

The Safety of Call Arity

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

December 14, 2021

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 <i>expr</i>	Terms in [Bre13]
Stack	type-synonym <i>stack</i>	SestoftConf
Configuration	type-synonym <i>conf</i>	SestoftConf
Semantics (\Rightarrow)	inductive <i>step</i>	Sestoft
Arity	typedef <i>Arity</i>	Arity
Eta-expansion	lift-definition <i>Aeta-expand</i>	ArityEtaExpansion

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

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	7
1.4	AList-Utills-HOLCF	8
1.5	List-Interleavings	9
2	Small-step Semantics	10
2.1	SestoftConf	10
	2.1.1 Invariants of the semantics	14
2.2	Sestoft	16
	2.2.1 Equivariance	17
	2.2.2 Invariants	17
2.3	SestoftGC	18
2.4	BalancedTraces	20
2.5	SestoftCorrect	22
3	Arity	23
3.1	Arity	23
3.2	AEnv	26
3.3	Arity-Nominal	26
3.4	ArityStack	27
4	Eta-Expansion	27
4.1	EtaExpansion	27
4.2	EtaExpansionSafe	28
4.3	TransformTools	29
4.4	ArityEtaExpansion	31

4.5	ArityEtaExpansionSafe	32
5	Arity Analysis	32
5.1	ArityAnalysisSig	32
5.2	ArityAnalysisAbinds	33
5.2.1	Lifting arity analysis to recursive groups	33
5.3	ArityAnalysisSpec	35
5.4	TrivialArityAnal	36
5.5	ArityAnalysisStack	37
5.6	ArityAnalysisFix	38
5.7	ArityAnalysisFixProps	40
6	Arity Transformation	41
6.1	AbstractTransform	41
6.2	ArityTransform	43
7	Arity Analysis Safety (without Cardinality)	44
7.1	ArityConsistent	44
7.2	ArityTransformSafe	46
8	Cardinality Analysis	48
8.1	Cardinality-Domain	48
8.2	CardinalityAnalysisSig	49
8.3	CardinalityAnalysisSpec	50
8.4	NoCardinalityAnalysis	51
8.5	CardArityTransformSafe	53
9	Trace Trees	55
9.1	TTree	55
9.1.1	Prefix-closed sets of lists	55
9.1.2	The type of infinite labeled trees	56
9.1.3	Deconstructors	56
9.1.4	Trees as set of paths	56
9.1.5	The carrier of a tree	57
9.1.6	Repeatable trees	57
9.1.7	Simple trees	57
9.1.8	Intersection of two trees	59
9.1.9	Disjoint union of trees	59
9.1.10	Merging of trees	60
9.1.11	Removing elements from a tree	62
9.1.12	Multiple variables, each called at most once	63
9.1.13	Substituting trees for every node	64
9.2	TTree-HOLCF	68

10 Trace Tree Cardinality Analysis	72
10.1 AnalBinds	72
10.2 TTreeAnalysisSig	74
10.3 Cardinality-Domain-Lists	74
10.4 TTreeAnalysisSpec	76
10.5 TTreeImplCardinality	77
10.6 TTreeImplCardinalitySafe	77
11 Co-Call Graphs	79
11.1 CoCallGraph	79
11.2 CoCallGraph-Nominal	87
12 Co-Call Cardinality Analysis	88
12.1 CoCallAnalysisSig	88
12.2 CoCallAnalysisBinds	88
12.3 CoCallAritySig	90
12.4 CoCallAnalysisSpec	91
12.5 CoCallFix	91
12.5.1 The non-recursive case	94
12.5.2 Combining the cases	96
12.6 CoCallGraph-TTree	96
12.7 CoCallImplTTree	102
12.8 CoCallImplTTreeSafe	102
13 CoCall Cardinality Implementation	104
13.1 CoCallAnalysisImpl	104
13.2 CoCallImplSafe	108
14 End-to-end Saftey Results and Example	110
14.1 CallArityEnd2End	110
14.2 CallArityEnd2EndSafe	110
15 Functional Correctness of the Arity Analysis	111
15.1 ArityAnalysisCorrDenotational	111

1 Various Utilities

1.1 ConstOn

```
theory ConstOn
imports Main
begin
```

```
definition const-on :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b ⇒ bool
```

where $\text{const-on } f \ S \ x = (\forall \ y \in S . f \ y = x)$

lemma const-onI[intro] : $(\bigwedge y. y \in S \implies f \ y = x) \implies \text{const-on } f \ S \ x$
 $\langle \text{proof} \rangle$

lemma const-onD[dest] : $\text{const-on } f \ S \ x \implies y \in S \implies f \ y = x$
 $\langle \text{proof} \rangle$

lemma $\text{const-on-insert[simp]}$: $\text{const-on } f \ (\text{insert } x \ S) \ y \longleftrightarrow \text{const-on } f \ S \ y \wedge f \ x = y$
 $\langle \text{proof} \rangle$

lemma $\text{const-on-union[simp]}$: $\text{const-on } f \ (S \cup S') \ y \longleftrightarrow \text{const-on } f \ S \ y \wedge \text{const-on } f \ S' \ y$
 $\langle \text{proof} \rangle$

lemma $\text{const-on-subset[elim]}$: $\text{const-on } f \ S \ y \implies S' \subseteq S \implies \text{const-on } f \ S' \ y$
 $\langle \text{proof} \rangle$

end

1.2 Set-Cpo

theory Set-Cpo
imports HOLCF
begin

default-sort type

instantiation $\text{set} :: (\text{type}) \ \text{below}$

begin
 definition below-set **where** $(\sqsubseteq) = (\subseteq)$
 instance $\langle \text{proof} \rangle$
end

instance $\text{set} :: (\text{type}) \ \text{po}$
 $\langle \text{proof} \rangle$

lemma is-lub-set :
 $S \ll \bigcup S$
 $\langle \text{proof} \rangle$

lemma lub-set : $\text{lub } S = \bigcup S$
 $\langle \text{proof} \rangle$

instance $\text{set} :: (\text{type}) \ \text{cpo}$
 $\langle \text{proof} \rangle$

```

lemma minimal-set:  $\{\} \subseteq S$ 
  <proof>

instance set :: (type) pcpo
  <proof>

lemma set-contI:
  assumes  $\bigwedge Y. \text{chain } Y \implies f (\bigsqcup i. Y i) = \bigcup (f \text{ ` } \text{range } Y)$ 
  shows cont f
  <proof>

lemma set-set-contI:
  assumes  $\bigwedge S. f (\bigcup S) = \bigcup (f \text{ ` } S)$ 
  shows cont f
  <proof>

lemma adm-subseteq[simp]:
  assumes cont f
  shows adm  $(\lambda a. f a \subseteq S)$ 
  <proof>

lemma adm-Ball[simp]: adm  $(\lambda S. \forall x \in S. P x)$ 
  <proof>

lemma finite-subset-chain:
  fixes  $Y :: \text{nat} \Rightarrow 'a \text{ set}$ 
  assumes chain Y
  assumes  $S \subseteq \bigcup (Y \text{ ` } \text{UNIV})$ 
  assumes finite S
  shows  $\exists i. S \subseteq Y i$ 
  <proof>

lemma diff-cont[THEN cont-compose, simp, cont2cont]:
  fixes  $S' :: 'a \text{ set}$ 
  shows cont  $(\lambda S. S - S')$ 
  <proof>

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  $(\lambda f. \text{edom } f)$ 
  <proof>

```

end

1.4 AList-Utills-HOLCF

theory *AList-Utills-HOLCF*

imports *Launchbury.HOLCF-Utills Launchbury.HOLCF-Join-Classes Launchbury.AList-Utills*
begin

syntax

$-BLubMap :: [pttrn, pttrn, 'a \rightarrow 'b, 'b] \Rightarrow 'b ((\exists \sqcup / \mapsto / \in / \cdot / -) [0, 0, 0, 10] 10)$

translations

$\sqcup k \mapsto v \in m. e == CONST\ lub\ (CONST\ mapCollect\ (\lambda k\ v.\ e)\ m)$

lemma *below-lubmapI*[*intro*]:

$m\ k = Some\ v \Longrightarrow (e\ k\ v :: 'a :: Join-cpo) \sqsubseteq (\sqcup k \mapsto v \in m. e\ k\ v)$
<proof>

lemma *lubmap-belowI*[*intro*]:

$(\bigwedge k\ v.\ m\ k = Some\ v \Longrightarrow (e\ k\ v :: 'a :: Join-cpo) \sqsubseteq u) \Longrightarrow (\sqcup k \mapsto v \in m. e\ k\ v) \sqsubseteq u$
<proof>

lemma *lubmap-const-bottom*[*simp*]:

$(\sqcup k \mapsto v \in m. \perp) = (\perp :: 'a :: Join-cpo)$
<proof>

lemma *lubmap-map-upd*[*simp*]:

fixes $e :: 'a \Rightarrow 'b \Rightarrow ('c :: Join-cpo)$
shows $(\sqcup k \mapsto v \in m (k' \mapsto v'). e\ k\ v) = e\ k'\ v' \sqcup (\sqcup k \mapsto v \in m (k' := None). e\ k\ v)$
<proof>

lemma *lubmap-below-cong*:

assumes $\bigwedge k\ v.\ m\ k = Some\ v \Longrightarrow f1\ k\ v \sqsubseteq (f2\ k\ v :: 'a :: Join-cpo)$
shows $(\sqcup k \mapsto v \in m. f1\ k\ v) \sqsubseteq (\sqcup k \mapsto v \in m. f2\ k\ v)$
<proof>

lemma *cont2cont-lubmap*[*simp*, *cont2cont*]:

assumes $(\bigwedge k\ v.\ cont\ (f\ k\ v))$
shows $cont\ (\lambda x.\ \sqcup k \mapsto v \in m. (f\ k\ v\ x) :: 'a :: Join-cpo)$
<proof>

end

1.5 List-Interleavings

theory *List-Interleavings*

imports *Main*

begin

inductive *interleave'* :: 'a list \Rightarrow 'a list \Rightarrow 'a list \Rightarrow bool

where [*simp*]: *interleave'* [] [] []

 | *interleave'* xs ys zs \Longrightarrow *interleave'* (x#xs) ys (x#zs)

 | *interleave'* xs ys zs \Longrightarrow *interleave'* xs (x#ys) (x#zs)

definition *interleave* :: 'a list \Rightarrow 'a list \Rightarrow 'a list set (**infixr** \otimes 64)

where $xs \otimes ys = \text{Collect } (\text{interleave}' \text{ xs } ys)$

lemma *elim-interleave'*[*pred-set-conv*]: *interleave'* xs ys zs \longleftrightarrow $zs \in xs \otimes ys$ *<proof>*

lemmas *interleave-intros*[*intro?*] = *interleave'*.*intros*[*to-set*]

lemmas *interleave-intros*(1)[*simp*]

lemmas *interleave-induct*[*consumes 1, induct set: interleave, case-names Nil left right*] = *interleave'*.*induct*[*to-set*]

lemmas *interleave-cases*[*consumes 1, cases set: interleave*] = *interleave'*.*cases*[*to-set*]

lemmas *interleave-simps* = *interleave'*.*simps*[*to-set*]

inductive-cases *interleave-ConsE*[*elim*]: $(x\#xs) \in ys \otimes zs$

inductive-cases *interleave-ConsConsE*[*elim*]: $xs \in y\#ys \otimes z\#zs$

inductive-cases *interleave-ConsE2*[*elim*]: $xs \in x\#ys \otimes zs$

inductive-cases *interleave-ConsE3*[*elim*]: $xs \in ys \otimes x\#zs$

lemma *interleave-comm*: $xs \in ys \otimes zs \Longrightarrow xs \in zs \otimes ys$
<proof>

lemma *interleave-Nil1*[*simp*]: $[] \otimes xs = \{xs\}$
<proof>

lemma *interleave-Nil2*[*simp*]: $xs \otimes [] = \{xs\}$
<proof>

lemma *interleave-nil-simp*[*simp*]: $[] \in xs \otimes ys \longleftrightarrow xs = [] \wedge ys = []$
<proof>

lemma *append-interleave*: $xs @ ys \in xs \otimes ys$
<proof>

lemma *interleave-assoc1*: $a \in xs \otimes ys \Longrightarrow b \in a \otimes zs \Longrightarrow \exists c. c \in ys \otimes zs \wedge b \in xs \otimes c$
<proof>

lemma *interleave-assoc2*: $a \in ys \otimes zs \Longrightarrow b \in xs \otimes a \Longrightarrow \exists c. c \in xs \otimes ys \wedge b \in c \otimes zs$
<proof>

lemma *interleave-set*: $zs \in xs \otimes ys \Longrightarrow \text{set } zs = \text{set } xs \cup \text{set } ys$

<proof>

lemma *interleave-tl*: $xs \in ys \otimes zs \implies tl\ xs \in tl\ ys \otimes zs \vee tl\ xs \in ys \otimes (tl\ zs)$

<proof>

lemma *interleave-butlast*: $xs \in ys \otimes zs \implies butlast\ xs \in butlast\ ys \otimes zs \vee butlast\ xs \in ys \otimes (butlast\ zs)$

<proof>

lemma *interleave-take*: $zs \in xs \otimes ys \implies \exists\ n_1\ n_2. n = n_1 + n_2 \wedge take\ n\ zs \in take\ n_1\ xs \otimes take\ n_2\ ys$

<proof>

lemma *filter-interleave*: $xs \in ys \otimes zs \implies filter\ P\ xs \in filter\ P\ ys \otimes filter\ P\ zs$

<proof>

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

<proof>

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)*

<proof>

termination *<proof>*

lemmas *list-induct2'' = foo.induct[case-names NilNil ConsNil NilCons ConsCons]*

lemma *interleave-filter*:

assumes $xs \in filter\ P\ ys \otimes filter\ P\ zs$

obtains xs' **where** $xs' \in ys \otimes zs$ **and** $xs = filter\ P\ xs'$

<proof>

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 $\pi \cdot x = (\text{case } x \text{ of } (\text{Alts } e1 \ e2) \Rightarrow \text{Alts } (\pi \cdot e1) (\pi \cdot e2) \mid (\text{Arg } v) \Rightarrow \text{Arg } (\pi \cdot v) \mid (\text{Upd } v) \Rightarrow \text{Upd } (\pi \cdot v) \mid (\text{Dummy } v) \Rightarrow \text{Dummy } (\pi \cdot v))$
instance
 $\langle \text{proof} \rangle$
end

lemma $\text{Alts-eqvt}[\text{eqvt}]: \pi \cdot (\text{Alts } e1 \ e2) = \text{Alts } (\pi \cdot e1) (\pi \cdot e2)$
and $\text{Arg-eqvt}[\text{eqvt}]: \pi \cdot (\text{Arg } v) = \text{Arg } (\pi \cdot v)$
and $\text{Upd-eqvt}[\text{eqvt}]: \pi \cdot (\text{Upd } v) = \text{Upd } (\pi \cdot v)$
and $\text{Dummy-eqvt}[\text{eqvt}]: \pi \cdot (\text{Dummy } v) = \text{Dummy } (\pi \cdot v)$
 $\langle \text{proof} \rangle$

lemma $\text{supp-Alts}[\text{simp}]: \text{supp } (\text{Alts } e1 \ e2) = \text{supp } e1 \cup \text{supp } e2 \ \langle \text{proof} \rangle$
lemma $\text{supp-Arg}[\text{simp}]: \text{supp } (\text{Arg } v) = \text{supp } v \ \langle \text{proof} \rangle$
lemma $\text{supp-Upd}[\text{simp}]: \text{supp } (\text{Upd } v) = \text{supp } v \ \langle \text{proof} \rangle$
lemma $\text{supp-Dummy}[\text{simp}]: \text{supp } (\text{Dummy } v) = \text{supp } v \ \langle \text{proof} \rangle$
lemma $\text{fresh-Alts}[\text{simp}]: a \# \text{Alts } e1 \ e2 = (a \# e1 \wedge a \# e2) \ \langle \text{proof} \rangle$
lemma $\text{fresh-star-Alts}[\text{simp}]: a \#* \text{Alts } e1 \ e2 = (a \#* e1 \wedge a \#* e2) \ \langle \text{proof} \rangle$
lemma $\text{fresh-Arg}[\text{simp}]: a \# \text{Arg } v = a \# v \ \langle \text{proof} \rangle$
lemma $\text{fresh-Upd}[\text{simp}]: a \# \text{Upd } v = a \# v \ \langle \text{proof} \rangle$
lemma $\text{fresh-Dummy}[\text{simp}]: a \# \text{Dummy } v = a \# v \ \langle \text{proof} \rangle$
lemma $\text{fv-Alts}[\text{simp}]: \text{fv } (\text{Alts } e1 \ e2) = \text{fv } e1 \cup \text{fv } e2 \ \langle \text{proof} \rangle$
lemma $\text{fv-Arg}[\text{simp}]: \text{fv } (\text{Arg } v) = \text{fv } v \ \langle \text{proof} \rangle$
lemma $\text{fv-Upd}[\text{simp}]: \text{fv } (\text{Upd } v) = \text{fv } v \ \langle \text{proof} \rangle$
lemma $\text{fv-Dummy}[\text{simp}]: \text{fv } (\text{Dummy } v) = \text{fv } v \ \langle \text{proof} \rangle$

instance $\text{stack-elem} :: \text{fs}$
 $\langle \text{proof} \rangle$

type-synonym $\text{stack} = \text{stack-elem list}$

fun $\text{ap} :: \text{stack} \Rightarrow \text{var set}$ **where**
 $\text{ap } [] = \{\}$
 $\mid \text{ap } (\text{Alts } e1 \ e2 \ \# \ S) = \text{ap } S$
 $\mid \text{ap } (\text{Arg } x \ \# \ S) = \text{insert } x (\text{ap } S)$
 $\mid \text{ap } (\text{Upd } x \ \# \ S) = \text{ap } S$
 $\mid \text{ap } (\text{Dummy } x \ \# \ S) = \text{ap } S$
fun $\text{upds} :: \text{stack} \Rightarrow \text{var set}$ **where**
 $\text{upds } [] = \{\}$
 $\mid \text{upds } (\text{Alts } e1 \ e2 \ \# \ S) = \text{upds } S$
 $\mid \text{upds } (\text{Upd } x \ \# \ S) = \text{insert } x (\text{upds } S)$
 $\mid \text{upds } (\text{Arg } x \ \# \ S) = \text{upds } S$
 $\mid \text{upds } (\text{Dummy } x \ \# \ S) = \text{upds } S$
fun $\text{dummies} :: \text{stack} \Rightarrow \text{var set}$ **where**
 $\text{dummies } [] = \{\}$
 $\mid \text{dummies } (\text{Alts } e1 \ e2 \ \# \ S) = \text{dummies } S$
 $\mid \text{dummies } (\text{Upd } x \ \# \ S) = \text{dummies } S$
 $\mid \text{dummies } (\text{Arg } x \ \# \ S) = \text{dummies } S$

| *dummies* (*Dummy* $x \# S$) = *insert* x (*dummies* S)

fun *flattn* :: *stack* \Rightarrow *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* \Rightarrow *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

<proof>

lemma *ups-fv-subset*: *ups* $S \subseteq$ *fv* S

<proof>

lemma *fresh-distinct-ups*: *atom* ‘ $V \#* S \implies V \cap$ *ups* $S = \{\}$

<proof>

lemma *ap-fv-subset*: *ap* $S \subseteq$ *fv* S

<proof>

lemma *dummies-fv-subset*: *dummies* $S \subseteq$ *fv* S

<proof>

lemma *fresh-flattn[simp]*: *atom* ($a::$ *var*) $\#$ *flattn* $S \longleftrightarrow$ *atom* $a \# S$

<proof>

lemma *fresh-star-flattn[simp]*: *atom* ‘ ($as::$ *var set*) $\#* \text{flattn } S \longleftrightarrow$ *atom* ‘ $as \#* S$

<proof>

lemma *fresh-upds-list[simp]*: *atom* $a \# S \implies$ *atom* ($a::$ *var*) $\#$ *upds-list* S

<proof>

lemma *fresh-star-upds-list[simp]*: *atom* ‘ ($as::$ *var set*) $\#* S \implies$ *atom* ‘ ($as::$ *var set*) $\#* \text{upds-list } S$

<proof>

lemma *upds-append[simp]*: *upds* ($S@S'$) = *upds* $S \cup$ *upds* S'

<proof>

lemma *upds-map-Dummy[simp]*: *upds* (*map* *Dummy* l) = $\{\}$

<proof>

lemma *upds-list-append[simp]*: *upds-list* ($S@S'$) = *upds-list* $S @$ *upds-list* S'

<proof>

lemma *upds-list-map-Dummy[simp]*: *upds-list* (*map* *Dummy* l) = []

<proof>

lemma *dummies-append[simp]*: *dummies* ($S@S'$) = *dummies* $S \cup$ *dummies* S'

<proof>

lemma *dummies-map-Dummy*[simp]: $dummies (map Dummy l) = set l$
 ⟨proof⟩

lemma *map-Dummy-inj*[simp]: $map Dummy l = map Dummy l' \longleftrightarrow l = l'$
 ⟨proof⟩

type-synonym *conf* = $(heap \times exp \times stack)$

inductive *boring-step* **where**
isVal $e \implies boring\text{-}step (\Gamma, e, Upd\ x \# S)$

fun *restr-stack* :: $var\ set \Rightarrow stack \Rightarrow stack$
where *restr-stack* $V [] = []$
 | *restr-stack* $V (Alts\ e1\ e2 \# S) = Alts\ e1\ e2 \# restr\text{-}stack\ V\ S$
 | *restr-stack* $V (Arg\ x \# S) = Arg\ x \# restr\text{-}stack\ V\ S$
 | *restr-stack* $V (Upd\ x \# S) = (if\ x \in V\ then\ Upd\ x \# restr\text{-}stack\ V\ S\ else\ restr\text{-}stack\ V\ S)$
 | *restr-stack* $V (Dummy\ x \# S) = Dummy\ x \# restr\text{-}stack\ V\ S$

lemma *restr-stack-cong*:
 $(\bigwedge x. x \in upds\ S \implies x \in V \longleftrightarrow x \in V') \implies restr\text{-}stack\ V\ S = restr\text{-}stack\ V'\ S$
 ⟨proof⟩

lemma *upds-restr-stack*[simp]: $upds (restr\text{-}stack\ V\ S) = upds\ S \cap V$
 ⟨proof⟩

lemma *fresh-star-restrict-stack*[intro]:
 $a \#* S \implies a \#* restr\text{-}stack\ V\ S$
 ⟨proof⟩

lemma *restr-stack-restr-stack*[simp]:
 $restr\text{-}stack\ V (restr\text{-}stack\ V'\ S) = restr\text{-}stack\ (V \cap V')\ S$
 ⟨proof⟩

lemma *Upd-eq-restr-stackD*:
assumes $Upd\ x \# S = restr\text{-}stack\ V\ S'$
shows $x \in V$
 ⟨proof⟩

lemma *Upd-eq-restr-stackD2*:
assumes $restr\text{-}stack\ V\ S' = Upd\ x \# S$
shows $x \in V$
 ⟨proof⟩

lemma *restr-stack-noop*[simp]:
 $restr\text{-}stack\ V\ S = S \longleftrightarrow upds\ S \subseteq V$
 ⟨proof⟩

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 ⇒ rel x y ⇒ I x ⇒ I y ⟨*proof*⟩

lemma *invariant-starE*:
rtranclp rel x y ⇒ *invariant* rel I ⇒ I x ⇒ I y
 ⟨*proof*⟩

lemma *invariant-True*:
invariant rel (λ -. True)
 ⟨*proof*⟩

lemma *invariant-conj*:
invariant rel I1 ⇒ *invariant* rel I2 ⇒ *invariant* rel (λ x. I1 x ∧ I2 x)
 ⟨*proof*⟩

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 (∧ y z. *r*** a y ⇒ r y z ⇒ I y ⇒ I z ⇒ P y ⇒ P z)
shows P b
 ⟨*proof*⟩

fun *closed* :: *conf* ⇒ bool
where *closed* (Γ, e, S) ⇔ fv (Γ, e, S) ⊆ domA Γ ∪ upds S

fun *heap-upds-ok* **where** *heap-upds-ok* (Γ, S) ⇔ domA Γ ∩ upds S = {} ∧ *distinct* (upds-list S)

abbreviation *heap-upds-ok-conf* :: *conf* ⇒ bool
where *heap-upds-ok-conf* c ≡ *heap-upds-ok* (fst c, snd (snd c))

lemma *heap-upds-okE*: *heap-upds-ok* (Γ, S) ⇒ x ∈ domA Γ ⇒ x ∉ upds S
 ⟨*proof*⟩

lemma *heap-upds-ok-Nil*[*simp*]: *heap-upds-ok* (Γ, []) ⟨*proof*⟩

lemma *heap-upds-ok-app1*: *heap-upds-ok* (Γ, S) ⇒ *heap-upds-ok* (Γ, Arg x # S) ⟨*proof*⟩

lemma *heap-upds-ok-app2*: *heap-upds-ok* (Γ, Arg x # S) ⇒ *heap-upds-ok* (Γ, S) ⟨*proof*⟩

lemma *heap-upds-ok-alts1*: *heap-upds-ok* (Γ, S) ⇒ *heap-upds-ok* (Γ, Alts e1 e2 # S) ⟨*proof*⟩

lemma *heap-upds-ok-alts2*: *heap-upds-ok* (Γ, Alts e1 e2 # S) ⇒ *heap-upds-ok* (Γ, S) ⟨*proof*⟩

lemma *heap-upds-ok-append*:

assumes $domA \Delta \cap upds S = \{\}$

assumes *heap-upds-ok* (Γ, S)

shows *heap-upds-ok* $(\Delta @ \Gamma, S)$

<proof>

lemma *heap-upds-ok-let*:

assumes *atom* ' $domA \Delta \#* S$

assumes *heap-upds-ok* (Γ, S)

shows *heap-upds-ok* $(\Delta @ \Gamma, S)$

<proof>

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

$x \in domA \Gamma \implies heap-upds-ok (\Gamma, S) \implies heap-upds-ok (delete\ x\ \Gamma, Upd\ x\ \#S)$

<proof>

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

$map-of\ \Gamma\ x = Some\ e \implies heap-upds-ok (\Gamma, S) \implies heap-upds-ok (delete\ x\ \Gamma, Upd\ x\ \#S)$

<proof>

lemma *heap-upds-ok-delete*:

$heap-upds-ok (\Gamma, S) \implies heap-upds-ok (delete\ x\ \Gamma, S)$

<proof>

lemma *heap-upds-ok-restrictA*:

$heap-upds-ok (\Gamma, S) \implies heap-upds-ok (restrictA\ V\ \Gamma, S)$

<proof>

lemma *heap-upds-ok-restr-stack*:

$heap-upds-ok (\Gamma, S) \implies heap-upds-ok (\Gamma, restr-stack\ V\ S)$

<proof>

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

$heap-upds-ok (\Gamma, Upd\ x\ \#S) \implies heap-upds-ok ((x,e)\ \#\ \Gamma, S)$

<proof>

lemma *heap-upds-ok-reorder*:

$x \in domA \Gamma \implies heap-upds-ok (\Gamma, S) \implies heap-upds-ok ((x,e)\ \#\ delete\ x\ \Gamma, S)$

<proof>

lemma *heap-upds-ok-upd*:

$heap-upds-ok (\Gamma, Upd\ x\ \#S) \implies x \notin domA \Gamma \wedge x \notin upds S$

<proof>

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

$heap\text{-}upds\text{-}ok\text{-}restrictA$ $heap\text{-}upds\text{-}ok\text{-}restr\text{-}stack$
 $heap\text{-}upds\text{-}ok\text{-}let$
lemmas $heap\text{-}upds\text{-}ok.simps[simp\ del]$

end

2.2 Sestoft

theory $Sestoft$
imports $SestoftConf$
begin

inductive $step :: conf \Rightarrow conf \Rightarrow bool$ (**infix** $\Rightarrow 50$) **where**
 $app_1: (\Gamma, App\ e\ x, S) \Rightarrow (\Gamma, e, Arg\ x\ \# S)$
 $| app_2: (\Gamma, Lam\ [y].\ e, Arg\ x\ \# S) \Rightarrow (\Gamma, e[y ::= x], S)$
 $| var_1: map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow (\Gamma, Var\ x, S) \Rightarrow (delete\ x\ \Gamma, e, Upd\ x\ \# S)$
 $| var_2: x \notin domA\ \Gamma \Longrightarrow isVal\ e \Longrightarrow (\Gamma, e, Upd\ x\ \# S) \Rightarrow ((x,e)\# \Gamma, e, S)$
 $| let_1: atom\ 'domA\ \Delta\ \#\#*\ \Gamma \Longrightarrow atom\ 'domA\ \Delta\ \#\#*\ S$
 $\qquad \qquad \qquad \Longrightarrow (\Gamma, Let\ \Delta\ e, S) \Rightarrow (\Delta@ \Gamma, e, S)$
 $| if_1: (\Gamma, scrut\ ?\ e1 : e2, S) \Rightarrow (\Gamma, scrut, Alts\ e1\ e2\ \# S)$
 $| if_2: (\Gamma, Bool\ b, Alts\ e1\ e2\ \# S) \Rightarrow (\Gamma, if\ b\ then\ e1\ else\ e2, S)$

abbreviation $steps$ (**infix** $\Rightarrow^* 50$) **where** $steps \equiv step^{**}$

lemma $SmartLet\text{-}stepI$:

$atom\ 'domA\ \Delta\ \#\#*\ \Gamma \Longrightarrow atom\ 'domA\ \Delta\ \#\#*\ S \Longrightarrow (\Gamma, SmartLet\ \Delta\ e, S) \Rightarrow^* (\Delta@ \Gamma, e, S)$
 $\langle proof \rangle$

lemma $lambda\text{-}var$: $map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow isVal\ e \Longrightarrow (\Gamma, Var\ x, S) \Rightarrow^* ((x,e)\# delete\ x\ \Gamma, e, S)$
 $\langle proof \rangle$

lemma $let_1\text{-}closed$:

assumes $closed\ (\Gamma, Let\ \Delta\ e, S)$
assumes $domA\ \Delta \cap domA\ \Gamma = \{\}$
assumes $domA\ \Delta \cap upds\ S = \{\}$
shows $(\Gamma, Let\ \Delta\ e, S) \Rightarrow (\Delta@ \Gamma, e, S)$
 $\langle proof \rangle$

An induction rule that skips the annoying case of a lambda taken off the heap

lemma $step\text{-}invariant\text{-}induction[consumes\ 4, case\text{-}names\ app_1\ app_2\ think\ lamvar\ var_2\ let_1\ if_1\ if_2\ refl\ trans]$:

assumes $c \Rightarrow^* c'$
assumes $\neg boring\text{-}step\ c'$
assumes $invariant\ (\Rightarrow)\ I$
assumes $I\ c$

assumes app_1 : $\bigwedge \Gamma e x S . I (\Gamma, App e x, S) \implies P (\Gamma, App e x, S) (\Gamma, e, Arg x \# S)$
assumes app_2 : $\bigwedge \Gamma y e x S . I (\Gamma, Lam [y]. e, Arg x \# S) \implies P (\Gamma, Lam [y]. e, Arg x \# S) (\Gamma, e[y ::= x], S)$
assumes $thunk$: $\bigwedge \Gamma x e S . map-of \Gamma x = Some e \implies \neg isVal e \implies I (\Gamma, Var x, S) \implies P (\Gamma, Var x, S) (delete x \Gamma, e, Upd x \# S)$
assumes $lamvar$: $\bigwedge \Gamma x e S . map-of \Gamma x = Some e \implies isVal e \implies I (\Gamma, Var x, S) \implies P (\Gamma, Var x, S) ((x,e) \# delete x \Gamma, e, S)$
assumes var_2 : $\bigwedge \Gamma x e S . x \notin domA \Gamma \implies isVal e \implies I (\Gamma, e, Upd x \# S) \implies P (\Gamma, e, Upd x \# S) ((x,e) \# \Gamma, e, S)$
assumes let_1 : $\bigwedge \Delta \Gamma e S . atom ' domA \Delta \#* \Gamma \implies atom ' domA \Delta \#* S \implies I (\Gamma, Let \Delta e, S) \implies P (\Gamma, Let \Delta e, S) (\Delta @ \Gamma, e, S)$
assumes if_1 : $\bigwedge \Gamma scrut e1 e2 S . I (\Gamma, scrut ? e1 : e2, S) \implies P (\Gamma, scrut ? e1 : e2, S) (\Gamma, scrut, Alts e1 e2 \# S)$
assumes if_2 : $\bigwedge \Gamma b e1 e2 S . I (\Gamma, Bool b, Alts e1 e2 \# S) \implies P (\Gamma, Bool b, Alts e1 e2 \# S) (\Gamma, if b then e1 else e2, S)$
assumes $refl$: $\bigwedge c . P c c$
assumes $trans[trans]$: $\bigwedge c c' c'' . c \Rightarrow^* c' \implies c' \Rightarrow^* c'' \implies P c c' \implies P c' c'' \implies P c c''$
shows $P c c'$
(proof)

lemma *step-induction*[*consumes 2, case-names app₁ app₂ thunk lamvar var₂ let₁ if₁ if₂ refl trans*]:

assumes $c \Rightarrow^* c'$
assumes $\neg boring-step c'$
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 $thunk$: $\bigwedge \Gamma x e S . map-of \Gamma x = Some e \implies \neg isVal e \implies P (\Gamma, Var x, S) (delete x \Gamma, e, Upd x \# S)$
assumes $lamvar$: $\bigwedge \Gamma x e S . map-of \Gamma x = Some e \implies isVal e \implies P (\Gamma, Var x, S) ((x,e) \# delete x \Gamma, e, S)$
assumes var_2 : $\bigwedge \Gamma x e S . x \notin domA \Gamma \implies isVal e \implies P (\Gamma, e, Upd x \# S) ((x,e) \# \Gamma, e, S)$
assumes let_1 : $\bigwedge \Delta \Gamma e S . atom ' domA \Delta \#* \Gamma \implies atom ' domA \Delta \#* S \implies P (\Gamma, Let \Delta e, S) (\Delta @ \Gamma, e, S)$
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 $trans[trans]$: $\bigwedge c c' c'' . c \Rightarrow^* c' \implies c' \Rightarrow^* c'' \implies P c c' \implies P c' c'' \implies P c c''$
shows $P c c'$
(proof)

