF this(2)] IfThenElse(3)[OF this(3)]
  show ?case
    by (cases ⟦ scrut ⟧ϱ2) auto
next
  case (Let a Γ e)

  have eqϱ (Aheap Γ e·a ⊔ Aexp (Let Γ e)·a) (⦃Γ⦄ϱ1) (⦃Γ⦄ϱ2)
  proof(induction rule: parallel-HSem-ind[case-names adm bottom step])
    case adm thus ?case by (intro eqϱ-adm cont2cont)
  next
    case bottom show ?case by simp
  next
    case (step ϱ1' ϱ2')
    show ?case
    proof (rule eqϱI)
      fix x a'
      assume ass: (Aheap Γ e·a ⊔ Aexp (Let Γ e)·a) x = up·a'
      show eq a' ((ϱ1 ++domA Γ ⟦ Γ ⟧ϱ1') x) ((ϱ2 ++domA Γ ⟦ Γ ⟧ϱ2') x)
      proof(cases x ∈ 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·a) x = up·a' using ass by simp
         hence Aexp e'·a' ⊑ Aheap Γ e·a ⊔ Aexp (Let Γ e)·a using ‹map-of - - = -› by (rule
Aexp-heap-below-Aheap)
        hence eqϱ (Aexp e'·a') ϱ1' ϱ2' using step(1) by (rule eqϱ-mono)
        hence eq a' (⟦ e' ⟧ϱ1') (⟦ e' ⟧ϱ2')
          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 ∉ edom (Aheap Γ e·a) by blast
        hence (Aexp (Let Γ e)·a) x = up·a' using ass by (simp add: edomIff)
        with ‹eqϱ (Aexp (Let Γ e)·a) ϱ1 ϱ2›
        have eq a' (ϱ1 x) (ϱ2 x) by (auto elim: eqϱE)
        thus ?thesis by simp
      qed
    qed
  qed
  hence eqϱ (Aexp e·a) (⦃Γ⦄ϱ1) (⦃Γ⦄ϱ2) by (rule eqϱ-mono[OF Aexp-body-below-Aheap])
  hence eq a (⟦ e ⟧⦃Γ⦄ϱ1) (⟦ e ⟧⦃Γ⦄ϱ2) by (rule Let(2)[simplified])
  thus ?case by simp
qed

lemma ESem-ignores-fresh[simp]: ⟦ e ⟧ϱ(fresh-var e := v) = ⟦ e ⟧ϱ
  by (metis ESem-fresh-cong env-restr-fun-upd-other fresh-var-not-free)

lemma eq-Aeta-expand: eq a (⟦ Aeta-expand a e ⟧ϱ) (⟦e⟧ϱ)
  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* ⟧$_{\{\!|map\text{-}transform\ Aeta\text{-}expand\ (Aheap\ \Gamma\ e{\cdot}a)\ (map\text{-}transform\ tran}$
by *simp*
also have *eq a* ... (⟦ *e* ⟧$_{\{\!|map\text{-}transform\ Aeta\text{-}expand\ (Aheap\ \Gamma\ e{\cdot}a)\ (map\text{-}transform\ transform\ (Aheap\ \Gamma\ e{\cdot}a)\ \Gamma)|\!\}\varrho}$)
using *Let*(*2*) **by** *simp*
also have *eq a* ... (⟦ *e* ⟧$_{\{\!|\Gamma|\!\}\varrho}$)
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* $\varrho 1$ $\varrho 2$)
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) ⟧$_{\varrho 1}$) (⟦Γ⟧$_{\varrho 2}$)
proof(*rule eq*ϱ*I*)
fix *x a*′
assume *ass*: (*Aheap* Γ *e*·*a* ⊔ *Aexp* (*Let* Γ *e*)·*a*) *x* = *up*·*a*′
show *eq a*′ ((⟦ *map-transform Aeta-expand* (*Aheap* Γ *e*·*a*) (*map-transform transform* (*Aheap* Γ *e*·*a*) Γ) ⟧$_{\varrho 1}$) *x*) ((⟦Γ⟧$_{\varrho 2}$) *x*)
proof(*cases x* ∈ *domA* Γ)
case [*simp*]: *True*

```
      then obtain e' where [simp]: map-of Γ x = Some e' by (metis domA-map-of-Some-the)
        from ass have ass': (Aheap Γ e·a) x = up·a' by simp

         have (⟦ map-transform Aeta-expand (Aheap Γ e·a) (map-transform transform (Aheap
Γ e·a) Γ) ⟧ϱ1) x =
           ⟦Aeta-expand a' (transform a' e')⟧ϱ1
          by (simp add: lookupEvalHeap' map-of-map-transform ass')
         also have eq a' ... (⟦transform a' e'⟧ϱ1)
          by (rule eq-Aeta-expand)
         also have eq a' ... (⟦e'⟧ϱ1)
          by (rule Let(1)[OF map-of-SomeD[OF ‹map-of - - = -›]])
         also have eq a' ... (⟦e'⟧ϱ2)
         proof (rule Aexp-correct)
          from ass' ‹map-of - - = -›
          have Aexp e'·a' ⊑ Aheap Γ e·a ⊔ Aexp (Let Γ e)·a by (rule Aexp-heap-below-Aheap)
           thus eqϱ (Aexp e'·a') ϱ1 ϱ2 using step by (rule eqϱ-mono)
         qed
         also have ... = (⟦Γ⟧ϱ2) 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
eqϱ-join)
     qed
     thus eqϱ (Aexp e·a) (⦃⟦map-transform Aeta-expand (Aheap Γ e·a) (map-transform transform
(Aheap Γ e·a) Γ)⦄ϱ) (⦃⟦Γ⦄ϱ)
       by (rule eqϱ-mono[OF Aexp-body-below-Aheap])
  qed
  also have ... = ⟦ Let Γ e ⟧ϱ
    by simp
  finally show ?case.
qed

corollary Arity-transformation-correct':
  ⟦ 𝒯0 e ⟧ϱ = ⟦ e ⟧ϱ
  using Arity-transformation-correct[where a = 0] by simp

end
end
```