The Safety of Call Arity

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

March 17, 2025

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

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

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

Artifact correspondence table

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

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

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

References

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

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

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

[Huf12] Brian Huffman, *HOLCF '11: A definitional domain theory for verifying functional programs*, Ph.D. thesis, Portland State University, 2012.

[Lau93] John Launchbury, *A natural semantics for lazy evaluation*, POPL '93, 1993, pp. 144–154.

[Ses97] Peter Sestoft, *Deriving a lazy abstract machine*, Journal of Functional Programming **7** (1997), 231–264.

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

Contents

1 Various Utilities

1.1 ConstOn

theory *ConstOn*
imports *Main*
begin

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

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

lemma *const-onI*[*intro*]: (⋀*y*. *y* ∈ *S* ⟹ *f y* = *x*) ⟹ *const-on f S x*
⟨*proof*⟩

lemma *const-onD*[*dest*]: *const-on f S x* ⟹ *y* ∈ *S* ⟹ *f y* = *x*
⟨*proof*⟩

lemma *const-on-insert*[*simp*]: *const-on f* (*insert x S*) *y* ⟷ *const-on f S y* ∧ *f x* = *y*
⟨*proof*⟩

lemma *const-on-union*[*simp*]: *const-on f* (*S* ∪ *S'*) *y* ⟷ *const-on f S y* ∧ *const-on f S' y*
⟨*proof*⟩

lemma *const-on-subset*[*elim*]: *const-on f S y* ⟹ *S'* ⊆ *S* ⟹ *const-on f S' y*
⟨*proof*⟩

end

1.2 Set-Cpo

theory *Set−Cpo*
imports *HOLCF*
begin

default-sort *type*

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

instance *set* :: (*type*) *po*
⟨*proof*⟩

lemma *is-lub-set*:
S <<| ⋃*S*
⟨*proof*⟩

lemma *lub-set*: *lub S* = ⋃*S*
⟨*proof*⟩

instance *set* :: (*type*) *cpo*
⟨*proof*⟩

lemma *minimal-set*: $\{\} \sqsubseteq S$
⟨*proof*⟩

instance *set* :: (*type*) *pcpo*
⟨*proof*⟩

lemma *set-contI*:
assumes $\bigwedge Y.\ chain\ Y \Longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = \bigcup\ (f\ `\ range\ Y)$
shows *cont f*
⟨*proof*⟩

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

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

lemma *adm-Ball*[*simp*]: *adm* $(\lambda S.\ \forall x{\in}S.\ P\ x)$
⟨*proof*⟩

lemma *finite-subset-chain*:
fixes $Y :: nat \Rightarrow 'a\ set$
assumes *chain Y*
assumes $S \subseteq \bigcup(Y\ `\ UNIV)$
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* $(\lambda S.\ S - S')$
⟨*proof*⟩

end

1.3 Env-Set-Cpo

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

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

end

1.4 AList-Utils-HOLCF

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

syntax
-BLubMap :: [*pttrn*, *pttrn*, *'a ⇀ 'b*, *'b*] ⇒ *'b* (‹(*3*⨆/-/↦/-/∈/-/./ -)› [*0*,*0*,*0*, *10*] *10*)

syntax-consts
-BLubMap == *lub*

translations
⨆ *k↦v∈m*. *e* == *CONST lub* (*CONST mapCollect* (λ*k v* . *e*) *m*)

lemma *below-lubmapI*[*intro*]:
m k = *Some v* ⟹ (*e k v*::*'a*::*Join-cpo*) ⊑ (⨆ *k↦v∈m*. *e k v*)
⟨*proof*⟩

lemma *lubmap-belowI*[*intro*]:
(⋀ *k v* . *m k* = *Some v* ⟹ (*e k v*::*'a*::*Join-cpo*) ⊑ *u*) ⟹ (⨆ *k↦v∈m*. *e k v*) ⊑ *u*
⟨*proof*⟩

lemma *lubmap-const-bottom*[*simp*]:
(⨆ *k↦v∈m*. ⊥) = (⊥::*'a*::*Join-cpo*)
⟨*proof*⟩

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

lemma *lubmap-below-cong*:
assumes ⋀ *k v*. *m k* = *Some v* ⟹ *f1 k v* ⊑ (*f2 k v* :: *'a* :: *Join-cpo*)
shows (⨆ *k↦v∈m*. *f1 k v*) ⊑ (⨆ *k↦v∈m*. *f2 k v*)
⟨*proof*⟩

lemma *cont2cont-lubmap*[*simp*, *cont2cont*]:
assumes (⋀*k v* . *cont* (*f k v*))
shows *cont* (λ*x*. ⨆ *k↦v∈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* (*x*#*ys*) (*x*#*zs*)

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

lemma *interleave-comm*: *xs* ∈ *ys* ⊗ *zs* ⟹ *xs* ∈ *zs* ⊗ *ys*
  ⟨*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*

⟨*proof*⟩

lemma *interleave-tl*: $xs \in ys \otimes zs \Longrightarrow tl\ xs \in tl\ ys \otimes zs \vee tl\ xs \in ys \otimes (tl\ zs)$
⟨*proof*⟩

lemma *interleave-butlast*: $xs \in ys \otimes zs \Longrightarrow butlast\ xs \in butlast\ ys \otimes zs \vee butlast\ xs \in ys \otimes (butlast\ zs)$
⟨*proof*⟩

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

lemma *filter-interleave*: $xs \in ys \otimes zs \Longrightarrow filter\ P\ xs \in filter\ P\ ys \otimes filter\ P\ zs$
⟨*proof*⟩

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

function *foo* **where**
$foo\ []\ [] = undefined$
$|\ foo\ xs\ [] = undefined$
$|\ foo\ []\ ys = undefined$
$|\ foo\ (x\#xs)\ (y\#ys) = undefined\ (foo\ xs\ (y\#ys))\ (foo\ (x\#xs)\ ys)$
⟨*proof*⟩
termination ⟨*proof*⟩
lemmas $list\text{-}induct2'' = foo.induct[case\text{-}names\ NilNil\ ConsNil\ NilCons\ ConsCons]$

lemma *interleave-filter*:
assumes $xs \in filter\ P\ ys \otimes filter\ P\ zs$
obtains xs' **where** $xs' \in ys \otimes zs$ **and** $xs = filter\ P\ xs'$
⟨*proof*⟩

end

2 Small-step Semantics

2.1 SestoftConf

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

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

instantiation *stack-elem* :: *pt*

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

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

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

instance *stack-elem* :: *fs*
$\langle proof \rangle$

type-synonym *stack* = *stack-elem list*

fun *ap* :: *stack* ⇒ *var set* **where**
ap [] = {}
| *ap* (*Alts e1 e2* # *S*) = *ap S*
| *ap* (*Arg x* # *S*) = *insert x* (*ap S*)
| *ap* (*Upd x* # *S*) = *ap S*
| *ap* (*Dummy x* # *S*) = *ap S*
fun *upds* :: *stack* ⇒ *var set* **where**
upds [] = {}
| *upds* (*Alts e1 e2* # *S*) = *upds S*
| *upds* (*Upd x* # *S*) = *insert x* (*upds S*)
| *upds* (*Arg x* # *S*) = *upds S*
| *upds* (*Dummy x* # *S*) = *upds S*
fun *dummies* :: *stack* ⇒ *var set* **where**
dummies [] = {}
| *dummies* (*Alts e1 e2* # *S*) = *dummies S*
| *dummies* (*Upd x* # *S*) = *dummies S*
| *dummies* (*Arg x* # *S*) = *dummies S*

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

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

lemma *ups-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* (*Alts e1 e2* # *S*) = *Alts e1 e2* # *restr-stack V S*
| *restr-stack V* (*Arg x* # *S*) = *Arg x* # *restr-stack V S*
| *restr-stack V* (*Upd x* # *S*) = (*if x* ∈ *V then Upd x* # *restr-stack V S else restr-stack V S*)
| *restr-stack V* (*Dummy x* # *S*) = *Dummy x* # *restr-stack V S*

lemma *restr-stack-cong*:
(⋀ *x*. *x* ∈ *upds S* ⟹ *x* ∈ *V* ⟷ *x* ∈ *V'*) ⟹ *restr-stack V S* = *restr-stack V' S*
⟨*proof*⟩

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

lemma *fresh-star-restict-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-stackD*:
assumes *Upd x* # *S* = *restr-stack V S'*
shows *x* ∈ *V*
⟨*proof*⟩
lemma *Upd-eq-restr-stackD2*:
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.intros*[*case-names step*]

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

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

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

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

lemma *rtranclp-invariant-induct*[*consumes 3*, *case-names base step*]:
  assumes *r*** *a b*
  assumes *invariant r I*
  assumes *I a*
  assumes *P a*
  assumes (⋀*y z*. *r*** *a y* ⟹ *r y z* ⟹ *I y* ⟹ *I z* ⟹ *P y* ⟹ *P z*)
  shows *P b*
⟨*proof*⟩

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

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

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

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

lemma *heap-upds-ok-Nil*[*simp*]: *heap-upds-ok* (Γ, []) ⟨*proof*⟩
lemma *heap-upds-ok-app1*: *heap-upds-ok* (Γ, *S*) ⟹ *heap-upds-ok* (Γ,*Arg x* # *S*) ⟨*proof*⟩
lemma *heap-upds-ok-app2*: *heap-upds-ok* (Γ, *Arg x* # *S*) ⟹ *heap-upds-ok* (Γ, *S*) ⟨*proof*⟩
lemma *heap-upds-ok-alts1*: *heap-upds-ok* (Γ, *S*) ⟹ *heap-upds-ok* (Γ,*Alts e1 e2* # *S*) ⟨*proof*⟩
lemma *heap-upds-ok-alts2*: *heap-upds-ok* (Γ, *Alts e1 e2* # *S*) ⟹ *heap-upds-ok* (Γ, *S*) ⟨*proof*⟩

lemma *heap-upds-ok-append*:
assumes *domA* $\Delta \cap$ *upds* $S = \{\}$
assumes *heap-upds-ok* (Γ,S)
shows *heap-upds-ok* (Δ@Γ, S)
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *heap-upds-ok-upd*:
heap-upds-ok (Γ, *Upd* x # S) $\Longrightarrow$ $x \notin$ *domA* $\Gamma \wedge x \notin$ *upds* S
⟨*proof*⟩

lemmas *heap-upds-ok-intros*[*intro*] =
heap-upds-ok-to-heap heap-upds-ok-to-stack heap-upds-ok-to-stack′ heap-upds-ok-reorder
heap-upds-ok-app1 heap-upds-ok-app2 heap-upds-ok-alts1 heap-upds-ok-alts2 heap-upds-ok-delete

heap-upds-ok-restrictA heap-upds-ok-restr-stack
heap-upds-ok-let
lemmas *heap-upds-ok.simps*[*simp del*]

end

2.2 Sestoft

theory *Sestoft*
imports *SestoftConf*
begin

inductive *step* :: *conf* ⇒ *conf* ⇒ *bool* (**infix** ‹⇒› *50*) **where**
  app_1: (Γ, *App e x*, *S*) ⇒ (Γ, *e* , *Arg x* # *S*)
| app_2: (Γ, *Lam* [*y*]. *e*, *Arg x* # *S*) ⇒ (Γ, *e*[*y* ::= *x*] , *S*)
| var_1: *map-of* Γ *x* = *Some e* ⟹ (Γ, *Var x*, *S*) ⇒ (*delete x* Γ, *e* , *Upd x* # *S*)
| var_2: *x* ∉ *domA* Γ ⟹ *isVal e* ⟹ (Γ, *e*, *Upd x* # *S*) ⇒ ((*x*,*e*)# Γ, *e* , *S*)
| let_1: *atom* ‘ *domA* Δ ♯* Γ ⟹ *atom* ‘ *domA* Δ ♯* *S*
                ⟹ (Γ, *Let* Δ *e*, *S*) ⇒ (Δ@Γ, *e* , *S*)
| if_1: (Γ, *scrut ? e1 : e2*, *S*) ⇒ (Γ, *scrut*, *Alts e1 e2* # *S*)
| if_2: (Γ, *Bool b*, *Alts e1 e2* # *S*) ⇒ (Γ, *if b then e1 else e2*, *S*)

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

lemma *SmartLet-stepI*:
  atom ‘ *domA* Δ ♯* Γ ⟹ *atom* ‘ *domA* Δ ♯* *S* ⟹ (Γ, *SmartLet* Δ *e*, *S*) ⇒* (Δ@Γ, *e* , *S*)
⟨*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 let_1*-closed*:
  assumes *closed* (Γ, *Let* Δ *e*, *S*)
  assumes *domA* Δ ∩ *domA* Γ = {}
  assumes *domA* Δ ∩ *upds S* = {}
  shows (Γ, *Let* Δ *e*, *S*) ⇒ (Δ@Γ, *e* , *S*)
⟨*proof*⟩

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

lemma *step-invariant-induction*[*consumes 4*, *case-names* app_1 app_2 *thunk lamvar* var_2 let_1 if_1 if_2 *refl trans*]:
  assumes *c* ⇒* *c′*
  assumes ¬ *boring-step c′*
  assumes *invariant* (⇒) *I*
  assumes *I c*

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

lemma *step-induction*[*consumes 2*, *case-names* app_1 app_2 *thunk lamvar* var_2 let_1 if_1 if_2 *refl trans*]:
assumes $c \Rightarrow^*\ c'$
assumes ¬ *boring-step* c'
assumes app_1: $\bigwedge \Gamma\ e\ x\ S\ .\ P\ (\Gamma,\ App\ e\ x,\ S)\ \ (\Gamma,\ e\ ,\ Arg\ x\ \#\ S)$
assumes app_2: $\bigwedge \Gamma\ y\ e\ x\ S\ .\ P\ (\Gamma,\ Lam\ [y].\ e,\ Arg\ x\ \#\ S)\ (\Gamma,\ e[y ::= x]\ ,\ S)$
assumes *thunk*: $\bigwedge \Gamma\ x\ e\ S\ .\ map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow \neg\ isVal\ e \Longrightarrow P\ (\Gamma,\ Var\ x,\ S)\ (delete\ x\ \Gamma,\ e\ ,\ Upd\ x\ \#\ S)$
assumes *lamvar*: $\bigwedge \Gamma\ x\ e\ S\ .\ map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow isVal\ e \Longrightarrow P\ (\Gamma,\ Var\ x,\ S)\ ((x,e)\ \#\ delete\ x\ \Gamma,\ e\ ,\ S)$
assumes var_2: $\bigwedge \Gamma\ x\ e\ S\ .\ x \notin domA\ \Gamma \Longrightarrow isVal\ e \Longrightarrow P\ (\Gamma,\ e,\ Upd\ x\ \#\ S)\ ((x,e)\#\ \Gamma,\ e\ ,\ S)$
assumes let_1: $\bigwedge \Delta\ \Gamma\ e\ S\ .\ atom\ `\ domA\ \Delta\ \sharp*\ \Gamma \Longrightarrow atom\ `\ domA\ \Delta\ \sharp*\ S \Longrightarrow P\ (\Gamma,\ Let\ \Delta\ e,\ S)\ (\Delta@\Gamma,\ e,\ S)$
assumes if_1: $\bigwedge\Gamma\ scrut\ e1\ e2\ S.\ P\ (\Gamma,\ scrut\ ?\ e1\ :\ e2,\ S)\ (\Gamma,\ scrut,\ Alts\ e1\ e2\ \#\ S)$
assumes if_2: $\bigwedge\Gamma\ b\ e1\ e2\ S.\ P\ (\Gamma,\ Bool\ b,\ Alts\ e1\ e2\ \#\ S)\ (\Gamma,\ if\ b\ then\ e1\ else\ e2,\ S)$
assumes *refl*: $\bigwedge\ c.\ P\ c\ c$
assumes *trans*[*trans*]: $\bigwedge\ c\ c'\ c''.\ c \Rightarrow^*\ c' \Longrightarrow c' \Rightarrow^*\ c'' \Longrightarrow P\ c\ c' \Longrightarrow P\ c'\ c'' \Longrightarrow P\ c\ c''$
shows $P\ c\ c'$
⟨*proof*⟩

2.2.1 Equivariance

lemma *step-eqvt*[*eqvt*]: $step\ x\ y \Longrightarrow step\ (\pi \cdot x)\ (\pi \cdot y)$
⟨*proof*⟩

2.2.2 Invariants

lemma *closed-invariant*:

invariant step closed
⟨*proof*⟩

lemma *heap-upds-ok-invariant*:
invariant step heap-upds-ok-conf
⟨*proof*⟩

end

2.3 SestoftGC

theory *SestoftGC*
imports *Sestoft*
begin

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

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

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

lemma *var-onceI*:
assumes *map-of* Γ *x* = *Some e*
shows $(\Gamma, Var\ x, S) \Rightarrow_G^* (delete\ x\ \Gamma, e\ , S@[Dummy\ x])$
⟨*proof*⟩

lemma *normal-trans*: $c \Rightarrow^* c' \Longrightarrow c \Rightarrow_G^* c'$
⟨*proof*⟩

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

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*)

⟨*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 *sestoftUnGCStack*:
assumes *heap-upds-ok* (Γ, *S*)
obtains Γ′ *S*′ **where**
(Γ, *e*, *S*) ⇒* (Γ′, *e*, *S*′)
to-gc-conf r (Γ, *e*, *S*) = *to-gc-conf r* (Γ′, *e*, *S*′)
¬ *isVal e* ∨ *safe-hd S*′ = *safe-hd* (*restr-stack* (− *set r*) *S*′)
⟨*proof*⟩

lemma *perm-exI-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-upd-ok-to-gc-conf*:
heap-upds-ok (Γ, *S*) ⟹ *to-gc-conf r* (Γ, *e*, *S*) = (Γ″, *e*″, *S*″) ⟹ *heap-upds-ok* (Γ″, *S*″)
⟨*proof*⟩

lemma *delete-restrictA-conv*:
delete x Γ = *restrictA* (−{*x*}) Γ

⟨*proof*⟩

lemma *sestoftUnGCstep*:
assumes *to-gc-conf r c* $\Rightarrow_G$ *d*
assumes *heap-upds-ok-conf c*
assumes *closed c*
and *r-ok r c*
shows $\exists\ r'\ c'.\ c \Rightarrow^*\ c' \wedge\ d =$ *to-gc-conf r' c'* $\wedge$ *r-ok r' c'*
⟨*proof*⟩

lemma *sestoftUnGC*:
assumes (*to-gc-conf r c*) $\Rightarrow_G{}^*$ *d* **and** *heap-upds-ok-conf c* **and** *closed c* **and** *r-ok r c*
shows $\exists\ r'\ c'.\ c \Rightarrow^*\ c' \wedge\ d =$ *to-gc-conf r' c'* $\wedge$ *r-ok r' c'*
⟨*proof*⟩

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

lemma *sestoftUnGC'*:
assumes ([], *e*, []) $\Rightarrow_G{}^*$ (Γ, *e'*, *map Dummy r*)
assumes *isVal e'*
assumes *fv e* = ({}::*var set*)
shows $\exists\ \Gamma''.$ ([], *e*, []) $\Rightarrow^*$ (Γ'', *e'*, []) $\wedge\ \Gamma =$ *restrictA* (− *set r*) Γ'' $\wedge$ *set r* $\subseteq$ *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*]:

trace x1 x2 final $\Longrightarrow$ *P final* [] *final* $\Longrightarrow$ ($\bigwedge$*conf' T conf. trace conf' T final* $\Longrightarrow$ *P conf' T final* $\Longrightarrow$ *conf* $\Rightarrow$ *conf'* $\Longrightarrow$ *P conf* (*conf'* # *T*) *final*) $\Longrightarrow$ *P x1 x2 final*
⟨*proof*⟩

lemma *build-trace*:
c $\Rightarrow^*$ *c'* $\Longrightarrow$ $\exists$ *T. trace c T c'*
⟨*proof*⟩

lemma *destruct-trace*: *trace c T c'* $\Longrightarrow$ *c* $\Rightarrow^*$ *c'*
⟨*proof*⟩

lemma *traceWhile*:
assumes *trace* c_1 *T* c_4
assumes *P* c_1
assumes $\neg$ *P* c_4
obtains T_1 c_2 c_3 T_2
where *T* = T_1 @ c_3 # T_2 **and** *trace* c_1 T_1 c_2 **and** $\forall$ *x*$\in$*set* T_1. *P x* **and** *P* c_2 **and** c_2 $\Rightarrow$ c_3 **and** $\neg$ *P* c_3 **and** *trace* c_3 T_2 c_4
⟨*proof*⟩

lemma *traces-list-all*:
trace c T c' $\Longrightarrow$ *P c'* $\Longrightarrow$ ($\bigwedge$ *c c'. c* $\Rightarrow$ *c'* $\Longrightarrow$ *P c'* $\Longrightarrow$ *P c*) $\Longrightarrow$ ($\forall$ *x*$\in$*set T. P x*) $\wedge$ *P c*
⟨*proof*⟩

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

end

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

lemma *extends-refl*[*simp*]: *S* $\lesssim$ *S* ⟨*proof*⟩
lemma *extends-cons*[*simp*]: *S* $\lesssim$ *x* # *S* ⟨*proof*⟩
lemma *extends-append*[*simp*]: *S* $\lesssim$ *L* @ *S* ⟨*proof*⟩
lemma *extends-not-cons*[*simp*]: $\neg$ (*x* # *S*) $\lesssim$ *S* ⟨*proof*⟩
lemma *extends-trans*[*trans*]: *S* $\lesssim$ *S'* $\Longrightarrow$ *S'* $\lesssim$ *S''* $\Longrightarrow$ *S* $\lesssim$ *S''* ⟨*proof*⟩

locale *balance-trace* = *traces* +
fixes *stack* :: *'a* $\Rightarrow$ *'s list*
assumes *one-step-only*: *c* $\Rightarrow$ *c'* $\Longrightarrow$ (*stack c*) = (*stack c'*) $\vee$ ($\exists$ *x. stack c'* = *x* # *stack c*) $\vee$ ($\exists$ *x. stack c* = *x* # *stack c'*)
begin

inductive *bal* :: *'a* $\Rightarrow$ *'a list* $\Rightarrow$ *'a* $\Rightarrow$ *bool* **where**
balI[*intro*]: *trace c T c'* $\Longrightarrow$ $\forall$ *c'*$\in$ *set T. stack c* $\lesssim$ *stack c'* $\Longrightarrow$ *stack c'* = *stack c* $\Longrightarrow$ *bal c T c'*