2.2.1 Equivariance

lemma *step-eqv*[*eqvt*]: $step x y \implies step (\pi \cdot x) (\pi \cdot y)$
(proof)

2.2.2 Invariants

lemma *closed-invariant*:

invariant step closed
<proof>

lemma *heap-upds-ok-invariant*:
invariant step heap-upds-ok-conf
<proof>

end

2.3 SestoftGC

theory *SestoftGC*
imports *Sestoft*
begin

inductive *gc-step* :: *conf* \Rightarrow *conf* \Rightarrow *bool* (**infix** \Rightarrow_G 50) **where**
 normal: $c \Rightarrow c' \Longrightarrow c \Rightarrow_G c'$
 | *dropUpd*: $(\Gamma, e, \text{Upd } x \# S) \Rightarrow_G (\Gamma, e, S @ [\text{Dummy } x])$

lemmas *gc-step-intros*[*intro*] =
 normal[*OF step.intros*(1)] *normal*[*OF step.intros*(2)] *normal*[*OF step.intros*(3)]
 normal[*OF step.intros*(4)] *normal*[*OF step.intros*(5)] *dropUpd*

abbreviation *gc-steps* (**infix** \Rightarrow_{G^*} 50) **where** *gc-steps* \equiv *gc-step***
lemmas *converse-rtranclp-into-rtranclp*[*of gc-step, OF - r-into-rtranclp, trans*]

lemma *var-onceI*:
 assumes *map-of* $\Gamma x = \text{Some } e$
 shows $(\Gamma, \text{Var } x, S) \Rightarrow_{G^*} (\text{delete } x \Gamma, e, S @ [\text{Dummy } x])$
<proof>

lemma *normal-trans*: $c \Rightarrow^* c' \Longrightarrow c \Rightarrow_{G^*} c'$
<proof>

fun *to-gc-conf* :: *var list* \Rightarrow *conf* \Rightarrow *conf*
 where *to-gc-conf* $r (\Gamma, e, S) = (\text{restrictA } (- \text{ set } r) \Gamma, e, \text{restr-stack } (- \text{ set } r) S @ (\text{map Dummy } (\text{rev } r)))$

lemma *restr-stack-map-Dummy*[*simp*]: *restr-stack* $V (\text{map Dummy } l) = \text{map Dummy } l$
<proof>

lemma *restr-stack-append*[*simp*]: *restr-stack* $V (l @ l') = \text{restr-stack } V l @ \text{restr-stack } V l'$
<proof>

lemma *to-gc-conf-append*[*simp*]:
to-gc-conf $(r @ r') c = \text{to-gc-conf } r (\text{to-gc-conf } r' c)$

<proof>

lemma *to-gc-conf-eqE[elim!]*:

assumes *to-gc-conf r c = (Γ, e, S)*

obtains $\Gamma' S'$ **where** $c = (\Gamma', e, S')$ **and** $\Gamma = \text{restrictA } (- \text{ set } r) \Gamma'$ **and** $S = \text{restr-stack } (- \text{ set } r) S' @ \text{map Dummy } (\text{rev } r)$

<proof>

fun *safe-hd* :: 'a list \Rightarrow 'a option

where *safe-hd* (x#-) = Some x
| *safe-hd* [] = None

lemma *safe-hd-None[simp]*: *safe-hd xs = None* \longleftrightarrow *xs = []*

<proof>

abbreviation *r-ok* :: var list \Rightarrow conf \Rightarrow bool

where *r-ok r c* \equiv *set r* \subseteq *domA* (*fst c*) \cup *upds* (*snd* (*snd c*))

lemma *subset-bound-invariant*:

invariant step (*r-ok r*)

<proof>

lemma *safe-hd-restr-stack[simp]*:

Some a = safe-hd (restr-stack V (a # S)) \longleftrightarrow *restr-stack V (a # S) = a # restr-stack V S*

<proof>

lemma *sestoftUnGCStack*:

assumes *heap-upds-ok* (Γ, S)

obtains $\Gamma' S'$ **where**

$(\Gamma, e, S) \Rightarrow^* (\Gamma', e, S')$

to-gc-conf r (Γ, e, S) = *to-gc-conf r* (Γ', e, S')

$\neg \text{isVal } e \vee \text{safe-hd } S' = \text{safe-hd } (\text{restr-stack } (- \text{ set } r) S')$

<proof>

lemma *perm-exI-trivial*:

$P x x \Longrightarrow \exists \pi. P (\pi \cdot x) x$

<proof>

lemma *upds-list-restr-stack[simp]*:

upds-list (restr-stack V S) = filter ($\lambda x. x \in V$) (*upds-list S*)

<proof>

lemma *heap-upd-ok-to-gc-conf*:

heap-upds-ok (Γ, S) \Longrightarrow *to-gc-conf r* (Γ, e, S) = (Γ'', e'', S'') \Longrightarrow *heap-upds-ok* (Γ'', S'')

<proof>

lemma *delete-restrictA-conv*:

delete x $\Gamma = \text{restrictA } (-\{x\}) \Gamma$

<proof>

lemma *sestoftUnGCstep*:

assumes *to-gc-conf* $r\ c \Rightarrow_G d$

assumes *heap-upds-ok-conf* c

assumes *closed* c

and *r-ok* $r\ c$

shows $\exists r'\ c'. c \Rightarrow^* c' \wedge d = \text{to-gc-conf } r'\ c' \wedge \text{r-ok } r'\ c'$

<proof>

lemma *sestoftUnGC*:

assumes $(\text{to-gc-conf } r\ c) \Rightarrow_G^* d$ **and** *heap-upds-ok-conf* c **and** *closed* c **and** *r-ok* $r\ c$

shows $\exists r'\ c'. c \Rightarrow^* c' \wedge d = \text{to-gc-conf } r'\ c' \wedge \text{r-ok } r'\ c'$

<proof>

lemma *dummies-unchanged-invariant*:

invariant step $(\lambda (\Gamma, e, S) . \text{dummies } S = V)$ (**is invariant** - ?I)

<proof>

lemma *sestoftUnGC'*:

assumes $([], e, []) \Rightarrow_G^* (\Gamma, e', \text{map Dummy } r)$

assumes *isVal* e'

assumes *fv* $e = (\{\}::\text{var set})$

shows $\exists \Gamma''. ([], e, []) \Rightarrow^* (\Gamma'', e', []) \wedge \Gamma = \text{restrictA } (- \text{set } r) \Gamma'' \wedge \text{set } r \subseteq \text{domA } \Gamma''$

<proof>

end

2.4 BalancedTraces

theory *BalancedTraces*

imports *Main*

begin

locale *traces* =

fixes *step* :: $'c \Rightarrow 'c \Rightarrow \text{bool}$ (**infix** \Rightarrow 50)

begin

abbreviation *steps* (**infix** \Rightarrow^* 50) **where** $\text{steps} \equiv \text{step}^*$

inductive *trace* :: $'c \Rightarrow 'c \text{ list} \Rightarrow 'c \Rightarrow \text{bool}$ **where**

trace-nil[*iff*]: $\text{trace } \text{final } [] \text{ final}$

| *trace-cons*[*intro*]: $\text{trace } \text{conf}'\ T \text{ final} \Longrightarrow \text{conf} \Rightarrow \text{conf}' \Longrightarrow \text{trace } \text{conf} (\text{conf}'\#T) \text{ final}$

inductive-cases *trace-consE*: $\text{trace } \text{conf} (\text{conf}'\#T) \text{ final}$

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

$trace\ x1\ x2\ final \implies P\ final \sqcap final \implies (\bigwedge conf' T\ conf. trace\ conf' T\ final \implies P\ conf' T\ final \implies conf \Rightarrow conf' \implies P\ conf\ (conf' \# T)\ final) \implies P\ x1\ x2\ final$
 ⟨proof⟩

lemma *build-trace*:

$c \Rightarrow^* c' \implies \exists T. trace\ c\ T\ c'$
 ⟨proof⟩

lemma *destruct-trace*: $trace\ c\ T\ c' \implies c \Rightarrow^* c'$

⟨proof⟩

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⟩

lemma *traces-list-all*:

$trace\ c\ T\ c' \implies P\ c' \implies (\bigwedge c\ c'. c \Rightarrow c' \implies P\ c' \implies P\ c) \implies (\forall x \in set\ T. P\ x) \wedge P\ c$
 ⟨proof⟩

lemma *trace-nil[simp]*: $trace\ c \sqcap c' \longleftrightarrow c = c'$

⟨proof⟩

end

definition *extends* :: 'a list \Rightarrow 'a list \Rightarrow bool (**infix** $\lesssim 50$) **where**

$S \lesssim S' = (\exists S''. S' = S'' @ S)$

lemma *extends-refl[simp]*: $S \lesssim S$ ⟨proof⟩

lemma *extends-cons[simp]*: $S \lesssim x \# S$ ⟨proof⟩

lemma *extends-append[simp]*: $S \lesssim L @ S$ ⟨proof⟩

lemma *extends-not-cons[simp]*: $\neg (x \# S) \lesssim S$ ⟨proof⟩

lemma *extends-trans[trans]*: $S \lesssim S' \implies S' \lesssim S'' \implies S \lesssim S''$ ⟨proof⟩

locale *balance-trace* = *traces* +

fixes $stack :: 'a \Rightarrow 's\ list$

assumes *one-step-only*: $c \Rightarrow c' \implies (stack\ c) = (stack\ c') \vee (\exists x. stack\ c' = x \# stack\ c) \vee (\exists x. stack\ c = x \# stack\ c')$

begin

inductive *bal* :: 'a \Rightarrow 'a list \Rightarrow 'a \Rightarrow bool **where**

balI[intro]: $trace\ c\ T\ c' \implies \forall c' \in set\ T. stack\ c \lesssim stack\ c' \implies stack\ c' = stack\ c \implies bal\ c\ T\ c'$

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

lemma *bal-nil[simp]*: $bal\ c\ []\ c' \longleftrightarrow c = c'$
 ⟨*proof*⟩

lemma *bal-stackD*: $bal\ c\ T\ c' \implies stack\ c' = stack\ c$ ⟨*proof*⟩

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*⟩

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** $c_3 \Rightarrow c_4\ bal\ c_4\ T_2\ c_5$
 ⟨*proof*⟩

end

end

2.5 SestoftCorrect

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

lemma *lemma-2*:
assumes $\Gamma : e \Downarrow_L \Delta : z$
and $fv\ (\Gamma, e, S) \subseteq set\ L \cup domA\ \Gamma$
shows $(\Gamma, e, S) \Rightarrow^* (\Delta, z, S)$
 ⟨*proof*⟩

type-synonym *trace* = *conf list*

fun *stack* :: *conf* \Rightarrow *stack* **where** *stack* $(\Gamma, e, S) = S$

interpretation *traces step* ⟨*proof*⟩

abbreviation *trace-syn* $(- \Rightarrow^* - [50,50,50] 50)$ **where** *trace-syn* $\equiv trace$

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

$(\Gamma, e, S) \Rightarrow^*_T \text{final} \Longrightarrow (\bigwedge \Gamma e S. \text{final} = (\Gamma, e, S) \Longrightarrow P \Gamma e S \square (\Gamma, e, S)) \Longrightarrow (\bigwedge \Gamma e S T$
 $\Gamma' e' S'. (\Gamma', e', S') \Rightarrow^*_T \text{final} \Longrightarrow P \Gamma' e' S' T \text{final} \Longrightarrow (\Gamma, e, S) \Rightarrow (\Gamma', e', S') \Longrightarrow P \Gamma e S$
 $((\Gamma', e', S') \# T) \text{final}) \Longrightarrow P \Gamma e S T \text{final}$
 <proof>

interpretation *balance-trace step stack*
 <proof>

abbreviation *bal-syn* $(- \Rightarrow^{b*} - [50,50,50] 50)$ **where** *bal-syn* $\equiv \text{bal}$

lemma *isVal-stops*:
assumes *isVal* e
assumes $(\Gamma, e, S) \Rightarrow^{b*}_T (\Delta, z, S)$
shows $T = \square$
 <proof>

lemma *Ball-subst[simp]*:
 $(\forall p \in \text{set } (\Gamma[y::h=x]). f p) \longleftrightarrow (\forall p \in \text{set } \Gamma. \text{case } p \text{ of } (z, e) \Rightarrow f(z, e[y::=x]))$
 <proof>

lemma *lemma-3*:
assumes $(\Gamma, e, S) \Rightarrow^{b*}_T (\Delta, z, S)$
assumes *isVal* z
shows $\Gamma : e \Downarrow_{\text{upds-list}} S \Delta : z$
 <proof>

lemma *dummy-stack-extended*:
 $\text{set } S \subseteq \text{Dummy } ' \text{UNIV} \Longrightarrow x \notin \text{Dummy } ' \text{UNIV} \Longrightarrow (S \lesssim x \# S') \longleftrightarrow S \lesssim S'$
 <proof>

lemma[*simp*]: *Arg* $x \notin \text{range Dummy Upd}$ $x \notin \text{range Dummy Alts}$ $e_1 e_2 \notin \text{range Dummy}$
 <proof>

lemma *dummy-stack-balanced*:
assumes $\text{set } S \subseteq \text{Dummy } ' \text{UNIV}$
assumes $(\Gamma, e, S) \Rightarrow^* (\Delta, z, S)$
obtains T **where** $(\Gamma, e, S) \Rightarrow^{b*}_T (\Delta, z, S)$
 <proof>

end

3 Arity

3.1 Arity

theory *Arity*
imports *Launchbury.HOLCF-Join-Classes*

begin

typedef *Arity* = *UNIV* :: *nat set*
morphisms *Rep-Arity to-Arity* \langle *proof* \rangle

setup-lifting *type-definition-Arity*

instantiation *Arity* :: *po*

begin

lift-definition *below-Arity* :: *Arity* \Rightarrow *Arity* \Rightarrow *bool* **is** $\lambda x y . y \leq x$ \langle *proof* \rangle

instance

\langle *proof* \rangle

end

instance *Arity* :: *chfin*

\langle *proof* \rangle

instance *Arity* :: *cpo* \langle *proof* \rangle

lift-definition *inc-Arity* :: *Arity* \Rightarrow *Arity* **is** *Suc* \langle *proof* \rangle

lift-definition *pred-Arity* :: *Arity* \Rightarrow *Arity* **is** $(\lambda x . x - 1)$ \langle *proof* \rangle

lemma *inc-Arity-cont*[*simp*]: *cont inc-Arity*
 \langle *proof* \rangle

lemma *pred-Arity-cont*[*simp*]: *cont pred-Arity*
 \langle *proof* \rangle

definition *inc* :: *Arity* \rightarrow *Arity* **where**

inc = $(\Lambda x . \text{inc-Arity } x)$

definition *pred* :: *Arity* \rightarrow *Arity* **where**

pred = $(\Lambda x . \text{pred-Arity } x)$

lemma *inc-inj*[*simp*]: *inc*·*n* = *inc*·*n'* \longleftrightarrow *n* = *n'*
 \langle *proof* \rangle

lemma *pred-inc*[*simp*]: *pred*·(*inc*·*n*) = *n*
 \langle *proof* \rangle

lemma *inc-below-inc*[*simp*]: *inc*·*a* \sqsubseteq *inc*·*b* \longleftrightarrow *a* \sqsubseteq *b*
 \langle *proof* \rangle

lemma *inc-below-below-pred*[*elim*]:

inc·*a* \sqsubseteq *b* \implies *a* \sqsubseteq *pred* · *b*

\langle *proof* \rangle

lemma *Rep-Arity-inc[simp]*: $\text{Rep-Arity } (\text{inc}\cdot a') = \text{Suc } (\text{Rep-Arity } a')$
 ⟨proof⟩

instantiation *Arity :: zero*

begin

lift-definition *zero-Arity :: Arity is 0*⟨proof⟩

instance⟨proof⟩

end

instantiation *Arity :: one*

begin

lift-definition *one-Arity :: Arity is 1*⟨proof⟩

instance ⟨proof⟩

end

lemma *one-is-inc-zero*: $1 = \text{inc}\cdot 0$

⟨proof⟩

lemma *inc-not-0[simp]*: $\text{inc}\cdot n = 0 \longleftrightarrow \text{False}$

⟨proof⟩

lemma *pred-0[simp]*: $\text{pred}\cdot 0 = 0$

⟨proof⟩

lemma *Arity-ind*: $P\ 0 \implies (\bigwedge n. P\ n \implies P\ (\text{inc}\cdot n)) \implies P\ n$

⟨proof⟩

lemma *Arity-total*:

fixes $x\ y :: \text{Arity}$

shows $x \sqsubseteq y \vee y \sqsubseteq x$

⟨proof⟩

instance *Arity :: Finite-Join-cpo*

⟨proof⟩

lemma *Arity-zero-top[simp]*: $(x :: \text{Arity}) \sqsubseteq 0$

⟨proof⟩

lemma *Arity-above-top[simp]*: $0 \sqsubseteq (a :: \text{Arity}) \longleftrightarrow a = 0$

⟨proof⟩

lemma *Arity-zero-join[simp]*: $(x :: \text{Arity}) \sqcup 0 = 0$

⟨proof⟩

lemma *Arity-zero-join2[simp]*: $0 \sqcup (x :: \text{Arity}) = 0$

⟨proof⟩

lemma *Arity-up-zero-join[simp]*: $(x :: \text{Arity}_\perp) \sqcup \text{up}\cdot 0 = \text{up}\cdot 0$

⟨proof⟩

lemma *Arity-up-zero-join2[simp]*: $up \cdot 0 \sqcup (x :: Arity_{\perp}) = up \cdot 0$
 $\langle proof \rangle$

lemma *up-zero-top[simp]*: $x \sqsubseteq up \cdot (0 :: Arity)$
 $\langle proof \rangle$

lemma *Arity-above-up-top[simp]*: $up \cdot 0 \sqsubseteq (a :: Arity_{\perp}) \longleftrightarrow a = up \cdot 0$
 $\langle proof \rangle$

lemma *Arity-exhaust*: $(y = 0 \implies P) \implies (\bigwedge x. y = inc \cdot x \implies P) \implies P$
 $\langle proof \rangle$

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]*: $\pi \cdot (x \sqcup (y :: 'a :: \{Finite-Join-cpo, cont-pt\})) = (\pi \cdot x) \sqcup (\pi \cdot y)$
 $\langle proof \rangle$

instantiation *Arity* :: *pure*

begin
definition $p \cdot (a :: Arity) = a$

instance

$\langle proof \rangle$

end

instance *Arity* :: *cont-pt* $\langle proof \rangle$

instance *Arity* :: *pure-cont-pt* $\langle proof \rangle$

end

3.4 ArityStack

```
theory ArityStack
imports Arity SestoftConf
begin

fun Astack :: stack  $\Rightarrow$  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)  $\sqsubseteq$  Astack S
  <proof>

lemma Astack-map-Dummy[simp]:
  Astack (map Dummy l) = 0
  <proof>

lemma Astack-append-map-Dummy[simp]:
  Astack S' = 0  $\implies$  Astack (S @ S') = Astack S
  <proof>

end
```

4 Eta-Expansion

4.1 EtaExpansion

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

definition fresh-var :: exp  $\Rightarrow$  var where
  fresh-var e = (SOME v. v  $\notin$  fv e)

lemma fresh-var-not-free:
  fresh-var e  $\notin$  fv e
  <proof>

lemma fresh-var-fresh[simp]:
  atom (fresh-var e) # e
  <proof>

lemma fresh-var-subst[simp]:
```

$e[\text{fresh-var } e ::= x] = e$
 ⟨proof⟩

fun *eta-expand* :: *nat* ⇒ *exp* ⇒ *exp* **where**
 eta-expand 0 *e* = *e*
 | *eta-expand* (*Suc* *n*) *e* = (*Lam* [*fresh-var* *e*]. *eta-expand* *n* (*App* *e* (*fresh-var* *e*)))

lemma *eta-expand-eqvt*[*eqvt*]:
 $\pi \cdot (\text{eta-expand } n \ e) = \text{eta-expand } (\pi \cdot n) \ (\pi \cdot e)$
 ⟨proof⟩

lemma *fresh-eta-expand*[*simp*]: $a \# \text{eta-expand } n \ e \longleftrightarrow a \# e$
 ⟨proof⟩

lemma *subst-eta-expand*: $(\text{eta-expand } n \ e)[x ::= y] = \text{eta-expand } n \ (e[x ::= y])$
 ⟨proof⟩

lemma *isLam-eta-expand*:
 $\text{isLam } e \implies \text{isLam } (\text{eta-expand } n \ e)$ **and** $n > 0 \implies \text{isLam } (\text{eta-expand } n \ e)$
 ⟨proof⟩

lemma *isVal-eta-expand*:
 $\text{isVal } e \implies \text{isVal } (\text{eta-expand } n \ e)$ **and** $n > 0 \implies \text{isVal } (\text{eta-expand } n \ e)$
 ⟨proof⟩

end

4.2 EtaExpansionSafe

theory *EtaExpansionSafe*
imports *EtaExpansion Sestoft*
begin

theorem *eta-expansion-safe*:
assumes $set \ T \subseteq range \ Arg$
shows $(\Gamma, \text{eta-expand } (length \ T) \ e, T@S) \Rightarrow^* (\Gamma, e, T@S)$
 ⟨proof⟩

fun *arg-prefix* :: *stack* ⇒ *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 $n \leq \text{arg-prefix } S$
shows $(\Gamma, \text{eta-expand } n \ e, S) \Rightarrow^* (\Gamma, e, S)$

<proof>

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 \Rightarrow exp \Rightarrow exp) \Rightarrow ('a_⊥ \Rightarrow exp \Rightarrow exp)
 where *lift-transform* t *Ibottom* e = e
 | *lift-transform* t (*Iup* a) e = t a e

lemma *lift-transform-simps[simp]*:

lift-transform t \perp e = e
lift-transform t (*up*.a) e = t a e
<proof>

lemma *lift-transform-eqt[eqt]*: $\pi \cdot \text{lift-transform } t \ a \ e = \text{lift-transform } (\pi \cdot t) \ (\pi \cdot a) \ (\pi \cdot e)$
<proof>

lemma *lift-transform-fun-cong[fundef-cong]*:

$(\bigwedge a. t1 \ a \ e1 = t2 \ a \ e1) \Longrightarrow a1 = a2 \Longrightarrow e1 = e2 \Longrightarrow \text{lift-transform } t1 \ a1 \ e1 = \text{lift-transform } t2 \ a2 \ e2$
<proof>

lemma *subst-lift-transform*:

assumes $\bigwedge a. (t \ a \ e)[x ::= y] = t \ a \ (e[x ::= y])$
shows $(\text{lift-transform } t \ a \ e)[x ::= y] = \text{lift-transform } t \ a \ (e[x ::= y])$
<proof>

definition

map-transform :: ('a::cont-pt \Rightarrow exp \Rightarrow exp) \Rightarrow (var \Rightarrow 'a_⊥) \Rightarrow heap \Rightarrow heap
where *map-transform* t ae = *map-ran* ($\lambda x \ e. \text{lift-transform } t \ (ae \ x) \ e$)

lemma *map-transform-eqt[eqt]*: $\pi \cdot \text{map-transform } t \ ae = \text{map-transform } (\pi \cdot t) \ (\pi \cdot ae)$
<proof>

lemma *domA-map-transform[simp]*: *domA* (*map-transform* t ae Γ) = *domA* Γ
<proof>

lemma *length-map-transform[simp]*: *length* (*map-transform* t ae *xs*) = *length* *xs*
<proof>

lemma *map-transform-delete*:

$map\text{-}transform\ t\ ae\ (delete\ x\ \Gamma) = delete\ x\ (map\text{-}transform\ t\ ae\ \Gamma)$
 $\langle proof \rangle$

lemma *map-transform-restrA*:

$map\text{-}transform\ t\ ae\ (restrictA\ S\ \Gamma) = restrictA\ S\ (map\text{-}transform\ t\ ae\ \Gamma)$
 $\langle proof \rangle$

lemma *delete-map-transform-env-delete*:

$delete\ x\ (map\text{-}transform\ t\ (env\text{-}delete\ x\ ae)\ \Gamma) = delete\ x\ (map\text{-}transform\ t\ ae\ \Gamma)$
 $\langle proof \rangle$

lemma *map-transform-Nil[simp]*:

$map\text{-}transform\ t\ ae\ [] = []$
 $\langle proof \rangle$

lemma *map-transform-Cons*:

$map\text{-}transform\ t\ ae\ ((x,e)\#\ \Gamma) = (x,\ lift\text{-}transform\ t\ (ae\ x)\ e)\ \#\ (map\text{-}transform\ t\ ae\ \Gamma)$
 $\langle proof \rangle$

lemma *map-transform-append*:

$map\text{-}transform\ t\ ae\ (\Delta @ \Gamma) = map\text{-}transform\ t\ ae\ \Delta @ map\text{-}transform\ t\ ae\ \Gamma$
 $\langle proof \rangle$

lemma *map-transform-fundef-cong[fundef-cong]*:

$(\bigwedge x\ e\ a.\ (x,e) \in set\ m1 \implies t1\ a\ e = t2\ a\ e) \implies ae1 = ae2 \implies m1 = m2 \implies map\text{-}transform\ t1\ ae1\ m1 = map\text{-}transform\ t2\ ae2\ m2$
 $\langle proof \rangle$

lemma *map-transform-cong*:

$(\bigwedge x.\ x \in domA\ m1 \implies ae\ x = ae'\ x) \implies m1 = m2 \implies map\text{-}transform\ t\ ae\ m1 = map\text{-}transform\ t\ ae'\ m2$
 $\langle proof \rangle$

lemma *map-of-map-transform*: $map\text{-}of\ (map\text{-}transform\ t\ ae\ \Gamma)\ x = map\text{-}option\ (lift\text{-}transform\ t\ (ae\ x))\ (map\text{-}of\ \Gamma\ x)$

$\langle proof \rangle$

lemma *supp-map-transform-step*:

assumes $\bigwedge x\ e\ a.\ (x, e) \in set\ \Gamma \implies supp\ (t\ a\ e) \subseteq supp\ e$

shows $supp\ (map\text{-}transform\ t\ ae\ \Gamma) \subseteq supp\ \Gamma$

$\langle proof \rangle$

lemma *subst-map-transform*:

assumes $\bigwedge x'\ e\ a.\ (x',e) : set\ \Gamma \implies (t\ a\ e)[x ::= y] = t\ a\ (e[x ::= y])$

shows $(map\text{-}transform\ t\ ae\ \Gamma)[x ::= h=y] = map\text{-}transform\ t\ ae\ (\Gamma[x ::= h=y])$

$\langle proof \rangle$

locale *supp-bounded-transform* =

fixes $trans :: 'a :: cont\text{-}pt \Rightarrow exp \Rightarrow exp$

assumes *supp-trans*: $\text{supp } (\text{trans } a \ e) \subseteq \text{supp } e$
begin
lemma *supp-lift-transform*: $\text{supp } (\text{lift-transform } \text{trans } a \ e) \subseteq \text{supp } e$
 ⟨*proof*⟩
lemma *supp-map-transform*: $\text{supp } (\text{map-transform } \text{trans } a \ e \ \Gamma) \subseteq \text{supp } \Gamma$
 ⟨*proof*⟩
lemma *fresh-transform[intro]*: $a \ \# \ e \implies a \ \# \ \text{trans } n \ e$
 ⟨*proof*⟩
lemma *fresh-star-transform[intro]*: $a \ \#\ast \ e \implies a \ \#\ast \ \text{trans } n \ e$
 ⟨*proof*⟩
lemma *fresh-map-transform[intro]*: $a \ \# \ \Gamma \implies a \ \# \ \text{map-transform } \text{trans } a \ e \ \Gamma$
 ⟨*proof*⟩
lemma *fresh-star-map-transform[intro]*: $a \ \#\ast \ \Gamma \implies a \ \#\ast \ \text{map-transform } \text{trans } a \ e \ \Gamma$
 ⟨*proof*⟩
end

end

4.4 ArityEtaExpansion

theory *ArityEtaExpansion*
imports *EtaExpansion Arity-Nominal TransformTools*
begin
lift-definition *Aeta-expand* :: $\text{Arity} \Rightarrow \text{exp} \Rightarrow \text{exp}$ **is** *eta-expand*⟨*proof*⟩
lemma *Aeta-expand-eqt[eqt]*: $\pi \cdot \text{Aeta-expand } a \ e = \text{Aeta-expand } (\pi \cdot a) (\pi \cdot e)$
 ⟨*proof*⟩
lemma *Aeta-expand-0[simp]*: $\text{Aeta-expand } 0 \ e = e$
 ⟨*proof*⟩
lemma *Aeta-expand-inc[simp]*: $\text{Aeta-expand } (\text{inc} \cdot n) \ e = (\text{Lam } [\text{fresh-var } e]. \text{Aeta-expand } n \ (\text{App } e \ (\text{fresh-var } e)))$
 ⟨*proof*⟩
lemma *subst-Aeta-expand*:
 $(\text{Aeta-expand } n \ e)[x::=y] = \text{Aeta-expand } n \ e[x::=y]$
 ⟨*proof*⟩
lemma *isLam-Aeta-expand*: $\text{isLam } e \implies \text{isLam } (\text{Aeta-expand } a \ e)$
 ⟨*proof*⟩

lemma *isVal-Aeta-expand*: $isVal\ e \implies isVal\ (Aeta\text{-}expand\ a\ e)$
 ⟨*proof*⟩

lemma *Aeta-expand-fresh[simp]*: $a \# Aeta\text{-}expand\ n\ e = a \# e$ ⟨*proof*⟩

lemma *Aeta-expand-fresh-star[simp]*: $a \#* Aeta\text{-}expand\ n\ e = a \#* e$ ⟨*proof*⟩

interpretation *supp-bounded-transform Aeta-expand*
 ⟨*proof*⟩

end

4.5 ArityEtaExpansionSafe

theory *ArityEtaExpansionSafe*

imports *EtaExpansionSafe ArityStack ArityEtaExpansion*

begin

lemma *Aeta-expand-safe*:

assumes $Astack\ S \sqsubseteq a$

shows $(\Gamma, Aeta\text{-}expand\ a\ e, S) \Rightarrow^* (\Gamma, e, S)$

⟨*proof*⟩

end

5 Arity Analysis

5.1 ArityAnalysisSig

theory *ArityAnalysisSig*

imports *Launchbury.Terms AEnv Arity-Nominal Launchbury.Nominal-HOLCF Launchbury.Substitution*

begin

locale *ArityAnalysis* =

fixes $Aexp :: exp \Rightarrow Arity \rightarrow AEnv$

begin

abbreviation $Aexp\text{-}syn\ (\mathcal{A}.)$ **where** $\mathcal{A}_a\ e \equiv Aexp\ e \cdot a$

abbreviation $Aexp\text{-}bot\text{-}syn\ (\mathcal{A}^\perp.)$

where $\mathcal{A}^\perp_a\ e \equiv fup \cdot (Aexp\ e) \cdot a$

end

locale *ArityAnalysisHeap* =

fixes $Aheap :: heap \Rightarrow exp \Rightarrow Arity \rightarrow AEnv$

locale *EdomArityAnalysis* = *ArityAnalysis* +

assumes $Aexp\text{-}edom: edom\ (\mathcal{A}_a\ e) \subseteq fv\ e$

begin

lemma *fup-Aexp-edom*: $\text{edom } (\mathcal{A}^\perp_a e) \subseteq \text{fv } e$
<proof>

lemma *Aexp-fresh-bot[simp]*: **assumes** $\text{atom } v \# e$ **shows** $\mathcal{A}_a e v = \perp$
<proof>

end

locale *ArityAnalysisHeapEqvt* = *ArityAnalysisHeap* +
assumes *Aheap-eqvt*[*eqvt*]: $\pi \cdot \text{Aheap} = \text{Aheap}$

end

5.2 ArityAnalysisAbinds

theory *ArityAnalysisAbinds*

imports *ArityAnalysisSig*

begin

context *ArityAnalysis*

begin

5.2.1 Lifting arity analysis to recursive groups

definition *ABind* :: $\text{var} \Rightarrow \text{exp} \Rightarrow (\text{AEnv} \rightarrow \text{AEnv})$
where $\text{ABind } v e = (\Lambda \text{ ae. fup} \cdot (\text{Aexp } e) \cdot (\text{ae } v))$

lemma *ABind-eq[simp]*: $\text{ABind } v e \cdot \text{ae} = \mathcal{A}^\perp_{\text{ae } v} e$
<proof>

fun *ABinds* :: $\text{heap} \Rightarrow (\text{AEnv} \rightarrow \text{AEnv})$
where $\text{ABinds } [] = \perp$
| $\text{ABinds } ((v,e)\#bnds) = \text{ABind } v e \sqcup \text{ABinds } (\text{delete } v \text{ bnds})$

lemma *ABinds-strict[simp]*: $\text{ABinds } \Gamma \cdot \perp = \perp$
<proof>

lemma *Abinds-reorder1*: $\text{map-of } \Gamma v = \text{Some } e \implies \text{ABinds } \Gamma = \text{ABind } v e \sqcup \text{ABinds } (\text{delete } v \Gamma)$
<proof>

lemma *ABind-below-ABinds*: $\text{map-of } \Gamma v = \text{Some } e \implies \text{ABind } v e \sqsubseteq \text{ABinds } \Gamma$
<proof>

lemma *Abinds-reorder*: $\text{map-of } \Gamma = \text{map-of } \Delta \implies \text{ABinds } \Gamma = \text{ABinds } \Delta$
<proof>

lemma *Abinds-env-cong*: $(\bigwedge x. x \in \text{dom} A \ \Delta \implies ae \ x = ae' \ x) \implies ABinds \ \Delta \cdot ae = ABinds \ \Delta \cdot ae'$

<proof>

lemma *Abinds-env-restr-cong*: $ae \ f|' \ \text{dom} A \ \Delta = ae' \ f|' \ \text{dom} A \ \Delta \implies ABinds \ \Delta \cdot ae = ABinds \ \Delta \cdot ae'$

<proof>

lemma *ABinds-env-restr[simp]*: $ABinds \ \Delta \cdot (ae \ f|' \ \text{dom} A \ \Delta) = ABinds \ \Delta \cdot ae$

<proof>

lemma *Abinds-join-fresh*: $ae' \ ' \ (\text{dom} A \ \Delta) \subseteq \{\perp\} \implies ABinds \ \Delta \cdot (ae \sqcup ae') = (ABinds \ \Delta \cdot ae)$

<proof>

lemma *ABinds-delete-bot*: $ae \ x = \perp \implies ABinds \ (\text{delete } x \ \Gamma) \cdot ae = ABinds \ \Gamma \cdot ae$

<proof>

lemma *ABinds-restr-fresh*:

assumes *atom* ' $S \ \#\ast \ \Gamma$

shows $ABinds \ \Gamma \cdot ae \ f|' \ (- \ S) = ABinds \ \Gamma \cdot (ae \ f|' \ (- \ S)) \ f|' \ (- \ S)$

<proof>

lemma *ABinds-restr*:

assumes $\text{dom} A \ \Gamma \subseteq S$

shows $ABinds \ \Gamma \cdot ae \ f|' \ S = ABinds \ \Gamma \cdot (ae \ f|' \ S) \ f|' \ S$

<proof>

lemma *ABinds-restr-subst*:

assumes $\bigwedge x' \ e \ a. (x', e) \in \text{set } \Gamma \implies Aexp \ e[x::=y] \cdot a \ f|' \ S = Aexp \ e \cdot a \ f|' \ S$

assumes $x \notin S$

assumes $y \notin S$

assumes $\text{dom} A \ \Gamma \subseteq S$

shows $ABinds \ \Gamma[x::h=y] \cdot ae \ f|' \ S = ABinds \ \Gamma \cdot (ae \ f|' \ S) \ f|' \ S$

<proof>

lemma *Abinds-append-disjoint*: $\text{dom} A \ \Delta \cap \text{dom} A \ \Gamma = \{\} \implies ABinds \ (\Delta \ @ \ \Gamma) \cdot ae = ABinds \ \Delta \cdot ae \sqcup ABinds \ \Gamma \cdot ae$

<proof>

lemma *ABinds-restr-subset*: $S \subseteq S' \implies ABinds \ (\text{restrict} A \ S \ \Gamma) \cdot ae \sqsubseteq ABinds \ (\text{restrict} A \ S' \ \Gamma) \cdot ae$

<proof>

lemma *ABinds-restrict-edom*: $ABinds \ (\text{restrict} A \ (\text{edom } ae) \ \Gamma) \cdot ae = ABinds \ \Gamma \cdot ae$

<proof>

lemma *ABinds-restrict-below*: $ABinds \ (\text{restrict} A \ S \ \Gamma) \cdot ae \sqsubseteq ABinds \ \Gamma \cdot ae$

<proof>

lemma *ABinds-delete-below*: $ABinds (delete\ x\ \Gamma).ae \sqsubseteq ABinds\ \Gamma.ae$

<proof>

end

lemma *ABind-eqvt[eqvt]*: $\pi \cdot (ArityAnalysis.ABind\ Aexp\ v\ e) = ArityAnalysis.ABind\ (\pi \cdot Aexp)$

$(\pi \cdot v)\ (\pi \cdot e)$

<proof>

lemma *ABinds-eqvt[eqvt]*: $\pi \cdot (ArityAnalysis.ABinds\ Aexp\ \Gamma) = ArityAnalysis.ABinds\ (\pi \cdot Aexp)\ (\pi \cdot \Gamma)$

<proof>

lemma *Abinds-cong[fundef-cong]*:

$\llbracket (\bigwedge e. e \in snd\ 'set\ heap2 \implies aexp1\ e = aexp2\ e) ; heap1 = heap2 \rrbracket$

$\implies ArityAnalysis.ABinds\ aexp1\ heap1 = ArityAnalysis.ABinds\ aexp2\ heap2$

<proof>

context *EdomArityAnalysis*

begin

lemma *fup-Aexp-lookup-fresh*: $atom\ v\ \# e \implies (fup.(Aexp\ e).a)\ v = \perp$

<proof>

lemma *edom-AnalBinds*: $edom\ (ABinds\ \Gamma.ae) \subseteq fv\ \Gamma$

<proof>

end

end

5.3 ArityAnalysisSpec

theory *ArityAnalysisSpec*

imports *ArityAnalysisAbinds*

begin

locale *SubstArityAnalysis* = *EdomArityAnalysis* +

assumes *Aexp-subst-restr*: $x \notin S \implies y \notin S \implies (Aexp\ e[x::=y] \cdot a)\ f|'S = (Aexp\ e \cdot a)\ f|'S$

locale *ArityAnalysisSafe* = *SubstArityAnalysis* +

assumes *Aexp-Var*: $up \cdot n \sqsubseteq (Aexp\ (Var\ x).n)\ x$

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*: $env\ delete\ y\ (Aexp\ e \cdot (pred \cdot n)) \sqsubseteq Aexp\ (Lam\ [y].\ e) \cdot n$

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*: $x \notin \text{dom}A \Gamma \implies y \notin \text{dom}A \Gamma \implies \text{Aheap } \Gamma[x::h=y] e[x ::= y] = \text{Aheap } \Gamma e$

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

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

context *SubstArityAnalysis*

begin

lemma *Aexp-subst-upd*: $(\text{Aexp } e[y::=x] \cdot n) \sqsubseteq (\text{Aexp } e \cdot n)(y := \perp, x := \text{up} \cdot 0)$
 $\langle \text{proof} \rangle$

lemma *Aexp-subst*: $\text{Aexp } (e[y::=x]) \cdot a \sqsubseteq \text{env-delete } y ((\text{Aexp } e) \cdot a) \sqsubseteq \text{esing } x \cdot (\text{up} \cdot 0)$
 $\langle \text{proof} \rangle$

end

context *ArityAnalysisSafe*

begin

lemma *Aexp-Var-singleton*: $\text{esing } x \cdot (\text{up} \cdot n) \sqsubseteq \text{Aexp } (\text{Var } x) \cdot n$
 $\langle \text{proof} \rangle$

lemma *fup-Aexp-Var*: $\text{esing } x \cdot n \sqsubseteq \text{fup} \cdot (\text{Aexp } (\text{Var } x)) \cdot n$
 $\langle \text{proof} \rangle$

end

context *ArityAnalysisLetSafe*

begin

lemma *Aheap-nonrec*:
assumes *nonrec* Δ
shows $\text{Aexp } e \cdot a \upharpoonright^c \text{dom}A \Delta \sqsubseteq \text{Aheap } \Delta e \cdot a$
 $\langle \text{proof} \rangle$

end

end

5.4 TrivialArityAnal

theory *TrivialArityAnal*

imports *ArityAnalysisSpec Launchbury.Env-Nominal*

begin

definition *Trivial-Aexp* :: $\text{exp} \Rightarrow \text{Arity} \rightarrow \text{AEnv}$
where $\text{Trivial-Aexp } e = (\Lambda n. (\lambda x. \text{up} \cdot 0) \upharpoonright^c \text{fv } e)$

lemma *Trivial-Aexp-simp*: $\text{Trivial-Aexp } e \cdot n = (\lambda x. \text{up}\cdot 0) f |' \text{fv } e$
<proof>

lemma *edom-Trivial-Aexp[simp]*: $\text{edom } (\text{Trivial-Aexp } e \cdot n) = \text{fv } e$
<proof>

lemma *Trivial-Aexp-eq[iff]*: $\text{Trivial-Aexp } e \cdot n = \text{Trivial-Aexp } e' \cdot n' \iff \text{fv } e = (\text{fv } e' :: \text{var set})$
<proof>

lemma *below-Trivial-Aexp[simp]*: $(ae \sqsubseteq \text{Trivial-Aexp } e \cdot n) \iff \text{edom } ae \subseteq \text{fv } e$
<proof>

interpretation *ArityAnalysis Trivial-Aexp**<proof>*

interpretation *EdomArityAnalysis Trivial-Aexp*
<proof>

interpretation *ArityAnalysisSafe Trivial-Aexp*
<proof>

definition *Trivial-Aheap* :: $\text{heap} \Rightarrow \text{exp} \Rightarrow \text{Arity} \rightarrow \text{AEnv}$ **where**
 $\text{Trivial-Aheap } \Gamma e = (\Lambda a. (\lambda x. \text{up}\cdot 0) f |' \text{domA } \Gamma)$

lemma *Trivial-Aheap-eqvt[eqvt]*: $\pi \cdot (\text{Trivial-Aheap } \Gamma e) = \text{Trivial-Aheap } (\pi \cdot \Gamma) (\pi \cdot e)$
<proof>

lemma *Trivial-Aheap-simp*: $\text{Trivial-Aheap } \Gamma e \cdot a = (\lambda x. \text{up}\cdot 0) f |' \text{domA } \Gamma$
<proof>

lemma *Trivial-fup-Aexp-below-fv*: $\text{fup} \cdot (\text{Trivial-Aexp } e) \cdot a \sqsubseteq (\lambda x. \text{up}\cdot 0) f |' \text{fv } e$
<proof>

lemma *Trivial-ABinds-below-fv*: $\text{ABinds } \Gamma \cdot ae \sqsubseteq (\lambda x. \text{up}\cdot 0) f |' \text{fv } \Gamma$
<proof>

interpretation *ArityAnalysisLetSafe Trivial-Aexp Trivial-Aheap*
<proof>

end

5.5 ArityAnalysisStack

theory *ArityAnalysisStack*

imports *SestoftConf ArityAnalysisSig*

begin

context *ArityAnalysis*

