The Safety of Call Arity

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

August 16, 2018

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 high-level 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 concepts in this development.

<table>
<thead>
<tr>
<th>Concept</th>
<th>corresponds to</th>
<th>in theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>nominal-datatype expr</td>
<td>Terms in [Bre13]</td>
</tr>
<tr>
<td>Stack</td>
<td>type-synonym stack</td>
<td>SestoftConf</td>
</tr>
<tr>
<td>Configuration</td>
<td>type-synonym conf</td>
<td>SestoftConf</td>
</tr>
<tr>
<td>Semantics (⇒)</td>
<td>inductive step</td>
<td>Sestoft</td>
</tr>
<tr>
<td>Arity</td>
<td>typedef Arity</td>
<td>Arity</td>
</tr>
<tr>
<td>Eta-expansion</td>
<td>lift-definition Aeta-expand</td>
<td>ArityEtaExpansion</td>
</tr>
</tbody>
</table>
Lemma 1 theorem A eta-exp and-safe
\[ A_\alpha(\Gamma, e) \]
locale ArityAnalysisHeap
locale ArityAnalysisSig

Definition 2 locale ArityAnalysisLetSafe
Definition 3 locale ArityAnalysisLetSafeNoCard
Definition 4 inductive a-consistent
Definition 5 inductive consistent
Lemma 2 lemma arity-transform-safe
Card
type-synonym two
\[ C_\alpha(\Gamma, e) \]
locale CardinalityHeap
locale CardinalityAnalysisSig
Definition 6 locale CardinalityAnalysisSpec
Definition 7 \( \Rightarrow \# \) inductive gc-step
Definition 8 inductive consistent
Lemma 3 lemma card-arity-transform-safe
Trace trees typedef 'a three
Function s lift-definition substitute
\[ \mathcal{T}_\alpha(e) \]
locale TTreeAnalysis
locale TTreeAnalysisSig
locale TTreeAnalysisSpec
Definition 9 locale TTreeAnalysisCardinalityHeap
Definition 10 sublocale CardinalityAnalysisSpec
Lemma 4 sublocale TTreeAnalysisCardinalityHeap
Co-Call graphs typedef CoCalls
Function g lift-definition ccApprox
Function t lift-definition ccTTree
\[ \mathcal{G}_\alpha(e) \]
locale CoCallAnalysis
locale CoCallAnalysisSig
locale CoCallAnalysisSpec
Definition 10 locale CoCallAritySafe
Lemma 5 sublocale TTreeAnalysisCardinalityHeap
Call Arity nominal-function cCCexp
Theorem 1 lemma end2end-closed

References


Contents

1 Various Utilities ........................................ 5
   1.1 ConstOn ........................................ 5
   1.2 Set-Cpo ........................................ 6
   1.3 Env-Set-Cpo ..................................... 7
   1.4 AList-Utils-HOLCF .............................. 8
   1.5 List-Interleavings ............................... 9

2 Small-step Semantics ................................... 10
   2.1 Sestoft Conf ..................................... 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 ......................... 55
  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 .................................... 73
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 .......................... 95
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 Safety 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 \( \text{const-onI} \)\[\text{intro}\]: \( (\forall y \in S \implies f y = x) \implies \text{const-on} f S x \)

\[\langle \text{proof} \rangle \]

lemma \( \text{const-onD} \)\[\text{dest}\]: \( \text{const-on} f S x \implies y \in S \implies f y = x \)

\[\langle \text{proof} \rangle \]

lemma \( \text{const-on-insert} \)\[\text{simp}\]: \( \text{const-on} f (\text{insert} x S) y \iff \text{const-on} f S y \land f x = y \)

\[\langle \text{proof} \rangle \]

lemma \( \text{const-on-union} \)\[\text{simp}\]: \( \text{const-on} f (S \cup S') y \iff \text{const-on} f S y \land \text{const-on} f S' y \)

\[\langle \text{proof} \rangle \]

lemma \( \text{const-on-subset} \)\[\text{elim}\]: \( \text{const-on} f S y \implies S' \subseteq S \implies \text{const-on} f S' y \)

\[\langle \text{proof} \rangle \]

end

1.2 Set-Cpo

thory Set-Cpo
imports HOLCF
begin

default-sort type

instantiation set :: (type) below
begin
  definition below-set where \( (\subseteq) = (\subseteq) \)
  instance \[\langle \text{proof} \rangle \]
end

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

lemma is-lub-set:
\( S \ll \bigcup S \)
\[\langle \text{proof} \rangle \]

lemma lub-set: lub \( S = \bigcup S \)
\[\langle \text{proof} \rangle \]

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

6
lemma minimal-set: \{\} \subseteq S
  ⟨proof⟩

instance set :: (type) pcpo
  ⟨proof⟩

lemma set-contI:
  assumes \( \bigwedge Y. \text{chain } Y \implies f (\bigcup i. Y i) = \bigcup (f \cdot \text{range } Y) \)
  shows cont f
  ⟨proof⟩

lemma set-set-contI:
  assumes \( \bigwedge S. f (\bigcup S) = \bigcup (f \cdot S) \)
  shows cont f
  ⟨proof⟩

lemma adm-subseteq[simp]:
  assumes cont f
  shows adm (λa. f a \subseteq S)
  ⟨proof⟩

lemma adm-Ball[simp]; adm (λS. \forall x \in S. P x)
  ⟨proof⟩

lemma finite-subset-chain:
  fixes Y :: nat ⇒ 'a set
  assumes chain Y
  assumes S \subseteq UNION UNIV Y
  assumes finite S
  shows \( \exists i. S \subseteq Y i \)
  ⟨proof⟩

lemma diff-cont[THEN cont-compose, simp, cont2cont]:
  fixes S' :: 'a set
  shows cont (λS. 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 (λ f. edom f)
  ⟨proof⟩
1.4 AList-Utils-HOLCF

theory AList-Utils-HOLCF
imports Launchbury.HOLCF-Utils Launchbury.HOLCF-Join-Classes Launchbury.AList-Utils
begin

syntax
-BLubMap :: [pttrn, pttrn, 'a → 'b, 'b] ⇒ 'b ((3\Unleftarrow{}/-/-/-/∈/-/-) [0,0,0, 10] 10)

translations
\Unleftarrow{} k\rightarrow{}v\in{}m. e == CONST lub (CONST mapCollect (λk v . e) m)

lemma below-lubmapI[intro]:
m k = Some v \implies (e k v ::'a::Join-cpo) ⊑ (\Unleftarrow{} k\rightarrow{}v\in{}m. e k v)
⟨proof⟩

lemma lubmap-belowI[intro]:
(∀ k v . m k = Some v \implies (e k v ::'a::Join-cpo) ⊑ u) \implies (\Unleftarrow{} k\rightarrow{}v\in{}m. e k v) ⊑ u
⟨proof⟩

lemma lubmap-const-bottom[simp]:
(\Unleftarrow{} k\rightarrow{}v\in{}m. ⊥) = (⊥::'a::Join-cpo)
⟨proof⟩

lemma lubmap-map-upd[simp]:
fixes e :: 'a ⇒ 'b ⇒ ('c :: Join-cpo)
shows (\Unleftarrow{} k\rightarrow{}v\in{}m(k'\rightarrow{}v'). e k v) = e k' v' ⊔ (\Unleftarrow{} k\rightarrow{}v\in{}m(k':=None). e k v)
⟨proof⟩

lemma lubmap-below-cong:
assumes \wedge k v . m k = Some v \implies f1 k v ⊑ (f2 k v :: 'a :: Join-cpo)
shows (\Unleftarrow{} k\rightarrow{}v\in{}m. f1 k v) ⊑ (\Unleftarrow{} k\rightarrow{}v\in{}m. f2 k v)
⟨proof⟩

lemma cont2cont-lubmap[simp, cont2cont]:
assumes (\\wedge k v . cont (f k v))
shows cont (λx. \Unleftarrow{} k\rightarrow{}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 ⇒ 'a list ⇒ 'a list ⇒ bool
  where [simp]: interleave' [] [] []
    | interleave' xs ys zs ⇒ interleave' (x # xs) ys (x # zs)
    | interleave' xs ys zs ⇒ interleave' xs (y # ys) (x # zs)

definition interleave :: 'a list ⇒ 'a list ⇒ 'a list set (infixr ⊗)
  where xs ⊗ ys = Collect (interleave' xs ys)

lemma elim-interleave [pred-set-conv]: interleave' xs ys zs ⇔ zs ∈ xs ⊗ 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) ∈ ys ⊗ zs
inductive-cases interleave-ConsConsE[elim]: zss ∈ y # ys ⊗ x # zs
inductive-cases interleave-ConsE2[elim]: xs ∈ y # ys ⊗ z
inductive-cases interleave-ConsE3[elim]: xs ∈ ys ⊗ x

lemma interleave-comm: xs ∈ ys ⊗ zs ⇒ zs ∈ xs ⊗ ys ⟨proof⟩

lemma interleave-Nil1 [simp]: [] ⊗ xs = {xs}
⟨proof⟩

lemma interleave-Nil2 [simp]: xs ⊗ [] = {xs}
⟨proof⟩

lemma interleave-nil-simp [simp]: [] ∈ xs ⊗ ys ⇔ xs = [] ∧ ys = []
⟨proof⟩

lemma append-interleave: xs @ ys ∈ xs ⊗ ys
⟨proof⟩

lemma interleave-assoc1: a ∈ xs ⊗ ys ⇒ b ∈ a ⊗ zs ⇒ ∃ c. c ∈ ys ⊗ zs ∧ b ∈ xs ⊗ c
⟨proof⟩

lemma interleave-assoc2: a ∈ ys ⊗ zs ⇒ b ∈ xs ⊗ a ⇒ ∃ c. c ∈ xs ⊗ ys ∧ b ∈ c ⊗ zs
⟨proof⟩

lemma interleave-set: zs ∈ xs ⊗ ys ⇒ set zs = set xs ∪ set ys

9
lemma interleave-tl: \( xs \in ys \otimes zs \implies \text{tl } xs \in \text{tl } ys \otimes zs \lor \text{tl } xs \in ys \otimes (\text{tl } zs) \)

(\text{proof})

lemma interleave-butlast: \( xs \in ys \otimes zs \implies \text{butlast } xs \in \text{butlast } ys \otimes zs \lor \text{butlast } xs \in ys \otimes (\text{butlast } zs) \)

(\text{proof})

lemma interleave-take: \( zs \in xs \otimes ys \implies \exists n_1 n_2. n = n_1 + n_2 \land \text{take } n zs \in \text{take } n_1 xs \otimes \text{take } n_2 ys \)

(\text{proof})

lemma filter-interleave: \( xs \in ys \otimes zs \implies \text{filter } P xs \in \text{filter } P ys \otimes \text{filter } P zs \)

(\text{proof})

lemma interleave-filtered: \( xs \in \text{interleave } (\text{filter } P xs) \) (\( \text{filter } (\lambda x'. \neg P x') xs \))

(\text{proof})

function \text{foo} where

\[ \text{foo } \[] \[] = \text{undefined} \]
\[ \text{foo } xs \[] = \text{undefined} \]
\[ \text{foo } \[] ys = \text{undefined} \]
\[ \text{foo } (x\#xs) (y\#ys) = \text{undefined } (\text{foo } xs (y\#ys)) (\text{foo } (x\#xs) ys) \]

(\text{proof})

termination (\text{proof})

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

lemma interleave-filter:

assumes \( xs \in \text{filter } P ys \otimes \text{filter } P zs \)

obtains \( xs' \text{ where } xs' \in ys \otimes zs \text{ and } xs = \text{filter } P xs' \)

(\text{proof})

end

2 Small-step Semantics

2.1 SestoftConf

theory SestoftConf
imports Launchbury.Terms Launchbury.Substitution
begin

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

instantiation stack-elem :: pt
begin

definition \(\pi \cdot x = (\text{case } x \text{ of } \text{Alts } e1 e2) \Rightarrow \text{Alts } (\pi \cdot e1) (\pi \cdot e2) | (\text{Arg } v) \Rightarrow \text{Arg } (\pi \cdot v) | (\text{Upd } v) \Rightarrow \text{Upd } (\pi \cdot v) | (\text{Dummy } v) \Rightarrow \text{Dummy } (\pi \cdot v)\)

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 \not\in \text{Alts } e1 e2 = (a \not\in e1 \land a \not\in e2) \ \langle \text{proof} \rangle

lemma \text{fresh-star-Alts}[\text{simp}]: a ^* \text{Alts } e1 e2 = (a ^* e1 \land a ^* e2) \ \langle \text{proof} \rangle

lemma \text{fresh-Arg}[\text{simp}]: a \not\in \text{Arg } v = a \not\in v \ \langle \text{proof} \rangle

lemma \text{fresh-Upd}[\text{simp}]: a \not\in \text{Upd } v = a \not\in v \ \langle \text{proof} \rangle

lemma \text{fresh-Dummy}[\text{simp}]: a \not\in \text{Dummy } v = a \not\in 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} \ \text{where}

\text{ap} [] = {}

| \text{ap } (\text{Alts } e1 e2 \# S) = \text{ap } S

| \text{ap } (\text{Arg } x \# S) = \text{insert } x (\text{ap } S)

| \text{ap } (\text{Upd } x \# S) = \text{ap } S

| \text{ap } (\text{Dummy } x \# S) = \text{ap } S

fun \text{upds} :: \text{stack} \Rightarrow \text{var set} \ \text{where}

\text{upds} [] = {}

| \text{upds } (\text{Alts } e1 e2 \# S) = \text{upds } S

| \text{upds } (\text{Upd } x \# S) = \text{insert } x (\text{upds } S)

| \text{upds } (\text{Arg } x \# S) = \text{upds } S

| \text{upds } (\text{Dummy } x \# S) = \text{upds } S

fun \text{dummies} :: \text{stack} \Rightarrow \text{var set} \ \text{where}

dummies [] = {}

| \text{dummies } (\text{Alts } e1 e2 \# S) = \text{dummies } S

| \text{dummies } (\text{Upd } x \# S) = \text{dummies } S

| \text{dummies } (\text{Arg } x \# S) = \text{dummies } S


dummies (Dummy x # S) = insert x (dummies S)

fun flattn :: stack ⇒ var list where
  flattn [] = []
  flattn (Alts e1 e2 # S) = fv-list e1 @ fv-list e2 @ flattn S
  flattn (Upd x # S) = x # flattn S
  flattn (Arg x # S) = x # flattn S
  flattn (Dummy x # S) = x # flattn S

fun upds-list :: stack ⇒ var list where
  upds-list [] = []
  upds-list (Alts e1 e2 # S) = upds-list S
  upds-list (Upd x # S) = x # upds-list S
  upds-list (Arg x # S) = upds-list S
  upds-list (Dummy x # S) = upds-list S

lemma set-upds-list[simp]:
  set (upds-list S) = upds S
  ⟨proof⟩

lemma upds-fv-subset: upds S ⊆ fv S
  ⟨proof⟩

lemma fresh-distinct-ups: atom ‘ V #* S ⇒ V ∩ upds S = {}
  ⟨proof⟩

lemma ap-fv-subset: ap S ⊆ fv S
  ⟨proof⟩

lemma dummies-fv-subset: dummies S ⊆ fv S
  ⟨proof⟩

lemma fresh-flattn[simp]: atom (a::var) #* flattn S ←→ atom a #* S
  ⟨proof⟩

lemma fresh-star-flattn[simp]: atom ‘ (as::var set) #* flattn S ←→ atom ‘ as #* S
  ⟨proof⟩

lemma fresh-upds-list[simp]: atom a #* S ⇒ atom (a::var) #* upds-list S
  ⟨proof⟩

lemma fresh-star-upds-list[simp]: atom ‘ (as::var set) #* S ⇒ atom ‘ (as::var set) #* upds-list S
  ⟨proof⟩

lemma upds-append[simp]: upds (S # S') = upds S ∪ 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 ∪ 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' ←→ l = l'
⟨proof⟩

Type-synonym conf = (heap × exp × stack)

Inductive boring-step where
  isVal e ⇒ boring-step (Γ, e, Upd x # S)

Fun restr-stack :: var set ⇒ stack ⇒ stack
  where restr-stack V [] = []
      | restr-stack V (Alls e1 e2 # S) = Alls e1 e2 # restr-stack V S
      | restr-stack V (Arg x # S) = Arg x # restr-stack V S
      | restr-stack V (Upd x # S) = (if x ∈ V then Upd x # restr-stack V S else restr-stack V S)
  restr-stack V (Dummy x # S) = Dummy x # restr-stack V S

Lemma restr-stack-cong:
  (∀ x. x ∈ upds S ⇒ x ∈ V ←→ x ∈ V') ⇒ restr-stack V S = restr-stack V' S
⟨proof⟩

Lemma upds-restr-stack[simp]: upds (restr-stack V S) = upds S ∩ V
⟨proof⟩

Lemma fresh-star-restr-stack[intro]:
  a * S ⇒ a * restr-stack V S
⟨proof⟩

Lemma restr-stack-restr-stack[simp]:
  restr-stack V (restr-stack V' S) = restr-stack (V ∩ V') S
⟨proof⟩

Lemma Upd-eq-restr-stack1:
  assumes Upd x # S = restr-stack V S'
  shows x ∈ V
⟨proof⟩

Lemma Upd-eq-restr-stack2:
  assumes restr-stack V S' = Upd x # S
  shows x ∈ V
⟨proof⟩

Lemma restr-stack-noop[simp]:
  restr-stack V S = S ←→ upds S ⊆ 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.intrs[case-names step]

**lemma** invariantE:
invariant rel I ⇒ rel x y ⇒ I x ⇒ I y ⟨proof⟩

**lemma** invariant-starE:
rtmselp 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** rtmselp-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 Δ \cap upds S = {}
  assumes heap-upds-ok (Γ, S)
  shows heap-upds-ok (Δ@Γ, S)
 ⟨proof⟩

lemma heap-upds-ok-let:
  assumes atom ' domA Δ \#* S
  assumes heap-upds-ok (Γ, S)
  shows heap-upds-ok (Δ@Γ, S)
 ⟨proof⟩

lemma heap-upds-ok-to-stack:
  x \in domA Γ \implies heap-upds-ok (Γ, S) \implies heap-upds-ok (delete x Γ, Upd x #S)
 ⟨proof⟩

lemma heap-upds-ok-to-stack':
  map-of Γ x = Some e \implies heap-upds-ok (Γ, S) \implies heap-upds-ok (delete x Γ, Upd x #S)
 ⟨proof⟩

lemma heap-upds-ok-delete:
  heap-upds-ok (Γ, S) \implies heap-upds-ok (delete x Γ, S)
 ⟨proof⟩

lemma heap-upds-ok-restrictA:
  heap-upds-ok (Γ, S) \implies heap-upds-ok (restrictA V Γ, S)
 ⟨proof⟩

lemma heap-upds-ok-restr-stack:
  heap-upds-ok (Γ, S) \implies heap-upds-ok (Γ, restr-stack V S)
 ⟨proof⟩

lemma heap-upds-ok-to-heap:
  heap-upds-ok (Γ, Upd x # S) \implies heap-upds-ok ((x,e) # Γ, S)
 ⟨proof⟩

lemma heap-upds-ok-reorder:
  x \in domA Γ \implies heap-upds-ok (Γ, S) \implies heap-upds-ok ((x,e) # delete x Γ, S)
 ⟨proof⟩

lemma heap-upds-ok-upd:
  heap-upds-ok (Γ, Upd x # S) \implies x \notin domA Γ \land 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-alls1 heap-upds-ok-alls2 heap-upds-ok-delete
heap-upds-ok-restrict-A heap-upds-ok-restr-stack
heap-upds-ok-let
lemmas heap-upds-ok.simps[simp del]

end

2.2 Sestoft

theory Sestoft
imports SestoftConf
begin

inductive step :: conf ⇒ conf ⇒ bool (infix ⇒ 50) where
  app1: (Γ, App e x, S) ⇒ (Γ, e, Arg x # S)
  app2: (Γ, Lam [y], e, Arg x # S) ⇒ (Γ, e[y := x], S)
  var1: map-of Γ x = Some e ⇒ (Γ, Var x, S) ⇒ (delete x Γ, e, Upd x # S)
  var2: x ∉ domA Γ ⇒ isVal e ⇒ (Γ, e, Upd x # S) ⇒ ((x,e)# Γ, e, S)
  let1: atom ` domA ∆♯ Γ ⇒ atom ` domA ∆♯ S ⇒ (Γ, Let ∆ e, S) ⇒ (∆@Γ, e, S)
  let2: (Γ, Bool b, Alts e1 e2 # S) ⇒ (Γ, if b then e1 else e2, S)

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

lemma SmartLet-step1:
  atom ` domA ∆♯ Γ ⇒ atom ` domA ∆♯ S ⇒ (Γ, SmartLet ∆ e, S) ⇒* (∆@Γ, e, S)
⟨proof⟩

lemma lambda-var: map-of Γ x = Some e ⇒ isVal e ⇒ (Γ, Var x, S) ⇒* ((x,e) # delete x Γ, e, S)
⟨proof⟩

lemma let1-closed:
  assumes closed (Γ, Let ∆ e, S)
  assumes domA ∆ ∩ domA Γ = {}
  assumes domA ∆ ∩ upds S = {}
  shows (Γ, Let ∆ e, S) ⇒ (∆@Γ, e, S)
⟨proof⟩

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

lemma step-invariant-induction[consumes 4, case-names app1 app2 thunk lamvar var2 let1 if1 if2 refl trans]:
  assumes c ⇒* c'
  assumes ¬ boring-step c'
  assumes invariant (⇒) I
  assumes I c

16
assumes \( \text{app} \_1 \): \( \forall \Gamma \, e \, S \, x \, . \, I \, (\Gamma, \, App \, e \, x \, S) \implies P \, (\Gamma, \, App \, e \, x, \, S) \) 
assumes \( \text{app} \_2 \): \( \forall \Gamma \, y \, e \, x \, S \, . \, I \, (\Gamma, \, Lam \, [y], \, e, \, Arg \, x \# \, S) \implies P \, (\Gamma, \, Lam \, [y], \, e, \, Arg \, x \# \, S) \) 

assumes \( \text{thunk} \): \( \forall \Gamma \, x \, e \, S \, . \, map-of \, \Gamma \, x \, = \, Some \, e \implies \neg \, isVal \, e \implies I \, (\Gamma, \, Var \, x, \, S) \implies P \, (\Gamma, \, Var \, x, \, S) \) 

assumes \( \text{lamvar} \): \( \forall \Gamma \, x \, e \, S \, . \, map-of \, \Gamma \, x \, = \, Some \, e \implies isVal \, e \implies I \, (\Gamma, \, Var \, x, \, S) \implies P \, (\Gamma, \, Var \, x, \, S) \) 

assumes \( \text{var} \_2 \): \( \forall \Gamma \, x \, e \, S \, . \, x \notin \, domA \, \Gamma \implies isVal \, e \implies I \, (\Gamma, \, e, \, Upd \, x \# \, S) \implies P \, (\Gamma, \, e, \, Upd \, x \# \, S) \) 

assumes \( \text{let} \_1 \): \( \forall \Delta \, \Gamma \, e \, S \, . \, atom \, \Delta \, \# \, \Gamma \implies atom \, \Delta \, \# \, \Gamma \implies I \, (\Gamma, \, Let \, \Delta \, e, \, S) \implies P \, (\Gamma, \, Let \, \Delta \, \, e, \, S) \) 

assumes \( \text{if} \_1 \): \( \forall \Gamma \, b \, e \, e \_1 \, e \_2 \, S \, . \, I \, (\Gamma, \, Bool \, b, \, Alts \, e \_1 \, e \_2 \# \, S) \implies P \, (\Gamma, \, Bool \, b, \, Alts \, e \_1 \, e \_2 \# \, S) \) 

assumes \( \text{refl} \): \( \forall \Gamma \, c \, p \, e \) 

assumes \( \text{trans} \_\text{[trans]} \): \( \forall c \, c' \, e''. \, c \implies c' \implies e' \implies e'' \implies P \, c \, e' \implies P \, e' \implies e'' \implies P \, c \, e'' \) 

shows \( P \, c \, c' \) 

(proof)

\[ \text{lemma step-induction} \] consumes 2, case-names app\_1 app\_2 thunk lamvar var\_2 let\_1 if\_1 if\_2 refl trans:\]

assumes \( c \implies c' \) 

assumes \( \neg \, \text{boring-step} \, c' \) 

assumes \( \text{app} \_1 \): \( \forall \Gamma \, e \, x \, S \, . \, P \, (\Gamma, \, App \, e \, x \, S) \) 

assumes \( \text{app} \_2 \): \( \forall \Gamma \, y \, e \, x \, S \, . \, P \, (\Gamma, \, Lam \, [y], \, e, \, Arg \, x \# \, S) \) 

assumes \( \text{thunk} \): \( \forall \Gamma \, x \, e \, S \, . \, map-of \, \Gamma \, x \, = \, Some \, e \implies \neg \, isVal \, e \implies P \, (\Gamma, \, Var \, x, \, S) \) 

assumes \( \text{lamvar} \): \( \forall \Gamma \, x \, e \, S \, . \, map-of \, \Gamma \, x \, = \, Some \, e \implies isVal \, e \implies P \, (\Gamma, \, Var \, x, \, S) \) 

assumes \( \text{var} \_2 \): \( \forall \Gamma \, x \, e \, S \, . \, x \notin \, domA \, \Gamma \implies isVal \, e \implies P \, (\Gamma, \, e, \, Upd \, x \# \, S) \) 

assumes \( \text{let} \_1 \): \( \forall \Delta \, \Gamma \, e \, S \, . \, atom \, \Delta \, \# \, \Gamma \implies atom \, \Delta \, \# \, \Gamma \implies P \, (\Gamma, \, Let \, \Delta \, e, \, S) \) 

assumes \( \text{if} \_1 \): \( \forall \Gamma \, b \, e \, e \_1 \, e \_2 \, S \, . \, I \, (\Gamma, \, Bool \, b, \, Alts \, e \_1 \, e \_2 \# \, S) \implies P \, (\Gamma, \, Bool \, b, \, Alts \, e \_1 \, e \_2 \# \, S) \) 

assumes \( \text{refl} \): \( \forall \Gamma \, c \, p \, e \) 

assumes \( \text{trans} \_\text{[trans]} \): \( \forall c \, c' \, e''. \, c \implies c' \implies e' \implies e'' \implies P \, c \, e' \implies P \, e' \implies e'' \implies P \, c \, e'' \) 

shows \( P \, c \, c' \) 

(proof)

\[ \text{2.2.1 Equivariance} \]

\[ \text{lemma step-eqvt} \] eqvt: \( \text{step} \, x \, y \implies \text{step} \, (\pi \, \cdot \, x) \, (\pi \, \cdot \, y) \) 

(proof)

\[ \text{2.2.2 Invariants} \]

\[ \text{lemma closal-invariant:} \]
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 ⇒ conf ⇒ bool (infix ⇒G 50) where
  normal: c ⇒ c' ⇒ c ⇒G c'
| dropUp: (Γ, e, Upd x # S) ⇒G (Γ, e, S @ [[Dummy x]])

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

abbreviation gc-steps (infix ⇒G* 50) where gc-steps ≡ gc-step**
lemmas converse-rtranclp-into-rtranclp[of gc-step, OF - r-into-rtranclp, trans]

lemma var-oncel:
  assumes map-of Γ x = Some e
  shows (Γ, Var x, S) ⇒G* (delete x Γ, e, S@[Dummy x])
(proof)

lemma normal-trans: c ⇒* c' ⇒ c ⇒G* c'
(proof)

fun to-gc-conf :: var list ⇒ conf ⇒ conf
  where to-gc-conf r (Γ, e, S) = (restrictA (¬ set r) Γ, e, restr-stack (¬ set r) S @ (map Dummy (rev r)))

lemma restr-stack-map-Dummy[simp]: restr-stack V (map Dummy l) = map Dummy l
(proof)

lemma restr-stack-append[simp]: restr-stack V (l@l') = restr-stack V l @ restr-stack V l'
(proof)

lemma to-gc-conf-append[simp]:
  to-gc-conf (r@r') c = to-gc-conf r (to-gc-conf r' c)

18
proof

lemma to-gc-conf-eqE[elim!]:
assumes to-gc-conf r c = (Γ', e, S)
obtains Γ' S' where c = (Γ', e, S') and Γ = restrictA (∼ set r) Γ' and S = restr-stack (∼ set r) S' @ map Dummy (rev r)