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

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

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

lemma *stack-passes-lower-bound*:
assumes $c_3 \Rightarrow c_4$
assumes *stack* $c_2 \lesssim$ *stack* c_3
assumes ¬ *stack* $c_2 \lesssim$ *stack* c_4
shows *stack* c_3 = *stack* c_2 **and** *stack* c_4 = *tl* (*stack* c_2)
⟨*proof*⟩

lemma *bal-consE*:
assumes *bal* c_1 (c_2 # *T*) c_5
and c_2: *stack* c_2 = *s* # *stack* c_1
obtains T_1 c_3 c_4 T_2
where *T* = T_1 @ c_4 # T_2 **and** *bal* c_2 T_1 c_3 **and** $c_3 \Rightarrow c_4$ *bal* c_4 T_2 c_5
⟨*proof*⟩

end

end

2.5 SestoftCorrect

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

lemma *lemma-2*:
assumes Γ : *e* $\Downarrow_L$ Δ : *z*
and *fv* (Γ, *e*, *S*) ⊆ *set L* ∪ *domA* Γ
shows (Γ, *e*, *S*) $\Rightarrow^*$ (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*]:

$(\Gamma,\ e,\ S) \Rightarrow^*{}_T\ final \Longrightarrow (\bigwedge \Gamma\ e\ S.\ final = (\Gamma,\ e,\ S) \Longrightarrow P\ \Gamma\ e\ S\ []\ (\Gamma,\ e,\ S)) \Longrightarrow (\bigwedge \Gamma\ e\ S\ T\ \Gamma'\ e'\ S'.\ (\Gamma',\ e',\ S') \Rightarrow^*{}_T\ final \Longrightarrow P\ \Gamma'\ e'\ S'\ T\ final \Longrightarrow (\Gamma,\ e,\ S) \Rightarrow (\Gamma',\ e',\ S') \Longrightarrow P\ \Gamma\ e\ S\ ((\Gamma',\ e',\ S')\ \#\ T)\ final) \Longrightarrow P\ \Gamma\ e\ S\ T\ final$
⟨*proof*⟩

interpretation *balance-trace step stack*
⟨*proof*⟩

abbreviation *bal-syn* (‹- $\Rightarrow^{b*}$_ -› [50,50,50] 50) **where** *bal-syn* ≡ *bal*

lemma *isVal-stops*:
assumes *isVal e*
assumes $(\Gamma,\ e,\ S) \Rightarrow^{b*}{}_T\ (\Delta,\ z,\ S)$
shows $T=[]$
⟨*proof*⟩

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

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

lemma *dummy-stack-extended*:
$set\ S \subseteq\ Dummy\ `\ UNIV \Longrightarrow x \notin Dummy\ `\ UNIV \Longrightarrow (S \lesssim x\ \#\ S') \longleftrightarrow\ S \lesssim S'$
⟨*proof*⟩

lemma[*simp*]: *Arg x* ∉ *range Dummy Upd x* ∉ *range Dummy Alts* $e_1\ e_2$ ∉ *range Dummy*
⟨*proof*⟩

lemma *dummy-stack-balanced*:
assumes *set S* ⊆ *Dummy ` UNIV*
assumes $(\Gamma,\ e,\ S) \Rightarrow^*\ (\Delta,\ z,\ S)$
obtains *T* **where** $(\Gamma,\ e,\ S) \Rightarrow^{b*}{}_T\ (\Delta,\ z,\ S)$
⟨*proof*⟩

end

3 Arity

3.1 Arity

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

begin

typedef *Arity* = *UNIV* :: *nat set*
  morphisms *Rep-Arity to-Arity* ⟨*proof*⟩

setup-lifting *type-definition-Arity*

instantiation *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*$_\perp$) ⊔ *up*·*0* = *up*·*0*
⟨*proof*⟩

lemma *Arity-up-zero-join2*[*simp*]: $up{\cdot}0 \sqcup (x :: Arity_\perp) = up{\cdot}0$
⟨*proof*⟩
lemma *up-zero-top*[*simp*]: $x \sqsubseteq up{\cdot}(0{::}Arity)$
⟨*proof*⟩
lemma *Arity-above-up-top*[*simp*]: $up{\cdot}0 \sqsubseteq (a :: Arity_\perp) \longleftrightarrow a = up{\cdot}0$
⟨*proof*⟩

lemma *Arity-exhaust*: $(y = 0 \Longrightarrow P) \Longrightarrow (\bigwedge x.\ y = inc \cdot x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

end

3.2 AEnv

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

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

end

3.3 Arity-Nominal

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

lemma *join-eqvt*[*eqvt*]: $\pi \cdot (x \sqcup (y :: 'a :: \{Finite\text{-}Join\text{-}cpo,\ cont\text{-}pt\})) = (\pi \cdot x) \sqcup (\pi \cdot y)$
⟨*proof*⟩

instantiation *Arity* :: *pure*
begin
definition $p \cdot (a{::}Arity) = a$
instance
⟨*proof*⟩
end

instance *Arity* :: *cont-pt* ⟨*proof*⟩
instance *Arity* :: *pure-cont-pt* ⟨*proof*⟩

end

3.4 AritySt ack

theory *AritySt ack*
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*]:

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

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

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

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

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

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

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

end

4.2 EtaExpansionSafe

theory *EtaExpansionSafe*
imports *EtaExpansion Sestoft*
begin

theorem *eta-expansion-safe*:
  assumes *set T* ⊆ *range Arg*
  shows (Γ, *eta-expand* (*length T*) *e*, *T*@*S*) ⇒* (Γ, *e*, *T*@*S*)
⟨*proof*⟩

fun *arg-prefix* :: *stack* ⇒ *nat* **where**
  arg-prefix [] = *0*
| *arg-prefix* (*Arg x* # *S*) = *Suc* (*arg-prefix S*)
| *arg-prefix* (*Alts e1 e2* # *S*) = *0*
| *arg-prefix* (*Upd x* # *S*) = *0*
| *arg-prefix* (*Dummy x* # *S*) = *0*

theorem *eta-expansion-safe′*:
  assumes *n* ≤ *arg-prefix S*
  shows (Γ, *eta-expand n e*, *S*) ⇒* (Γ, *e*, *S*)

⟨*proof*⟩

end

4.3 TransformTools

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

default-sort *type*

fun *lift-transform* :: (′*a*::*cont-pt* ⇒ *exp* ⇒ *exp*) ⇒ (′$a_\perp$ ⇒ *exp* ⇒ *exp*)
  where *lift-transform t Ibottom e* = *e*
  | *lift-transform t* (*Iup a*) *e* = *t a e*

lemma *lift-transform-simps*[*simp*]:
  lift-transform t ⊥ *e* = *e*
  lift-transform t (*up*·*a*) *e* = *t a e*
  ⟨*proof*⟩

lemma *lift-transform-eqvt*[*eqvt*]: 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* ⇒ ′$a_\perp$) ⇒ *heap* ⇒ *heap*
  where *map-transform t ae* = *map-ran* (λ *x e* . *lift-transform t* (*ae x*) *e*)

lemma *map-transform-eqvt*[*eqvt*]: π · *map-transform t ae* = *map-transform* (π · *t*) (π · *ae*)
  ⟨*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 ⟹ supp (t a e) ⊆ supp e*
shows *supp (map-transform t ae Γ) ⊆ 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* =
fixes *trans* :: *'a::cont-pt ⇒ exp ⇒ exp*

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

lemma *supp-map-transform*: $supp\ (map\text{-}transform\ trans\ ae\ \Gamma) \subseteq supp\ \Gamma$
⟨*proof*⟩

lemma *fresh-transform*[*intro*]: $a \sharp e \Longrightarrow a \sharp trans\ n\ e$
⟨*proof*⟩

lemma *fresh-star-transform*[*intro*]: $a \sharp* e \Longrightarrow a \sharp* trans\ n\ e$
⟨*proof*⟩

lemma *fresh-map-transform*[*intro*]: $a \sharp \Gamma \Longrightarrow a \sharp map\text{-}transform\ trans\ ae\ \Gamma$
⟨*proof*⟩

lemma *fresh-star-map-transform*[*intro*]: $a \sharp* \Gamma \Longrightarrow a \sharp* map\text{-}transform\ trans\ ae\ \Gamma$
⟨*proof*⟩
end

end

4.4 ArityEtaExpansion

theory *ArityEtaExpansion*
imports *EtaExpansion Arity−Nominal TransformTools*
begin

lift-definition *Aeta-expand* :: *Arity* ⇒ *exp* ⇒ *exp* **is** *eta-expand*⟨*proof*⟩

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

lemma *Aeta-expand-0*[*simp*]: $Aeta\text{-}expand\ 0\ e = e$
⟨*proof*⟩

lemma *Aeta-expand-inc*[*simp*]: $Aeta\text{-}expand\ (inc{\cdot}n)\ e = (Lam\ [fresh\text{-}var\ e].\ Aeta\text{-}expand\ n\ (App\ e\ (fresh\text{-}var\ e)))$
⟨*proof*⟩

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

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

lemma *isVal-Aeta-expand*: *isVal e* $\Longrightarrow$ *isVal* (*Aeta-expand a e*)
⟨*proof*⟩

lemma *Aeta-expand-fresh*[*simp*]: *a* ♯ *Aeta-expand n e* = *a* ♯ *e* ⟨*proof*⟩
lemma *Aeta-expand-fresh-star*[*simp*]: *a* ♯* *Aeta-expand n e* = *a* ♯* *e* ⟨*proof*⟩

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

end

4.5 ArityEtaExpansionSafe

theory *ArityEtaExpansionSafe*
imports *EtaExpansionSafe ArityStack ArityEtaExpansion*
begin

lemma *Aeta-expand-safe*:
assumes *Astack S* ⊑ *a*
shows (Γ, *Aeta-expand a e*, *S*) ⇒* (Γ, *e*, *S*)
⟨*proof*⟩

end

5 Arity Analysis

5.1 ArityAnalysisSig

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

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

end

locale *ArityAnalysisHeap* =
fixes *Aheap* :: *heap* ⇒ *exp* ⇒ *Arity* → *AEnv*

locale *EdomArityAnalysis* = *ArityAnalysis* +
assumes *Aexp-edom*: *edom* ($\mathcal{A}_a$ *e*) ⊆ *fv e*

begin

lemma *fup-Aexp-edom*: *edom* ($\mathcal{A}^{\perp}{}_a$ *e*) $\subseteq$ *fv e*
⟨*proof*⟩

lemma *Aexp-fresh-bot*[*simp*]: **assumes** *atom v* $\sharp$ *e* **shows** $\mathcal{A}_a$ *e v* = $\bot$
⟨*proof*⟩
end

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

end

5.2 ArityAnalysisAbinds

theory *ArityAnalysisAbinds*
imports *ArityAnalysisSig*
begin

context *ArityAnalysis*
begin

5.2.1 Lifting arity analysis to recursive groups

definition *ABind* :: *var* ⇒ *exp* ⇒ (*AEnv* → *AEnv*)
where *ABind v e* = (Λ *ae*. *fup*·(*Aexp e*)·(*ae v*))

lemma *ABind-eq*[*simp*]: *ABind v e* · *ae* = $\mathcal{A}^{\perp}{}_{ae}$ *v e*
⟨*proof*⟩

fun *ABinds* :: *heap* ⇒ (*AEnv* → *AEnv*)
where *ABinds* [] = $\bot$
| *ABinds* ((*v*,*e*)#*binds*) = *ABind v e* ⊔ *ABinds* (*delete v binds*)

lemma *ABinds-strict*[*simp*]: *ABinds* Γ·$\bot$=$\bot$
⟨*proof*⟩

lemma *Abinds-reorder1*: *map-of* Γ *v* = *Some e* ⟹ *ABinds* Γ = *ABind v e* ⊔ *ABinds* (*delete v* Γ)
⟨*proof*⟩

lemma *ABind-below-ABinds*: *map-of* Γ *v* = *Some e* ⟹ *ABind v e* ⊑ *ABinds* Γ
⟨*proof*⟩

lemma *Abinds-reorder*: *map-of* Γ = *map-of* Δ ⟹ *ABinds* Γ = *ABinds* Δ
⟨*proof*⟩

lemma *Abinds-env-cong*: $(\bigwedge x.\ x{\in}domA\ \Delta \Longrightarrow ae\ x = ae'\ x) \Longrightarrow ABinds\ \Delta{\cdot}ae = ABinds\ \Delta{\cdot}ae'$
⟨*proof*⟩

lemma *Abinds-env-restr-cong*: $ae\ f|`\ domA\ \Delta = ae'\ f|`\ domA\ \Delta \Longrightarrow ABinds\ \Delta{\cdot}ae = ABinds\ \Delta{\cdot}ae'$
⟨*proof*⟩

lemma *ABinds-env-restr*[*simp*]: $ABinds\ \Delta{\cdot}(ae\ f|`\ domA\ \Delta) = ABinds\ \Delta{\cdot}ae$
⟨*proof*⟩

lemma *Abinds-join-fresh*: $ae'\ `\ (domA\ \Delta) \subseteq \{\bot\} \Longrightarrow ABinds\ \Delta{\cdot}(ae \sqcup ae') = (ABinds\ \Delta{\cdot}ae)$
⟨*proof*⟩

lemma *ABinds-delete-bot*: $ae\ x = \bot \Longrightarrow ABinds\ (delete\ x\ \Gamma){\cdot}ae = ABinds\ \Gamma{\cdot}ae$
⟨*proof*⟩

lemma *ABinds-restr-fresh*:
assumes $atom\ `\ S\ \sharp{*}\ \Gamma$
shows $ABinds\ \Gamma{\cdot}ae\ f|`\ (-\ S) = ABinds\ \Gamma{\cdot}(ae\ \ f|`\ (-\ S))\ f|`\ (-\ S)$
⟨*proof*⟩

lemma *ABinds-restr*:
assumes $domA\ \Gamma \subseteq S$
shows $ABinds\ \Gamma{\cdot}ae\ f|`\ S = ABinds\ \Gamma{\cdot}(ae\ \ f|`\ S)\ f|`\ S$
⟨*proof*⟩

lemma *ABinds-restr-subst*:
assumes $\bigwedge x'\ e\ a.\ (x',e) \in set\ \Gamma \Longrightarrow Aexp\ e[x::=y]{\cdot}a\ f|`\ S = Aexp\ e{\cdot}a\ f|`\ S$
assumes $x \notin S$
assumes $y \notin S$
assumes $domA\ \Gamma \subseteq S$
shows $ABinds\ \Gamma[x::h=y]{\cdot}ae\ f|`\ S = ABinds\ \Gamma{\cdot}(ae\ \ f|`\ S)\ f|`\ S$
⟨*proof*⟩

lemma *Abinds-append-disjoint*: $domA\ \Delta \cap domA\ \Gamma = \{\} \Longrightarrow ABinds\ (\Delta\ @\ \Gamma){\cdot}ae = ABinds\ \Delta{\cdot}ae \sqcup ABinds\ \Gamma{\cdot}ae$
⟨*proof*⟩

lemma *ABinds-restr-subset*: $S \subseteq S' \Longrightarrow ABinds\ (restrictA\ S\ \Gamma){\cdot}ae \sqsubseteq ABinds\ (restrictA\ S'\ \Gamma){\cdot}ae$
⟨*proof*⟩

lemma *ABinds-restrict-edom*: $ABinds\ (restrictA\ (edom\ ae)\ \Gamma){\cdot}ae = ABinds\ \Gamma{\cdot}ae$
⟨*proof*⟩

lemma *ABinds-restrict-below*: $ABinds\ (restrictA\ S\ \Gamma){\cdot}ae \sqsubseteq ABinds\ \Gamma{\cdot}ae$
⟨*proof*⟩

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

lemma *ABind-eqvt*[*eqvt*]: π ∙ (*ArityAnalysis.ABind Aexp v e*) = *ArityAnalysis.ABind* (π ∙ *Aexp*) (π ∙ *v*) (π ∙ *e*)
⟨*proof*⟩

lemma *ABinds-eqvt*[*eqvt*]: π ∙ (*ArityAnalysis.ABinds Aexp* Γ) = *ArityAnalysis.ABinds* (π ∙ *Aexp*) (π ∙ Γ)
⟨*proof*⟩

lemma *Abinds-cong*[*fundef-cong*]:
⟦ (⋀ *e*. *e* ∈ *snd* ‘ *set heap2* ⟹ *aexp1 e* = *aexp2 e*) ; *heap1* = *heap2* ⟧
⟹ *ArityAnalysis.ABinds aexp1 heap1* = *ArityAnalysis.ABinds aexp2 heap2*
⟨*proof*⟩

context *EdomArityAnalysis*
begin
lemma *fup-Aexp-lookup-fresh*: *atom v* ♯ *e* ⟹ (*fup*·(*Aexp e*)·*a*) *v* = ⊥
⟨*proof*⟩

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

end

5.3 ArityAnalysisSpec

theory *ArityAnalysisSpec*
imports *ArityAnalysisAbinds*
begin

locale *SubstArityAnalysis* = *EdomArityAnalysis* +
assumes *Aexp-subst-restr*: *x* ∉ *S* ⟹ *y* ∉ *S* ⟹ (*Aexp e*[*x*::=*y*] ∙ *a*) *f*|‘ *S* = (*Aexp e*·*a*) *f*|‘ *S*

locale *ArityAnalysisSafe* = *SubstArityAnalysis* +
assumes *Aexp-Var*: *up* ∙ *n* ⊑ (*Aexp* (*Var x*)·*n*) *x*
assumes *Aexp-App*: *Aexp e* ·(*inc*·*n*) ⊔ *esing x* ∙ (*up*·*0*) ⊑ *Aexp* (*App e x*) ∙ *n*
assumes *Aexp-Lam*: *env-delete y* (*Aexp e* ·(*pred*·*n*)) ⊑ *Aexp* (*Lam* [*y*]. *e*) ∙ *n*
assumes *Aexp-IfThenElse*: *Aexp scrut*·*0* ⊔ *Aexp e1*·*a* ⊔ *Aexp e2*·*a* ⊑ *Aexp* (*scrut ? e1* : *e2*)·*a*

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

assumes *Aheap-subst*: $x \notin domA\ \Gamma \Longrightarrow y \notin domA\ \Gamma \Longrightarrow Aheap\ \Gamma[x::h=y]\ e[x ::= y]\ = Aheap\ \Gamma\ e$

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

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

context *SubstArityAnalysis*
begin
lemma *Aexp-subst-upd*: $(Aexp\ e[y::=x] \cdot n) \sqsubseteq (Aexp\ e \cdot n)(y := \bot,\ x := up \cdot 0)$
⟨*proof*⟩

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

context *ArityAnalysisSafe*
begin

lemma *Aexp-Var-singleton*: $esing\ x \cdot (up \cdot n) \sqsubseteq Aexp\ (Var\ x) \cdot n$
⟨*proof*⟩

lemma *fup-Aexp-Var*: $esing\ x \cdot n \sqsubseteq fup \cdot (Aexp\ (Var\ x)) \cdot n$
⟨*proof*⟩
end

context *ArityAnalysisLetSafe*
begin
lemma *Aheap-nonrec*:
assumes *nonrec* Δ
shows $Aexp\ e \cdot a\ f|`\ domA\ \Delta \sqsubseteq Aheap\ \Delta\ e \cdot a$
⟨*proof*⟩
end

end

5.4 TrivialArityAnal

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

definition *Trivial-Aexp* :: $exp \Rightarrow Arity \to AEnv$
where $Trivial\text{-}Aexp\ e = (\Lambda\ n.\ (\lambda\ x.\ up \cdot 0)\ f|`\ fv\ e)$

lemma *Trivial-Aexp-simp*: *Trivial-Aexp e · n* = (λ *x. up·0*) *f|‘ fv e*
⟨*proof*⟩

lemma *edom-Trivial-Aexp*[*simp*]: *edom* (*Trivial-Aexp e · n*) = *fv e*
⟨*proof*⟩

lemma *Trivial-Aexp-eq*[*iff*]: *Trivial-Aexp e · n* = *Trivial-Aexp e′ · n′* ⟷ *fv e* = (*fv e′* :: *var set*)
⟨*proof*⟩

lemma *below-Trivial-Aexp*[*simp*]: (*ae* ⊑ *Trivial-Aexp e · n*) ⟷ *edom ae* ⊆ *fv e*
⟨*proof*⟩

interpretation *ArityAnalysis Trivial-Aexp*⟨*proof*⟩
interpretation *EdomArityAnalysis Trivial-Aexp*
⟨*proof*⟩

interpretation *ArityAnalysisSafe Trivial-Aexp*
⟨*proof*⟩

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

lemma *Trivial-Aheap-eqvt*[*eqvt*]: π ∙ (*Trivial-Aheap* Γ *e*) = *Trivial-Aheap* (π ∙ Γ) (π ∙ *e*)
⟨*proof*⟩

lemma *Trivial-Aheap-simp*: *Trivial-Aheap* Γ *e· a* = (λ *x. up·0*) *f|‘ domA* Γ
⟨*proof*⟩

lemma *Trivial-fup-Aexp-below-fv*: *fup·*(*Trivial-Aexp e*)*·a* ⊑ (λ *x* . *up·0*) *f|‘ fv e*
⟨*proof*⟩

lemma *Trivial-Abinds-below-fv*: *ABinds* Γ*·ae* ⊑ (λ *x* . *up·0*) *f|‘ fv* Γ
⟨*proof*⟩

interpretation *ArityAnalysisLetSafe Trivial-Aexp Trivial-Aheap*
⟨*proof*⟩

end

5.5 ArityAnalysisStack

theory *ArityAnalysisStack*
imports *SestoftConf ArityAnalysisSig*
begin

context *ArityAnalysis*

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

context *EdomArityAnalysis*
begin
lemma *edom-AEstack*: *edom* (*AEstack as S*) ⊆ *fv S*
⟨*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*

