Renaming-Enriched Sets (Rensets) and Renaming-Based Recursion

Andrei Popescu

December 9, 2023

Abstract

I formalize the notion of *renaming-enriched sets* (*rensets* sor short) and renaming-based recursion introduced in my IJCAR 2022 paper "Rensets and Renaming-Based Recursion for Syntax with Bindings" [3]. Rensets are an algebraic axiomatization of renaming (variable-for-variable substitution). The formalization includes a connection with nominal sets [1, 2], showing that any renset naturally gives rise to a nominal set. It also includes examples of deploying the renaming-based recursor: semantic interpretation, counting functions for free and bound occurrences, unary and parallel substitution, etc. Finally, it includes a variation of rensets that axiomatize term-for-variable substitution, called *substitutive sets*, which yields a corresponding recursion principle.

Contents

1 Lambda Terms

theory *Lambda-Terms*
imports *Main*
begin

This theory defines pre-terms and alpha-equivalence, and then defines terms as alpha-equivalence classes of pre-terms.

hide-type *var*

abbreviation (*input*) *any* ≡ *undefined*

1.1 Variables

datatype *var* = *Variable nat*

1.2 Pre-terms and operators on them

datatype *ptrm* = *PVr var* | *PAp ptrm ptrm* | *PLm var ptrm*

inductive *pfresh* :: *var* ⇒ *ptrm* ⇒ *bool* **where**
  PVr[*intro*]: $z \neq x \Longrightarrow$ *pfresh z* (*PVr x*)
|*PAp*[*intro*]: *pfresh z t1* ⟹ *pfresh z t2* ⟹ *pfresh z* (*PAp t1 t2*)
|*PLm*[*intro*]: $z = x \vee$ *pfresh z t* ⟹ *pfresh z* (*PLm x t*)

lemma *pfresh-simps*[*simp*]:
  pfresh z (*PVr x*) ⟷ $z \neq x$
  pfresh z (*PAp t1 t2*) ⟷ *pfresh z t1* ∧ *pfresh z t2*
  pfresh z (*PLm x t*) ⟷ $z = x \vee$ *pfresh z t*
  ⟨*proof*⟩

lemma *inj-Variable*: *inj Variable*
  ⟨*proof*⟩

lemma *infinite-var*: *infinite* (*UNIV*::*var set*)
  ⟨*proof*⟩

lemma *exists-var*:
  assumes *finite X*
  shows ∃ *x*::*var*. $x \notin X$
  ⟨*proof*⟩

lemma *finite-neg-imp*:
  assumes *finite* {*x*. ¬ φ *x*} **and** *finite* {*x*. χ *x*}
  shows *finite* {*x*. φ *x* ⟶ χ *x*}
  ⟨*proof*⟩

lemma *cofinite-pfresh*: *finite* {*x* . ¬ *pfresh x t*}
  ⟨*proof*⟩

lemma *cofinite-list-ptrm*: *finite* {*x* . ∃ *t* ∈ *set ts*. ¬ *pfresh x t*}
⟨*proof*⟩

lemma *exists-pfresh-set*:
  assumes *finite X*
  shows ∃ *z*. $z \notin X \wedge z \notin$ *set xs* ∧ (∀ *t* ∈ *set ts*. *pfresh z t*)
⟨*proof*⟩

lemma *exists-pfresh*:
  ∃ *z*. $z \notin$ *set xs* ∧ (∀ *t* ∈ *set ts*. *pfresh z t*)
  ⟨*proof*⟩

definition *pickFreshS* :: *var set* ⇒ *var list* ⇒ *ptrm list* ⇒ *var* **where**
pickFreshS X xs ts ≡ *SOME z*. *z* ∉ *X* ∧ *z* ∉ *set xs* ∧ (∀ *t* ∈ *set ts*. *pfresh z t*)

lemma *pickFreshS*:
assumes *finite X*
shows *pickFreshS X xs ts* ∉ *X* ∧ *pickFreshS X xs ts* ∉ *set xs* ∧
(∀ *t* ∈ *set ts*. *pfresh* (*pickFreshS X xs ts*) *t*)
⟨*proof*⟩