⟨proof⟩

fun safe-hd :: 'a list ⇒ 'a option
where safe-hd (x#-) = Some x
| safe-hd [] = None

lemma safe-hd-None[simp]: safe-hd xs = None ⟷ xs = []
⟨proof⟩

abbreviation r-ok :: var list ⇒ conf ⇒ bool
where r-ok r c ≡ set r ⊆ domA (fst c) ∪ upds (snd (snd c))

lemma subset-bound-invariant:
invariant step (r-ok r)
⟨proof⟩

lemma safe-hd-restr-stack[simp]:
Some a = safe-hd (restr-stack V (a ≠ S)) ⟷ restr-stack V (a ≠ S) = a ≠ restr-stack V S
⟨proof⟩

lemma sestofUnGCStack:
assumes he ap-up ds-ok (Γ, S)
obtains Γ' S' where
(Γ, e, S) ⇒* (Γ', e, S')
to-gc-conf r (Γ, e, S) = to-gc-conf r (Γ', e, S')
¬ isVal e ∨ safe-hd S' = safe-hd (restr-stack (∼ set r) S')
⟨proof⟩

lemma perm-ex1-trivial:
P x x ⇒ ∃ π. P (π • x) x
⟨proof⟩

lemma upds-list-restr-stack[simp]:
upds-list (restr-stack V S) = filter (λ x. x ∈ V) (upds-list S)
⟨proof⟩

lemma heap-upds-ok-to-gc-conf:
heap-upds-ok (Γ, S) ⇒ to-gc-conf r (Γ, e, S) = (Γ'', e'', S'') ⇒ heap-upds-ok (Γ'', S'')
⟨proof⟩

lemma delete-restrictA-conv:
delete x Γ = restrictA (∼{x}) Γ
lemma sestofofUnGCstep:
assumes to-ge-conf r c ⇒ G d
assumes heap-upds-ok-conf c
assumes closed c
and r-ok r c
shows ∃ r′ c′. c ⇒ * c′ ∧ d = to-ge-conf r′ c′ ∧ r-ok r′ c′
(proof)

lemma sestofofUnGC:
assumes (to-ge-conf r c) ⇒ G * d and heap-upds-ok-conf c and closed c and r-ok r c
shows ∃ r′ c′. c ⇒ * c′ ∧ d = to-ge-conf r′ c′ ∧ r-ok r′ c′
(proof)

lemma dummies-unchanged-invariant:
  invariant step (λ (Γ, e, S). dummies S = V) (is invariant ?I)
(proof)

lemma sestofofUnGC′:
assumes ([], e, []) ⇒ G * (Γ, e′, map Dummy r)
assumes isVal e′
assumes fv e = ({}::var set)
shows ∃ Γ′′, (Γ, e, []) ⇒ * (Γ′′, e′, []) ∧ Γ = restrictA (∼ set r) Γ′′ ∧ set r ⊆ domA Γ′′
(proof)

end

2.4 BalancedTraces

theory BalancedTraces
imports Main
begin

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

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

inductive trace :: 'c => 'c list => 'c => bool where
  trace-nil[iff]: trace final [] final
  | trace-cons[intro]: trace conf' T final ⇒ conf ⇒ conf' ⇒ trace conf (conf'#T) final

inductive-cases trace-consE: trace conf (conf'#T) final

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

20
trace \( x1 \) \( z2 \) final \( \Rightarrow \) \( P \) final \[ \] final \( \Rightarrow \) \( (\forall \text{conf}' T \ \text{conf}. \ \text{trace conf}' T \) final \( \Rightarrow \) \( P \) conf' T final \( \Rightarrow \) \( \text{conf} \) \( \Rightarrow \) \( \text{conf}' \) \( \Rightarrow \) \( P \) conf' (conf' # T) final) \( \Rightarrow \) \( P \) \( x1 \) \( z2 \) final

(\text{proof})

lemma \text{build-trace}:
\( c \Rightarrow^* c' \Rightarrow \exists \ T. \ \text{trace} c \ T \ c' \)
(\text{proof})

lemma \text{destruct-trace}:
\( \text{trace} \ c \ T \ c' \Rightarrow \ c \Rightarrow^* \ c' \)
(\text{proof})

lemma \text{tracWhile}:
\( \text{assumes} \ \text{trace} c_1 \ T \ c_4 \)
\( \text{assumes} \ \neg \ P \ c_4 \)
\( \text{obtains} \ T_1 \ c_2 \ T_2 \)
\( \text{where} \ T = T_1 \otimes c_3 \ # \ T_2 \) \( \text{and} \ \text{trace} c_1 \ T_1 \ c_2 \ T_2 \) \( \forall x \in \text{set} \ T_1. \ \neg \ P \ x \) \( \text{and} \ P \ c_2 \)
(\text{proof})

lemma \text{traces-list-all}:
\( \text{trace} c \ T \ c' \Rightarrow P \ c' \Rightarrow (\bigwedge c c' \Rightarrow c \Rightarrow P \ c' \Rightarrow P \ c) \Rightarrow (\forall x \in \text{set} \ T. \ P \ x) \land P \ c \)
(\text{proof})

lemma \text{trace-nil}[simp]: \( \text{trace} \ c \ [] \ c' \leftarrow c = c' \)
(\text{proof})

end

definition \text{extends} :: 'a \ list \Rightarrow 'a \ list \Rightarrow \text{bool} \ (\text{infix} \ \lesssim 50) \ where
\( S \lesssim \ S' = (\exists \ S''. \ S' = S'' @ S) \)

lemma \text{extends-refl}[simp]; \( S \lesssim S \) (\text{proof})
lemma \text{extends-cons}[simp]; \( S \lesssim x \ # \ S \) (\text{proof})
lemma \text{extends-append}[simp]; \( S \lessapprox L \ @ \ S \) (\text{proof})
lemma \text{extends-not-cons}[simp]; \( \neg (x \ # \ S) \lesssim S \) (\text{proof})
lemma \text{extends-trns}[trans]; \( S \lesssim S' \Rightarrow S' \lesssim S'' \Rightarrow S \lesssim S'' \) (\text{proof})

locale \text{balance-trace} = \text{traces} +
\text{fixes} \ stack :: 'a \Rightarrow 's \ list
\text{assumes} \ one-step-only: c \Rightarrow c' \ (\text{stack} c) = (\text{stack} c') \lor (\exists x. \ \text{stack} c' = x \ # \ \text{stack} c) \lor (\exists x. \ \text{stack} c = x \ # \ \text{stack} c')
begin

inductive \text{bal} :: 'a \Rightarrow 'a \ list \Rightarrow 'a \Rightarrow \text{bool} \ where
\( \text{balI}[\text{intro}]: \text{trace} \ c \ T \ c' \Rightarrow \forall c' \in \text{set} \ T. \ \text{stack} c \lesssim \text{stack} c' \Rightarrow \text{stack} c' = \text{stack} c \Rightarrow \text{bal} \ c \ T \ c' \)

inductive-cases \text{balE}: \text{bal} \ c \ T \ c'
lemma bal-nil[simp]: bal c [] c' ↔ c = c'  
(proof)

lemma bal-stackD: bal c T c' → stack c' = stack c  
(proof)

lemma stack-passes-lower-bound:
assumes c_3 ⇒ c_4
assumes stack c_2 ⊆ stack c_3
assumes ¬ stack c_2 ⊆ 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 ⇒ 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 Γ : e ⇓_L Δ : z
and fv (Γ, e, S) ⊆ set L ∪ domA Γ
shows (Γ, e, S) ⇒∗ (Δ, z, S)  
(proof)

type-synonym trace = conf list

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

interpretation traces step  
(proof)

abbreviation trace-syn (- ⇒∗ - [50,50,50] [50]) where trace-syn ≡ trace

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

22
\[(\Gamma, e, S) \Rightarrow_
arrow T \text{ final} \Rightarrow (\land \Gamma \ e \ S. \text{ final} = (\Gamma, e, S) \Rightarrow P \ \Gamma \ e \ S \ [ (\Gamma, e, S)]) \Rightarrow (\land \Gamma \ e \ S T \Gamma' \ e' S' \text{ final} \Rightarrow P \ \Gamma' \ e' S' \ T \text{ final} \Rightarrow (\Gamma, e, S) \Rightarrow (\Gamma', e', S') \Rightarrow P \ \Gamma' \ e' S' T \text{ final}) \Rightarrow P \ \Gamma' \ e' S' T \text{ final} \]\\

**interpretation** balance-trace step stack

\(\langle \text{proof} \rangle \) in terpretation balance-trace step stack

**abbreviation** bal-syn (- \Rightarrow b \ast - [50, 50, 50] 50) where bal-syn \(\equiv\) bal

**lemma** isVal-stops:
- assumes isVal e
- assumes \((\Gamma, e, S) \Rightarrow_
arrow b \ast T (\Delta, z, S)\)
- shows \(T = [[\text{proof}]]\)

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

**lemma** lemma-3:
- assumes \((\Gamma, e, S) \Rightarrow_
arrow b \ast T (\Delta, z, S)\)
- assumes isVal z
- shows \(\Gamma : e \ \downarrow_{\text{upds-list} S} \Delta : z\)

**lemma** dummy-stack-extended:
- set \(S \subseteq \text{Dummy} \ \text{\# \ UNIV} \Rightarrow x \notin \text{Dummy} \ \text{\# \ UNIV} \Rightarrow (S \subseteq x \ \# \ S') \leftrightarrow S \subseteq S'\)

**lemma**[simp]: Arty \(x \notin \text{range Dummy Upd} \ x \notin \text{range Dummy} \text{ Alts } e_1 e_2 \notin \text{range Dummy}\)

**lemma** dummy-stack-balanced:
- assumes set \(S \subseteq \text{Dummy} \ \text{\# \ UNIV}\)
- assumes \((\Gamma, e, S) \Rightarrow_
arrow (\Delta, z, S)\)
- obtains T where \((\Gamma, e, S) \Rightarrow_{b \ast} T (\Delta, z, S)\)

\(\langle \text{proof} \rangle\)

**end**

3 Arity

3.1 Arity

**theory** Arity

**imports** Launchbury,HOLCF−Join−Classes
typedef Arity = UNIV :: nat set
morphism Rep-Arity to-Arity (proof)

setup-lifting type-definition-Arity

instantiate Arity :: po
begin
lift-definition below-Arity :: Arity ⇒ Arity ⇒ bool is λ x y . y ≤ x (proof)

instance (proof)
end

instance Arity :: chfin (proof)

instance Arity :: cpo (proof)

lift-definition inc-Arity :: Arity ⇒ Arity is Suc (proof)
lift-definition pred-Arity :: Arity ⇒ Arity is (λ x . x − 1) (proof)

lemma inc-Arity-cont[simp]: cont inc-Arity (proof)

lemma pred-Arity-cont[simp]: cont pred-Arity (proof)

definition inc :: Arity ⇒ Arity where
inc = (Λ x. inc-Arity x)

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

lemma inc-inj[simp]: inc·n = inc·n’ ←→ n = n’ (proof)

lemma pred-inc[simp]: pred·(inc·n) = n (proof)

lemma inc-below-inc[simp]: inc·a ⊆ inc·b ←→ a ⊆ b (proof)

lemma inc-below-below-pred[elim]:
inc·a ⊆ b ⇒ a ⊆ pred· b (proof)
lemma Rep-Arity-inc[simp]: Rep-Arity (inc·a') = Suc (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 = inc·0
⟨proof⟩

lemma inc-not-0[simp]: inc·n = 0 ⇔ False
⟨proof⟩

lemma pred-0[simp]: pred·0 = 0
⟨proof⟩

lemma Arity-ind: P 0 ⇒ (∀ n. P n ⇒ P (inc·n)) ⇒ P n
⟨proof⟩

lemma Arity-total:
  fixes x y :: Arity
  shows x ⊑ y ∨ y ⊑ x
⟨proof⟩

instance Arity :: Finite-Join-cpo
⟨proof⟩

lemma Arity-zero-top[simp]: (x :: Arity) ⊑ 0
⟨proof⟩

lemma Arity-above-top[simp]: 0 ⊑ (a :: Arity) ⇔ a = 0
⟨proof⟩

lemma Arity-zero-join[simp]: (x :: Arity) ⊔ 0 = 0
⟨proof⟩
lemma Arity-zero-join2[simp]: 0 ⊔ (x :: Arity) = 0
⟨proof⟩

lemma Arity-up-zero-join[simp]: (x :: Arity⊥) ⊔ up·0 = up·0
⟨proof⟩
lemma Arity-up-zero-join2[simp]: \( \text{up} \cdot 0 \sqcup (x :: \text{Arity}_\bot) = \text{up} \cdot 0 \)  
\langle proof \rangle

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

lemma Arity-above-up-top[simp]: \( \text{up} \cdot 0 \sqsubseteq (a :: \text{Arity}_\bot) \iff a = \text{up} \cdot 0 \)  
\langle proof \rangle

lemma Arity-exhaust: \( (y = 0 \Rightarrow P) \Rightarrow (\forall x. y = \text{inc} \cdot x \Rightarrow P) \Rightarrow P \)  
\langle proof \rangle

end

3.2 AEnv

theory AEnv

imports Arity Launchbury.Vars Launchbury.Env

begin


type-synonym AEnv = \text{var} \Rightarrow \text{Arity}_\bot

end

3.3 Arity-Nominal

theory Arity-Nominal

imports Arity Launchbury.Nominal-HOLCF

begin

lemma join-eqv[simp]: \( \pi \cdot (x \sqcup (y :: a :: \{\text{Finite-Join-cpo, cont-pt}\})) = (\pi \cdot x) \sqcup (\pi \cdot y) \)  
\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 ⇒ Arity
where Astack [] = 0
| Astack (Arg x # S) = inc·(Astack S)
| Astack (Alts e1 e2 # S) = 0
| Astack (Upd x # S) = 0
| Astack (Dummy x # S) = 0

lemma Astack-restr-stack-below:
Astack (restr-stack V S) ⊑ Astack S
⟨proof⟩

lemma Astack-map-Dummy[simp]:
Astack (map Dummy l) = 0
⟨proof⟩

lemma Astack-append-map-Dummy[simp]:
Astack S' = 0 ⇒ 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 ⇒ var where
fresh-var e = (SOME v. v /∈ fv e)

lemma fresh-var-not-free:
fresh-var e /∈ fv e
⟨proof⟩

lemma fresh-var-fresh[simp]:
atom (fresh-var e) ≡ e
⟨proof⟩

lemma fresh-var-subst[simp]:
\[ e^{fresh\cdot var\ e::=x} = e \]

\[ \begin{align*}
\text{fun}\ eta\text{-expand} & : \text{nat} \Rightarrow \exp \Rightarrow \exp \text{ where} \\
eta\text{-expand} \ 0\ e & = e \\
| \eta\text{-expand} \ (Suc\ n)\ e & = (\text{Lam}\ [fresh\cdot var\ e].\ \eta\text{-expand}\ n\ (\text{App}\ e\ (fresh\cdot var\ e)))
\end{align*} \]

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

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

\[ \begin{align*}
\text{lemma}\ subst\cdot eta\text{-expand}: (\eta\text{-expand}\ n\ e)[x ::= y] & = \eta\text{-expand}\ n\ (e[x ::= y]) \\
\text{⟨proof⟩}
\end{align*} \]

\[ \begin{align*}
\text{lemma}\ isLam\cdot eta\text{-expand}:
\text{isLam}\ e & \Rightarrow \text{isLam}\ (\eta\text{-expand}\ n\ e) \text{ and } n > 0 \Rightarrow \text{isLam}\ (\eta\text{-expand}\ n\ e) \\
\text{⟨proof⟩}
\end{align*} \]

\[ \begin{align*}
\text{lemma}\ isVal\cdot eta\text{-expand}:
\text{isVal}\ e & \Rightarrow \text{isVal}\ (\eta\text{-expand}\ n\ e) \text{ and } n > 0 \Rightarrow \text{isVal}\ (\eta\text{-expand}\ n\ e) \\
\text{⟨proof⟩}
\end{align*} \]

\[ \text{end} \]

\[ \text{4.2 EtaExpansionSafe} \]

\[ \text{theory}\ EtaExpansionSafe \]
\[ \text{imports}\ EtaExpansion\ Sesoft \]
\[ \text{begin} \]

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

\[ \begin{align*}
\text{fun}\ arg\cdotprefix\ ::\ \text{stack} \Rightarrow \text{nat} \text{ where} \\
\text{arg}\cdotprefix\ [] & = 0 \\
| \text{arg}\cdotprefix\ (\text{Arg}\ x \ #\ S) & = \text{Suc}\ (\text{arg}\cdotprefix\ S) \\
| \text{arg}\cdotprefix\ (\text{Alts}\ e1\ e2 \ #\ S) & = 0 \\
| \text{arg}\cdotprefix\ (\text{Upd}\ x \ #\ S) & = 0 \\
| \text{arg}\cdotprefix\ (\text{Dummy}\ x \ #\ S) & = 0
\end{align*} \]

\[ \begin{align*}
\text{theorem}\ eta\text{-expansion-safe!’:\} \\
\text{assumes}\ n \leq\ \text{arg}\cdotprefix\ S \\
\text{shows}\ (\Gamma,\ \eta\text{-expand}\ n\ e,\ S) \Rightarrow^* \ (\Gamma,\ e,\ S)
\end{align*} \]

28
4.3 Transform Tools

default-sort type

fun lift-transform :: ('a :: cont-pt ⇒ exp ⇒ exp) ⇒ ('a ⊥ ⇒ exp ⇒ exp)
  where lift-transform t ⊥bottom e = e
     | lift-transform t (lup a) e = t a e

lemma lift-transform-simps[simp]:
  lift-transform t ⊥ e = e
  lift-transform t (wp a) e = t a e

⟨proof⟩

lemma lift-transform-equvt[equiv]: π · lift-transform t a e = lift-transform (π · t) (π · a) (π · e)
  ⟨proof⟩

lemma lift-transform-fun-cong[fundef-cong]:
  (∀ a. t1 a e1 = t2 a e1) → a1 = a2 → e1 = e2 → lift-transform t1 a1 e1 = lift-transform t2 a2 e2
  ⟨proof⟩

lemma subst-lift-transform:
  assumes ∀ a. (t a e)[x ::= y] = t a (e[x ::= y])
  shows (lift-transform t a e)[x ::= y] = lift-transform t a (e[x ::= y])
  ⟨proof⟩

definition map-transform :: ('a :: cont-pt ⇒ exp ⇒ exp) ⇒ (var ⇒ 'a ⊥) ⇒ heap ⇒ heap
  where map-transform t ae = map-rn (λ x e . lift-transform t (ae x) e)

lemma map-transform-equvt[equiv]: π · map-transform t ae = map-transform (π · t) (π · 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-transform t ae (delete x Γ) = delete x (map-transform t ae Γ)
⟨proof⟩

lemma map-transform-restrA:
map-transform t ae (restrictA S Γ) = restrictA S (map-transform t ae Γ)
⟨proof⟩

lemma delete-map-transform-env-delete:
delete x (map-transform t (env-delete x ae) Γ) = delete x (map-transform t ae Γ)
⟨proof⟩

lemma map-transform-Nil[simp]:
map-transform t ae [] = []
⟨proof⟩

lemma map-transform-Cons:
map-transform t ae ((x,e) # Γ) = (x, lift-transform t (ae x) e) # (map-transform t ae Γ)
⟨proof⟩

lemma map-transform-append:
map-transform t ae (Δ @ Γ) = map-transform t ae Δ @ map-transform t ae Γ
⟨proof⟩

lemma map-transform-fundef-cong[fundef-cong]:
(∃ x e a. (x,e) ∈ set m1 ⇒ t1 a e = t2 a e) ⇒ ae1 = ae2 ⇒ m1 = m2 ⇒ map-transform t1 ae1 m1 = map-transform t2 ae2 m2
⟨proof⟩

lemma map-transform-cong:
(∃ x. x ∈ domA m1 ⇒ ae x = ae' x) ⇒ m1 = m2 ⇒ map-transform t ae m1 = map-transform t ae' m2
⟨proof⟩

lemma map-of-map-transform: map-of (map-transform t ae Γ) x = map-option (lift-transform t (ae x)) (map-of Γ x)
⟨proof⟩

lemma supp-map-transform-step:
assumes \( x e a. (x,e) ∈ set Γ \) \( t a e \) \( \subseteq \) \( supp e \)
shows supp (map-transform t ae Γ) \( \subseteq \) supp Γ
⟨proof⟩

lemma subst-map-transform:
assumes \( x e a. (x',e) : set Γ \) \( t a e \) \( x ::= y \) = \( t a (e[x ::= y]) \)
shows (map-transform t ae Γ)[x ::= h=y] = map-transform t ae (Γ[x ::h=y])
⟨proof⟩

locale supp-bounded-transform =

30
\textbf{fixes} \textit{trans} :: 'a::cont-pt \Rightarrow \textit{exp} \Rightarrow \textit{exp}

\textbf{assumes} \ suppt\textit{-trans}: \ supp\ (\textit{trans} \ a \ e) \subseteq \ supp \ e

\textbf{begin}

\textbf{lemma} \ suppt\textit{-lift-transform}: \ supp\ (\textit{lift-transform tran} \ a \ e) \subseteq \ supp \ e

\textbf{lemma} \ suppt\textit{-map-transform}: \ supp\ (\textit{map-transform tran} \ a \ \Gamma) \subseteq \ supp \ \Gamma

\textbf{lemma} \ fresht\textit{-transform[intro]}: \ a \ \sharp \ e \Rightarrow \ a \ \sharp \ \textit{trans} \ n \ e

\textbf{lemma} \ fresht\textit{-star-transform[intro]}: \ a \ \sharp* \ e \Rightarrow \ a \ \sharp* \ \textit{trans} \ n \ e

\textbf{lemma} \ fresht\textit{-map-transform[intro]}: \ a \ \sharp \ \Gamma \Rightarrow \ a \ \sharp \ \textit{map-transform tran} \ a \ \Gamma \ \Gamma

\textbf{lemma} \ fresht\textit{-star-map-transform[intro]}: \ a \ \sharp* \ \Gamma \Rightarrow \ a \ \sharp* \ \textit{map-transform tran} \ a \ \Gamma

\textbf{end}

\textbf{end}

\section{4.4 ArityEtaExpansion}

\textbf{theory} \ ArityEtaExpansion

\textbf{imports} \ EtaExpansion Arity–Nominal TransformTools

\textbf{begin}

\textbf{lift-definition} \ \textit{Arity-expand} :: \ Arity \Rightarrow \textit{exp} \Rightarrow \textit{exp} \ is \ \textit{eta-expand}(\text{proof})

\textbf{lemma} \ \textit{Arity-expand-equiv}[equiv]: \ \pi \cdot \ \textit{Arity-expand} \ a \ e = \ \textit{Arity-expand} \ (\pi \cdot a) \ (\pi \cdot e)

\textbf{lemma} \ \textit{Arity-expand-0}[simp]: \ \textit{Arity-expand} \ 0 \ e = e

\textbf{lemma} \ \textit{Arity-expand-inc}[simp]: \ \textit{Arity-expand} \ (\text{inc-}n) \ e = (\text{Lam} \ [\text{fresh-var} \ e]. \ \textit{Arity-expand} \ n \ (\text{App} \ e \ (\text{fresh-var} \ e)))

\textbf{lemma} \ \textit{subst-Arity-expand}:
\ (\textit{Arity-expand} \ n \ e)[x::=y] = \textit{Arity-expand} \ n \ e[x::=y]

\textbf{lemma} \ \textit{isLam-Arity-expand}: \ \textit{isLam} \ e \Rightarrow \ \textit{isLam} \ (\textit{Arity-expand} \ a \ e)

\textbf{end}

31
\textbf{lemma} isVal-A eta-exp: \( \text{isVal } e \implies \text{isVal } (\text{A eta-exp } a \ e) \) \\
\hspace{1cm} \langle \text{proof} \rangle

\textbf{lemma} A eta-exp-fresh[simp]: \( a \ \ddagger \ A \text{ eta-exp } n \ e = a \ \ddagger \ e \) \langle \text{proof} \rangle
\textbf{lemma} A eta-exp-fresh-star[simp]: \( a \ \ddagger \ast \ A \text{ eta-exp } n \ e = a \ \ddagger \ast \ e \) \langle \text{proof} \rangle

\textbf{interpretation} supp-bounded-transform A eta-exp \\
\hspace{1cm} \langle \text{proof} \rangle

\textbf{end}

4.5 Arit yEtaExpansionSafe

\textbf{theory} A rit yEtaExpansionSafe \\
\textbf{imports} EtaExpansionSafe A rit yStack A rit yEtaExpansion \\
\textbf{begin}

\textbf{lemma} A eta-exp-safe: \\
\hspace{1cm} \textbf{assumes} Astack S \sqsubseteq a \\
\hspace{1.5cm} \textbf{shows} (\Gamma, A eta-exp a \ e, S) \Rightarrow^* (\Gamma, e, S) \\
\hspace{1cm} \langle \text{proof} \rangle

\textbf{end}

5 Arit y Analysis

5.1 Arit yAnalysisSig

\textbf{theory} A rit yAnalysisSig \\
\textbf{imports} Launchbury.Terms AEnv A rit y Nominal Launchbury.Nominal – HOLCF Launchbury.Substitution \\
\textbf{begin}

\textbf{locale} A rit yAnalysis = \\
\hspace{1cm} \textbf{fixes} Aexp :: exp \Rightarrow Arit y \Rightarrow AEnv \\
\hspace{1cm} \langle \text{proof} \rangle

\textbf{locale} A rit yAnalysisHeap = \\
\hspace{1cm} \textbf{fixes} Aheap :: heap \Rightarrow exp \Rightarrow Arit y \Rightarrow AEnv \\
\hspace{1cm} \langle \text{proof} \rangle

\textbf{locale} EdomA rit yAnalysis = A rit yAnalysis +
assumes $A_{exp-dom}: dom (A_a e) \subseteq fv e$

begin

lemma $fup-A_{exp-dom}: dom (A_{\downarrow}^a e) \subseteq fv e$
(\langle proof \rangle)

lemma $A_{exp-fresh-bot}[simp]:$ assumes atom $v \notin e$ shows $A_a e v = \bot$
(\langle proof \rangle)

end

locale ArityAnalysisHeapEqvt = ArityAnalysisHeap +
assumes Aheap-eqv\[eqvt\]: $\pi \cdot Aheap = Aheap$

end

5.2 ArityAnalysisAbinds

theory ArityAnalysisAbinds
imports ArityAnalysisSig
begin

context ArityAnalysis
begin

5.2.1 Lifting arity analysis to recursive groups

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

lemma $ABind-eq[simp]:$ $ABind v e \cdot ae = A_{\downarrow}^{ae} v e$
(\langle proof \rangle)

fun $ABind\_eq ::$ heap $\Rightarrow$ (AEnv $\rightarrow$ AEnv)
where $ABind\_eq [] = \bot$
| $ABind\_eq ((v, e)\#binds) = ABind v e \sqcup ABind\_eq (delete v binds)$

lemma $ABind\_eq\_strict[simp]:$ $ABind\_eq\_strict \bot = \bot$
(\langle proof \rangle)

lemma Abinds-order1: map-of $\Gamma v = Some e \Longrightarrow ABind\_eq\_strict \Gamma = ABind v e \sqcup ABind\_eq\_strict (delete v \Gamma)$
(\langle proof \rangle)

lemma ABind-below-ABinds: map-of $\Gamma v = Some e \Longrightarrow ABind\_eq\_strict \Gamma \sqsubseteq ABind\_eq\_strict\_strict\_strict \Delta$
(\langle proof \rangle)

lemma Abinds-order: map-of $\Gamma = map-of \Delta \Longrightarrow ABind\_eq\_strict \Gamma = ABind\_eq\_strict \Delta$
(\langle proof \rangle)
lemma Abinds-env-cong: (\( x. x \in \text{dom}A \Rightarrow ae x = ae' x \)) \(\Rightarrow\) Abinds \(\Delta ae \Rightarrow\) Abinds \(\Delta ae' \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-env-restr-cong: \(ae f \Downarrow \text{dom}A \Rightarrow ae' f \Downarrow \text{dom}A \Rightarrow\) Abinds \(\Delta ae \Rightarrow\) Abinds \(\Delta ae' \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-env-restr[simp]: Abinds \(\Delta (ae f \Downarrow \text{dom}A) \Rightarrow\) Abinds \(\Delta ae \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-join-fresh: ae' (domA \(\Delta \) \(\subseteq\) \(\perp\)) \(\Rightarrow\) Abinds \(\Delta (ae \sqcup ae') \Rightarrow\) (Abinds \(\Delta ae \))
(\(\langle\text{proof}\rangle\))

lemma Abinds-delete-bot: ae x = \(\perp\) \(\Rightarrow\) Abinds (delete x \(\Gamma\))ae = Abinds \(\Gamma \cdot ae \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-restr-fresh:
assumes atom ' S \(\sharp\) \(\ast\) \(\Gamma\)
shows Abinds \(\Gamma \cdot ae f \Downarrow (- S) \Rightarrow\) Abinds \(\Gamma \cdot (ae f \Downarrow (- S)) f \Downarrow (- S) \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-restr:
assumes domA \(\Gamma \subseteq S\)
shows Abinds \(\Gamma \cdot ae f \Downarrow S \Rightarrow\) Abinds \(\Gamma \cdot (ae f \Downarrow S) f \Downarrow S \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-restr-subst:
assumes \(\bigwedge x' e a. (x',e) \in \text{set} \Gamma \Rightarrow\) Aexp \(e[x::=y] a f \Downarrow S \Rightarrow\) Abinds \(\Gamma \cdot (ae f \Downarrow S) f \Downarrow S \)
assumes x \(\notin S\)
assumes y \(\notin S\)
assumes domA \(\Gamma \subseteq S\)
shows Abinds \(\Gamma[x::h:=y] ae f \Downarrow S \Rightarrow\) Abinds \(\Gamma \cdot (ae f \Downarrow S) f \Downarrow S \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-append-disjoint: domA \(\Delta \cap \text{dom} \Gamma = \{\}\) \(\Rightarrow\) Abinds \(\Delta ae \cup \text{Abinds} \Gamma \cdot ae \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-restr-subset: \(S \subseteq S' \Rightarrow\) Abinds \((\text{restrictA} S \Gamma) \cdot ae \subseteq\) Abinds \((\text{restrictA} S') \Gamma \cdot ae \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-restrict-edom: Abinds \((\text{restrictA} (\text{edom} ae) \Gamma) \cdot ae \Rightarrow\) Abinds \(\Gamma \cdot ae \)
(\(\langle\text{proof}\rangle\))

lemma Abinds-restrict-below: Abinds \((\text{restrictA} S \Gamma) \cdot ae \subseteq\) Abinds \(\Gamma \cdot ae \)