⟨*proof*⟩

lemma *Afix-reorder*: *map-of* Γ = *map-of* Δ ⟹ *Afix* Γ = *Afix* Δ
⟨*proof*⟩

lemma *Afix-repeat-singleton*: (μ *xa*. *Afix* Γ·(*esing* *x*·(*n* ⊔ *xa* *x*) ⊔ *ae*)) = *Afix* Γ·(*esing* *x*·*n* ⊔ *ae*)
⟨*proof*⟩

lemma *Afix-join-fresh*: *ae*′ ‘ (*domA* Δ) ⊆ {⊥} ⟹ *Afix* Δ·(*ae* ⊔ *ae*′) = (*Afix* Δ·*ae*) ⊔ *ae*′
⟨*proof*⟩

lemma *Afix-restr-fresh*:
assumes *atom* ‘ *S* ♯∗ Γ
shows *Afix* Γ·*ae* *f*|‘ (− *S*) = *Afix* Γ·(*ae* *f*|‘ (− *S*)) *f*|‘ (− *S*)
⟨*proof*⟩

lemma *Afix-restr*:
assumes *domA* Γ ⊆ *S*
shows *Afix* Γ·*ae* *f*|‘ *S* = *Afix* Γ·(*ae* *f*|‘ *S*) *f*|‘ *S*
⟨*proof*⟩

lemma *Afix-restr-subst*′:
assumes ⋀ *x*′ *e* *a*. (*x*′,*e*) ∈ *set* Γ ⟹ *Aexp* *e*[*x*::=*y*]·*a* *f*|‘ *S* = *Aexp* *e*·*a* *f*|‘ *S*
assumes *x* ∉ *S*
assumes *y* ∉ *S*
assumes *domA* Γ ⊆ *S*
shows *Afix* Γ[*x*::*h*=*y*]·*ae* *f*|‘ *S* = *Afix* Γ·(*ae* *f*|‘ *S*) *f*|‘ *S*
⟨*proof*⟩

lemma *Afix-subst-approx*:
assumes ⋀ *v* *n*. *v* ∈ *domA* Γ ⟹ *Aexp* (*the* (*map-of* Γ *v*))[*y*::=*x*]·*n* ⊑ (*Aexp* (*the* (*map-of* Γ *v*))·*n*)(*y* := ⊥, *x* := *up*·*0*)
assumes *x* ∉ *domA* Γ
assumes *y* ∉ *domA* Γ
shows *Afix* Γ[*y*::*h*= *x*]·(*ae*(*y* := ⊥, *x* := *up*·*0*)) ⊑ (*Afix* Γ·*ae*)(*y* := ⊥, *x* := *up*·*0*)
⟨*proof*⟩

end

lemma *Afix-eqvt*[*eqvt*]: π ∙ (*ArityAnalysis.Afix* *Aexp* Γ) = *ArityAnalysis.Afix* (π ∙ *Aexp*) (π ∙ Γ)
⟨*proof*⟩

lemma *Afix-cong*[*fundef-cong*]:
⟦ (⋀ *e*. *e* ∈ *snd* ‘ *set heap2* ⟹ *aexp1 e* = *aexp2 e*); *heap1* = *heap2* ⟧
⟹ *ArityAnalysis.Afix aexp1 heap1* = *ArityAnalysis.Afix aexp2 heap2*
⟨*proof*⟩

context *EdomArityAnalysis*
begin

lemma *Afix-edom*: *edom* (*Afix* Γ · *ae*) ⊆ *fv* Γ ∪ *edom ae*
⟨*proof*⟩

lemma *ABinds-lookup-fresh*:
atom v ♯ Γ ⟹ (*ABinds* Γ·*ae*) *v* = ⊥
⟨*proof*⟩

lemma *Afix-lookup-fresh*:
assumes *atom v* ♯ Γ
shows (*Afix* Γ·*ae*) *v* = *ae v*
⟨*proof*⟩

lemma *Afix-comp2join-fresh*:
atom ‘ (*domA* Δ) ♯* Γ ⟹ *ABinds* Δ·(*Afix* Γ·*ae*) = *ABinds* Δ·*ae*
⟨*proof*⟩

lemma *Afix-append-fresh*:
assumes *atom* ‘ *domA* Δ ♯* Γ
shows *Afix* (Δ @ Γ)·*ae* = *Afix* Γ·(*Afix* Δ·*ae*)
⟨*proof*⟩

lemma *Afix-e-to-heap*:
Afix (*delete x* Γ)·(*fup*·(*Aexp e*)·*n* ⊔ *ae*) ⊑ *Afix* ((*x*, *e*) # *delete x* Γ)·(*esing x*·*n* ⊔ *ae*)
⟨*proof*⟩

lemma *Afix-e-to-heap′*:
Afix (*delete x* Γ)·(*Aexp e*·*n*) ⊑ *Afix* ((*x*, *e*) # *delete x* Γ)·(*esing x*·(*up*·*n*))
⟨*proof*⟩

end

end

5.7 ArityAnalysisFixProps

theory *ArityAnalysisFixProps*
imports *ArityAnalysisFix ArityAnalysisSpec*

begin

context *SubstArityAnalysis*
begin

lemma *Afix-restr-subst*:
assumes $x \notin S$
assumes $y \notin S$
assumes *domA* $\Gamma \subseteq S$
shows *Afix* $\Gamma[x::h=y]\cdot ae\ f|`S = Afix\ \Gamma\cdot(ae\ f|`S)\ f|`S$
⟨*proof*⟩
end

end

6 Arity Transformation

6.1 AbstractTransform

theory *AbstractTransform*
imports *Launchbury.Terms TransformTools*
begin

locale *AbstractAnalProp* =
fixes *PropApp* :: $'a \Rightarrow 'a$::*cont-pt*
fixes *PropLam* :: $'a \Rightarrow 'a$
fixes *AnalLet* :: *heap* ⇒ *exp* ⇒ $'a \Rightarrow 'b$::*cont-pt*
fixes *PropLetBody* :: $'b \Rightarrow 'a$
fixes *PropLetHeap* :: $'b \Rightarrow$ *var* ⇒ $'a_\perp$
fixes *PropIfScrut* :: $'a \Rightarrow 'a$
assumes *PropApp-eqvt*: $\pi \cdot$ *PropApp* ≡ *PropApp*
assumes *PropLam-eqvt*: $\pi \cdot$ *PropLam* ≡ *PropLam*
assumes *AnalLet-eqvt*: $\pi \cdot$ *AnalLet* ≡ *AnalLet*
assumes *PropLetBody-eqvt*: $\pi \cdot$ *PropLetBody* ≡ *PropLetBody*
assumes *PropLetHeap-eqvt*: $\pi \cdot$ *PropLetHeap* ≡ *PropLetHeap*
assumes *PropIfScrut-eqvt*: $\pi \cdot$ *PropIfScrut* ≡ *PropIfScrut*

locale *AbstractAnalPropSubst* = *AbstractAnalProp* +
assumes *AnalLet-subst*: $x \notin$ *domA* $\Gamma \Longrightarrow y \notin$ *domA* $\Gamma \Longrightarrow$ *AnalLet* $(\Gamma[x::h=y])\ (e[x::=y])\ a$ = *AnalLet* $\Gamma\ e\ a$

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

assumes *TransVar-eqvt*: $\pi \cdot \mathit{TransVar} = \mathit{TransVar}$
assumes *TransApp-eqvt*: $\pi \cdot \mathit{TransApp} = \mathit{TransApp}$
assumes *TransLam-eqvt*: $\pi \cdot \mathit{TransLam} = \mathit{TransLam}$
assumes *TransLet-eqvt*: $\pi \cdot \mathit{TransLet} = \mathit{TransLet}$
assumes *SuppTransLam*: $\mathit{supp}\ (\mathit{TransLam}\ a\ v\ e) \subseteq \mathit{supp}\ e - \mathit{supp}\ v$
assumes *SuppTransLet*: $\mathit{supp}\ (\mathit{TransLet}\ b\ \Gamma\ e) \subseteq \mathit{supp}\ (\Gamma,\ e) - \mathit{atom}\ `\ \mathit{domA}\ \Gamma$
begin
nominal-function *transform* **where**
$\mathit{transform}\ a\ (\mathit{App}\ e\ x) = \mathit{TransApp}\ a\ (\mathit{transform}\ (\mathit{PropApp}\ a)\ e)\ x$
| $\mathit{transform}\ a\ (\mathit{Lam}\ [x].\ e) = \mathit{TransLam}\ a\ x\ (\mathit{transform}\ (\mathit{PropLam}\ a)\ e)$
| $\mathit{transform}\ a\ (\mathit{Var}\ x) = \mathit{TransVar}\ a\ x$
| $\mathit{transform}\ a\ (\mathit{Let}\ \Gamma\ e) = \mathit{TransLet}\ (\mathit{AnalLet}\ \Gamma\ e\ a)$
$(\mathit{map\text{-}transform}\ \mathit{transform}\ (\mathit{PropLetHeap}\ (\mathit{AnalLet}\ \Gamma\ e\ a))\ \Gamma)$
$(\mathit{transform}\ (\mathit{PropLetBody}\ (\mathit{AnalLet}\ \Gamma\ e\ a))\ e)$
| $\mathit{transform}\ a\ (\mathit{Bool}\ b) = (\mathit{Bool}\ b)$
| $\mathit{transform}\ a\ (\mathit{scrut}\ ?\ e1 : e2) = (\mathit{transform}\ (\mathit{PropIfScrut}\ a)\ \mathit{scrut}\ ?\ \mathit{transform}\ a\ e1 : \mathit{transform}\ a\ e2)$
⟨*proof*⟩
nominal-termination ⟨*proof*⟩

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

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

end

locale *AbstractTransformSubst* = *AbstractTransform* + *AbstractAnalPropSubst* +
assumes *TransVar-subst*: $(\mathit{TransVar}\ a\ v)[x ::= y] = (\mathit{TransVar}\ a\ v[x ::v= y])$
assumes *TransApp-subst*: $(\mathit{TransApp}\ a\ e\ v)[x ::= y] = (\mathit{TransApp}\ a\ e[x ::= y]\ v[x ::v= y])$
assumes *TransLam-subst*: $\mathit{atom}\ v\ \sharp\ (x,y) \Longrightarrow (\mathit{TransLam}\ a\ v\ e)[x ::= y] = (\mathit{TransLam}\ a\ v[x ::v= y]\ e[x ::= y])$
assumes *TransLet-subst*: $\mathit{atom}\ `\ \mathit{domA}\ \Gamma\ \sharp*\ (x,y) \Longrightarrow (\mathit{TransLet}\ b\ \Gamma\ e)[x ::= y] = (\mathit{TransLet}\ b\ \Gamma[x ::h= y]\ e[x ::= y])$
begin
lemma *subst-transform*: $(\mathit{transform}\ a\ e)[x ::= y] = \mathit{transform}\ a\ e[x ::= y]$
⟨*proof*⟩
end

locale *AbstractTransformBound* = *AbstractAnalProp* + *supp-bounded-transform* +
constrains *PropApp* :: $'a \Rightarrow 'a$::*pure-cont-pt*
constrains *PropLetHeap* :: $'b$::*cont-pt* $\Rightarrow$ *var* $\Rightarrow$ $'a_\perp$
constrains *trans* :: $'c$::*cont-pt* $\Rightarrow$ *exp* $\Rightarrow$ *exp*
fixes *PropLetHeapTrans* :: $'b \Rightarrow$ *var* $\Rightarrow$ $'c_\perp$
assumes *PropLetHeapTrans-eqvt*: $\pi \cdot \mathit{PropLetHeapTrans} = \mathit{PropLetHeapTrans}$
assumes *TransBound-eqvt*: $\pi \cdot \mathit{trans} = \mathit{trans}$
begin

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

lemma *isLam-transform*[*simp*]:
isLam (*transform a e*) ⟷ *isLam e*
⟨*proof*⟩

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

end

locale *AbstractTransformBoundSubst* = *AbstractAnalPropSubst* + *AbstractTransformBound* +

assumes *TransBound-subst*: (*trans a e*)[*x*::=*y*] = *trans a e*[*x*::=*y*]
begin
sublocale *AbstractTransformSubst PropApp PropLam AnalLet PropLetBody PropLetHeap PropIfScrut*
(λ *a*. *Var*)
(λ *a*. *App*)
(λ *a*. *Terms.Lam*)
(λ *b* Γ *e* . *Let* (*map-transform trans* (*PropLetHeapTrans b*) Γ) *e*)
⟨*proof*⟩
end

end

6.2 ArityTransform

theory *ArityTransform*
imports *ArityAnalysisSig AbstractTransform ArityEtaExpansionSafe*
begin

context *ArityAnalysisHeapEqvt*
begin
sublocale *AbstractTransformBound*
λ *a* . *inc*·*a*
λ *a* . *pred*·*a*
λ Δ *e a* . (*a*, *Aheap* Δ *e*·*a*)
fst
snd
λ -. *0*

Aeta-expand
snd
⟨*proof*⟩

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

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

end

7 Arity Analysis Safety (without Cardinality)

7.1 ArityConsistent

theory *ArityConsistent*
imports *ArityAnalysisSpec ArityStack ArityAnalysisStack*
begin

context *ArityAnalysisLetSafe*
begin

type-synonym *astate* = (*AEnv* × *Arity* × *Arity list*)

inductive *stack-consistent* :: *Arity list* ⇒ *stack* ⇒ *bool*
where
stack-consistent [] []
| *Astack S* ⊑ *a* ⟹ *stack-consistent as S* ⟹ *stack-consistent* (*a*#*as*) (*Alts e1 e2* # *S*)
| *stack-consistent as S* ⟹ *stack-consistent as* (*Upd x* # *S*)
| *stack-consistent as S* ⟹ *stack-consistent as* (*Arg x* # *S*)
inductive-simps *stack-consistent-foo*[*simp*]:
stack-consistent [] [] *stack-consistent* (*a*#*as*) (*Alts e1 e2* # *S*) *stack-consistent as* (*Upd x* # *S*) *stack-consistent as* (*Arg x* # *S*)
inductive-cases [*elim*!]: *stack-consistent as* (*Alts e1 e2* # *S*)

inductive *a-consistent* :: *astate* ⇒ *conf* ⇒ *bool* **where**
a-consistentI:
edom ae ⊆ *domA* Γ ∪ *upds S*
⟹ *Astack S* ⊑ *a*

$\Longrightarrow$ (*ABinds* $\Gamma{\cdot}ae \sqcup$ *Aexp* $e{\cdot}a \sqcup$ *AEstack as S*) $f|`$ (*domA* $\Gamma \cup$ *upds* S) $\sqsubseteq$ *ae*
$\Longrightarrow$ *stack-consistent as S*
$\Longrightarrow$ *a-consistent* (*ae*, *a*, *as*) (Γ, *e*, *S*)
inductive-cases *a-consistentE*: *a-consistent* (*ae*, *a*, *as*) (Γ, *e*, *S*)

lemma *a-consistent-restrictA*:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *e*, *S*)
assumes *edom ae* $\subseteq$ *V*
shows *a-consistent* (*ae*, *a*, *as*) (*restrictA V* Γ, *e*, *S*)
⟨*proof*⟩

lemma *a-consistent-edom-subsetD*:
a-consistent (*ae*, *a*, *as*) (Γ, *e*, *S*) $\Longrightarrow$ *edom ae* $\subseteq$ *domA* $\Gamma \cup$ *upds S*
⟨*proof*⟩

lemma *a-consistent-stackD*:
a-consistent (*ae*, *a*, *as*) (Γ, *e*, *S*) $\Longrightarrow$ *Astack S* $\sqsubseteq$ *a*
⟨*proof*⟩

lemma *a-consistent-app*$_1$:
a-consistent (*ae*, *a*, *as*) (Γ, *App e x*, *S*) $\Longrightarrow$ *a-consistent* (*ae*, *inc*·*a*, *as*) (Γ, *e*, *Arg x # S*)
⟨*proof*⟩

lemma *a-consistent-app*$_2$:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, (*Lam* [*y*]. *e*), *Arg x # S*)
shows *a-consistent* (*ae*, (*pred*·*a*), *as*) (Γ, *e*[*y*::=*x*], *S*)
⟨*proof*⟩

lemma *a-consistent-thunk-0*:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *Var x*, *S*)
assumes *map-of* Γ *x* = *Some e*
assumes *ae x* = *up*·*0*
shows *a-consistent* (*ae*, *0*, *as*) (*delete x* Γ, *e*, *Upd x # S*)
⟨*proof*⟩

lemma *a-consistent-thunk-once*:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *Var x*, *S*)
assumes *map-of* Γ *x* = *Some e*
assumes [*simp*]: *ae x* = *up*·*u*
assumes *heap-upds-ok* (Γ, *S*)
shows *a-consistent* (*env-delete x ae*, *u*, *as*) (*delete x* Γ, *e*, *S*)
⟨*proof*⟩

lemma *a-consistent-lamvar*:
assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *Var x*, *S*)
assumes *map-of* Γ *x* = *Some e*
assumes [*simp*]: *ae x* = *up*·*u*
shows *a-consistent* (*ae*, *u*, *as*) ((*x*,*e*)# *delete x* Γ, *e*, *S*)

⟨*proof*⟩

lemma
  assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *e*, *Upd x # S*)
  shows *a-consistent-var*$_2$: *a-consistent* (*ae*, *a*, *as*) ((*x*, *e*) # Γ, *e*, *S*)
    and *a-consistent-UpdD*: *ae x* = *up*·*0a* = *0*
    ⟨*proof*⟩

lemma *a-consistent-let*:
  assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *Let* Δ *e*, *S*)
  assumes *atom* ' *domA* Δ ♯* Γ
  assumes *atom* ' *domA* Δ ♯* *S*
  assumes *edom ae* ∩ *domA* Δ = {}
  shows *a-consistent* (*Aheap* Δ *e*·*a* ⊔ *ae*, *a*, *as*) (Δ @ Γ, *e*, *S*)
⟨*proof*⟩

lemma *a-consistent-if*$_1$:
  assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *scrut ? e1 : e2*, *S*)
  shows *a-consistent* (*ae*, *0*, *a#as*) (Γ, *scrut*, *Alts e1 e2 # S*)
⟨*proof*⟩

lemma *a-consistent-if*$_2$:
  assumes *a-consistent* (*ae*, *a*, *a'#as'*) (Γ, *Bool b*, *Alts e1 e2 # S*)
  shows *a-consistent* (*ae*, *a'*, *as'*) (Γ, *if b then e1 else e2*, *S*)
⟨*proof*⟩

lemma *a-consistent-alts-on-stack*:
  assumes *a-consistent* (*ae*, *a*, *as*) (Γ, *Bool b*, *Alts e1 e2 # S*)
  obtains *a' as'* **where** *as* = *a' # as' a* = *0*
⟨*proof*⟩

lemma *closed-a-consistent*:
  fv e = ({}::*var set*) ⟹ *a-consistent* (⊥, *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*

$\lambda\ a\ .\ inc{\cdot}a$
$\lambda\ a\ .\ pred{\cdot}a$
$\lambda\ \Delta\ e\ a\ .\ (a,\ Aheap\ \Delta\ e{\cdot}a)$
fst
snd
λ -. *0*
Aeta-expand
snd
⟨*proof*⟩

abbreviation *ccTransform* **where** *ccTransform* ≡ *transform*

lemma *supp-transform*: *supp* (*transform a e*) ⊆ *supp e*
⟨*proof*⟩
interpretation *supp-bounded-transform transform*
⟨*proof*⟩

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

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

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 Aeta-expand ae* (*map-transform ccTransform ae* Γ),
ccTransform a e,
transform-alts as S)

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

inductive *consistent* :: *astate* ⇒ *conf* ⇒ *bool* **where**
consistentI[*intro*!]:
a-consistent (*ae*, *a*, *as*) (Γ, *e*, *S*)
⟹ (⋀ *x*. *x* ∈ *thunks* Γ ⟹ *ae x* = *up*·*0*)
⟹ *consistent* (*ae*, *a*, *as*) (Γ, *e*, *S*)
inductive-cases *consistentE*[*elim*!]: *consistent* (*ae*, *a*, *as*) (Γ, *e*, *S*)

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

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

end

8 Cardinality Analysis

8.1 Cardinality-Domain

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

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

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

lemma *many-max*[*simp*]: *a* ⊑ *many* ⟨*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* = (Λ *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* = *up*·*u* ⟹ *isVal e* ⟹ *prognosis ae as u* (Γ, *e*, *S*) ⊑ *record-call x* · (*prognosis ae as a* (Γ, *Var x*, *S*))
  assumes *prognosis-Var-thunk*: *map-of* Γ *x* = *Some e* ⟹ *ae x* = *up*·*u* ⟹ ¬ *isVal e* ⟹ *prognosis ae as u* (*delete x* Γ, *e*, *Upd x* # *S*) ⊑ *record-call x* · (*prognosis ae as a* (Γ, *Var x*, *S*))
  assumes *prognosis-Var2*: *isVal e* ⟹ *x* ∉ *domA* Γ ⟹ *prognosis ae as 0* ((*x*, *e*) # Γ, *e*, *S*) ⊑ *prognosis ae as 0* (Γ, *e*, *Upd x* # *S*)

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

locale *CardinalityPrognosisLet* = *CardinalityPrognosis* + *CardinalityHeap* + *ArityAnalysisHeap* +
  assumes *prognosis-Let*:

atom ‘ domA Δ ♯∗ Γ ⟹ atom ‘ domA Δ ♯∗ S ⟹ edom ae ⊆ domA Γ ∪ upds S ⟹ prognosis (Aheap Δ e·a ⊔ ae) as a (Δ @ Γ, e, S) ⊑ cHeap Δ e·a ⊔ prognosis ae as a (Γ, Terms.Let Δ e, S)

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

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

end

8.4 NoCardinalityAnalysis

theory *NoCardinalityAnalysis*
imports *CardinalityAnalysisSpec ArityAnalysisStack*
begin

locale *NoCardinalityAnalysis = ArityAnalysisLetSafe +*
  assumes *Aheap-thunk: x ∈ thunks Γ ⟹ (Aheap Γ e·a) x = up·0*
begin

definition *a2c :: Arity⊥ → two* **where** *a2c = (Λ a. if a ⊑ ⊥ then ⊥ else many)*
lemma *a2c-simp: a2c·a = (if a ⊑ ⊥ then ⊥ else many)*
  ⟨*proof*⟩

lemma *a2c-eqvt[eqvt]: π · a2c = a2c*
  ⟨*proof*⟩

definition *ae2ce :: AEnv ⇒ (var ⇒ two)* **where** *ae2ce ae x = a2c·(ae x)*

lemma *ae2ce-cont: cont ae2ce*
  ⟨*proof*⟩
lemmas *cont-compose[OF ae2ce-cont, cont2cont, simp]*

lemma *ae2ce-eqvt[eqvt]: π · ae2ce ae x = ae2ce (π · ae) (π · x)*
  ⟨*proof*⟩

lemma *ae2ce-to-env-restr*: *ae2ce ae* = (λ- . *many*) *f*|‘ *edom ae*
⟨*proof*⟩

lemma *edom-ae2ce*[*simp*]: *edom* (*ae2ce ae*) = *edom ae*
⟨*proof*⟩

definition *cHeap* :: *heap* ⇒ *exp* ⇒ *Arity* → (*var* ⇒ *two*)
where *cHeap* Γ *e* = (Λ *a*. *ae2ce* (*Aheap* Γ *e*·*a*))
lemma *cHeap-simp*[*simp*]: *cHeap* Γ *e*·*a* = *ae2ce* (*Aheap* Γ *e*·*a*)
⟨*proof*⟩

sublocale *CardinalityHeap cHeap*⟨*proof*⟩

sublocale *CardinalityHeapSafe cHeap Aheap*
⟨*proof*⟩

fun *prognosis* **where**
prognosis ae as a (Γ, *e*, *S*) = ((λ-. *many*) *f*|‘ (*edom* (*ABinds* Γ·*ae*) ∪ *edom* (*Aexp e*·*a*) ∪ *edom* (*AEstack as S*)))

lemma *record-all-noop*[*simp*]:
record-call x·((λ-. *many*) *f*|‘ *S*) = (λ-. *many*) *f*|‘ *S*
⟨*proof*⟩

lemma *const-on-restr-constI*[*intro*]:
S′ ⊆ *S* ⟹ *const-on* ((λ -. *x*) *f*|‘ *S*) *S*′ *x*
⟨*proof*⟩

lemma *ap-subset-edom-AEstack*: *ap S* ⊆ *edom* (*AEstack as S*)
⟨*proof*⟩

sublocale *CardinalityPrognosis prognosis*⟨*proof*⟩

sublocale *CardinalityPrognosisShape prognosis*
⟨*proof*⟩

sublocale *CardinalityPrognosisApp prognosis*
⟨*proof*⟩

sublocale *CardinalityPrognosisLam prognosis*
⟨*proof*⟩

sublocale *CardinalityPrognosisVar prognosis*
⟨*proof*⟩

sublocale *CardinalityPrognosisIfThenElse prognosis*
⟨*proof*⟩

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

sublocale *CardinalityPrognosisEdom prognosis*
⟨*proof*⟩

sublocale *CardinalityPrognosisSafe prognosis cHeap Aheap Aexp*⟨*proof*⟩
end

end

8.5 CardArityTransformSafe

theory *CardArityTransformSafe*
imports *ArityTransform CardinalityAnalysisSpec AbstractTransform Sestoft SestoftGC ArityEta-ExpansionSafe ArityAnalysisStack ArityConsistent*
begin

context *CardinalityPrognosisSafe*
begin
sublocale *AbstractTransformBoundSubst*
λ a . inc·a
λ a . pred·a
λ Δ e a . (a, Aheap Δ e·a)
fst
snd
λ -. 0
Aeta-expand
snd
⟨*proof*⟩

abbreviation *ccTransform* **where** *ccTransform ≡ transform*

lemma *supp-transform*: *supp (transform a e) ⊆ supp e*
⟨*proof*⟩
interpretation *supp-bounded-transform transform*
⟨*proof*⟩