```

begin
  fun AEstack :: Arity list  $\Rightarrow$  stack  $\Rightarrow$  AEnv
    where
      AEstack - [] =  $\perp$ 
      | AEstack (a#as) (Alts e1 e2 # S) = Aexp e1·a  $\sqcup$  Aexp e2·a  $\sqcup$  AEstack as S
      | AEstack as (Upd x # S) = esing x·(up·0)  $\sqcup$  AEstack as S
      | AEstack as (Arg x # S) = esing x·(up·0)  $\sqcup$  AEstack as S
      | AEstack as (- # S) = AEstack as S
end

```

context *EdomArityAnalysis*

```

begin
  lemma edom-AEstack: edom (AEstack as S)  $\subseteq$  fv S
     $\langle$ proof $\rangle$ 
end

```

end

5.6 ArityAnalysisFix

```

theory ArityAnalysisFix
imports ArityAnalysisSig ArityAnalysisAbinds
begin

```

```

context ArityAnalysis
begin

```

```

definition Afix :: heap  $\Rightarrow$  (AEnv  $\rightarrow$  AEnv)
  where Afix  $\Gamma$  = ( $\Lambda$  ae. ( $\mu$  ae'. ABinds  $\Gamma$  · ae'  $\sqcup$  ae))

```

```

lemma Afix-eq: Afix  $\Gamma$ ·ae = ( $\mu$  ae'. (ABinds  $\Gamma$ ·ae')  $\sqcup$  ae)
   $\langle$ proof $\rangle$ 

```

```

lemma Afix-strict[simp]: Afix  $\Gamma$ · $\perp$  =  $\perp$ 
   $\langle$ proof $\rangle$ 

```

```

lemma Afix-least-below: ABinds  $\Gamma$  · ae'  $\sqsubseteq$  ae'  $\implies$  ae  $\sqsubseteq$  ae'  $\implies$  Afix  $\Gamma$  · ae  $\sqsubseteq$  ae'
   $\langle$ proof $\rangle$ 

```

```

lemma Afix-unroll: Afix  $\Gamma$ ·ae = ABinds  $\Gamma$  · (Afix  $\Gamma$ ·ae)  $\sqcup$  ae
   $\langle$ proof $\rangle$ 

```

```

lemma Abinds-below-Afix: ABinds  $\Delta$   $\sqsubseteq$  Afix  $\Delta$ 
   $\langle$ proof $\rangle$ 

```

```

lemma Afix-above-arg: ae  $\sqsubseteq$  Afix  $\Gamma$  · ae
   $\langle$ proof $\rangle$ 

```

```

lemma Abinds-Afix-below[simp]: ABinds  $\Gamma$ ·(Afix  $\Gamma$ ·ae)  $\sqsubseteq$  Afix  $\Gamma$ ·ae

```

<proof>

lemma *Afix-reorder*: $\text{map-of } \Gamma = \text{map-of } \Delta \implies \text{Afix } \Gamma = \text{Afix } \Delta$

<proof>

lemma *Afix-repeat-singleton*: $(\mu \text{ xa. Afix } \Gamma \cdot (\text{esing } x \cdot (n \sqcup \text{xa } x) \sqcup \text{ae})) = \text{Afix } \Gamma \cdot (\text{esing } x \cdot n \sqcup \text{ae})$

<proof>

lemma *Afix-join-fresh*: $\text{ae}' \text{ ' } (\text{domA } \Delta) \subseteq \{\perp\} \implies \text{Afix } \Delta \cdot (\text{ae} \sqcup \text{ae}') = (\text{Afix } \Delta \cdot \text{ae}) \sqcup \text{ae}'$

<proof>

lemma *Afix-restr-fresh*:

assumes $\text{atom}' S \ \#^* \Gamma$

shows $\text{Afix } \Gamma \cdot \text{ae } f|' (- S) = \text{Afix } \Gamma \cdot (\text{ae } f|' (- S)) f|' (- S)$

<proof>

lemma *Afix-restr*:

assumes $\text{domA } \Gamma \subseteq S$

shows $\text{Afix } \Gamma \cdot \text{ae } f|' S = \text{Afix } \Gamma \cdot (\text{ae } f|' S) f|' S$

<proof>

lemma *Afix-restr-subst'*:

assumes $\bigwedge x' e a. (x', e) \in \text{set } \Gamma \implies \text{Aexp } e[x::=y] \cdot a f|' S = \text{Aexp } e \cdot a f|' S$

assumes $x \notin S$

assumes $y \notin S$

assumes $\text{domA } \Gamma \subseteq S$

shows $\text{Afix } \Gamma[x::h=y] \cdot \text{ae } f|' S = \text{Afix } \Gamma \cdot (\text{ae } f|' S) f|' S$

<proof>

lemma *Afix-subst-approx*:

assumes $\bigwedge v n. v \in \text{domA } \Gamma \implies \text{Aexp } (\text{the } (\text{map-of } \Gamma v)) [y::=x] \cdot n \sqsubseteq (\text{Aexp } (\text{the } (\text{map-of } \Gamma v)) \cdot n) (y := \perp, x := \text{up } 0)$

assumes $x \notin \text{domA } \Gamma$

assumes $y \notin \text{domA } \Gamma$

shows $\text{Afix } \Gamma[y::h=x] \cdot (\text{ae}(y := \perp, x := \text{up } 0)) \sqsubseteq (\text{Afix } \Gamma \cdot \text{ae})(y := \perp, x := \text{up } 0)$

<proof>

end

lemma *Afix-eqvt[eqvt]*: $\pi \cdot (\text{ArietyAnalysis.Afix Aexp } \Gamma) = \text{ArietyAnalysis.Afix } (\pi \cdot \text{Aexp}) (\pi \cdot \Gamma)$

<proof>

lemma *Afix-cong[fundef-cong]*:

$\llbracket (\bigwedge e. e \in \text{snd } \text{'set heap2} \implies \text{aexp1 } e = \text{aexp2 } e); \text{heap1} = \text{heap2} \rrbracket$
 $\implies \text{ArityAnalysis.Afix aexp1 heap1} = \text{ArityAnalysis.Afix aexp2 heap2}$
<proof>

context *EdomArityAnalysis*
begin

lemma *Afix-edom*: $\text{edom } (\text{Afix } \Gamma \cdot \text{ae}) \subseteq \text{fv } \Gamma \cup \text{edom } \text{ae}$
<proof>

lemma *ABinds-lookup-fresh*:
 $\text{atom } v \# \Gamma \implies (\text{ABinds } \Gamma \cdot \text{ae}) v = \perp$
<proof>

lemma *Afix-lookup-fresh*:
assumes $\text{atom } v \# \Gamma$
shows $(\text{Afix } \Gamma \cdot \text{ae}) v = \text{ae } v$
<proof>

lemma *Afix-comp2join-fresh*:
 $\text{atom } \text{'(domA } \Delta) \# \Gamma \implies \text{ABinds } \Delta \cdot (\text{Afix } \Gamma \cdot \text{ae}) = \text{ABinds } \Delta \cdot \text{ae}$
<proof>

lemma *Afix-append-fresh*:
assumes $\text{atom } \text{'domA } \Delta \# \Gamma$
shows $\text{Afix } (\Delta @ \Gamma) \cdot \text{ae} = \text{Afix } \Gamma \cdot (\text{Afix } \Delta \cdot \text{ae})$
<proof>

lemma *Afix-e-to-heap*:
 $\text{Afix } (\text{delete } x \Gamma) \cdot (\text{fup } \Delta \cdot (\text{Aexp } e \cdot n \sqcup \text{ae})) \sqsubseteq \text{Afix } ((x, e) \# \text{delete } x \Gamma) \cdot (\text{esing } x \cdot n \sqcup \text{ae})$
<proof>

lemma *Afix-e-to-heap'*:
 $\text{Afix } (\text{delete } x \Gamma) \cdot (\text{Aexp } e \cdot n) \sqsubseteq \text{Afix } ((x, e) \# \text{delete } x \Gamma) \cdot (\text{esing } x \cdot (\text{up} \cdot n))$
<proof>

end

end

5.7 ArityAnalysisFixProps

theory *ArityAnalysisFixProps*
imports *ArityAnalysisFix ArityAnalysisSpec*


```

begin

context SubstArityAnalysis
begin

lemma Afix-restr-subst:
  assumes  $x \notin S$ 
  assumes  $y \notin S$ 
  assumes  $\text{dom}A \ \Gamma \subseteq S$ 
  shows  $\text{Afix } \Gamma[x::h=y].ae \ f|'S = \text{Afix } \Gamma.(ae \ f|'S) \ f|'S$ 
  <proof>
end

end

```

6 Arity Transformation

6.1 AbstractTransform

```

theory AbstractTransform
imports Launchbury.Terms TransformTools
begin

locale AbstractAnalProp =
  fixes PropApp :: 'a  $\Rightarrow$  'a::cont-pt
  fixes PropLam :: 'a  $\Rightarrow$  'a
  fixes AnalLet :: heap  $\Rightarrow$  exp  $\Rightarrow$  'a  $\Rightarrow$  'b::cont-pt
  fixes PropLetBody :: 'b  $\Rightarrow$  'a
  fixes PropLetHeap :: 'b  $\Rightarrow$  var  $\Rightarrow$  'a⊥
  fixes PropIfScrut :: 'a  $\Rightarrow$  'a
  assumes PropApp-eqvt:  $\pi \cdot \text{PropApp} \equiv \text{PropApp}$ 
  assumes PropLam-eqvt:  $\pi \cdot \text{PropLam} \equiv \text{PropLam}$ 
  assumes AnalLet-eqvt:  $\pi \cdot \text{AnalLet} \equiv \text{AnalLet}$ 
  assumes PropLetBody-eqvt:  $\pi \cdot \text{PropLetBody} \equiv \text{PropLetBody}$ 
  assumes PropLetHeap-eqvt:  $\pi \cdot \text{PropLetHeap} \equiv \text{PropLetHeap}$ 
  assumes PropIfScrut-eqvt:  $\pi \cdot \text{PropIfScrut} \equiv \text{PropIfScrut}$ 

locale AbstractAnalPropSubst = AbstractAnalProp +
  assumes AnalLet-subst:  $x \notin \text{dom}A \ \Gamma \Longrightarrow y \notin \text{dom}A \ \Gamma \Longrightarrow \text{AnalLet } (\Gamma[x::h=y]) \ (e[x::=y]) \ a$ 
  = AnalLet  $\Gamma \ e \ a$ 

locale AbstractTransform = AbstractAnalProp +
  constrains AnalLet :: heap  $\Rightarrow$  exp  $\Rightarrow$  'a::pure-cont-pt  $\Rightarrow$  'b::cont-pt
  fixes TransVar :: 'a  $\Rightarrow$  var  $\Rightarrow$  exp
  fixes TransApp :: 'a  $\Rightarrow$  exp  $\Rightarrow$  var  $\Rightarrow$  exp
  fixes TransLam :: 'a  $\Rightarrow$  var  $\Rightarrow$  exp  $\Rightarrow$  exp
  fixes TransLet :: 'b  $\Rightarrow$  heap  $\Rightarrow$  exp  $\Rightarrow$  exp

```

assumes *TransVar-eqvt*: $\pi \cdot \text{TransVar} = \text{TransVar}$
assumes *TransApp-eqvt*: $\pi \cdot \text{TransApp} = \text{TransApp}$
assumes *TransLam-eqvt*: $\pi \cdot \text{TransLam} = \text{TransLam}$
assumes *TransLet-eqvt*: $\pi \cdot \text{TransLet} = \text{TransLet}$
assumes *SuppTransLam*: $\text{supp} (\text{TransLam } a \ v \ e) \subseteq \text{supp } e - \text{supp } v$
assumes *SuppTransLet*: $\text{supp} (\text{TransLet } b \ \Gamma \ e) \subseteq \text{supp} (\Gamma, e) - \text{atom } \text{' } \text{dom}A \ \Gamma$

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>

nominal-termination *<proof>*

lemma *supp-transform*: $\text{supp} (\text{transform } a \ e) \subseteq \text{supp } e$
<proof>

lemma *fv-transform*: $\text{fv} (\text{transform } a \ e) \subseteq \text{fv } e$
<proof>

end

locale *AbstractTransformSubst* = *AbstractTransform* + *AbstractAnalPropSubst* +

assumes *TransVar-subst*: $(\text{TransVar } a \ v)[x ::= y] = (\text{TransVar } a \ v[x ::= y])$
assumes *TransApp-subst*: $(\text{TransApp } a \ e \ v)[x ::= y] = (\text{TransApp } a \ e[x ::= y] \ v[x ::= y])$
assumes *TransLam-subst*: $\text{atom } v \ \sharp (x, y) \implies (\text{TransLam } a \ v \ e)[x ::= y] = (\text{TransLam } a \ v[x ::= y] \ e[x ::= y])$
assumes *TransLet-subst*: $\text{atom } \text{' } \text{dom}A \ \Gamma \ \sharp^* (x, y) \implies (\text{TransLet } b \ \Gamma \ e)[x ::= y] = (\text{TransLet } b \ \Gamma[x ::= h = y] \ e[x ::= y])$

begin

lemma *subst-transform*: $(\text{transform } a \ e)[x ::= y] = \text{transform } a \ e[x ::= y]$
<proof>

end

locale *AbstractTransformBound* = *AbstractAnalProp* + *supp-bounded-transform* +

constrains *PropApp* :: '*a* \Rightarrow '*a*::*pure-cont-pt*
constrains *PropLetHeap* :: '*b*::*cont-pt* \Rightarrow *var* \Rightarrow '*a*_⊥
constrains *trans* :: '*c*::*cont-pt* \Rightarrow *exp* \Rightarrow *exp*
fixes *PropLetHeapTrans* :: '*b* \Rightarrow *var* \Rightarrow '*c*_⊥
assumes *PropLetHeapTrans-eqvt*: $\pi \cdot \text{PropLetHeapTrans} = \text{PropLetHeapTrans}$
assumes *TransBound-eqvt*: $\pi \cdot \text{trans} = \text{trans}$

begin

sublocale *AbstractTransform PropApp PropLam AnalLet PropLetBody PropLetHeap PropIf-Scrut*

($\lambda a. \text{Var}$)
($\lambda a. \text{App}$)
($\lambda a. \text{Terms.Lam}$)
($\lambda b \Gamma e. \text{Let } (\text{map-transform trans } (\text{PropLetHeapTrans } b) \Gamma) e$)
 $\langle \text{proof} \rangle$

lemma *isLam-transform[simp]*:
 $\text{isLam } (\text{transform } a \ e) \longleftrightarrow \text{isLam } e$
 $\langle \text{proof} \rangle$

lemma *isVal-transform[simp]*:
 $\text{isVal } (\text{transform } a \ e) \longleftrightarrow \text{isVal } e$
 $\langle \text{proof} \rangle$

end

locale *AbstractTransformBoundSubst = AbstractAnalPropSubst + AbstractTransformBound +*

assumes *TransBound-subst*: $(\text{trans } a \ e)[x::=y] = \text{trans } a \ e[x::=y]$

begin

sublocale *AbstractTransformSubst PropApp PropLam AnalLet PropLetBody PropLetHeap PropIf-Scrut*

($\lambda a. \text{Var}$)
($\lambda a. \text{App}$)
($\lambda a. \text{Terms.Lam}$)
($\lambda b \Gamma e. \text{Let } (\text{map-transform trans } (\text{PropLetHeapTrans } b) \Gamma) e$)
 $\langle \text{proof} \rangle$

end

end

6.2 ArityTransform

theory *ArityTransform*

imports *ArityAnalysisSig AbstractTransform ArityEtaExpansionSafe*

begin

context *ArityAnalysisHeapEqvt*

begin

sublocale *AbstractTransformBound*

$\lambda a. \text{inc} \cdot a$
 $\lambda a. \text{pred} \cdot a$
 $\lambda \Delta e a. (a, \text{Aheap } \Delta \ e \cdot a)$
 fst
 snd
 $\lambda -. 0$

Aeta-expand
snd
 ⟨*proof*⟩

abbreviation *transform-syn* (\mathcal{T}_-) **where** $\mathcal{T}_a \equiv \text{transform } a$

lemma *transform-simps*:

$\mathcal{T}_a (\text{App } e \ x) = \text{App } (\mathcal{T}_{\text{inc}\cdot a} e) \ x$
 $\mathcal{T}_a (\text{Lam } [x]. e) = \text{Lam } [x]. \mathcal{T}_{\text{pred}\cdot a} e$
 $\mathcal{T}_a (\text{Var } x) = \text{Var } x$
 $\mathcal{T}_a (\text{Let } \Gamma \ e) = \text{Let } (\text{map-transform } \text{Aeta-expand } (\text{Aheap } \Gamma \ e\cdot a) (\text{map-transform } (\lambda a. \mathcal{T}_a) (\text{Aheap } \Gamma \ e\cdot a) \ \Gamma)) (\mathcal{T}_a e)$
 $\mathcal{T}_a (\text{Bool } b) = \text{Bool } b$
 $\mathcal{T}_a (\text{scrut } ? e1 : e2) = (\mathcal{T}_0 \text{ scrut } ? \mathcal{T}_a e1 : \mathcal{T}_a e2)$
 ⟨*proof*⟩

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* $\Gamma \cup \text{upds } S$
 ⇒ *Astack S* ⊆ *a*

$\implies (ABinds \Gamma \cdot ae \sqcup Aexp \ e \cdot a \sqcup AEstack \ as \ S) f|' (domA \ \Gamma \cup upds \ S) \sqsubseteq ae$
 $\implies \text{stack-consistent as } S$
 $\implies a\text{-consistent } (ae, a, as) (\Gamma, e, S)$

inductive-cases $a\text{-consistent}E$: $a\text{-consistent } (ae, a, as) (\Gamma, e, S)$

lemma $a\text{-consistent-restrict}A$:

assumes $a\text{-consistent } (ae, a, as) (\Gamma, e, S)$

assumes $edom \ ae \subseteq V$

shows $a\text{-consistent } (ae, a, as) (\text{restrict}A \ V \ \Gamma, e, S)$

$\langle \text{proof} \rangle$

lemma $a\text{-consistent-edom-subset}D$:

$a\text{-consistent } (ae, a, as) (\Gamma, e, S) \implies edom \ ae \subseteq domA \ \Gamma \cup upds \ S$

$\langle \text{proof} \rangle$

lemma $a\text{-consistent-stack}D$:

$a\text{-consistent } (ae, a, as) (\Gamma, e, S) \implies Astack \ S \sqsubseteq a$

$\langle \text{proof} \rangle$

lemma $a\text{-consistent-app}_1$:

$a\text{-consistent } (ae, a, as) (\Gamma, App \ e \ x, S) \implies a\text{-consistent } (ae, inc \cdot a, as) (\Gamma, e, Arg \ x \ \# \ S)$

$\langle \text{proof} \rangle$

lemma $a\text{-consistent-app}_2$:

assumes $a\text{-consistent } (ae, a, as) (\Gamma, (Lam \ [y]. \ e), Arg \ x \ \# \ S)$

shows $a\text{-consistent } (ae, (pred \cdot a), as) (\Gamma, e[y::=x], S)$

$\langle \text{proof} \rangle$

lemma $a\text{-consistent-thunk-0}$:

assumes $a\text{-consistent } (ae, a, as) (\Gamma, Var \ x, S)$

assumes $map\text{-of } \Gamma \ x = Some \ e$

assumes $ae \ x = up \cdot 0$

shows $a\text{-consistent } (ae, 0, as) (\text{delete } x \ \Gamma, e, Upd \ x \ \# \ S)$

$\langle \text{proof} \rangle$

lemma $a\text{-consistent-thunk-once}$:

assumes $a\text{-consistent } (ae, a, as) (\Gamma, Var \ x, S)$

assumes $map\text{-of } \Gamma \ x = Some \ e$

assumes $[simp]$: $ae \ x = up \cdot u$

assumes $heap\text{-upds-ok } (\Gamma, S)$

shows $a\text{-consistent } (env\text{-delete } x \ ae, u, as) (\text{delete } x \ \Gamma, e, S)$

$\langle \text{proof} \rangle$

lemma $a\text{-consistent-lamvar}$:

assumes $a\text{-consistent } (ae, a, as) (\Gamma, Var \ x, S)$

assumes $map\text{-of } \Gamma \ x = Some \ e$

assumes $[simp]$: $ae \ x = up \cdot u$

shows $a\text{-consistent } (ae, u, as) ((x,e)\# \ \text{delete } x \ \Gamma, e, S)$

<proof>

lemma

assumes *a-consistent* (*ae*, *a*, *as*) (Γ , *e*, *Upd* *x* # *S*)
shows *a-consistent-var₂*: *a-consistent* (*ae*, *a*, *as*) ((*x*, *e*) # Γ , *e*, *S*)
and *a-consistent-UpdD*: *ae* *x* = *up*·*0a* = 0
<proof>

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* \cap *domA* Δ = {}
shows *a-consistent* (*Aheap* Δ *e*·*a* \sqcup *ae*, *a*, *as*) (Δ @ Γ , *e*, *S*)
<proof>

lemma *a-consistent-if₁*:

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>

lemma *a-consistent-if₂*:

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*)
<proof>

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
<proof>

lemma *closed-a-consistent*:

fv *e* = ({}::*var set*) \implies *a-consistent* (\perp , 0, \square) (\square , *e*, \square)
<proof>

end

end

7.2 ArityTransformSafe

theory *ArityTransformSafe*

imports *ArityTransform* *ArityConsistent* *ArityAnalysisSpec* *ArityEtaExpansionSafe* *AbstractTransform* *ConstOn*

begin

locale *CardinalityArityTransformation* = *ArityAnalysisLetSafeNoCard*

begin

sublocale *AbstractTransformBoundSubst*