34
lemma ABinds-delete-below: ABinds (delete x Γ)·ae ⊆ ABinds Γ·ae
⟨proof⟩
end

lemma ABind-eqvt[eqvt]: π · (AarityAnalysis.ABind Aexp v e) = AarityAnalysis.ABind (π · Aexp) (π · v) (π · e)
⟨proof⟩
end

lemma ABinds-eqvt[eqvt]: π · (AarityAnalysis.ABinds Aexp Γ) = AarityAnalysis.ABinds (π · Aexp) (π · Γ)
⟨proof⟩
end

lemma ABinds-cong[fundef-cong]:
\[
\begin{array}{c}
(\forall \ e, e \in \text{snd} \cdot \text{set heap2} \Rightarrow \text{aexp1} e = \text{aexp2} e) \Rightarrow \text{heap1} = \text{heap2}
\end{array}
\]
end

context EdomAarityAnalysis
begin

lemma fup-Aexp-lookup-fresh: atom v ≠ e \Rightarrow (fup·(Aexp e)·a) v = ⊥
⟨proof⟩
end

lemma edom-AnalBinds: edom (ABinds Γ·ae) ⊆ fv Γ
⟨proof⟩
end

end

5.3 ArityAnalysisSpec

theory ArityAnalysisSpec
imports ArityAnalysisAbinds
begin

locale SubstAarityAnalysis = EdomAarityAnalysis +
assumes Aexp-subst-restr: x ∉ S \Rightarrow y ∉ S \Rightarrow (Aexp e[x:=y]·a) f |' S = (Aexp e·a) f |' S

locale AarityAnalysisSafe = SubstAarityAnalysis +
assumes Aexp-Var: wp · n ⊆ (Aexp (Var x)·n) x
assumes Aexp-App: Aexp e · (inc·n) \cup esing x · (up·0) ⊆ Aexp (App e x) · n
assumes Aexp-Lam: env-delete y (Aexp e · (pred·n)) ⊆ Aexp (Lam [y]. e) · n
assumes Aexp-IfThenElse: Aexp scrut·0 \cup Aexp e1·a \cup Aexp e2·a ⊆ Aexp ( scrut ? e1 : e2)·a

locale AarityAnalysisHeapSafe = ArityAnalysisSafe + ArityAnalysisHeapEqvt +
assumes edom-Aheap: edom (Aheap Γ·e·a) ⊆ domA Γ

35
assumes \( \text{Aheap-subst}: x \notin \text{dom}A \Gamma \implies y \notin \text{dom}A \Gamma \implies \text{Aheap} \Gamma[x:=y] e[x:=y] = \text{Aheap} \Gamma e \)

locale \text{ArityAnalysisLetSafe} = \text{ArityAnalysisHeapSafe} + 
assumes \( \text{Aexp-\text{Let}}: \text{ABinds} \Gamma.(\text{Aheap} \Gamma e\cdot a) \sqcup \text{Aexp} e\cdot a \subseteq \text{Aheap} \Gamma e\cdot a \sqcup \text{Aexp} (\text{Let} \Gamma e)\cdot a \)

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

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

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

context \text{ArityAnalysisSafe}
begin
lemma \( \text{Aexp-Var-singleton}: \text{esing} x \cdot (\text{up} \cdot n) \sqsubseteq \text{Aexp} \ (\text{Var} x) \cdot n \)
\langle proof \rangle
de\end

context \text{ArityAnalysisLetSafe}
begin
lemma \( \text{Aheap-nonrec}: \)
\begin{align*}
\text{assumes } & \text{nonrec } \Delta \\
\text{shows } & \text{Aexp} e\cdot a \not\vdash \text{dom}A \Delta \sqsubseteq \text{Aheap} \Delta e\cdot a
\end{align*}
\langle proof \rangle
de\end

\section{5.4 TrivialArityAnal}

definition \text{Trivial-Aexp} :: \text{exp} \Rightarrow \text{Arity} \rightarrow \text{AEnv}
where \( \text{Trivial-Aexp} e = (\lambda n. (\lambda x. \text{up} \cdot 0) f) \cdot f e) \)
lemma Trivial-Aexp-simp: Trivial-Aexp e \cdot n = (\lambda x. up \cdot 0) f \mid^e fv e
(proof)

lemma edom-Trivial-Aexp[simp]: edom (Trivial-Aexp e \cdot n) = fv e
(proof)

lemma Trivial-Aexp-eq[sff]: Trivial-Aexp e \cdot n = Trivial-Aexp e' \cdot n' \iff fv e = (fv e' :: var set)
(proof)

lemma below-Trivial-Aexp[simp]: (ae \sqsubseteq Trivial-Aexp e \cdot n) \iff edom ae \subseteq fv e
(proof)

interpretation ArityAnalysis Trivial-Aexp
(proof)

interpretation EdomArityAnalysis Trivial-Aexp
(proof)

interpretation ArityAnalysisSafe Trivial-Aexp
(proof)

definition Trivial-Aheap :: heap \Rightarrow exp \Rightarrow Arity \Rightarrow AEnv where
Trivial-Aheap \Gamma e = (\Lambda a. (\lambda x. up \cdot 0) f \mid^e domA \Gamma)

lemma Trivial-Aheap-eqvt[eqvt]: \pi \cdot (Trivial-Aheap \Gamma e) = Trivial-Aheap (\pi \cdot \Gamma) (\pi \cdot e)
(proof)

lemma Trivial-Aheap-simp: Trivial-Aheap \Gamma e \cdot a = (\lambda x. up \cdot 0) f \mid^e domA \Gamma
(proof)

lemma Trivial-fup-Aexp-below-fv: fup \cdot (Trivial-Aexp e) \cdot a \subseteq (\lambda x. up \cdot 0) f \mid^e fv e
(proof)

lemma Trivial-Abinds-below-fv: Abinds \Gamma ae \subseteq (\lambda x. up \cdot 0) f \mid^e fv \Gamma
(proof)

interpretation ArityAnalysisLetSafe Trivial-Aexp Trivial-Aheap
(proof)

end

5.5 ArityAnalysisStack

theory ArityAnalysisStack
imports SestoftConf ArityAnalysisSig
begin

context ArityAnalysis
begin
fun AStack :: Arity list ⇒ stack ⇒ AEnv
where
  AStack [] = ⊥
| AStack (a # as) (Alts e1 e2 # S) = Aexp e1 · a △ Aexp e2 · a △ AStack as S
| AStack as (Upd x # S) = eising x · (up · 0) △ AStack as S
| AStack as (Arg x # S) = eising x · (up · 0) △ AStack as S
| AStack as (- # S) = AStack as S
end

context EdomArityAnalysis
begin
lemma edom-AStack: edom (AStack as S) ⊆ fv S
⟨proof⟩
end
end

5.6 ArityAnalysisFix

theory ArityAnalysisFix
imports ArityAnalysisSig ArityAnalysisAbinds
begin

context ArityAnalysis
begin

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

lemma Afix-eq: Afix Γ · ae = (μ ae′. (ABinds Γ · ae′) △ ae)
⟨proof⟩

lemma Afix-strict[simp]: Afix Γ · ⊥ = ⊥
⟨proof⟩

lemma Afix-least-below: ABinds Γ · ae′ ⊆ ae′ ⇒ ae ⊆ ae′ ⇒ Afix Γ · ae ⊆ ae′
⟨proof⟩

lemma Afix-unroll: Afix Γ · ae = ABinds Γ · (Afix Γ · ae) △ ae
⟨proof⟩

lemma Abinds-below-Afix: ABinds Δ ⊆ Afix Δ
⟨proof⟩

lemma Afix-above-arg: ae ⊆ Afix Γ · ae
⟨proof⟩

lemma Abinds-Afix-below[simp]: ABinds Γ · (Afix Γ · ae) ⊆ Afix Γ · ae
end
\textbf{Proof}

\textbf{Lemma 1:} \texttt{Ax-reorder:} \texttt{map-of} $\Gamma = \texttt{map-of} \ \Delta \Rightarrow \texttt{Ax} \ \Gamma = \texttt{Ax} \ \Delta$

\textbf{Proof}

\textbf{Lemma 2:} \texttt{Ax-repeat-singleton:} $(\mu \ xa. \texttt{Ax} \ (\texttt{esing} \ x \cdot (n \sqcup xa \ x) \sqcup ae)) = \texttt{Ax} \ (\texttt{esing} \ x \cdot n \sqcup ae)$

\textbf{Proof}

\textbf{Lemma 3:} \texttt{Ax-join-fresh:} $ae' \cdot (\texttt{domA} \ \Delta) \subseteq \{\bot\} \Rightarrow \texttt{Ax} \ \Delta \cdot (ae \sqcup ae') = (\texttt{Ax} \ \Delta \cdot ae) \sqcup ae'$

\textbf{Proof}

\textbf{Lemma 4:} \texttt{Ax-restr-fresh:}
\begin{itemize}
  \item \text{assumes} \ $\text{atom} \ 'S \ \not\subseteq \ \Gamma$
  \item \text{shows} \ $\texttt{Ax} \ \Gamma \cdot ae \ |' \ (-S) = \texttt{Ax} \ \Gamma \cdot (ae \ |' (-S)) \ |' (-S)$
\end{itemize}

\textbf{Proof}

\textbf{Lemma 5:} \texttt{Ax-restr:}
\begin{itemize}
  \item \text{assumes} \ $\text{domA} \ \Gamma \subseteq S$
  \item \text{shows} \ $\texttt{Ax} \ \Gamma \cdot ae \ |' \ S = \texttt{Ax} \ \Gamma \cdot (ae \ |' S) \ |' S$
\end{itemize}

\textbf{Proof}

\textbf{Lemma 6:} \texttt{Ax-restr-subst':}
\begin{itemize}
  \item \text{assumes} \ $\forall \ x \cdot a. \ (x', e) \in \text{set} \ \Gamma \Rightarrow \texttt{Aexp} \ e[\cdot := y] \cdot a \ |' \ S = \texttt{Aexp} \ e \cdot a \ |' \ S$
  \item \text{assumes} \ $x \notin S$
  \item \text{assumes} \ $y \notin S$
  \item \text{assumes} \ $\text{domA} \ \Gamma \subseteq S$
  \item \text{shows} \ $\texttt{Ax} \ \Gamma [x := h = y \cdot ae \ |' \ S = \texttt{Ax} \ \Gamma \cdot (ae \ |' S) \ |' S$
\end{itemize}

\textbf{Proof}

\textbf{Lemma 7:} \texttt{Ax-subst-approx:}
\begin{itemize}
  \item \text{assumes} \ $\forall \ v, n. \ v \in \text{domA} \ \Gamma \Rightarrow \texttt{Aexp} \ (\texttt{the} \ (\texttt{map-of} \ \Gamma \ v))[y := x \cdot : n] \subseteq (\texttt{Aexp} \ (\texttt{the} \ (\texttt{map-of} \ \Gamma \ v)))[y := x \cdot : n]$
  \item \text{assumes} \ $x \notin \text{domA} \ \Gamma$
  \item \text{assumes} \ $y \notin \text{domA} \ \Gamma$
  \item \text{shows} \ $\texttt{Ax} \ \Gamma [y := h = x \cdot (ae(y := \bot, x := \texttt{up} \cdot 0))] \subseteq (\texttt{Ax} \ \Gamma \cdot ae)(y := \bot, x := \texttt{up} \cdot 0)$
\end{itemize}

\textbf{Proof}

\textbf{end}

\textbf{Lemma 8:} \texttt{Ax-eqvt[eqvt]}: $\pi \cdot (\texttt{ArityAnalysis.Aexp} \ \Gamma) = \texttt{ArityAnalysis.Afix} \ (\pi \cdot \texttt{Aexp}) \ (\pi \cdot \Gamma)$

\textbf{Proof}

39
lemma \(\text{Ax-c ong}/\text{fundef-c ong}]:\)
\[
\begin{align*}
&\left(\forall e.\ e \in \text{snd} \cdot \text{set heap2} \implies \text{aexp1 } e = \text{aexp2 } e;\ \text{heap1} = \text{heap2}\right) \\
\implies \text{ArityAnalysis.Afix aexp1 heap1} = \text{ArityAnalysis.Afix aexp2 heap2}
\end{align*}
\]
(proof)

context EdomArityAnalysis
begin

lemma \(\text{Ax-e dom}:\)
\[
\text{edom} (\text{Afix } \Gamma \cdot \text{ae}) \subseteq \text{fv} \Gamma \cup \text{edom} \text{ae}
\]
(proof)

lemma \(\text{ABinds-lookup-fresh}:\)
\[
\text{atom } v \not\in \Gamma \implies (\text{ABinds } \Gamma \cdot \text{ae}) v = \bot
\]
(proof)

lemma \(\text{Afix-lookup-fresh}:\)
\[
\begin{align*}
&\text{assumes}\ \text{atom } v \not\in \Gamma \\
&\text{shows}\ (\text{Afix } \Gamma \cdot \text{ae}) v = \text{ae } v
\end{align*}
\]
(proof)

lemma \(\text{Afix-comp2join-fresh}:\)
\[
\begin{align*}
&\text{atom } \cdot (\text{domA } \Delta) \not\in \Gamma \implies \text{ABinds } \Delta \cdot (\text{Afix } \Gamma \cdot \text{ae}) = \text{ABinds } \Delta \cdot \text{ae}
\end{align*}
\]
(proof)

lemma \(\text{Afix-append-fresh}:\)
\[
\begin{align*}
&\text{assumes}\ \text{atom } \cdot \text{domA } \Delta \not\in \Gamma \\
&\text{shows}\ (\text{Afix } (\Delta @ \Gamma) \cdot \text{ae}) = (\text{Afix } \Delta \cdot \text{ae})
\end{align*}
\]
(proof)

lemma \(\text{Afix-e-to-heap}:\)
\[
\text{Afix } (\text{delete } x \Gamma)(\text{fup} \cdot (\text{Aexp } e \cdot \text{n} \sqcup \text{ae})) \subseteq (\text{Afix } ((x, e) \# \text{delete } x \Gamma) \cdot (\text{esing } x \cdot \text{n} \sqcup \text{ae})
\]
(proof)

lemma \(\text{Afix-e-to-heap}':\)
\[
\text{Afix } (\text{delete } x \Gamma)(\text{Aexp } e \cdot \text{n}) \subseteq (\text{Afix } ((x, e) \# \text{delete } x \Gamma) \cdot (\text{esing } x \cdot (\text{up} \cdot \text{n}))
\]
(proof)

end

end

5.7 ArityAnalysisFixProps

theory ArityAnalysisFixProps
imports ArityAnalysisFix ArityAnalysisSpec
context SubstArityAnalysis

begin

lemma Afx-restr-subst:
assumes x /∈ S
assumes y /∈ S
assumes domA Γ ⊆ S
shows Afx Γ[x::=y] ae f | S = Afx (ae f) | S)
⟨proof⟩
end

end

6 Arit y Transformation

6.1 Abstract Transform

theory AbstractTransform
imports Launchbury.Terms TransformTools
begin

locale AbstractAnalProp =
fixes PropApp : 'a ⇒ 'a::cont-pt
fixes PropLam : 'a ⇒ 'a
fixes AnalLet : heap ⇒ exp ⇒ 'a ⇒ 'b::cont-pt
fixes PropLetBody : 'b ⇒ 'a
fixes PropLetHeap : 'b ⇒ var ⇒ 'a⊥
fixes PropIfScrut : 'a ⇒ 'a
assumes PropApp-eqv: π · PropApp ≡ PropApp
assumes PropLam-eqv: π · PropLam ≡ PropLam
assumes AnalLet-eqv: π · AnalLet ≡ AnalLet
assumes PropLetBody-eqv: π · PropLetBody ≡ PropLetBody
assumes PropLetHeap-eqv: π · PropLetHeap ≡ PropLetHeap
assumes PropIfScrut-eqv: π · PropIfScrut ≡ PropIfScrut

locale AbstractAnalPropSubst = AbstractAnalProp +
assumes AnalLet-subst: x /∈ domA Γ ⇒ y /∈ domA Γ ⇒ AnalLet (Γ[x::=y]) (e[x::=y])
\ a = AnalLet Γ e a

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

41
assumes \(\text{TransVar-eqvt: } \pi \cdot \text{TransVar} = \text{TransVar} \)
assumes \(\text{TransApp-eqvt: } \pi \cdot \text{TransApp} = \text{TransApp} \)
assumes \(\text{TransLam-eqvt: } \pi \cdot \text{TransLam} = \text{TransLam} \)
assumes \(\text{TransLet-eqvt: } \pi \cdot \text{TransLet} = \text{TransLet} \)
assumes \(\text{SuppTransLam: } \text{supp} (\text{TransLam} a v e) \subseteq \text{supp} e - \text{supp} v \)
assumes \(\text{SuppTransLet: } \text{supp} (\text{TransLet} b \Gamma e) \subseteq \text{supp} (\Gamma, e) - \text{atom ` domA} \Gamma \)

begin

nominal-function transform where
\( \text{transform a} (\text{App} e x) = \text{TransApp} a (\text{transform} (\text{PropApp} a) e) x \)
| \( \text{transform a} (\text{Lam} [x]. e) = \text{TransLam} a x (\text{transform} (\text{PropLam} a) e) \)
| \( \text{transform a} (\text{Var} x) = \text{TransVar} a x \)
| \( \text{transform a} (\text{Let} \Gamma e) = \text{TransLet} (\text{AnalLet} \Gamma e a) \)
| \( (\text{map-transform} \text{transform} (\text{PropLetHeap} (\text{AnalLet} \Gamma e a)) \Gamma) \)
| \( (\text{transform} (\text{PropLetBody} (\text{AnalLet} \Gamma e a)) e) \)
| \( \text{transform a} (\text{Bool} b) = (\text{Bool} b) \)
| \( \text{transform a} (\text{ scrut ?} e1 : e2) = (\text{transform} (\text{PropIfScrut} a) \text{ scrut ?} \text{transform a} e1 : \text{transform a} e2) \)
\(\langle \text{proof} \rangle\)

nominal-termination \(\langle \text{proof} \rangle\)

lemma supp-transform: \(\text{supp} (\text{transform a} e) \subseteq \text{supp} e \)
\(\langle \text{proof} \rangle\)

lemma \(\text{fv-transform}: \text{fv} (\text{transform a} e) \subseteq \text{fv} e \)
\(\langle \text{proof} \rangle\)

end

locale AbstractTransformSubst = AbstractTransform + AbstractAnalPropSubst +
assumes \(\text{TransVar-subst: } (\text{TransVar} a v)[x ::= y] = (\text{TransVar} a v[x ::= y]) \)
assumes \(\text{TransApp-subst: } (\text{TransApp} a e v)[x ::= y] = (\text{TransApp} a e[x ::= y] v[x ::= y]) \)
assumes \(\text{TransLam-subst: } \text{atom} v \not\in (x, y) \Rightarrow (\text{TransLam} a v e)[x ::= y] = (\text{TransLam} a v[x ::= y] e[x ::= y]) \)
assumes \(\text{TransLet-subst: } \text{atom ` domA} \Gamma \not\in (x, y) \Rightarrow (\text{TransLet} b \Gamma e)[x ::= y] = (\text{TransLet} b \Gamma[x ::= h = y] e[x ::= y]) \)

begin
lemmas subst-transform: \(\text{transform a} e[x ::= y] = \text{transform a} e[x ::= y] \)
\(\langle \text{proof} \rangle\)

end

locale AbstractTransformBound = AbstractAnalProp + supp-bounded-transform +
\text{constrains} \text{PropApp} :: \text{`a} \Rightarrow \text{`a}::\text{pure-cont-pl} \)
\text{constrains} \text{PropLetHeap} :: \text{`b}::\text{cont-pl} \Rightarrow \text{var} \Rightarrow \text{`a} \)
\text{constrains} \text{trans} :: \text{`c}::\text{cont-pl} \Rightarrow \text{exp} \Rightarrow \text{exp} \)
\text{fixes} \text{PropLetHeapTrans} :: \text{`b} \Rightarrow \text{var} \Rightarrow \text{`c} \)
assumes \(\text{PropLetHeapTrans-eqvt: } \pi \cdot \text{PropLetHeapTrans} = \text{PropLetHeapTrans} \)
assumes \(\text{TransBound-eqvt: } \pi \cdot \text{trans} = \text{trans} \)

begin
locale AbstractTransform BoundSubst = AbstractAnalPropSubst + AbstractTransformBound + 

assumes TransBound-subst: (trans a e)[x:::=y] = trans a e[x:::=y]

begin
  sublocale AbstractTransformSubst PropApp PropLam AnalLet PropLetBody PropLetHeap PropIfScrut
    (λ a . Var)
    (λ a . App)
    (λ a . Terms.Lam)
    (λ b Γ e . Let (map-transform trans (PropLetHeapTrans b) Γ) e)
  ⟨proof⟩

end

end

6.2 ArityTransform

theory ArityTransform
imports ArityAnalysisSig AbstractTransform ArityEtaExpansionSafe
begin

context ArityAnalysisHeap Eqvt
begin
sublocale AbstractTransformBound
  λ a . inc a
  λ a . pred a
  λ Δ e a . (a, Aheap Δ e · a)
  fst
  snd
  λ · . 0
  Aeta-expand

43
abbreviation transform-syn \( (\mathcal{T}_a) \) where \( \mathcal{T}_a \equiv \text{transform} \ a \)

lemma transform-simps:

\[
\begin{align*}
\mathcal{T}_a (\text{App} \ e \ x) &= \text{App} \ (\mathcal{T}_\text{inc} \ a \ e) \ x \\
\mathcal{T}_a (\text{Lam} \ [x], \ e) &= \text{Lam} \ [x], \ \mathcal{T}_\text{pred} \ a \ e \\
\mathcal{T}_a (\text{Var} \ x) &= \text{Var} \ x \\
\mathcal{T}_a (\text{Let} \ \Gamma \ e) &= \text{Let} \ (\text{map-} \mathcal{T} \ \text{A eta-expand} \ (\text{Aheap} \ \Gamma \ \text{e-a}) \ (\text{map-} \mathcal{T} \ \text{A exp} \ \lambda) \ \mathcal{T}_a) \\
\mathcal{T}_a (\text{Bo ol} \ b) &= \text{Bo ol} \ b \\
\mathcal{T}_a (\text{ scrut} \ ? \ e1 : e2) &= (\mathcal{T}_0 \ \text{ scrut} \ ? \ \mathcal{T}_a \ e1 : \mathcal{T}_a \ e2)
\end{align*}
\]

\[\langle \text{proof} \rangle\]

\[\text{end} \]

\[\text{end} \]

7 Arity Analysis Safety (without Cardinality)

7.1 ArityConsistent

theory ArityConsistent
imports ArityAnalysisSpec ArityStack ArityAnalysisStack
begin

context ArityAnalysisLetSafe
begin

type-synonym astate = (AEnv \times\times\text{Arity} \times\text{Arity list})

inductive stack-consistent :: Arity list \Rightarrow\Rightarrow stack \Rightarrow bool

where

\[
\text{stack-consistent} [] [] \\
| \text{As t a c k} \ S \subseteq a \Rightarrow \text{stack-consistent as} \ S \Rightarrow \text{stack-consistent (a\#as)} \ (\text{Alts} \ e1 e2 \# \ S) \\
| \text{stack-consistent as} \ S \Rightarrow \text{stack-consistent as} \ (\text{Upd} \ x \# \ S) \\
| \text{stack-consistent as} \ S \Rightarrow \text{stack-consistent as} \ (\text{Arg} \ x \# \ S)
\]

inductive-simps stack-consistent-foo[simp]:

\[
\text{stack-consistent} [] [] \text{stack-consistent (a\#as)} \ (\text{Alts} \ e1 e2 \# \ S) \text{stack-consistent as} \ (\text{Upd} \ x \# \ S) \\
\text{stack-consistent as} \ (\text{Arg} \ x \# \ S)
\]

inductive-cases [elim!]: stack-consistent as (\text{Alts} \ e1 e2 \# \ S)

inductive a-consistent :: astate \Rightarrow\Rightarrow\text{conf} \Rightarrow bool

where

a-consistentI:

\[
\text{ed om} \ ae \subseteq\subseteq \text{dom A} \ \Gamma \cup \text{up ds} \ S \\
\Rightarrow \text{As t a c k} \ S \subseteq\subseteq a \\
\Rightarrow \ (\text{Ab inds} \ \Gamma \ \text{ae} \cup \text{Aexp} \ e \ a \cup \text{AEstack as} \ S) \ f \uparrow \ (\text{dom A} \ \Gamma \cup \text{up ds} \ S) \subseteq\subseteq \text{ae}
\]

44
\( \Rightarrow \) stack-consistent as \( S \)
\( \Rightarrow \) a-consistent \((ae, a, as) \) \((\Gamma, e, S)\)

**Inductive Cases**

**a-consistentE:** a-consistent \((ae, a, as) \) \((\Gamma, e, S)\)

**lemma a-consistent-restrictA:**
assumes a-consistent \((ae, a, as) \) \((\Gamma, e, S)\)
assumes \( edom \ ae \subseteq V \)
shows a-consistent \((ae, a, as) \) \((\text{restrict} A V, e, S)\)
(proof)

**lemma a-consistent-edom-subsetD:**
a-consistent \((ae, a, as) \) \((\Gamma, e, S)\) \( \Rightarrow \) edom \( ae \subseteq \text{dom} A \cup \text{upds} S \)
(proof)

**lemma a-consistent-stackD:**
a-consistent \((ae, a, as) \) \((\Gamma, e, S)\) \( \Rightarrow \) \( A^{\text{stack}} S \) \( \subseteq a \)
(proof)

**lemma a-consistent-app1:**
a-consistent \((ae, a, as) \) \((\Gamma, \text{App} e x, S)\) \( \Rightarrow \) a-consistent \((ae, \text{inc} \cdot a, as) \) \((\Gamma, e, \text{Arg} x \# S)\)
(proof)

**lemma a-consistent-app2:**
assumes a-consistent \((ae, a, as) \) \((\Gamma, \text{Lam}[y]. e), \text{Arg} x \# S)\)
shows a-consistent \((ae, (\text{pred} \cdot a), as) \) \((\Gamma, e[x::=y], S)\)
(proof)

**lemma a-consistent-thunk-0:**
assumes a-consistent \((ae, a, as) \) \((\Gamma, \text{Var} x, S)\)
assumes \( \text{map-of} \ \Gamma \ x = \text{Some} \ e \)
assumes \( ae \ x = \text{up} \cdot 0 \)
shows a-consistent \((ae, 0, as) \) \((\text{delete} x \ \Gamma, e, \text{Upd} x \# S)\)
(proof)

**lemma a-consistent-thunk-once:**
assumes a-consistent \((ae, a, as) \) \((\Gamma, \text{Var} x, S)\)
assumes \( \text{map-of} \ \Gamma \ x = \text{Some} \ e \)
assumes \[ \text{simpl}: ae \ x = \text{up} \cdot u \]
assumes \( \text{heap-upds-ok} \ (\Gamma, S) \)
shows a-consistent \((\text{env-delete} x \ ae, u, as) \) \((\text{delete} x \ \Gamma, e, S)\)
(proof)

**lemma a-consistent-lamvar:**
assumes a-consistent \((ae, a, as) \) \((\Gamma, \text{Var} x, S)\)
assumes \( \text{map-of} \ \Gamma \ x = \text{Some} \ e \)
assumes \[ \text{simpl}: ae \ x = \text{up} \cdot u \]
shows a-consistent \((ae, u, as) \) \((x,e)\# \text{delete} x \ \Gamma, e, S)\)
(proof)
lemma
assumes a-consistent (\langle ae, a, as \rangle) (\Gamma, e, Upd x \neq S)
shows a-consistent-var: a-consistent (\langle ae, a, as \rangle) ((x, e) \neq \Gamma, e, S)
and a-consistent-UpdD: ae x = up-tha = 0
(proof)

lemma a-consistent-let:
assumes a-consistent (\langle ae, a, as \rangle) (\Gamma, Let \Delta e, S)
assumes atom \ ' domA \Delta \neq \Gamma
assumes atom \ ' domA \Delta \neq S
assumes edom ae \cap domA \Delta = \{\}
shows a-consistent (Aheap \Delta e-a \cup ae, a, as) (\Delta @ \Gamma, e, S)
(proof)

lemma a-consistent-if 1:
assumes a-consistent (\langle ae, a, as \rangle) (\Gamma, scrut ? e1 : e2, S)
shows a-consistent (ae, 0, a#as) (\Gamma, scrut, Alts e1 e2 \neq S)
(proof)

lemma a-consistent-if 2:
assumes a-consistent (\langle ae, a, a'\#as' \rangle) (\Gamma, Bool b, Alts e1 e2 \neq S)
shows a-consistent (ae, a', as') (\Gamma, if b then e1 else e2, S)
(proof)

lemma a-consistent-alts-on-stack:
assumes a-consistent (\langle ae, a, as \rangle) (\Gamma, Bool b, Alts e1 e2 \neq S)
obtains a' as' where as = a' \# as' a = 0
(proof)

lemma closed-a-consistent:
fo e = (\{\}:var set) \implies a-consistent (\bot, 0, []) ([], e, [])
(proof)

end

end

7.2 ArityTransformSafe

theory ArityTransformSafe
imports ArityTransform ArityConsistent ArityAnalysisSpec ArityEtaExpansionSafe Abstract-
Transform ConstOn
begin
locale CardinalityArityTransformation = ArityAnalysisLetSafeNoCard
begin
sublocale AbstractTransformBoundSubst
\lambda a . inc-a
\[ \lambda a. \ \text{pred}\ a \]
\[ \lambda \Delta e a. (a, \text{Heap} \ \Delta \cdot e \cdot a) \]

\[ \text{fst} \]
\[ \text{snd} \]
\[ \lambda \cdot. 0 \]
\[ \text{Aeta-expand} \]
\[ \text{snd} \]

\begin{verbatim}
abbreviation \text{cctransform} where \\
\text{cctransform} \equiv \text{transform}

lemma \text{supp-transform}: \text{supp} (\text{transform} \ a \ e) \subseteq \text{supp} \ e

interpretation \text{supp-bounded-transform transform}

fun \text{transform-alts} :: \text{Arity list} \Rightarrow \text{stack} \Rightarrow \text{stack}
where
  \text{transform-alts} - \text{[]} = \text{[]} \|
  \text{transform-alts} (a \# \text{as}) (\text{Alts} e2 \# S) = (\text{Alts} (\text{cctransform} \ a \ e2) \ (e2)) \ \\
  \# \text{transform-alts as} (x \# S) = x \# \text{transform-alts as} S

lemma \text{transform-alts-Nil} [\text{simp}]: \text{transform-alts} \text{[]} \ S = S

lemma \text{Astack-transform-alts} [\text{simp}]: \text{Astack} \ (\text{transform-alts as} S) = \text{Astack} \ S

lemma \text{fresh-star-transform-alts} [\text{intro}]: \ a \ast S \Rightarrow a \ast \text{transform-alts} \ S \ S