lemmas *pickFreshS-set* = *pickFreshS*[*THEN conjunct1*]
and *pickFreshS-var* = *pickFreshS*[*THEN conjunct2*, *THEN conjunct1*]
and *pickFreshS-ptrm* = *pickFreshS*[*THEN conjunct2*, *THEN conjunct2*, *unfolded Ball-def*, *rule-format*]

definition *pickFresh* ≡ *pickFreshS* {}

lemmas *pickFresh-var* = *pickFreshS-var*[*OF finite.emptyI*, *unfolded pickFresh-def*[*symmetric*]]
and *pickFresh-ptrm* = *pickFreshS-ptrm*[*OF finite.emptyI*, *unfolded pickFresh-def*[*symmetric*]]

definition *sw* :: *var* ⇒ *var* ⇒ *var* ⇒ *var* **where**
sw x y z ≡ *if x* = *y then z else if x* = *z then y else x*

lemma *sw-eqL*[*simp*,*intro*!]: ⋀ *x y z*. *sw x x y* = *y*
and *sw-eqR*[*simp*,*intro*!]: ⋀ *x y z*. *sw x y x* = *y*
and *sw-diff*[*simp*]: ⋀ *x y z*. *x* ≠ *y* ⟹ *x* ≠ *z* ⟹ *sw x y z* = *x*
⟨*proof*⟩

lemma *sw-sym*: *sw x y z* = *sw x z y*
and *sw-id*[*simp*]: *sw x y y* = *x*
and *sw-sw*: *sw* (*sw x y z*) *y1 z1* = *sw* (*sw x y1 z1*) (*sw y y1 z1*) (*sw z y1 z1*)
and *sw-invol*[*simp*]: *sw* (*sw x y z*) *y z* = *x*
⟨*proof*⟩

lemma *sw-invol2*: *sw* (*sw x y z*) *z y* = *x*
⟨*proof*⟩

lemma *sw-inj*[*iff*]: *sw x z1 z2* = *sw y z1 z2* ⟷ *x* = *y*
⟨*proof*⟩

lemma *sw-surj*: ∃ *y*. *x* = *sw y z1 z2*
⟨*proof*⟩

fun *pswap* :: *ptrm* ⇒ *var* ⇒ *var* ⇒ *ptrm* **where**
PVr: *pswap* (*PVr x*) *z1 z2* = *PVr* (*sw x z1 z2*)
|*PAp*: *pswap* (*PAp s t*) *z1 z2* = *PAp* (*pswap s z1 z2*) (*pswap t z1 z2*)

|*PLm*: *pswap* (*PLm x t*) *z1 z2* = *PLm* (*sw x z1 z2*) (*pswap t z1 z2*)

lemma *pswap-sym*: *pswap s y z* = *pswap s z y*
⟨*proof*⟩

lemma *pswap-id*[*simp*]: *pswap s y y* = *s*
⟨*proof*⟩

lemma *pswap-pswap*:
pswap (*pswap s y z*) *y1 z1* = *pswap* (*pswap s y1 z1*) (*sw y y1 z1*) (*sw z y1 z1*)
⟨*proof*⟩

lemma *pswap-invol*[*simp*]: *pswap* (*pswap s y z*) *y z* = *s*
⟨*proof*⟩

lemma *pswap-invol2*: *pswap* (*pswap s y z*) *z y* = *s*
⟨*proof*⟩

lemma *pswap-inj*[*iff*]: *pswap s z1 z2* = *pswap t z1 z2* ⟷ *s* = *t*
⟨*proof*⟩

lemma *pswap-surj*: ∃ *t*. *s* = *pswap t z1 z2*
⟨*proof*⟩

lemma *pswap-pfresh-iff*[*simp*]:
pfresh (*sw x z1 z2*) (*pswap s z1 z2*) ⟷ *pfresh x s*
⟨*proof*⟩

lemma *pfresh-pswap-iff*:
pfresh x (*pswap s z1 z2*) = *pfresh* (*sw x z1 z2*) *s*
⟨*proof*⟩