```

λ a . inc·a
λ a . pred·a
λ Δ e a . (a, Ahead Δ e·a)
fst
snd
λ -. 0
Aeta-expand
snd
⟨proof⟩

```

abbreviation *ccTransform* **where** *ccTransform* \equiv *transform*

lemma *supp-transform*: $\text{supp} (\text{transform } a \ e) \subseteq \text{supp } e$
⟨proof⟩

interpretation *supp-bounded-transform* *transform*
⟨proof⟩

```

fun transform-alts :: Arity list  $\Rightarrow$  stack  $\Rightarrow$  stack
where
  transform-alts - [] = []
  | transform-alts (a#as) (Alts e1 e2 # S) = (Alts (ccTransform a e1) (ccTransform a e2))
# transform-alts as S
  | transform-alts as (x # S) = x # transform-alts as S

```

lemma *transform-alt*s-Nil[*simp*]: *transform-alt*s [] S = S
⟨proof⟩

lemma *Astack-transform-alt*s[*simp*]:
Astack (*transform-alt*s as S) = *Astack* S
⟨proof⟩

lemma *fresh-star-transform-alt*s[*intro*]: $a \#* S \Longrightarrow a \#* \text{transform-alt } as \ S$
⟨proof⟩

```

fun a-transform :: astate  $\Rightarrow$  conf  $\Rightarrow$  conf
where a-transform (ae, a, as) (Γ, e, S) =
  (map-transform Aeta-expand ae (map-transform ccTransform ae Γ),
   ccTransform a e,
   transform-alts as S)

```

```

fun restr-conf :: var set  $\Rightarrow$  conf  $\Rightarrow$  conf
where restr-conf V (Γ, e, S) = (restrictA V Γ, e, restr-stack V S)

```

inductive *consistent* :: astate \Rightarrow conf \Rightarrow bool **where**

```

consistentI[intro!]:
  a-consistent (ae, a, as) (Γ, e, S)
 $\Longrightarrow$  ( $\bigwedge x. x \in \text{thunks } \Gamma \Longrightarrow ae \ x = up \cdot 0$ )
 $\Longrightarrow$  consistent (ae, a, as) (Γ, e, S)

```

inductive-cases *consistentE*[*elim!*]: *consistent* (ae, a, as) (Γ, e, S)

```

lemma closed-consistent:
  assumes fv e = ({ }::var set)
  shows consistent ( $\perp$ ,  $\theta$ ,  $\square$ ) ( $\square$ , e,  $\square$ )
  <proof>

lemma arity-transform-safe:
  fixes c c'
  assumes c  $\Rightarrow^*$  c' and  $\neg$  boring-step c' and heap-upds-ok-conf c and consistent (ae,a,as)
  c
  shows  $\exists ae' a' as'. consistent (ae',a',as') c' \wedge a\text{-transform } (ae,a,as) c \Rightarrow^* a\text{-transform}$ 
   $(ae',a',as') c'$ 
  <proof>
end

end

```

8 Cardinality Analysis

8.1 Cardinality-Domain

```

theory Cardinality-Domain
imports Launchbury.HOLCF-Utills
begin

type-synonym oneShot = one
abbreviation notOneShot :: oneShot where notOneShot  $\equiv$  ONE
abbreviation oneShot :: oneShot where oneShot  $\equiv$   $\perp$ 

type-synonym two = oneShot  $\perp$ 
abbreviation many :: two where many  $\equiv$  up.notOneShot
abbreviation once :: two where once  $\equiv$  up.oneShot
abbreviation none :: two where none  $\equiv$   $\perp$ 

lemma many-max[simp]: a  $\sqsubseteq$  many <proof>

lemma two-conj: c = many  $\vee$  c = once  $\vee$  c = none <proof>

lemma two-cases[case-names many once none]:
  obtains c = many | c = once | c = none <proof>

definition two-pred where two-pred = ( $\Lambda x. if x  $\sqsubseteq$  once then  $\perp$  else x)$ 

lemma two-pred-simp: two-pred.c = (if c  $\sqsubseteq$  once then  $\perp$  else c)
  <proof>

lemma two-pred-simps[simp]:
  two-pred.many = many

```


$two\text{-}pred \cdot once = none$
 $two\text{-}pred \cdot none = none$
 <proof>

lemma *two-pred-below-arg*: $two\text{-}pred \cdot f \sqsubseteq f$
 <proof>

lemma *two-pred-none*: $two\text{-}pred \cdot c = none \longleftrightarrow c \sqsubseteq once$
 <proof>

definition *record-call* **where** $record\text{-}call\ x = (\Lambda\ ce.\ (\lambda\ y.\ \text{if } x = y \text{ then } two\text{-}pred \cdot (ce\ y) \text{ else } ce\ y))$

lemma *record-call-simp*: $(record\text{-}call\ x \cdot f)\ x' = (\text{if } x = x' \text{ then } two\text{-}pred \cdot (f\ x') \text{ else } f\ x')$
 <proof>

lemma *record-call[simp]*: $(record\text{-}call\ x \cdot f)\ x = two\text{-}pred \cdot (f\ x)$
 <proof>

lemma *record-call-other[simp]*: $x' \neq x \implies (record\text{-}call\ x \cdot f)\ x' = f\ x'$
 <proof>

lemma *record-call-below-arg*: $record\text{-}call\ x \cdot f \sqsubseteq f$
 <proof>

definition *two-add* :: $two \rightarrow two \rightarrow two$
where $two\text{-}add = (\Lambda\ x.\ (\Lambda\ y.\ \text{if } x \sqsubseteq \perp \text{ then } y \text{ else } (\text{if } y \sqsubseteq \perp \text{ then } x \text{ else } many)))$

lemma *two-add-simp*: $two\text{-}add \cdot x \cdot y = (\text{if } x \sqsubseteq \perp \text{ then } y \text{ else } (\text{if } y \sqsubseteq \perp \text{ then } x \text{ else } many))$
 <proof>

lemma *two-pred-two-add-once*: $c \sqsubseteq two\text{-}pred \cdot (two\text{-}add \cdot once \cdot c)$
 <proof>

end

8.2 CardinalityAnalysisSig

theory *CardinalityAnalysisSig*
imports *AEnv Cardinality-Domain SestoftConf*
begin

locale *CardinalityPrognosis* =
fixes $prognosis :: AEnv \Rightarrow Arity\ list \Rightarrow Arity \Rightarrow conf \Rightarrow (var \Rightarrow two)$

locale *CardinalityHeap* =
fixes $cHeap :: heap \Rightarrow exp \Rightarrow Arity \rightarrow (var \Rightarrow two)$

end

8.3 CardinalityAnalysisSpec

theory *CardinalityAnalysisSpec*

imports *ArityAnalysisSpec* *CardinalityAnalysisSig* *ConstOn*

begin

locale *CardinalityPrognosisEdom* = *CardinalityPrognosis* +

assumes *edom-prognosis*:

$edom (prognosis\ ae\ as\ a\ (\Gamma, e, S)) \subseteq fv\ \Gamma \cup fv\ e \cup fv\ S$

locale *CardinalityPrognosisShape* = *CardinalityPrognosis* +

assumes *prognosis-env-cong*: $ae\ f|' domA\ \Gamma = ae'\ f|' domA\ \Gamma \implies prognosis\ ae\ as\ u\ (\Gamma, e, S) = prognosis\ ae'\ as\ u\ (\Gamma, e, S)$

assumes *prognosis-reorder*: $map-of\ \Gamma = map-of\ \Delta \implies prognosis\ ae\ as\ u\ (\Gamma, e, S) = prognosis\ ae\ as\ u\ (\Delta, e, S)$

assumes *prognosis-ap*: $const-on\ (prognosis\ ae\ as\ a\ (\Gamma, e, S))\ (ap\ S)\ many$

assumes *prognosis-upd*: $prognosis\ ae\ as\ u\ (\Gamma, e, S) \sqsubseteq prognosis\ ae\ as\ u\ (\Gamma, e, Upd\ x\ \# S)$

assumes *prognosis-not-called*: $ae\ x = \perp \implies prognosis\ ae\ as\ a\ (\Gamma, e, S) \sqsubseteq prognosis\ ae\ as\ a\ (delete\ x\ \Gamma, e, S)$

assumes *prognosis-called*: $once \sqsubseteq prognosis\ ae\ as\ a\ (\Gamma, Var\ x, S)\ x$

locale *CardinalityPrognosisApp* = *CardinalityPrognosis* +

assumes *prognosis-App*: $prognosis\ ae\ as\ (inc \cdot a)\ (\Gamma, e, Arg\ x\ \# S) \sqsubseteq prognosis\ ae\ as\ a\ (\Gamma, App\ e\ x, S)$

locale *CardinalityPrognosisLam* = *CardinalityPrognosis* +

assumes *prognosis-subst-Lam*: $prognosis\ ae\ as\ (pred \cdot a)\ (\Gamma, e[y::=x], S) \sqsubseteq prognosis\ ae\ as\ a\ (\Gamma, Lam\ [y].\ e, Arg\ x\ \# S)$

locale *CardinalityPrognosisVar* = *CardinalityPrognosis* +

assumes *prognosis-Var-lam*: $map-of\ \Gamma\ x = Some\ e \implies ae\ x = up \cdot u \implies isVal\ e \implies prognosis\ ae\ as\ u\ (\Gamma, e, S) \sqsubseteq record-call\ x \cdot (prognosis\ ae\ as\ a\ (\Gamma, Var\ x, S))$

assumes *prognosis-Var-thunk*: $map-of\ \Gamma\ x = Some\ e \implies ae\ x = up \cdot u \implies \neg isVal\ e \implies prognosis\ ae\ as\ u\ (delete\ x\ \Gamma, e, Upd\ x\ \# S) \sqsubseteq record-call\ x \cdot (prognosis\ ae\ as\ a\ (\Gamma, Var\ x, S))$

assumes *prognosis-Var2*: $isVal\ e \implies x \notin domA\ \Gamma \implies prognosis\ ae\ as\ 0\ ((x, e) \# \Gamma, e, S) \sqsubseteq prognosis\ ae\ as\ 0\ (\Gamma, e, Upd\ x\ \# S)$

locale *CardinalityPrognosisIfThenElse* = *CardinalityPrognosis* +

assumes *prognosis-IfThenElse*: $prognosis\ ae\ (a \# as)\ 0\ (\Gamma, scrut, Alts\ e1\ e2\ \# S) \sqsubseteq prognosis\ ae\ as\ a\ (\Gamma, scrut\ ?\ e1 : e2, S)$

assumes *prognosis-Alts*: $prognosis\ ae\ as\ a\ (\Gamma, if\ b\ then\ e1\ else\ e2, S) \sqsubseteq prognosis\ ae\ (a \# as)\ 0\ (\Gamma, Bool\ b, Alts\ e1\ e2\ \# S)$

locale *CardinalityPrognosisLet* = *CardinalityPrognosis* + *CardinalityHeap* + *ArityAnalysisHeap* +

assumes *prognosis-Let*:

$atom \text{ ' } domA \Delta \#* \Gamma \implies atom \text{ ' } domA \Delta \#* S \implies edom \text{ } ae \subseteq domA \Gamma \cup upds S \implies prognosis$
 $(Aheap \Delta e \cdot a \sqcup ae) \text{ as } a (\Delta @ \Gamma, e, S) \sqsubseteq cHeap \Delta e \cdot a \sqcup prognosis \text{ } ae \text{ as } a (\Gamma, Terms.Let \Delta e,$
 $S)$

locale *CardinalityHeapSafe* = *CardinalityHeap* + *ArietyAnalysisHeap* +
assumes *Aheap-heap3*: $x \in thunks \Gamma \implies many \sqsubseteq (cHeap \Gamma e \cdot a) x \implies (Aheap \Gamma e \cdot a) x =$
 $up \cdot 0$
assumes *edom-cHeap*: $edom (cHeap \Delta e \cdot a) = edom (Aheap \Delta e \cdot a)$

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

end

8.4 NoCardinalityAnalysis

theory *NoCardinalityAnalysis*
imports *CardinalityAnalysisSpec* *ArietyAnalysisStack*
begin

locale *NoCardinalityAnalysis* = *ArietyAnalysisLetSafe* +
assumes *Aheap-thunk*: $x \in thunks \Gamma \implies (Aheap \Gamma e \cdot a) x = up \cdot 0$
begin

definition *a2c* :: $Ariety_{\perp} \rightarrow two$ **where** $a2c = (\Lambda a. \text{if } a \sqsubseteq \perp \text{ then } \perp \text{ else } many)$

lemma *a2c-simp*: $a2c \cdot a = (\text{if } a \sqsubseteq \perp \text{ then } \perp \text{ else } many)$
 $\langle proof \rangle$

lemma *a2c-eqvt[eqvt]*: $\pi \cdot a2c = a2c$
 $\langle proof \rangle$

definition *ae2ce* :: $AEnv \Rightarrow (var \Rightarrow two)$ **where** $ae2ce \text{ } ae \text{ } x = a2c \cdot (ae \text{ } x)$

lemma *ae2ce-cont*: $cont \text{ } ae2ce$
 $\langle proof \rangle$

lemmas *cont-compose[OF ae2ce-cont, cont2cont, simp]*

lemma *ae2ce-eqvt[eqvt]*: $\pi \cdot ae2ce \text{ } ae \text{ } x = ae2ce (\pi \cdot ae) (\pi \cdot x)$
 $\langle proof \rangle$

lemma *ae2ce-to-env-restr*: $ae2ce\ ae = (\lambda\ .\ many)\ f|' edom\ ae$
<proof>

lemma *edom-ae2ce[simp]*: $edom\ (ae2ce\ ae) = edom\ ae$
<proof>

definition *cHeap* :: $heap \Rightarrow exp \Rightarrow Arity \rightarrow (var \Rightarrow two)$
where $cHeap\ \Gamma\ e = (\Lambda\ a.\ ae2ce\ (Aheap\ \Gamma\ e\ a))$

lemma *cHeap-simp[simp]*: $cHeap\ \Gamma\ e\ a = ae2ce\ (Aheap\ \Gamma\ e\ a)$
<proof>

sublocale *CardinalityHeap* *cHeap**<proof>*

sublocale *CardinalityHeapSafe* *cHeap* *Aheap*
<proof>

fun *prognosis* **where**

$prognosis\ ae\ as\ a\ (\Gamma,\ e,\ S) = ((\lambda\ .\ many)\ f|' (edom\ (ABinds\ \Gamma\ ae) \cup edom\ (Aexp\ e\ a) \cup edom\ (AEstack\ as\ S)))$

lemma *record-all-noop[simp]*:
 $record\ call\ x\ ((\lambda\ .\ many)\ f|' S) = (\lambda\ .\ many)\ f|' S$
<proof>

lemma *const-on-restr-constI[intro]*:
 $S' \subseteq S \implies const\ on\ ((\lambda\ .\ x)\ f|' S)\ S'\ x$
<proof>

lemma *ap-subset-edom-AEstack*: $ap\ S \subseteq edom\ (AEstack\ as\ S)$
<proof>

sublocale *CardinalityPrognosis* *prognosis**<proof>*

sublocale *CardinalityPrognosisShape* *prognosis*
<proof>

sublocale *CardinalityPrognosisApp* *prognosis*
<proof>

sublocale *CardinalityPrognosisLam* *prognosis*
<proof>

sublocale *CardinalityPrognosisVar* *prognosis*
<proof>

sublocale *CardinalityPrognosisIfThenElse* *prognosis*
<proof>

sublocale *CardinalityPrognosisLet prognosis cHeap Ahead*
 ⟨*proof*⟩

sublocale *CardinalityPrognosisEdom prognosis*
 ⟨*proof*⟩

sublocale *CardinalityPrognosisSafe prognosis cHeap Ahead Aexp*⟨*proof*⟩
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

⟨*proof*⟩

abbreviation *ccTransform* **where** *ccTransform* \equiv *transform*

lemma *supp-transform*: *supp* (*transform a e*) \subseteq *supp e*

⟨*proof*⟩

interpretation *supp-bounded-transform transform*

⟨*proof*⟩

type-synonym *tstate* = (*AEnv* \times (*var* \Rightarrow *two*) \times *Arity* \times *Arity list* \times *var list*)

fun *transform-alts* :: *Arity list* \Rightarrow *stack* \Rightarrow *stack*

where

transform-alts - [] = []

| *transform-alts* (*a*#*as*) (*Alts e1 e2* # *S*) = (*Alts* (*ccTransform a e1*) (*ccTransform a e2*))

transform-alts as S

| *transform-alts as* (*x* # *S*) = *x* # *transform-alts as S*

lemma *transform-alts-Nil[simp]*: $\text{transform-alts } [] S = S$
 ⟨proof⟩

lemma *Astack-transform-alts[simp]*:
 $Astack (\text{transform-alts } as S) = Astack S$
 ⟨proof⟩

lemma *fresh-star-transform-alts[intro]*: $a \#* S \implies a \#* \text{transform-alts } as S$
 ⟨proof⟩

fun *a-transform* :: $astate \Rightarrow conf \Rightarrow conf$
where *a-transform* (ae, a, as) (Γ, e, S) =
 ($\text{map-transform } Aeta\text{-expand } ae (\text{map-transform } ccTransform ae \Gamma),$
 $ccTransform a e,$
 $\text{transform-alts } as S$)

fun *restr-conf* :: $var\ set \Rightarrow conf \Rightarrow conf$
where *restr-conf* $V (\Gamma, e, S) = (\text{restrictA } V \Gamma, e, \text{restr-stack } V S)$

fun *add-dummies-conf* :: $var\ list \Rightarrow conf \Rightarrow conf$
where *add-dummies-conf* $l (\Gamma, e, S) = (\Gamma, e, S @ \text{map } Dummy (\text{rev } l))$

fun *conf-transform* :: $tstate \Rightarrow conf \Rightarrow conf$
where *conf-transform* (ae, ce, a, as, r) $c = \text{add-dummies-conf } r ((a\text{-transform } (ae, a, as)$
 $(\text{restr-conf } (- \text{set } r) c)))$

inductive *consistent* :: $tstate \Rightarrow conf \Rightarrow bool$ **where**
consistentI[intro!]:
 $a\text{-consistent } (ae, a, as) (\text{restr-conf } (- \text{set } r) (\Gamma, e, S))$
 $\implies \text{edom } ae = \text{edom } ce$
 $\implies \text{prognosis } ae \text{ as } a (\Gamma, e, S) \sqsubseteq ce$
 $\implies (\bigwedge x. x \in \text{thunks } \Gamma \implies \text{many } \sqsubseteq ce x \implies ae x = \text{up}\cdot 0)$
 $\implies \text{set } r \subseteq (\text{domA } \Gamma \cup \text{upds } S) - \text{edom } ce$
 $\implies \text{consistent } (ae, ce, a, as, r) (\Gamma, e, S)$

inductive-cases *consistentE[elim!]*: $\text{consistent } (ae, ce, a, as) (\Gamma, e, S)$

lemma *closed-consistent*:
assumes $fv\ e = (\{\}\ :: var\ set)$
shows $\text{consistent } (\perp, \perp, \theta, [], []) ([], e, [])$
 ⟨proof⟩

lemma *card-arity-transform-safe*:
fixes $c\ c'$
assumes $c \Rightarrow^* c'$ **and** $\neg \text{boring-step } c'$ **and** $\text{heap-upds-ok-conf } c$ **and** $\text{consistent } (ae, ce, a, as, r)$
 c
shows $\exists ae' ce' a' as' r'. \text{consistent } (ae', ce', a', as', r') c' \wedge \text{conf-transform } (ae, ce, a, as, r) c$
 $\Rightarrow_{G^*} \text{conf-transform } (ae', ce', a', as', r') c'$
 ⟨proof⟩

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  $\Rightarrow$  bool where
  downset xss = ( $\forall x n. x \in xss \longrightarrow take\ n\ x \in xss$ )
```

```
lemma downsetE[elim]:
  downset xss  $\Longrightarrow$  xs  $\in$  xss  $\Longrightarrow$  butlast xs  $\in$  xss
<proof>
```

```
lemma downset-appendE[elim]:
  downset xss  $\Longrightarrow$  xs@ys  $\in$  xss  $\Longrightarrow$  xs  $\in$  xss
<proof>
```

```
lemma downset-hdE[elim]:
  downset xss  $\Longrightarrow$  xs  $\in$  xss  $\Longrightarrow$  xs  $\neq$  []  $\Longrightarrow$  [hd xs]  $\in$  xss
<proof>
```

```
lemma downsetI[intro]:
  assumes  $\bigwedge xs. xs \in xss \Longrightarrow xs \neq [] \Longrightarrow butlast\ xs \in xss$ 
  shows downset xss
<proof>
```

```
lemma [simp]: downset {[]} <proof>
```

```
lemma downset-mapI: downset xss  $\Longrightarrow$  downset (map f ' xss)
<proof>
```

```
lemma downset-filter:
  assumes downset xss
  shows downset (filter P ' xss)
<proof>
```

```
lemma downset-set-subset:
  downset ({xs. set xs  $\subseteq$  S})
<proof>
```

9.1.2 The type of infinite labeled trees

typedef 'a ttree = {xss :: 'a list set . [] ∈ xss ∧ downset xss} ⟨proof⟩

setup-lifting type-definition-ttree

9.1.3 Deconstructors

lift-definition possible :: 'a ttree ⇒ 'a ⇒ bool
is λ xss x. ∃ xs. x#xs ∈ xss ⟨proof⟩

lift-definition next :: 'a ttree ⇒ 'a ⇒ 'a ttree
is λ xss x. insert [] {xs | xs. x#xs ∈ xss}
⟨proof⟩

9.1.4 Trees as set of paths

lift-definition paths :: 'a ttree ⇒ 'a list set **is** (λ x. x) ⟨proof⟩

lemma paths-inj: paths t = paths t' ⇒ t = t' ⟨proof⟩

lemma paths-injs-simps[simp]: paths t = paths t' ⇔ t = t' ⟨proof⟩

lemma paths-Nil[simp]: [] ∈ paths t ⟨proof⟩

lemma paths-not-empty[simp]: (paths t = {}) ⇔ False ⟨proof⟩

lemma paths-Cons-next:
possible t x ⇒ xs ∈ paths (next t x) ⇒ (x#xs) ∈ paths t
⟨proof⟩

lemma paths-Cons-next-iff:
possible t x ⇒ xs ∈ paths (next t x) ⇔ (x#xs) ∈ paths t
⟨proof⟩

lemma possible-mono:
paths t ⊆ paths t' ⇒ possible t x ⇒ possible t' x
⟨proof⟩

lemma next-mono:
paths t ⊆ paths t' ⇒ paths (next t x) ⊆ paths (next t' x)
⟨proof⟩

lemma tree-eqI: (∧ x xs. x#xs ∈ paths t ⇔ x#xs ∈ paths t') ⇒ t = t'
⟨proof⟩

lemma paths-next[elim]:
assumes xs ∈ paths (next t x)
obtains x#xs ∈ paths t | xs = []
⟨proof⟩

lemma *Cons-path*: $x \# xs \in \text{paths } t \longleftrightarrow \text{possible } t \ x \wedge xs \in \text{paths } (\text{next } t \ x)$
 ⟨proof⟩

lemma *Cons-pathI[intro]*:
 assumes $\text{possible } t \ x \longleftrightarrow \text{possible } t' \ x$
 assumes $\text{possible } t \ x \implies \text{possible } t' \ x \implies xs \in \text{paths } (\text{next } t \ x) \longleftrightarrow xs \in \text{paths } (\text{next } t' \ x)$
 shows $x \# xs \in \text{paths } t \longleftrightarrow x \# xs \in \text{paths } t'$
 ⟨proof⟩

lemma *paths-next-eq*: $xs \in \text{paths } (\text{next } t \ x) \longleftrightarrow xs = [] \vee x \# xs \in \text{paths } t$
 ⟨proof⟩

lemma *tree-coinduct*:
 assumes $P \ t \ t'$
 assumes $\bigwedge t \ t' \ x . P \ t \ t' \implies \text{possible } t \ x \longleftrightarrow \text{possible } t' \ x$
 assumes $\bigwedge t \ t' \ x . P \ t \ t' \implies \text{possible } t \ x \implies \text{possible } t' \ x \implies P \ (\text{next } t \ x) \ (\text{next } t' \ x)$
 shows $t = t'$
 ⟨proof⟩

9.1.5 The carrier of a tree

lift-definition *carrier* :: 'a tree \Rightarrow 'a set is $\lambda \text{ xss} . \bigcup (\text{set } ' \ \text{xss})$ ⟨proof⟩

lemma *carrier-mono*: $\text{paths } t \subseteq \text{paths } t' \implies \text{carrier } t \subseteq \text{carrier } t'$ ⟨proof⟩

lemma *carrier-possible*:
 $\text{possible } t \ x \implies x \in \text{carrier } t$ ⟨proof⟩

lemma *carrier-possible-subset*:
 $\text{carrier } t \subseteq A \implies \text{possible } t \ x \implies x \in A$ ⟨proof⟩

lemma *carrier-next-subset*:
 $\text{carrier } (\text{next } t \ x) \subseteq \text{carrier } t$
 ⟨proof⟩

lemma *Union-paths-carrier*: $(\bigcup x \in \text{paths } t . \text{set } x) = \text{carrier } t$
 ⟨proof⟩

9.1.6 Repeatable trees

definition *repeatable* **where** $\text{repeatable } t = (\forall x . \text{possible } t \ x \longrightarrow \text{next } t \ x = t)$

lemma *next-repeatable[simp]*: $\text{repeatable } t \implies \text{possible } t \ x \implies \text{next } t \ x = t$
 ⟨proof⟩

9.1.7 Simple trees

lift-definition *empty* :: 'a tree is $\{[]\}$ ⟨proof⟩

lemma *possible-empty[simp]*: *possible empty* $x' \longleftrightarrow \text{False}$
<proof>

lemma *next-not-possible[simp]*: $\neg \text{possible } t \ x \implies \text{next } t \ x = \text{empty}$
<proof>

lemma *paths-empty[simp]*: *paths empty* = $\{\{\}\}$ *<proof>*

lemma *carrier-empty[simp]*: *carrier empty* = $\{\}$ *<proof>*

lemma *repeatable-empty[simp]*: *repeatable empty* *<proof>*

lift-definition *single* :: $'a \Rightarrow 'a \text{ tree}$ **is** $\lambda x. \{\[], [x]\}$
<proof>

lemma *possible-single[simp]*: *possible (single* $x) \ x' \longleftrightarrow x = x'$
<proof>

lemma *next-single[simp]*: *next (single* $x) \ x' = \text{empty}$
<proof>

lemma *carrier-single[simp]*: *carrier (single* $y) = \{y\}$
<proof>

lemma *paths-single[simp]*: *paths (single* $x) = \{\[], [x]\}$
<proof>

lift-definition *many-calls* :: $'a \Rightarrow 'a \text{ tree}$ **is** $\lambda x. \text{range } (\lambda n. \text{replicate } n \ x)$
<proof>

lemma *possible-many-calls[simp]*: *possible (many-calls* $x) \ x' \longleftrightarrow x = x'$
<proof>

lemma *next-many-calls[simp]*: *next (many-calls* $x) \ x' = (\text{if } x' = x \text{ then many-calls } x \text{ else empty})$
<proof>

lemma *repeatable-many-calls*: *repeatable (many-calls* $x)$
<proof>

lemma *carrier-many-calls[simp]*: *carrier (many-calls* $x) = \{x\}$ *<proof>*

lift-definition *anything* :: $'a \text{ tree}$ **is** *UNIV*
<proof>

lemma *possible-anything[simp]*: *possible anything* $x' \longleftrightarrow \text{True}$
<proof>

lemma *next-anything[simp]*: *next anything* $x = \text{anything}$

<proof>

lemma *paths-anything[simp]*:
paths anything = UNIV <proof>

lemma *carrier-anything[simp]*:
carrier anything = UNIV
<proof>

lift-definition *many-among* :: 'a set \Rightarrow 'a tree **is** $\lambda S. \{xs . set xs \subseteq S\}$
<proof>

lemma *carrier-many-among[simp]*: *carrier (many-among S) = S*
<proof>

9.1.8 Intersection of two trees

lift-definition *intersect* :: 'a tree \Rightarrow 'a tree \Rightarrow 'a tree (**infixl** \cap 80)
is (\cap)
<proof>

lemma *paths-intersect[simp]*: *paths (t \cap t') = paths t \cap paths t'*
<proof>

lemma *carrier-intersect*: *carrier (t \cap t') \subseteq carrier t \cap carrier t'*
<proof>

9.1.9 Disjoint union of trees

lift-definition *either* :: 'a tree \Rightarrow 'a tree \Rightarrow 'a tree (**infixl** \oplus 80)
is (\cup)
<proof>

lemma *either-empty1[simp]*: *empty \oplus t = t*
<proof>

lemma *either-empty2[simp]*: *t \oplus empty = t*
<proof>

lemma *either-sym[simp]*: *t \oplus t2 = t2 \oplus t*
<proof>

lemma *either-idem[simp]*: *t \oplus t = t*
<proof>

lemma *possible-either[simp]*: *possible (t \oplus t') x \longleftrightarrow possible t x \vee possible t' x*
<proof>

lemma *nxt-either[simp]*: *nxt (t \oplus t') x = nxt t x \oplus nxt t' x*
<proof>

lemma *paths-either[simp]*: *paths (t \oplus t') = paths t \cup paths t'*
<proof>

lemma *carrier-either*[simp]:
 $\text{carrier } (t \oplus \oplus t') = \text{carrier } t \cup \text{carrier } t'$
 ⟨proof⟩

lemma *either-contains-arg1*: $\text{paths } t \subseteq \text{paths } (t \oplus \oplus t')$
 ⟨proof⟩

lemma *either-contains-arg2*: $\text{paths } t' \subseteq \text{paths } (t \oplus \oplus t')$
 ⟨proof⟩

lift-definition *Either* :: 'a tree set \Rightarrow 'a tree **is** $\lambda S. \text{insert } [] (\bigcup S)$
 ⟨proof⟩

lemma *paths-Either*: $\text{paths } (\text{Either } ts) = \text{insert } [] (\bigcup (\text{paths } ' ts))$
 ⟨proof⟩

9.1.10 Merging of trees

lemma *ex-ex-eq-hint*: $(\exists x. (\exists xs \ ys. x = f \ xs \ ys \wedge P \ xs \ ys) \wedge Q \ x) \longleftrightarrow (\exists xs \ ys. Q \ (f \ xs \ ys) \wedge P \ xs \ ys)$
 ⟨proof⟩

lift-definition *both* :: 'a tree \Rightarrow 'a tree \Rightarrow 'a tree (**infixl** $\otimes \otimes$ 86)
is $\lambda xss \ yss . \bigcup \{xs \otimes \ ys \mid xs \ ys. xs \in xss \wedge ys \in yss\}$
 ⟨proof⟩

lemma *both-assoc*[simp]: $t \otimes \otimes (t' \otimes \otimes t'') = (t \otimes \otimes t') \otimes \otimes t''$
 ⟨proof⟩

lemma *both-comm*: $t \otimes \otimes t' = t' \otimes \otimes t$
 ⟨proof⟩

lemma *both-empty1*[simp]: $\text{empty} \otimes \otimes t = t$
 ⟨proof⟩

lemma *both-empty2*[simp]: $t \otimes \otimes \text{empty} = t$
 ⟨proof⟩

lemma *paths-both*: $xs \in \text{paths } (t \otimes \otimes t') \longleftrightarrow (\exists \ ys \in \text{paths } t. \exists \ zs \in \text{paths } t'. xs \in ys \otimes \ zs)$
 ⟨proof⟩

lemma *both-contains-arg1*: $\text{paths } t \subseteq \text{paths } (t \otimes \otimes t')$
 ⟨proof⟩

lemma *both-contains-arg2*: $\text{paths } t' \subseteq \text{paths } (t \otimes \otimes t')$
 ⟨proof⟩

lemma *both-mono1*:

$paths\ t \subseteq paths\ t' \implies paths\ (t \otimes t') \subseteq paths\ (t' \otimes t')$
 ⟨proof⟩

lemma *both-mono2*:

$paths\ t \subseteq paths\ t' \implies paths\ (t'' \otimes t) \subseteq paths\ (t'' \otimes t')$
 ⟨proof⟩

lemma *possible-both[simp]*: $possible\ (t \otimes t')\ x \longleftrightarrow possible\ t\ x \vee possible\ t'\ x$
 ⟨proof⟩

lemma *next-both*:

$next\ (t' \otimes t)\ x = (if\ possible\ t'\ x \wedge possible\ t\ x\ then\ next\ t'\ x \otimes t \oplus \oplus t' \otimes next\ t\ x\ else$
 $if\ possible\ t'\ x\ then\ next\ t'\ x \otimes t\ else$
 $if\ possible\ t\ x\ then\ t' \otimes next\ t\ x\ else$
 $empty)$