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

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

inductive \text{consistent} :: \text{astate} \Rightarrow \text{conf} \Rightarrow \text{bool} where
\text{consistentI} [\text{intro}!]:
  \text{a-consistent} (ae, a, as) (\Gamma, e, S)
  \Rightarrow (\bigwedge x. x \in \text{thunks} \ \Gamma \Rightarrow ae x = \text{wp} \cdot 0)
\Rightarrow \text{consistent} (ae, a, as) (\Gamma, e, S)

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

47
lemma closed-consistent:
assumes f e = (({})::var set)
shows consistent (⊥, 0, []) ([], e, [])
⟨proof⟩

lemma arity-transform-safe:
fixes c c'
assumes c ⇒ c' and ¬ boring-step c' and heap-upds-ok-conf c and consistent (ae,a,as)
c
shows ∃ ae' a' as', consistent (ae',a',as') c' ∧ a-transform (ae,a,as) c ⇒ a-transform (ae',a',as') c'
⟨proof⟩
end

end

8 Cardinality Analysis

8.1 Cardinality-Domain

type Cardinality-Domain
imports Launchbury.HOLCF-Utils
begin

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

type-synonym two = oneShot⊥
abbreviation many :: two where many ≡ up-notOneShot
abbreviation once :: two where once ≡ up-oneShot
abbreviation none :: two where none ≡ ⊥

lemma many-max[simp]: a ⊑ many ⟨proof⟩

lemma two-conj: c = many ∨ c = once ∨ 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 = (∀ x. if x ⊑ once then ⊥ else x)

lemma two-pred-simp: two-pred c = (if c ⊑ once then ⊥ else c) ⟨proof⟩

lemma two-pred-simps[simp]:
two-pred-many = many
two-pred-once = none
two-pred·none = none
⟨proof⟩

lemma two-pred-below-arg: two-pred · f ⊑ f
⟨proof⟩

lemma two-pred-none: two-pred·c = none ↔ c ⊑ once
⟨proof⟩

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

lemma record-call-simp: (record-call x · f) x' = (if x = x' then two-pred · (f x') else f x')
⟨proof⟩

lemma record-call[simp]: (record-call x · f) x = two-pred · (f x)
⟨proof⟩

lemma record-call-other[simp]: x' ≠ x ⇒ (record-call x · f) x' = f x'
⟨proof⟩

lemma record-call-below-arg: record-call x · f ⊑ f
⟨proof⟩

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

lemma two-add-simp: two-add·x·y = (if x ⊑ ⊥ then y else (if y ⊑ ⊥ then x else many))
⟨proof⟩

lemma two-pred-two-add-once: c ⊑ two-pred·(two-add·once·c)
⟨proof⟩

end

8.2 CardinalityAnalysisSig

theory CardinalityAnalysisSig
imports Arity AEnv Cardinality Domain SestoftConf
begin

locale CardinalityPrognosis =
   fixes prognosis :: AEnv ⇒ Arity list ⇒ Arity ⇒ conf ⇒ (var ⇒ two)

locale CardinalityHeap =
   fixes cHeap :: heap ⇒ exp ⇒ Arity ⇒ (var ⇒ two)
end
8.3 CardinalityAnalysisSpec

theory CardinalityAnalysisSpec

imports ArityAnalysisSpec CardinalityAnalysisSig ConstOn

begin

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

locale CardinalityPrognosisShape = CardinalityPrognosis +
assumes prognosis-env-cong: ae f |' domA (Γ = ae' f |' domA) Γ ⇒ prognosis ae as u (Γ, e, S) = prognosis ae' as u (Γ, e, S)
assumes prognosis-reorder: map-of Γ = map-of Δ ⇒ prognosis ae as u (Γ, e, S) = prognosis ae as u (Δ, e, S)
assumes prognosis-ap: const-on (prognosis ae as a (Γ, e, S)) (ap S) many
assumes prognosis-upd: prognosis ae as u (Γ, e, S) ⊆ prognosis ae as u (Γ, e, Upd x ≠ S)
assumes prognosis-not-called: ae x = ⊥ ⇒ prognosis ae as a (Γ, e, S) ⊆ prognosis ae as a (delete x Γ, e, S)
assumes prognosis-called: once ⊆ prognosis ae as a (Γ, Var x, S) x

locale CardinalityPrognosisApp = CardinalityPrognosis +
assumes prognosis-App: prognosis ae as (inc-a) (Γ, e, Arg x ≠ S) ⊆ prognosis ae as a (Γ, App e x, S)

locale CardinalityPrognosisLam = CardinalityPrognosis +
assumes prognosis-subst-Lam: prognosis ae as (pred-a) (Γ, e[y::=x]), S) ⊆ prognosis ae as a (Γ, Lam [y]. e, Arg x ≠ S)

locale CardinalityPrognosisVar = CardinalityPrognosis +
assumes prognosis-Var-lam: map-of Γ x = Some e ⇒ ae x = wp u ⇒ isVal e ⇒ prognosis ae as u (Γ, e, S) ⊆ record-call x · (prognosis ae as a (Γ, Var x, S))
assumes prognosis-Var-thunk: map-of Γ x = Some e ⇒ ae x = wp u ⇒ ¬ isVal e ⇒ prognosis ae as u (delete x Γ, e, Upd x ≠ S) ⊆ record-call x · (prognosis ae as a (Γ, Var x, S))
assumes prognosis-Var2: isVal e ⇒ x ∉ domA Γ ⇒ prognosis ae as 0 ((x, e) ≠ Γ, e, S) ⊆ prognosis ae as 0 (Γ, e, Upd x ≠ S)

locale CardinalityPrognosisIfThenElse = CardinalityPrognosis +
assumes prognosis-IfThenElse: prognosis ae (a#as) 0 (Γ, scrut, Alts e1 e2 ≠ S) ⊆ prognosis ae as a (Γ, scrut ? e1 : e2, S)
assumes prognosis-Alts: prognosis ae as a (Γ, if b then e1 else e2, S) ⊆ prognosis ae (a#as) 0 (Γ, Bool b, Alts e1 e2 ≠ S)

locale CardinalityPrognosisLet = CardinalityPrognosis + CardinalityHeap + ArityAnalysisHeap +
assumes prognosis-Let:
atom · domA Δ #* Γ ⇒ atom · domA Δ #* S ⇒ edom ae ⊆ domA Γ ∪ updS S ⇒ prognosis (Aheap Δ e-a ∪ ae) as a (Δ @ Γ, e, S) ⊆ cHeap Δ e-a ∪ prognosis ae as a (Γ, Terms.Let Δ e, S)
locale CardinalityHeapSafe = CardinalityHeap + ArityAnalysisHeap +
  assumes Aheap-heap3: \( x \in \text{thunks} \Gamma \implies \text{many} \subseteq (c\text{Heap} \Gamma e\cdot a) \ x \mapsto (\text{Aheap} \Gamma e\cdot a) \ x = \up\cdot 0 \)
  assumes edom-cHeap: \( \text{edom} (c\text{Heap} \Delta e\cdot a) = \text{edom} (\text{Aheap} \Delta e\cdot a) \)

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

end

8.4 NoCardinalityAnalysis

theory NoCardinalityAnalysis
imports CardinalityAnalysisSpec ArityAnalysisStack
begin

locale NoCardinalityAnalysis = ArityAnalysisLetSafe +
  assumes Aheap-thunk: \( x \in \text{thunks} \Gamma \implies (\text{Aheap} \Gamma e\cdot a) \ x = \up\cdot 0 \)
begin

definition a2c :: Arity \(\bot\) \(\rightarrow\) two where a2c = (\(\Lambda\, a\). if \( a \subseteq \bot \) then \(\bot\) else many)
lemma a2c-simp: a2c\( a \) = (if \( a \subseteq \bot \) then \(\bot\) else many)
  ⟨proof⟩

lemma a2c-eqt[eqvt]: \(\pi \cdot a2c = a2c\)
  ⟨proof⟩

definition ae2ce :: AEnv \(\Rightarrow\) (var \(\Rightarrow\) two) where ae2ce\( ae \)\( x \) = a2c\(\cdot (ae \ x)\)
lemma ae2ce-cont: cont ae2ce
  ⟨proof⟩
lemmas cont-compose[OF ae2ce-cont, cont2cont, simp]

lemma ae2ce-eqt[eqvt]: \(\pi \cdot ae2ce\ x = ae2ce\ (\pi \cdot ae)\ (\pi \cdot x)\)
  ⟨proof⟩

lemma ae2ce-to-env-restr: ae2ce\( ae = (\lambda-\cdot \text{many}) f|' \text{edom} ae\)
  ⟨proof⟩

end
lemma edom-ae2ce[simp]: \( \text{edom} \ (\text{ae2ce} \ ae) = \text{edom} \ ae \)

\[\langle \text{proof} \rangle\]

definition cHeap :: heap \Rightarrow exp \Rightarrow \text{Arity} \Rightarrow (\text{var} \Rightarrow \text{two})
  where cHeap \( \Gamma \ e \) = (\( \Lambda \ a. \ \text{ae2ce} \ (\text{Aheap} \ \Gamma \ e \cdot a) \))

lemma cHeap-simp[simp]: cHeap \( \Gamma \ e \cdot a \) = ae2ce \( (\text{Aheap} \ \Gamma \ e \cdot a) \)

\[\langle \text{proof} \rangle\]

sublocale CardinalityHeap cHeap \langle proof \rangle

sublocale CardinalityHeapSafe cHeap Aheap

\[\langle \text{proof} \rangle\]

fun prognosis where
  prognosis ae as a (\( \Gamma, \ e, \ S \)) = ((\lambda \cdot. \ \text{many} \ f \mid \left. S \right) \ (\text{edom} \ (\text{ABinds} \ \Gamma \ ae) \cup \text{edom} \ (\text{Aexp} \ e \cdot a) \cup \text{edom} \ (\text{AEstack as} \ S)))

lemma record-all-noop[simp]:
  record-call x\cdot(\left. (\lambda \cdot. \ \text{many} \ f) \right) \ S = (\lambda \cdot. \ \text{many} \ f) \mid \left. S \right)

\[\langle \text{proof} \rangle\]

lemma const-on-restr-constI[intro]:
  \( S' \subseteq S \implies \text{const-on} \ ((\lambda \cdot. \ x) \ f \mid \left. S \right) \ S' \ x \)

\[\langle \text{proof} \rangle\]

lemma ap-subset-edom-AEstack: ap \( S \subseteq \text{atom} \ (\text{AEstack as} \ S) \)

\[\langle \text{proof} \rangle\]

sublocale CardinalityPrognosis prognosis \langle proof \rangle

sublocale CardinalityPrognosisShape prognosis

\[\langle \text{proof} \rangle\]

sublocale CardinalityPrognosisApp prognosis

\[\langle \text{proof} \rangle\]

sublocale CardinalityPrognosisLam prognosis

\[\langle \text{proof} \rangle\]

sublocale CardinalityPrognosisVar prognosis

\[\langle \text{proof} \rangle\]

sublocale CardinalityPrognosisIfThenElse prognosis

\[\langle \text{proof} \rangle\]

sublocale CardinalityPrognosisLet prognosis cHeap Aheap

52
\textbf{(proof)}

\textbf{sublocale} \textit{CardinalityPrognosisEdom prognosis} \textbf{(proof)}

\textbf{sublocale} \textit{CardinalityPrognosisSafe prognosis cHeap Aheap Aexp (proof)}
\textbf{end}
\textbf{end}

\textbf{8.5 CardArityTransformSafe}

\textbf{theory} \textit{CardArityTransformSafe}

\textbf{imports} \textit{ArityTransform CardinalityAnalysisSpec AbstractTransform Sestoft SestoftGC ArityEta-ExpansionSafe ArityAnalysisStack ArityConsistent}

\textbf{begin}

\textbf{context} \textit{CardinalityPrognosisSafe}

\textbf{begin}

\textbf{sublocale} \textit{AbstractTransformBoundSubst}

\begin{align*}
\lambda & a . \text{inc-}a \\
\lambda & a . \text{pred-}a \\
\lambda & \Delta e a . (a, \text{Aheap} \Delta e a) \\
\text{fst} \\
\text{snd} \\
\lambda & . \text{-}0 \\
\text{Aeta-expand} \\
\text{snd} \\
\end{align*}

\textbf{(proof)}

\textbf{abbreviation} \texttt{ccTransform} where \texttt{ccTransform} \equiv \texttt{transform}

\textbf{lemma} \texttt{supp-transform}: \texttt{supp (transform a e) \subseteq supp e} \textbf{(proof)}

\textbf{interpretation} \texttt{supp-bounded-transform transform} \textbf{(proof)}

\textbf{type-synonym} \texttt{tstate} = (\texttt{AEnv} \times (\texttt{var} \Rightarrow \texttt{two}) \times \texttt{Arity} \times \texttt{Arity list} \times \texttt{var list})

\textbf{fun} \texttt{transform-alts :: Arity list} \Rightarrow \texttt{stack} \Rightarrow \texttt{stack}

\textbf{where}

\begin{align*}
\text{transform-alts} \cdot [] & = [] \\
\text{transform-alts} (a \# as) (\texttt{Alts e1 e2} \# S) & = (\texttt{Alts (ccTransform a e1) (ccTransform a e2)}) \\
\texttt{#} \text{transform-alts} as S & = \text{transform-alts} as (x \# S) = x \# \text{transform-alts} as S \\
\textbf{lemma} \texttt{transform-alts-Nil[simp]}: \texttt{transform-alts} [] S = S \textbf{(proof)}
\end{align*}

53
lemma Astack-transform-alts[simp]:
Astack (transform-alts as S) = Astack S
⟨proof⟩

lemma fresh-star-transform-alts[intro]: a ⨆ S ⇒ a ⨆ transform-alts as S
⟨proof⟩

fun a-transform :: astate ⇒ conf ⇒ conf
where a-transform (ae, a, as) (Γ, e, S) =
(map-transform A eta-expand ae (map-transform ccTransform ae Γ),
ccTransform a e,
transform-alts as S)

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

fun add-dummies-conf :: var list ⇒ conf ⇒ conf
where add-dummies-conf l (Γ, e, S) = (Γ, e, S @ map Dummy (rev l))

fun conf-transform :: tstate ⇒ conf ⇒ conf
where conf-transform (ae, ce, a, as, r) c = add-dummies-conf r ((a-transform (ae, a, as)
(restr-conf (− set r) c)))

inductive consistent :: tstate ⇒ conf ⇒ bool where
consistentI[intro!]:
a-consistent (ae, a, as) (restr-conf (− set r) (Γ, e, S))
⇒ edom ce = edom ae
⇒ prognosis ae as a (Γ, e, S) ⊆ ce
⇒ (∃ x. x ∈ thunks Γ ⇒ many ⊆ ce x ⇒ ae x = up 0)
⇒ set r ⊆ (domA Γ ∪ upds S) − edom ce
⇒ consistent (ae, ce, a, as, r) (Γ, e, S)
inductive-cases consistentE[elim!]: consistent (ae, ce, a, as) (Γ, e, S)

lemma closed-consistent:
assumes fv e = ({}) :: var set
shows consistent (⊥, ⊥, ⊥, [], []) ([], e, [])
⟨proof⟩

lemma card-arity-transform-safe:
fixes c c’
assumes c ⇒ c’ and ¬ boring-step c’ and heap-upds-ok-conf c and consistent (ae,ce,a,as,r) c
s shows ∃ ae’ ce’ a’ as’ r’. consistent (ae’,ce’,a’,as’,r’) c’ ∧ conf-transform (ae,ce,a,as,r) c
g⇒c’ 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 ⇒ bool where
downset xss = (∀ x n. x ∈ xss → take n x ∈ xss)

lemma downsetE[elim]:
downset xss ⇒ xs ∈ xss ⇒ butlast xs ∈ xss
(proof)

lemma downset-appendE[elim]:
downset xss ⇒ xs @ ys ∈ xss ⇒ xs ∈ xss
(proof)

lemma downset-hdE[elim]:
downset xss ⇒ xs ∈ xss ⇒ xs ≠ [] ⇒ [hd xs] ∈ xss
(proof)

lemma downsetI[intro]:
assumes ∃ xs. xs ∈ xss ⇒ xs ≠ [] ⇒ butlast xs ∈ xss
shows downset xss
(proof)

lemma [simp]: downset {[]} (proof)

lemma downset-mapI: downset xss ⇒ 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 ⊆ 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-tree

9.1.3 Deconstructors

\textbf{lift-definition possible} :: 'a tree ⇒ 'a ⇒ bool
\textbf{is} \ λ \ xs \ x. \ \exists \ xs. \ x # xs \in \ xss\langle\text{proof}\rangle

\textbf{lift-definition nxt} :: 'a tree ⇒ 'a ⇒ 'a tree
\textbf{is} \ λ \ xs \ x. \ \text{insert} \ [[xs | xs. \ x # xs \in \ xss}\langle\text{proof}\rangle

9.1.4 Trees as set of paths

\textbf{lift-definition paths} :: 'a tree ⇒ 'a list set is (\lambda x. x)\langle\text{proof}\rangle

\textbf{lemma paths-inj}: paths t = paths t' \implies t = t' \langle\text{proof}\rangle

\textbf{lemma paths-injs-simps[simp]}: paths t = paths t' \iff t = t' \langle\text{proof}\rangle

\textbf{lemma paths-nil[simp]}: [] \in paths t \langle\text{proof}\rangle

\textbf{lemma paths-not-empty[simp]}: (paths t = {}) \iff False \langle\text{proof}\rangle

\textbf{lemma paths-Cons-nxt}: possible t x \implies xs \in paths (nxt t x) \implies (x # xs) \in paths t
\langle\text{proof}\rangle

\textbf{lemma paths-Cons-nxt-iff}:
possible t x \implies xs \in paths (nxt t x) \iff (x # xs) \in paths t
\langle\text{proof}\rangle

\textbf{lemma possible-mono}:
paths t \subseteq paths t' \implies possible t x \implies possible t' x
\langle\text{proof}\rangle

\textbf{lemma nxt-mono}:
paths t \subseteq paths t' \implies paths (nxt t x) \subseteq paths (nxt t' x)
\langle\text{proof}\rangle

\textbf{lemma tree-eqI}:(\land x xs. \ x # xs \in paths t \iff x # xs \in paths t') \implies t = t'
\langle\text{proof}\rangle

\textbf{lemma paths-nxt[elim]}:
assumes xs \in paths (nxt t x)
obtains x # xs \in paths t \mid xs = []
\langle\text{proof}\rangle

\textbf{lemma Cons-path}: x \# xs \in paths t \iff possible t x \land xs \in paths (nxt t x)
\langle\text{proof}\rangle

56
lemma Cons-path I[intro]:
  assumes \possible t x \leftrightarrow \possible t' x
  assumes \possible t x \Rightarrow \possible t' x \Rightarrow \xs \in \paths (\nxt t x) \leftrightarrow \xs \in \paths (\nxt t' x)
shows \xs \in \paths t \leftrightarrow x \# \xs \in \paths t' 
(proof)

lemma paths-nxt-eq: \xs \in \paths (\nxt t x) \leftrightarrow \xs = [] \lor x \# \xs \in \paths t 
(proof)

lemma tree-coinduct:
  assumes \P t t'
  assumes \bigcap t t' x . \P t t' \Rightarrow \possible t x \leftrightarrow \possible t' x
  assumes \bigcap t t' x . \P t t' \Rightarrow \possible t x \Rightarrow \possible t' x \Rightarrow \P (\nxt t x) (\nxt t' x)
shows t = t' 
(proof)

9.1.5 The carrier of a tree

lift-definition carrier :: 'a tree \Rightarrow 'set is \lambda xss. \bigcup (set \ ' xss) 
(proof)

lemma carrier-mono: \paths t \subseteq \paths t' \Rightarrow carrier t \subseteq carrier t' 
(proof)

lemma carrier-possible:
  \possible t x \Rightarrow x \in carrier t 
(proof)

lemma carrier-possible-subset:
  carrier t \subseteq A \Rightarrow \possible t x \Rightarrow x \in A 
(proof)

lemma carrier-nxt-subset:
  carrier (\nxt t x) \subseteq carrier t 
(proof)

lemma Union-paths-carrier: (\bigcup \xs \in \paths t . set x) = carrier t 
(proof)

9.1.6 Repeatable trees

definition repeatable where repeatable t = (\forall x . \possible t x \rightarrow \nxt t x = t)

lemma nxt-repeatable[simp]: repeatable t \Rightarrow \possible t x \Rightarrow \nxt t x = t 
(proof)

9.1.7 Simple trees

lift-definition empty :: 'a tree is [ ] 
(proof)

lemma possible-empty[simp]: \possible empty x' \leftrightarrow False 
(proof)

lemma nxt-not-possible[simp]: \neg \possible t x \Rightarrow \nxt t x = 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 => 'a ttree is \lambda x. {[}, [x]}

proof

lemma possible-single [simp]: possible (single x) x' <-> x = x'

proof

lemma nxt-single [simp]: nxt (single x) x' = 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 => 'a ttree is \lambda x. range (\lambda n. replicate n x)

proof

lemma possible-many-calls [simp]: possible (many-calls x) x' <-> x = x'

proof

lemma nxt-many-calls [simp]: nxt (many-calls x) x' = (if x' = x then many-calls x 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 ttree is UNIV

proof

lemma possible-anything [simp]: possible anything x' <-> True

proof

lemma nxt-anything [simp]: nxt anything x = anything

proof

lemma paths-anything [simp]: paths anything = UNIV

proof

58
lemma carrier-anything [simp]:
carrier anything = UNIV
⟨proof⟩

lift-definition many-among :: 'a set ⇒ 'a tree is λ S. {xs . set xs ⊆ 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 ⇒ 'a tree ⇒ 'a tree (infixl ∩ 80)
is (∩)
⟨proof⟩

lemma paths-intersect [simp]:
paths (t ∩ t') = paths t ∩ paths t'
⟨proof⟩

lemma carrier-intersect:
carrier (t ∩ t') ⊆ carrier t ∩ carrier t'
⟨proof⟩

9.1.9 Disjoint union of trees

lift-definition either :: 'a tree ⇒ 'a tree ⇒ 'a tree (infixl ⊕ 80)
is (∪)
⟨proof⟩

lemma either-empty1 [simp]: empty ⊕ t = t
⟨proof⟩

lemma either-empty2 [simp]: t ⊕ empty = t
⟨proof⟩

lemma either-sym [simp]: t ⊕ t2 = t2 ⊕ t
⟨proof⟩

lemma either-idem [simp]: t ⊕ t = t
⟨proof⟩

lemma possible-either [simp]:
possible (t ⊕ t') x ←→ possible t x ∨ possible t' x
⟨proof⟩

lemma nxt-either [simp]:
nxt (t ⊕ t') x = nxt t x ⊕ nxt t' x
⟨proof⟩

lemma paths-either [simp]:
paths (t ⊕ t') = paths t ∪ paths t'
⟨proof⟩

lemma carrier-either [simp]:
carrier (t ⊕ t') = carrier t ∪ carrier t'
⟨proof⟩
lemma either-contains-arg1: paths t ⊆ paths (t ⊕ t')
  ⟨proof⟩
lemma either-contains-arg2: paths t' ⊆ paths (t ⊕ t')
  ⟨proof⟩
lift-definition Either :: 'a ttree set ⇒ 'a ttree is λ S. insert [] (⋃ S)
  ⟨proof⟩
lemma paths-Either: paths (Either ts) = insert [] (⋃ (paths ' ts))
  ⟨proof⟩

9.1.10 Merging of trees
lemma ex-ex-eq-hint: (∃ x. (∃ xs ys. x = f xs ys ∧ P xs ys) ∧ Q x) ←→ (∃ xs ys. Q (f xs ys) ∧ P xs ys)
  ⟨proof⟩
lift-definition both :: 'a ttree ⇒ 'a ttree ⇒ 'a ttree (infixl ⊗⊗ 86)
is λ xss yss . ⋃ {xs ⊗ ys | xs ys. xs ∈ xss ∧ ys ∈ yss}
  ⟨proof⟩
lemma both-assoc[simp]: t ⊗⊗ (t' ⊗⊗ t'') = (t ⊗⊗ t') ⊗⊗ t''
  ⟨proof⟩
lemma both-comm: t ⊗⊗ t' = t' ⊗⊗ t
  ⟨proof⟩
lemma both-empty1[simp]: empty ⊗⊗ t = t
  ⟨proof⟩
lemma both-empty2[simp]: t ⊗⊗ empty = t
  ⟨proof⟩
lemma paths-both: xs ∈ paths (t ⊗⊗ t') ←→ (∃ ys ∈ paths t. ∃ zs ∈ paths t'. xs ∈ ys ⊗ zs)
  ⟨proof⟩
lemma both-contains-arg1: paths t ⊆ paths (t ⊗ t')
  ⟨proof⟩
lemma both-contains-arg2: paths t' ⊆ paths (t ⊗ t')
  ⟨proof⟩
lemma both-mono1:
paths t ⊆ paths t' ⇒ paths (t ⊗ t') ⊆ paths (t' ⊗ t'')
  ⟨proof⟩
lemma both-mono2:
paths \( t \subseteq \text{paths} \ t' \implies \text{paths} \ (t'' \otimes \ t) \subseteq \text{paths} \ (t'' \otimes \ t') \)

\langle \text{proof} \rangle

\textbf{lemma} \ possible-both[simp]: possible \ (t \otimes \ t') \ x \leftrightarrow \text{possible} \ t \ x \lor \text{possible} \ t' \ x

\langle \text{proof} \rangle

\textbf{lemma} \ \text{nxt-both}:
\[
\text{nxt} \ (t' \otimes \ t) \ x = (\text{if possible} \ t' \ x \land \text{possible} \ t \ x \text{ then nxt} \ t' \ x \otimes t' \otimes \text{nxt} \ t \ x \text{ else}
\]
\[
\text{if possible} \ t' \ x \text{ then nxt} \ t' \ x \otimes t \text{ else}
\]
\[
\text{if possible} \ t \ x \text{ then t' \otimes \text{nxt} \ t \ x \text{ else empty})}
\]

\langle \text{proof} \rangle

\textbf{lemma} \ Cons-both:
\[
x \neq \ xs \in \text{paths} \ (t' \otimes \ t) \leftrightarrow (\text{if possible} \ t' \ x \land \text{possible} \ t \ x \text{ then xs } \in \text{paths} \ (nxt \ t' \ x \otimes t) \lor \text{xs } \in \text{paths} \ (t' \otimes \text{nxt} \ t \ x) \text{ else}
\]
\[
\text{if possible} \ t' \ x \text{ then xs } \in \text{paths} \ (nxt \ t' \ x \otimes t) \text{ else}
\]
\[
\text{if possible} \ t \ x \text{ then \ False
\]

\langle \text{proof} \rangle

\textbf{lemma} \ Cons-both-possible-leftE: possible \ t \ x \implies \text{xs } \in \text{paths} \ (nxt \ t \ x \otimes t) \implies \text{x\#xs } \in \text{paths} \ (t \otimes t')

\langle \text{proof} \rangle

\textbf{lemma} \ Cons-both-possible-rightE: possible \ t' \ x \implies \text{xs } \in \text{paths} \ (t \otimes nxt \ t' \ x) \implies \text{x\#xs } \in \text{paths} \ (t \otimes t')

\langle \text{proof} \rangle

\textbf{lemma} \ either-both-distr[simp]:
\[
t' \otimes (t \oplus t' \otimes t'') = t' \otimes (t \oplus t')
\]

\langle \text{proof} \rangle

\textbf{lemma} \ either-both-distr2[simp]:
\[
t' \otimes t \oplus t' \otimes t'' = t' \otimes (t \oplus t'') \otimes t
\]

\langle \text{proof} \rangle

\textbf{lemma} \ nxt-both-repeatable[simp]:
\text{assumes \ [simp]: repeatable} \ t'
\text{assumes \ [simp]: possible} \ t' \ x
\text{shows \ nxt} \ (t' \otimes t) \ x = t' \otimes (t \oplus \text{nxt} \ t \ x)

\langle \text{proof} \rangle

\textbf{lemma} \ nxt-both-many-calls[simp]:\ nxt \ (\text{many-calls} \ x \otimes \ t) \ x = \text{many-calls} \ x \otimes (t \oplus \text{nxt} \ t \ x)