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

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

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

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

lemma *fresh-star-transform-alts*[*intro*]: $a \sharp* S \Longrightarrow a \sharp* \textit{transform-alts as } S$
⟨*proof*⟩

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

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

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

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

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

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

lemma *card-arity-transform-safe*:
fixes *c c′*
assumes *c* ⇒* *c′* **and** ¬ *boring-step c′* **and** *heap-upds-ok-conf c* **and** *consistent* (*ae*,*ce*,*a*,*as*,*r*) *c*
shows ∃ *ae′ ce′ a′ as′ r′*. *consistent* (*ae′*,*ce′*,*a′*,*as′*,*r′*) *c′* ∧ *conf-transform* (*ae*,*ce*,*a*,*as*,*r*) *c* ⇒$_G$* *conf-transform* (*ae′*,*ce′*,*a′*,*as′*,*r′*) *c′*
⟨*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-ttree*

9.1.3 Deconstructors

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

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

9.1.4 Trees as set of paths

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

lemma *paths-inj*: *paths t* = *paths t'* ⟹ *t* = *t'* ⟨*proof*⟩

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

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

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

lemma *paths-Cons-nxt*:
possible t x ⟹ *xs* ∈ *paths* (*nxt t x*) ⟹ (*x*#*xs*) ∈ *paths t*
⟨*proof*⟩

lemma *paths-Cons-nxt-iff*:
possible t x ⟹ *xs* ∈ *paths* (*nxt t x*) ⟷ (*x*#*xs*) ∈ *paths t*
⟨*proof*⟩

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

lemma *nxt-mono*:
paths t ⊆ *paths t'* ⟹ *paths* (*nxt t x*) ⊆ *paths* (*nxt t' x*)
⟨*proof*⟩

lemma *ttree-eqI*: (⋀ *x xs*. *x*#*xs* ∈ *paths t* ⟷ *x*#*xs* ∈ *paths t'*) ⟹ *t* = *t'*
⟨*proof*⟩

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

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

lemma *Cons-pathI*[*intro*]:
assumes $possible\ t\ x \longleftrightarrow possible\ t'\ x$
assumes $possible\ t\ x \Longrightarrow possible\ t'\ x \Longrightarrow xs \in paths\ (nxt\ t\ x) \longleftrightarrow xs \in paths\ (nxt\ t'\ x)$
shows $x \# xs \in paths\ t \longleftrightarrow x \# xs \in paths\ t'$
⟨*proof*⟩

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

lemma *ttree-coinduct*:
assumes $P\ t\ t'$
assumes $\bigwedge\ t\ t'\ x\ .\ P\ t\ t' \Longrightarrow possible\ t\ x \longleftrightarrow possible\ t'\ x$
assumes $\bigwedge\ t\ t'\ x\ .\ P\ t\ t' \Longrightarrow possible\ t\ x \Longrightarrow possible\ t'\ x \Longrightarrow P\ (nxt\ t\ x)\ (nxt\ t'\ x)$
shows $t = t'$
⟨*proof*⟩

9.1.5 The carrier of a tree

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

lemma *carrier-mono*: $paths\ t \subseteq paths\ t' \Longrightarrow carrier\ t \subseteq carrier\ t'$ ⟨*proof*⟩

lemma *carrier-possible*:
$possible\ t\ x \Longrightarrow x \in carrier\ t$ ⟨*proof*⟩

lemma *carrier-possible-subset*:
$carrier\ t \subseteq A \Longrightarrow possible\ t\ x \Longrightarrow x \in A$ ⟨*proof*⟩

lemma *carrier-nxt-subset*:
$carrier\ (nxt\ t\ x) \subseteq carrier\ t$
⟨*proof*⟩

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

9.1.6 Repeatable trees

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

lemma *nxt-repeatable*[*simp*]: $repeatable\ t \Longrightarrow possible\ t\ x \Longrightarrow nxt\ t\ x = t$
⟨*proof*⟩

9.1.7 Simple trees

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

lemma *possible-empty*[*simp*]: *possible empty* $x' \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *nxt-not-possible*[*simp*]: ¬ *possible t x* $\Longrightarrow$ *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** λ *x*. {[], [*x*]}
⟨*proof*⟩

lemma *possible-single*[*simp*]: *possible* (*single x*) $x' \longleftrightarrow x = x'$
⟨*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** λ *x*. *range* (λ *n*. *replicate n x*)
⟨*proof*⟩

lemma *possible-many-calls*[*simp*]: *possible* (*many-calls x*) $x' \longleftrightarrow x = x'$
⟨*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* $x' \longleftrightarrow$ *True*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *paths-anything*[*simp*]:
paths anything = *UNIV* ⟨*proof*⟩

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

lift-definition *many-among* :: *'a set* ⇒ *'a ttree* **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 ttree* ⇒ *'a ttree* ⇒ *'a ttree* (**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 ttree* ⇒ *'a ttree* ⇒ *'a ttree* (**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 $(t \oplus\oplus t') = \mathit{carrier}\ t \cup \mathit{carrier}\ t'$
⟨*proof*⟩

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

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

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

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

9.1.10 Merging of trees

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

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

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

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

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

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

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

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

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

lemma *both-mono1*:

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

lemma *both-mono2*:
paths t $\subseteq$ *paths t'* $\Longrightarrow$ *paths* (*t''* $\otimes\otimes$ *t*) $\subseteq$ *paths* (*t''* $\otimes\otimes$ *t'*)
⟨*proof*⟩

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

lemma *nxt-both*:
nxt (*t'* $\otimes\otimes$ *t*) *x* = (*if possible t' x* $\wedge$ *possible t x then nxt t' x* $\otimes\otimes$ *t* $\oplus\oplus$ *t'* $\otimes\otimes$ *nxt t x else*
if possible t' x then nxt t' x $\otimes\otimes$ *t else*
if possible t x then t' $\otimes\otimes$ *nxt t x else*
empty)
⟨*proof*⟩

lemma *Cons-both*:
x # *xs* $\in$ *paths* (*t'* $\otimes\otimes$ *t*) $\longleftrightarrow$ (*if possible t' x* $\wedge$ *possible t x then xs* $\in$ *paths* (*nxt t' x* $\otimes\otimes$ *t*) $\vee$ *xs* $\in$ *paths* (*t'* $\otimes\otimes$ *nxt t x*) *else*
if possible t' x then xs $\in$ *paths* (*nxt t' x* $\otimes\otimes$ *t*) *else*
if possible t x then xs $\in$ *paths* (*t'* $\otimes\otimes$ *nxt t x*) *else*
False)
⟨*proof*⟩

lemma *Cons-both-possible-leftE*: *possible t x* $\Longrightarrow$ *xs* $\in$ *paths* (*nxt t x* $\otimes\otimes$ *t'*) $\Longrightarrow$ *x*#*xs* $\in$ *paths* (*t* $\otimes\otimes$ *t'*)
⟨*proof*⟩
lemma *Cons-both-possible-rightE*: *possible t' x* $\Longrightarrow$ *xs* $\in$ *paths* (*t* $\otimes\otimes$ *nxt t' x*) $\Longrightarrow$ *x*#*xs* $\in$ *paths* (*t* $\otimes\otimes$ *t'*)
⟨*proof*⟩

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

lemma *either-both-distr2*[*simp*]:
t' $\otimes\otimes$ *t* $\oplus\oplus$ *t''* $\otimes\otimes$ *t* = (*t'* $\oplus\oplus$ *t''*) $\otimes\otimes$ *t*
⟨*proof*⟩

lemma *nxt-both-repeatable*[*simp*]:
assumes [*simp*]: *repeatable t'*
assumes [*simp*]: *possible t' x*
shows *nxt* (*t'* $\otimes\otimes$ *t*) *x* = *t'* $\otimes\otimes$ (*t* $\oplus\oplus$ *nxt t x*)
⟨*proof*⟩

lemma *nxt-both-many-calls*[*simp*]: *nxt* (*many-calls x* $\otimes\otimes$ *t*) *x* = *many-calls x* $\otimes\otimes$ (*t* $\oplus\oplus$ *nxt t x*)
⟨*proof*⟩

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

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

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

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

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

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

9.1.11 Removing elements from a tree

lift-definition *without* :: *'a* ⇒ *'a ttree* ⇒ *'a ttree*
is λ *x xss. filter* (λ *x'. x'* ≠ *x*) ‘ *xss*
⟨*proof*⟩

lemma *paths-withoutI*:
assumes *xs* ∈ *paths t*
assumes *x* ∉ *set xs*
shows *xs* ∈ *paths* (*without x t*)
⟨*proof*⟩

lemma *carrier-without*[*simp*]: *carrier* (*without x t*) = *carrier t* − {*x*}
⟨*proof*⟩

lift-definition *ttree-restr* :: *'a set* ⇒ *'a ttree* ⇒ *'a ttree* **is** λ *S xss*. *filter* (λ *x'*. *x'* ∈ *S*) ‘ *xss*
⟨*proof*⟩

lemma *filter-paths-conv-free-restr*:
filter (λ *x'* . *x'* ∈ *S*) ‘ *paths t* = *paths* (*ttree-restr S t*) ⟨*proof*⟩

lemma *filter-paths-conv-free-restr2*:
filter (λ *x'* . *x'* ∉ *S*) ‘ *paths t* = *paths* (*ttree-restr* (− *S*) *t*) ⟨*proof*⟩

lemma *filter-paths-conv-free-without*:
filter (λ *x'* . *x'* ≠ *y*) ‘ *paths t* = *paths* (*without y t*) ⟨*proof*⟩

lemma *ttree-restr-is-empty*: *carrier t* ∩ *S* = {} ⟹ *ttree-restr S t* = *empty*
⟨*proof*⟩

lemma *ttree-restr-noop*: *carrier t* ⊆ *S* ⟹ *ttree-restr S t* = *t*
⟨*proof*⟩

lemma *ttree-restr-tree-restr*[*simp*]:
ttree-restr S (*ttree-restr S'* *t*) = *ttree-restr* (*S'* ∩ *S*) *t*
⟨*proof*⟩

lemma *ttree-restr-both*:
ttree-restr S (*t* ⊗⊗ *t'*) = *ttree-restr S t* ⊗⊗ *ttree-restr S t'*
⟨*proof*⟩

lemma *ttree-restr-nxt-subset*: *x* ∈ *S* ⟹ *paths* (*ttree-restr S* (*nxt t x*)) ⊆ *paths* (*nxt* (*ttree-restr S t*) *x*)
⟨*proof*⟩

lemma *ttree-restr-nxt-subset2*: *x* ∉ *S* ⟹ *paths* (*ttree-restr S* (*nxt t x*)) ⊆ *paths* (*ttree-restr S t*)
⟨*proof*⟩

lemma *ttree-restr-possible*: *x* ∈ *S* ⟹ *possible t x* ⟹ *possible* (*ttree-restr S t*) *x*
⟨*proof*⟩

lemma *ttree-restr-possible2*: *possible* (*ttree-restr S t'*) *x* ⟹ *x* ∈ *S*
⟨*proof*⟩

lemma *carrier-ttree-restr*[*simp*]:
carrier (*ttree-restr S t*) = *S* ∩ *carrier t*
⟨*proof*⟩

9.1.12 Multiple variables, each called at most once

lift-definition *singles* :: *'a set* ⇒ *'a ttree* **is** λ *S*. {*xs*. ∀ *x* ∈ *S*. *length* (*filter* (λ *x'*. *x'* = *x*) *xs*) ≤ *1*}
⟨*proof*⟩

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

lemma *length-filter-mono*[*intro*]:
assumes (⋀ *x*. *P x* ⟹ *Q x*)
shows *length* (*filter P xs*) ≤ *length* (*filter Q xs*)
⟨*proof*⟩

lemma *nxt-singles*[*simp*]: *nxt* (*singles S*) *x'* = (*if x'* ∈ *S then without x'* (*singles S*) *else singles S*)
⟨*proof*⟩

lemma *carrier-singles*[*simp*]:
carrier (*singles S*) = *UNIV*
⟨*proof*⟩

lemma *singles-mono*:
S ⊆ *S'* ⟹ *paths* (*singles S'*) ⊆ *paths* (*singles S*)
⟨*proof*⟩

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

9.1.13 Substituting trees for every node

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

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

lemma *f-nxt-mono1*: (⋀ *x*. *paths* (*f x*) ⊆ *paths* (*f' x*)) ⟹ *paths* (*f-nxt f T x x'*) ⊆ *paths* (*f-nxt f' T x x'*)
⟨*proof*⟩

lemma *f-nxt-empty-set*[*simp*]: *f-nxt f* {} *x* = *f* ⟨*proof*⟩

lemma *downset-substitute*: *downset* (*Collect* (*substitute' f T t*))
⟨*proof*⟩

lift-definition *substitute* :: ('*a* ⇒ '*a ttree*) ⇒ '*a set* ⇒ '*a ttree* ⇒ '*a ttree*
is λ *f T t*. *Collect* (*substitute' f T t*)
⟨*proof*⟩

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

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

lemma *substitute-mono2*:
assumes *paths* $t \subseteq$ *paths* t'
shows *paths* (*substitute* f T t) $\subseteq$ *paths* (*substitute* f T t')
⟨*proof*⟩

lemma *substitute-mono1*:
assumes $\bigwedge x.$ *paths* (f x) $\subseteq$ *paths* (f' x)
shows *paths* (*substitute* f T t) $\subseteq$ *paths* (*substitute* f' T t)
⟨*proof*⟩

lemma *substitute-monoT*:
assumes $T \subseteq T'$
shows *paths* (*substitute* f T' t) $\subseteq$ *paths* (*substitute* f T t)
⟨*proof*⟩

lemma *substitute-contains-arg*: *paths* $t \subseteq$ *paths* (*substitute* f T t)
⟨*proof*⟩

lemma *possible-substitute*[*simp*]: *possible* (*substitute* f T t) $x \longleftrightarrow$ *possible* t x
⟨*proof*⟩

lemma *nxt-substitute*[*simp*]: *possible* t $x \Longrightarrow$ *nxt* (*substitute* f T t) x = *substitute* (*f-nxt* f T x) T (*nxt* t $x \otimes\otimes$ f x)
⟨*proof*⟩

lemma *substitute-either*: *substitute* f T ($t \oplus\oplus t'$) = *substitute* f T $t \oplus\oplus$ *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 $f\ x = empty$
shows $substitute\ f\ (T - \{x\})\ t = substitute\ f\ T\ t$
⟨*proof*⟩

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

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

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

lemma *repeatable-f-nxt-upd*[*simp*]:
$repeatable\ (f\ x) \Longrightarrow repeatable\ (f\text{-}nxt\ f\ T\ x'\ x)$
⟨*proof*⟩

lemma *substitute-remove-anyways-aux*:
assumes $repeatable\ (f\ x)$
assumes $xs \in paths\ (substitute\ f\ T\ t)$
assumes $t \otimes\otimes f\ x = t$
shows $xs \in paths\ (substitute\ (f(x := empty))\ T\ t)$
⟨*proof*⟩

lemma *substitute-remove-anyways*:
assumes $repeatable\ t$
assumes $f\ x = t$
shows $substitute\ f\ T\ (t \otimes\otimes t') = substitute\ (f(x := empty))\ T\ (t \otimes\otimes t')$
⟨*proof*⟩

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

lemma *f-nxt-cong*: $f\ x' = f'\ x' \Longrightarrow f\text{-}nxt\ f\ T\ x\ x' = f\text{-}nxt\ f'\ T\ x\ x'$
⟨*proof*⟩

lemma *substitute-cong′*:
assumes $xs \in paths\ (substitute\ f\ T\ t)$
assumes $\bigwedge x\ n.\ x \in A \Longrightarrow carrier\ (f\ x) \subseteq A$
assumes $carrier\ t \subseteq A$
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x = f'\ x$
shows $xs \in paths\ (substitute\ f'\ T\ t)$
⟨*proof*⟩

lemma *substitute-cong-induct*:
assumes $\bigwedge x.\ x \in A \Longrightarrow carrier\ (f\ x) \subseteq A$
assumes $carrier\ t \subseteq A$
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x = f'\ x$
shows $substitute\ f\ T\ t = substitute\ f'\ T\ t$
⟨*proof*⟩

lemma *carrier-substitute-aux*:
assumes $xs \in paths\ (substitute\ f\ T\ t)$
assumes $carrier\ t \subseteq A$
assumes $\bigwedge x.\ x \in A \Longrightarrow carrier\ (f\ x) \subseteq A$
shows $set\ xs \subseteq A$
⟨*proof*⟩

lemma *carrier-substitute-below*:
assumes $\bigwedge x.\ x \in A \Longrightarrow carrier\ (f\ x) \subseteq A$
assumes $carrier\ t \subseteq A$
shows $carrier\ (substitute\ f\ T\ t) \subseteq A$
⟨*proof*⟩

lemma *f-nxt-eq-empty-iff*:
$f\text{-}nxt\ f\ T\ x\ x' = empty \longleftrightarrow f\ x' = empty \vee (x' = x \wedge x \in T)$
⟨*proof*⟩

lemma *substitute-T-cong′*:
assumes $xs \in paths\ (substitute\ f\ T\ t)$
assumes $\bigwedge x.\ \ (x \in T \longleftrightarrow x \in T') \vee f\ x = empty$
shows $xs \in paths\ (substitute\ f\ T'\ t)$
⟨*proof*⟩

lemma *substitute-cong-T*:
assumes $\bigwedge x.\ \ (x \in T \longleftrightarrow x \in T') \vee f\ x = empty$
shows $substitute\ f\ T = substitute\ f\ T'$
⟨*proof*⟩

lemma *carrier-substitute1*: $carrier\ t \subseteq carrier\ (substitute\ f\ T\ t)$
⟨*proof*⟩

lemma *substitute-cong*:

assumes $\bigwedge x.\ x \in carrier\ (substitute\ f\ T\ t) \Longrightarrow f\ x = f'\ x$
shows *substitute f T t = substitute f' T t*
⟨*proof*⟩

lemma *substitute-substitute*:
assumes $\bigwedge x.\ const\text{-}on\ f'\ (carrier\ (f\ x))\ empty$
shows *substitute f T (substitute f' T t) = substitute* $(\lambda\ x.\ f\ x \otimes\otimes f'\ x)\ T\ t$
⟨*proof*⟩

lemma *ttree-rest-substitute*:
assumes $\bigwedge x.\ carrier\ (f\ x) \cap S = \{\}$
shows *ttree-restr S (substitute f T t) = ttree-restr S t*
⟨*proof*⟩

An alternative characterization of substitution

inductive *substitute''* :: *('a ⇒ 'a ttree) ⇒ 'a set ⇒ 'a list ⇒ 'a list ⇒ bool* **where**
substitute''-Nil: *substitute'' f T [] []*
| *substitute''-Cons*:
zs ∈ paths (f x) ⟹ xs' ∈ interleave xs zs ⟹ substitute'' (f-nxt f T x) T xs' ys
⟹ substitute'' f T (x#xs) (x#ys)
inductive-cases *substitute''-NilE*[*elim*]: *substitute'' f T xs [] substitute'' f T [] xs*
inductive-cases *substitute''-ConsE*[*elim*]: *substitute'' f T (x#xs) ys*

lemma *substitute-substitute''*:
xs ∈ paths (substitute f T t) ⟷ (∃ xs' ∈ paths t. substitute'' f T xs' xs)
⟨*proof*⟩

lemma *paths-substitute-substitute''*:
paths (substitute f T t) = $\bigcup((\lambda\ xs\ .\ Collect\ (substitute''\ f\ T\ xs))\ `\ paths\ t)$
⟨*proof*⟩

lemma *ttree-rest-substitute2*:
assumes $\bigwedge x.\ carrier\ (f\ x) \subseteq S$
assumes *const-on f (−S) empty*
shows *ttree-restr S (substitute f T t) = substitute f T (ttree-restr S t)*
⟨*proof*⟩

end

9.2 TTree-HOLCF

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

instantiation *ttree* :: (*type*) *below*
begin

lift-definition *below-ttree* :: *'a ttree* ⇒ *'a ttree* ⇒ *bool* **is** (⊆)⟨*proof*⟩
instance⟨*proof*⟩
end

lemma *paths-mono*: $t \sqsubseteq t' \Longrightarrow$ *paths* $t \sqsubseteq$ *paths* t'
⟨*proof*⟩

lemma *paths-mono-iff*: *paths* $t \sqsubseteq$ *paths* $t' \longleftrightarrow t \sqsubseteq t'$
⟨*proof*⟩

lemma *ttree-belowI*: $(\bigwedge xs.\ xs \in$ *paths* $t \Longrightarrow xs \in$ *paths* $t') \Longrightarrow t \sqsubseteq t'$
⟨*proof*⟩

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

instance *ttree* :: (*type*) *po*
⟨*proof*⟩

lemma *is-lub-ttree*:
S <<| *Either S*
⟨*proof*⟩

lemma *lub-is-either*: *lub S* = *Either S*
⟨*proof*⟩

instance *ttree* :: (*type*) *cpo*
⟨*proof*⟩

lemma *minimal-ttree*[*simp*, *intro*!]: *empty* ⊑ *S*
⟨*proof*⟩

instance *ttree* :: (*type*) *pcpo*
⟨*proof*⟩

lemma *empty-is-bottom*: *empty* = ⊥
⟨*proof*⟩

lemma *carrier-bottom*[*simp*]: *carrier* ⊥ = {}
⟨*proof*⟩

lemma *below-anything*[*simp*]:
t ⊑ *anything*
⟨*proof*⟩

lemma *carrier-mono*: $t \sqsubseteq t' \Longrightarrow$ *carrier* $t \subseteq$ *carrier* t'
⟨*proof*⟩