⟨proof⟩

lemma *Cons-both*:

$x \# xs \in paths\ (t' \otimes t) \longleftrightarrow (if\ possible\ t'\ x \wedge possible\ t\ x\ then\ xs \in paths\ (next\ t'\ x \otimes t)$
 $\vee xs \in paths\ (t' \otimes next\ t\ x)\ else$
 $if\ possible\ t'\ x\ then\ xs \in paths\ (next\ t'\ x \otimes t)\ else$
 $if\ possible\ t\ x\ then\ xs \in paths\ (t' \otimes next\ t\ x)\ else$
 $False)$

⟨proof⟩

lemma *Cons-both-possible-leftE*: $possible\ t\ x \implies xs \in paths\ (next\ t\ x \otimes t') \implies x \# xs \in paths\ (t \otimes t')$

⟨proof⟩

lemma *Cons-both-possible-rightE*: $possible\ t'\ x \implies xs \in paths\ (t \otimes next\ t'\ x) \implies x \# xs \in paths\ (t \otimes t')$

⟨proof⟩

lemma *either-both-distr[simp]*:

$t' \otimes t \oplus \oplus t' \otimes t'' = t' \otimes (t \oplus \oplus t'')$

⟨proof⟩

lemma *either-both-distr2[simp]*:

$t' \otimes t \oplus \oplus t'' \otimes t = (t' \oplus \oplus t'') \otimes t$

⟨proof⟩

lemma *next-both-repeatable[simp]*:

assumes [simp]: *repeatable* t'

assumes [simp]: *possible* $t'\ x$

shows $next\ (t' \otimes t)\ x = t' \otimes (t \oplus \oplus next\ t\ x)$

⟨proof⟩

lemma *next-both-many-calls[simp]*: $next\ (many-calls\ x \otimes t)\ x = many-calls\ x \otimes (t \oplus \oplus next\ t\ x)$

⟨proof⟩

lemma *repeatable-both-self*[simp]:
assumes [simp]: *repeatable t*
shows $t \otimes t = t$
 $\langle proof \rangle$

lemma *repeatable-both-both*[simp]:
assumes *repeatable t*
shows $t \otimes t' \otimes t = t \otimes t'$
 $\langle proof \rangle$

lemma *repeatable-both-both2*[simp]:
assumes *repeatable t*
shows $t' \otimes t \otimes t = t' \otimes t$
 $\langle proof \rangle$

lemma *repeatable-both-nxt*:
assumes *repeatable t*
assumes *possible t' x*
assumes $t' \otimes t = t'$
shows $nxt\ t'\ x \otimes t = nxt\ t'\ x$
 $\langle proof \rangle$

lemma *repeatable-both-both-nxt*:
assumes $t' \otimes t = t'$
shows $t' \otimes t'' \otimes t = t' \otimes t''$
 $\langle proof \rangle$

lemma *carrier-both*[simp]:
 $carrier\ (t \otimes t') = carrier\ t \cup carrier\ t'$
 $\langle proof \rangle$

9.1.11 Removing elements from a tree

lift-definition *without* :: $'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$
is $\lambda x\ xss. filter\ (\lambda x'. x' \neq x)\ 'xss$
 $\langle proof \rangle$

lemma *paths-withoutI*:
assumes $xs \in paths\ t$
assumes $x \notin set\ xs$
shows $xs \in paths\ (without\ x\ t)$
 $\langle proof \rangle$

lemma *carrier-without*[simp]: $carrier\ (without\ x\ t) = carrier\ t - \{x\}$
 $\langle proof \rangle$

lift-definition *ttree-restr* :: 'a set \Rightarrow 'a ttree \Rightarrow 'a ttree **is** λS xss. filter ($\lambda x'. x' \in S$) ' xss
 <proof>

lemma *filter-paths-conv-free-restr*:
 filter ($\lambda x'. x' \in S$) ' paths t = paths (ttree-restr S t) <proof>

lemma *filter-paths-conv-free-restr2*:
 filter ($\lambda x'. x' \notin S$) ' paths t = paths (ttree-restr (- S) t) <proof>

lemma *filter-paths-conv-free-without*:
 filter ($\lambda x'. x' \neq y$) ' paths t = paths (without y t) <proof>

lemma *ttree-restr-is-empty*: carrier t $\cap S = \{\}$ \Longrightarrow ttree-restr S t = empty
 <proof>

lemma *ttree-restr-noop*: carrier t $\subseteq S \Longrightarrow$ ttree-restr S t = t
 <proof>

lemma *ttree-restr-tree-restr[simp]*:
 ttree-restr S (ttree-restr S' t) = ttree-restr (S' \cap S) t
 <proof>

lemma *ttree-restr-both*:
 ttree-restr S (t $\otimes \otimes$ t') = ttree-restr S t $\otimes \otimes$ ttree-restr S t'
 <proof>

lemma *ttree-restr-nxt-subset*: $x \in S \Longrightarrow$ paths (ttree-restr S (nxt t x)) \subseteq paths (nxt (ttree-restr S t) x)
 <proof>

lemma *ttree-restr-nxt-subset2*: $x \notin S \Longrightarrow$ paths (ttree-restr S (nxt t x)) \subseteq paths (ttree-restr S t)
 <proof>

lemma *ttree-restr-possible*: $x \in S \Longrightarrow$ possible t x \Longrightarrow possible (ttree-restr S t) x
 <proof>

lemma *ttree-restr-possible2*: possible (ttree-restr S t') x \Longrightarrow $x \in S$
 <proof>

lemma *carrier-ttree-restr[simp]*:
 carrier (ttree-restr S t) = S \cap carrier t
 <proof>

9.1.12 Multiple variables, each called at most once

lift-definition *singles* :: 'a set \Rightarrow 'a ttree **is** λS . {xs. $\forall x \in S$. length (filter ($\lambda x'. x' = x$) xs) ≤ 1 }
 <proof>

lemma *possible-singles[simp]*: *possible (singles S) x*
 ⟨*proof*⟩

lemma *length-filter-mono[intro]*:
assumes $(\bigwedge x. P x \implies Q x)$
shows $\text{length (filter P xs)} \leq \text{length (filter Q xs)}$
 ⟨*proof*⟩

lemma *next-singles[simp]*: $\text{next (singles S) } x' = (\text{if } x' \in S \text{ then without } x' \text{ (singles S) else singles S})$
 ⟨*proof*⟩

lemma *carrier-singles[simp]*:
 $\text{carrier (singles S)} = \text{UNIV}$
 ⟨*proof*⟩

lemma *singles-mono*:
 $S \subseteq S' \implies \text{paths (singles S')} \subseteq \text{paths (singles S)}$
 ⟨*proof*⟩

lemma *paths-many-calls-subset*:
 $\text{paths } t \subseteq \text{paths (many-calls } x \otimes \otimes \text{ without } x \ t)$
 ⟨*proof*⟩

9.1.13 Substituting trees for every node

definition *f-next* :: $('a \Rightarrow 'a \text{ tree}) \Rightarrow 'a \text{ set} \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \text{ tree})$
where $f\text{-next } f \ T \ x = (\text{if } x \in T \text{ then } f(x := \text{empty}) \text{ else } f)$

fun *substitute'* :: $('a \Rightarrow 'a \text{ tree}) \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ tree} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**
 $\text{substitute}'\text{-Nil: } \text{substitute}' \ f \ T \ t \ [] \longleftrightarrow \text{True}$
 $| \text{substitute}'\text{-Cons: } \text{substitute}' \ f \ T \ t \ (x \# \text{xs}) \longleftrightarrow$
 $\text{possible } t \ x \wedge \text{substitute}' \ (f\text{-next } f \ T \ x) \ T \ (\text{next } t \ x \otimes \otimes \ f \ x) \ \text{xs}$

lemma *f-next-mono1*: $(\bigwedge x. \text{paths (f } x) \subseteq \text{paths (f' } x)) \implies \text{paths (f-next f T x x')} \subseteq \text{paths (f-next f' T x x')}$
 ⟨*proof*⟩

lemma *f-next-empty-set[simp]*: $f\text{-next } f \ \{\} \ x = f$ ⟨*proof*⟩

lemma *downset-substitute*: $\text{downset (Collect (substitute}' \ f \ T \ t))}$
 ⟨*proof*⟩

lift-definition *substitute* :: $('a \Rightarrow 'a \text{ tree}) \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ tree} \Rightarrow 'a \text{ tree}$
is $\lambda f \ T \ t. \text{Collect (substitute}' \ f \ T \ t)$
 ⟨*proof*⟩

lemma *elim-substitute'*[*pred-set-conv*]: $\text{substitute}' f T t xs \longleftrightarrow xs \in \text{paths} (\text{substitute } f T t)$
(*proof*)

lemmas *substitute-induct*[*case-names Nil Cons*] = *substitute'.induct*

lemmas *substitute-simps*[*simp*] = *substitute'.simps*[*unfolded elim-substitute'*]

lemma *substitute-mono2*:

assumes $\text{paths } t \subseteq \text{paths } t'$

shows $\text{paths} (\text{substitute } f T t) \subseteq \text{paths} (\text{substitute } f T t')$

(*proof*)

lemma *substitute-mono1*:

assumes $\bigwedge x. \text{paths} (f x) \subseteq \text{paths} (f' x)$

shows $\text{paths} (\text{substitute } f T t) \subseteq \text{paths} (\text{substitute } f' T t)$

(*proof*)

lemma *substitute-monoT*:

assumes $T \subseteq T'$

shows $\text{paths} (\text{substitute } f T' t) \subseteq \text{paths} (\text{substitute } f T t)$

(*proof*)

lemma *substitute-contains-arg*: $\text{paths } t \subseteq \text{paths} (\text{substitute } f T t)$

(*proof*)

lemma *possible-substitute*[*simp*]: $\text{possible} (\text{substitute } f T t) x \longleftrightarrow \text{possible } t x$

(*proof*)

lemma *nxt-substitute*[*simp*]: $\text{possible } t x \implies \text{nxt} (\text{substitute } f T t) x = \text{substitute} (f\text{-nxt } f T x)$

$T (\text{nxt } t x \otimes \otimes f x)$

(*proof*)

lemma *substitute-either*: $\text{substitute } f T (t \oplus \oplus t') = \text{substitute } f T t \oplus \oplus \text{substitute } f T t'$

(*proof*)

lemma *f-nxt-T-delete*:

assumes $f x = \text{empty}$

shows $f\text{-nxt } f (T - \{x\}) x' = f\text{-nxt } f T x'$

(*proof*)

lemma *f-nxt-empty*[*simp*]:

assumes $f x = \text{empty}$

shows $f\text{-nxt } f T x' x = \text{empty}$

(*proof*)

lemma *f-nxt-empty'[simp]*:
assumes $f\ x = \text{empty}$
shows $f\text{-nxt}\ f\ T\ x = f$
<proof>

lemma *substitute-T-delete*:
assumes $f\ x = \text{empty}$
shows $\text{substitute}\ f\ (T - \{x\})\ t = \text{substitute}\ f\ T\ t$
<proof>

lemma *substitute-only-empty*:
assumes $\text{const-on}\ f\ (\text{carrier}\ t)\ \text{empty}$
shows $\text{substitute}\ f\ T\ t = t$
<proof>

lemma *substitute-only-empty-both*: $\text{const-on}\ f\ (\text{carrier}\ t')\ \text{empty} \implies \text{substitute}\ f\ T\ (t \otimes t') = \text{substitute}\ f\ T\ t \otimes t'$
<proof>

lemma *f-nxt-upd-empty[simp]*:
 $f\text{-nxt}\ (f(x' := \text{empty}))\ T\ x = (f\text{-nxt}\ f\ T\ x)(x' := \text{empty})$
<proof>

lemma *repeatable-f-nxt-upd[simp]*:
 $\text{repeatable}\ (f\ x) \implies \text{repeatable}\ (f\text{-nxt}\ f\ T\ x'\ x)$
<proof>

lemma *substitute-remove-anyways-aux*:
assumes $\text{repeatable}\ (f\ x)$
assumes $xs \in \text{paths}\ (\text{substitute}\ f\ T\ t)$
assumes $t \otimes f\ x = t$
shows $xs \in \text{paths}\ (\text{substitute}\ (f(x := \text{empty}))\ T\ t)$
<proof>

lemma *substitute-remove-anyways*:
assumes $\text{repeatable}\ t$
assumes $f\ x = t$
shows $\text{substitute}\ f\ T\ (t \otimes t') = \text{substitute}\ (f(x := \text{empty}))\ T\ (t \otimes t')$
<proof>

lemma *carrier-f-nxt*: $\text{carrier}\ (f\text{-nxt}\ f\ T\ x\ x') \subseteq \text{carrier}\ (f\ x')$
<proof>

lemma *f-nxt-cong*: $f\ x' = f'\ x' \implies f\text{-nxt}\ f\ T\ x\ x' = f\text{-nxt}\ f'\ T\ x\ x'$
<proof>

lemma *substitute-cong'*:

assumes $xs \in \text{paths } (\text{substitute } f \ T \ t)$
assumes $\bigwedge x \ n. \ x \in A \implies \text{carrier } (f \ x) \subseteq A$
assumes $\text{carrier } t \subseteq A$
assumes $\bigwedge x. \ x \in A \implies f \ x = f' \ x$
shows $xs \in \text{paths } (\text{substitute } f' \ T \ t)$
<proof>

lemma *substitute-cong-induct*:

assumes $\bigwedge x. \ x \in A \implies \text{carrier } (f \ x) \subseteq A$
assumes $\text{carrier } t \subseteq A$
assumes $\bigwedge x. \ x \in A \implies f \ x = f' \ x$
shows $\text{substitute } f \ T \ t = \text{substitute } f' \ T \ t$
<proof>

lemma *carrier-substitute-aux*:

assumes $xs \in \text{paths } (\text{substitute } f \ T \ t)$
assumes $\text{carrier } t \subseteq A$
assumes $\bigwedge x. \ x \in A \implies \text{carrier } (f \ x) \subseteq A$
shows $\text{set } xs \subseteq A$
<proof>

lemma *carrier-substitute-below*:

assumes $\bigwedge x. \ x \in A \implies \text{carrier } (f \ x) \subseteq A$
assumes $\text{carrier } t \subseteq A$
shows $\text{carrier } (\text{substitute } f \ T \ t) \subseteq A$
<proof>

lemma *f-nxt-eq-empty-iff*:

$f\text{-nxt } f \ T \ x \ x' = \text{empty} \iff f \ x' = \text{empty} \vee (x' = x \wedge x \in T)$
<proof>

lemma *substitute-T-cong'*:

assumes $xs \in \text{paths } (\text{substitute } f \ T \ t)$
assumes $\bigwedge x. \ (x \in T \iff x \in T') \vee f \ x = \text{empty}$
shows $xs \in \text{paths } (\text{substitute } f \ T' \ t)$
<proof>

lemma *substitute-cong-T*:

assumes $\bigwedge x. \ (x \in T \iff x \in T') \vee f \ x = \text{empty}$
shows $\text{substitute } f \ T = \text{substitute } f \ T'$
<proof>

lemma *carrier-substitute1*: $\text{carrier } t \subseteq \text{carrier } (\text{substitute } f \ T \ t)$

<proof>

lemma *substitute-cong*:

assumes $\bigwedge x. x \in \text{carrier } (\text{substitute } f \ T \ t) \implies f \ x = f' \ x$
shows $\text{substitute } f \ T \ t = \text{substitute } f' \ T \ t$
 $\langle \text{proof} \rangle$

lemma *substitute-substitute*:

assumes $\bigwedge x. \text{const-on } f' \ (\text{carrier } (f \ x)) \ \text{empty}$
shows $\text{substitute } f \ T \ (\text{substitute } f' \ T \ t) = \text{substitute } (\lambda x. f \ x \otimes f' \ x) \ T \ t$
 $\langle \text{proof} \rangle$

lemma *tree-rest-substitute*:

assumes $\bigwedge x. \text{carrier } (f \ x) \cap S = \{\}$
shows $\text{tree-restr } S \ (\text{substitute } f \ T \ t) = \text{tree-restr } S \ t$
 $\langle \text{proof} \rangle$

An alternative characterization of substitution

inductive *substitute''* :: $(\ 'a \Rightarrow \ 'a \ \text{tree}) \Rightarrow \ 'a \ \text{set} \Rightarrow \ 'a \ \text{list} \Rightarrow \ 'a \ \text{list} \Rightarrow \ \text{bool}$ **where**
substitute''-Nil: $\text{substitute'' } f \ T \ [] \ []$
| *substitute''-Cons*:
 $zs \in \text{paths } (f \ x) \implies xs' \in \text{interleave } xs \ zs \implies \text{substitute'' } (f\text{-next } f \ T \ x) \ T \ xs' \ ys$
 $\implies \text{substitute'' } f \ T \ (x\#xs) \ (x\#ys)$
inductive-cases *substitute''-NilE*[*elim*]: $\text{substitute'' } f \ T \ xs \ [] \ \text{substitute'' } f \ T \ [] \ xs$
inductive-cases *substitute''-ConsE*[*elim*]: $\text{substitute'' } f \ T \ (x\#xs) \ ys$

lemma *substitute-substitute''*:

$xs \in \text{paths } (\text{substitute } f \ T \ t) \iff (\exists xs' \in \text{paths } t. \text{substitute'' } f \ T \ xs' \ xs)$
 $\langle \text{proof} \rangle$

lemma *paths-substitute-substitute''*:

$\text{paths } (\text{substitute } f \ T \ t) = \bigcup ((\lambda xs. \text{Collect } (\text{substitute'' } f \ T \ xs)) \ ` \ \text{paths } t)$
 $\langle \text{proof} \rangle$

lemma *tree-rest-substitute2*:

assumes $\bigwedge x. \text{carrier } (f \ x) \subseteq S$
assumes $\text{const-on } f \ (-S) \ \text{empty}$
shows $\text{tree-restr } S \ (\text{substitute } f \ T \ t) = \text{substitute } f \ T \ (\text{tree-restr } S \ t)$
 $\langle \text{proof} \rangle$

end

9.2 TTree-HOLCF

theory *TTree-HOLCF*

imports *TTree Launchbury.HOLCF-Utills Set-Cpo Launchbury.HOLCF-Join-Classes*

begin

instantiation *tree* :: $(\ \text{type}) \ \text{below}$

begin

lift-definition *below-ttree* :: 'a ttree \Rightarrow 'a ttree \Rightarrow bool **is** (\sqsubseteq)
instance *proof*
end

lemma *paths-mono*: $t \sqsubseteq t' \Longrightarrow \text{paths } t \sqsubseteq \text{paths } t'$
proof

lemma *paths-mono-iff*: $\text{paths } t \sqsubseteq \text{paths } t' \longleftrightarrow t \sqsubseteq t'$
proof

lemma *tree-belowI*: $(\bigwedge xs. xs \in \text{paths } t \Longrightarrow xs \in \text{paths } t') \Longrightarrow t \sqsubseteq t'$
proof

lemma *paths-belowI*: $(\bigwedge x xs. x\#xs \in \text{paths } t \Longrightarrow x\#xs \in \text{paths } t') \Longrightarrow t \sqsubseteq t'$
proof

instance *tree* :: (type) po
proof

lemma *is-lub-ttree*:
 $S \ll\mid \text{Either } S$
proof

lemma *lub-is-either*: $\text{lub } S = \text{Either } S$
proof

instance *tree* :: (type) cpo
proof

lemma *minimal-ttree[simp, intro!]*: $\text{empty} \sqsubseteq S$
proof

instance *tree* :: (type) pcpo
proof

lemma *empty-is-bottom*: $\text{empty} = \perp$
proof

lemma *carrier-bottom[simp]*: $\text{carrier } \perp = \{\}$
proof

lemma *below-anything[simp]*:
 $t \sqsubseteq \text{anything}$
proof

lemma *carrier-mono*: $t \sqsubseteq t' \Longrightarrow \text{carrier } t \subseteq \text{carrier } t'$
proof

lemma *nxt-mono*: $t \sqsubseteq t' \Longrightarrow \text{nxt } t \ x \sqsubseteq \text{nxt } t' \ x$

<proof>

lemma *either-above-arg1*: $t \sqsubseteq t \oplus \oplus t'$

<proof>

lemma *either-above-arg2*: $t' \sqsubseteq t \oplus \oplus t'$

<proof>

lemma *either-belowI*: $t \sqsubseteq t'' \implies t' \sqsubseteq t'' \implies t \oplus \oplus t' \sqsubseteq t''$

<proof>

lemma *both-above-arg1*: $t \sqsubseteq t \otimes \otimes t'$

<proof>

lemma *both-above-arg2*: $t' \sqsubseteq t \otimes \otimes t'$

<proof>

lemma *both-mono1'*:

$t \sqsubseteq t' \implies t \otimes \otimes t'' \sqsubseteq t' \otimes \otimes t''$

<proof>

lemma *both-mono2'*:

$t \sqsubseteq t' \implies t'' \otimes \otimes t \sqsubseteq t'' \otimes \otimes t'$

<proof>

lemma *next-both-left*:

possible $t x \implies \text{next } t x \otimes \otimes t' \sqsubseteq \text{next } (t \otimes \otimes t') x$

<proof>

lemma *next-both-right*:

possible $t' x \implies t \otimes \otimes \text{next } t' x \sqsubseteq \text{next } (t \otimes \otimes t') x$

<proof>

lemma *substitute-mono1'*: $f \sqsubseteq f' \implies \text{substitute } f T t \sqsubseteq \text{substitute } f' T t$

<proof>

lemma *substitute-mono2'*: $t \sqsubseteq t' \implies \text{substitute } f T t \sqsubseteq \text{substitute } f T t'$

<proof>

lemma *substitute-above-arg*: $t \sqsubseteq \text{substitute } f T t$

<proof>

lemma *tree-contI*:

assumes $\bigwedge S. f (Either S) = Either (f ' S)$

shows *cont* f

<proof>

lemma *tree-contI2*:

assumes $\bigwedge x. \text{paths } (f x) = \bigcup (t \text{ ' paths } x)$

assumes $\square \in t \square$

shows *cont f*

<proof>

lemma *cont-paths*[*THEN cont-compose, cont2cont, simp*]:

cont paths

<proof>

lemma *tree-contI3*:

assumes *cont* $(\lambda x. \text{paths } (f x))$

shows *cont f*

<proof>

lemma *cont-substitute*[*THEN cont-compose, cont2cont, simp*]:

cont (substitute f T)

<proof>

lemma *cont-both1*:

cont $(\lambda x. \text{both } x y)$

<proof>

lemma *cont-both2*:

cont $(\lambda x. \text{both } y x)$

<proof>

lemma *cont-both*[*cont2cont,simp*]: *cont f* \implies *cont g* \implies *cont* $(\lambda x. f x \otimes \otimes g x)$

<proof>

lemma *cont-intersect1*:

cont $(\lambda x. \text{intersect } x y)$

<proof>

lemma *cont-intersect2*:

cont $(\lambda x. \text{intersect } y x)$

<proof>

lemma *cont-intersect*[*cont2cont,simp*]: *cont f* \implies *cont g* \implies *cont* $(\lambda x. f x \cap \cap g x)$

<proof>

lemma *cont-without*[*THEN cont-compose, cont2cont,simp*]: *cont* $(\text{without } x)$

<proof>

lemma *paths-many-calls-subset*:

$t \sqsubseteq \text{many-calls } x \otimes \otimes \text{without } x t$

$\langle \text{proof} \rangle$

lemma *single-below*:

$[x] \in \text{paths } t \implies \text{single } x \sqsubseteq t \langle \text{proof} \rangle$

lemma *cont-ttree-restr*[*THEN cont-compose, cont2cont,simp*]: *cont (ttree-restr S)*

$\langle \text{proof} \rangle$

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

lemma *range-filter*[*simp*]: *range (filter P) = {xs. set xs \subseteq Collect P}*

$\langle \text{proof} \rangle$

lemma *tree-restr-anything-cont*[*THEN cont-compose, simp, cont2cont*]:

cont ($\lambda S. \text{tree-restr } S \text{ anything}$)

$\langle \text{proof} \rangle$

instance *tree :: (type) Finite-Join-cpo*

$\langle \text{proof} \rangle$

lemma *tree-join-is-either*:

$t \sqcup t' = t \oplus \oplus t'$

$\langle \text{proof} \rangle$

lemma *tree-join-transfer*[*transfer-rule*]: *rel-fun (pcr-ttree (=)) (rel-fun (pcr-ttree (=)) (pcr-ttree (=))) (\cup) (\sqcup)*

$\langle \text{proof} \rangle$

lemma *tree-restr-join*[*simp*]:

tree-restr S (t \sqcup t') = tree-restr S t \sqcup tree-restr S t'

$\langle \text{proof} \rangle$

lemma *nxt-singles-below-singles*:

nxt (singles S) x \sqsubseteq singles S

$\langle \text{proof} \rangle$

lemma *in-carrier-fup*[*simp*]:

$x' \in \text{carrier } (fup \cdot f \cdot u) \iff (\exists u'. u = up \cdot u' \wedge x' \in \text{carrier } (f \cdot u'))$

$\langle \text{proof} \rangle$

end

10 Trace Tree Cardinality Analysis

10.1 AnalBinds

theory *AnalBinds*


```

imports Launchbury.Terms Launchbury.HOLCF-Utills Launchbury.Env
begin

locale ExpAnalysis =
  fixes exp :: exp ⇒ 'a::cpo → 'b::pcpo
begin

fun AnalBinds :: heap ⇒ (var ⇒ 'a⊥) → (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 = ⊥ ⟨proof⟩

lemma AnalBinds-Cons[simp]:
  AnalBinds ((x,e)#Γ)·ae = (AnalBinds Γ·ae)(x := fup·(exp e)·(ae x))
  ⟨proof⟩

lemmas AnalBinds.simps[simp del]

lemma AnalBinds-not-there: x ∉ domA Γ ⇒ (AnalBinds Γ·ae) x = ⊥
  ⟨proof⟩

lemma AnalBinds-cong:
  assumes ae f |' domA Γ = ae' f |' domA Γ
  shows AnalBinds Γ·ae = AnalBinds Γ·ae'
  ⟨proof⟩

lemma AnalBinds-lookup: (AnalBinds Γ·ae) x = (case map-of Γ x of Some e ⇒ fup·(exp e)·(ae
x) | None ⇒ ⊥)
  ⟨proof⟩

lemma AnalBinds-delete-bot: ae x = ⊥ ⇒ AnalBinds (delete x Γ)·ae = AnalBinds Γ·ae
  ⟨proof⟩

lemma AnalBinds-delete-below: AnalBinds (delete x Γ)·ae ⊆ AnalBinds Γ·ae
  ⟨proof⟩

lemma AnalBinds-delete-lookup[simp]: (AnalBinds (delete x Γ)·ae) x = ⊥
  ⟨proof⟩

lemma AnalBinds-delete-to-fun-upd: AnalBinds (delete x Γ)·ae = (AnalBinds Γ·ae)(x := ⊥)
  ⟨proof⟩

lemma edom-AnalBinds: edom (AnalBinds Γ·ae) ⊆ domA Γ ∩ edom ae
  ⟨proof⟩

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<proof>
  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 []  $\longleftrightarrow$  True
| no-call-in-path x (y#xs)  $\longleftrightarrow$  y  $\neq$  x  $\wedge$  no-call-in-path x xs
```

```
fun one-call-in-path where
  one-call-in-path x []  $\longleftrightarrow$  True
| one-call-in-path x (y#xs)  $\longleftrightarrow$  (if x = y then no-call-in-path x xs else one-call-in-path x xs)
```

```
lemma no-call-in-path-set-conv:
  no-call-in-path x p  $\longleftrightarrow$  x  $\notin$  set p
  <proof>
```

```
lemma one-call-in-path-filter-conv:
  one-call-in-path x p  $\longleftrightarrow$  length (filter ( $\lambda$  x'. x' = x) p)  $\leq$  1
  <proof>
```

```
lemma no-call-in-tail: no-call-in-path x (tl p)  $\longleftrightarrow$  (no-call-in-path x p  $\vee$  one-call-in-path x p  $\wedge$ 
hd p = x)
  <proof>
```

```
lemma no-imp-one: no-call-in-path x p  $\implies$  one-call-in-path x p
  <proof>
```

```
lemma one-imp-one-tail: one-call-in-path x p  $\implies$  one-call-in-path x (tl p)
  <proof>
```

```
lemma more-than-one-setD:
```

$\neg \text{one-call-in-path } x \ p \implies x \in \text{set } p$
 ⟨proof⟩

lemma *no-call-in-path[eqvt]*: $\text{no-call-in-path } p \ x \implies \text{no-call-in-path } (\pi \cdot p) \ (\pi \cdot x)$
 ⟨proof⟩

lemma *one-call-in-path[eqvt]*: $\text{one-call-in-path } p \ x \implies \text{one-call-in-path } (\pi \cdot p) \ (\pi \cdot x)$
 ⟨proof⟩

definition *pathCard* :: $\text{var list} \Rightarrow (\text{var} \Rightarrow \text{two})$
where *pathCard* $p \ x = (\text{if } \text{no-call-in-path } x \ p \ \text{then } \text{none} \ \text{else } (\text{if } \text{one-call-in-path } x \ p \ \text{then } \text{once} \ \text{else } \text{many}))$

lemma *pathCard-Nil[simp]*: $\text{pathCard } [] = \perp$
 ⟨proof⟩

lemma *pathCard-Cons[simp]*: $\text{pathCard } (x\#xs) \ x = \text{two-add} \cdot \text{once} \cdot (\text{pathCard } xs \ x)$
 ⟨proof⟩

lemma *pathCard-Cons-other[simp]*: $x' \neq x \implies \text{pathCard } (x\#xs) \ x' = \text{pathCard } xs \ x'$
 ⟨proof⟩

lemma *no-call-in-path-filter[simp]*: $\text{no-call-in-path } x \ [x \leftarrow xs \ . \ x \in S] \longleftrightarrow \text{no-call-in-path } x \ xs \ \vee \ x \notin S$
 ⟨proof⟩

lemma *one-call-in-path-filter[simp]*: $\text{one-call-in-path } x \ [x \leftarrow xs \ . \ x \in S] \longleftrightarrow \text{one-call-in-path } x \ xs \ \vee \ x \notin S$
 ⟨proof⟩

definition *pathsCard* :: $\text{var list set} \Rightarrow (\text{var} \Rightarrow \text{two})$
where *pathsCard* $ps \ x = (\text{if } (\forall p \in ps. \text{no-call-in-path } x \ p) \ \text{then } \text{none} \ \text{else } (\text{if } (\forall p \in ps. \text{one-call-in-path } x \ p) \ \text{then } \text{once} \ \text{else } \text{many}))$

lemma *paths-Card-above*:
 $p \in ps \implies \text{pathCard } p \sqsubseteq \text{pathsCard } ps$
 ⟨proof⟩

lemma *pathsCard-below*:
assumes $\bigwedge p. p \in ps \implies \text{pathCard } p \sqsubseteq ce$
shows $\text{pathsCard } ps \sqsubseteq ce$
 ⟨proof⟩

lemma *pathsCard-mono*:
 $ps \subseteq ps' \implies \text{pathsCard } ps \sqsubseteq \text{pathsCard } ps'$
 ⟨proof⟩

lemmas $\text{pathsCard-mono}' = \text{pathsCard-mono}[\text{folded below-set-def}]$

lemma *record-call-pathsCard*:

$pathsCard (\{ tl\ p \mid p \cdot p \in fs \wedge hd\ p = x \}) \sqsubseteq record\text{-}call\ x \cdot (pathsCard\ fs)$
 $\langle proof \rangle$

lemma *pathCards-noneD*:

$pathsCard\ ps\ x = none \implies x \notin \bigcup (set\ 'ps)$
 $\langle proof \rangle$

lemma *cont-pathsCard*[*THEN cont-compose, cont2cont, simp*]:

$cont\ pathsCard$
 $\langle proof \rangle$

lemma *pathsCard-eqvt*[*eqvt*]: $\pi \cdot pathsCard\ ps\ x = pathsCard (\pi \cdot ps) (\pi \cdot x)$

$\langle proof \rangle$

lemma *edom-pathsCard*[*simp*]: $edom (pathsCard\ ps) = \bigcup (set\ 'ps)$

$\langle proof \rangle$

lemma *env-restr-pathsCard*[*simp*]: $pathsCard\ ps\ f|'S = pathsCard (filter (\lambda x. x \in S) 'ps)$