\langle \text{proof} \rangle

\textbf{lemma} \ repeatable-both-self[simp]:
\text{assumes \ [simp]: repeatable} \ t
\text{shows} \ t \otimes t = t

\langle \text{proof} \rangle
proof

lemma repeatable-both-both [simp]:
assumes repeatable t
shows t \otimes t' \otimes t = t \otimes t'
(proof)

lemma repeatable-both-both2 [simp]:
assumes repeatable t
shows t' \otimes t \otimes t = t' \otimes t
(proof)

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
(proof)

lemma repeatable-both-nxt:
assumes t' \otimes t = t'
shows t' \otimes t'' \otimes t = t' \otimes t''
(proof)

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

9.1.11 Removing elements from a tree

lift-definition without :: 'a \Rightarrow 'a ttree \Rightarrow 'a ttree
is \lambda \ x \ ss. \ filter (\lambda \ x'. \ x' \neq x) \ ' \ ss
(proof)

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

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

lift-definition ttree-restr :: 'a set \Rightarrow 'a ttree \Rightarrow 'a ttree is \lambda \ S \ ss. \ filter (\lambda \ x'. \ x' \in\ S) \ ' \ ss
(proof)

lemma filter-paths-conv-free-restr:
\text{filter} \ (\lambda \ x'. \ x' \in S) \cdot \text{paths} \ t = \text{paths} \ (\text{tree-restr} \ S \ t) \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{filter-paths-conv-free-restr} 2:\nfilter \ (\lambda \ x'. \ x' \notin S) \cdot \text{paths} \ t = \text{paths} \ (\text{tree-restr} \ (- S) \ t) \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{filter-paths-conv-free-without}:\nfilter \ (\lambda \ x'. \ x' \neq y) \cdot \text{paths} \ t = \text{paths} \ (\text{without} \ y \ t) \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{three-restr-is-empty}: \text{carrier} \ t \cap S = \{\} \implies \text{three-restr} \ S \ t = \text{empty} \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{three-restr-noop}: \text{carrier} \ t \subseteq S \implies \text{three-restr} \ S \ t = t \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{three-restr-tree-restr}[\text{simpl}]:\n\text{three-restr} \ S \ (\text{three-restr} \ S' \ t) = \text{three-restr} \ (S' \cap S) \ t \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{three-restr-both}:\n\text{three-restr} \ S \ (t \otimes t') = \text{three-restr} \ S \ t \otimes \text{three-restr} \ S \ t' \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{three-restr-nxt-subset}: x \in S \implies \text{paths} \ (\text{three-restr} \ S \ (\text{nxt} \ t \ x)) \subseteq \text{paths} \ (\text{nxt} \ (\text{three-restr} \ S \ t) \ x) \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{three-restr-nxt-subset2}: x \notin S \implies \text{paths} \ (\text{three-restr} \ S \ (\text{nxt} \ t \ x)) \subseteq \text{paths} \ (\text{three-restr} \ S \ t) \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{three-restr-possible}: x \in S \implies \text{possible} \ t \ x \implies \text{possible} \ (\text{three-restr} \ S \ t) \ x \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{three-restr-possible2}: \text{possible} \ (\text{three-restr} \ S \ t') \ x \implies x \in S \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{carrier-tree-restr}[\text{simpl}]:\n\text{carrier} \ (\text{three-restr} \ S \ t) = S \cap \text{carrier} \ t \ \langle \text{proof} \rangle

\textbf{9.1.12} Multiple variables, each called at most once

\textbf{lift-definition} \ \text{singles} :: \ \text{'a set} \Rightarrow \text{'a tree is} S \cdot \{x \in S. \ \text{length} \ (\text{filter} \ (\lambda \ x'. \ x' = x) \ xs) \leq t\} \ \langle \text{proof} \rangle

\textbf{lemma} \ \text{possible-singles}[\text{simpl}]: \text{possible} \ (\text{singles} \ S) \ x \ \langle \text{proof} \rangle
lemma length-filter-mono[intro]:
assumes \( (\forall x. P x \implies Q x) \)
shows \( \text{length (filter } P \text{ } xs) \leq \text{length (filter } Q \text{ } xs) \)
(proof)

lemma nxt-singles[simp]: \( \text{nxt (singles } S \text{) } x' = \) (if \( x' \in S \) then without \( x' \) (singles \( S \)) else singles \( S \))
(proof)

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

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

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

9.1.13 Substituting trees for every node

definition f-nxt :: ('a ⇒ 'a tree) ⇒ 'a set ⇒ 'a ⇒ ('a ⇒ 'a tree)
where f-nxt \( f \text{ } T \text{ } x = \) (if \( x \in T \) then \( f \text{ } \text{ } x:=\text{empty} \) else \( f \))

fun substitute' :: ('a ⇒ 'a tree) ⇒ 'a set ⇒ 'a tree ⇒ 'a list ⇒ bool where
\( \text{substitute'}-\text{Nil: substitute'} \text{ } f \text{ } T \text{ } t \text{ } [] \leftarrow \text{True} \)
\( | \text{substitute'}-\text{Cons: substitute'} \text{ } f \text{ } T \text{ } (x\#xs) \leftarrow \text{ If possible } t \text{ } x \cdot \text{substitute'} (f-nxt \text{ } f \text{ } T \text{ } x \text{ } x') \text{ } T \text{ } (\text{nxt } t \text{ } x \otimes \otimes \text{ } f) \text{ } xs \)

lemma f-nxt-mono1: \( (\forall x. \text{paths } (f x) \subseteq \text{paths } (f' x)) \implies \text{paths } (f-nxt \text{ } f \text{ } T \text{ } x \text{ } x') \subseteq \text{paths } (f-nxt \text{ } f' \text{ } T \text{ } x \text{ } x') \)
(proof)

lemma f-nxt-empty-set[simp]: \( f-nxt \text{ } f \text{ } \text{ } \{} \text{ } \text{ } x \text{ } \text{ } = f \)
(proof)

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

lift-definition substitute :: ('a ⇒ 'a tree) ⇒ 'a set ⇒ 'a tree ⇒ 'a tree
is \( \lambda f \text{ } T \text{ } t. \text{Collect (substitute'} \text{ } f \text{ } T \text{ } t) \)
(proof)

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

lemmas substitute-induct[case-names Nil Cons] = substitute'.induct
lemmas substitute-simps[simp] = substitute'.simps[unfolded elim-substitute']

lemma substitute-mono2:
  assumes paths t ⊆ paths t'
  shows paths (substitute f T t) ⊆ paths (substitute f T t')
(\proof)

lemma substitute-mono1:
  assumes \( \forall x. \text{paths} (f x) \subseteq \text{paths} (\text{f' x}) \)
  shows paths (substitute f T t) ⊆ paths (substitute f' T t)
(\proof)

lemma substitute-monoT:
  assumes T ⊆ T'
  shows paths (substitute f T' t) ⊆ paths (substitute f T t)
(\proof)

lemma substitute-contains-arg: paths t ⊆ paths (substitute f T t)
(\proof)

lemma possible-substitute[simp]: possible (substitute f T t) x ↔ possible t x
(\proof)

lemma nxt-substitute[simp]: possible t x \(\Rightarrow\) nxt (substitute f T t) x = substitute (f-nxt f T x) T (nxt t x \otimes f x)
(\proof)

lemma substitute-either: substitute f T (t ⊕ t') = substitute f T t ⊕ substitute f T t'
(\proof)

lemma f-nxt-T-delete:
  assumes f x = empty
  shows f-nxt f (T - \{x\}) x' = f-nxt f T x'
(\proof)

lemma f-nxt-empty[simp]:
  assumes f x = empty
  shows f-nxt f T x' x = empty
(\proof)

lemma f-nxt-empty'[simp]:
  assumes f x = empty
  shows f-nxt f T x = f
(\proof)
lemma substitute-T-delete:
  assumes \( f \ x = \text{empty} \)
  shows \( \text{substitute} \ f \ (T - \{x\}) \ t = \text{substitute} \ f \ T \ t \)
(\text{proof})

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

lemma substitute-only-empty-both: \( \text{const-on} \ f \ (\text{carrier} \ t') \ \text{empty} \Rightarrow \text{substitute} \ f \ T \ (t \otimes t') = \text{substitute} \ f \ T \ t \otimes t' \)
(\text{proof})

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

lemma repeatable-\(f\)-nxt-upd[simp]:
  repeatable \(f \ x \) \(\Rightarrow\) repeatable \((f\text{-nxt} f T \ x' \ x)\)
(\text{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 \(xs \in \text{paths} \ (\text{substitute} \ (f(\ x := \text{empty})) \ T \ t)\)
(\text{proof})

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

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

lemma \(f\)-nxt-cong: \(f \ x' = f' \ x' \Rightarrow f\text{-nxt} f T \ x \ x' = f\text{-nxt} f' T \ x \ x'\)
(\text{proof})

lemma substitute-cong':
  assumes \(xs \in \text{paths} \ (\text{substitute} f T \ t)\)
  assumes \(\forall x. \ n. \ x \in A \Rightarrow \text{carrier} \ (f \ x) \subseteq A\)
  assumes \(\text{carrier} \ t \subseteq A\)
\begin{align*}
\text{assumes } & \bigwedge x. x \in A \implies f x = f' x \\
\text{shows } & xs \in \text{paths (substitute } f' T t) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma substitute-cong-induct:}
\begin{align*}
\text{assumes } & \bigwedge x. x \in A \implies \text{carrier } (f x) \subseteq A \\
\text{assumes } & \text{carrier } t \subseteq A \\
\text{assumes } & \bigwedge x. x \in A \implies f x = f' x \\
\text{shows } & \text{substitute } f T t = \text{substitute } f' T t \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma carrier-substitute-aux:}
\begin{align*}
\text{assumes } & xs \in \text{paths (substitute } f T t) \\
\text{assumes } & \text{carrier } t \subseteq A \\
\text{assumes } & \bigwedge x. x \in A \implies \text{carrier } (f x) \subseteq A \\
\text{shows } & \text{set } xs \subseteq A \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma carrier-substitute-below:}
\begin{align*}
\text{assumes } & \bigwedge x. x \in A \implies \text{carrier } (f x) \subseteq A \\
\text{assumes } & \text{carrier } t \subseteq A \\
\text{shows } & \text{carrier } (\text{substitute } f T t) \subseteq A \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma f-nxt-eq-empty-iff:}
\begin{align*}
\text{f-nxt } f T x x' = \text{empty } & \iff \text{empty } \lor (x' = x \land x \in T) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma substitute-T-cong':}
\begin{align*}
\text{assumes } & xs \in \text{paths (substitute } f T t) \\
\text{assumes } & \bigwedge x. (x \in T \iff x \in T') \lor f x = \text{empty} \\
\text{shows } & xs \in \text{paths (substitute } f T' t) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma substitute-cong-T:}
\begin{align*}
\text{assumes } & \bigwedge x. (x \in T \iff x \in T') \lor f x = \text{empty} \\
\text{shows } & \text{substitute } f T = \text{substitute } f T' \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma carrier-substitute1:}
\begin{align*}
\text{carrier } t & \subseteq \text{carrier } (\text{substitute } f T t) \\
\langle \text{proof} \rangle
\end{align*}

\textbf{lemma substitute-cong:}
\begin{align*}
\text{assumes } & \bigwedge x. x \in \text{carrier } (\text{substitute } f T t) \implies f x = f' x \\
\text{shows } & \text{substitute } f T t = \text{substitute } f' T t \\
\langle \text{proof} \rangle
\end{align*}
lemma substitute-substitute:
  assumes \( \forall x. \text{ carrier } (f x) \cap S = \emptyset \)
  shows \( \text{ substitute } F T (\text{ substitute } f T t) = \text{ substitute } (\lambda x. f x \otimes f' x) T t \)
  (proof)

lemma three-rest-substitute:
  assumes \( \forall x. \text{ carrier } (f x) \cap S = \emptyset \)
  shows \( \text{ three-restr } S (\text{ substitute } f T t) = \text{ three-restr } S t \)
  (proof)

An alternative characterization of substitution

inductive substitute'' :: \( \langle a \Rightarrow \text{ a tree } \rangle \Rightarrow \langle a \Rightarrow \text{ list } \rangle \Rightarrow \langle a \Rightarrow \text{ list } \Rightarrow \text{ bool } \rangle \)
  | substitute''-Nil: substitute'' f T [] []
  | substitute''-Cons:
    \( z s \in \text{ paths } (f x) \quad \Rightarrow \quad x s' \in \text{ interleave } x s z s \quad \Rightarrow \quad \text{ substitute'' } (f \text{-nxt } f T x) T x s' y s \)
    \( \Rightarrow \quad \text{ substitute'' } f T (x \# x s) \quad (x \# y s) \)

inductive-cases substitute''-Nil[\(\downarrow\)elim]: substitute'' f T x s [] substitute'' f T [] x s
inductive-cases substitute''-Cons[\(\downarrow\)elim]: substitute'' f T (x \# x s) y s

lemma substitute-substitute'':
  \( x s \in \text{ paths } (\text{ substitute } f T t) \quad \leftrightarrow \quad (\exists x s' \in \text{ paths } t. \text{ substitute'' } f T x s' x s) \)
  (proof)

lemma paths-substitute-substitute'':
  \( \text{ paths } (\text{ substitute } f T t) = \bigcup ((\lambda x s . \text{ Collect } (\text{ substitute'' } f T x s)) \quad \langle \text{ paths } t \rangle) \)
  (proof)

lemma three-rest-substitute2:
  assumes \( \forall x. \text{ carrier } (f x) \subseteq S \)
  assumes \( \text{ const-on } f (-S) \text{ empty} \)
  shows \( \text{ three-restr } S (\text{ substitute } f T t) \quad \Rightarrow \quad \text{ substitute } f T (\text{ three-restr } S t) \)
  (proof)

end

9.2 TTree-HOLCF

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

instantiation three :: (type) below
begin

lift-definition below-three :: \( \langle a \text{ tree } \Rightarrow \langle a \text{ tree } \Rightarrow \text{ bool } \rangle \quad (\subseteq) \rangle \)
instance(\(\downarrow\)proof)

end
\textbf{lemma} paths-mono: \( t \subseteq t' \implies \text{paths } t \subseteq \text{paths } t' \)
\begin{proof}
\end{proof}

\textbf{lemma} paths-mono-if: \( \text{paths } t \subseteq \text{paths } t' \iff t \subseteq t' \)
\begin{proof}
\end{proof}

\textbf{lemma} three-below1: \((\forall \text{xs. } \text{xs} \in \text{paths } t \implies \text{xs} \in \text{paths } t') \implies t \subseteq t' \)
\begin{proof}
\end{proof}

\textbf{lemma} paths-below1: \((\forall \text{x. } \text{x} \# \text{xs} \in \text{paths } t \implies \text{x} \# \text{xs} \in \text{paths } t') \implies t \subseteq t' \)
\begin{proof}
\end{proof}

\textbf{instance} three :: (type) po
\begin{proof}
\end{proof}

\textbf{lemma} is-lub-three:
\( S \ll | \text{Either } S \)
\begin{proof}
\end{proof}

\textbf{lemma} lub-is-either: \( \text{lub } S = \text{Either } S \)
\begin{proof}
\end{proof}

\textbf{instance} three :: (type) cpo
\begin{proof}
\end{proof}

\textbf{lemma} minimal-three[simp, intro!]: \( \text{empty} \subseteq S \)
\begin{proof}
\end{proof}

\textbf{instance} three :: (type) pcpo
\begin{proof}
\end{proof}

\textbf{lemma} empty-is-bottom: \( \text{empty} = \bot \)
\begin{proof}
\end{proof}

\textbf{lemma} carrier-bottom[simp]: \( \text{carrier } \bot = \{\} \)
\begin{proof}
\end{proof}

\textbf{lemma} below-anything[simp]:
\( t \subseteq \text{anything} \)
\begin{proof}
\end{proof}

\textbf{lemma} carrier-mono: \( t \subseteq t' \implies \text{carrier } t \subseteq \text{carrier } t' \)
\begin{proof}
\end{proof}

\textbf{lemma} nxt-mono: \( t \subseteq t' \implies \text{nxt } t x \subseteq \text{nxt } t' x \)
\begin{proof}
\end{proof}

\textbf{lemma} either-above-arg1: \( t \subseteq t \oplus t' \)
\begin{proof}
\end{proof}
\begin{verbatim}
lemma either-above-arg2: $t' \sqsubseteq t \oplus t'$
  (proof)

lemma either-below1: $t \sqsubseteq t'' \implies t' \sqsubseteq t'' \implies t \oplus t' \sqsubseteq t''$
  (proof)

lemma both-above-arg1: $t \sqsubseteq t \otimes t'$
  (proof)

lemma both-above-arg2: $t' \sqsubseteq t \otimes t'$
  (proof)

lemma both-mono1':
  $t \sqsubseteq t' \implies t' \otimes t'' \sqsubseteq t \otimes t''$
  (proof)

lemma both-mono2':
  $t \sqsubseteq t' \implies t'' \otimes t \sqsubseteq t'' \otimes t'$
  (proof)

lemma nxt-both-left:
  possible $t \implies t x \otimes t x' \sqsubseteq \text{nxt}(t \otimes t') x$
  (proof)

lemma nxt-both-right:
  possible $t' \implies t \otimes \text{nxt} t' x \sqsubseteq \text{nxt}(t \otimes t') x$
  (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 three-cont1:
  assumes $\forall S. f (\text{Either } S) = \text{Either } (f \cdot S)$
  shows $\text{cont } f$
  (proof)

lemma three-cont12:
  assumes $\forall x. \text{paths } (f x) = \bigcup(t \cdot \text{paths } x)$
  assumes $[] \in t []$
  shows $\text{cont } f$
\end{verbatim}
lemma cont-paths[THEN cont-compose, cont2cont, simp]:
  cont paths
  ⟨proof⟩

lemma three-contI3:
  assumes cont (λ x. paths (f x))
  shows cont f
  ⟨proof⟩

lemma cont-substitute[THEN cont-compose, cont2cont, simp]:
  cont (substitute f T)
  ⟨proof⟩

lemma cont-both1:
  cont (λ x. both x y)
  ⟨proof⟩

lemma cont-both2:
  cont (λ x. both y x)
  ⟨proof⟩

lemma cont-both[cont2cont,simp]: cont f ⇒ cont g ⇒ cont (λ x. f x ⊗ g x)
  ⟨proof⟩

lemma cont-intersect11:
  cont (λ x. intersect x y)
  ⟨proof⟩

lemma cont-intersect2:
  cont (λ x. intersect y x)
  ⟨proof⟩

lemma cont-intersect[cont2cont,simp]: cont f ⇒ cont g ⇒ cont (λ x. f x ∩ g x)
  ⟨proof⟩

lemma cont-without[THEN cont-compose, cont2cont,simp]: cont (without x)
  ⟨proof⟩

lemma paths-many-calls-subset:
  t ⊆ many-calls x ⊗ without x t
  ⟨proof⟩

lemma single-below:
  [x] ∈ paths t ⇒ single x ⊆ t ⟨proof⟩
lemma cont-three-restr [THEN cont-compose, cont2cont, simp]: cont (three-restr S)
⟨proof⟩

lemmas three-restr-mono = cont2monofunE [OF cont-three-restr [OF cont-id]]

lemma mnge-filter [simp]: mnge (filter P) = {xs. set xs ⊆ Collect P}
⟨proof⟩

lemma three-restr-anything-cont [THEN cont-compose, simp, cont2cont]:
cont (λ S. three-restr S anything)
⟨proof⟩

instance three :: (type) Finite-Join-epo
⟨proof⟩

lemma three-join-is-either:
    t ⊔ t' = t ⊕ t'
⟨proof⟩

lemma three-join-transfer [transfer-rule]: rel-fun (per-three (=)) (rel-fun (per-three (=))) (per-three (=))(⊔)(⊓)
⟨proof⟩

lemma three-restr-join [simp]:
    three-restr S (t ⊔ t') = three-restr S t ⊔ three-restr S t'
⟨proof⟩

lemma nxt-singles-below-singles:
    nxt (singles S) x ⊑ singles S
⟨proof⟩

lemma in-carrier-fup [simp]:
    x' ∈ carrier (fup · f · u) ↔ (∃ u'. u = up · u' ∧ x' ∈ carrier (f · u'))
⟨proof⟩

end

10 Trace Tree Cardinality Analysis

10.1 AnalBinds

theory AnalBinds
imports Launchbury.Terms Launchbury.HOLCF –Utils Launchbury.Env
begin

locale ExpAnalysis =
fixes \( \text{exp} :: \text{exp} \Rightarrow \text{'a::cpo} \rightarrow \text{'b::cpo} \)

begin

fun AnalBinds :: heap \Rightarrow (\text{var} \Rightarrow \text{'a}_\bot) \rightarrow (\text{var} \Rightarrow \text{'b})

where AnalBinds \[\] = (\Lambda \text{ae}. \bot) 

| AnalBinds (\langle x, e \rangle \# \Gamma) = (\Lambda \text{ae}. (AnalBinds \Gamma \cdot \text{ae})(x := \text{fup} \cdot (\text{exp} e) \cdot (\text{ae} x)))

lemma AnalBinds-Nil-simp[simp]: AnalBinds \[\] \cdot \text{ae} = \bot \langle \text{proof} \rangle

lemma AnalBinds-Cons[simp]:

AnalBinds (\langle x, e \rangle \# \Gamma) \cdot \text{ae} = (AnalBinds \Gamma \cdot \text{ae})(x := \text{fup} \cdot (\text{exp} e) \cdot (\text{ae} x))

\langle \text{proof} \rangle

lemmas AnalBinds.simps[simp del]

lemma AnalBinds-not-there: \text{x} \notin \text{dom} A \Gamma \Longrightarrow (\text{AnalBinds} \Gamma \cdot \text{ae}) x = \bot

\langle \text{proof} \rangle

lemma AnalBinds-cong:

assumes \text{ae} f \mid\mid \text{dom} A \Gamma = \text{ae}' f \mid\mid \text{dom} A \Gamma

shows AnalBinds \Gamma \cdot \text{ae} = AnalBinds \Gamma \cdot \text{ae}'

\langle \text{proof} \rangle

lemma AnalBinds-lookup: (AnalBinds \Gamma \cdot \text{ae}) x = (\text{case map-of} \Gamma \text{ of} \text{Some} \ e \Rightarrow \text{fup} \cdot (\text{exp} e) \cdot (\text{ae} x) \mid \text{None} \Rightarrow \bot)

\langle \text{proof} \rangle

lemma AnalBinds-delete-bot: \text{ae} x = \bot \Longrightarrow AnalBinds (\text{delete} x \Gamma) \cdot \text{ae} = AnalBinds \Gamma \cdot \text{ae}

\langle \text{proof} \rangle

lemma AnalBinds-delete-below: AnalBinds (\text{delete} x \Gamma) \cdot \text{ae} \subseteq AnalBinds \Gamma \cdot \text{ae}

\langle \text{proof} \rangle

lemma AnalBinds-delete-lookup[simp]: (AnalBinds (\text{delete} x \Gamma) \cdot \text{ae}) x = \bot

\langle \text{proof} \rangle

lemma AnalBinds-delete-to-fun-upd: AnalBinds (\text{delete} x \Gamma) \cdot \text{ae} = (AnalBinds \Gamma \cdot \text{ae})(x := \bot)

\langle \text{proof} \rangle

lemma edom-AnalBinds: edom (AnalBinds \Gamma \cdot \text{ae}) \subseteq \text{dom} A \Gamma \cap \text{edom} \text{ae}

\langle \text{proof} \rangle

end

end

10.2 TTreeAnalysisSig

theory TTreeAnalysisSig
imports Arity TTree HOLCF AnalBinds
begin

locale TTreeAnalysis = 
fixes Texp :: exp ⇒ Arity ⇒ 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 [] ←→ True
| no-call-in-path x (y # xs) ←→ y ≠ x ∧ no-call-in-path x xs

fun one-call-in-path where
  one-call-in-path x [] ←→ True
| one-call-in-path x (y # xs) ←→ (if x = y then no-call-in-path x xs else one-call-in-path x xs)

lemma no-call-in-path-set-conv:
  no-call-in-path x p ←→ x /∈ set p
⟨proof⟩

lemma one-call-in-path-filter-conv:
  one-call-in-path x p ←→ length (filter (λ x'. x' = x) p) ≤ 1
⟨proof⟩

⟨proof⟩

lemma no-imp-one: no-call-in-path x p =⇒ one-call-in-path x p
⟨proof⟩

lemma one-imp-one-tail: one-call-in-path x p =⇒ one-call-in-path x (tl p)
⟨proof⟩

lemma more-than-one-setD:
  ¬ one-call-in-path x p =⇒ x ∈ set p
⟨proof⟩


lemma no-call-in-path[eqvt]: no-call-in-path p x \rightleftharpoons no-call-in-path (\pi \cdot p) (\pi \cdot x)
(\langle proof \rangle)

lemma one-call-in-path[eqvt]: one-call-in-path p x \rightleftharpoons one-call-in-path (\pi \cdot p) (\pi \cdot x)
(\langle proof \rangle)

definition \text{pathCard} :: \text{var list} \Rightarrow (\text{var} \Rightarrow \text{two})
where \text{pathCard} p x = (if no-call-in-path x p then none else (if one-call-in-path x p then once else many))

lemma \text{pathCard-Nil}[simp]: \text{pathCard} [] = \bot
(\langle proof \rangle)

lemma \text{pathCard-Cons}[simp]: \text{pathCard} (x \# xs) x = \text{two-add-one}(\text{pathCard} xs x)
(\langle proof \rangle)

lemma \text{pathCard-Cons-other}[simp]: x' \neq x \Longrightarrow \text{pathCard} (x \# xs) x' = \text{pathCard} xs x'
(\langle proof \rangle)

lemma no-call-in-path-filter[simp]: no-call-in-path x [x \leftarrow xs . x \in S] \leftarrow no-call-in-path xs \lor x \notin S
(\langle proof \rangle)

lemma one-call-in-path-filter[simp]: one-call-in-path x [x \leftarrow xs . x \in S] \leftarrow one-call-in-path x xs \lor x \notin S
(\langle proof \rangle)

definition \text{pathsCard} :: \text{var list set} \Rightarrow (\text{var} \Rightarrow \text{two})
where \text{pathsCard} ps x = (if (\forall p \in ps. no-call-in-path x p) then none else (if (\forall p \in ps. one-call-in-path x p) then once else many))

lemma \text{paths-Car d-ab ove}:
p \in ps \Longrightarrow \text{pathCard} p \subseteq \text{pathsCard} ps
(\langle proof \rangle)

lemma \text{pathsCard-b elow}:
assumes \land p, p \in ps \Longrightarrow \text{pathCard} p \subseteq \alpha
shows \text{pathsCard} ps \subseteq \alpha
(\langle proof \rangle)

lemma \text{pathsCard-mono}:
ps \subseteq ps' \Longrightarrow \text{pathsCard} ps \subseteq \text{pathsCard} ps'
(\langle proof \rangle)

lemmas \text{pathsCard-mono'} = \text{pathsCard-mono[folded below-set-def]}

lemma \text{record-call-pathsCard}:
\text{pathsCard} (( \{ tl p \mid p \in fs \land hd p = x \}) \sqsubseteq \text{record-call x.(pathsCard fs)}
(\langle proof \rangle)
lemma pathCards-noneD:
    pathsCard ps x = none \(\Rightarrow\) \(x \notin \bigcup\{\text{set } ps\}\)
    (proof)

lemma cont-pathsCard[THEN cont-compose, cont2cont, simp]:
    cont pathsCard
    (proof)

lemma pathsCard-eqv[eqvt]: \(\pi \cdot \text{pathsCard } ps x = \text{pathsCard } (\pi \cdot ps) (\pi \cdot x)\)
    (proof)

lemma edom-pathsCard[simp]: edom (pathsCard ps) = \(\bigcup\{\text{set } ps\}\)
    (proof)

lemma env-restr-pathsCard[simp]: pathsCard ps f \mid S = pathsCard (filter (\(\lambda x. x \in S\)) ps)
    (proof)

end

10.4 TTreeAnalysisSpec

theory TTreeAnalysisSpec
imports TTreeAnalysisSig ArityAnalysisSpec Cardinality−Domain−Lists
begin

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

locale TTreeAnalysisSafe = TTreeAnalysisCarrier +
    assumes Texp-App: many-calls x \(\otimes\otimes\) (Texp e-(inc-a)) \(\subseteq\) Texp (App e x)\(\cdot\)a
    assumes Texp-Lam: without y (Texp e-(pred\cdot n)) \(\subseteq\) Texp (Lam [y]. e) \(\cdot\) n
    assumes Texp-subst: Texp (e[y::=x])\(\cdot\)a \(\subseteq\) many-calls x \(\otimes\otimes\) without y (Texp e)\(\cdot\)a
    assumes Texp-Var: single v \(\subseteq\) Texp (Var v)\(\cdot\)a
    assumes Fun-repeatable: isVal e \(\Rightarrow\) repeatable (Texp e-0)
    assumes Texp-IfThenElse: Texp scrut\(\otimes\otimes\) (Texp e1-a \(\otimes\otimes\) Texp e2-a) \(\subseteq\) Texp (scrut ? e1 : e2)\(\cdot\)a

locale TTreeAnalysisCardinalityHeap =
    TTreeAnalysisSafe + ArityAnalysisLetSafe +
    fixes Texp :: heap \(\Rightarrow\) exp \(\Rightarrow\) Arity \(\Rightarrow\) var three
    assumes carrier-Heap: carrier (Texp \Gamma e-a) = edom (Aheap \Gamma e-a)
    assumes Texp-thunk: x \(\in\) thanks \(\Gamma\) \(\Longrightarrow\) p \(\in\) paths (Texp \Gamma e-a) \(\Longrightarrow\) \(\neg\) one-call-in-path x p
        \(\Longrightarrow\) (Aheap \Gamma e-a) x = up\(\cdot\)0
    assumes Texp-substitute: tree-restr (domA \Delta) (substitute (FBinds \Delta\cdot(Aheap \Delta e-a))) (thanks \Delta) (Texp e-a) \(\subseteq\) Texp \Delta e-a
    assumes Texp-Let: tree-restr (\(\neg\) domA \Delta) (substitute (FBinds \Delta\cdot(Aheap \Delta e-a))) (thanks \Delta) (Texp e-a) \(\subseteq\) Texp (Terms.Let \Delta e)\(\cdot\)a