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

⟨*proof*⟩

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

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

lemma *either-belowI*: $t \sqsubseteq t'' \Longrightarrow t' \sqsubseteq t'' \Longrightarrow t \oplus\oplus t' \sqsubseteq t''$
⟨*proof*⟩

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

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

lemma *both-mono1'*:
$t \sqsubseteq t' \Longrightarrow t \otimes\otimes t'' \sqsubseteq t' \otimes\otimes t''$
⟨*proof*⟩

lemma *both-mono2'*:
$t \sqsubseteq t' \Longrightarrow t'' \otimes\otimes t \sqsubseteq t'' \otimes\otimes t'$
⟨*proof*⟩

lemma *nxt-both-left*:
possible $t\ x \Longrightarrow$ *nxt* $t\ x \otimes\otimes t' \sqsubseteq$ *nxt* $(t \otimes\otimes t')\ x$
⟨*proof*⟩

lemma *nxt-both-right*:
possible $t'\ x \Longrightarrow t \otimes\otimes$ *nxt* $t'\ x \sqsubseteq$ *nxt* $(t \otimes\otimes t')\ x$
⟨*proof*⟩

lemma *substitute-mono1'*: $f \sqsubseteq f' \Longrightarrow$ *substitute* $f\ T\ t \sqsubseteq$ *substitute* $f'\ T\ t$
⟨*proof*⟩

lemma *substitute-mono2'*: $t \sqsubseteq t' \Longrightarrow$ *substitute* $f\ T\ t \sqsubseteq$ *substitute* $f\ T\ t'$
⟨*proof*⟩

lemma *substitute-above-arg*: $t \sqsubseteq$ *substitute* $f\ T\ t$
⟨*proof*⟩

lemma *ttree-contI*:
assumes $\bigwedge S.\ f\ (Either\ S) = Either\ (f\ `\ S)$
shows *cont* f
⟨*proof*⟩

lemma *ttree-contI2*:
assumes $\bigwedge x.\ paths\ (f\ x) = \bigcup (t\ `\ paths\ x)$
assumes $[] \in t\ []$
shows *cont f*
⟨*proof*⟩

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

lemma *ttree-contI3*:
assumes *cont* $(\lambda\ x.\ paths\ (f\ x))$
shows *cont f*
⟨*proof*⟩

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

lemma *cont-both1*:
cont $(\lambda\ x.\ both\ x\ y)$
⟨*proof*⟩

lemma *cont-both2*:
cont $(\lambda\ x.\ both\ y\ x)$
⟨*proof*⟩

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

lemma *cont-intersect1*:
cont $(\lambda\ x.\ intersect\ x\ y)$
⟨*proof*⟩

lemma *cont-intersect2*:
cont $(\lambda\ x.\ intersect\ y\ x)$
⟨*proof*⟩

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

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

lemma *paths-many-calls-subset*:
$t \sqsubseteq$ *many-calls x* $\otimes\otimes$ *without x t*

⟨*proof*⟩

lemma *single-below*:
[*x*] ∈ *paths t* ⟹ *single x* ⊑ *t* ⟨*proof*⟩

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

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

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

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

instance *ttree* :: (*type*) *Finite-Join-cpo*
⟨*proof*⟩

lemma *ttree-join-is-either*:
t ⊔ *t*′ = *t* ⊕⊕ *t*′
⟨*proof*⟩

lemma *ttree-join-transfer*[*transfer-rule*]: *rel-fun* (*pcr-ttree* (=)) (*rel-fun* (*pcr-ttree* (=)) (*pcr-ttree* (=))) (∪) (⊔)
⟨*proof*⟩

lemma *ttree-restr-join*[*simp*]:
ttree-restr S (*t* ⊔ *t*′) = *ttree-restr S t* ⊔ *ttree-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 *exp* :: *exp* ⇒ *'a*::*cpo* → *'b*::*pcpo*
begin

fun *AnalBinds* :: *heap* ⇒ (*var* ⇒ *'a*$_\perp$) → (*var* ⇒ *'b*)
  where *AnalBinds* [] = (Λ *ae*. ⊥)
    | *AnalBinds* ((*x*,*e*)#Γ) = (Λ *ae*. (*AnalBinds* Γ·*ae*)(*x* := *fup*·(*exp e*)·(*ae x*)))

lemma *AnalBinds-Nil-simp*[*simp*]: *AnalBinds* []·*ae* = ⊥ ⟨*proof*⟩

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

lemmas *AnalBinds.simps*[*simp del*]

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

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

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

lemma *AnalBinds-delete-bot*: *ae x* = ⊥ ⟹ *AnalBinds* (*delete x* Γ)·*ae* = *AnalBinds* Γ·*ae*
  ⟨*proof*⟩

lemma *AnalBinds-delete-below*: *AnalBinds* (*delete x* Γ)·*ae* ⊑ *AnalBinds* Γ·*ae*
  ⟨*proof*⟩

lemma *AnalBinds-delete-lookup*[*simp*]: (*AnalBinds* (*delete x* Γ)·*ae*) *x* = ⊥
  ⟨*proof*⟩

lemma *AnalBinds-delete-to-fun-upd*: *AnalBinds* (*delete x* Γ)·*ae* = (*AnalBinds* Γ·*ae*)(*x* := ⊥)
  ⟨*proof*⟩

lemma *edom-AnalBinds*: *edom* (*AnalBinds* Γ·*ae*) ⊆ *domA* Γ ∩ *edom ae*
  ⟨*proof*⟩

end

end

10.2 TTreeAnalysisSig

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

locale *TTreeAnalysis* =
fixes *Texp* :: *exp* ⇒ *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*⟩

lemma *no-call-in-tail*: *no-call-in-path x* (*tl p*) ⟷ (*no-call-in-path x p* ∨ *one-call-in-path x p* ∧ *hd p* = *x*)
⟨*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*:

$\neg$ *one-call-in-path x p* $\Longrightarrow$ $x \in$ *set p*
⟨*proof*⟩

lemma *no-call-in-path*[*eqvt*]: *no-call-in-path p x* $\Longrightarrow$ *no-call-in-path* ($\pi \cdot p$) ($\pi \cdot x$)
⟨*proof*⟩

lemma *one-call-in-path*[*eqvt*]: *one-call-in-path p x* $\Longrightarrow$ *one-call-in-path* ($\pi \cdot p$) ($\pi \cdot x$)
⟨*proof*⟩

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

lemma *pathCard-Nil*[*simp*]: *pathCard* [] = $\bot$
⟨*proof*⟩

lemma *pathCard-Cons*[*simp*]: *pathCard* (*x*#*xs*) *x* = *two-add*·*once*·(*pathCard xs x*)
⟨*proof*⟩

lemma *pathCard-Cons-other*[*simp*]: $x' \neq x$ $\Longrightarrow$ *pathCard* (*x*#*xs*) x' = *pathCard xs* x'
⟨*proof*⟩

lemma *no-call-in-path-filter*[*simp*]: *no-call-in-path x* [*x*←*xs* . $x \in S$] $\longleftrightarrow$ *no-call-in-path x xs* $\vee$ $x \notin S$
⟨*proof*⟩

lemma *one-call-in-path-filter*[*simp*]: *one-call-in-path x* [*x*←*xs* . $x \in S$] $\longleftrightarrow$ *one-call-in-path x xs* $\vee$ $x \notin S$
⟨*proof*⟩

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

lemma *paths-Card-above*:
$p \in ps$ $\Longrightarrow$ *pathCard p* $\sqsubseteq$ *pathsCard ps*
⟨*proof*⟩

lemma *pathsCard-below*:
assumes $\bigwedge p.\ p \in ps \Longrightarrow$ *pathCard p* $\sqsubseteq$ *ce*
shows *pathsCard ps* $\sqsubseteq$ *ce*
⟨*proof*⟩

lemma *pathsCard-mono*:
$ps \subseteq ps'$ $\Longrightarrow$ *pathsCard ps* $\sqsubseteq$ *pathsCard ps'*
⟨*proof*⟩

lemmas *pathsCard-mono'* = *pathsCard-mono*[*folded below-set-def*]

lemma *record-call-pathsCard*:
pathsCard ({ *tl p* | *p* . *p* ∈ *fs* ∧ *hd p* = *x*}) ⊑ *record-call x*·(*pathsCard fs*)
⟨*proof*⟩

lemma *pathCards-noneD*:
pathsCard ps x = *none* ⟹ *x* ∉ ⋃(*set* ‘ *ps*)
⟨*proof*⟩

lemma *cont-pathsCard*[*THEN cont-compose*, *cont2cont*, *simp*]:
cont pathsCard
⟨*proof*⟩

lemma *pathsCard-eqvt*[*eqvt*]: π · *pathsCard ps x* = *pathsCard* (π · *ps*) (π · *x*)
⟨*proof*⟩

lemma *edom-pathsCard*[*simp*]: *edom* (*pathsCard ps*) = ⋃(*set* ‘ *ps*)
⟨*proof*⟩

lemma *env-restr-pathsCard*[*simp*]: *pathsCard ps f*|‘ *S* = *pathsCard* (*filter* (λ *x*. *x* ∈ *S*) ‘ *ps*)
⟨*proof*⟩

end

10.4 TTreeAnalysisSpec

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

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

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

locale *TTreeAnalysisCardinalityHeap* =
TTreeAnalysisSafe + *AbstractAnalysisLetSafe* +
fixes *Theap* :: *heap* ⇒ *exp* ⇒ *Arity* → *var ttree*
assumes *carrier-Fheap*: *carrier* (*Theap* Γ *e*·*a*) = *edom* (*Aheap* Γ *e*·*a*)
assumes *Theap-thunk*: *x* ∈ *thunks* Γ ⟹ *p* ∈ *paths* (*Theap* Γ *e*·*a*) ⟹ ¬ *one-call-in-path x p* ⟹ (*Aheap* Γ *e*·*a*) *x* = *up*·*0*

assumes *Theap-substitute*: *ttree-restr* (*domA* Δ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)) $\sqsubseteq$ *Theap* Δ *e*·*a*
assumes *Texp-Let*: *ttree-restr* (− *domA* Δ) (*substitute* (*FBinds* Δ·(*Aheap* Δ *e*·*a*)) (*thunks* Δ) (*Texp e*·*a*)) $\sqsubseteq$ *Texp* (*Terms.Let* Δ *e*)·*a*

end

10.5 TTreeImplCardinality

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

context *TTreeAnalysis*
begin

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

fun *Fstack* :: *Arity list* ⇒ *stack* ⇒ *var ttree*
  where *Fstack* - [] = ⊥
  | *Fstack* (*a*#*as*) (*Alts e1 e2* # *S*) = (*Texp e1*·*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-edom-disj*: *const-on f S empty* ⟷ *edom f* ∩ *S* = {}
⟨*proof*⟩

context *TTreeAnalysisCarrier*
begin
lemma *carrier-Fstack*: *carrier* (*Fstack as S*) ⊆ *fv S*
⟨*proof*⟩

lemma *carrier-FBinds*: *carrier* ((*FBinds* Γ·*ae*) *x*) ⊆ *fv* Γ
⟨*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 Γ *e* = (Λ *a*. *pathsCard* (*paths* (*Theap* Γ *e*·*a*)))

lemma *cHeap-simp*: (*cHeap* Γ *e*)·*a* = *pathsCard* (*paths* (*Theap* Γ *e*·*a*))
⟨*proof*⟩

sublocale *CardinalityHeap cHeap*⟨*proof*⟩

sublocale *CardinalityHeapSafe cHeap Aheap*

⟨*proof*⟩

sublocale *CardinalityPrognosisEdom prognosis*
⟨*proof*⟩

sublocale *CardinalityPrognosisLet prognosis cHeap*
⟨*proof*⟩

sublocale *CardinalityPrognosisSafe prognosis cHeap Aheap Aexp* ⟨*proof*⟩
end

end

11 Co-Call Graphs

11.1 CoCallGraph

theory *CoCallGraph*
imports *Launchbury.Vars Launchbury.HOLCF−Join−Classes Launchbury.HOLCF−Utils Set−Cpo*
begin

default-sort *type*

typedef *CoCalls* = {*G* :: (*var* × *var*) *set*. *sym G*}
morphisms *Rep-CoCall Abs-CoCall*
⟨*proof*⟩

setup-lifting *type-definition-CoCalls*

instantiation *CoCalls* :: *po*
begin
lift-definition *below-CoCalls* :: *CoCalls* ⇒ *CoCalls* ⇒ *bool* **is** (⊆)⟨*proof*⟩
instance
⟨*proof*⟩
end

lift-definition *coCallsLub* :: *CoCalls set* ⇒ *CoCalls* **is** λ *S*. ⋃ *S*
⟨*proof*⟩

lemma *coCallsLub-is-lub*: *S* <<| *coCallsLub S*
⟨*proof*⟩

instance *CoCalls* :: *cpo*
⟨*proof*⟩

lemma *ccLubTransfer*[*transfer-rule*]: (*rel-set pcr-CoCalls* ===> *pcr-CoCalls*) *Union lub*
⟨*proof*⟩

lift-definition *is-cc-lub* :: *CoCalls set* ⇒ *CoCalls* ⇒ *bool* **is** (λ *S x* . *x* = *Union S*)⟨*proof*⟩

lemma *ccis-lubTransfer*[*transfer-rule*]: (*rel-set pcr-CoCalls* ===> *pcr-CoCalls* ===> (=)) (λ *S x* . *x* = *Union S*) (<<|)
⟨*proof*⟩

lift-definition *coCallsJoin* :: *CoCalls* ⇒ *CoCalls* ⇒ *CoCalls* **is** (∪)
⟨*proof*⟩

lemma *ccJoinTransfer*[*transfer-rule*]: (*pcr-CoCalls* ===> *pcr-CoCalls* ===> *pcr-CoCalls*) (∪) (⊔)
⟨*proof*⟩

lift-definition *ccEmpty* :: *CoCalls* **is** {} ⟨*proof*⟩

lemma *ccEmpty-below*[*simp*]: *ccEmpty* ⊑ *G*
⟨*proof*⟩

instance *CoCalls* :: *pcpo*
⟨*proof*⟩

lemma *ccBotTransfer*[*transfer-rule*]: *pcr-CoCalls* {} ⊥
⟨*proof*⟩

lemma *cc-lub-below-iff*:
fixes *G* :: *CoCalls*
shows *lub X* ⊑ *G* ⟷ (∀ *G′*∈*X*. *G′* ⊑ *G*)
⟨*proof*⟩

lift-definition *ccField* :: *CoCalls* ⇒ *var set* **is** *Field*⟨*proof*⟩

lemma *ccField-nil*[*simp*]: *ccField* ⊥ = {}
⟨*proof*⟩

lift-definition
inCC :: *var* ⇒ *var* ⇒ *CoCalls* ⇒ *bool* (‹-−−-∈-› [*1000*, *1000*, *900*] *900*)
is λ *x y s*. (*x*,*y*) ∈ *s*⟨*proof*⟩

abbreviation
notInCC :: *var* ⇒ *var* ⇒ *CoCalls* ⇒ *bool* (‹-−−-∉-› [*1000*, *1000*, *900*] *900*)
where *x*−−*y*∉*S* ≡ ¬ *x*−−*y*∈*S*

lemma *notInCC-bot*[*simp*]: *x*−−*y*∈⊥ ⟷ *False*
⟨*proof*⟩

lemma *below-CoCallsI*:
(⋀ *x y*. *x*−−*y*∈*G* ⟹ *x*−−*y*∈*G′*) ⟹ *G* ⊑ *G′*
⟨*proof*⟩

lemma *CoCalls-eqI*:
(⋀ *x y*. *x*−−*y*∈*G* ⟷ *x*−−*y*∈*G*′) ⟹ *G* = *G*′
⟨*proof*⟩

lemma *in-join*[*simp*]:
x−−*y* ∈ (*G*⊔*G*′) ⟷ *x*−−*y*∈*G* ∨ *x*−−*y*∈*G*′
⟨*proof*⟩

lemma *in-lub*[*simp*]: *x*−−*y*∈(*lub S*) ⟷ (∃ *G*∈*S*. *x*−−*y*∈*G*)
⟨*proof*⟩

lemma *in-CoCallsLubI*:
x−−*y*∈*G* ⟹ *G* ∈ *S* ⟹ *x*−−*y*∈*lub S*
⟨*proof*⟩

lemma *adm-not-in*[*simp*]:
assumes *cont t*
shows *adm* (λ*a*. *x*−−*y*∉*t a*)
⟨*proof*⟩

lift-definition *cc-delete* :: *var* ⇒ *CoCalls* ⇒ *CoCalls*
is λ *z*. *Set.filter* (λ (*x*,*y*) . *x* ≠ *z* ∧ *y* ≠ *z*)
⟨*proof*⟩

lemma *ccField-cc-delete*: *ccField* (*cc-delete x S*) ⊆ *ccField S* − {*x*}
⟨*proof*⟩

lift-definition *ccProd* :: *var set* ⇒ *var set* ⇒ *CoCalls* (**infixr** ‹*G*×› *90*)
is λ *S1 S2*. *S1* × *S2* ∪ *S2* × *S1*
⟨*proof*⟩

lemma *ccProd-empty*[*simp*]: {} *G*× *S* = ⊥ ⟨*proof*⟩

lemma *ccProd-empty*′[*simp*]: *S G*× {} = ⊥ ⟨*proof*⟩

lemma *ccProd-union2*[*simp*]: *S G*× (*S*′ ∪ *S*′′) = *S G*× *S*′ ⊔ *S G*× *S*′′
⟨*proof*⟩

lemma *ccProd-Union2*[*simp*]: *S G*× ⋃*S*′ = (⨆ *X*∈*S*′. *ccProd S X*)
⟨*proof*⟩

lemma *ccProd-Union2*′[*simp*]: *S G*× (⋃*X*∈*S*′. *f X*) = (⨆ *X*∈*S*′. *ccProd S* (*f X*))
⟨*proof*⟩

lemma *in-ccProd*[*simp*]: *x*−−*y*∈(*S G*× *S*′) = (*x* ∈ *S* ∧ *y* ∈ *S*′ ∨ *x* ∈ *S*′ ∧ *y* ∈ *S*)
⟨*proof*⟩

lemma *ccProd-union1*[*simp*]: (*S*′ ∪ *S*′′) *G*× *S* = *S*′ *G*× *S* ⊔ *S*′′ *G*× *S*

⟨*proof*⟩

lemma *ccProd-insert2*: $S\ G\times\ insert\ x\ S' = S\ G\times\ \{x\} \sqcup S\ G\times\ S'$
⟨*proof*⟩

lemma *ccProd-insert1*: $insert\ x\ S'\ G\times\ S = \{x\}\ G\times\ S \sqcup S'\ G\times\ S$
⟨*proof*⟩

lemma *ccProd-mono1*: $S' \subseteq S'' \Longrightarrow S'\ G\times\ S \sqsubseteq S''\ G\times\ S$
⟨*proof*⟩

lemma *ccProd-mono2*: $S' \subseteq S'' \Longrightarrow S\ G\times\ S' \sqsubseteq S\ G\times\ S''$
⟨*proof*⟩

lemma *ccProd-mono*: $S \subseteq S' \Longrightarrow T \subseteq T' \Longrightarrow S\ G\times\ T \sqsubseteq S'\ G\times\ T'$
⟨*proof*⟩

lemma *ccProd-comm*: $S\ G\times\ S' = S'\ G\times\ S$ ⟨*proof*⟩

lemma *ccProd-belowI*:
$(\bigwedge x\ y.\ x \in S \Longrightarrow y \in S' \Longrightarrow x{-}{-}y{\in}G) \Longrightarrow S\ G\times\ S' \sqsubseteq G$
⟨*proof*⟩

lift-definition *cc-restr* :: *var set* ⇒ *CoCalls* ⇒ *CoCalls*
is $\lambda\ S.$ *Set.filter* $(\lambda\ (x,y)\ .\ x \in S \wedge y \in S)$
⟨*proof*⟩

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

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

lemma *ccField-cc-restr*: *ccField* $(G\ G|`\ S) \subseteq$ *ccField* $G \cap S$
⟨*proof*⟩

lemma *cc-restr-empty*: *ccField* $G \subseteq -\ S \Longrightarrow G\ G|`\ S = \bot$
⟨*proof*⟩

lemma *cc-restr-empty-set*[*simp*]: *cc-restr* $\{\}\ G = \bot$
⟨*proof*⟩

lemma *cc-restr-noop*[*simp*]: *ccField* $G \subseteq S \Longrightarrow$ *cc-restr* $S\ G = G$
⟨*proof*⟩

lemma *cc-restr-bot*[*simp*]: *cc-restr* $S\ \bot = \bot$
⟨*proof*⟩

lemma *ccRestr-ccDelete*[*simp*]: *cc-restr* $(-\{x\})\ G =$ *cc-delete* $x\ G$

⟨*proof*⟩

lemma *cc-restr-join*[*simp*]:
cc-restr S (*G* ⊔ *G'*) = *cc-restr S G* ⊔ *cc-restr S G'*
⟨*proof*⟩

lemma *cont-cc-restr*: *cont* (*cc-restr S*)
⟨*proof*⟩

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

lemma *cc-restr-mono1*:
S ⊆ *S'* ⟹ *cc-restr S G* ⊑ *cc-restr S' G* ⟨*proof*⟩

lemma *cc-restr-mono2*:
G ⊑ *G'* ⟹ *cc-restr S G* ⊑ *cc-restr S G'* ⟨*proof*⟩

lemma *cc-restr-below-arg*:
cc-restr S G ⊑ *G* ⟨*proof*⟩

lemma *cc-restr-lub*[*simp*]:
cc-restr S (*lub X*) = (⨆ *G*∈*X*. *cc-restr S G*) ⟨*proof*⟩

lemma *elem-to-ccField*: *x*−−*y*∈*G* ⟹ *x* ∈ *ccField G* ∧ *y* ∈ *ccField G*
⟨*proof*⟩

lemma *ccField-to-elem*: *x* ∈ *ccField G* ⟹ ∃ *y*. *x*−−*y*∈*G*
⟨*proof*⟩

lemma *cc-restr-intersect*: *ccField G* ∩ ((*S* − *S'*) ∪ (*S'* − *S*)) = {} ⟹ *cc-restr S G* = *cc-restr S' G*
⟨*proof*⟩

lemma *cc-restr-cc-restr*[*simp*]: *cc-restr S* (*cc-restr S' G*) = *cc-restr* (*S* ∩ *S'*) *G*
⟨*proof*⟩

lemma *cc-restr-twist*: *cc-restr S* (*cc-restr S' G*) = *cc-restr S'* (*cc-restr S G*)
⟨*proof*⟩

lemma *cc-restr-cc-delete-twist*: *cc-restr x* (*cc-delete S G*) = *cc-delete S* (*cc-restr x G*)
⟨*proof*⟩

lemma *cc-restr-ccProd*[*simp*]:
cc-restr S (*ccProd* S_1 S_2) = *ccProd* (S_1 ∩ *S*) (S_2 ∩ *S*)
⟨*proof*⟩

lemma *ccProd-below-cc-restr*:
ccProd S S' ⊑ *cc-restr S'' G* ⟷ *ccProd S S'* ⊑ *G* ∧ (*S* = {} ∨ *S'* = {} ∨ *S* ⊆ *S''* ∧ *S'* ⊆ *S''*)