$\langle proof \rangle$

end

10.4 TTreeAnalysisSpec

theory *TTreeAnalysisSpec*

imports *TTreeAnalysisSig ArityAnalysisSpec Cardinality-Domain-Lists*

begin

locale *TTreeAnalysisCarrier* = *TTreeAnalysis* + *EdomArityAnalysis* +

assumes *carrier-Fexp*: $carrier (Texp\ e \cdot a) = edom (Aexp\ e \cdot a)$

locale *TTreeAnalysisSafe* = *TTreeAnalysisCarrier* +

assumes *Texp-App*: $many\text{-}calls\ x \otimes \otimes (Texp\ e) \cdot (inc \cdot a) \sqsubseteq Texp (App\ e\ x) \cdot a$

assumes *Texp-Lam*: $without\ y (Texp\ e \cdot (pred \cdot n)) \sqsubseteq Texp (Lam\ [y].\ e) \cdot n$

assumes *Texp-subst*: $Texp (e[y := x]) \cdot a \sqsubseteq many\text{-}calls\ x \otimes \otimes without\ y ((Texp\ e) \cdot a)$

assumes *Texp-Var*: $single\ v \sqsubseteq Texp (Var\ v) \cdot a$

assumes *Fun-repeatable*: $isVal\ e \implies repeatable (Texp\ e \cdot 0)$

assumes *Texp-IfThenElse*: $Texp\ scrut \cdot 0 \otimes \otimes (Texp\ e1 \cdot a \oplus \oplus Texp\ e2 \cdot a) \sqsubseteq Texp (scrut\ ?\ e1 : e2) \cdot a$

locale *TTreeAnalysisCardinalityHeap* =

TTreeAnalysisSafe + *ArityAnalysisLetSafe* +

fixes *Theap* :: $heap \Rightarrow exp \Rightarrow Arity \rightarrow var\ ttree$

assumes *carrier-Fheap*: $carrier (Theap\ \Gamma\ e \cdot a) = edom (Aheap\ \Gamma\ e \cdot a)$

assumes *Theap-thunk*: $x \in thunks\ \Gamma \implies p \in paths (Theap\ \Gamma\ e \cdot a) \implies \neg one\text{-}call\text{-}in\text{-}path\ x\ p \implies (Aheap\ \Gamma\ e \cdot a)\ x = up \cdot 0$

```

assumes Theap-substitute: ttree-restr (domA  $\Delta$ ) (substitute (FBinds  $\Delta$ .(Aheap  $\Delta$  e·a)) (thunks
 $\Delta$ ) (Texp e·a))  $\sqsubseteq$  Theap  $\Delta$  e·a
assumes Texp-Let: ttree-restr ( $-$  domA  $\Delta$ ) (substitute (FBinds  $\Delta$ .(Aheap  $\Delta$  e·a)) (thunks
 $\Delta$ ) (Texp e·a))  $\sqsubseteq$  Texp (Terms.Let  $\Delta$  e·a

```

end

10.5 TTreeImplCardinality

```

theory TTreeImplCardinality

```

```

imports TTreeAnalysisSig CardinalityAnalysisSig Cardinality-Domain-Lists

```

```

begin

```

```

context TTreeAnalysis

```

```

begin

```

```

fun unstack :: stack  $\Rightarrow$  exp  $\Rightarrow$  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 :: Ariety list  $\Rightarrow$  stack  $\Rightarrow$  var tree

```

```

where Fstack - [] =  $\perp$ 
| Fstack (a#as) (Alts e1 e2 # S) = (Texp e1·a  $\oplus\oplus$  Texp e2·a)  $\otimes\otimes$  Fstack as S
| Fstack as (Arg x # S) = many-calls x  $\otimes\otimes$  Fstack as S
| Fstack as ( $-$  # S) = Fstack as S

```

```

fun prognosis :: AEnv  $\Rightarrow$  Ariety list  $\Rightarrow$  Ariety  $\Rightarrow$  conf  $\Rightarrow$  var  $\Rightarrow$  two

```

```

where prognosis ae as a ( $\Gamma$ , e, S) = pathsCard (paths (substitute (FBinds  $\Gamma$ ·ae) (thunks  $\Gamma$ )
(Texp e·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))  $\sqsubseteq$  record-call x·(pathsCard (paths f))

```

<proof>

lemma *pathsCards-none*: $\text{pathsCard } (\text{paths } t) \ x = \text{none} \implies x \notin \text{carrier } t$
<proof>

lemma *const-on-edom-disj*: $\text{const-on } f \ S \ \text{empty} \iff \text{edom } f \cap S = \{\}$
<proof>

context *TTreeAnalysisCarrier*

begin

lemma *carrier-Fstack*: $\text{carrier } (F\text{stack as } S) \subseteq \text{fv } S$
<proof>

lemma *carrier-FBinds*: $\text{carrier } ((F\text{Binds } \Gamma \cdot a e) \ x) \subseteq \text{fv } \Gamma$
<proof>

end

context *TTreeAnalysisSafe*

begin

sublocale *CardinalityPrognosisShape prognosis*
<proof>

sublocale *CardinalityPrognosisApp prognosis*
<proof>

sublocale *CardinalityPrognosisLam prognosis*
<proof>

sublocale *CardinalityPrognosisVar prognosis*
<proof>

sublocale *CardinalityPrognosisIfThenElse prognosis*
<proof>

end

context *TTreeAnalysisCardinalityHeap*

begin

definition *cHeap where*
 $c\text{Heap } \Gamma \ e = (\Lambda \ a. \ \text{pathsCard } (\text{paths } (T\text{heap } \Gamma \ e \cdot a)))$

lemma *cHeap-simp*: $(c\text{Heap } \Gamma \ e) \cdot a = \text{pathsCard } (\text{paths } (T\text{heap } \Gamma \ e \cdot a))$
<proof>

sublocale *CardinalityHeap cHeap**<proof>*

sublocale *CardinalityHeapSafe cHeap Ahead*

<proof>

sublocale *CardinalityPrognosisEdom prognosis*

<proof>

sublocale *CardinalityPrognosisLet prognosis cHeap*

<proof>

sublocale *CardinalityPrognosisSafe prognosis cHeap Ahead Aexp <proof>*
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*

<proof>

setup-lifting *type-definition-CoCalls*

instantiation *CoCalls :: po*

begin

lift-definition *below-CoCalls :: CoCalls ⇒ CoCalls ⇒ bool is (⊆)<proof>*

instance

<proof>

end

lift-definition *coCallsLub :: CoCalls set ⇒ CoCalls is λ S. ⋃ S*

<proof>

lemma *coCallsLub-is-lub: S <<| coCallsLub S*

<proof>

instance *CoCalls :: cpo*

<proof>

lemma *ccLubTransfer[transfer-rule]: (rel-set pcr-CoCalls ==> pcr-CoCalls) Union lub*

<proof>

lift-definition *is-cc-lub* :: *CoCalls set* \Rightarrow *CoCalls* \Rightarrow *bool* **is** ($\lambda S x . x = \text{Union } S$) \langle *proof* \rangle

lemma *ccis-lubTransfer*[*transfer-rule*]: (*rel-set pcr-CoCalls* \implies *pcr-CoCalls* \implies $(=)$) ($\lambda S x . x = \text{Union } S$) (\ll)
 \langle *proof* \rangle

lift-definition *coCallsJoin* :: *CoCalls* \Rightarrow *CoCalls* \Rightarrow *CoCalls* **is** (\cup)
 \langle *proof* \rangle

lemma *ccJoinTransfer*[*transfer-rule*]: (*pcr-CoCalls* \implies *pcr-CoCalls* \implies *pcr-CoCalls*)
(\cup) (\sqcup)
 \langle *proof* \rangle

lift-definition *ccEmpty* :: *CoCalls* **is** $\{\}$ \langle *proof* \rangle

lemma *ccEmpty-below*[*simp*]: *ccEmpty* \sqsubseteq *G*
 \langle *proof* \rangle

instance *CoCalls* :: *pcpo*
 \langle *proof* \rangle

lemma *ccBotTransfer*[*transfer-rule*]: *pcr-CoCalls* $\{\}$ \perp
 \langle *proof* \rangle

lemma *cc-lub-below-iff*:
fixes *G* :: *CoCalls*
shows *lub X* \sqsubseteq *G* \iff ($\forall G' \in X . G' \sqsubseteq G$)
 \langle *proof* \rangle

lift-definition *ccField* :: *CoCalls* \Rightarrow *var set* **is** *Field* \langle *proof* \rangle

lemma *ccField-nil*[*simp*]: *ccField* $\perp = \{\}$
 \langle *proof* \rangle

lift-definition
inCC :: *var* \Rightarrow *var* \Rightarrow *CoCalls* \Rightarrow *bool* ($\text{---} \in \text{---} [1000, 1000, 900] 900$)
is $\lambda x y s . (x, y) \in s$ \langle *proof* \rangle

abbreviation
notInCC :: *var* \Rightarrow *var* \Rightarrow *CoCalls* \Rightarrow *bool* ($\text{---} \notin \text{---} [1000, 1000, 900] 900$)
where $x \text{---} y \notin S \equiv \neg x \text{---} y \in S$

lemma *notInCC-bot*[*simp*]: $x \text{---} y \in \perp \iff \text{False}$
 \langle *proof* \rangle

lemma *below-CoCallsI*:
($\bigwedge x y . x \text{---} y \in G \implies x \text{---} y \in G'$) $\implies G \sqsubseteq G'$
 \langle *proof* \rangle

lemma *CoCalls-eqI*:

$$(\bigwedge x y. x \dashv\vdash y \in G \longleftrightarrow x \dashv\vdash y \in G') \implies G = G'$$

<proof>

lemma *in-join[simp]*:

$$x \dashv\vdash y \in (G \sqcup G') \longleftrightarrow x \dashv\vdash y \in G \vee x \dashv\vdash y \in G'$$

<proof>

lemma *in-lub[simp]*: $x \dashv\vdash y \in (\text{lub } S) \longleftrightarrow (\exists G \in S. x \dashv\vdash y \in G)$

<proof>

lemma *in-CoCallsLubI*:

$$x \dashv\vdash y \in G \implies G \in S \implies x \dashv\vdash y \in \text{lub } S$$

<proof>

lemma *adm-not-in[simp]*:

assumes *cont t*

shows *adm* $(\lambda a. x \dashv\vdash y \notin t a)$

<proof>

lift-definition *cc-delete* :: *var* \Rightarrow *CoCalls* \Rightarrow *CoCalls*

is $\lambda z. \text{Set.filter } (\lambda (x,y). x \neq z \wedge y \neq z)$

<proof>

lemma *ccField-cc-delete*: $\text{ccField } (\text{cc-delete } x S) \subseteq \text{ccField } S - \{x\}$

<proof>

lift-definition *ccProd* :: *var set* \Rightarrow *var set* \Rightarrow *CoCalls* (**infixr** $G \times 90$)

is $\lambda S1 S2. S1 \times S2 \cup S2 \times S1$

<proof>

lemma *ccProd-empty[simp]*: $\{\} G \times S = \perp$ *<proof>*

lemma *ccProd-empty'[simp]*: $S G \times \{\} = \perp$ *<proof>*

lemma *ccProd-union2[simp]*: $S G \times (S' \cup S'') = S G \times S' \sqcup S G \times S''$

<proof>

lemma *ccProd-Union2[simp]*: $S G \times \bigcup S' = (\bigsqcup_{X \in S'} \text{ccProd } S X)$

<proof>

lemma *ccProd-Union2'[simp]*: $S G \times (\bigcup_{X \in S'} f X) = (\bigsqcup_{X \in S'} \text{ccProd } S (f X))$

<proof>

lemma *in-ccProd[simp]*: $x \dashv\vdash y \in (S G \times S') = (x \in S \wedge y \in S' \vee x \in S' \wedge y \in S)$

<proof>

lemma *ccProd-union1[simp]*: $(S' \cup S'') G \times S = S' G \times S \sqcup S'' G \times S$

<proof>

lemma *ccProd-insert2*: $S \times G \times \text{insert } x \ S' = S \times G \times \{x\} \sqcup S \times G \times S'$
<proof>

lemma *ccProd-insert1*: $\text{insert } x \ S' \times G \times S = \{x\} \times G \times S \sqcup S' \times G \times S$
<proof>

lemma *ccProd-mono1*: $S' \subseteq S'' \implies S' \times G \times S \subseteq S'' \times G \times S$
<proof>

lemma *ccProd-mono2*: $S' \subseteq S'' \implies S \times G \times S' \subseteq S \times G \times S''$
<proof>

lemma *ccProd-mono*: $S \subseteq S' \implies T \subseteq T' \implies S \times G \times T \subseteq S' \times G \times T'$
<proof>

lemma *ccProd-comm*: $S \times G \times S' = S' \times G \times S$ *<proof>*

lemma *ccProd-belowI*:
 $(\bigwedge x \ y. x \in S \implies y \in S' \implies x \dashv\vdash y \in G) \implies S \times G \times S' \subseteq G$
<proof>

lift-definition *cc-restr* :: *var set* \Rightarrow *CoCalls* \Rightarrow *CoCalls*
is $\lambda S. \text{Set.filter } (\lambda (x,y) . x \in S \wedge y \in S)$
<proof>

abbreviation *cc-restr-sym* (**infixl** $G|' \ 110$) **where** $G \ G|' \ S \equiv \text{cc-restr } S \ G$

lemma *elem-cc-restr[simp]*: $x \dashv\vdash y \in (G \ G|' \ S) = (x \dashv\vdash y \in G \wedge x \in S \wedge y \in S)$
<proof>

lemma *ccField-cc-restr*: $\text{ccField } (G \ G|' \ S) \subseteq \text{ccField } G \cap S$
<proof>

lemma *cc-restr-empty*: $\text{ccField } G \subseteq - \ S \implies G \ G|' \ S = \perp$
<proof>

lemma *cc-restr-empty-set[simp]*: $\text{cc-restr } \{\} \ G = \perp$
<proof>

lemma *cc-restr-noop[simp]*: $\text{ccField } G \subseteq S \implies \text{cc-restr } S \ G = G$
<proof>

lemma *cc-restr-bot[simp]*: $\text{cc-restr } S \ \perp = \perp$
<proof>

lemma *ccRestr-ccDelete[simp]*: $\text{cc-restr } (-\{x\}) \ G = \text{cc-delete } x \ G$

$\langle proof \rangle$

lemma *cc-restr-join[simp]*:

$$cc\text{-restr } S (G \sqcup G') = cc\text{-restr } S G \sqcup cc\text{-restr } S G'$$

$\langle proof \rangle$

lemma *cont-cc-restr*: $cont (cc\text{-restr } S)$

$\langle proof \rangle$

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

lemma *cc-restr-mono1*:

$$S \subseteq S' \implies cc\text{-restr } S G \sqsubseteq cc\text{-restr } S' G \langle proof \rangle$$

lemma *cc-restr-mono2*:

$$G \sqsubseteq G' \implies cc\text{-restr } S G \sqsubseteq cc\text{-restr } S G' \langle proof \rangle$$

lemma *cc-restr-below-arg*:

$$cc\text{-restr } S G \sqsubseteq G \langle proof \rangle$$

lemma *cc-restr-lub[simp]*:

$$cc\text{-restr } S (\text{lub } X) = (\bigsqcup_{G \in X} cc\text{-restr } S G) \langle proof \rangle$$

lemma *elem-to-ccField*: $x \text{--} y \in G \implies x \in cc\text{Field } G \wedge y \in cc\text{Field } G$

$\langle proof \rangle$

lemma *ccField-to-elem*: $x \in cc\text{Field } G \implies \exists y. x \text{--} y \in G$

$\langle proof \rangle$

lemma *cc-restr-intersect*: $cc\text{Field } G \cap ((S - S') \cup (S' - S)) = \{\} \implies cc\text{-restr } S G = cc\text{-restr } S' G$

$\langle proof \rangle$

lemma *cc-restr-cc-restr[simp]*: $cc\text{-restr } S (cc\text{-restr } S' G) = cc\text{-restr } (S \cap S') G$

$\langle proof \rangle$

lemma *cc-restr-twist*: $cc\text{-restr } S (cc\text{-restr } S' G) = cc\text{-restr } S' (cc\text{-restr } S G)$

$\langle proof \rangle$

lemma *cc-restr-cc-delete-twist*: $cc\text{-restr } x (cc\text{-delete } S G) = cc\text{-delete } S (cc\text{-restr } x G)$

$\langle proof \rangle$

lemma *cc-restr-ccProd[simp]*:

$$cc\text{-restr } S (cc\text{Prod } S_1 S_2) = cc\text{Prod } (S_1 \cap S) (S_2 \cap S)$$

$\langle proof \rangle$

lemma *ccProd-below-cc-restr*:

$$cc\text{Prod } S S' \sqsubseteq cc\text{-restr } S'' G \iff cc\text{Prod } S S' \sqsubseteq G \wedge (S = \{\} \vee S' = \{\} \vee S \subseteq S'' \wedge S' \subseteq S'')$$

<proof>

lemma *cc-restr-eq-subset*: $S \subseteq S' \implies \text{cc-restr } S' G = \text{cc-restr } S' G2 \implies \text{cc-restr } S G = \text{cc-restr } S G2$

<proof>

definition *ccSquare* (-² [80] 80)

where $S^2 = \text{ccProd } S S$

lemma *ccField-ccSquare[simp]*: $\text{ccField } (S^2) = S$

<proof>

lemma *below-ccSquare[iff]*: $(G \sqsubseteq S^2) = (\text{ccField } G \subseteq S)$

<proof>

lemma *cc-restr-ccSquare[simp]*: $(S^2) G \upharpoonright S = (S' \cap S)^2$

<proof>

lemma *ccSquare-empty[simp]*: $\{\}^2 = \perp$

<proof>

lift-definition *ccNeighbors* :: $\text{var} \Rightarrow \text{CoCalls} \Rightarrow \text{var set}$

is $\lambda x G. \{y \mid (y,x) \in G \vee (x,y) \in G\}$ *<proof>*

lemma *ccNeighbors-bot[simp]*: $\text{ccNeighbors } x \perp = \{\}$ *<proof>*

lemma *cont-ccProd1*:

$\text{cont } (\lambda S. \text{ccProd } S S')$

<proof>

lemma *cont-ccProd2*:

$\text{cont } (\lambda S'. \text{ccProd } S S')$

<proof>

lemmas *cont-compose2*[OF *cont-ccProd1 cont-ccProd2, simp, cont2cont*]

lemma *cont-ccNeighbors*[THEN *cont-compose, cont2cont, simp*]:

$\text{cont } (\lambda y. \text{ccNeighbors } x y)$

<proof>

lemma *ccNeighbors-join[simp]*: $\text{ccNeighbors } x (G \sqcup G') = \text{ccNeighbors } x G \cup \text{ccNeighbors } x G'$

<proof>

lemma *ccNeighbors-ccProd*:

$\text{ccNeighbors } x (\text{ccProd } S S') = (\text{if } x \in S \text{ then } S' \text{ else } \{\}) \cup (\text{if } x \in S' \text{ then } S \text{ else } \{\})$

<proof>

lemma *ccNeighbors-ccSquare*:

$$ccNeighbors\ x\ (ccSquare\ S) = (if\ x \in S\ then\ S\ else\ \{\})$$

<proof>

lemma *ccNeighbors-cc-restr[simp]*:

$$ccNeighbors\ x\ (cc-restr\ S\ G) = (if\ x \in S\ then\ ccNeighbors\ x\ G \cap S\ else\ \{\})$$

<proof>

lemma *ccNeighbors-mono*:

$$G \sqsubseteq G' \implies ccNeighbors\ x\ G \subseteq ccNeighbors\ x\ G'$$

<proof>

lemma *subset-ccNeighbors*:

$$S \subseteq ccNeighbors\ x\ G \iff ccProd\ \{x\}\ S \sqsubseteq G$$

<proof>

lemma *elem-ccNeighbors[simp]*:

$$y \in ccNeighbors\ x\ G \iff (y \dashv\ x \in G)$$

<proof>

lemma *ccNeighbors-ccField*:

$$ccNeighbors\ x\ G \subseteq ccField\ G\ \langle proof \rangle$$

lemma *ccNeighbors-disjoint-empty[simp]*:

$$ccNeighbors\ x\ G = \{\} \iff x \notin ccField\ G$$

<proof>

instance *CoCalls :: Join-cpo*

<proof>

lemma *ccNeighbors-lub[simp]*: $ccNeighbors\ x\ (lub\ Gs) = lub\ (ccNeighbors\ x\ 'Gs)$

<proof>

inductive *list-pairs* :: $'a\ list \Rightarrow ('a \times 'a) \Rightarrow bool$

where $list-pairs\ xs\ p \implies list-pairs\ (x\#\ xs)\ p$
 $| y \in set\ xs \implies list-pairs\ (x\#\ xs)\ (x,y)$

lift-definition *ccFromList* :: $var\ list \Rightarrow CoCalls$ **is** $\lambda xs. \{(x,y). list-pairs\ xs\ (x,y) \vee list-pairs\ xs\ (y,x)\}$

<proof>

lemma *ccFromList-Nil[simp]*: $ccFromList\ [] = \perp$

<proof>

lemma *ccFromList-Cons[simp]*: $ccFromList\ (x\#\ xs) = ccProd\ \{x\}\ (set\ xs) \sqcup ccFromList\ xs$

<proof>

lemma *ccFromList-append[simp]*: $ccFromList\ (xs@ys) = ccFromList\ xs \sqcup ccFromList\ ys \sqcup ccProd\ (set\ xs)\ (set\ ys)$

<proof>

$\langle \text{proof} \rangle$

lemma *ccFromList-filter*[simp]:

$ccFromList (\text{filter } P \text{ } xs) = cc\text{-restr } \{x. P \ x\} (ccFromList \text{ } xs)$

$\langle \text{proof} \rangle$

lemma *ccFromList-rotate*[simp]: $ccFromList (\text{rotate } n \ x) = (\text{if } n \leq 1 \text{ then } \perp \text{ else } ccProd \{x\} \{x\})$

$\langle \text{proof} \rangle$

definition *ccLinear* :: $var \ set \Rightarrow CoCalls \Rightarrow bool$

where $ccLinear \ S \ G = (\forall \ x \in S. \forall \ y \in S. x \text{---} y \notin G)$

lemma *ccLinear-bottom*[simp]:

$ccLinear \ S \ \perp$

$\langle \text{proof} \rangle$

lemma *ccLinear-empty*[simp]:

$ccLinear \ \{\} \ G$

$\langle \text{proof} \rangle$

lemma *ccLinear-lub*[simp]:

$ccLinear \ S \ (\text{lub } X) = (\forall \ G \in X. ccLinear \ S \ G)$

$\langle \text{proof} \rangle$

lemma *ccLinear-cc-restr*[intro]:

$ccLinear \ S \ G \Longrightarrow ccLinear \ S \ (cc\text{-restr } S' \ G)$

$\langle \text{proof} \rangle$

lemma *ccLinear-join*[simp]:

$ccLinear \ S \ (G \sqcup G') \longleftrightarrow ccLinear \ S \ G \wedge ccLinear \ S \ G'$

$\langle \text{proof} \rangle$

lemma *ccLinear-ccProd*[simp]:

$ccLinear \ S \ (ccProd \ S_1 \ S_2) \longleftrightarrow S_1 \cap S = \{\} \vee S_2 \cap S = \{\}$

$\langle \text{proof} \rangle$

lemma *ccLinear-mono1*: $ccLinear \ S' \ G \Longrightarrow S \subseteq S' \Longrightarrow ccLinear \ S \ G$

$\langle \text{proof} \rangle$

lemma *ccLinear-mono2*: $ccLinear \ S \ G' \Longrightarrow G \sqsubseteq G' \Longrightarrow ccLinear \ S \ G$

$\langle \text{proof} \rangle$

lemma *ccField-join*[simp]:

$ccField \ (G \sqcup G') = ccField \ G \cup ccField \ G' \langle \text{proof} \rangle$

lemma *ccField-lub*[*simp*]:
 $ccField (lub S) = \bigcup (ccField \text{ ` } S) \langle proof \rangle$

lemma *ccField-ccProd*:
 $ccField (ccProd S S') = (if S = \{\} then \{\} else if S' = \{\} then \{\} else S \cup S') \langle proof \rangle$

lemma *ccField-ccProd-subset*:
 $ccField (ccProd S S') \subseteq S \cup S' \langle proof \rangle$

lemma *cont-ccField*[*THEN cont-compose, simp, cont2cont*]:
 $cont ccField \langle proof \rangle$

end

11.2 CoCallGraph-Nominal

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

instantiation *CoCalls* :: *pt*
begin
lift-definition *permute-CoCalls* :: $perm \Rightarrow CoCalls \Rightarrow CoCalls$ **is** *permute*
 $\langle proof \rangle$
instance
 $\langle proof \rangle$
end

instance *CoCalls* :: *cont-pt*
 $\langle proof \rangle$

lemmas *lub-eqvt*[*OF exists-lub, simp, eqvt*]

lemma *cc-restr-perm*:
fixes $G :: CoCalls$
assumes $supp p \#* S$ **and** [*simp*]: $finite S$
shows $cc-restr S (p \cdot G) = cc-restr S G \langle proof \rangle$

lemma *inCC-eqvt*[*eqvt*]: $\pi \cdot (x \dashv\vdash y \in G) = (\pi \cdot x) \dashv\vdash (\pi \cdot y) \in (\pi \cdot G) \langle proof \rangle$

lemma *cc-restr-eqvt*[*eqvt*]: $\pi \cdot cc-restr S G = cc-restr (\pi \cdot S) (\pi \cdot G) \langle proof \rangle$

lemma *ccProd-eqvt*[*eqvt*]: $\pi \cdot ccProd\ S\ S' = ccProd\ (\pi \cdot S)\ (\pi \cdot S')$
 ⟨*proof*⟩
lemma *ccSquare-eqvt*[*eqvt*]: $\pi \cdot ccSquare\ S = ccSquare\ (\pi \cdot S)$
 ⟨*proof*⟩
lemma *ccNeighbors-eqvt*[*eqvt*]: $\pi \cdot ccNeighbors\ S\ G = ccNeighbors\ (\pi \cdot S)\ (\pi \cdot G)$
 ⟨*proof*⟩

end

12 Co-Call Cardinality Analysis

12.1 CoCallAnalysisSig

theory *CoCallAnalysisSig*
imports *Launchbury.Terms Arity CoCallGraph*
begin

locale *CoCallAnalysis* =
fixes *ccExp* :: *exp* \Rightarrow *Arity* \rightarrow *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 *CoCallAnalysisHeap* =
fixes *ccHeap* :: *heap* \Rightarrow *exp* \Rightarrow *Arity* \rightarrow *CoCalls*

end

12.2 CoCallAnalysisBinds

theory *CoCallAnalysisBinds*
imports *CoCallAnalysisSig AEnv AList-Utills-HOLCF Arity-Nominal CoCallGraph-Nominal*
begin

context *CoCallAnalysis*

begin

definition *ccBind* :: *var* \Rightarrow *exp* \Rightarrow $((AEnv \times CoCalls) \rightarrow CoCalls)$
where $ccBind\ v\ e = (\Lambda\ (ae, G). \text{if } (v \dashv\vdash v \notin G) \vee \neg\ isVal\ e \text{ then } cc-restr\ (fv\ e)\ (fup \cdot (ccExp\ e) \cdot (ae\ v)) \text{ else } ccSquare\ (fv\ e))$

lemma *ccBind-eq*:

$ccBind\ v\ e \cdot (ae, G) = (\text{if } v \dashv\vdash v \notin G \vee \neg\ isVal\ e \text{ then } \mathcal{G}^\perp_{ae}\ v\ e\ G \upharpoonright' fv\ e \text{ else } (fv\ e)^2)$

$\langle \text{proof} \rangle$

lemma *ccBind-strict[simp]*: $ccBind\ v\ e \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *ccField-ccBind*: $ccField\ (ccBind\ v\ e \cdot (ae, G)) \subseteq fv\ e$
 $\langle \text{proof} \rangle$

definition *ccBinds* :: $heap \Rightarrow ((AEnv \times CoCalls) \rightarrow CoCalls)$
where $ccBinds\ \Gamma = (\Lambda\ i.\ (\bigsqcup\ v \mapsto e \in \text{map-of}\ \Gamma.\ ccBind\ v\ e \cdot i))$

lemma *ccBinds-eq*:
 $ccBinds\ \Gamma \cdot i = (\bigsqcup\ v \mapsto e \in \text{map-of}\ \Gamma.\ ccBind\ v\ e \cdot i)$
 $\langle \text{proof} \rangle$

lemma *ccBinds-strict[simp]*: $ccBinds\ \Gamma \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *ccBinds-strict'[simp]*: $ccBinds\ \Gamma \cdot (\perp, \perp) = \perp$
 $\langle \text{proof} \rangle$

lemma *ccBinds-reorder1*:
assumes $\text{map-of}\ \Gamma\ v = \text{Some}\ e$
shows $ccBinds\ \Gamma = ccBind\ v\ e \sqcup ccBinds\ (\text{delete}\ v\ \Gamma)$
 $\langle \text{proof} \rangle$

lemma *ccBinds-Nil[simp]*:
 $ccBinds\ [] = \perp$
 $\langle \text{proof} \rangle$

lemma *ccBinds-Cons[simp]*:
 $ccBinds\ ((x, e) \# \Gamma) = ccBind\ x\ e \sqcup ccBinds\ (\text{delete}\ x\ \Gamma)$
 $\langle \text{proof} \rangle$

lemma *ccBind-below-ccBinds*: $\text{map-of}\ \Gamma\ x = \text{Some}\ e \implies ccBind\ x\ e \cdot ae \sqsubseteq (ccBinds\ \Gamma \cdot ae)$
 $\langle \text{proof} \rangle$

lemma *ccField-ccBinds*: $ccField\ (ccBinds\ \Gamma \cdot (ae, G)) \subseteq fv\ \Gamma$
 $\langle \text{proof} \rangle$

definition *ccBindsExtra* :: $heap \Rightarrow ((AEnv \times CoCalls) \rightarrow CoCalls)$
where $ccBindsExtra\ \Gamma = (\Lambda\ i.\ \text{snd}\ i \sqcup ccBinds\ \Gamma \cdot i \sqcup (\bigsqcup\ x \mapsto e \in \text{map-of}\ \Gamma.\ ccProd\ (fv\ e)\ (ccNeighbors\ x\ (\text{snd}\ i))))$

lemma *ccBindsExtra-simp*: $ccBindsExtra\ \Gamma \cdot i = \text{snd}\ i \sqcup ccBinds\ \Gamma \cdot i \sqcup (\bigsqcup\ x \mapsto e \in \text{map-of}\ \Gamma.\ ccProd\ (fv\ e)\ (ccNeighbors\ x\ (\text{snd}\ i)))$
 $\langle \text{proof} \rangle$

lemma *ccBindsExtra-eq*: $ccBindsExtra\ \Gamma \cdot (ae, G) =$

$G \sqcup \text{ccBinds } \Gamma \cdot (\text{ae}, G) \sqcup (\bigsqcup x \mapsto e \in \text{map-of } \Gamma. \text{fv } e \ G \times \text{ccNeighbors } x \ G)$
 ⟨proof⟩

lemma *ccBindsExtra-strict[simp]*: $\text{ccBindsExtra } \Gamma \cdot \perp = \perp$
 ⟨proof⟩

lemma *ccField-ccBindsExtra*:
 $\text{ccField } (\text{ccBindsExtra } \Gamma \cdot (\text{ae}, G)) \subseteq \text{fv } \Gamma \cup \text{ccField } G$
 ⟨proof⟩

end

lemma *ccBind-eqvt[eqvt]*: $\pi \cdot (\text{CoCallAnalysis.ccBind } \text{cccExp } x \ e) = \text{CoCallAnalysis.ccBind } (\pi \cdot \text{cccExp}) (\pi \cdot x) (\pi \cdot e)$
 ⟨proof⟩

lemma *ccBinds-eqvt[eqvt]*: $\pi \cdot (\text{CoCallAnalysis.ccBinds } \text{cccExp } \Gamma) = \text{CoCallAnalysis.ccBinds } (\pi \cdot \text{cccExp}) (\pi \cdot \Gamma)$
 ⟨proof⟩

lemma *ccBindsExtra-eqvt[eqvt]*: $\pi \cdot (\text{CoCallAnalysis.ccBindsExtra } \text{cccExp } \Gamma) = \text{CoCallAnalysis.ccBindsExtra } (\pi \cdot \text{cccExp}) (\pi \cdot \Gamma)$
 ⟨proof⟩

lemma *ccBind-cong[fundef-cong]*:
 $\text{cccExp1 } e = \text{cccExp2 } e \implies \text{CoCallAnalysis.ccBind } \text{cccExp1 } x \ e = \text{CoCallAnalysis.ccBind } \text{cccExp2 } x \ e$
 ⟨proof⟩

lemma *ccBinds-cong[fundef-cong]*:
 $\llbracket (\bigwedge e. e \in \text{snd } \text{'set heap2} \implies \text{cccExp1 } e = \text{cccExp2 } e); \text{heap1} = \text{heap2} \rrbracket$
 $\implies \text{CoCallAnalysis.ccBinds } \text{cccExp1 } \text{heap1} = \text{CoCallAnalysis.ccBinds } \text{cccExp2 } \text{heap2}$
 ⟨proof⟩

lemma *ccBindsExtra-cong[fundef-cong]*:
 $\llbracket (\bigwedge e. e \in \text{snd } \text{'set heap2} \implies \text{cccExp1 } e = \text{cccExp2 } e); \text{heap1} = \text{heap2} \rrbracket$
 $\implies \text{CoCallAnalysis.ccBindsExtra } \text{cccExp1 } \text{heap1} = \text{CoCallAnalysis.ccBindsExtra } \text{cccExp2 } \text{heap2}$
 ⟨proof⟩