inductive *alpha* :: *ptrm* ⇒ *ptrm* ⇒ *bool* **where**
PVr[*intro*]: *alpha* (*PVr x*) (*PVr x*)
|*PAp*[*intro*]: *alpha s s′* ⟹ *alpha t t′* ⟹ *alpha* (*PAp s t*) (*PAp s′ t′*)
|*PLm*[*intro*]:
(*z* = *x* ∨ *pfresh z t*) ⟹ (*z* = *x′* ∨ *pfresh z t′*)
⟹ *alpha* (*pswap t z x*) (*pswap t′ z x′*) ⟹ *alpha* (*PLm x t*) (*PLm x′ t′*)

lemma *alpha-PVr-eq*[*simp*]: *alpha* (*PVr x*) *t* ⟷ *t* = *PVr x*
⟨*proof*⟩

lemma *alpha-eq-PVr*[*simp*]: *alpha t* (*PVr x*) ⟷ *t* = *PVr x*
⟨*proof*⟩

lemma *alpha-PAp-cases*[*elim*, *case-names PApc*]:
assumes *alpha* (*PAp s1 s2*) *t*
obtains *t1 t2* **where** *t* = *PAp t1 t2* **and** *alpha s1 t1* **and** *alpha s2 t2*
⟨*proof*⟩

lemma *alpha-PAp-cases2*[*elim*, *case-names PApc*]:
assumes *alpha t* (*PAp s1 s2*)
obtains *t1 t2* **where** *t* = *PAp t1 t2* **and** *alpha t1 s1* **and** *alpha t2 s2*
⟨*proof*⟩

lemma *alpha-PLm-cases*[*elim*, *case-names PLmc*]:
assumes *alpha* (*PLm x s*) *t'*
obtains *x' s' z* **where** *t'* = *PLm x' s'*
and *z* = *x* ∨ *pfresh z s* **and** *z* = *x'* ∨ *pfresh z s'*
and *alpha* (*pswap s z x*) (*pswap s' z x'*)
⟨*proof*⟩

lemma *alpha-pswap*:
assumes *alpha s s'* **shows** *alpha* (*pswap s z1 z2*) (*pswap s' z1 z2*)
⟨*proof*⟩

lemma *alpha-refl*[*simp*,*intro*!]: *alpha s s*
⟨*proof*⟩

lemma *alpha-sym*:
assumes *alpha s t* **shows** *alpha t s*
⟨*proof*⟩

lemma *alpha-pfresh-imp*:
assumes *alpha s t* **and** *pfresh x s* **shows** *pfresh x t*
⟨*proof*⟩

lemma *alpha-pfresh-iff*:
assumes *alpha s t*
shows *pfresh x s* ⟷ *pfresh x t*
⟨*proof*⟩

lemma *pswap-pfresh-alpha*:
assumes *pfresh z1 t* **and** *pfresh z2 t*
shows *alpha* (*pswap t z1 z2*) *t*
⟨*proof*⟩

fun *depth* :: *ptrm* ⇒ *nat* **where**
depth (*PVr x*) = *0*
|*depth* (*PAp t1 t2*) = *depth t1* + *depth t2* + *1*
|*depth* (*PLm x t*) = *depth t* + *1*

lemma *pswap-same-depth*:
depth (*pswap t1 x y*) = *depth t1*
⟨*proof*⟩

lemma *alpha-same-depth*:
assumes *alpha t1 t2* **shows** *depth t1 = depth t2*
⟨*proof*⟩

lemma *alpha-trans*:
assumes *alpha s t* **and** *alpha t u*
shows *alpha s u*
⟨*proof*⟩

lemma *alpha-PLm-strong-elim*:
assumes *alpha* (*PLm x t*) (*PLm x′ t′*)
and $z = x \vee$ *pfresh z t* **and** $z = x' \vee$ *pfresh z t′*
shows *alpha* (*pswap t z x*) (*pswap t′ z x′*)
⟨*proof*⟩

lemma *pfresh-pswap-alpha*:
assumes $y = x \vee$ *pfresh y t* **and** $z = x \vee$ *pfresh z t*
shows *alpha* (*pswap* (*pswap t y x*) *z y*) (*pswap t z x*)
⟨*proof*⟩

lemma *pfresh-sw-pswap-pswap*:
assumes *sw y′ z1 z2* $\neq y$ **and** $y =$ *sw x z1 z2* $\vee$ *pfresh y* (*pswap t z1 z2*)
and $y' = x \vee$ *pfresh y′ t*
shows *pfresh* (*sw y′ z1 z2*) (*pswap* (*pswap t z1 z2*) *y* (*sw x z1 z2*))
⟨*proof*⟩