⟨*proof*⟩

lemma *cc-restr-eq-subset*: $S \subseteq S' \Longrightarrow$ *cc-restr* S' G = *cc-restr* S' *G2* $\Longrightarrow$ *cc-restr* S G = *cc-restr* S *G2*
⟨*proof*⟩

definition *ccSquare* (‹-²› [*80*] *80*)
where S^2 = *ccProd* S S

lemma *ccField-ccSquare*[*simp*]: *ccField* (S^2) = S
⟨*proof*⟩

lemma *below-ccSquare*[*iff*]: ($G \sqsubseteq S^2$) = (*ccField* $G \subseteq S$)
⟨*proof*⟩

lemma *cc-restr-ccSquare*[*simp*]: (S'^2) $G|`$ S = ($S' \cap S$)²
⟨*proof*⟩

lemma *ccSquare-empty*[*simp*]: {}² = ⊥
⟨*proof*⟩

lift-definition *ccNeighbors* :: *var* ⇒ *CoCalls* ⇒ *var set*
is λ x G. {y .(y,x) ∈ G ∨ (x,y) ∈ G}⟨*proof*⟩

lemma *ccNeighbors-bot*[*simp*]: *ccNeighbors* x ⊥ = {} ⟨*proof*⟩

lemma *cont-ccProd1*:
cont (λ S. *ccProd* S S')
⟨*proof*⟩

lemma *cont-ccProd2*:
cont (λ S'. *ccProd* S S')
⟨*proof*⟩

lemmas *cont-compose2*[*OF cont-ccProd1 cont-ccProd2*, *simp*, *cont2cont*]

lemma *cont-ccNeighbors*[*THEN cont-compose*, *cont2cont*, *simp*]:
cont (λy. *ccNeighbors* x y)
⟨*proof*⟩

lemma *ccNeighbors-join*[*simp*]: *ccNeighbors* x (G ⊔ G') = *ccNeighbors* x G ∪ *ccNeighbors* x G'
⟨*proof*⟩

lemma *ccNeighbors-ccProd*:
ccNeighbors x (*ccProd* S S') = (*if* $x \in S$ *then* S' *else* {}) ∪ (*if* $x \in S'$ *then* S *else* {})
⟨*proof*⟩

lemma *ccNeighbors-ccSquare*:
ccNeighbors x (ccSquare S) = (if x ∈ S then S else {})
⟨*proof*⟩

lemma *ccNeighbors-cc-restr*[*simp*]:
ccNeighbors x (cc-restr S G) = (if x ∈ S then ccNeighbors x G ∩ S else {})
⟨*proof*⟩

lemma *ccNeighbors-mono*:
G ⊑ G' ⟹ ccNeighbors x G ⊆ ccNeighbors x G'
⟨*proof*⟩

lemma *subset-ccNeighbors*:
S ⊆ ccNeighbors x G ⟷ ccProd {x} S ⊑ G
⟨*proof*⟩

lemma *elem-ccNeighbors*[*simp*]:
y ∈ ccNeighbors x G ⟷ (y−−x∈G)
⟨*proof*⟩

lemma *ccNeighbors-ccField*:
ccNeighbors x G ⊆ ccField G ⟨*proof*⟩

lemma *ccNeighbors-disjoint-empty*[*simp*]:
ccNeighbors x G = {} ⟷ x ∉ ccField G
⟨*proof*⟩

instance *CoCalls* :: *Join-cpo*
⟨*proof*⟩

lemma *ccNeighbors-lub*[*simp*]: *ccNeighbors x (lub Gs) = lub (ccNeighbors x ‘ Gs)*
⟨*proof*⟩

inductive *list-pairs* :: *'a list ⇒ ('a × 'a) ⇒ bool*
where *list-pairs xs p ⟹ list-pairs (x#xs) p*
| *y ∈ set xs ⟹ list-pairs (x#xs) (x,y)*

lift-definition *ccFromList* :: *var list ⇒ CoCalls* **is** *λ xs. {(x,y). list-pairs xs (x,y) ∨ list-pairs xs (y,x)}*
⟨*proof*⟩

lemma *ccFromList-Nil*[*simp*]: *ccFromList [] = ⊥*
⟨*proof*⟩

lemma *ccFromList-Cons*[*simp*]: *ccFromList (x#xs) = ccProd {x} (set xs) ⊔ ccFromList xs*
⟨*proof*⟩

lemma *ccFromList-append*[*simp*]: *ccFromList (xs@ys) = ccFromList xs ⊔ ccFromList ys ⊔ ccProd (set xs) (set ys)*

⟨*proof*⟩

lemma *ccFromList-filter*[*simp*]:
ccFromList (*filter P xs*) = *cc-restr* {*x*. *P x*} (*ccFromList xs*)
⟨*proof*⟩

lemma *ccFromList-replicate*[*simp*]: *ccFromList* (*replicate n x*) = (*if n* ≤ *1 then* ⊥ *else ccProd* {*x*} {*x*})
⟨*proof*⟩

definition *ccLinear* :: *var set* ⇒ *CoCalls* ⇒ *bool*
where *ccLinear S G* = (∀ *x*∈*S*. ∀ *y*∈*S*. *x*−−*y*∉*G*)

lemma *ccLinear-bottom*[*simp*]:
ccLinear S ⊥
⟨*proof*⟩

lemma *ccLinear-empty*[*simp*]:
ccLinear {} *G*
⟨*proof*⟩

lemma *ccLinear-lub*[*simp*]:
ccLinear S (*lub X*) = (∀ *G*∈*X*. *ccLinear S G*)
⟨*proof*⟩

lemma *ccLinear-cc-restr*[*intro*]:
ccLinear S G ⟹ *ccLinear S* (*cc-restr S*′ *G*)
⟨*proof*⟩

lemma *ccLinear-join*[*simp*]:
ccLinear S (*G* ⊔ *G*′) ⟷ *ccLinear S G* ∧ *ccLinear S G*′
⟨*proof*⟩

lemma *ccLinear-ccProd*[*simp*]:
ccLinear S (*ccProd S*$_1$ *S*$_2$) ⟷ *S*$_1$ ∩ *S* = {} ∨ *S*$_2$ ∩ *S* = {}
⟨*proof*⟩

lemma *ccLinear-mono1*: *ccLinear S*′ *G* ⟹ *S* ⊆ *S*′ ⟹ *ccLinear S G*
⟨*proof*⟩

lemma *ccLinear-mono2*: *ccLinear S G*′ ⟹ *G* ⊑ *G*′ ⟹ *ccLinear S G*
⟨*proof*⟩

lemma *ccField-join*[*simp*]:
ccField (*G* ⊔ *G*′) = *ccField G* ∪ *ccField G*′ ⟨*proof*⟩

lemma *ccField-lub*[*simp*]:
ccField (*lub S*) = ⋃(*ccField* ‘ *S*) ⟨*proof*⟩

lemma *ccField-ccProd*:
ccField (*ccProd S S′*) = (*if S* = {} *then* {} *else if S′* = {} *then* {} *else* *S* ∪ *S′*)
⟨*proof*⟩

lemma *ccField-ccProd-subset*:
ccField (*ccProd S S′*) ⊆ *S* ∪ *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* ⇒ *CoCalls* ⇒ *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* ♯∗ *S* **and** [*simp*]: *finite S*
shows *cc-restr S* (*p* ∙ *G*) = *cc-restr S G*
⟨*proof*⟩

lemma *inCC-eqvt*[*eqvt*]: *π* ∙ (*x*−−*y*∈*G*) = (*π*∙*x*)−−(*π*∙*y*)∈(*π*∙*G*)
⟨*proof*⟩
lemma *cc-restr-eqvt*[*eqvt*]: *π* ∙ *cc-restr S G* = *cc-restr* (*π* ∙ *S*) (*π* ∙ *G*)
⟨*proof*⟩

lemma *ccProd-eqvt*[*eqvt*]: $\pi \cdot ccProd\ S\ S' = ccProd\ (\pi \cdot S)\ (\pi \cdot\ S')$
⟨*proof*⟩
lemma *ccSquare-eqvt*[*eqvt*]: $\pi \cdot ccSquare\ S = ccSquare\ (\pi \cdot S)$
⟨*proof*⟩
lemma *ccNeighbors-eqvt*[*eqvt*]: $\pi \cdot ccNeighbors\ S\ G = ccNeighbors\ (\pi \cdot S)\ (\pi \cdot G)$
⟨*proof*⟩

end

12 Co-Call Cardinality Analysis

12.1 CoCallAnalysisSig

theory *CoCallAnalysisSig*
imports *Launchbury.Terms Arity CoCallGraph*
begin

locale *CoCallAnalysis* =
fixes *ccExp* :: *exp* ⇒ *Arity* → *CoCalls*
begin
abbreviation *ccExp-syn* (‹$\mathcal{G}$_›)
where $\mathcal{G}_a \equiv (\lambda e.\ ccExp\ e{\cdot}a)$
abbreviation *ccExp-bot-syn* (‹$\mathcal{G}^\perp$_›)
where $\mathcal{G}^\perp{}_a \equiv (\lambda e.\ fup{\cdot}(ccExp\ e){\cdot}a)$
end

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

end

12.2 CoCallAnalysisBinds

theory *CoCallAnalysisBinds*
imports *CoCallAnalysisSig AEnv AList−Utils−HOLCF Arity−Nominal CoCallGraph−Nominal*
begin

context *CoCallAnalysis*
begin
definition *ccBind* :: *var* ⇒ *exp* ⇒ ((*AEnv* × *CoCalls*) → *CoCalls*)
where *ccBind v e* = (Λ (*ae*, *G*). *if* (*v*−−*v*∉*G*) ∨ ¬ *isVal e then cc-restr* (*fv e*) (*fup*·(*ccExp e*)·(*ae v*)) *else ccSquare* (*fv e*))

lemma *ccBind-eq*:
ccBind v e·(*ae*, *G*) = (*if v*−−*v*∉*G* ∨ ¬ *isVal e then* $\mathcal{G}^\perp{}_{ae\ v}\ e\ G$|‘ *fv e else* (*fv e*)²)

⟨*proof*⟩

lemma *ccBind-strict*[*simp*]: *ccBind v e* · ⊥ = ⊥
⟨*proof*⟩

lemma *ccField-ccBind*: *ccField* (*ccBind v e*·(*ae*,*G*)) ⊆ *fv e*
⟨*proof*⟩

definition *ccBinds* :: *heap* ⇒ ((*AEnv* × *CoCalls*) → *CoCalls*)
where *ccBinds* Γ = (Λ *i*. (⨆ *v*↦*e*∈*map-of* Γ. *ccBind v e*·*i*))

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

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

lemma *ccBindsExtra-eq*: *ccBindsExtra* Γ·(*ae*,*G*) =

$G \sqcup$ *ccBinds* $\Gamma{\cdot}(ae, G) \sqcup (\bigsqcup x{\mapsto}e{\in}$*map-of* Γ. *fv e* $G\times$ *ccNeighbors x* $G)$
⟨*proof*⟩

lemma *ccBindsExtra-strict*[*simp*]: *ccBindsExtra* $\Gamma \cdot \bot = \bot$
⟨*proof*⟩

lemma *ccField-ccBindsExtra*:
ccField (*ccBindsExtra* $\Gamma{\cdot}(ae, G)$) $\subseteq$ *fv* $\Gamma \cup$ *ccField G*
⟨*proof*⟩

end

lemma *ccBind-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallAnalysis.ccBind cccExp x e*) = *CoCallAnalysis.ccBind* (π $\cdot$ *cccExp*) ($\pi \cdot x$) ($\pi \cdot e$)
⟨*proof*⟩

lemma *ccBinds-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallAnalysis.ccBinds cccExp* Γ) = *CoCallAnalysis.ccBinds* (π $\cdot$ *cccExp*) ($\pi \cdot \Gamma$)
⟨*proof*⟩

lemma *ccBindsExtra-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallAnalysis.ccBindsExtra cccExp* Γ) = *CoCallAnalysis.ccBindsExtra* ($\pi \cdot$ *cccExp*) ($\pi \cdot \Gamma$)
⟨*proof*⟩

lemma *ccBind-cong*[*fundef-cong*]:
cccexp1 e = *cccexp2 e* $\Longrightarrow$ *CoCallAnalysis.ccBind cccexp1 x e* = *CoCallAnalysis.ccBind cccexp2 x e*
⟨*proof*⟩

lemma *ccBinds-cong*[*fundef-cong*]:
⟦ ($\bigwedge$ *e*. *e* $\in$ *snd* ‘ *set heap2* $\Longrightarrow$ *cccexp1 e* = *cccexp2 e*); *heap1* = *heap2* ⟧
$\Longrightarrow$ *CoCallAnalysis.ccBinds cccexp1 heap1* = *CoCallAnalysis.ccBinds cccexp2 heap2*
⟨*proof*⟩

lemma *ccBindsExtra-cong*[*fundef-cong*]:
⟦ ($\bigwedge$ *e*. *e* $\in$ *snd* ‘ *set heap2* $\Longrightarrow$ *cccexp1 e* = *cccexp2 e*); *heap1* = *heap2* ⟧
$\Longrightarrow$ *CoCallAnalysis.ccBindsExtra cccexp1 heap1* = *CoCallAnalysis.ccBindsExtra cccexp2 heap2*
⟨*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* + *CoCallAnalyisHeap* + *ArityAnalysisLetSafe* +
  assumes *ccExp-App*: *ccExp e·(inc·a)* ⊔ *ccProd* {*x*} (*insert x* (*fv e*)) ⊑ *ccExp* (*App e x*)·*a*
  assumes *ccExp-Lam*: *cc-restr* (*fv* (*Lam* [*y*]. *e*)) (*ccExp e·*(*pred·n*)) ⊑ *ccExp* (*Lam* [*y*]. *e*)·*n*
  assumes *ccExp-subst*: *x* ∉ *S* ⟹ *y* ∉ *S* ⟹ *cc-restr S* (*ccExp e*[*y*::=*x*]·*a*) ⊑ *cc-restr S* (*ccExp e·a*)
  assumes *ccExp-pap*: *isVal e* ⟹ *ccExp e·0* = *ccSquare* (*fv e*)
  assumes *ccExp-Let*: *cc-restr* (−*domA* Γ) (*ccHeap* Γ *e·a*) ⊑ *ccExp* (*Let* Γ *e*)·*a*
  assumes *ccExp-IfThenElse*: *ccExp scrut·0* ⊔ (*ccExp e1·a* ⊔ *ccExp e2·a*) ⊔ *ccProd* (*edom* (*Aexp scrut·0*)) (*edom* (*Aexp e1·a*) ∪ *edom* (*Aexp e2·a*)) ⊑ *ccExp* (*scrut ? e1* : *e2*)·*a*

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

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

end

12.5 CoCallFix

theory *CoCallFix*
imports *CoCallAnalysisSig CoCallAnalysisBinds ArityAnalysisSig Launchbury.Env−Nominal ArityAnalysisFix*
begin

locale *CoCallArityAnalysis* =
  fixes *cccExp* :: *exp* ⇒ (*Arity* → *AEnv* × *CoCalls*)
begin

definition *Aexp* :: *exp* ⇒ (*Arity* → *AEnv*)
  where *Aexp e* = (Λ *a*. *fst* (*cccExp e* · *a*))

sublocale *ArityAnalysis Aexp*⟨*proof*⟩

abbreviation *Aexp-syn'* (‹$\mathcal{A}$_›) **where** $\mathcal{A}_a \equiv$ (λ*e*. *Aexp e*·*a*)
abbreviation *Aexp-bot-syn'* (‹$\mathcal{A}^\perp$_›) **where** $\mathcal{A}^\perp{}_a \equiv$ (λ*e*. *fup*·(*Aexp e*)·*a*)

lemma *Aexp-eq*:
  $\mathcal{A}_a$ *e* = *fst* (*cccExp e* · *a*)
  ⟨*proof*⟩

lemma *fup-Aexp-eq*:
  fup·(*Aexp e*)·*a* = *fst* (*fup*·(*cccExp e*)·*a*)
  ⟨*proof*⟩

definition *CCexp* :: *exp* ⇒ (*Arity* → *CoCalls*) **where** *CCexp* Γ = (Λ *a*. *snd* (*cccExp* Γ·*a*))
lemma *CCexp-eq*:
  CCexp e·*a* = *snd* (*cccExp e* · *a*)
  ⟨*proof*⟩

lemma *fup-CCexp-eq*:
  fup·(*CCexp e*)·*a* = *snd* (*fup*·(*cccExp e*)·*a*)
  ⟨*proof*⟩

sublocale *CoCallAnalysis CCexp*⟨*proof*⟩

definition *CCfix* :: *heap* ⇒ (*AEnv* × *CoCalls*) → *CoCalls*
  where *CCfix* Γ = (Λ *aeG*. (μ *G'*. *ccBindsExtra* Γ·(*fst aeG* , *G'*) ⊔ *snd aeG*))

lemma *CCfix-eq*:
  CCfix Γ·(*ae*,*G*) = (μ *G'*. *ccBindsExtra* Γ·(*ae*, *G'*) ⊔ *G*)
  ⟨*proof*⟩

lemma *CCfix-unroll*: *CCfix* Γ·(*ae*,*G*) = *ccBindsExtra* Γ·(*ae*, *CCfix* Γ·(*ae*,*G*)) ⊔ *G*
  ⟨*proof*⟩

lemma *fup-ccExp-restr-subst'*:
  assumes ⋀ *a*. *cc-restr S* (*CCexp e*[*x*::=*y*]·*a*) = *cc-restr S* (*CCexp e*·*a*)
  shows *cc-restr S* (*fup*·(*CCexp e*[*x*::=*y*])·*a*) = *cc-restr S* (*fup*·(*CCexp e*)·*a*)
  ⟨*proof*⟩

lemma *ccBindsExtra-restr-subst'*:
  assumes ⋀ *x' e a*. (*x'*,*e*) ∈ *set* Γ ⟹ *cc-restr S* (*CCexp e*[*x*::=*y*]·*a*) = *cc-restr S* (*CCexp e*·*a*)
  assumes *x* ∉ *S*
  assumes *y* ∉ *S*

assumes *domA* $\Gamma \subseteq S$
shows *cc-restr S* (*ccBindsExtra* $\Gamma[x::h=y]\cdot(ae,\ G)$)
$=$ *cc-restr S* (*ccBindsExtra* $\Gamma\cdot(ae\ f|`\ S$, *cc-restr S G*))
⟨*proof*⟩

lemma *ccBindsExtra-restr*:
assumes *domA* $\Gamma \subseteq S$
shows *cc-restr S* (*ccBindsExtra* $\Gamma\cdot(ae,\ G)$) $=$ *cc-restr S* (*ccBindsExtra* $\Gamma\cdot(ae\ f|`\ S$, *cc-restr S G*))
⟨*proof*⟩

lemma *CCfix-restr*:
assumes *domA* $\Gamma \subseteq S$
shows *cc-restr S* (*CCfix* $\Gamma\cdot(ae,\ G)$) $=$ *cc-restr S* (*CCfix* $\Gamma\cdot(ae\ f|`\ S$, *cc-restr S G*))
⟨*proof*⟩

lemma *ccField-CCfix*:
shows *ccField* (*CCfix* $\Gamma\cdot(ae,\ G)$) $\subseteq$ *fv* $\Gamma \cup$ *ccField G*
⟨*proof*⟩

lemma *CCfix-restr-subst′*:
assumes $\bigwedge$ *x′ e a.* (*x′*,*e*) $\in$ *set* $\Gamma \Longrightarrow$ *cc-restr S* (*CCexp* $e[x::=y]\cdot a$) $=$ *cc-restr S* (*CCexp* $e\cdot a$)
assumes $x \notin S$
assumes $y \notin S$
assumes *domA* $\Gamma \subseteq S$
shows *cc-restr S* (*CCfix* $\Gamma[x::h=y]\cdot(ae,\ G)$) $=$ *cc-restr S* (*CCfix* $\Gamma\cdot(ae\ f|`\ S$, *cc-restr S G*))
⟨*proof*⟩

end

lemma *Aexp-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.Aexp cccExp e*) $=$ *CoCallArityAnalysis.Aexp* ($\pi \cdot$ *cccExp*) ($\pi \cdot$ *e*)
⟨*proof*⟩

lemma *CCexp-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.CCexp cccExp e*) $=$ *CoCallArityAnalysis.CCexp* ($\pi \cdot$ *cccExp*) ($\pi \cdot$ *e*)
⟨*proof*⟩

lemma *CCfix-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.CCfix cccExp* Γ) $=$ *CoCallArityAnalysis.CCfix* ($\pi \cdot$ *cccExp*) ($\pi \cdot \Gamma$)
⟨*proof*⟩

lemma *ccFix-cong*[*fundef-cong*]:
⟦ ($\bigwedge$ *e. e* $\in$ *snd* ` *set heap2* $\Longrightarrow$ *cccexp1 e* $=$ *cccexp2 e*); *heap1* $=$ *heap2* ⟧
$\Longrightarrow$ *CoCallArityAnalysis.CCfix cccexp1 heap1* $=$ *CoCallArityAnalysis.CCfix cccexp2 heap2*
⟨*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*]:
⟦ (⋀ *e*. *e* ∈ *snd* ‘ *set heap2* ⟹ *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*$_\perp$
where
ABind-nonrec x e = (Λ *i*. (*if isVal e* ∨ *x*−−*x*∉(*snd i*) *then fst i x else up*·0))

lemma *ABind-nonrec-eq*:
ABind-nonrec x e·(*ae*,*G*) = (*if isVal e* ∨ *x*−−*x*∉*G then ae x else up*·0)
⟨*proof*⟩

lemma *ABind-nonrec-eqvt*[*eqvt*]: π · (*ABind-nonrec x e*) = *ABind-nonrec* (π · *x*) (π · *e*)
⟨*proof*⟩

lemma *ABind-nonrec-above-arg*:
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-eqvt'*[*eqvt*]:
$\pi \cdot$ (*Aheap-nonrec x e*) = *Aheap-nonrec* ($\pi \cdot x$) ($\pi \cdot e$)
⟨*proof*⟩

context *CoCallArityAnalysis*
begin

definition *Afix-nonrec*
where *Afix-nonrec x e* = (Λ *i*. *fup*·(*Aexp e*)·(*ABind-nonrec x e* · *i*) ⊔ *fst i*)

lemma *Afix-nonrec-eq*[*simp*]:
Afix-nonrec x e · *i* = *fup*·(*Aexp e*)·(*ABind-nonrec x e* · *i*) ⊔ *fst i*
⟨*proof*⟩

definition *CCfix-nonrec*
where *CCfix-nonrec x e* = (Λ *i*. *ccBind x e* · (*Aheap-nonrec x e*·*i*, *snd i*) ⊔ *ccProd* (*fv e*) (*ccNeighbors x* (*snd i*) − (*if isVal e then* {} *else* {*x*})) ⊔ *snd i*)

lemma *CCfix-nonrec-eq*[*simp*]:
CCfix-nonrec x e · *i* = *ccBind x e*·(*Aheap-nonrec x e*·*i*, *snd i*) ⊔ *ccProd* (*fv e*) (*ccNeighbors x* (*snd i*) − (*if isVal e then* {} *else* {*x*})) ⊔ *snd i*
⟨*proof*⟩

definition *cccFix-nonrec* :: *var* ⇒ *exp* ⇒ ((*AEnv* × *CoCalls*) → (*AEnv* × *CoCalls*))
where *cccFix-nonrec x e* = (Λ *i*. (*Afix-nonrec x e* ·*i* , *CCfix-nonrec x e* ·*i*))

lemma *cccFix-nonrec-eq*[*simp*]:
cccFix-nonrec x e·*i* = (*Afix-nonrec x e* ·*i* , *CCfix-nonrec x e* ·*i*)
⟨*proof*⟩

end

lemma *AFix-nonrec-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.Afix-nonrec cccExp x e*) = *CoCallArityAnalysis.Afix-nonrec* ($\pi \cdot$ *cccExp*) ($\pi \cdot x$) ($\pi \cdot e$)
⟨*proof*⟩

lemma *CCFix-nonrec-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.CCfix-nonrec cccExp x e*) = *CoCallArityAnalysis.CCfix-nonrec* ($\pi \cdot$ *cccExp*) ($\pi \cdot x$) ($\pi \cdot e$)
⟨*proof*⟩

lemma *cccFix-nonrec-eqvt*[*eqvt*]: $\pi \cdot$ (*CoCallArityAnalysis.cccFix-nonrec cccExp x e*) = *CoCallArityAnalysis.cccFix-nonrec* ($\pi \cdot$ *cccExp*) ($\pi \cdot x$) ($\pi \cdot e$)
⟨*proof*⟩