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* + *CoCallAnalysisHeap* + *ArityAnalysisLetSafe* +
assumes *ccExp-App*: $ccExp\ e.(inc.a) \sqcup ccProd\ \{x\}\ (insert\ x\ (fv\ e)) \sqsubseteq ccExp\ (App\ e\ x).a$
assumes *ccExp-Lam*: $cc-restr\ (fv\ (Lam\ [y].\ e))\ (ccExp\ e.(pred.n)) \sqsubseteq ccExp\ (Lam\ [y].\ e).n$
assumes *ccExp-subst*: $x \notin S \implies y \notin S \implies cc-restr\ S\ (ccExp\ e[y::=x].a) \sqsubseteq cc-restr\ S\ (ccExp\ e.a)$

assumes *ccExp-pap*: $isVal\ e \implies ccExp\ e.0 = ccSquare\ (fv\ e)$
assumes *ccExp-Let*: $cc-restr\ (-domA\ \Gamma)\ (ccHeap\ \Gamma\ e.a) \sqsubseteq ccExp\ (Let\ \Gamma\ e).a$
assumes *ccExp-IfThenElse*: $ccExp\ scrut.0 \sqcup (ccExp\ e1.a \sqcup ccExp\ e2.a) \sqcup ccProd\ (edom\ (Aexp\ scrut.0))\ (edom\ (Aexp\ e1.a) \cup edom\ (Aexp\ e2.a)) \sqsubseteq ccExp\ (scrut\ ?\ e1\ : e2).a$

assumes *ccHeap-Exp*: $ccExp\ e.a \sqsubseteq ccHeap\ \Delta\ e.a$
assumes *ccHeap-Heap*: $map-of\ \Delta\ x = Some\ e' \implies (Aheap\ \Delta\ e.a)\ x = up.a' \implies ccExp\ e'.a' \sqsubseteq ccHeap\ \Delta\ e.a$
assumes *ccHeap-Extra-Edges*:
 $map-of\ \Delta\ x = Some\ e' \implies (Aheap\ \Delta\ e.a)\ x = up.a' \implies ccProd\ (fv\ e')\ (ccNeighbors\ x\ (ccHeap\ \Delta\ e.a) - \{x\} \cap thunks\ \Delta) \sqsubseteq ccHeap\ \Delta\ e.a$

assumes *aHeap-thunks-rec*: $\neg nonrec\ \Gamma \implies x \in thunks\ \Gamma \implies x \in edom\ (Aheap\ \Gamma\ e.a) \implies (Aheap\ \Gamma\ e.a)\ x = up.0$

assumes *aHeap-thunks-nonrec*: $nonrec\ \Gamma \implies x \in thunks\ \Gamma \implies x--x \in ccExp\ e.a \implies (Aheap\ \Gamma\ e.a)\ x = up.0$

end

12.5 CoCallFix

theory *CoCallFix*
imports *CoCallAnalysisSig CoCallAnalysisBinds ArityAnalysisSig Launchbury.Env-Nominal ArityAnalysisFix*
begin

locale *CoCallArityAnalysis* =
fixes *cccExp* :: $exp \Rightarrow (Arity \rightarrow AEnv \times CoCalls)$
begin

definition $Aexp :: exp \Rightarrow (Aarity \rightarrow AEnv)$
where $Aexp\ e = (\Lambda\ a.\ fst\ (cccExp\ e\ \cdot\ a))$

sublocale $AarityAnalysis\ Aexp\langle proof \rangle$

abbreviation $Aexp\text{-}syn'\ (\mathcal{A}_\cdot)$ **where** $\mathcal{A}_a \equiv (\lambda e.\ Aexp\ e\ \cdot\ a)$

abbreviation $Aexp\text{-}bot\text{-}syn'\ (\mathcal{A}^\perp_\cdot)$ **where** $\mathcal{A}^\perp_a \equiv (\lambda e.\ fup\ \cdot\ (Aexp\ e)\ \cdot\ a)$

lemma $Aexp\text{-}eq$:

$\mathcal{A}_a\ e = fst\ (cccExp\ e\ \cdot\ a)$
 $\langle proof \rangle$

lemma $fup\text{-}Aexp\text{-}eq$:

$fup\ \cdot\ (Aexp\ e)\ \cdot\ a = fst\ (fup\ \cdot\ (cccExp\ e)\ \cdot\ a)$
 $\langle proof \rangle$

definition $CCexp :: exp \Rightarrow (Aarity \rightarrow CoCalls)$ **where** $CCexp\ \Gamma = (\Lambda\ a.\ snd\ (cccExp\ \Gamma\ \cdot\ a))$

lemma $CCexp\text{-}eq$:

$CCexp\ e\ \cdot\ a = snd\ (cccExp\ e\ \cdot\ a)$
 $\langle proof \rangle$

lemma $fup\text{-}CCexp\text{-}eq$:

$fup\ \cdot\ (CCexp\ e)\ \cdot\ a = snd\ (fup\ \cdot\ (cccExp\ e)\ \cdot\ a)$
 $\langle proof \rangle$

sublocale $CoCallAnalysis\ CCexp\langle proof \rangle$

definition $CCfix :: heap \Rightarrow (AEnv \times CoCalls) \rightarrow CoCalls$

where $CCfix\ \Gamma = (\Lambda\ aeG.\ (\mu\ G'.\ ccBindsExtra\ \Gamma\ \cdot\ (fst\ aeG\ ,\ G')\ \sqcup\ snd\ aeG))$

lemma $CCfix\text{-}eq$:

$CCfix\ \Gamma\ \cdot\ (ae,\ G) = (\mu\ G'.\ ccBindsExtra\ \Gamma\ \cdot\ (ae,\ G')\ \sqcup\ G)$
 $\langle proof \rangle$

lemma $CCfix\text{-}unroll$: $CCfix\ \Gamma\ \cdot\ (ae,\ G) = ccBindsExtra\ \Gamma\ \cdot\ (ae,\ CCfix\ \Gamma\ \cdot\ (ae,\ G))\ \sqcup\ G$

$\langle proof \rangle$

lemma $fup\text{-}ccExp\text{-}restr\text{-}subst'$:

assumes $\bigwedge a.\ cc\text{-}restr\ S\ (CCexp\ e[x::=y]\ \cdot\ a) = cc\text{-}restr\ S\ (CCexp\ e\ \cdot\ a)$
shows $cc\text{-}restr\ S\ (fup\ \cdot\ (CCexp\ e[x::=y])\ \cdot\ a) = cc\text{-}restr\ S\ (fup\ \cdot\ (CCexp\ e)\ \cdot\ a)$
 $\langle proof \rangle$

lemma $ccBindsExtra\text{-}restr\text{-}subst'$:

assumes $\bigwedge x'\ e\ a.\ (x',\ e) \in set\ \Gamma \implies cc\text{-}restr\ S\ (CCexp\ e[x::=y]\ \cdot\ a) = cc\text{-}restr\ S\ (CCexp\ e\ \cdot\ a)$

assumes $x \notin S$

assumes $y \notin S$

assumes $domA \Gamma \subseteq S$
shows $cc\text{-restr } S (cc\text{BindsExtra } \Gamma[x::h=y].(ae, G))$
 $= cc\text{-restr } S (cc\text{BindsExtra } \Gamma.(ae \text{ f}' S, cc\text{-restr } S G))$
 $\langle proof \rangle$

lemma $cc\text{BindsExtra-restr}$:

assumes $domA \Gamma \subseteq S$
shows $cc\text{-restr } S (cc\text{BindsExtra } \Gamma.(ae, G)) = cc\text{-restr } S (cc\text{BindsExtra } \Gamma.(ae \text{ f}' S, cc\text{-restr } S G))$
 $\langle proof \rangle$

lemma $CCfix-restr$:

assumes $domA \Gamma \subseteq S$
shows $cc\text{-restr } S (CCfix \Gamma.(ae, G)) = cc\text{-restr } S (CCfix \Gamma.(ae \text{ f}' S, cc\text{-restr } S G))$
 $\langle proof \rangle$

lemma $ccField-CCfix$:

shows $ccField (CCfix \Gamma.(ae, G)) \subseteq fv \Gamma \cup ccField G$
 $\langle proof \rangle$

lemma $CCfix-restr\text{-subst}'$:

assumes $\bigwedge x' e a. (x', e) \in set \Gamma \implies cc\text{-restr } S (CCexp e[x::=y].a) = cc\text{-restr } S (CCexp e.a)$
assumes $x \notin S$
assumes $y \notin S$
assumes $domA \Gamma \subseteq S$
shows $cc\text{-restr } S (CCfix \Gamma[x::h=y].(ae, G)) = cc\text{-restr } S (CCfix \Gamma.(ae \text{ f}' S, cc\text{-restr } S G))$
 $\langle proof \rangle$

end

lemma $Aexp\text{-eqvt}[eqvt]$: $\pi \cdot (CoCallArityAnalysis.Aexp cccExp e) = CoCallArityAnalysis.Aexp (\pi \cdot cccExp) (\pi \cdot e)$
 $\langle proof \rangle$

lemma $CCexp\text{-eqvt}[eqvt]$: $\pi \cdot (CoCallArityAnalysis.CCexp cccExp e) = CoCallArityAnalysis.CCexp (\pi \cdot cccExp) (\pi \cdot e)$
 $\langle proof \rangle$

lemma $CCfix\text{-eqvt}[eqvt]$: $\pi \cdot (CoCallArityAnalysis.CCfix cccExp \Gamma) = CoCallArityAnalysis.CCfix (\pi \cdot cccExp) (\pi \cdot \Gamma)$
 $\langle proof \rangle$

lemma $ccFix\text{-cong}[fundef\text{-cong}]$:

$\llbracket (\bigwedge e. e \in snd \text{ ' set heap2} \implies cccexp1 e = cccexp2 e); heap1 = heap2 \rrbracket$
 $\implies CoCallArityAnalysis.CCfix cccexp1 heap1 = CoCallArityAnalysis.CCfix cccexp2 heap2$
 $\langle proof \rangle$

context *CoCallAriyAnalysis*

begin

definition *cccFix* :: *heap* \Rightarrow $((AEnv \times CoCalls) \rightarrow (AEnv \times CoCalls))$

where *cccFix* $\Gamma = (\Lambda i. (Afix \Gamma.(fst i \sqcup (\lambda-.up \cdot 0) f|' \text{thunks } \Gamma), CCfix \Gamma.(Afix \Gamma.(fst i \sqcup (\lambda-.up \cdot 0) f|' (\text{thunks } \Gamma)), snd i))$

lemma *cccFix-eq*:

cccFix $\Gamma.i = (Afix \Gamma.(fst i \sqcup (\lambda-.up \cdot 0) f|' \text{thunks } \Gamma), CCfix \Gamma.(Afix \Gamma.(fst i \sqcup (\lambda-.up \cdot 0) f|' (\text{thunks } \Gamma)), snd i))$

$\langle proof \rangle$

end

lemma *cccFix-eqvt*[*eqvt*]: $\pi \cdot (CoCallAriyAnalysis.cccFix \text{ cccExp } \Gamma) = CoCallAriyAnalysis.cccFix (\pi \cdot \text{ cccExp}) (\pi \cdot \Gamma)$

$\langle proof \rangle$

lemma *cccFix-cong*[*fundef-cong*]:

$\llbracket (\bigwedge e. e \in \text{snd } ' \text{ set heap2} \implies \text{cccexp1 } e = \text{cccexp2 } e); \text{heap1} = \text{heap2} \rrbracket$
 $\implies CoCallAriyAnalysis.cccFix \text{ cccexp1 } \text{heap1} = CoCallAriyAnalysis.cccFix \text{ cccexp2 } \text{heap2}$

$\langle proof \rangle$

12.5.1 The non-recursive case

definition *ABind-nonrec* :: *var* \Rightarrow *exp* \Rightarrow $AEnv \times CoCalls \rightarrow Arity_{\perp}$

where

ABind-nonrec $x e = (\Lambda i. (\text{if isVal } e \vee x \dashv\vdash x \notin (\text{snd } i) \text{ then } \text{fst } i \ x \text{ else } \text{up} \cdot 0))$

lemma *ABind-nonrec-eq*:

ABind-nonrec $x e.(ae, G) = (\text{if isVal } e \vee x \dashv\vdash x \notin G \text{ then } ae \ x \text{ else } \text{up} \cdot 0)$

$\langle proof \rangle$

lemma *ABind-nonrec-eqvt*[*eqvt*]: $\pi \cdot (ABind-nonrec \ x \ e) = ABind-nonrec (\pi \cdot x) (\pi \cdot e)$

$\langle proof \rangle$

lemma *ABind-nonrec-above-arg*:

$ae \ x \sqsubseteq ABind-nonrec \ x \ e \cdot (ae, G)$

$\langle proof \rangle$

definition *Aheap-nonrec where*

Aheap-nonrec $x e = (\Lambda i. \text{esing } x.(ABind-nonrec \ x \ e.i))$

lemma *Aheap-nonrec-simp*:

Aheap-nonrec $x e.i = \text{esing } x.(ABind-nonrec \ x \ e.i)$

$\langle proof \rangle$

lemma *Aheap-nonrec-lookup*[*simp*]:

$(Aheap-nonrec \ x \ e.i) \ x = ABind-nonrec \ x \ e.i$

<proof>

lemma *Aheap-nonrec-eqvt'*[*eqvt*]:

$$\pi \cdot (\text{Aheap-nonrec } x \ e) = \text{Aheap-nonrec } (\pi \cdot x) \ (\pi \cdot e)$$

<proof>

context *CoCallArietyAnalysis*

begin

definition *Afix-nonrec*

$$\text{where } \text{Afix-nonrec } x \ e = (\Lambda \ i. \text{fup} \cdot (\text{Aexp } e) \cdot (\text{ABind-nonrec } x \ e \cdot i) \sqcup \text{fst } i)$$

lemma *Afix-nonrec-eq*[*simp*]:

$$\text{Afix-nonrec } x \ e \cdot i = \text{fup} \cdot (\text{Aexp } e) \cdot (\text{ABind-nonrec } x \ e \cdot i) \sqcup \text{fst } i$$

<proof>

definition *CCfix-nonrec*

$$\text{where } \text{CCfix-nonrec } x \ e = (\Lambda \ i. \text{ccBind } x \ e \cdot (\text{Aheap-nonrec } x \ e \cdot i, \text{snd } i) \sqcup \text{ccProd } (\text{fv } e) \\ (\text{ccNeighbors } x \ (\text{snd } i) - (\text{if isVal } e \text{ then } \{\} \text{ else } \{x\})) \sqcup \text{snd } i)$$

lemma *CCfix-nonrec-eq*[*simp*]:

$$\text{CCfix-nonrec } x \ e \cdot i = \text{ccBind } x \ e \cdot (\text{Aheap-nonrec } x \ e \cdot i, \text{snd } i) \sqcup \text{ccProd } (\text{fv } e) \\ (\text{ccNeighbors } x \ (\text{snd } i) - (\text{if isVal } e \text{ then } \{\} \text{ else } \{x\})) \sqcup \text{snd } i$$

<proof>

definition *cccFix-nonrec* :: *var* \Rightarrow *exp* \Rightarrow ((*AEnv* \times *CoCalls*) \rightarrow (*AEnv* \times *CoCalls*))

$$\text{where } \text{cccFix-nonrec } x \ e = (\Lambda \ i. (\text{Afix-nonrec } x \ e \cdot i, \text{CCfix-nonrec } x \ e \cdot i))$$

lemma *cccFix-nonrec-eq*[*simp*]:

$$\text{cccFix-nonrec } x \ e \cdot i = (\text{Afix-nonrec } x \ e \cdot i, \text{CCfix-nonrec } x \ e \cdot i)$$

<proof>

end

lemma *AFix-nonrec-eqvt*[*eqvt*]: $\pi \cdot (\text{CoCallArietyAnalysis.Afix-nonrec } \text{cccExp } x \ e) = \text{CoCallArietyAnalysis.Afix-nonrec } (\pi \cdot \text{cccExp}) \ (\pi \cdot x) \ (\pi \cdot e)$

<proof>

lemma *CCFix-nonrec-eqvt*[*eqvt*]: $\pi \cdot (\text{CoCallArietyAnalysis.CCfix-nonrec } \text{cccExp } x \ e) = \text{CoCallArietyAnalysis.CCfix-nonrec } (\pi \cdot \text{cccExp}) \ (\pi \cdot x) \ (\pi \cdot e)$

<proof>

lemma *cccFix-nonrec-eqvt*[*eqvt*]: $\pi \cdot (\text{CoCallArietyAnalysis.cccFix-nonrec } \text{cccExp } x \ e) = \text{CoCallArietyAnalysis.cccFix-nonrec } (\pi \cdot \text{cccExp}) \ (\pi \cdot x) \ (\pi \cdot e)$

<proof>

12.5.2 Combining the cases

context *CoCallAriyAnalysis*

begin

definition *cccFix-choose* :: *heap* \Rightarrow $((AEnv \times CoCalls) \rightarrow (AEnv \times CoCalls))$

where *cccFix-choose* $\Gamma = (if\ nonrec\ \Gamma\ then\ case\ -prod\ cccFix\ -nonrec\ (hd\ \Gamma)\ else\ cccFix\ \Gamma)$

lemma *cccFix-choose-simp1*[*simp*]:

$\neg\ nonrec\ \Gamma \Longrightarrow cccFix\ -choose\ \Gamma = cccFix\ \Gamma$

<proof>

lemma *cccFix-choose-simp2*[*simp*]:

$x \notin fv\ e \Longrightarrow cccFix\ -choose\ [(x,e)] = cccFix\ -nonrec\ x\ e$

<proof>

end

lemma *cccFix-choose-eqvt*[*eqvt*]: $\pi \cdot (CoCallAriyAnalysis.cccFix\ -choose\ cccExp\ \Gamma) = CoCallAriyAnalysis.cccFix\ -choose\ (\pi \cdot cccExp)\ (\pi \cdot \Gamma)$

<proof>

lemma *cccFix-nonrec-cong*[*fundef-cong*]:

$cccExp1\ e = cccExp2\ e \Longrightarrow CoCallAriyAnalysis.cccFix\ -nonrec\ cccExp1\ x\ e = CoCallAriyAnalysis.cccFix\ -nonrec\ cccExp2\ x\ e$

<proof>

lemma *cccFix-choose-cong*[*fundef-cong*]:

$\llbracket (\bigwedge e. e \in snd\ 'set\ heap2 \Longrightarrow cccExp1\ e = cccExp2\ e); heap1 = heap2 \rrbracket$

$\Longrightarrow CoCallAriyAnalysis.cccFix\ -choose\ cccExp1\ heap1 = CoCallAriyAnalysis.cccFix\ -choose\ cccExp2\ heap2$

<proof>

end

12.6 CoCallGraph-TTree

theory *CoCallGraph-TTree*

imports *CoCallGraph TTree-HOLCF*

begin

lemma *interleave-ccFromList*:

$xs \in interleave\ ys\ zs \Longrightarrow ccFromList\ xs = ccFromList\ ys \sqcup ccFromList\ zs \sqcup ccProd\ (set\ ys)\ (set\ zs)$

<proof>

lift-definition *ccApprox* :: *var ttree* \Rightarrow *CoCalls*

is $\lambda\ xss. lub\ (ccFromList\ 'xss)$ *<proof>*

lemma *ccApprox-paths*: $ccApprox\ t = lub\ (ccFromList\ \text{'}\ (paths\ t))\ \langle proof \rangle$

lemma *ccApprox-strict[simp]*: $ccApprox\ \perp = \perp$
 $\langle proof \rangle$

lemma *in-ccApprox*: $(x\ \text{---}\ y \in (ccApprox\ t)) \longleftrightarrow (\exists\ xs \in paths\ t. (x\ \text{---}\ y \in (ccFromList\ xs)))$
 $\langle proof \rangle$

lemma *ccApprox-mono*: $paths\ t \subseteq paths\ t' \implies ccApprox\ t \sqsubseteq ccApprox\ t'$
 $\langle proof \rangle$

lemma *ccApprox-mono'*: $t \sqsubseteq t' \implies ccApprox\ t \sqsubseteq ccApprox\ t'$
 $\langle proof \rangle$

lemma *ccApprox-belowI*: $(\bigwedge\ xs. xs \in paths\ t \implies ccFromList\ xs \sqsubseteq G) \implies ccApprox\ t \sqsubseteq G$
 $\langle proof \rangle$

lemma *ccApprox-below-iff*: $ccApprox\ t \sqsubseteq G \longleftrightarrow (\forall\ xs \in paths\ t. ccFromList\ xs \sqsubseteq G)$
 $\langle proof \rangle$

lemma *cc-restr-ccApprox-below-iff*: $cc-restr\ S\ (ccApprox\ t) \sqsubseteq G \longleftrightarrow (\forall\ xs \in paths\ t. cc-restr\ S\ (ccFromList\ xs) \sqsubseteq G)$
 $\langle proof \rangle$

lemma *ccFromList-below-ccApprox*:
 $xs \in paths\ t \implies ccFromList\ xs \sqsubseteq ccApprox\ t$
 $\langle proof \rangle$

lemma *ccApprox-nxt-below*:
 $ccApprox\ (nxt\ t\ x) \sqsubseteq ccApprox\ t$
 $\langle proof \rangle$

lemma *ccApprox-ttree-restr-nxt-below*:
 $ccApprox\ (ttree-restr\ S\ (nxt\ t\ x)) \sqsubseteq ccApprox\ (ttree-restr\ S\ t)$
 $\langle proof \rangle$

lemma *ccApprox-ttree-restr[simp]*: $ccApprox\ (ttree-restr\ S\ t) = cc-restr\ S\ (ccApprox\ t)$
 $\langle proof \rangle$

lemma *ccApprox-both*: $ccApprox\ (t \otimes t') = ccApprox\ t \sqcup ccApprox\ t' \sqcup ccProd\ (carrier\ t)\ (carrier\ t')$
 $\langle proof \rangle$

lemma *ccApprox-many-calls[simp]*:
 $ccApprox\ (many-calls\ x) = ccProd\ \{x\}\ \{x\}$
 $\langle proof \rangle$

lemma *ccApprox-single[simp]*:
 $ccApprox\ (TTree.single\ y) = \perp$

$\langle \text{proof} \rangle$

lemma *ccApprox-either*[simp]: $ccApprox (t \oplus \oplus t') = ccApprox t \sqcup ccApprox t'$
 $\langle \text{proof} \rangle$

lemma *wild-recursion*:

assumes $ccApprox t \sqsubseteq G$
assumes $\bigwedge x. x \notin S \implies f x = \text{empty}$
assumes $\bigwedge x. x \in S \implies ccApprox (f x) \sqsubseteq G$
assumes $\bigwedge x. x \in S \implies ccProd (ccNeighbors x G) (carrier (f x)) \sqsubseteq G$
shows $ccApprox (ttree-restr (-S) (substitute f T t)) \sqsubseteq G$

$\langle \text{proof} \rangle$

lemma *wild-recursion-thunked*:

assumes $ccApprox t \sqsubseteq G$
assumes $\bigwedge x. x \notin S \implies f x = \text{empty}$
assumes $\bigwedge x. x \in S \implies ccApprox (f x) \sqsubseteq G$
assumes $\bigwedge x. x \in S \implies ccProd (ccNeighbors x G - \{x\} \cap T) (carrier (f x)) \sqsubseteq G$
shows $ccApprox (ttree-restr (-S) (substitute f T t)) \sqsubseteq G$

$\langle \text{proof} \rangle$

inductive-set *valid-lists* :: $var\ set \Rightarrow CoCalls \Rightarrow var\ list\ set$
for $S\ G$

where $\square \in \text{valid-lists } S\ G$

$| set\ xs \subseteq ccNeighbors\ x\ G \implies xs \in \text{valid-lists } S\ G \implies x \in S \implies x\#\!xs \in \text{valid-lists } S\ G$

inductive-simps *valid-lists-simps*[simp]: $\square \in \text{valid-lists } S\ G (x\#\!xs) \in \text{valid-lists } S\ G$

inductive-cases *valid-lists-ConsE*: $(x\#\!xs) \in \text{valid-lists } S\ G$

lemma *valid-lists-downset-aux*:

$xs \in \text{valid-lists } S\ CoCalls \implies \text{butlast } xs \in \text{valid-lists } S\ CoCalls$

$\langle \text{proof} \rangle$

lemma *valid-lists-subset*: $xs \in \text{valid-lists } S\ G \implies set\ xs \subseteq S$

$\langle \text{proof} \rangle$

lemma *valid-lists-mono1*:

assumes $S \subseteq S'$

shows $\text{valid-lists } S\ G \subseteq \text{valid-lists } S'\ G$

$\langle \text{proof} \rangle$

lemma *valid-lists-chain1*:

assumes $chain\ Y$

assumes $xs \in \text{valid-lists } (\bigcup (Y \text{ ' } UNIV))\ G$

shows $\exists i. xs \in \text{valid-lists } (Y\ i)\ G$

$\langle \text{proof} \rangle$

lemma *valid-lists-chain2*:

assumes *chain Y*

assumes $xs \in \text{valid-lists } S \ (\sqcup i. Y i)$

shows $\exists i. xs \in \text{valid-lists } S \ (Y i)$

<proof>

lemma *valid-lists-cc-restr*: $\text{valid-lists } S \ G = \text{valid-lists } S \ (\text{cc-restr } S \ G)$

<proof>

lemma *interleave-valid-list*:

$xs \in ys \otimes zs \implies ys \in \text{valid-lists } S \ G \implies zs \in \text{valid-lists } S' \ G' \implies xs \in \text{valid-lists } (S \cup S')$
 $(G \sqcup (G' \sqcup \text{ccProd } S \ S'))$

<proof>

lemma *interleave-valid-list'*:

$xs \in \text{valid-lists } (S \cup S') \ G \implies \exists ys \ zs. xs \in ys \otimes zs \wedge ys \in \text{valid-lists } S \ G \wedge zs \in \text{valid-lists } S' \ G$

<proof>

lemma *many-calls-valid-list*:

$xs \in \text{valid-lists } \{x\} \ (\text{ccProd } \{x\} \ \{x\}) \implies xs \in \text{range } (\lambda n. \text{replicate } n \ x)$

<proof>

lemma *filter-valid-lists*:

$xs \in \text{valid-lists } S \ G \implies \text{filter } P \ xs \in \text{valid-lists } \{a \in S. P \ a\} \ G$

<proof>

lift-definition *ccTTree* :: $\text{var set} \Rightarrow \text{CoCalls} \Rightarrow \text{var tree}$ **is** $\lambda S \ G. \text{valid-lists } S \ G$

<proof>

lemma *paths-ccTTree[simp]*: $\text{paths } (\text{ccTTree } S \ G) = \text{valid-lists } S \ G$ *<proof>*

lemma *carrier-ccTTree[simp]*: $\text{carrier } (\text{ccTTree } S \ G) = S$

<proof>

lemma *valid-lists-ccFromList*:

$xs \in \text{valid-lists } S \ G \implies \text{ccFromList } xs \sqsubseteq \text{cc-restr } S \ G$

<proof>

lemma *ccApprox-ccTTree[simp]*: $\text{ccApprox } (\text{ccTTree } S \ G) = \text{cc-restr } S \ G$

<proof>

lemma *below-ccTTreeI*:

assumes $\text{carrier } t \subseteq S$ **and** $\text{ccApprox } t \sqsubseteq G$

shows $t \sqsubseteq \text{ccTTree } S \ G$

<proof>

lemma *ccTTree-mono1*:

$S \subseteq S' \implies \text{ccTTree } S \ G \sqsubseteq \text{ccTTree } S' \ G$
 ⟨proof⟩

lemma *cont-ccTTree1*:
 $\text{cont } (\lambda S. \text{ccTTree } S \ G)$
 ⟨proof⟩

lemma *ccTTree-mono2*:
 $G \sqsubseteq G' \implies \text{ccTTree } S \ G \sqsubseteq \text{ccTTree } S \ G'$
 ⟨proof⟩

lemma *ccTTree-mono*:
 $S \subseteq S' \implies G \sqsubseteq G' \implies \text{ccTTree } S \ G \sqsubseteq \text{ccTTree } S' \ G'$
 ⟨proof⟩

lemma *cont-ccTTree2*:
 $\text{cont } (\text{ccTTree } S)$
 ⟨proof⟩

lemmas *cont-ccTTree = cont-compose2*[**where** $c = \text{ccTTree}$, *OF cont-ccTTree1 cont-ccTTree2, simp, cont2cont*]

lemma *ccTTree-below-singleI*:
assumes $S \cap S' = \{\}$
shows $\text{ccTTree } S \ G \sqsubseteq \text{singles } S'$
 ⟨proof⟩

lemma *ccTTree-cc-restr*: $\text{ccTTree } S \ G = \text{ccTTree } S \ (\text{cc-restr } S \ G)$
 ⟨proof⟩

lemma *ccTTree-cong-below*: $\text{cc-restr } S \ G \sqsubseteq \text{cc-restr } S \ G' \implies \text{ccTTree } S \ G \sqsubseteq \text{ccTTree } S \ G'$
 ⟨proof⟩

lemma *ccTTree-cong*: $\text{cc-restr } S \ G = \text{cc-restr } S \ G' \implies \text{ccTTree } S \ G = \text{ccTTree } S \ G'$
 ⟨proof⟩

lemma *either-ccTTree*:
 $\text{ccTTree } S \ G \oplus \oplus \text{ccTTree } S' \ G' \sqsubseteq \text{ccTTree } (S \cup S') \ (G \sqcup G')$
 ⟨proof⟩

lemma *interleave-ccTTree*:
 $\text{ccTTree } S \ G \otimes \otimes \text{ccTTree } S' \ G' \sqsubseteq \text{ccTTree } (S \cup S') \ (G \sqcup G' \sqcup \text{ccProd } S \ S')$
 ⟨proof⟩

lemma *interleave-ccTTree'*:
 $\text{ccTTree } (S \cup S') \ G \sqsubseteq \text{ccTTree } S \ G \otimes \otimes \text{ccTTree } S' \ G$

<proof>

lemma *many-calls-ccTTree*:

shows *many-calls* $x = ccTTree \{x\} (ccProd \{x\} \{x\})$

<proof>

lemma *filter-valid-lists'*:

$xs \in valid-lists \{x' \in S. P x'\} G \implies xs \in filter P ' valid-lists S G$

<proof>

lemma *without-ccTTree[simp]*:

without $x (ccTTree S G) = ccTTree (S - \{x\}) G$

<proof>

lemma *tree-restr-ccTTree[simp]*:

tree-restr $S' (ccTTree S G) = ccTTree (S \cap S') G$

<proof>

lemma *repeatable-ccTTree-ccSquare*: $S \subseteq S' \implies repeatable (ccTTree S (ccSquare S'))$

<proof>

An alternative definition

inductive *valid-lists'* :: *var set* \Rightarrow *CoCalls* \Rightarrow *var set* \Rightarrow *var list* \Rightarrow *bool*

for $S G$

where *valid-lists'* $S G prefix []$

$| prefix \subseteq ccNeighbors x G \implies valid-lists' S G (insert x prefix) xs \implies x \in S \implies valid-lists' S G prefix (x\#xs)$

inductive-simps *valid-lists'-simps[simp]*: *valid-lists'* $S G prefix []$ *valid-lists'* $S G prefix (x\#xs)$

inductive-cases *valid-lists'-ConsE*: *valid-lists'* $S G prefix (x\#xs)$

lemma *valid-lists-valid-lists'*:

$xs \in valid-lists S G \implies ccProd prefix (set xs) \sqsubseteq G \implies valid-lists' S G prefix xs$

<proof>

lemma *valid-lists'-valid-lists-aux*:

$valid-lists' S G prefix xs \implies x \in prefix \implies ccProd (set xs) \{x\} \sqsubseteq G$

<proof>

lemma *valid-lists'-valid-lists*:

$valid-lists' S G prefix xs \implies xs \in valid-lists S G$

<proof>

Yet another definition

lemma *valid-lists-characterization*:

$xs \in valid-lists S G \iff set xs \subseteq S \wedge (\forall n. ccProd (set (take n xs)) (set (drop n xs))) \sqsubseteq G$

<proof>

end

12.7 CoCallImplTTree

```
theory CoCallImplTTree
imports TTreeAnalysisSig Env-Set-Cpo CoCallAriySig CoCallGraph-TTree
begin

context CoCallAriy
begin
  definition Texp :: exp  $\Rightarrow$  Ariy  $\rightarrow$  var ttree
    where Texp e = ( $\Lambda$  a. ccTTree (edom (Aexp e  $\cdot$  a)) (ccExp e.a))

  lemma Texp-simp: Texp e.a = ccTTree (edom (Aexp e  $\cdot$  a)) (ccExp e.a)
    <proof>

  sublocale TTreeAnalysis Texp <proof>
end

end
```