76
10.5 TTreeImplCardinality

theory TTreeImplCardinality
imports TTreeAnalysisSig CardinalityAnalysisSig Cardinality Domain Lists
begin

context TTreeAnalysis
begin

fun unstack :: stack ⇒ exp ⇒ exp where
  unstack [] e = e
| unstack (Alts e1 e2 # S) e = unstack S e
| unstack (Upd x # S) e = unstack S e
| unstack (Arg x # S) e = unstack S (App e x)
| unstack (Dummy x # S) e = unstack S e

fun Fstack :: Arity list ⇒ stack ⇒ var three
  where Fstack - [] = ⊥
  | Fstack (a#as) (Alts e1 e2 # S) = (Texp e1-a ⊕ Texp e2-a) ⊗ Fstack as S
  | Fstack as (Arg x # S) = many-calls x ⊗ Fstack as S
  | Fstack as (- # S) = Fstack as S

fun prognosis :: AEnv ⇒ Arity list ⇒ Arity ⇒ conf ⇒ var ⇒ two
  where prognosis ae as a (Γ, e, S) = pathsCard (paths (substitute (FBinds Γ ae) (thunks Γ) (Texp e-a) ⊗ Fstack as S)))
end
end

10.6 TTreeImplCardinalitySafe

theory TTreeImplCardinalitySafe
imports TTreeImplCardinality TTreeAnalysisSpec CardinalityAnalysisSpec
begin

lemma pathsCard-paths-nxt: pathsCard (paths (nxt f x)) ⊆ record-call x.(pathsCard (paths f))
(proof)

lemma pathsCards-none: pathsCard (paths t) x = none ⇒ x /∈ carrier t
(proof)

**lemma** const-on-dom-disj: const-on f S empty \(\iff\) dom f \(\cap\) S = \{
(proof)

**context** TTreeAnalysisCarrier
begin

*lemma* carrier-Fstack: carrier (Fstack as S) \(\subseteq\) fv S
(proof)

*lemma* carrier-FBinds: carrier ((FBinds \(\Gamma\cdot ae\) x) \(\subseteq\) 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
cHeap \(\Gamma\) e = (\(\Lambda\) a. pathsCard (paths (Heap \(\Gamma\) e·a)))

*lemma* cHeap-simp: (cHeap \(\Gamma\) e·a) = pathsCard (paths (Heap \(\Gamma\) e·a))
(proof)

sublocale CardinalityHeap cHeap (proof)

sublocale CardinalityHeapSafe cHeap Aheap
(proof)

sublocale CardinalityPrognosisEdom prognosis

78
subsection "CardinalityPrognosisLet prognosis eHeap" (proof)
subsection "CardinalityPrognosisSafe prognosis eHeap Aheap Aexp" (proof)
end
end

11 Co-Call Graphs

11.1 CoCallGraph

theory CoCallGraph
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 per-CoCalls =⇒⇒ per-CoCalls) Union lub (proof)

lift-definition is-cclub :: CoCalls set ⇒ CoCalls ⇒ bool is (λ S x . x = Union S)(proof)

79
lemma ccis-lubTransfer[transfer-rule]: (rel-set per-CoCalls ===> per-CoCalls ===> (=)) (λ S x . x = Union S) (<<)
(proof)

lift-definition coCallsJoin :: CoCalls ⇒ CoCalls ⇒ CoCalls is (\union)
(proof)

lemma ccJoinTransfer[transfer-rule]: (per-CoCalls ===> per-CoCalls ===> per-CoCalls)
(\union) (\subseteq)
(proof)

lift-definition ccEmpty :: CoCalls is { }
(proof)

lemma ccEmpty-below[simp]: ccEmpty \subseteq G
(proof)

instance CoCalls :: pcpo
(proof)

lemma ccBotTransfer[transfer-rule]: per-CoCalls { } ⊥
(proof)

lemma cc-lub-below-iff:
fixes G :: CoCalls
shows lub X \subseteq G ⇔ (∀ G'∈X. G' \subseteq G)
(proof)

lift-definition ccField :: CoCalls ⇒ var set is Field
(proof)

lemma ccField-nil[simp]: ccField ⊥ = { }
(proof)

lift-definition inCC :: var ⇒ var ⇒ CoCalls ⇒ bool (\neg \neg \in \ [1000 , 1000 , 900] 900)
is λ x y s . (x,y) ∈ s
(proof)

abbreviation notInCC :: var ⇒ var ⇒ CoCalls ⇒ bool (\neg \neg \notin \ [1000 , 1000 , 900] 900)
where x\neg\neg\notin S ≡ \neg \neg x\neg\notin S

lemma notInCC-bot[simp]: x\neg\notin\subseteq ⊥ \iff False
(proof)

lemma below-CoCallsI:
(\land x y . x\neg\notin G \implies x\neg\notin G') \implies G \subseteq G'
(proof)

lemma CoCalls-\neg1:
(\land x y . x\neg\notin G \iff x\neg\notin G') \implies G = G'

80
lemma in-join [simp]:
\[ x \rightarrow y \in (G \sqcup G') \iff x \rightarrow y \in G \lor x \rightarrow y \in G' \]

lemma in-lub [simp]:
\[ x \rightarrow y \in \text{lub } S \iff \exists G \in S. x \rightarrow y \in G \]

lemma in-CoCall lsLubI:
\[ x \rightarrow y \in G \Rightarrow G \in S \Rightarrow x \rightarrow y \in \text{lub } S \]

lemma adm-not-in [simp]:
\[ \text{assumes } \text{cont } t \rightarrow \text{shows } \text{adm } (\lambda a. x \rightarrow y \not\in t a) \]

lift-definition cc-delete :: var \Rightarrow CoCalls \Rightarrow CoCalls
\[ \text{is } \lambda \ z. \text{Set.filter } (\lambda (x,y) . x \neq z \land y \neq z) \]

lemma ccField-cc-delete: \(\alpha\text{Field } (\text{cc-delete } x S) \subseteq \text{ccField } S \setminus \{x\} \]

lift-definition ccProd :: \text{var set} \Rightarrow \text{var set} \Rightarrow CoCalls (infix \( G \times 90 \))
\[ \text{is } \lambda S1 S2. S1 \times S2 \cup S2 \times S1 \]

lemma ccProd-empty [simp]: \(\{ \}\times S = \bot \)

lemma ccProd-empty' [simp]: \(S \times \{ \}\ = \bot \)

lemma ccProd-union2 [simp]: \(S \times (S' \cup S'') = S \times S' \cup S \times S'' \)

lemma ccProd-Union2 [simp]: \(S \times \bigcup S' = (\bigcup X \in S'. \text{ccProd } S X) \)

lemma ccProd-Union2' [simp]: \(S \times (\bigcup X \in S'. f X) = (\bigcup X \in S'. \text{ccProd } S (f X)) \)

lemma in-ccProd [simp]: \(x \rightarrow y \in (S \times S') = (x \in S \land y \in S' \lor x \in S' \land y \in S) \)

lemma ccProd-union1 [simp]: \((S' \cup S'') \times S = S' \times S \cup S'' \times S \)

lemma ccProd-insert2: \(S \times \text{insert } x S' = S \times \{x\} \cup S \times S' \)
lemma ccProd-insert1: insert x S' G × S = {x} G × S ∪ S' G × S
(\langle proof \rangle)

lemma ccProd-mono1: S' ⊆ S'' \implies S' G × S ⊆ S'' G × S
(\langle proof \rangle)

lemma ccProd-mono2: S' ⊆ S'' \implies S' G × S ⊑ S'' G × S
(\langle proof \rangle)

lemma ccProd-mono: S ⊆ S' =⇒ T ⊆ T' =⇒ S G × T ⊑ S' G × T'
(\langle proof \rangle)

lemma ccProd-comm: S G × S' = S' G × S
(\langle proof \rangle)

lemma ccProd-below1:
(∀ x y. x ∈ S =⇒ y ∈ S' =⇒ x−−y ∈ G) =⇒ S G × S' ⊑ G
(\langle proof \rangle)

lift-definition cc-restr :: var set ⇒ CoCalls ⇒ CoCalls
is λ S. Set.filter (λ (x,y) . x ∈ S ∧ y ∈ S)
(\langle proof \rangle)

abbreviation cc-restr-sym (infixl G|· 110) where G G|· S = cc-restr S G
(\langle proof \rangle)

lemma elem-cc-restr[simp]: x−−y ∈ (G G|· S) = (x−−y ∈ G ∧ x ∈ S ∧ y ∈ S)
(\langle proof \rangle)

lemma ccField-cc-restr: ccField (G G|· S) ⊆ ccField G ∩ S
(\langle proof \rangle)

lemma cc-restr-empty: ccField G ⊆ − S =⇒ G G|· S = ⊥
(\langle proof \rangle)

lemma cc-restr-empty-set[simp]: cc-restr { } G = ⊥
(\langle proof \rangle)

lemma cc-restr-noop[simp]: ccField G ⊆ S =⇒ cc-restr S G = G
(\langle proof \rangle)

lemma cc-restr-bot[simp]: cc-restr S ⊥ = ⊥
(\langle proof \rangle)

lemma ccRestr-ccDelete[simp]: cc-restr (−{x}) G = cc-delete x G
(\langle proof \rangle)

lemma cc-restr-join[simp]:

82
\[\text{cc-restr } S \ (G \uplus G') = \text{cc-restr } S \ (G \uplus \text{cc-restr } S \ G')\]

**lemma** *cont-cc-restr*: *cont* (cc-restr *S*)

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

**lemma** *cc-restr-mono1*:
\[S \subseteq S' \implies \text{cc-restr } S \ G \subseteq \text{cc-restr } S' \ G\] (proof)

**lemma** *cc-restr-mono2*:
\[G \subseteq G' \implies \text{cc-restr } S \ G \subseteq \text{cc-restr } S \ G'\] (proof)

**lemma** *cc-restr-below-arg*:
\[\text{cc-restr } S \ G \subseteq G\] (proof)

**lemma** *cc-restr-hub*[simp]:
\[\text{cc-restr } S \ (\bigcup X) = \bigcup \{G \in X. \ \text{cc-restr } S \ G\}\] (proof)

**lemma** *elem-to-ccField*:
\[x \cdashdot y \in G \implies x \in \text{ccField } G \land y \in \text{ccField } G\]

**lemma** *ccField-to-elem*:
\[x \in \text{ccField } G \implies \exists \ y. \ x \cdashdot y \in G\]

**lemma** *cc-restr-intersect*:
\[\text{ccField } G \cap ((S - S') \cup (S' - S)) = \{\} \implies \text{cc-restr } S \ G = \text{cc-restr } S' \ G\]

**lemma** *cc-restr-cc-restr*[simp]:
\[\text{cc-restr } S \ (\text{cc-restr } S' \ G) = \text{cc-restr } (S \cap S') \ G\]

**lemma** *cc-restr-twist*:
\[\text{cc-restr } S \ (\text{cc-restr } S' \ G) = \text{cc-restr } S' \ (\text{cc-restr } S \ G)\]

**lemma** *cc-restr-cc-delete-twist*:
\[\text{cc-restr } x \ (\text{cc-delete } S \ G) = \text{cc-delete } S \ (\text{cc-restr } x \ G)\]

**lemma** *cc-restr-ccProd*[simp]:
\[\text{cc-restr } S \ (\text{ccProd } S_1 \ S_2) = \text{ccProd } (S_1 \cap S) \ (S_2 \cap S)\]

**lemma** *ccProd-below-cc-restr*:
\[\text{ccProd } S \ S' \subseteq \text{cc-restr } S'' \ G \iff \text{ccProd } S \ S' \subseteq G \land (S = \{\} \lor S' = \{\} \lor S \subseteq S'' \land S' \subseteq S'')\]

**lemma** *cc-restr-eq-subset*:
\[S \subseteq S' \implies \text{cc-restr } S' \ G = \text{cc-restr } S' \ G_2 \implies \text{cc-restr } S \ G = \text{cc-restr} \]

83
\textbf{definition} \texttt{ccSquare} \ ((^2 \{80\} \ 80) \\
\texttt{where} \ S^2 = \texttt{ccProd} \ S \ S \\

\textbf{lemma} \texttt{ccField-ccSquare[simp]}: \texttt{ccField} \ (S^2) = S \\
\langle \textit{proof} \rangle \\

\textbf{lemma} \texttt{below-ccSquare[iff]}: \ (G \subseteq S^2) = (\texttt{ccField} \ G \subseteq S) \\
\langle \textit{proof} \rangle \\

\textbf{lemma} \texttt{cc-restr-ccSquare[simp]}: \ (S^2) \ G \cdot S = (S' \cap S)^2 \\
\langle \textit{proof} \rangle \\

\textbf{lemma} \texttt{ccSquare-empty[simp]}: \ \{\}^2 = \bot \\
\langle \textit{proof} \rangle \\

\textbf{lift-definition} \ \texttt{ccNeighbors} :: \ \texttt{var} \Rightarrow \ \texttt{CoCalls} \Rightarrow \ \texttt{var set} \\
\texttt{is} \ \lambda \ x \ G. \ \{y . (y,x) \in G \lor (x,y) \in G\} \langle \text{proof} \rangle \\

\textbf{lemma} \texttt{ccNeighbors-bot[simp]}: \ \texttt{ccNeighbors} \ x \ \bot = \ \{\} \ \langle \textit{proof} \rangle \\

\textbf{lemma} \texttt{cont-ccProd1}:
\begin{align*}
\texttt{cont} \ (\lambda S. \ \texttt{ccProd} \ S \ S')
\end{align*} \\
\langle \textit{proof} \rangle \\

\textbf{lemma} \texttt{cont-ccProd2}:
\begin{align*}
\texttt{cont} \ (\lambda S'. \ \texttt{ccProd} \ S \ S')
\end{align*} \\
\langle \textit{proof} \rangle \\

\textbf{lemmas} \texttt{cont-compose2[OF \ cont-ccProd1 \ cont-ccProd2, \ simp, \ cont2cont]} \\

\textbf{lemma} \texttt{cont-ccNeighbors[THEN \ cont-compose, \ cont2cont, \ simp]}:
\begin{align*}
\texttt{cont} \ (\lambda y. \ \texttt{ccNeighbors} \ x \ y)
\end{align*} \\
\langle \textit{proof} \rangle \\

\textbf{lemma} \texttt{ccNeighbors-join[simp]}: \ \texttt{ccNeighbors} \ x \ (G \cup G') = \texttt{ccNeighbors} \ x \ G \cup \texttt{ccNeighbors} \ x \ G' \\
\langle \textit{proof} \rangle \\

\textbf{lemma} \texttt{ccNeighbors-ccProd}:
\begin{align*}
\texttt{ccNeighbors} \ x \ (\texttt{ccProd} \ S \ S') = (\texttt{if} \ x \in S \ \texttt{then} \ S' \ \texttt{else} \ \{\}) \cup (\texttt{if} \ x \in S' \ \texttt{then} \ S \ \texttt{else} \ \{\})
\end{align*} \\
\langle \textit{proof} \rangle \\

\textbf{lemma} \texttt{ccNeighbors-ccSquare}:
\begin{align*}
\texttt{ccNeighbors} \ x \ (\texttt{ccSquare} \ S) = (\texttt{if} \ x \in S \ \texttt{then} \ S \ \texttt{else} \ \{\})
\end{align*} \\
\langle \textit{proof} \rangle \\

84
lemma \texttt{ccNeighbors-cc-restr[simp]}:
\[
\text{ccNeighbors } x \ (\text{cc-restr } S \ G) = (\text{if } x \in S \text{ then ccNeighbors } x \cap S \text{ else } \{\})
\]
(proof)

lemma \texttt{ccNeighbors-mono}:
\[
G \subseteq G' \implies \text{ccNeighbors } x \ G \subseteq \text{ccNeighbors } x \ G'
\]
(proof)

lemma \texttt{subset-ccNeighbors}:
\[
S \subseteq \text{ccNeighbors } x \ G \iff \text{Prod } \{x\} \ S \subseteq G
\]
(proof)

lemma \texttt{elem-ccNeighbors[simp]}:
\[
y \in \text{ccNeighbors } x \ G \iff (y \leftarrow x \in G)
\]
(proof)

lemma \texttt{ccNeighbors-ccField}:
\[
\text{ccNeighbors } x \ G \subseteq \text{ccField } G
\]
(proof)

lemma \texttt{ccNeighbors-disjoint-empty[simp]}:
\[
\text{ccNeighbors } x \ G = \{\} \iff x \notin \text{ccField } G
\]
(proof)

instance \texttt{CoCalls :: Join-cpo}
(proof)

lemma \texttt{ccNeighbors-hub[simp]}:
\[
\text{ccNeighbors } x \ (\text{hub } Gs) = \text{hub } (\text{ccNeighbors } x \cdot Gs)
\]
(proof)

inductive \texttt{list-pairs :: 'a list \Rightarrow ('a \times 'a) \Rightarrow bool}
where
\[
\text{list-pairs } xs \ p \implies \text{list-pairs } (x\#xs) \ p
\]
| \ y \in \text{set } xs \implies \text{list-pairs } (x\#xs) (x,y)

lift-definition \texttt{ccFromList :: var list \Rightarrow CoCalls \ is \ xs.} \ (x,y). list-pairs xs (x,y) \lor list-pairs xs (y,x)
(proof)

lemma \texttt{ccFromList-Nil[simp]}:
\[
\text{ccFromList } [] = \bot
\]
(proof)

lemma \texttt{ccFromList-Cons[simp]}:
\[
\text{ccFromList } (x\#xs) = \text{ccProd } \{x\} \ (\text{set } xs) \sqcup \text{ccFromList } xs
\]
(proof)

lemma \texttt{ccFromList-append[simp]}:
\[
\text{ccFromList } (xs@ys) = \text{ccFromList } xs \sqcup \text{ccFromList } ys \sqcup \text{ccProd } (\text{set } xs) \ (\text{set } ys)
\]
(proof)

lemma \texttt{ccFromList-filler[simp]}:
\( \text{ccFromList (filter } P \text{ xs)} = \text{cc-restr} \{ x. \ P \ x \} (\text{ccFromList} \ \text{xs}) \) 

\textbf{lemma} \( \text{ccFromList-replicate[simp]}: \text{ccFromList} (\text{replicate} \ n \ x) = (\text{if } n \leq 1 \ \text{then } \bot \ \text{else} \ \text{ccProd} \ \{x\} \ \{x\}) \) 

\textbf{definition} \( \text{ccLinear :: var set } \Rightarrow \text{CoCalls} \Rightarrow \text{bool} \) 
where \( \text{ccLinear} \ S \ G = (\forall \ x \in S. \ \forall \ y \in S. \ x \neq y \notin G) \) 

\textbf{lemma} \( \text{ccLinear-bottom[simp]}: \) 
\( \text{ccLinear} \ S \bot \) 
\( \langle \text{proof} \rangle \) 

\textbf{lemma} \( \text{ccLinear-empty[simp]}: \) 
\( \text{ccLinear} \ \{\} \ G \) 
\( \langle \text{proof} \rangle \) 

\textbf{lemma} \( \text{ccLinear-lub[simp]}: \) 
\( \text{ccLinear} \ S \ (\text{lub} \ X) = (\forall \ G \in X. \ \text{ccLinear} \ S \ G) \) 
\( \langle \text{proof} \rangle \) 

\textbf{lemma} \( \text{ccLinear-cc-restr[intro]}: \) 
\( \text{ccLinear} \ S \ G \Rightarrow \text{ccLinear} \ S \ (\text{cc-restr} \ S' \ G) \) 
\( \langle \text{proof} \rangle \) 

\textbf{lemma} \( \text{ccLinear-ccProd[simp]}: \) 
\( \text{ccLinear} \ S \ (\text{ccProd} \ S_1 \ S_2) \iff S_1 \cap S = \{\} \ \vee \ S_2 \cap S = \{\} \) 
\( \langle \text{proof} \rangle \) 

\textbf{lemma} \( \text{ccLinear-mono1}: \text{ccLinear} \ S' \ G \Rightarrow S \subseteq S' \Rightarrow \text{ccLinear} \ S \ G \) 
\( \langle \text{proof} \rangle \) 

\textbf{lemma} \( \text{ccLinear-mono2}: \text{ccLinear} \ S \ G' \Rightarrow G \subseteq G' \Rightarrow \text{ccLinear} \ S \ G \) 
\( \langle \text{proof} \rangle \) 

\textbf{lemma} \( \text{ccField-join[simp]}: \) 
\( \text{ccField} \ (G \sqcup G') = \text{ccField} \ G \cup \text{ccField} \ G' \) 
\( \langle \text{proof} \rangle \) 

\textbf{lemma} \( \text{ccField-lub[simp]}: \) 
\( \text{ccField} \ (\text{lub} \ S) = \bigcup (\text{ccField} \ S') \) 
\( \langle \text{proof} \rangle \)
lemma $ccField$-$ccProd$:
$ccField (ccProd \, S \, S') = (if \, S = \{\} \, then \, \{\} \, else \, if \, S' = \{\} \, then \, \{\} \, else \, S \cup S')$
(proof)

lemma $ccField$-$ccProd$-subset:
$ccField (ccProd \, S \, S') \subseteq S \cup S'$
(proof)

lemma $cont$-$ccField$[THEN $cont$-compose, simp, $cont2cont$]:
$cont \, ccField$
(proof)

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
(proof)
instance
(proof)
end

instance CoCalls :: $cont$-pt
(proof)

lemmas lub-eqvt[OF exists-lub, simp, eqvt]

lemma $cc$-restr-perm:
fixes $G$ :: CoCalls
assumes supp $p 
\star S$ and [simp]: finite $S$
shows $cc$-restr $S \, (p \cdot G) = cc$-restr $S \, G$
(proof)

lemma inCC-eqvt[eqvt]: $\pi \cdot (x \cdot \neg y \in G) = (\pi \cdot x) \cdot \neg (\pi \cdot y) \in (\pi \cdot G)$
(proof)

lemma $cc$-restr-eqvt[eqvt]: $\pi \cdot cc$-restr $S \, G = cc$-restr $\, (\pi \cdot S) \, (\pi \cdot G)$
(proof)

lemma $ccProd$-eqvt[eqvt]: $\pi \cdot ccProd \, S \, S' = ccProd \, (\pi \cdot S) \, (\pi \cdot S')$
(proof)

lemma $ccSquare$-eqvt[eqvt]: $\pi \cdot ccSquare \, S = ccSquare \, (\pi \cdot S)$

87
\[\text{proof}\]

\textbf{lemma} \text{ccNeighbors-eqvt}[\text{eqvt}]: \(\pi \cdot \text{ccNeighbors} S G = \text{ccNeighbors} (\pi \cdot S) (\pi \cdot G)\)

\[\text{proof}\]

\[\text{end}\]

\section{12 Co-Call Cardinality Analysis}

\subsection{12.1 CoCallAnalysisSig}

\textbf{theory} \text{CoCallAnalysisSig}

\textbf{imports} \text{Launchbury.Terms Arity CoCallGraph}

\textbf{begin}

\textbf{locale} \text{CoCallAnalysis} =

\begin{itemize}
  \item \textbf{fixes} \text{ccExp} :: \text{exp} \Rightarrow \text{Arity} \Rightarrow \text{CoCalls}
\end{itemize}

\textbf{begin}

\textbf{abbreviation} \text{ccExp-syn} (G)

\begin{itemize}
  \item where \(G_a \equiv (\lambda e. \text{ccExp} e \cdot a)\)
\end{itemize}

\textbf{abbreviation} \text{ccExp-bot-syn} (G⊥)

\begin{itemize}
  \item where \(G_\perp a \equiv (\lambda e. \text{fup} \cdot (\text{ccExp} e) \cdot a)\)
\end{itemize}

\textbf{end}

\textbf{locale} \text{CoCallAnalysisHeap} =

\begin{itemize}
  \item \textbf{fixes} \text{ccHeap} :: \text{heap} \Rightarrow \text{exp} \Rightarrow \text{Arity} \Rightarrow \text{CoCalls}
\end{itemize}

\textbf{end}

\subsection{12.2 CoCallAnalysisBinds}

\textbf{theory} \text{CoCallAnalysisBinds}

\textbf{imports} \text{CoCallAnalysisSig AEnv AList−Utils−HOLCF Arity−Nominal CoCallGraph−Nominal}

\textbf{begin}

\textbf{context} \text{CoCallAnalysis}

\textbf{begin}

\textbf{definition} \text{ccBind} :: \text{var} \Rightarrow \text{exp} \Rightarrow ((\text{AEnv} \times \text{CoCalls}) \Rightarrow \text{CoCalls})

\begin{itemize}
  \item where \text{ccBind} v e = (\Lambda (ae,G). \text{if } (e-\neg v \notin G) \lor \neg \text{isVal e then cc-restr } (fv e) (\text{fup} \cdot (\text{ccExp} e) \cdot (ae v)) \text{ else ccSquare } (fv e))
\end{itemize}

\textbf{lemma} \text{ccBind-eq}:

\[\text{ccBind} v e (ae, G) = (\text{if } e-\neg v \notin G \lor \neg \text{isVal e then } G_{\perp ae v} e G \mid fv e \text{ else } (fv e)^2)\]

\[\text{proof}\]

\textbf{lemma} \text{ccBind-strict}[\text{simp}]: \text{ccBind} v e \cdot \perp = \perp
lemma ccField-ccBind: ccField (ccBind v e·(ae,G)) ⊆ fv e
(proof)

definition ccBinds :: heap ⇒ ((Env × CoCalls) → CoCalls)
  where ccBinds Γ = (Λ i. (∫ v→e∈map-of Γ. ccBind v e·i))

lemma ccBinds-eq:
  ccBinds Γ·i = (∫ v→e∈map-of Γ. ccBind v e·i)
(proof)

lemma ccBinds-strict[simp]: ccBinds Γ·⊥=⊥
(proof)

lemma ccBinds-strict'[simp]: ccBinds Γ·(⊥,⊥)=⊥
(proof)

lemma ccBinds-reorder1:
  assumes map-of Γ v = Some e
  shows ccBinds Γ = ccBind v e ⊔ ccBinds (delete v Γ)
(proof)

lemma ccBinds-Nil[simp]:
  ccBinds [] = ⊥
(proof)

lemma ccBinds-Cons[simp]:
  ccBinds ((x,e)#Γ) = ccBind x e ⊔ ccBinds (delete x Γ)
(proof)

lemma ccBind-below-ccBinds: map-of Γ x = Some e ⇒ ccBind x e·ae ⊆ (ccBinds Γ·ae)
(proof)

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

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

lemma ccBindsExtra-simp: ccBindsExtra Γ·i =snd i ⊔ ccBinds Γ·i ⊔ (∫ x→e∈map-of Γ. ccProd (fv e) (ccNeighbors x (snd i)))
(proof)

lemma ccBindsExtra-eq: ccBindsExtra Γ·(ae,G) =
  G ⊔ ccBinds Γ·(ae,G) ⊔ (∫ x→e∈map-of Γ. fv e G × ccNeighbors x G)
(proof)
lemma ccBindsExtra-strict[simp]: ccBindsExtra \( \Gamma \cdot \bot = \bot \)
\begin{proof}
\end{proof}

lemma ccField-ccBindsExtra:
\[
ccField (ccBindsExtra \ \Gamma \cdot (ae, G)) \subseteq fv \ \Gamma \cup ccField \ G
\]
\begin{proof}
\end{proof}

end

lemma ccBind-eqvt[eqvt]: \( \pi \cdot (CoCallAnalysis.ccBind \ cccExp x e) = CoCallAnalysis.ccBind (\pi \cdot cccExp) (\pi \cdot x) (\pi \cdot e) \)
\begin{proof}
\end{proof}

lemma ccBinds-eqvt[eqvt]: \( \pi \cdot (CoCallAnalysis.ccBinds \ cccExp \ \Gamma) = CoCallAnalysis.ccBinds (\pi \cdot cccExp) (\pi \cdot \Gamma) \)
\begin{proof}
\end{proof}

lemma ccBindsExtra-eqvt[eqvt]: \( \pi \cdot (CoCallAnalysis.ccBindsExtra \ cccExp \ \Gamma) = CoCallAnalysis.ccBindsExtra (\pi \cdot cccExp) (\pi \cdot \Gamma) \)
\begin{proof}
\end{proof}

lemma ccBind-cong[fundef-cong]:
\[
cccexp1 e = cccexp2 e \implies CoCallAnalysis.ccBind \ cccexp1 x e = CoCallAnalysis.ccBind \ cccexp2 x e
\]
\begin{proof}
\end{proof}

lemma ccBinds-cong[fundef-cong]:
\[
[ (\forall \ e \in \snd \ set \ heap2 \implies cccexp1 e = cccexp2 e); \ heap1 = heap2 ] 
\implies CoCallAnalysis.ccBinds cccexp1 heap1 = CoCallAnalysis.ccBinds cccexp2 heap2
\]
\begin{proof}
\end{proof}

lemma ccBindsExtra-cong[fundef-cong]:
\[
[ (\forall \ e \in \snd \ set \ heap2 \implies cccexp1 e = cccexp2 e); \ heap1 = heap2 ] 
\implies CoCallAnalysis.ccBindsExtra cccexp1 heap1 = CoCallAnalysis.ccBindsExtra cccexp2 heap2
\]
\begin{proof}
\end{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) ⊔ ccProd { x } (insert x (fv e)) ⊆ ccExp (App e x)·a
  assumes ccExp-Lam: cc-restr (fv (Lam [y]. e)) (ccExp e · (pred-n)) ⊆ ccExp (Lam [y]. e)·n
  assumes ccExp-subst: x ∉ S ⟹ y ∉ S ⟹ cc-restr S (ccExp e[ y:= x] ) ⊆ cc-restr S (ccExp e · a)

  assumes ccExp-pap: isVal e ⟹ ccExp e · 0 = ccSquare (fv e)
  assumes ccExp-Let: cc-restr (−domA Γ) (ccHeap Γ e·a) ⊆ ccExp (Let Γ e · a)
  assumes ccExp-IfThenElse: ccExp scrut ⊔ (ccExp e1 · a ⊔ ccExp e2 · a) ⊆ ccProd (edom (Aexp scrut·0) ⊔ edom (Aexp e1·a) ⊔ edom (Aexp e2·a)) ⊆ ccExp ( scrut ? e1 : e2)·a