12.5.2 Combining the cases

context *CoCallArityAnalysis*
begin
definition *cccFix-choose* :: *heap* ⇒ ((*AEnv* × *CoCalls*) → (*AEnv* × *CoCalls*))
where *cccFix-choose* Γ = (*if nonrec* Γ *then case-prod cccFix-nonrec* (*hd* Γ) *else cccFix* Γ)

lemma *cccFix-choose-simp1*[*simp*]:
¬ *nonrec* Γ ⟹ *cccFix-choose* Γ = *cccFix* Γ
⟨*proof*⟩

lemma *cccFix-choose-simp2*[*simp*]:
x ∉ *fv e* ⟹ *cccFix-choose* [(*x*,*e*)] = *cccFix-nonrec x e*
⟨*proof*⟩

end

lemma *cccFix-choose-eqvt*[*eqvt*]: 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-TTree

theory *CoCallGraph−TTree*
imports *CoCallGraph TTree−HOLCF*
begin

lemma *interleave-ccFromList*:
xs ∈ *interleave ys zs* ⟹ *ccFromList xs* = *ccFromList ys* ⊔ *ccFromList zs* ⊔ *ccProd* (*set ys*) (*set zs*)
⟨*proof*⟩

lift-definition *ccApprox* :: *var ttree* ⇒ *CoCalls*
is λ *xss* . *lub* (*ccFromList* ‘ *xss*)⟨*proof*⟩

lemma *ccApprox-paths*: *ccApprox t = lub (ccFromList ‘ (paths t)) ⟨proof⟩*

lemma *ccApprox-strict*[*simp*]: *ccApprox ⊥ = ⊥*
⟨proof⟩

lemma *in-ccApprox*: *(x−−y∈(ccApprox t)) ⟷ (∃ xs ∈ paths t. (x−−y∈(ccFromList xs)))*
⟨proof⟩

lemma *ccApprox-mono*: *paths t ⊆ paths t′ ⟹ ccApprox t ⊑ ccApprox t′*
⟨proof⟩

lemma *ccApprox-mono′*: *t ⊑ t′ ⟹ ccApprox t ⊑ ccApprox t′*
⟨proof⟩

lemma *ccApprox-belowI*: *(⋀ xs. xs ∈ paths t ⟹ ccFromList xs ⊑ G) ⟹ ccApprox t ⊑ G*
⟨proof⟩

lemma *ccApprox-below-iff*: *ccApprox t ⊑ G ⟷ (∀ xs ∈ paths t. ccFromList xs ⊑ G)*
⟨proof⟩

lemma *cc-restr-ccApprox-below-iff*: *cc-restr S (ccApprox t) ⊑ G ⟷ (∀ xs ∈ paths t. cc-restr S (ccFromList xs) ⊑ G)*
⟨proof⟩

lemma *ccFromList-below-ccApprox*:
xs ∈ paths t ⟹ ccFromList xs ⊑ ccApprox t
⟨proof⟩

lemma *ccApprox-nxt-below*:
ccApprox (nxt t x) ⊑ ccApprox t
⟨proof⟩

lemma *ccApprox-ttree-restr-nxt-below*:
ccApprox (ttree-restr S (nxt t x)) ⊑ ccApprox (ttree-restr S t)
⟨proof⟩

lemma *ccApprox-ttree-restr*[*simp*]: *ccApprox (ttree-restr S t) = cc-restr S (ccApprox t)*
⟨proof⟩

lemma *ccApprox-both*: *ccApprox (t ⊗⊗ t′) = ccApprox t ⊔ ccApprox t′ ⊔ ccProd (carrier t) (carrier t′)*
⟨proof⟩

lemma *ccApprox-many-calls*[*simp*]:
ccApprox (many-calls x) = ccProd {x} {x}
⟨proof⟩

lemma *ccApprox-single*[*simp*]:
ccApprox (TTree.single y) = ⊥

⟨*proof*⟩

lemma *ccApprox-either*[*simp*]: *ccApprox* (*t* ⊕⊕ *t'*) = *ccApprox t* ⊔ *ccApprox t'*
⟨*proof*⟩

lemma *wild-recursion*:
assumes *ccApprox t* ⊑ *G*
assumes ⋀ *x*. *x* ∉ *S* ⟹ *f x* = *empty*
assumes ⋀ *x*. *x* ∈ *S* ⟹ *ccApprox* (*f x*) ⊑ *G*
assumes ⋀ *x*. *x* ∈ *S* ⟹ *ccProd* (*ccNeighbors x G*) (*carrier* (*f x*)) ⊑ *G*
shows *ccApprox* (*ttree-restr* (−*S*) (*substitute f T t*)) ⊑ *G*
⟨*proof*⟩

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

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

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

lemma *valid-lists-downset-aux*:
xs ∈ *valid-lists S CoCalls* ⟹ *butlast xs* ∈ *valid-lists S CoCalls*
⟨*proof*⟩

lemma *valid-lists-subset*: *xs* ∈ *valid-lists S G* ⟹ *set xs* ⊆ *S*
⟨*proof*⟩

lemma *valid-lists-mono1*:
assumes *S* ⊆ *S'*
shows *valid-lists S G* ⊆ *valid-lists S' G*
⟨*proof*⟩

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

lemma *valid-lists-chain2*:
assumes *chain Y*
assumes *xs* ∈ *valid-lists S* (⨆ *i*. *Y i*)
shows ∃ *i*. *xs* ∈ *valid-lists S* (*Y i*)
⟨*proof*⟩

lemma *valid-lists-cc-restr*: *valid-lists S G* = *valid-lists S* (*cc-restr S G*)
⟨*proof*⟩

lemma *interleave-valid-list*:
xs ∈ *ys* ⊗ *zs* ⟹ *ys* ∈ *valid-lists S G* ⟹ *zs* ∈ *valid-lists S′ G′* ⟹ *xs* ∈ *valid-lists* (*S* ∪ *S′*) (*G* ⊔ (*G′* ⊔ *ccProd S S′*))
⟨*proof*⟩

lemma *interleave-valid-list′*:
xs ∈ *valid-lists* (*S* ∪ *S′*) *G* ⟹ ∃ *ys zs*. *xs* ∈ *ys* ⊗ *zs* ∧ *ys* ∈ *valid-lists S G* ∧ *zs* ∈ *valid-lists S′ G*
⟨*proof*⟩

lemma *many-calls-valid-list*:
xs ∈ *valid-lists* {*x*} (*ccProd* {*x*} {*x*}) ⟹ *xs* ∈ *range* (λ*n*. *replicate n x*)
⟨*proof*⟩

lemma *filter-valid-lists*:
xs ∈ *valid-lists S G* ⟹ *filter P xs* ∈ *valid-lists* {*a* ∈ *S*. *P a*} *G*
⟨*proof*⟩

lift-definition *ccTTree* :: *var set* ⇒ *CoCalls* ⇒ *var ttree* **is** λ *S G*. *valid-lists S G*
⟨*proof*⟩

lemma *paths-ccTTree*[*simp*]: *paths* (*ccTTree S G*) = *valid-lists S G* ⟨*proof*⟩

lemma *carrier-ccTTree*[*simp*]: *carrier* (*ccTTree S G*) = *S*
⟨*proof*⟩

lemma *valid-lists-ccFromList*:
xs ∈ *valid-lists S G* ⟹ *ccFromList xs* ⊑ *cc-restr S G*
⟨*proof*⟩

lemma *ccApprox-ccTTree*[*simp*]: *ccApprox* (*ccTTree S G*) = *cc-restr S G*
⟨*proof*⟩

lemma *below-ccTTreeI*:
assumes *carrier t* ⊆ *S* **and** *ccApprox t* ⊑ *G*
shows *t* ⊑ *ccTTree S G*
⟨*proof*⟩

lemma *ccTTree-mono1*:

$S \subseteq S' \Longrightarrow ccTTree\ S\ G \sqsubseteq ccTTree\ S'\ G$
⟨*proof*⟩

lemma *cont-ccTTree1*:
$cont\ (\lambda\ S.\ ccTTree\ S\ G)$
⟨*proof*⟩

lemma *ccTTree-mono2*:
$G \sqsubseteq G' \Longrightarrow ccTTree\ S\ G \sqsubseteq ccTTree\ S\ G'$
⟨*proof*⟩

lemma *ccTTree-mono*:
$S \subseteq S' \Longrightarrow G \sqsubseteq G' \Longrightarrow ccTTree\ S\ G \sqsubseteq ccTTree\ S'\ G'$
⟨*proof*⟩

lemma *cont-ccTTree2*:
$cont\ (ccTTree\ S)$
⟨*proof*⟩

lemmas *cont-ccTTree* = *cont-compose2*[**where** $c = ccTTree$, *OF cont-ccTTree1 cont-ccTTree2*, *simp*, *cont2cont*]

lemma *ccTTree-below-singleI*:
assumes $S \cap S' = \{\}$
shows $ccTTree\ S\ G \sqsubseteq singles\ S'$
⟨*proof*⟩

lemma *ccTTree-cc-restr*: $ccTTree\ S\ G = ccTTree\ S\ (cc\text{-}restr\ S\ G)$
⟨*proof*⟩

lemma *ccTTree-cong-below*: $cc\text{-}restr\ S\ G \sqsubseteq cc\text{-}restr\ S\ G' \Longrightarrow ccTTree\ S\ G \sqsubseteq ccTTree\ S\ G'$
⟨*proof*⟩

lemma *ccTTree-cong*: $cc\text{-}restr\ S\ G = cc\text{-}restr\ S\ G' \Longrightarrow ccTTree\ S\ G = ccTTree\ S\ G'$
⟨*proof*⟩

lemma *either-ccTTree*:
$ccTTree\ S\ G \oplus\oplus ccTTree\ S'\ G' \sqsubseteq ccTTree\ (S \cup S')\ (G \sqcup G')$
⟨*proof*⟩

lemma *interleave-ccTTree*:
$ccTTree\ S\ G \otimes\otimes ccTTree\ S'\ G' \sqsubseteq ccTTree\ (S \cup S')\ (G \sqcup G' \sqcup ccProd\ S\ S')$
⟨*proof*⟩

lemma *interleave-ccTTree'*:
$ccTTree\ (S \cup S')\ G \sqsubseteq ccTTree\ S\ G \otimes\otimes ccTTree\ S'\ G$

⟨*proof*⟩

lemma *many-calls-ccTTree*:
shows *many-calls x* = *ccTTree* {*x*} (*ccProd* {*x*} {*x*})
⟨*proof*⟩

lemma *filter-valid-lists'*:
xs ∈ *valid-lists* {*x'* ∈ *S*. *P x'*} *G* ⟹ *xs* ∈ *filter P* ‘ *valid-lists S G*
⟨*proof*⟩

lemma *without-ccTTree*[*simp*]:
without x (*ccTTree S G*) = *ccTTree* (*S* − {*x*}) *G*
⟨*proof*⟩

lemma *ttree-restr-ccTTree*[*simp*]:
ttree-restr S' (*ccTTree S G*) = *ccTTree* (*S* ∩ *S'*) *G*
⟨*proof*⟩

lemma *repeatable-ccTTree-ccSquare*: *S* ⊆ *S'* ⟹ *repeatable* (*ccTTree S* (*ccSquare S'*))
⟨*proof*⟩

An alternative definition

inductive *valid-lists'* :: *var set* ⇒ *CoCalls* ⇒ *var set* ⇒ *var list* ⇒ *bool*
for *S G*
where *valid-lists' S G prefix* []
| *prefix* ⊆ *ccNeighbors x G* ⟹ *valid-lists' S G* (*insert x prefix*) *xs* ⟹ *x* ∈ *S* ⟹ *valid-lists' S G prefix* (*x*#*xs*)

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

lemma *valid-lists-valid-lists'*:
xs ∈ *valid-lists S G* ⟹ *ccProd prefix* (*set xs*) ⊑ *G* ⟹ *valid-lists' S G prefix xs*
⟨*proof*⟩

lemma *valid-lists'-valid-lists-aux*:
valid-lists' S G prefix xs ⟹ *x* ∈ *prefix* ⟹ *ccProd* (*set xs*) {*x*} ⊑ *G*
⟨*proof*⟩

lemma *valid-lists'-valid-lists*:
valid-lists' S G prefix xs ⟹ *xs* ∈ *valid-lists S G*
⟨*proof*⟩

Yet another definition

lemma *valid-lists-characterization*:
xs ∈ *valid-lists S G* ⟷ *set xs* ⊆ *S* ∧ (∀ *n*. *ccProd* (*set* (*take n xs*)) (*set* (*drop n xs*)) ⊑ *G*)
⟨*proof*⟩

end

12.7 CoCallImplTTree

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

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

lemma *Texp-simp*: *Texp e*·*a* = *ccTTree* (*edom* (*Aexp e* ·*a*)) (*ccExp e*·*a*)
⟨*proof*⟩

sublocale *TTreeAnalysis Texp*⟨*proof*⟩
end

end

12.8 CoCallImplTTreeSafe

theory *CoCallImplTTreeSafe*
imports *CoCallImplTTree CoCallAnalysisSpec TTreeAnalysisSpec*
begin

lemma *valid-lists-many-calls*:
assumes ¬ *one-call-in-path x p*
assumes *p* ∈ *valid-lists S G*
shows *x*−−*x* ∈ *G*
⟨*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* Δ·(*Aheap* Δ *e*·*a*)) *x*) ⊆ *edom* ((*ABinds* Δ)·(*Aheap* Δ *e*·*a*))
⟨*proof*⟩

sublocale *TTreeAnalysisCarrier Texp*

⟨*proof*⟩

sublocale *TTreeAnalysisSafe Texp*
⟨*proof*⟩

definition *Theap* :: *heap* ⇒ *exp* ⇒ *Arity* → *var ttree*
where *Theap* Γ *e* = (Λ *a*. *if nonrec* Γ *then ccTTree* (*edom* (*Aheap* Γ *e*·*a*)) (*ccExp e*·*a*) *else ttree-restr* (*edom* (*Aheap* Γ *e*·*a*)) *anything*)

lemma *Theap-simp*: *Theap* Γ *e*·*a* = (*if nonrec* Γ *then ccTTree* (*edom* (*Aheap* Γ *e*·*a*)) (*ccExp e*·*a*) *else ttree-restr* (*edom* (*Aheap* Γ *e*·*a*)) *anything*)
⟨*proof*⟩

lemma *carrier-Fheap'*:*carrier* (*Theap* Γ *e*·*a*) = *edom* (*Aheap* Γ *e*·*a*)
⟨*proof*⟩

sublocale *TTreeAnalysisCardinalityHeap Texp Aexp Aheap Theap*
⟨*proof*⟩
end

lemma *paths-singles*: *xs* ∈ *paths* (*singles S*) ⟷ (∀ *x* ∈ *S*. *one-call-in-path x xs*)
⟨*proof*⟩

lemma *paths-singles'*: *xs* ∈ *paths* (*singles S*) ⟷ (∀ *x* ∈ (*set xs* ∩ *S*). *one-call-in-path x xs*)
⟨*proof*⟩

lemma *both-below-singles1*:
assumes *t* ⊑ *singles S*
assumes *carrier t'* ∩ *S* = {}
shows *t* ⊗⊗ *t'* ⊑ *singles S*
⟨*proof*⟩

lemma *paths-ttree-restr-singles*: *xs* ∈ *paths* (*ttree-restr S'* (*singles S*)) ⟷ *set xs* ⊆ *S'* ∧ (∀ *x* ∈ *S*. *one-call-in-path x xs*)
⟨*proof*⟩

lemma *substitute-not-carrier*:
assumes *x* ∉ *carrier t*
assumes ⋀ *x'*. *x* ∉ *carrier* (*f x'*)
shows *x* ∉ *carrier* (*substitute f T t*)
⟨*proof*⟩

lemma *substitute-below-singlesI*:
assumes $t \sqsubseteq$ *singles* S
assumes $\bigwedge x.$ *carrier* $(f\ x) \cap S = \{\}$
shows *substitute* $f\ T\ t \sqsubseteq$ *singles* S
⟨*proof*⟩

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-eqvt*[*eqvt*]:
shows $\pi \cdot$ *combined-restrict* S p = *combined-restrict* $(\pi \cdot S)$ $(\pi \cdot p)$
⟨*proof*⟩

lemma *combined-restrict-cont*:
cont $(\lambda x.$ *combined-restrict* S $x)$
⟨*proof*⟩
lemmas *cont-compose*[*OF combined-restrict-cont*, *cont2cont*, *simp*]

lemma *combined-restrict-perm*:
assumes *supp* π ♯* S **and** [*simp*]: *finite* S
shows *combined-restrict* S $(\pi \cdot p)$ = *combined-restrict* S p
⟨*proof*⟩

definition *predCC* :: *var set* ⇒ (*Arity* → *CoCalls*) ⇒ (*Arity* → *CoCalls*)
where *predCC* S f = $(\Lambda\ a.$ *if* $a \neq 0$ *then cc-restr* S $(f{\cdot}(pred{\cdot}a))$ *else ccSquare* $S)$

lemma *predCC-eq*:
shows *predCC* S $f \cdot a$ = (*if* $a \neq 0$ *then cc-restr* S $(f{\cdot}(pred{\cdot}a))$ *else ccSquare* S)

⟨*proof*⟩

lemma *predCC-eqvt*[*eqvt*, *simp*]: $\pi \cdot (predCC\ S\ f) = predCC\ (\pi \cdot S)\ (\pi \cdot f)$
⟨*proof*⟩

lemma *cc-restr-predCC*:
$cc\text{-}restr\ S\ (predCC\ S'\ f{\cdot}n) = (predCC\ (S' \cap S)\ (\Lambda\ n.\ cc\text{-}restr\ S\ (f{\cdot}n))){\cdot}n$
⟨*proof*⟩

lemma *cc-restr-predCC'*[*simp*]:
$cc\text{-}restr\ S\ (predCC\ S\ f{\cdot}n) = predCC\ S\ f{\cdot}n$
⟨*proof*⟩

nominal-function
$cCCexp :: exp \Rightarrow (Arity \to AEnv \times CoCalls)$
where
$cCCexp\ (Var\ x) = (\Lambda\ n\ .\ (esing\ x \cdot (up \cdot n), \bot))$
| $cCCexp\ (Lam\ [x].\ e) = (\Lambda\ n\ .\ combined\text{-}restrict\ (fv\ (Lam\ [x].\ e))\ (fst\ (cCCexp\ e{\cdot}(pred{\cdot}n)), predCC\ (fv\ (Lam\ [x].\ e))\ (\Lambda\ a.\ snd(cCCexp\ e{\cdot}a)){\cdot}n))$
| $cCCexp\ (App\ e\ x) = (\Lambda\ n\ .\ (fst\ (cCCexp\ e{\cdot}(inc{\cdot}n)) \sqcup (esing\ x \cdot (up{\cdot}0)), snd\ (cCCexp\ e{\cdot}(inc{\cdot}n)) \sqcup ccProd\ \{x\}\ (insert\ x\ (fv\ e))))$
| $cCCexp\ (Let\ \Gamma\ e) = (\Lambda\ n\ .\ combined\text{-}restrict\ (fv\ (Let\ \Gamma\ e))\ (CoCallArityAnalysis.cccFix\text{-}choose\ cCCexp\ \Gamma \cdot (cCCexp\ e{\cdot}n)))$
| $cCCexp\ (Bool\ b) = \bot$
| $cCCexp\ (scrut\ ?\ e1\ :\ e2) = (\Lambda\ n.\ (fst\ (cCCexp\ scrut{\cdot}0) \sqcup fst\ (cCCexp\ e1{\cdot}n) \sqcup fst\ (cCCexp\ e2{\cdot}n),$
$snd\ (cCCexp\ scrut{\cdot}0) \sqcup (snd\ (cCCexp\ e1{\cdot}n) \sqcup snd\ (cCCexp\ e2{\cdot}n)) \sqcup ccProd\ (edom\ (fst\ (cCCexp\ scrut{\cdot}0)))\ (edom\ (fst\ (cCCexp\ e1{\cdot}n)) \cup edom\ (fst\ (cCCexp\ e2{\cdot}n)))))$
⟨*proof*⟩

nominal-termination (*eqvt*) ⟨*proof*⟩

locale *CoCallAnalysisImpl*
begin
sublocale *CoCallArityAnalysis cCCexp*⟨*proof*⟩
sublocale *ArityAnalysis Aexp*⟨*proof*⟩

abbreviation *Aexp-syn''* (‹$\mathcal{A}$_›) **where** $\mathcal{A}_a\ e \equiv Aexp\ e{\cdot}a$
abbreviation *Aexp-bot-syn''* (‹$\mathcal{A}^\bot$_›) **where** $\mathcal{A}^\bot{}_a\ e \equiv fup{\cdot}(Aexp\ e){\cdot}a$