12.8 CoCallImplTTreeSafe

```
theory CoCallImplTTreeSafe
imports CoCallImplTTree CoCallAnalysisSpec TTreeAnalysisSpec
begin

lemma valid-lists-many-calls:
  assumes  $\neg$  one-call-in-path x p
  assumes p  $\in$  valid-lists S G
  shows x--x  $\in$  G
  <proof>

context CoCallAriyEdom
begin
  lemma carrier-Fexp': carrier (Texp e.a)  $\subseteq$  fv e
    <proof>
end

context CoCallAriySafe
begin

lemma carrier-AnalBinds-below:
  carrier ((Texp.AnalBinds  $\Delta$ .(Aheap  $\Delta$  e.a)) x)  $\subseteq$  edom ((ABinds  $\Delta$ ).(Aheap  $\Delta$  e.a))
  <proof>

sublocale TTreeAnalysisCarrier Texp
```

<proof>

sublocale *TTreeAnalysisSafe Texp*

<proof>

definition *Theap* :: *heap* \Rightarrow *exp* \Rightarrow *Arity* \rightarrow *var tree*

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*)

<proof>

lemma *carrier-Fheap'*:*carrier* (*Theap* Γ *e*·*a*) = *edom* (*Aheap* Γ *e*·*a*)

<proof>

sublocale *TTreeAnalysisCardinalityHeap Texp Aexp Aheap Theap*

<proof>

end

lemma *paths-singles*: *xs* \in *paths* (*singles* *S*) \longleftrightarrow ($\forall x \in S$. *one-call-in-path* *x xs*)

<proof>

lemma *paths-singles'*: *xs* \in *paths* (*singles* *S*) \longleftrightarrow ($\forall x \in (\text{set } xs \cap S)$. *one-call-in-path* *x xs*)

<proof>

lemma *both-below-singles1*:

assumes *t* \sqsubseteq *singles* *S*

assumes *carrier* *t'* \cap *S* = {}

shows *t* $\otimes\otimes$ *t'* \sqsubseteq *singles* *S*

<proof>

lemma *paths-ttree-restr-singles*: *xs* \in *paths* (*ttree-restr* *S'* (*singles* *S*)) \longleftrightarrow *set xs* \subseteq *S'* \wedge ($\forall x \in S$. *one-call-in-path* *x xs*)

<proof>

lemma *substitute-not-carrier*:

assumes *x* \notin *carrier* *t*

assumes $\bigwedge x'$. *x* \notin *carrier* (*f* *x'*)

shows *x* \notin *carrier* (*substitute* *f* *T* *t*)

<proof>

lemma *substitute-below-singlesI*:
assumes $t \sqsubseteq \text{singles } S$
assumes $\bigwedge x. \text{carrier } (f x) \cap S = \{\}$
shows $\text{substitute } f T t \sqsubseteq \text{singles } S$
 $\langle \text{proof} \rangle$

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* :: $\text{var set} \Rightarrow (\text{AEnv} \times \text{CoCalls}) \Rightarrow (\text{AEnv} \times \text{CoCalls})$
where $\text{combined-restrict } S (\text{env}, G) = (\text{env } f|' S, \text{cc-restr } S G)$

lemma *fst-combined-restrict[simp]*:
 $\text{fst } (\text{combined-restrict } S p) = \text{fst } p f|' S$
 $\langle \text{proof} \rangle$

lemma *snd-combined-restrict[simp]*:
 $\text{snd } (\text{combined-restrict } S p) = \text{cc-restr } S (\text{snd } p)$
 $\langle \text{proof} \rangle$

lemma *combined-restrict-eqv[eqv]*:
shows $\pi \cdot \text{combined-restrict } S p = \text{combined-restrict } (\pi \cdot S) (\pi \cdot p)$
 $\langle \text{proof} \rangle$

lemma *combined-restrict-cont*:
 $\text{cont } (\lambda x. \text{combined-restrict } S x)$
 $\langle \text{proof} \rangle$

lemmas *cont-compose[OF combined-restrict-cont, cont2cont, simp]*

lemma *combined-restrict-perm*:
assumes $\text{supp } \pi \#* S$ **and** $[\text{simp}]$: $\text{finite } S$
shows $\text{combined-restrict } S (\pi \cdot p) = \text{combined-restrict } S p$
 $\langle \text{proof} \rangle$

definition *predCC* :: $\text{var set} \Rightarrow (\text{Arity} \rightarrow \text{CoCalls}) \Rightarrow (\text{Arity} \rightarrow \text{CoCalls})$
where $\text{predCC } S f = (\Lambda a. \text{if } a \neq 0 \text{ then } \text{cc-restr } S (f \cdot (\text{pred} \cdot a)) \text{ else } \text{ccSquare } S)$

lemma *predCC-eq*:
shows $\text{predCC } S f \cdot a = (\text{if } a \neq 0 \text{ then } \text{cc-restr } S (f \cdot (\text{pred} \cdot a)) \text{ else } \text{ccSquare } S)$

$\langle \text{proof} \rangle$

lemma *predCC-eqvt*[*eqvt*, *simp*]: $\pi \cdot (\text{predCC } S \ f) = \text{predCC } (\pi \cdot S) (\pi \cdot f)$
 $\langle \text{proof} \rangle$

lemma *cc-restr-predCC*:
 $\text{cc-restr } S (\text{predCC } S' \ f \cdot n) = (\text{predCC } (S' \cap S) (\Lambda \ n. \text{cc-restr } S (f \cdot n))) \cdot n$
 $\langle \text{proof} \rangle$

lemma *cc-restr-predCC'*[*simp*]:
 $\text{cc-restr } S (\text{predCC } S \ f \cdot n) = \text{predCC } S \ f \cdot n$
 $\langle \text{proof} \rangle$

nominal-function

$cCCexp :: exp \Rightarrow (Aarity \rightarrow AEnv \times CoCalls)$

where

$cCCexp (\text{Var } x) = (\Lambda \ n. (\text{esing } x \cdot (up \cdot n), \perp))$
 $| \ cCCexp (\text{Lam } [x]. \ e) = (\Lambda \ n. \text{combined-restrict } (fv (\text{Lam } [x]. \ e)) (fst (cCCexp \ e \cdot (\text{pred} \cdot n)), \text{predCC } (fv (\text{Lam } [x]. \ e)) (\Lambda \ a. \text{snd}(cCCexp \ e \cdot a)) \cdot n))$
 $| \ cCCexp (\text{App } e \ x) = (\Lambda \ n. (fst (cCCexp \ e \cdot (\text{inc} \cdot n)) \sqcup (\text{esing } x \cdot (up \cdot 0)), \text{snd } (cCCexp \ e \cdot (\text{inc} \cdot n)) \sqcup \text{ccProd } \{x\} (\text{insert } x (fv \ e))))$
 $| \ cCCexp (\text{Let } \Gamma \ e) = (\Lambda \ n. \text{combined-restrict } (fv (\text{Let } \Gamma \ e)) (\text{CoCallAarityAnalysis.cccFix-choose } cCCexp \ \Gamma \cdot (cCCexp \ e \cdot n)))$
 $| \ cCCexp (\text{Bool } b) = \perp$
 $| \ cCCexp (\text{scrut } ? \ e1 : \ e2) = (\Lambda \ n. (fst (cCCexp \ \text{scrut} \cdot 0) \sqcup fst (cCCexp \ e1 \cdot n) \sqcup fst (cCCexp \ e2 \cdot n), \text{snd } (cCCexp \ \text{scrut} \cdot 0) \sqcup (\text{snd } (cCCexp \ e1 \cdot n) \sqcup \text{snd } (cCCexp \ e2 \cdot n)) \sqcup \text{ccProd } (\text{edom } (fst (cCCexp \ \text{scrut} \cdot 0))) (\text{edom } (fst (cCCexp \ e1 \cdot n)) \cup \text{edom } (fst (cCCexp \ e2 \cdot n))))))$
 $\langle \text{proof} \rangle$

nominal-termination (*eqvt*) $\langle \text{proof} \rangle$

locale *CoCallAnalysisImpl*

begin

sublocale *CoCallAarityAnalysis* *cCCexp* $\langle \text{proof} \rangle$

sublocale *AarityAnalysis* *Aexp* $\langle \text{proof} \rangle$

abbreviation *Aexp-syn''* ($\mathcal{A} \cdot$) **where** $\mathcal{A}_a \ e \equiv Aexp \ e \cdot a$

abbreviation *Aexp-bot-syn''* ($\mathcal{A}^\perp \cdot$) **where** $\mathcal{A}^\perp_a \ e \equiv fup \cdot (Aexp \ e) \cdot a$

abbreviation *ccExp-syn''* ($\mathcal{G} \cdot$) **where** $\mathcal{G}_a \ e \equiv CCexp \ e \cdot a$

abbreviation *ccExp-bot-syn''* ($\mathcal{G}^\perp \cdot$) **where** $\mathcal{G}^\perp_a \ e \equiv fup \cdot (CCexp \ e) \cdot a$

lemma *cCCexp-eq*[*simp*]:

$cCCexp (\text{Var } x) \cdot n = (\text{esing } x \cdot (up \cdot n), \perp)$
 $cCCexp (\text{Lam } [x]. \ e) \cdot n = \text{combined-restrict } (fv (\text{Lam } [x]. \ e)) (fst (cCCexp \ e \cdot (\text{pred} \cdot n)), \text{predCC } (fv (\text{Lam } [x]. \ e)) (\Lambda \ a. \text{snd}(cCCexp \ e \cdot a)) \cdot n)$

$cCCexp (App\ e\ x) \cdot n = (fst\ (cCCexp\ e \cdot (inc \cdot n)) \sqcup (esing\ x \cdot (up \cdot 0))), \quad snd\ (cCCexp\ e \cdot (inc \cdot n)) \sqcup ccProd\ \{x\}\ (insert\ x\ (fv\ e))$
 $cCCexp (Let\ \Gamma\ e) \cdot n = combined-restrict\ (fv\ (Let\ \Gamma\ e))\ (CoCallArityAnalysis.cccFix-choose\ cCCexp\ \Gamma \cdot (cCCexp\ e \cdot n))$
 $cCCexp (Bool\ b) \cdot n = \perp$
 $cCCexp (scrut\ ?\ e1 : e2) \cdot n = (fst\ (cCCexp\ scrut \cdot 0) \sqcup fst\ (cCCexp\ e1 \cdot n) \sqcup fst\ (cCCexp\ e2 \cdot n)),$
 $snd\ (cCCexp\ scrut \cdot 0) \sqcup (snd\ (cCCexp\ e1 \cdot n) \sqcup snd\ (cCCexp\ e2 \cdot n)) \sqcup ccProd\ (edom\ (fst\ (cCCexp\ scrut \cdot 0)))\ (edom\ (fst\ (cCCexp\ e1 \cdot n)) \sqcup edom\ (fst\ (cCCexp\ e2 \cdot n)))$
 $\langle proof \rangle$
declare $cCCexp.simps[simp\ del]$

lemma *Aexp-pre-simps:*

$\mathcal{A}_a (Var\ x) = esing\ x \cdot (up \cdot a)$
 $\mathcal{A}_a (Lam\ [x].\ e) = Aexp\ e \cdot (pred \cdot a)\ f|' \cdot fv\ (Lam\ [x].\ e)$
 $\mathcal{A}_a (App\ e\ x) = Aexp\ e \cdot (inc \cdot a) \sqcup esing\ x \cdot (up \cdot 0)$
 $\neg nonrec\ \Gamma \implies$
 $\mathcal{A}_a (Let\ \Gamma\ e) = (Afix\ \Gamma \cdot (\mathcal{A}_a\ e \sqcup (\lambda \cdot up \cdot 0)\ f|' \cdot thanks\ \Gamma))\ f|' \cdot (fv\ (Let\ \Gamma\ e))$
 $x \notin fv\ e \implies$
 $\mathcal{A}_a (let\ x\ be\ e\ in\ exp) =$
 $(fup \cdot (Aexp\ e) \cdot (ABind-nonrec\ x\ e \cdot (\mathcal{A}_a\ exp,\ CCexp\ exp \cdot a)) \sqcup \mathcal{A}_a\ exp)$
 $f|' \cdot (fv\ (let\ x\ be\ e\ in\ exp))$
 $\mathcal{A}_a (Bool\ b) = \perp$
 $\mathcal{A}_a (scrut\ ?\ e1 : e2) = \mathcal{A}_0\ scrut \sqcup \mathcal{A}_a\ e1 \sqcup \mathcal{A}_a\ e2$
 $\langle proof \rangle$

lemma *CCexp-pre-simps:*

$CCexp (Var\ x) \cdot n = \perp$
 $CCexp (Lam\ [x].\ e) \cdot n = predCC\ (fv\ (Lam\ [x].\ e))\ (CCexp\ e) \cdot n$
 $CCexp (App\ e\ x) \cdot n = CCexp\ e \cdot (inc \cdot n) \sqcup ccProd\ \{x\}\ (insert\ x\ (fv\ e))$
 $\neg nonrec\ \Gamma \implies$
 $CCexp (Let\ \Gamma\ e) \cdot n = cc-restr\ (fv\ (Let\ \Gamma\ e))$
 $(CCfix\ \Gamma \cdot (Afix\ \Gamma \cdot (Aexp\ e \cdot n \sqcup (\lambda \cdot up \cdot 0)\ f|' \cdot thanks\ \Gamma),\ CCexp\ e \cdot n))$
 $x \notin fv\ e \implies CCexp (let\ x\ be\ e\ in\ exp) \cdot n =$
 $cc-restr\ (fv\ (let\ x\ be\ e\ in\ exp))$
 $(ccBind\ x\ e \cdot (Aheap-nonrec\ x\ e \cdot (Aexp\ exp \cdot n,\ CCexp\ exp \cdot n),\ CCexp\ exp \cdot n)$
 $\sqcup ccProd\ (fv\ e)\ (ccNeighbors\ x\ (CCexp\ exp \cdot n) - (if\ isVal\ e\ then\ \{\}\ else\ \{x\})) \sqcup CCexp\ exp \cdot n)$
 $CCexp (Bool\ b) \cdot n = \perp$
 $CCexp (scrut\ ?\ e1 : e2) \cdot n =$
 $CCexp\ scrut \cdot 0 \sqcup$
 $(CCexp\ e1 \cdot n \sqcup CCexp\ e2 \cdot n) \sqcup$
 $ccProd\ (edom\ (Aexp\ scrut \cdot 0))\ (edom\ (Aexp\ e1 \cdot n) \sqcup edom\ (Aexp\ e2 \cdot n))$
 $\langle proof \rangle$

lemma

shows $ccField-CCexp: ccField\ (CCexp\ e \cdot a) \subseteq fv\ e$ **and** $Aexp-edom': edom\ (\mathcal{A}_a\ e) \subseteq fv\ e$

<proof>

lemma *cc-restr-CCexp[simp]*:

$$cc\text{-restr } (fv\ e) (CCexp\ e\cdot a) = CCexp\ e\cdot a$$

<proof>

lemma *ccField-fup-CCexp*:

$$ccField\ (fup\cdot(CCexp\ e)\cdot n) \subseteq fv\ e$$

<proof>

lemma *cc-restr-fup-ccExp-useless[simp]*: $cc\text{-restr } (fv\ e) (fup\cdot(CCexp\ e)\cdot n) = fup\cdot(CCexp\ e)\cdot n$

<proof>

sublocale *EdomArityAnalysis Aexp* *<proof>*

lemma *CCexp-simps[simp]*:

$$\mathcal{G}_a(\text{Var } x) = \perp$$

$$\mathcal{G}_0(\text{Lam } [x].\ e) = (fv\ (\text{Lam } [x].\ e))^2$$

$$\mathcal{G}_{inc\cdot a}(\text{Lam } [x].\ e) = cc\text{-delete } x\ (\mathcal{G}_a\ e)$$

$$\mathcal{G}_a(\text{App } e\ x) = \mathcal{G}_{inc\cdot a}\ e \sqcup \{x\}\ G \times insert\ x\ (fv\ e)$$

$$\neg\ nonrec\ \Gamma \implies \mathcal{G}_a(\text{Let } \Gamma\ e) =$$

$$(CCfix\ \Gamma\cdot(Afix\ \Gamma\cdot(\mathcal{A}_a\ e \sqcup (\lambda\cdot up\cdot 0)\ f|'\ \text{thunks } \Gamma), \mathcal{G}_a\ e))\ G|'\ (-\ domA\ \Gamma)$$

$$x \notin fv\ e' \implies \mathcal{G}_a(\text{let } x\ \text{be } e'\ \text{in } e) =$$

$$cc\text{-delete } x$$

$$(ccBind\ x\ e'\cdot(Aheap\text{-nonrec } x\ e'\cdot(\mathcal{A}_a\ e, \mathcal{G}_a\ e), \mathcal{G}_a\ e)$$

$$\sqcup fv\ e'\ G \times (ccNeighbors\ x\ (\mathcal{G}_a\ e) - (\text{if } isVal\ e'\ \text{then } \{\} \ \text{else } \{x\})) \sqcup \mathcal{G}_a\ e)$$

$$\mathcal{G}_a(\text{Bool } b) = \perp$$

$$\mathcal{G}_a(\text{scrut } ?\ e1 : e2) =$$

$$\mathcal{G}_0\ \text{scrut} \sqcup (\mathcal{G}_a\ e1 \sqcup \mathcal{G}_a\ e2) \sqcup$$

$$edom\ (\mathcal{A}_0\ \text{scrut})\ G \times (edom\ (\mathcal{A}_a\ e1) \cup edom\ (\mathcal{A}_a\ e2))$$

<proof>

definition *Aheap where*

$Aheap\ \Gamma\ e = (\Lambda\ a.\ \text{if } nonrec\ \Gamma\ \text{then } (case\text{-prod } Aheap\text{-nonrec } (hd\ \Gamma))\cdot(Aexp\ e\cdot a, CCexp\ e\cdot a)$
 $\text{else } (Afix\ \Gamma\cdot(Aexp\ e\cdot a \sqcup (\lambda\cdot up\cdot 0)\ f|'\ \text{thunks } \Gamma))\ f|'\ domA\ \Gamma)$

lemma *Aheap-simp1[simp]*:

$$\neg\ nonrec\ \Gamma \implies Aheap\ \Gamma\ e\cdot a = (Afix\ \Gamma\cdot(Aexp\ e\cdot a \sqcup (\lambda\cdot up\cdot 0)\ f|'\ \text{thunks } \Gamma))\ f|'\ domA\ \Gamma$$

<proof>

lemma *Aheap-simp2[simp]*:

$$x \notin fv\ e' \implies Aheap\ [(x, e')]\ e\cdot a = Aheap\text{-nonrec } x\ e'\cdot(Aexp\ e\cdot a, CCexp\ e\cdot a)$$

<proof>

lemma *Aheap-eqvt'[eqvt]*:

$$\pi\cdot(Aheap\ \Gamma\ e) = Aheap\ (\pi\cdot\Gamma)\ (\pi\cdot e)$$

<proof>

sublocale *ArityAnalysisHeap Aheap* *<proof>*

sublocale *ArityAnalysisHeapEqvt Aheap*

<proof>

lemma *Aexp-lam-simp*: $Aexp (Lam [x]. e) \cdot n = env\text{-delete } x (Aexp e \cdot (pred \cdot n))$

<proof>

lemma *Aexp-Let-simp1*:

$\neg nonrec \Gamma \implies \mathcal{A}_a (Let \Gamma e) = (Afix \Gamma \cdot (\mathcal{A}_a e \sqcup (\lambda \cdot up \cdot 0) f | 'thunks \Gamma)) f | '(- domA \Gamma)$

<proof>

lemma *Aexp-Let-simp2*:

$x \notin fv e \implies \mathcal{A}_a (let x be e in exp) = env\text{-delete } x (\mathcal{A}^\perp ABind\text{-nonrec } x e \cdot (\mathcal{A}_a exp, CCexp exp \cdot a) e \sqcup \mathcal{A}_a exp)$

<proof>

lemma *Aexp-simps[simp]*:

$\mathcal{A}_a (Var x) = esing x \cdot (up \cdot a)$

$\mathcal{A}_a (Lam [x]. e) = env\text{-delete } x (\mathcal{A}_{pred \cdot a} e)$

$\mathcal{A}_a (App e x) = Aexp e \cdot (inc \cdot a) \sqcup esing x \cdot (up \cdot 0)$

$\neg nonrec \Gamma \implies \mathcal{A}_a (Let \Gamma e) =$

$(Afix \Gamma \cdot (\mathcal{A}_a e \sqcup (\lambda \cdot up \cdot 0) f | 'thunks \Gamma)) f | '(- domA \Gamma)$

$x \notin fv e' \implies \mathcal{A}_a (let x be e' in e) =$

$env\text{-delete } x (\mathcal{A}^\perp ABind\text{-nonrec } x e' \cdot (\mathcal{A}_a e, \mathcal{G}_a e) e' \sqcup \mathcal{A}_a e)$

$\mathcal{A}_a (Bool b) = \perp$

$\mathcal{A}_a (scrut ? e1 : e2) = \mathcal{A}_0 scrut \sqcup \mathcal{A}_a e1 \sqcup \mathcal{A}_a e2$

<proof>

end

end

13.2 CoCallImplSafe

theory *CoCallImplSafe*

imports *CoCallAnalysisImpl CoCallAnalysisSpec ArityAnalysisFixProps*

begin

locale *CoCallImplSafe*

begin

sublocale *CoCallAnalysisImpl* *<proof>*

lemma *ccNeighbors-Int-ccrestr*: $(ccNeighbors x G \cap S) = ccNeighbors x (cc\text{-restr } (insert x S) G) \cap S$

<proof>

lemma

assumes $x \notin S$ **and** $y \notin S$

shows $CCexp\text{-subst}: cc\text{-restr } S (CCexp\ e[y::=x]\cdot a) = cc\text{-restr } S (CCexp\ e\cdot a)$

and $Aexp\text{-restr}\text{-subst}: (Aexp\ e[y::=x]\cdot a)\ f|' S = (Aexp\ e\cdot a)\ f|' S$

<proof>

sublocale $ArityAnalysisSafe\ Aexp$

<proof>

sublocale $ArityAnalysisLetSafe\ Aexp\ Ahead$

<proof>

definition $ccHeap\text{-nonrec}$

where $ccHeap\text{-nonrec}\ x\ e\ exp = (\Lambda\ n.\ CCfix\text{-nonrec}\ x\ e\cdot (Aexp\ exp\cdot n,\ CCexp\ exp\cdot n))$

lemma $ccHeap\text{-nonrec}\text{-eq}$:

$ccHeap\text{-nonrec}\ x\ e\ exp\cdot n = CCfix\text{-nonrec}\ x\ e\cdot (Aexp\ exp\cdot n,\ CCexp\ exp\cdot n)$

<proof>

definition $ccHeap\text{-rec} :: heap \Rightarrow exp \Rightarrow Arity \rightarrow CoCalls$

where $ccHeap\text{-rec}\ \Gamma\ e = (\Lambda\ a.\ CCfix\ \Gamma\cdot (Afix\ \Gamma\cdot (Aexp\ e\cdot a \sqcup (\lambda\cdot up\cdot 0)\ f|' (thunks\ \Gamma)), CCexp\ e\cdot a))$

lemma $ccHeap\text{-rec}\text{-eq}$:

$ccHeap\text{-rec}\ \Gamma\ e\cdot a = CCfix\ \Gamma\cdot (Afix\ \Gamma\cdot (Aexp\ e\cdot a \sqcup (\lambda\cdot up\cdot 0)\ f|' (thunks\ \Gamma)), CCexp\ e\cdot a)$

<proof>

definition $ccHeap :: heap \Rightarrow exp \Rightarrow Arity \rightarrow CoCalls$

where $ccHeap\ \Gamma = (if\ nonrec\ \Gamma\ then\ case\text{-prod}\ ccHeap\text{-nonrec}\ (hd\ \Gamma)\ else\ ccHeap\text{-rec}\ \Gamma)$

lemma $ccHeap\text{-simp1}$:

$\neg nonrec\ \Gamma \Longrightarrow ccHeap\ \Gamma\ e\cdot a = CCfix\ \Gamma\cdot (Afix\ \Gamma\cdot (Aexp\ e\cdot a \sqcup (\lambda\cdot up\cdot 0)\ f|' (thunks\ \Gamma)), CCexp\ e\cdot a)$

<proof>

lemma $ccHeap\text{-simp2}$:

$x \notin fv\ e \Longrightarrow ccHeap\ [(x,e)]\ exp\cdot n = CCfix\text{-nonrec}\ x\ e\cdot (Aexp\ exp\cdot n,\ CCexp\ exp\cdot n)$

<proof>

sublocale $CoCallAritySafe\ CCexp\ Aexp\ ccHeap\ Ahead$

<proof>

end

end

14 End-to-end Saftey Results and Example

14.1 CallAriyEnd2End

```
theory CallAriyEnd2End
imports ArityTransform CoCallAnalysisImpl
begin

locale CallAriyEnd2End
begin
sublocale CoCallAnalysisImpl<proof>

lemma fresh-var-eqE[elim-format]: fresh-var e = x  $\implies$  x  $\notin$  fv e
  <proof>

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:  $\bigwedge a. CCexp\ e \cdot a = \perp$ 
  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  $\#$  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>

end
end
```

14.2 CallAriyEnd2EndSafe

```
theory CallAriyEnd2EndSafe
imports CallAriyEnd2End CardAriyTransformSafe CoCallImplSafe CoCallImplTTreeSafe TTreeImplCardinalitySafe
begin

locale CallAriyEnd2EndSafe
begin
sublocale CoCallImplSafe<proof>
sublocale CallAriyEnd2End<proof>

abbreviation transform-syn' ( $\mathcal{T}$ .) where  $\mathcal{T}_a \equiv transform\ a$ 

lemma end2end:
  c  $\Rightarrow^*$  c'  $\implies$ 
   $\neg boring\ step\ c' \implies$ 
```

$heap\text{-}upds\text{-}ok\text{-}conf\ c \implies$
 $consistent\ (ae, ce, a, as, r)\ c \implies$
 $\exists ae'\ ce'\ a'\ as'\ r'.\ consistent\ (ae', ce', a', as', r')\ c' \wedge conf\text{-}transform\ (ae, ce, a, as, r)\ c \Rightarrow_G^*$
 $conf\text{-}transform\ (ae', ce', a', as', r')\ c'$
 <proof>

theorem *end2end-closed*:

assumes *closed*: $fv\ e = (\{\} :: var\ set)$
assumes $(\[], e, \[])\ \Rightarrow^*\ (\Gamma, v, \[])$ **and** *isVal* v
obtains Γ' **and** v'
where $(\[], \mathcal{T}_0\ e, \[])\ \Rightarrow^*\ (\Gamma', v', \[])$ **and** *isVal* v'
and $card\ (domA\ \Gamma') \leq card\ (domA\ \Gamma)$

<proof>

lemma *fresh-var-eqE[elim-format]*: $fresh\text{-}var\ e = x \implies x \notin fv\ e$

<proof>

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*: $\bigwedge a.\ CCexp\ e \cdot a = \perp$
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*: $atom\ z \not\# e$
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>

end

end

15 Functional Correctness of the Arity Analysis

15.1 ArityAnalysisCorrDenotational

theory *ArityAnalysisCorrDenotational*

imports *ArityAnalysisSpec Launchbury.Denotational ArityTransform*

begin

context *ArityAnalysisLetSafe*

begin

inductive *eq* :: *Arity* \Rightarrow *Value* \Rightarrow *Value* \Rightarrow *bool* **where**

$eq\ 0\ v\ v$

| $(\bigwedge v.\ eq\ n\ (v1\ \downarrow Fn\ v)\ (v2\ \downarrow Fn\ v)) \implies eq\ (inc \cdot n)\ v1\ v2$

lemma [simp]: $eq\ 0\ v\ v' \longleftrightarrow v = v'$
⟨proof⟩

lemma eq-inc-simp:
 $eq\ (inc \cdot n)\ v1\ v2 \longleftrightarrow (\forall\ v.\ eq\ n\ (v1\ \downarrow Fn\ v)\ (v2\ \downarrow Fn\ v))$
⟨proof⟩

lemma eq-FnI:
 $(\bigwedge\ v.\ eq\ (pred \cdot n)\ (f1 \cdot v)\ (f2 \cdot v)) \implies eq\ n\ (Fn \cdot f1)\ (Fn \cdot f2)$
⟨proof⟩

lemma eq-refl[simp]: $eq\ a\ v\ v$
⟨proof⟩

lemma eq-trans[trans]: $eq\ a\ v1\ v2 \implies eq\ a\ v2\ v3 \implies eq\ a\ v1\ v3$
⟨proof⟩

lemma eq-Fn: $eq\ a\ v1\ v2 \implies eq\ (pred \cdot a)\ (v1\ \downarrow Fn\ v)\ (v2\ \downarrow Fn\ v)$
⟨proof⟩

lemma eq-inc-same: $eq\ a\ v1\ v2 \implies eq\ (inc \cdot a)\ v1\ v2$
⟨proof⟩

lemma eq-mono: $a \sqsubseteq a' \implies eq\ a'\ v1\ v2 \implies eq\ a\ v1\ v2$
⟨proof⟩

lemma eq-join[simp]: $eq\ (a \sqcup a')\ v1\ v2 \longleftrightarrow eq\ a\ v1\ v2 \wedge eq\ a'\ v1\ v2$
⟨proof⟩

lemma eq-adm: $cont\ f \implies cont\ g \implies adm\ (\lambda\ x.\ eq\ a\ (f\ x)\ (g\ x))$
⟨proof⟩

inductive eqQ :: $AEnv \Rightarrow (var \Rightarrow Value) \Rightarrow (var \Rightarrow Value) \Rightarrow bool$ **where**
eqQI: $(\bigwedge\ x\ a.\ ae\ x = up \cdot a \implies eq\ a\ (\rho1\ x)\ (\rho2\ x)) \implies eqQ\ ae\ \rho1\ \rho2$

lemma eqQE: $eqQ\ ae\ \rho1\ \rho2 \implies ae\ x = up \cdot a \implies eq\ a\ (\rho1\ x)\ (\rho2\ x)$
⟨proof⟩

lemma eqQ-refl[simp]: $eqQ\ ae\ \rho\ \rho$
⟨proof⟩

lemma eq-esing-up[simp]: $eqQ\ (esing\ x \cdot (up \cdot a))\ \rho1\ \rho2 \longleftrightarrow eq\ a\ (\rho1\ x)\ (\rho2\ x)$
⟨proof⟩

lemma eqQ-mono:
assumes $ae \sqsubseteq ae'$
assumes $eqQ\ ae'\ \rho1\ \rho2$
shows $eqQ\ ae\ \rho1\ \rho2$

<proof>

lemma *eq ρ -adm*: $cont\ f \implies cont\ g \implies adm\ (\lambda\ x.\ eq\ \rho\ a\ (f\ x)\ (g\ x))$

<proof>

lemma *up-join-eq-up[simp]*: $up.(n::'a::Finite-Join-cpo) \sqcup up.n' = up.(n \sqcup n')$

<proof>

lemma *eq ρ -join[simp]*: $eq\ \rho\ (ae \sqcup ae')\ \rho1\ \rho2 \iff eq\ \rho\ ae\ \rho1\ \rho2 \wedge eq\ \rho\ ae'\ \rho1\ \rho2$

<proof>

lemma *eq ρ -override[simp]*:

$eq\ \rho\ ae\ (\rho1\ ++_S\ \rho2)\ (\rho1'\ ++_S\ \rho2') \iff eq\ \rho\ ae\ (\rho1\ f|'(-S))\ (\rho1'\ f|'(-S)) \wedge eq\ \rho\ ae\ (\rho2\ f|'S)\ (\rho2'\ f|'S)$

<proof>

lemma *Aexp-heap-below-Aheap*:

assumes $(Aheap\ \Gamma\ e.a)\ x = up.a'$

assumes $map-of\ \Gamma\ x = Some\ e'$

shows $Aexp\ e'.a' \sqsubseteq Aheap\ \Gamma\ e.a \sqcup Aexp\ (Let\ \Gamma\ e).a$

<proof>

lemma *Aexp-body-below-Aheap*:

shows $Aexp\ e.a \sqsubseteq Aheap\ \Gamma\ e.a \sqcup Aexp\ (Let\ \Gamma\ e).a$

<proof>

lemma *Aexp-correct*: $eq\ \rho\ (Aexp\ e.a)\ \rho1\ \rho2 \implies eq\ a\ (\llbracket e \rrbracket_{\rho1})\ (\llbracket e \rrbracket_{\rho2})$

<proof>

lemma *ESem-ignores-fresh[simp]*: $\llbracket e \rrbracket_{\rho}(fresh-var\ e := v) = \llbracket e \rrbracket_{\rho}$

<proof>

lemma *eq-Aeta-expand*: $eq\ a\ (\llbracket Aeta-expand\ a\ e \rrbracket_{\rho})\ (\llbracket e \rrbracket_{\rho})$

<proof>

lemma *Arity-transformation-correct*: $eq\ a\ (\llbracket \mathcal{T}_a\ e \rrbracket_{\rho})\ (\llbracket e \rrbracket_{\rho})$

<proof>

corollary *Arity-transformation-correct'*:

$\llbracket \mathcal{T}_0\ e \rrbracket_{\rho} = \llbracket e \rrbracket_{\rho}$

<proof>

end

end