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

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

end

12.5 CoCallFix

theory CoCallFix
imports CoCallAnalysisSig CoCallAnalysisBinds ArityAnalysisSig Launchbury.Env−Nominal ArityAnalysisFix
begin

locale CoCallArityAnalysis =
  fixes ccExp :: exp ⇒ (Arity → AEnv × CoCalls)
begin

definition Aexp :: exp ⇒ (Arity → AEnv)
where Aexp e = (Λ a. fst (ccExp e · a))
sublocale ArityAnalysis Aexp ⟨proof⟩

abbreviation Aexp-syn′ (Ax) where \( A_a \equiv (\lambda e. Aexp \cdot e \cdot a) \)
abbreviation Aexp-bol-syn′ (Ax⁻) where \( A_a^{-} \equiv (\lambda e. fup \cdot (Aexp \cdot e) \cdot a) \)

lemma Aexp-eq:
\( A_a e = \text{fst} (\alpha \text{ccExp} \cdot e \cdot a) \)
⟨proof⟩

lemma fup-Aexp-eq:
\( fup \cdot (Aexp \cdot e) \cdot a = \text{fst} (fup \cdot (\alpha \text{ccExp} \cdot e) \cdot a) \)
⟨proof⟩

definition CCexp :: exp ⇒ (Arity → CoCalls) where CCexp \( \Gamma = (\Lambda a. \text{snd} (\alpha \text{ccExp} \cdot \Gamma \cdot a)) \)
lemma CCexp-eq:
\( CCexp \cdot a = \text{snd} (\alpha \text{ccExp} \cdot a \cdot a) \)
⟨proof⟩

lemma fup-CCexp-eq:
\( fup \cdot (CCexp \cdot e) \cdot a = \text{snd} (fup \cdot (\alpha \text{ccExp} \cdot e) \cdot a) \)
⟨proof⟩

sublocale CoCallAnalysis CCexp ⟨proof⟩

definition CCfix :: heap ⇒ (AEnv × CoCalls) → CoCalls where CCfix \( \Gamma = (\Lambda aeG. (\mu G', \alpha \text{ccBindsExtr} \cdot \Gamma \cdot ((\text{fst} aeG, G') \sqcup \text{snd} aeG)) \)
lemma CCfix-eq:
\( CCfix \cdot (ae, G) = (\mu G', \alpha \text{ccBindsExtr} \cdot \Gamma \cdot (ae, G') \sqcup G) \)
⟨proof⟩

lemma CCfix-unroll: CCfix \( \Gamma \cdot (ae, G) = \alpha \text{ccBindsExtr} \cdot \Gamma \cdot (ae, CCfix \cdot (ae, G)) \sqcup G \)
⟨proof⟩

lemma fup-ccExp-restr-subst′:
assumes \( \forall a. \alpha \text{cc-restr} S (CCexp e[x::=y] \cdot a) = \alpha \text{cc-restr} S (CCexp \cdot e \cdot a) \)
shows \( \alpha \text{cc-restr} S (fup \cdot (CCexp e[x::=y] \cdot a)) = \alpha \text{cc-restr} S (fup \cdot (CCexp \cdot e) \cdot a) \)
⟨proof⟩

lemma ccBindsExtr-restr-subst′:
assumes \( \forall x' e a. (x', e) \in \text{set} \ \Gamma \implies \alpha \text{cc-restr} S (CCexp e[x::=y] \cdot a) = \alpha \text{cc-restr} S (CCexp \cdot e \cdot a) \)
assumes \( x \notin S \)
assumes \( y \notin S \)
assumes \( \text{dom} A \Gamma \subseteq S \)
shows \( \alpha \text{cc-restr} S (ccBindsExtr \cdot \Gamma[x::=y] \cdot (ae, G)) = \alpha \text{cc-restr} S (ccBindsExtr \cdot (ae f \upharpoonright S, \alpha \text{cc-restr} S G)) \)
⟨proof⟩

92
lemma ccBindsExtra-restr:
  assumes domA Γ ⊆ S
  shows cc-restr S (ccBindsExtra Γ · (ae, G)) = cc-restr S (ccBindsExtra Γ · (ae f |· S, cc-restr S G))
⟨proof⟩

lemma CCfix-restr:
  assumes domA Γ ⊆ S
  shows cc-restr S (CCfix Γ · (ae, G)) = cc-restr S (CCfix Γ · (ae f |· S, cc-restr S G))
⟨proof⟩

lemma ccField-CCfix:
  shows ccField (CCfix Γ · (ae, G)) ⊆ fv Γ ∪ ccField G
⟨proof⟩

lemma CCfix-restr-subst':
  assumes ℓ · x' e a. (x',e) ∈ set Γ ⇒ cc-restr S (CCexp e[x::=y]·a) = cc-restr S (CCexp e·a)
  assumes x /∈ S
  assumes y /∈ S
  assumes domA Γ ⊆ S
  shows cc-restr S (CCfix Γ[x::=y]·(ae, G)) = cc-restr S (CCfix Γ · (ae f |· S, cc-restr S G))
⟨proof⟩

end

lemma Aexp-e qvt[eqvt]: π · (CoCallArityAnalysis.Aexp cccExp e) = CoCallArityAnalysis.Aexp (π · cccExp) (π · e)
⟨proof⟩

lemma CCexp-e qvt[eqvt]: π · (CoCallArityAnalysis.CCexp cccExp e) = CoCallArityAnalysis.CCexp (π · cccExp) (π · e)
⟨proof⟩

lemma CCfix-e qvt[eqvt]: π · (CoCallArityAnalysis.CCfix cccExp Γ) = CoCallArityAnalysis.CCfix (π · cccExp) (π · Γ)
⟨proof⟩

lemma ccFix-cong[fnfdef-cong]:
  \[ \forall e. e ∈ snd · set heap2 ⇒ cccexp1 e = cccexp2 e; \] heap1 = heap2 \[⇒ CoCallArityAnalysis.CCfix cccexp1 heap1 = CoCallArityAnalysis.CCfix cccexp2 heap2 \]
⟨proof⟩

context CoCallArityAnalysis
begin


definition cccFix :: heap ⇒ ((AEnv × CoCalls) → (AEnv × CoCalls))
where cccFix Γ = (Λ i. (Afix Γ · (fst i ⊔ (λ·up·0) f′ thunks Γ), CCfix Γ·(Afix Γ·(fst i ⊔ (λ·up·0) f′ thunks Γ)), snd i))

lemma cccFix-eq:
cccFix Γ i = (Afix Γ·(fst i ⊔ (λ·up·0) f′ thunks Γ), CCfix Γ·(Afix Γ·(fst i ⊔ (λ·up·0) f′ thunks Γ)), snd i))
⟨proof⟩ end

lemma cccFix-eqvt[eqvt]: π · (CoCallArityAnalysis.cccFix cccExp Γ) = CoCallArityAnalysis.cccFix (π · cccExp) (π · Γ)
⟨proof⟩

lemma cccFix-cong[fundef-cong]:
[ (Λ e, e ∈ snd · set heap) ⇒ cccexp1 e = cccexp2 e); heap1 = heap2 ]
⇒ CoCallArityAnalysis.cccFix cccexp1 heap1 = CoCallArityAnalysis.cccFix cccexp2 heap2
⟨proof⟩

12.5.1 The non-recursive case

definition ABind-nonrec :: var ⇒ exp ⇒ AEnv × CoCalls ⇒ Arity⊥
where
ABind-nonrec x e = (Λ i. (if isVal e ∨ x −→ x[sgiving i] then fst i x else up·0))

lemma ABind-nonrec-eq:
ABind-nonrec x e · (ae, G) = (if isVal e ∨ x −→ x[G then ae x else up·0)
⟨proof⟩

lemma ABind-nonrec-eqvteqvt[eqvt]: π · (ABind-nonrec x e) = ABind-nonrec (π · x) (π · e)
⟨proof⟩

lemma ABind-nonrec-abovearg:
ae x ⊑ ABind-nonrec x e · (ae, G)
⟨proof⟩

definition Aheap-nonrec where
Aheap-nonrec x e = (Λ i. esing x · (ABind-nonrec x e·i))

lemma Aheap-nonrec-simp:
Aheap-nonrec x e·i = esing x · (ABind-nonrec x e·i)
⟨proof⟩

lemma Aheap-nonrec-lookup[simp]:
(Aheap-nonrec x e·i) x = ABind-nonrec x e·i
⟨proof⟩

lemma Aheap-nonrec-eqvteqvt[eqvt]:

94
\[ \pi \cdot (\text{Aheap-nonrec} \ e) = \text{Aheap-nonrec} \ (\pi \cdot x) \ (\pi \cdot e) \]

\[ \langle \text{proof} \rangle \]

**context** CoCallArityAnalysis

**begin**

**definition** Afix-nonrec

where Afix-nonrec \( x \ e = (\Lambda i. \ \text{fup}(A\text{exp} \ e) \cdot (A\text{Bind-nonrec} \ x \ e \cdot i) \sqcup \text{fst} \ i) \)

**lemma** Afix-nonrec-eq [simp]:

Afix-nonrec \( x \ e \cdot i = \text{fup}(A\text{exp} \ e) \cdot (A\text{Bind-nonrec} \ x \ e \cdot i) \sqcup \text{fst} \ i \)

\[ \langle \text{proof} \rangle \]

**definition** CCfix-nonrec

where CCfix-nonrec \( x \ e = (\Lambda i. \ \text{ccBind} \ x \ e \cdot (\text{Aheap-nonrec} \ x \ e \cdot i, \ \text{snd} \ i) \sqcup \text{ccProd} \ (fv \ e) \ (cc\text{Neighbors} \ x \ (\text{snd} \ i) - (\text{isVal} \ e \ \text{then} \ {} \ \text{else} \ \{x\}) \sqcup \text{snd} \ i) \)

**lemma** CCfix-nonrec-eq [simp]:

CCfix-nonrec \( x \ e \cdot i = \text{ccBind} \ x \ e \cdot (\text{Aheap-nonrec} \ x \ e \cdot i, \ \text{snd} \ i) \sqcup \text{ccProd} \ (fv \ e) \ (cc\text{Neighbors} \ x \ (\text{snd} \ i) - (\text{isVal} \ e \ \text{then} \ {} \ \text{else} \ \{x\}) \sqcup \text{snd} \ i) \)

\[ \langle \text{proof} \rangle \]

**definition** cccFix-nonrec :: \( \text{var} \Rightarrow \text{exp} \Rightarrow ((A\text{Env} \times \text{CoCalls}) \rightarrow (A\text{Env} \times \text{CoCalls})) \)

where 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]:

cccFix-nonrec \( x \ e \cdot i = (\text{Afix-nonrec} \ x \ e \cdot i, \ \text{CCfix-nonrec} \ x \ e \cdot i) \)

\[ \langle \text{proof} \rangle \]

end

**lemma** AFix-nonrec-eqvt [eqvt]: \( \pi \cdot (\text{CoCallArityAnalysis} \ . \ Afix-nonrec \ ccc\text{Exp} \ x \ e) = \text{CoCallArityAnalysis} \ . \ Afix-nonrec \ (\pi \cdot ccc\text{Exp}) \ (\pi \cdot x) \ (\pi \cdot e) \)

\[ \langle \text{proof} \rangle \]

**lemma** CCFix-nonrec-eqvt [eqvt]: \( \pi \cdot (\text{CoCallArityAnalysis} \ . \ CCfix-nonrec \ ccc\text{Exp} \ x \ e) = \text{CoCallArityAnalysis} \ . \ CCfix-nonrec \ (\pi \cdot ccc\text{Exp}) \ (\pi \cdot x) \ (\pi \cdot e) \)

\[ \langle \text{proof} \rangle \]

**lemma** cccFix-nonrec-eqvt [eqvt]: \( \pi \cdot (\text{CoCallArityAnalysis} \ . \ cccFix-nonrec \ ccc\text{Exp} \ x \ e) = \text{CoCallArityAnalysis} \ . \ cccFix-nonrec \ (\pi \cdot ccc\text{Exp}) \ (\pi \cdot x) \ (\pi \cdot e) \)

\[ \langle \text{proof} \rangle \]

**12.5.2 Combining the cases**

**context** CoCallArityAnalysis

95
begin

definition cccFix-choose :: heap ⇒ (AEnv × CoCalls) → (AEnv × CoCalls)
where cccFix-choose Γ = (if nonrec Γ then case-prod cccFix-nonrec (hd Γ) else cccFix Γ)

lemma cccFix-choose-simp1[simp]:
¬ nonrec Γ ⇒ cccFix-choose Γ = cccFix Γ
⟨proof⟩

lemma cccFix-choose-simp2[simp]:
x /∈ fv e ⇒ cccFix-choose [(x,e)] = cccFix-nonrec x e
⟨proof⟩

end

lemma cccFix-choose-eqt[eqt]: π (CoCallArityAnalysis.cccFix-choose cccExp Γ) = CoCallArityAnalysis.cccFix-choose (π · cccExp) (π · Γ)
⟨proof⟩

lemma cccFix-nonrec-cong[fundef-cong]:
cccexp1 e = cccexp2 e ⇒ CoCallArityAnalysis.cccFix-nonrec cccexp1 x e = CoCallArityAnalysis.cccFix-nonrec cccexp2 x e
⟨proof⟩

lemma cccFix-choose-cong[fundef-cong]:
[ (∧ e. e ∈ snd · set heap2 ⇒ cccexp1 e = cccexp2 e); heap1 = heap2 ]
⇒ CoCallArityAnalysis.cccFix-choose cccexp1 heap1 = CoCallArityAnalysis.cccFix-choose cccexp2 heap2
⟨proof⟩

end

12.6 CoCallGraph-.Tree

theory CoCallGraph-Tree
imports CoCallGraph.TTree-HOLCF
begin

lemma interleave-ccFromList:
xs ∈ interleave ys zs ⇒ ccFromList xs = ccFromList ys ⊔ ccFromList zs ⊔ ccProd (set ys)
⟨set zs⟩
⟨proof⟩

lift-definition cccApprox :: var tree ⇒ CoCalls
is λ xss . lub (ccFromList · xss)⟨proof⟩

lemma cccApprox-paths: cccApprox t = lub (ccFromList · (paths t)) ⟨proof⟩

lemma cccApprox-strict[simp]: cccApprox ⊥ = ⊥

96
\(\langle \text{proof} \rangle\)

**lemma** `in-ccApprox`: \( (x - y \in (ccApprox t)) \iff (\exists \; xs \in \text{paths } t. \; (x - y \in (ccFromList xs))) \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-mono`: \( \text{paths } t \subseteq \text{paths } t' \implies \text{ccApprox } t \subseteq \text{ccApprox } t' \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-mono'`: \( t \subseteq t' \implies \text{ccApprox } t \subseteq \text{ccApprox } t' \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-belowI`: \( (\forall \; xs \in \text{paths } t. \; \text{ccFromList } xs \subseteq G) \implies \text{ccApprox } t \subseteq G \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-below-iff`: \( \text{ccApprox } t \subseteq G \iff (\forall \; xs \in \text{paths } t. \; \text{ccFromList } xs \subseteq G) \)

\(\langle \text{proof} \rangle\)

**lemma** `cc-restr-ccApprox-below-iff`: \( \text{cc-restr } S \; (ccApprox t) \subseteq G \iff (\forall \; xs \in \text{paths } t. \; \text{cc-restr } S \; (ccFromList xs) \subseteq G) \)

\(\langle \text{proof} \rangle\)

**lemma** `ccFromList-below-ccApprox`: \( \text{xs } \in \text{paths } t \implies \text{ccFromList } xs \subseteq \text{ccApprox } t \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-nxt-below`: \( \text{ccApprox } (\text{nxt } t \; x) \subseteq \text{ccApprox } t \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-three-restr-nxt-below`: \( \text{ccApprox } (\text{three-restr } S \; (\text{nxt } t \; x)) \subseteq \text{ccApprox } (\text{three-restr } S \; t) \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-three-restr[simp]`: \( \text{ccApprox } (\text{three-restr } S \; t) = \text{cc-restr } S \; (\text{ccApprox } t) \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-both`: \( \text{ccApprox } (t \otimes t') = \text{ccApprox } t \sqcup \text{ccApprox } t' \sqcup \text{ccProd } (\text{carrier } t) \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-many-calls[simp]`: \( \text{ccApprox } (\text{many-calls } x) = \text{ccProd } \{ x \} \{ x \} \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-single[simp]`: \( \text{ccApprox } (\text{TTree.single } y) = \perp \)

\(\langle \text{proof} \rangle\)

**lemma** `ccApprox-either[simp]`: \( \text{ccApprox } (t \oplus t') = \text{ccApprox } t \sqcup \text{ccApprox } t' \)
Lemma wild-recursion:
assumes ccApprox $t \subseteq G$
assumes $\forall x. x \notin S \implies f x = \text{empty}$
assumes $\forall x. x \in S \implies ccApprox (f x) \subseteq G$
shows $ccApprox (\text{three-restr} (-S) (\text{substitute} f T t)) \subseteq G$
(proof)

Lemma wild-recursion-thunked:
assumes ccApprox $t \subseteq G$
assumes $\forall x. x \notin S \implies f x = \text{empty}$
assumes $\forall x. x \in S \implies ccApprox (f x) \subseteq G$
assumes $\forall x. x \in S \implies ccProd (ccNeighbors x G) (\text{carrier} (f x)) \subseteq G$
shows $ccApprox (\text{three-restr} (-S) (\text{substitute} f T t)) \subseteq G$
(proof)

Inductive-set valid-lists :: var set $\Rightarrow$ CoCalls $\Rightarrow$ var list set
for $S G$
where $\emptyset \in \text{valid-lists} S G$
| 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]: $\emptyset \in \text{valid-lists} S G$ $(x \# xs) \in \text{valid-lists} S G$
Inductive-cases valid-lists-ConsE: $(x \# xs) \in \text{valid-lists} S G$

Lemma valid-lists-downset-aux:
$xs \in \text{valid-lists} S \text{CoCalls} \implies \text{butlast} xs \in \text{valid-lists} S \text{CoCalls}$
(proof)

Lemma valid-lists-subset: $xs \in \text{valid-lists} S G \implies \text{set} xs \subseteq S$
(proof)

Lemma valid-lists-mono1:
assumes $S \subseteq S'$
shows $\text{valid-lists} S G \subseteq \text{valid-lists} S' G$
(proof)

Lemma valid-lists-chain1:
assumes chain $Y$
assumes $xs \in \text{valid-lists} (\text{UNION} \text{UNIV} Y) G$
shows $\exists i. xs \in \text{valid-lists} (Y i) G$
(proof)

Lemma valid-lists-chain2:
assumes chain $Y$
assumes $xs \in \text{valid-lists } S \sqcup i. Y i$
shows $\exists i. xs \in \text{valid-lists } S \sqcup (Y i)$
(proof)

lemma valid-lists-cc-restr: $\text{valid-lists } S G = \text{valid-lists } (\text{cc-restr } S G)$
(proof)

lemma interleave-valid-list: $xs \in ys \otimes zs \implies ys \in \text{valid-lists } S G \implies zs \in \text{valid-lists } S' G' \implies xs \in \text{valid-lists } (S \sqcup S') (G \sqcup (G' \sqcup \text{ccProd } S S'))$
(proof)

lemma interleave-valid-list': $xs \in \text{valid-lists } (S \sqcup S') G \implies \exists ys zs. xs \in ys \otimes zs \land ys \in \text{valid-lists } S G \land zs \in \text{valid-lists } S' G$
(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 :: var set $\Rightarrow$ CoCalls $\Rightarrow$ 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: $xs \in \text{valid-lists } S G \implies \text{ccFromList } xs \subseteq \text{cc-restr } S G$
(proof)

lemma ccApprox-ccTTree[simp]: $\text{ccApprox } (\text{ccTTree } S G) = \text{cc-restr } S G$
(proof)

lemma below-ccTTree1: assumes $\text{carrier } t \subseteq S \text{ and } \text{ccApprox } t \subseteq G$
shows $t \subseteq \text{ccTTree } S G$
(proof)

lemma ccTTree-mono1: $S \subseteq S' \implies \text{ccTTree } S G \subseteq \text{ccTTree } S' G$
(proof)
lemma cont-\alphaTree1:
  \( \text{cont} (\lambda S. \alphaTree S G) \)
  \( \langle \text{proof} \rangle \)

lemma \alphaTree-mono2:
  \( G \subseteq G' \implies \alphaTree S G \subseteq \alphaTree S G' \)
  \( \langle \text{proof} \rangle \)

lemma \alphaTree-mono:
  \( S \subseteq S' \implies G \subseteq G' \implies \alphaTree S G \subseteq \alphaTree S' G' \)
  \( \langle \text{proof} \rangle \)

lemma cont-\alphaTree2:
  \( \text{cont} (\alphaTree S) \)
  \( \langle \text{proof} \rangle \)

lemmas cont-\alphaTree = cont-compose2[where \( c = \alphaTree \), OF cont-\alphaTree1 cont-\alphaTree2, simp, cont2and]

lemma \alphaTree-below-single1:
  \( \text{assumes } S \cap S' = \{\} \)
  \( \text{shows } \alphaTree S G \subseteq \text{singles } S' \)
  \( \langle \text{proof} \rangle \)

lemma \alphaTree-\alpha-restr:
  \( \alphaTree S G = \alphaTree (\alpha-restr S G) \)
  \( \langle \text{proof} \rangle \)

lemma \alphaTree-cong-below:
  \( \alpha-restr S G \subseteq \alpha-restr S G' \implies \alphaTree S G \subseteq \alphaTree S G' \)
  \( \langle \text{proof} \rangle \)

lemma \alphaTree-cong:
  \( \alpha-restr S G = \alpha-restr S G' \implies \alphaTree S G = \alphaTree S G' \)
  \( \langle \text{proof} \rangle \)

lemma either-\alphaTree:
  \( \alphaTree S G \oplus \alphaTree S' G' \subseteq \alphaTree (S \cup S') (G \cup G') \)
  \( \langle \text{proof} \rangle \)

lemma interleave-\alphaTree:
  \( \alphaTree S G \otimes \alphaTree S' G' \subseteq \alphaTree (S \cup S') (G \cup G' \cup \alphaProd S S') \)
  \( \langle \text{proof} \rangle \)

lemma interleave-\alphaTree':
  \( \alphaTree (S \cup S') G \subseteq \alphaTree S G \otimes \alphaTree S' G \)
  \( \langle \text{proof} \rangle \)

lemma many-calls-\alphaTree:

100
shows many-calls $x = \text{ccTTree} \{x\} \ (\text{ccProd} \ \{x\} \ \{x\})$

(\text{proof})

\textbf{lemma filter-valid-lists':}
\begin{align*}
  xs \in \text{valid-lists} \{x' \in S. \ P x' \} \ G \implies xs \in \text{filter} P' \ \text{valid-lists} \ S \ G
\end{align*}
(\text{proof})

\textbf{lemma without-ccTTree[simp]:}
\begin{align*}
  \text{without} x \ (\text{ccTTree} \ S \ G) = \text{ccTTree} \ (S - \{x\}) \ G
\end{align*}
(\text{proof})

\textbf{lemma tree-restr-ccTTree[simp]:}
\begin{align*}
  \text{tree-restr} S' \ (\text{ccTTree} \ S \ G) = \text{ccTTree} \ (S \cap S') \ G
\end{align*}
(\text{proof})

\textbf{lemma repeatable-ccTTree-ccSquare:}
\begin{align*}
  S \subseteq S' \implies \text{repeatable} \ (\text{ccTTree} \ S \ (\text{ccSquare} \ S'))
\end{align*}
(\text{proof})

An alternative definition

\textbf{inductive valid-lists' :: var set \Rightarrow CoCalls \Rightarrow var set \Rightarrow var list \Rightarrow bool}
\begin{align*}
  \text{for} \ S \ G
  \begin{align*}
  \text{where} \ & \text{valid-lists'} \ S \ G \ \text{prefix} [] \ \text{prefix} \subseteq \text{ccNeighbors} x \ G \implies \text{valid-lists'} \ S \ G \ (\text{insert} x \ \text{prefix}) \ xs \implies x \in S \implies \text{valid-lists'} \ S \ G \ \text{prefix} \ (x \# xs)
  \end{align*}
\end{align*}

\textbf{inductive-simps valid-lists'-simps[simp]:}
\begin{align*}
  \text{valid-lists'} \ S \ G \ \text{prefix} [] \ \text{valid-lists'} \ S \ G \ \text{prefix} \ (x \# xs)
\end{align*}

\textbf{inductive-cases valid-lists'-ConsE:}
\begin{align*}
  \text{valid-lists'} \ S \ G \ \text{prefix} \ (x \# xs)
\end{align*}

\textbf{lemma valid-lists-valid-lists':}
\begin{align*}
  xs \in \text{valid-lists} \ S \ G \implies \text{ccProd} \ \text{prefix} \ (\text{set} \ xs) \ \subseteq G \implies \text{valid-lists'} \ S \ G \ \text{prefix} \ xs
\end{align*}
(\text{proof})

\textbf{lemma valid-lists'-valid-lists-aux:}
\begin{align*}
  \text{valid-lists'} \ S \ G \ \text{prefix} \ xs \implies x \in \text{prefix} \implies \text{ccProd} \ \text{prefix} \ (\text{set} \ xs) \ \{x\} \ \subseteq G
\end{align*}
(\text{proof})

\textbf{lemma valid-lists'-valid-lists:}
\begin{align*}
  \text{valid-lists'} \ S \ G \ \text{prefix} \ xs \implies xs \in \text{valid-lists} \ S \ G
\end{align*}
(\text{proof})

Yet another definition

\textbf{lemma valid-lists-characterization:}
\begin{align*}
  xs \in \text{valid-lists} \ S \ G \iff \text{set} \ xs \subseteq S \land (\forall n. \ \text{ccProd} \ (\text{set} \ n \ xs)) \ (\text{set} \ (\text{drop} \ n \ xs)) \ \subseteq G
\end{align*}
(\text{proof})

\textbf{end
12.7 CoCallImplTTTree

theory CoCallImplTTTree
imports TTreeAnalysisSig Env Set CpCoCallAritySig CoCallGraph TTree
begin

context CoCallArity
begin

definition Texp :: exp ⇒ Arity ⇒ var TTree
  where Texp e = (Λ a. cCTree (edom (Aexp e · a)) (ccExp e · a))

lemma Texp-simp: Texp e · a = cCTree (edom (Aexp e · a)) (ccExp e · a)
⟨proof⟩

sublocale TTreeAnalysis Texp ⟨proof⟩
end

end

12.8 CoCallImplTTTreeSafe

theory CoCallImplTTTreeSafe
imports CoCallImplTTTree CoCallAnalysisSpec TTreeAnalysisSpec
begin

lemma valid-lists-many-calls:
  assumes ¬one-call-in-path x p
  assumes p ∈ valid-lists S G
  shows x −− x ∈ G
⟨proof⟩

context CoCallArityEdom
begin

lemma carrier-Fexp': carrier (Texp · e · a) ⊆ fv e
⟨proof⟩

end

context CoCallAritySafe
begin

lemma carrier-AnalBinds-below:
  carrier (((Texp. AnalBinds ∆(Aheap ∆ e · a)) x) ⊆ edom (((ABinds ∆):(Aheap ∆ e · a))
⟨proof⟩

sublocale TTreeAnalysisCarrier Texp

102
sublocale TTreeAnalysisSafe Texp

definition Theap :: heap ⇒ exp ⇒ Arity ⇒ var three
  where Theap Γ e = (Λ a. if nonrec Γ then ccTTree (edom (Aheap Γ e a)) (ccExp e a) else
tree-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
tree-restr (edom (Aheap Γ e a)) anything)

lemma carrier-Fheap’:carrier (Theap Γ e a) = edom (Aheap Γ e a)

sublocale TTreeAnalysisCardinalityHeap Texp Aexp Aheap Theap

lemma paths-singles: xs ∈ paths (singles S) ←→ (∀ x ∈ S. one-call-in-path x xs)

lemma paths-singles’: xs ∈ paths (singles S) ←→ (∀ x ∈ (set xs ∩ S). one-call-in-path x xs)

lemma both-below-singles1:
  assumes t ⊆ singles S
  assumes carrier t’ ∩ S = {}  
  shows t ⊗ t’ ⊆ singles S
  (proof)

lemma paths-three-restr-singles: xs ∈ paths (tree-restr S’ (singles S)) ←→ set xs ⊆ S’ ∧ (∀x ∈ S. one-call-in-path x xs)

lemma substitute-not-carrier:
  assumes x /∈ carrier t
  assumes \( \land \) \( x', x \notin carrier (f x') \)
  shows x /∈ carrier (substitute f T t)
  (proof)
lemma substitute-below-singles1:
  assumes t ⊑ singles S
  assumes ∃ x. carrier (f x) ∩ S = {}
  shows substitute f T t ⊑ singles S
⟨proof⟩
end