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

lemma *cCCexp-eq*[*simp*]:
$cCCexp\ (Var\ x){\cdot}n = (esing\ x \cdot (up \cdot n), \bot)$
$cCCexp\ (Lam\ [x].\ e){\cdot}n = combined\text{-}restrict\ (fv\ (Lam\ [x].\ e))\ (fst\ (cCCexp\ e{\cdot}(pred{\cdot}n)), predCC\ (fv\ (Lam\ [x].\ e))\ (\Lambda\ a.\ snd(cCCexp\ e{\cdot}a)){\cdot}n)$

cCCexp (*App e x*)·*n* = (*fst* (*cCCexp e*·(*inc*·*n*)) ⊔ (*esing x* · (*up*·*0*)), *snd* (*cCCexp e*·(*inc*·*n*)) ⊔ *ccProd* {*x*} (*insert x* (*fv e*)))
cCCexp (*Let* Γ *e*)·*n* = *combined-restrict* (*fv* (*Let* Γ *e*)) (*CoCallArityAnalysis.cccFix-choose cCCexp* Γ · (*cCCexp e*·*n*))
cCCexp (*Bool b*)·*n* = ⊥
cCCexp (*scrut ? e1 : e2*)·*n* = (*fst* (*cCCexp scrut*·*0*) ⊔ *fst* (*cCCexp e1*·*n*) ⊔ *fst* (*cCCexp e2*·*n*),
snd (*cCCexp scrut*·*0*) ⊔ (*snd* (*cCCexp e1*·*n*) ⊔ *snd* (*cCCexp e2*·*n*)) ⊔ *ccProd* (*edom* (*fst* (*cCCexp scrut*·*0*))) (*edom* (*fst* (*cCCexp e1*·*n*)) ∪ *edom* (*fst* (*cCCexp e2*·*n*))))
⟨*proof*⟩
declare *cCCexp.simps*[*simp del*]

lemma *Aexp-pre-simps*:
$\mathcal{A}_a$ (*Var x*) = *esing x*·(*up*·*a*)
$\mathcal{A}_a$ (*Lam* [*x*]. *e*) = *Aexp e*·(*pred*·*a*) *f*|‘ *fv* (*Lam* [*x*]. *e*)
$\mathcal{A}_a$ (*App e x*) = *Aexp e*·(*inc*·*a*) ⊔ *esing x*·(*up*·*0*)
¬ *nonrec* Γ ⟹
$\mathcal{A}_a$ (*Let* Γ *e*) = (*Afix* Γ·($\mathcal{A}_a$ *e* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ)) *f*|‘ (*fv* (*Let* Γ *e*))
x ∉ *fv e* ⟹
$\mathcal{A}_a$ (*let x be e in exp*) =
(*fup*·(*Aexp e*)·(*ABind-nonrec x e*·($\mathcal{A}_a$ *exp*, *CCexp exp*·*a*)) ⊔ $\mathcal{A}_a$ *exp*)
f|‘ (*fv* (*let x be e in exp*))
$\mathcal{A}_a$ (*Bool b*) = ⊥
$\mathcal{A}_a$ (*scrut ? e1 : e2*) = $\mathcal{A}_0$ *scrut* ⊔ $\mathcal{A}_a$ *e1* ⊔ $\mathcal{A}_a$ *e2*
⟨*proof*⟩

lemma *CCexp-pre-simps*:
CCexp (*Var x*)·*n* = ⊥
CCexp (*Lam* [*x*]. *e*)·*n* = *predCC* (*fv* (*Lam* [*x*]. *e*)) (*CCexp e*)·*n*
CCexp (*App e x*)·*n* = *CCexp e*·(*inc*·*n*) ⊔ *ccProd* {*x*} (*insert x* (*fv e*))
¬ *nonrec* Γ ⟹
CCexp (*Let* Γ *e*)·*n* = *cc-restr* (*fv* (*Let* Γ *e*))
(*CCfix* Γ·(*Afix* Γ·(*Aexp e*·*n* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ), *CCexp e*·*n*))
x ∉ *fv e* ⟹ *CCexp* (*let x be e in exp*)·*n* =
cc-restr (*fv* (*let x be e in exp*))
(*ccBind x e* ·(*Aheap-nonrec x e*·(*Aexp exp*·*n*, *CCexp exp*·*n*), *CCexp exp*·*n*)
⊔ *ccProd* (*fv e*) (*ccNeighbors x* (*CCexp exp*·*n*) − (*if isVal e then* {} *else* {*x*})) ⊔ *CCexp exp*·*n*)
CCexp (*Bool b*)·*n* = ⊥
CCexp (*scrut ? e1 : e2*)·*n* =
CCexp scrut·*0* ⊔
(*CCexp e1*·*n* ⊔ *CCexp e2*·*n*) ⊔
ccProd (*edom* (*Aexp scrut*·*0*)) (*edom* (*Aexp e1*·*n*) ∪ *edom* (*Aexp e2*·*n*))
⟨*proof*⟩

lemma
shows *ccField-CCexp*: *ccField* (*CCexp e*·*a*) ⊆ *fv e* **and** *Aexp-edom'*: *edom* ($\mathcal{A}_a$ *e*) ⊆ *fv e*

⟨*proof*⟩

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*]:
$\mathcal{G}_a$(*Var x*) = ⊥
$\mathcal{G}_0$(*Lam* [*x*]. *e*) = (*fv* (*Lam* [*x*]. *e*))^2
$\mathcal{G}_{inc\cdot a}$(*Lam* [*x*]. *e*) = *cc-delete x* ($\mathcal{G}_a$ *e*)
$\mathcal{G}_a$ (*App e x*) = $\mathcal{G}_{inc\cdot a}$ *e* ⊔ {*x*} *G*×*insert x* (*fv e*)
¬ *nonrec* Γ ⟹ $\mathcal{G}_a$ (*Let* Γ *e*) =
(*CCfix* Γ·(*Afix* Γ·($\mathcal{A}_a$ *e* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ), $\mathcal{G}_a$ *e*)) *G*|‘ (− *domA* Γ)
x ∉ *fv e*′ ⟹ $\mathcal{G}_a$ (*let x be e*′ *in e*) =
cc-delete x
(*ccBind x e*′ ·(*Aheap-nonrec x e*′·($\mathcal{A}_a$ *e*, $\mathcal{G}_a$ *e*), $\mathcal{G}_a$ *e*)
⊔ *fv e*′ *G*× (*ccNeighbors x* ($\mathcal{G}_a$ *e*) − (*if isVal e*′ *then* {} *else* {*x*})) ⊔ $\mathcal{G}_a$ *e*)
$\mathcal{G}_a$ (*Bool b*) = ⊥
$\mathcal{G}_a$ (*scrut ? e1 : e2*) =
$\mathcal{G}_0$ *scrut* ⊔ ($\mathcal{G}_a$ *e1* ⊔ $\mathcal{G}_a$ *e2*) ⊔
edom ($\mathcal{A}_0$ *scrut*) *G*× (*edom* ($\mathcal{A}_a$ *e1*) ∪ *edom* ($\mathcal{A}_a$ *e2*))
⟨*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*⟩

sublocale *ArityAnalysisHeapEqvt Aheap*
⟨*proof*⟩

lemma *Aexp-lam-simp*: *Aexp* (*Lam* [*x*]. *e*) · *n* = *env-delete x* (*Aexp e* · (*pred* · *n*))
⟨*proof*⟩

lemma *Aexp-Let-simp1*:
¬ *nonrec* Γ ⟹ $\mathcal{A}_a$ (*Let* Γ *e*) = (*Afix* Γ·($\mathcal{A}_a$ *e* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ)) *f*|‘ (− *domA* Γ)
⟨*proof*⟩

lemma *Aexp-Let-simp2*:
x ∉ *fv e* ⟹ $\mathcal{A}_a$(*let x be e in exp*) = *env-delete x* ($\mathcal{A}^{\perp}{}_{ABind\text{-}nonrec\ x\ e}$ · ($\mathcal{A}_a$ *exp*, *CCexp exp*·*a*) *e* ⊔ $\mathcal{A}_a$ *exp*)
⟨*proof*⟩

lemma *Aexp-simps*[*simp*]:
$\mathcal{A}_a$(*Var x*) = *esing x*·(*up*·*a*)
$\mathcal{A}_a$(*Lam* [*x*]. *e*) = *env-delete x* ($\mathcal{A}_{pred\cdot a}$ *e*)
$\mathcal{A}_a$(*App e x*) = *Aexp e*·(*inc*·*a*) ⊔ *esing x*·(*up*·*0*)
¬ *nonrec* Γ ⟹ $\mathcal{A}_a$(*Let* Γ *e*) =
(*Afix* Γ·($\mathcal{A}_a$ *e* ⊔ (λ-.*up*·*0*) *f*|‘ *thunks* Γ)) *f*|‘ (− *domA* Γ)
x ∉ *fv e*′ ⟹ $\mathcal{A}_a$(*let x be e*′ *in e*) =
env-delete x ($\mathcal{A}^{\perp}{}_{ABind\text{-}nonrec\ x\ e'}$·($\mathcal{A}_a$ *e*, $\mathcal{G}_a$ *e*) *e*′ ⊔ $\mathcal{A}_a$ *e*)
$\mathcal{A}_a$(*Bool b*) = ⊥
$\mathcal{A}_a$(*scrut ? e1 : e2*) = $\mathcal{A}_0$ *scrut* ⊔ $\mathcal{A}_a$ *e1* ⊔ $\mathcal{A}_a$ *e2*
⟨*proof*⟩

end

end

13.2 CoCallImplSafe

theory *CoCallImplSafe*
imports *CoCallAnalysisImpl CoCallAnalysisSpec ArityAnalysisFixProps*
begin

locale *CoCallImplSafe*
begin
sublocale *CoCallAnalysisImpl*⟨*proof*⟩

lemma *ccNeighbors-Int-ccrestr*: (*ccNeighbors x G* ∩ *S*) = *ccNeighbors x* (*cc-restr* (*insert x S*) *G*) ∩ *S*
⟨*proof*⟩

lemma
assumes $x \notin S$ **and** $y \notin S$
shows *CCexp-subst*: *cc-restr S* (*CCexp e*[*y*::=*x*]·*a*) = *cc-restr S* (*CCexp e*·*a*)
and *Aexp-restr-subst*: (*Aexp e*[*y*::=*x*]·*a*) *f*|‘ *S* = (*Aexp e*·*a*) *f*|‘ *S*
⟨*proof*⟩

sublocale *ArityAnalysisSafe Aexp*
⟨*proof*⟩

sublocale *ArityAnalysisLetSafe Aexp Aheap*
⟨*proof*⟩

definition *ccHeap-nonrec*
where *ccHeap-nonrec x e exp* = (Λ *n*. *CCfix-nonrec x e*·(*Aexp exp*·*n*, *CCexp exp*·*n*))

lemma *ccHeap-nonrec-eq*:
ccHeap-nonrec x e exp·*n* = *CCfix-nonrec x e*·(*Aexp exp*·*n*, *CCexp exp*·*n*)
⟨*proof*⟩

definition *ccHeap-rec* :: *heap* ⇒ *exp* ⇒ *Arity* → *CoCalls*
where *ccHeap-rec* Γ *e* = (Λ *a*. *CCfix* Γ·(*Afix* Γ·(*Aexp e*·*a* ⊔ (λ-.*up*·*0*) *f*|‘ (*thunks* Γ)), *CCexp e*·*a*))

lemma *ccHeap-rec-eq*:
ccHeap-rec Γ *e*·*a* = *CCfix* Γ·(*Afix* Γ·(*Aexp e*·*a* ⊔ (λ-.*up*·*0*) *f*|‘ (*thunks* Γ)), *CCexp e*·*a*)
⟨*proof*⟩

definition *ccHeap* :: *heap* ⇒ *exp* ⇒ *Arity* → *CoCalls*
where *ccHeap* Γ = (*if nonrec* Γ *then case-prod ccHeap-nonrec* (*hd* Γ) *else ccHeap-rec* Γ)

lemma *ccHeap-simp1*:
¬ *nonrec* Γ ⟹ *ccHeap* Γ *e*·*a* = *CCfix* Γ·(*Afix* Γ·(*Aexp e*·*a* ⊔ (λ-.*up*·*0*) *f*|‘ (*thunks* Γ)), *CCexp e*·*a*)
⟨*proof*⟩

lemma *ccHeap-simp2*:
x ∉ *fv e* ⟹ *ccHeap* [(*x*,*e*)] *exp*·*n* = *CCfix-nonrec x e*·(*Aexp exp*·*n*, *CCexp exp*·*n*)
⟨*proof*⟩

sublocale *CoCallAritySafe CCexp Aexp ccHeap Aheap*
⟨*proof*⟩
end

end

14 End-to-end Saftey 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* ⟹ *x* ∉ *fv e*
⟨*proof*⟩

lemma *example1*:
fixes *e* :: *exp*
fixes *f g x y z* :: *var*
assumes *Aexp-e*: ⋀*a*. *Aexp e*·*a* = *esing x*·(*up*·*a*) ⊔ *esing y*·(*up*·*a*)
assumes *ccExp-e*: ⋀*a*. *CCexp e*·*a* = ⊥
assumes [*simp*]: *transform 1 e* = *e*
assumes *isVal e*
assumes *disj*: *y* ≠ *f y* ≠ *g x* ≠ *y z* ≠ *f z* ≠ *g y* ≠ *x*
assumes *fresh*: *atom z* ♯ *e*
shows *transform 1* (*let y be App* (*Var f*) *g in* (*let x be e in* (*Var x*))) =
let y be (*Lam* [*z*]. *App* (*App* (*Var f*) *g*) *z*) *in* (*let x be* (*Lam* [*z*]. *App e z*) *in* (*Var x*))
⟨*proof*⟩

end
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'* (⟨$\mathcal{T}$_⟩) **where** $\mathcal{T}_a \equiv$ *transform a*

lemma *end2end*:
c ⇒* *c'* ⟹
¬ *boring-step c'* ⟹

heap-upds-ok-conf $c \Longrightarrow$
consistent $(ae,\ ce,\ a,\ as,\ r)\ c \Longrightarrow$
$\exists\, ae'\ ce'\ a'\ as'\ r'.$ *consistent* $(ae',\ ce',\ a',\ as',\ r')\ c' \wedge$ *conf-transform* $(ae,\ ce,\ a,\ as,\ r)\ c \Rightarrow_G{}^*$ *conf-transform* $(ae',\ ce',\ a',\ as',\ r')\ c'$
⟨*proof*⟩

theorem *end2end-closed*:
assumes *closed*: *fv e* = ({} :: *var set*)
assumes ([], *e*, []) ⇒* (Γ,*v*,[]) **and** *isVal v*
obtains Γ′ **and** *v*′
where ([], $\mathcal{T}_0$ *e*, []) ⇒* (Γ′,*v*′,[]) **and** *isVal v*′
and *card* (*domA* Γ′) ≤ *card* (*domA* Γ)
⟨*proof*⟩

lemma *fresh-var-eqE*[*elim-format*]: *fresh-var e* = *x* ⟹ *x* ∉ *fv e*
⟨*proof*⟩

lemma *example1*:
fixes *e* :: *exp*
fixes *f g x y z* :: *var*
assumes *Aexp-e*: ⋀*a*. *Aexp e*·*a* = *esing x*·(*up*·*a*) ⊔ *esing y*·(*up*·*a*)
assumes *ccExp-e*: ⋀*a*. *CCexp e*·*a* = ⊥
assumes [*simp*]: *transform 1 e* = *e*
assumes *isVal e*
assumes *disj*: *y* ≠ *f y* ≠ *g x* ≠ *y z* ≠ *f z* ≠ *g y* ≠ *x*
assumes *fresh*: *atom z* ♯ *e*
shows *transform 1* (*let y be App* (*Var f*) *g in* (*let x be e in* (*Var x*))) =
let y be (*Lam* [*z*]. *App* (*App* (*Var f*) *g*) *z*) *in* (*let x be* (*Lam* [*z*]. *App e z*) *in* (*Var x*))
⟨*proof*⟩

end
end

15 Functional Correctness of the Arity Analysis

15.1 ArityAnalysisCorrDenotational

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

context *ArityAnalysisLetSafe*
begin

inductive *eq* :: *Arity* ⇒ *Value* ⇒ *Value* ⇒ *bool* **where**
eq 0 v v
| (⋀ *v*. *eq n* (*v1* ↓*Fn v*) (*v2* ↓*Fn v*)) ⟹ *eq* (*inc*·*n*) *v1 v2*

lemma [*simp*]: $eq\ 0\ v\ v' \longleftrightarrow v = v'$
⟨*proof*⟩

lemma *eq-inc-simp*:
$eq\ (inc{\cdot}n)\ v1\ v2 \longleftrightarrow (\forall\ v\ .\ eq\ n\ (v1 \downarrow Fn\ v)\ (v2 \downarrow Fn\ v))$
⟨*proof*⟩

lemma *eq-FnI*:
$(\bigwedge v.\ eq\ (pred{\cdot}n)\ (f1{\cdot}v)\ (f2{\cdot}v)) \Longrightarrow eq\ n\ (Fn{\cdot}f1)\ (Fn{\cdot}f2)$
⟨*proof*⟩

lemma *eq-refl*[*simp*]: $eq\ a\ v\ v$
⟨*proof*⟩

lemma *eq-trans*[*trans*]: $eq\ a\ v1\ v2 \Longrightarrow eq\ a\ v2\ v3 \Longrightarrow eq\ a\ v1\ v3$
⟨*proof*⟩

lemma *eq-Fn*: $eq\ a\ v1\ v2 \Longrightarrow eq\ (pred{\cdot}a)\ (v1 \downarrow Fn\ v)\ (v2 \downarrow Fn\ v)$
⟨*proof*⟩

lemma *eq-inc-same*: $eq\ a\ v1\ v2 \Longrightarrow eq\ (inc{\cdot}a)\ v1\ v2$
⟨*proof*⟩

lemma *eq-mono*: $a \sqsubseteq a' \Longrightarrow eq\ a'\ v1\ v2 \Longrightarrow eq\ a\ v1\ v2$
⟨*proof*⟩

lemma *eq-join*[*simp*]: $eq\ (a \sqcup a')\ v1\ v2 \longleftrightarrow eq\ a\ v1\ v2 \wedge eq\ a'\ v1\ v2$
⟨*proof*⟩

lemma *eq-adm*: $cont\ f \Longrightarrow cont\ g \Longrightarrow adm\ (\lambda\ x.\ eq\ a\ (f\ x)\ (g\ x))$
⟨*proof*⟩

inductive $eq\varrho :: AEnv \Rightarrow (var \Rightarrow Value) \Rightarrow (var \Rightarrow Value) \Rightarrow bool$ **where**
$eq\varrho I$: $(\bigwedge x\ a.\ ae\ x = up{\cdot}a \Longrightarrow eq\ a\ (\varrho 1\ x)\ (\varrho 2\ x)) \Longrightarrow eq\varrho\ ae\ \varrho 1\ \varrho 2$

lemma $eq\varrho E$: $eq\varrho\ ae\ \varrho 1\ \varrho 2 \Longrightarrow ae\ x = up{\cdot}a \Longrightarrow eq\ a\ (\varrho 1\ x)\ (\varrho 2\ x)$
⟨*proof*⟩

lemma $eq\varrho$-*refl*[*simp*]: $eq\varrho\ ae\ \varrho\ \varrho$
⟨*proof*⟩

lemma *eq-esing-up*[*simp*]: $eq\varrho\ (esing\ x{\cdot}(up{\cdot}a))\ \varrho 1\ \varrho 2 \longleftrightarrow eq\ a\ (\varrho 1\ x)\ (\varrho 2\ x)$
⟨*proof*⟩

lemma $eq\varrho$-*mono*:
assumes $ae \sqsubseteq ae'$
assumes $eq\varrho\ ae'\ \varrho 1\ \varrho 2$
shows $eq\varrho\ ae\ \varrho 1\ \varrho 2$

⟨*proof*⟩

lemma *eqϱ-adm*: *cont f* ⟹ *cont g* ⟹ *adm* (λ *x*. *eqϱ a* (*f x*) (*g x*))
⟨*proof*⟩

lemma *up-join-eq-up*[*simp*]: *up*·(*n*::′*a*::*Finite-Join-cpo*) ⊔ *up*·*n*′ = *up*·(*n* ⊔ *n*′)
⟨*proof*⟩

lemma *eqϱ-join*[*simp*]: *eqϱ* (*ae* ⊔ *ae*′) *ϱ1 ϱ2* ⟷ *eqϱ ae ϱ1 ϱ2* ∧ *eqϱ ae*′ *ϱ1 ϱ2*
⟨*proof*⟩

lemma *eqϱ-override*[*simp*]:
eqϱ ae (*ϱ1* +$_S$ *ϱ2*) (*ϱ1*′ +$_S$*ϱ2*′) ⟷ *eqϱ ae* (*ϱ1 f*|‘ (− *S*)) (*ϱ1*′ *f*|‘ (− *S*)) ∧ *eqϱ ae* (*ϱ2 f*|‘ *S*) (*ϱ2*′ *f*|‘ *S*)
⟨*proof*⟩

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

lemma *Aexp-body-below-Aheap*:
shows *Aexp e*·*a* ⊑ *Aheap* Γ *e*·*a* ⊔ *Aexp* (*Let* Γ *e*)·*a*
⟨*proof*⟩

lemma *Aexp-correct*: *eqϱ* (*Aexp e*·*a*) *ϱ1 ϱ2* ⟹ *eq a* (⟦*e*⟧$_{ϱ1}$) (⟦*e*⟧$_{ϱ2}$)
⟨*proof*⟩

lemma *ESem-ignores-fresh*[*simp*]: ⟦ *e* ⟧$_{ϱ(\textit{fresh-var } e := v)}$ = ⟦ *e* ⟧$_ϱ$
⟨*proof*⟩

lemma *eq-Aeta-expand*: *eq a* (⟦ *Aeta-expand a e* ⟧$_ϱ$) (⟦*e*⟧$_ϱ$)
⟨*proof*⟩

lemma *Arity-transformation-correct*: *eq a* (⟦ $\mathcal{T}_a$ *e* ⟧$_ϱ$) (⟦ *e* ⟧$_ϱ$)
⟨*proof*⟩

corollary *Arity-transformation-correct*′:
⟦ $\mathcal{T}_0$ *e* ⟧$_ϱ$ = ⟦ *e* ⟧$_ϱ$
⟨*proof*⟩

end
end