13 CoCall Cardinality Implementation

13.1 CoCallAnalysisImpl

theory CoCallAnalysisImpl
imports Arity - Nominal Launchbury. Nominal - HOLCF Launchbury. Env - Nominal Env - Set - Cpo Launchbury. Env - HOLCF CoCallFix
begin

fun combined-restrict :: var set ⇒ (AEnv × CoCalls) ⇒ (AEnv × CoCalls)
  where combined-restrict S (env, G) = (env f |\ S, cc-restr S G)

lemma fst-combined-restrict[simp]:
  fst (combined-restrict S p) = fst p f |\ S
⟨proof⟩

lemma snd-combined-restrict[simp]:
  snd (combined-restrict S p) = cc-restr S (snd p)
⟨proof⟩

lemma combined-restrict-eqv[eqv]:
  shows π · combined-restrict S p = combined-restrict (π · S) (π · p)
⟨proof⟩

lemma combined-restrict-cont:
  cont (λx. combined-restrict S x)
⟨proof⟩

lemmas combined-restrict-perm:
  assumes supp π ⩾ S and [simp]: finite S
  shows combined-restrict S (π · p) = combined-restrict S p
⟨proof⟩

definition predCC :: var set ⇒ (Arity → CoCalls) ⇒ (Arity → CoCalls)
  where predCC S f = (Λ a. if a ≠ 0 then cc-restr S (f · (pred-a)) else ccSquare S)

lemma predCC-eq:
  shows predCC S f · a = (if a ≠ 0 then cc-restr S (f · (pred-a)) else ccSquare S)

104
lemma \textit{predCC\text{-eqt}[\text{eqvt, simp}]}: \pi \cdot (\text{predCC} \ S \ f) = \text{predCC} \ (\pi \cdot S) \ (\pi \cdot f)

\langle \text{proof} \rangle

lemma \textit{cc\text{-restt-predCC}}:
\texttt{cc\text{-restt} \ S \ (\text{predCC} \ (S' \cap S) \ (\Lambda \ n. \ cc\text{-restt} \ S \ (f \cdot n))) \cdot n}

\langle \text{proof} \rangle

lemma \textit{cc\text{-restt-predCC}[\text{simp}]}:
\texttt{cc\text{-restt} \ S \ (\text{predCC} \ S \cdot n) = \text{predCC} \ S \cdot n}

\langle \text{proof} \rangle

\textbf{nominal-function}
\texttt{cCCexp :: \text{exp} \Rightarrow (\text{Arity} \rightarrow \text{AEnv} \times \text{CoCalls})}
\texttt{where}
cCCexp \ (\text{Var} \ x) = (\Lambda \ n. \ (\text{esing} \ x \cdot (\text{up} \cdot n), \perp))
cCCexp \ (\text{Lam} \ [x], \ e) = (\Lambda \ n. \ (\text{combined-restrict} \ (\text{fv} \ (\text{Lam} \ [x], \ e)) \ (\text{fst} \ (cCCexp \ e\cdot(\text{pred}\cdot n))),
\text{predCC} \ (\text{fv} \ (\text{Lam} \ [x], \ e)) \ (\Lambda \ a. \ \text{snd}(cCCexp \ e\cdot a)\cdot n))
cCCexp \ (\text{App} \ x \ e) = (\Lambda \ n. \ (\text{fst} \ (cCCexp \ e\cdot(\text{inc}\cdot n)) \cup (\text{esing} \ x \cdot (\text{up} \cdot 0)), \ \text{snd} \ (cCCexp \ e\cdot(\text{inc} \cdot n)) \cup \text{ccProd} \ (\text{x} \ (\text{insert} \ x \ (\text{fv} \ e)))
cCCexp \ (\text{Let} \ \Gamma \ e) = (\Lambda \ n. \ (\text{combined-restrict} \ (\text{fv} \ (\text{Let} \ \Gamma \ e)) \ (\text{CoCallArityAnalysis.cccFix-choose cCCexp} \ \Gamma \cdot (cCCexp \ e\cdot n)))
cCCexp \ (\text{Bool} \ b) = \perp
\texttt{cCCexp \ (\text{ scrut} \ ? \ e1 : e2) = (\Lambda \ n. \ (\text{fst} \ (cCCexp \ \text{ scrut} \cdot \theta) \cup \text{fst} \ (cCCexp \ e1\cdot n) \cup \text{fst} \ (cCCexp \ e2\cdot n)),
\text{snd} \ (cCCexp \ \text{ scrut} \cdot \theta) \cup (\text{snd} \ (cCCexp \ e1\cdot n) \cup \text{snd} \ (cCCexp \ e2\cdot n)) \cup \text{ccProd} \ (\text{edom} \ (\text{fst} \ (cCCexp \ \text{ scrut} \cdot \theta))) \ (\text{edom} \ (\text{fst} \ (cCCexp \ e1\cdot n)) \cup \text{edom} \ (\text{fst} \ (cCCexp \ e2\cdot n))))}
\langle \text{proof} \rangle

\textbf{nominal-termination} \ (\text{eqvt}) \ (\text{proof})

\textbf{locale CoCallAnalysisImpl}
\textbf{begin}
\textbf{sublocale} CoCallArityAnalysis \ cCCexp\langle \text{proof} \rangle
\textbf{sublocale} ArityAnalysis \ Aexp\langle \text{proof} \rangle

\textbf{abbreviation} Aexp-syn\'' \ (A_\cdot) \ \text{where} \ A_a \ e \equiv Aexp \ e \cdot a
\textbf{abbreviation} Aexp-bol-syn\'' \ (A^\cdot \cdot) \ \text{where} \ A^+_a \ e \equiv fup \cdot (Aexp \ e) \cdot a

\textbf{abbreviation} ccExp-syn\'' \ (G_\cdot) \ \text{where} \ G_a \ e \equiv CCexp \ e \cdot a
\textbf{abbreviation} ccExp-bol-syn\'' \ (G^\cdot \cdot) \ \text{where} \ G^+_a \ e \equiv fup \cdot (CCexp \ e) \cdot a

\textbf{lemma} cCCexp\text{-eq\text{-simp}}:
cCCexp \ (\text{Var} \ x) \cdot n = (\text{esing} \ x \cdot (\text{up} \cdot n), \perp)
cCCexp \ (\text{Lam} \ [x], \ e) \cdot n = \text{combined-restrict} \ (\text{fv} \ (\text{Lam} \ [x], \ e)) \ (\text{fst} \ (cCCexp \ e\cdot(\text{pred}\cdot n)), \text{predCC} \ (\text{fv} \ (\text{Lam} \ [x], \ e)) \ (\Lambda \ a. \ \text{snd}(cCCexp \ e\cdot a)\cdot n))

\texttt{105}
\[\begin{align*}
cCCexp\ (\text{App } e\ x)\ &=\ (\text{fst}\ (cCCexp\ e\cdot(\text{inc}\cdot n)) \cup (\text{esing } x \cdot (\text{up}\cdot 0))), \\
\text{snd}\ (cCCexp e\cdot(\text{inc}\cdot n)) \cup \text{ccProd } \{x\} (\text{insert } x \ (fv\ e))
\end{align*}\]
cCCexp\ (\text{Let } \Gamma\ e)\ &=\ \text{combined-restr}\ (fv\ (\text{Let } \Gamma\ e)) \ (\text{CoCallArityAnalysis.ceeFix-choose})
cCCexp\ \Gamma\cdot(cCCexp\ e\cdot n))
cCCexp\ (\text{Bool } b)\ &=\ \bot
\end{equation*}\]
cCCexp\ (\text{ scrut } ?\ e1 : e2)\ &=\ (\text{fst}\ (cCCexp\ scrut\cdot0) \cup \text{fst}\ (cCCexp\ e1\cdot n) \cup \text{fst}\ (cCCexp\ e2\cdot n)),
\text{snd}\ (cCCexp\ scrut\cdot0) \cup (\text{snd}\ (cCCexp\ e1\cdot n) \cup \text{ccProd } (\text{fst}\ (cCCexp\ scrut\cdot0))) (\text{edom } (\text{fst}\ (cCCexp\ e1\cdot n)) \cup \text{edom } (\text{fst}\ (cCCexp\ e2\cdot n)))
\end{equation*}\]
\text{declare } cCCexp.simps[simp del]

**Lemma** \(\text{Aexp-pre-simps}:
\begin{align*}
\text{A}_a\ (\text{Var } x)\ &=\ \text{esing } x\cdot(\text{up}\cdot a) \\
\text{A}_a\ (\text{Lam } [x], e)\ &=\ \text{Aexp } e\cdot(\text{pred}\cdot a) f\cdot fv\ (\text{Lam } [x], e) \\
\text{A}_a\ (\text{App } e\ x)\ &=\ \text{Aexp } e\cdot(\text{inc}\cdot a) \cup \text{esing } x\cdot(\text{up}\cdot 0) \\
\neg\ \text{nonrec } \Gamma\ \implies\ \\
\text{A}_a\ (\text{Let } \Gamma\ e)\ &=\ (\text{Afix } \Gamma\cdot(\text{A}_a\ e \cup (\lambda\cdot.\text{up}\cdot 0) f\cdot \text{thunks } \Gamma)) f\cdot (fv\ (\text{Let } \Gamma\ e)) \\
x\ \notin\ f v\ e\ \implies\ \\
\text{A}_a\ (\text{let } x\ \text{be } e\ \text{in } exp)\ &=\ \\
(\text{sup } (\text{Aexp } e\cdot(\text{ABind-nonrec } x\cdot e\cdot (\text{A}_a\ \text{exp}, cCCexp\ \text{exp}\cdot n))) \cup \text{A}_a\ \text{exp}) f\cdot (fv\ (\text{let } x\ \text{be } e\ \text{in } exp)) \\
\text{A}_a\ (\text{Bool } b)\ &=\ \bot \\
\text{A}_a\ (\text{ scrut } ?\ e1 : e2)\ &=\ \text{A}_0\ \text{ scrut} \cup \text{A}_a\ e1 \cup \text{A}_a\ e2 \\
\end{align*}\]
\text{(proof)}

**Lemma** \(\text{CCexp-pre-simps}:
\begin{align*}
cCCexp\ (\text{Var } x)\ &=\ \bot \\
cCCexp\ (\text{Lam } [x], e)\ &=\ \text{prodCC } (fv\ (\text{Lam } [x], e)) \ (cCCexp\ e)\cdot n \\
cCCexp\ (\text{App } e\ x)\ &=\ cCCexp\ e\cdot(\text{inc}\cdot n) \cup \text{ccProd } \{x\} (\text{insert } x \ (fv\ e)) \\
\neg\ \text{nonrec } \Gamma\ \implies\ \\
cCCexp\ (\text{Let } \Gamma\ e)\ &=\ \text{cc-restr}\ (fv\ (\text{Let } \Gamma\ e)) \\
(cCCexp\ (\text{Afix } \Gamma\cdot(\text{Aexp } e\cdot n \cup (\lambda\cdot.\text{up}\cdot 0) f\cdot \text{thunks } \Gamma), cCCexp\ e\cdot n)) \\
x\ \notin\ f v\ e\ \implies\ cCCexp\ (\text{let } x\ \text{be } e\ \text{in } exp)\ &=\ \\
\text{cc-restr}\ (fv\ (\text{let } x\ \text{be } e\ \text{in } exp)) \\
(\text{ccBind } x\cdot e\cdot (\text{Aexp}\cdot\text{nonrec } x\cdot e\cdot (\text{Aexp}\cdot\text{exp}\cdot n), cCCexp\ e\cdot n), cCCexp\ e\cdot n) \\
\cup\ \text{ccProd } (fv\ e\cdot (\text{ccNeighbors } x\cdot (cCCexp\ e\cdot n) - (\text{ifVal } e\ \text{then } \{\} \ \text{else } \{x\})) \cup cCCexp\ e\cdot n) \\
cCCexp\ (\text{Bool } b)\ &=\ \bot \\
cCCexp\ (\text{ scrut } ?\ e1 : e2)\ &=\ \\
cCCexp\ scrut\cdot0 \cup \\
(cCCexp\ e1\cdot n \cup cCCexp\ e2\cdot n) \cup \\
\text{ccProd } (\text{edom } (\text{Aexp } scrut\cdot0)) (\text{edom } (\text{Aexp } e1\cdot n) \cup \text{edom } (\text{Aexp } e2\cdot n)) \\
\end{align*}\]
\text{(proof)}

**Lemma** \(\text{shows ccField-CCexp}: \text{ccField } (cCCexp\ e\cdot a) \subseteq fv\ e\ \text{and } Aexp-edom:\ \text{edom } (\text{A}_a\ e) \subseteq fv\ e\)

106
lemma cc-restr-CCexp[simp]:
  cc-restr (fv e) (CCexp e·a) = CCexp e·a
⟨proof⟩

lemma ccField-fup-CCexp:
  ccField (fup·(CCexp e)·n) ⊆ fv e
⟨proof⟩

lemma cc-restr-fup-ccExp-useless[simp]: cc-restr (fv e) (fup·(CCexp e)·n) = fup·(CCexp e)·n
⟨proof⟩

sublocale EdomArityAnalysis Aexp ⟨proof⟩

lemma CCexp-simps[simp]:
  Ga(Var x) = ⊥
  Ga(Lam [x]. e) = (fv (Lam [x]. e))^2
  Ginc-a(Lam [x]. e) = cc-delete x (Ga e)
  Ga(App e x) = Ginc-a e ⊗ {x} G×insert x (fv e)
  ¬nonrec Γ =⇒ Ga (Let Γ e) =
    (CCfix Γ·(Afix Γ·(Aa e ⊗ (λ·up·0) f| thunks Γ), Ga e)) G| (¬ domA Γ)
  x ∉ fv e' =⇒ Ga (let x be e' in e) =
    cc-delete x
      (ccBind x e'·(Aheap-nonrec x e'·(Aa e, Ga e), Ga e))
    ⊗ fv e' G× (ccNeighbors x (Ga e) - (if isVal e' then {} else {x})) ⊃ Ga e)
  Ga (Bool b) = ⊥
  Ga (scrad ? e1 : e2) =
    Ga scrad ⊃ (Ga a e1 ⊃ Ga e2) ⊃ edom (Aa scrad) G× (edom (Aa e1) ⊃ edom (Aa e2))
⟨proof⟩

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

lemma Aheap-simp1[simp]:
  ¬ nonrec Γ =⇒ Aheap Γ e · a = (Afix Γ·(Aexp e·a ⊗ (λ·up·0) f| thunks Γ)) f| domA Γ
⟨proof⟩

lemma Aheap-simp2[simp]:
  x /∈ fv e' =⇒ Aheap [(x,e')·e] · a = Aheap-nonrec x e'·(Aexp e·a, CCexp e·a)
⟨proof⟩

lemma Aheap-eqvt'[eqvt]:
  π · (Aheap Γ e) = Aheap (π · Γ) (π · e)
⟨proof⟩

sublocale ArityAnalysisHeap Aheap ⟨proof⟩

107
lemma Aexp-lam-simp: Aexp (Lam [x], e) · n = env-delete x (Aexp e · (pred · n))
(proof)

lemma Aexp-Let-simp1:
¬ nonrec Γ ⇒ Aa (Let Γ e) = (Afix Γ · (Aa e ⊔ (λ.-up·0) f ′ | thanks Γ)) f ′ · (¬ domA Γ)
(proof)

lemma Aexp-Let-simp2:
x /∈ fv e ⇒ Aa (let x be e in exp) = env-delete x (A⁺ Abind-nonrec x e · (Aa exp, C Cexp exp-a) e ⊔ Aa exp)
(proof)

lemma Aexp-simps[simp]:
Aa (Var x) = esing x · (up·a)
Aa (Lam [x], e) = env-delete x (A⁺ pred·e)
Aa (App e x) = Aexp e · (inc·a) ⊔ esing x · up·0
¬ nonrec Γ ⇒ Aa (Let Γ e) =
(Afix Γ · (Aa e ⊔ (λ.-up·0) f ′ | thanks Γ)) f ′ · (¬ domA Γ)
x /∈ fv e ′ ⇒ Aa (let x be e ′ in e) =
env-delete x (A⁺ Abind-nonrec x e ′ · (Aa e, ⊆ a e) e ′ ⊔ Aa e)
Aa (Bool b) = ⊥
Aa (scut ? e1 : e2) = A₀ scut ⊔ Aa e1 ⊔ Aa e2
(proof)

end

end

13.2 CoCallImplSafe

theory CoCallImplSafe
imports CoCallAnalysisImpl CoCallAnalysisSpec ArityAnalysisFixProps
begin

locale CoCallImplSafe
begin
sublocale CoCallAnalysisImpl(proof)

lemma ccNeighbors-Int-cresstr: (ccNeighbors x G ∩ S) = ccNeighbors x (cc-restr (insert x S) G) ∩ S
(proof)
lemma
assumes $x \notin S$ and $y \notin S$
shows $CCexp-subst \cdot cc-restr S (CCexp e[y::=x] \cdot a) = cc-restr S (CCexp e \cdot a) \wedge Aexp-restr-subst: (Aexp e[y::=x] \cdot a) f \cdot S = (Aexp e \cdot a) f \cdot S$
(proof)

sublocale ArityAnalysisSafe Aexp
(proof)

sublocale ArityAnalysisLetSafe Aexp Aheap
(proof)

definition ccHeap-nonrec
where $ccHeap-nonrec x e exp = (\Lambda n. CCfix-nonrec x e \cdot (Aexp exp \cdot n, CCexp exp \cdot n))$

lemma ccHeap-nonrec-eq:
$ccHeap-nonrec x e exp \cdot n = CCfix-nonrec x e \cdot (Aexp exp \cdot n, CCexp exp \cdot n)$
(proof)

definition ccHeap-rec :: heap $\Rightarrow$ exp $\Rightarrow$ Arity $\Rightarrow$ CoCalls
where $ccHeap-rec \Gamma e = (\Lambda a. CCfix \Gamma \cdot (Afix \Gamma \cdot (Aexp e \cdot a \sqcup (\lambda \cdot up \cdot 0) f \cdot (\text{thanks } \Gamma)), CCexp e \cdot a))$

lemma ccHeap-rec-eq:
$ccHeap-rec \Gamma e \cdot a = CCfix \Gamma \cdot (Afix \Gamma \cdot (Aexp e \cdot a \sqcup (\lambda \cdot up \cdot 0) f \cdot (\text{thanks } \Gamma)), CCexp e \cdot a)$
(proof)

definition ccHeap :: heap $\Rightarrow$ exp $\Rightarrow$ Arity $\Rightarrow$ CoCalls
where $ccHeap \Gamma = (\text{if nonrec } \Gamma \text{ then case-prod } ccHeap-nonrec \cdot \text{(hd } \Gamma) \text{ else } ccHeap-rec \Gamma)\)$

lemma ccHeap-simp1:
$\neg \text{ nonrec } \Gamma \implies ccHeap \Gamma e \cdot a = CCfix \Gamma \cdot (Afix \Gamma \cdot (Aexp e \cdot a \sqcup (\lambda \cdot up \cdot 0) f \cdot (\text{thanks } \Gamma)), CCexp e \cdot a)$
(proof)

lemma ccHeap-simp2:
$x \notin \text{fv } e \implies ccHeap [(x,e)] exp \cdot n = CCfix-nonrec x e \cdot (Aexp exp \cdot n, CCexp exp \cdot n)$
(proof)

sublocale CoCalAritySafe CCexp Aexp ccHeap Aheap
(proof)
end
14 End-to-end Safety Results and Example

14.1 CallArityEnd2End

theory CallArityEnd2End
imports ArityTransform CoCallAnalysisImpl
begin

locale CallArityEnd2End
begin

sublocale CoCallAnalysisImpl (proof)

lemma fresh-var-eqE[elim-format]: fresh-var \( e = x \Rightarrow x \notin \text{fv} \ e \)
⟨proof⟩

lemma example1:
  fixes \( f \ g \ x \ y \ z \) :: var
  assumes Aexp-e: \( \forall a. \ Aexp \ e \cdot a = \text{esing} \ x \cdot (\text{up} \cdot a) \sqcup \text{esing} \ y \cdot (\text{up} \cdot a) \)
  assumes CCexp-e: \( \forall a. \ CCexp \ e \cdot a = \bot \)
  assumes simp: transform \( I \) \( e = e \)
  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 \( z \) \overset{*}{\notin} e
  shows \( \text{transform} \ I \) (let \( y \) be \( \text{App} \ (\text{Var} f) \ g \) in \( \text{let} \ x \ be \ e \ in \ (\text{Var} x) \)) =
    let \( y \) be \( \text{Lam} \ [z]. \ \text{App} \ (\text{App} \ (\text{Var} f) \ g) \ z \) in \( \text{let} \ x \ be \ (\text{Lam} \ [z]. \ \text{App} e z) \ in \ (\text{Var} x) \)
⟨proof⟩

end

14.2 CallArityEnd2EndSafe

theory CallArityEnd2EndSafe
imports CallArityEnd2End CardArityTransformSafe CoCallImplSafe CoCallImplTTreeSafe TTreeImplCardinalitySafe
begin

locale CallArityEnd2EndSafe
begin

sublocale CoCallImplSafe ⟨proof⟩
sublocale CallArityEnd2End ⟨proof⟩

abbreviation transform-syn' (\( T_a \)) where \( T_a \equiv \text{transform} \ a \)

lemma end2end:
  \( c \Rightarrow^* c' \Rightarrow \)
  \( \neg \text{boring-step} \ c' \Rightarrow \)
  \( \text{heap-upds-ok-conf} \ c \Rightarrow \)
consistent \((ae, ce, a, as, r) \in\) 
\[\exists ae' ce' a' as' r'.\] consistent \((ae', ce', a', as', r') \in\) 
\[\Rightarrow G^*\] conf-transform \((ae', ce', a', as', r') \in\)

\(\langle\text{proof}\rangle\)

**Theorem end2end-closed:**
assumes closed: \(fv e = (\varnothing) :: \text{var set}\)
assumes \((\varnothing, e, \varnothing) \Rightarrow (\Gamma, v, \varnothing)\) and isVal v
obtains \(\Gamma'\) and \(v'\)
where \((\varnothing, T_0 e, \varnothing) \Rightarrow (\Gamma', v', \varnothing)\) and isVal v'
and card \((\text{domA} \Gamma') \leq \text{card} (\text{domA} \Gamma)\)

\(\langle\text{proof}\rangle\)

**Lemma fresh-var-eqE[dim-format]:** fresh-var \(e = x \implies x \notin fv e\)

\(\langle\text{proof}\rangle\)

**Lemma example1:**
fixes \(e :: \text{exp}\)
fixes \(f, g, x, y, z :: \text{var}\)
assumes \(\text{Aexp-e} :: \lambda a, \text{Aexp e a} = \text{esing} x \uplus \text{esing} y \uplus (\text{up} a)\)
assumes \(\text{ccExp-e} :: \lambda a, \text{ccExp e a} = \bot\)
assumes \([\text{simp}]: \text{transform 1 e = e}\)
assumes isVal e
assumes disj: \(y \neq f \\ y \neq g \\ x \\ y \neq f \neq g \neq y \neq x\)
assumes fresh: \(\text{atom z} \notin e\)
shows transform 1 (let \(y\) be \(\text{App (Var f) g in}\) (let \(x\) be \(e\) in \((\text{Var x})) = \\
\text{let y be (Lam [z]. \text{App (Var f) g)} z in} (\text{let x be (Lam [z]. \text{App e z}) in (Var x)})\)

\(\langle\text{proof}\rangle\)

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

\[eq \theta v v \]
\[\mid (\lambda v. eq n (v1 \downarrow Fn v) (v2 \downarrow Fn v)) \Rightarrow eq (inc\cdot n) v1 v2\]
lemma [simp]: $eq \theta v v' \iff v = v'$

-proof-

lemma $eq$-inc-simp:
$eq (inc \cdot n) v1 v2 \iff (\forall v. eq n (v1 \downarrow Fn v) (v2 \downarrow Fn v))$

-proof-

lemma $eq$-FnI:
$(\forall v. eq (pred \cdot n) (f1 \cdot v) (f2 \cdot v)) \implies eq n (Fn \cdot f1) (Fn \cdot f2)$

-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]: $eq (a \sqcup a') v1 v2 \iff eq a v1 v2 \land eq a' v1 v2$

-proof-

lemma $eq$-adm: $cont f \implies cont g \implies adm (\lambda x. eq a (f x) (g x))$

-proof-

inductive $eq \theta :: AEnv \Rightarrow (var \Rightarrow Value) \Rightarrow (var \Rightarrow Value) \Rightarrow bool$
where
$eq \theta 1$: $(\forall x a. ae x = up \cdot a \implies eq a (\theta 1 x) (\theta 2 x)) \implies eq \theta ae \theta 1 \theta 2$

-proof-

lemma $eq \theta E$: $eq \theta ae \theta 1 \theta 2 \Rightarrow ae x = up \cdot a \Rightarrow eq a (\theta 1 x) (\theta 2 x)$

-proof-

lemma $eq \theta$-refl[simp]: $eq \theta ae \theta \theta$

-proof-

lemma $eq$-esing-up[simp]: $eq \theta (esing x \cdot (up \cdot a)) \theta 1 \theta 2 \iff eq a (\theta 1 x) (\theta 2 x)$

-proof-

lemma $eq$-mono:
assumes $ae \sqsubseteq ae'$
assumes $eq \theta ae \theta 1 \theta 2$
shows $eq \theta ae \theta 1 \theta 2$

-proof-
\textbf{lemma eqg-adm}: \(\text{cont}\ f \implies \text{cont}\ g \implies \text{adm}\ (\lambda\ x.\ \text{eqg} a\ (f\ x)\ (g\ x))\)

\textit{(proof)}

\textbf{lemma up-join-eq-up\ [simp]}: \(\up\cdot (n::\text{'a::Finite-Join-cpo}) \sqcup \up\cdot n' = \up\cdot (n \sqcup n')\)

\textit{(proof)}

\textbf{lemma eqg-join\ [simp]}: \(\text{eqg}\ \text{(ae} \sqcup \text{ae'}) \text{g1} \text{g2} \iff \text{eqg}\ \text{ae}\ \text{g1} \text{g2} \land \text{eqg}\ \text{ae'} \text{g1} \text{g2}\)

\textit{(proof)}

\textbf{lemma eqg-override\ [simp]}: \(\text{eqg}\ \text{ae}\ (\text{g1}+++\text{g2}) (\text{g1'}+++\text{g2'}) \iff \text{eqg}\ \text{ae} (\text{g1} f\mid (-S)) (\text{g1'} f\mid (-S)) \land \text{eqg}\ \text{ae}\ (\text{g2} f\mid S) (\text{g2'} f\mid S)\)

\textit{(proof)}

\textbf{lemma Aexp-heap-below-Aheap}:
\begin{itemize}
  \item \textit{assumes} \(\text{Aheap}\ \Gamma\ e\cdot a\ x = \up\cdot a'\)
  \item \textit{assumes} \(\text{map-of}\ \Gamma\ x = \text{Some}\ e'\)
  \item \textit{shows} \(\text{Aexp}\ e'\cdot a' \sqsubseteq \text{Aheap}\ \Gamma\ e\cdot a \sqcup \text{Aexp}\ (\text{Let}\ \Gamma\ e)\cdot a\)
\end{itemize}

\textit{(proof)}

\textbf{lemma Aexp-body-below-Aheap}:
\begin{itemize}
  \item \textit{shows} \(\text{Aexp}\ e\cdot a \sqsubseteq \text{Aheap}\ \Gamma\ e\cdot a \sqcup \text{Aexp}\ (\text{Let}\ \Gamma\ e)\cdot a\)
\end{itemize}

\textit{(proof)}

\textbf{lemma Aexp-correct}:
\begin{itemize}
  \item \(\text{eqg}\ (\text{Aexp}\ e\cdot a) \text{g1} \text{g2} \implies \text{eq}\ a\ ([\text{]} \cdot \text{g1} ) (\text{[} \cdot \text{g2} )\)
\end{itemize}

\textit{(proof)}

\textbf{lemma ESem-ignores-fresh\ [simp]}: \(\text{[} e \cdot g(\text{fresh-var} e := v) = \text{[} e \cdot g\)

\textit{(proof)}

\textbf{lemma eq-\text{Acta-expand}}: \(\text{eq}\ a\ ([\text{Acta-expand}\ a\ e \cdot g ) (\text{[} e \cdot g\)

\textit{(proof)}

\textbf{lemma Arity-transformation-correct}:
\begin{itemize}
  \item \(\text{eq}\ a\ ([\text{}\Gamma a\ e \cdot g ) (\text{[} e \cdot g\)
\end{itemize}

\textit{(proof)}

\textbf{corollary Arity-transformation-correct'}:
\begin{itemize}
  \item \(\text{[} \text{\Gamma g e \cdot g ) = \text{[} e \cdot g\)
\end{itemize}

\textit{(proof)}

end

end