1.3 Terms via quotienting pre-terms

quotient-type *trm* = *ptrm* / *alpha*
⟨*proof*⟩

lift-definition *Vr* :: *var* ⇒ *trm* **is** *PVr* ⟨*proof*⟩
lift-definition *Ap* :: *trm* ⇒ *trm* ⇒ *trm* **is** *PAp* ⟨*proof*⟩
lift-definition *Lm* :: *var* ⇒ *trm* ⇒ *trm* **is** *PLm* ⟨*proof*⟩
lift-definition *swap* :: *trm* ⇒ *var* ⇒ *var* ⇒ *trm* **is** *pswap*
⟨*proof*⟩
lift-definition *fresh* :: *var* ⇒ *trm* ⇒ *bool* **is** *pfresh*
⟨*proof*⟩
lift-definition *ddepth* :: *trm* ⇒ *nat* **is** *depth*
⟨*proof*⟩

lemma *abs-trm-rep-trm*[*simp*]: *abs-trm* (*rep-trm t*) = *t*
⟨*proof*⟩

lemma *alpha-rep-trm-abs-trm*[*simp*,*intro*!]: *alpha* (*rep-trm* (*abs-trm t*)) *t*
⟨*proof*⟩

lemma *pfresh-rep-trm-abs-trm*[*simp*]: *pfresh z* (*rep-trm* (*abs-trm t*)) ⟷ *pfresh z*

t
⟨*proof*⟩

lemma *swap-id*[*simp*]:
swap (*swap t z x*) *z x* = *t*
⟨*proof*⟩

lemma *fresh-PVr*[*simp*]: *fresh x* (*Vr y*) ⟷ *x* ≠ *y*
⟨*proof*⟩

lemma *fresh-Ap*[*simp*]: *fresh z* (*Ap t1 t2*) ⟷ *fresh z t1* ∧ *fresh z t2*
⟨*proof*⟩

lemma *fresh-Lm*[*simp*]: *fresh z* (*Lm x t*) ⟷ (*z* = *x* ∨ *fresh z t*)
⟨*proof*⟩

lemma *Lm-swap-rename*:
assumes *z* = *x* ∨ *fresh z t*
shows *Lm z* (*swap t z x*) = *Lm x t*
⟨*proof*⟩

lemma *abs-trm-PVr*: *abs-trm* (*PVr x*) = *Vr x*
⟨*proof*⟩

lemma *abs-trm-PAp*: *abs-trm* (*PAp t1 t2*) = *Ap* (*abs-trm t1*) (*abs-trm t2*)
⟨*proof*⟩

lemma *abs-trm-PLm*: *abs-trm* (*PLm x t*) = *Lm x* (*abs-trm t*)
⟨*proof*⟩

lemma *abs-trm-pswap*: *abs-trm* (*pswap t z1 z2*) = *swap* (*abs-trm t*) *z1 z2*
⟨*proof*⟩

lemma *swap-Vr*[*simp*]: *swap* (*Vr x*) *z1 z2* = *Vr* (*sw x z1 z2*)
⟨*proof*⟩

lemma *swap-Ap*[*simp*]: *swap* (*Ap t1 t2*) *z1 z2* = *Ap* (*swap t1 z1 z2*) (*swap t2 z1 z2*)
⟨*proof*⟩

lemma *swap-Lm*[*simp*]: *swap* (*Lm x t*) *z1 z2* = *Lm* (*sw x z1 z2*) (*swap t z1 z2*)
⟨*proof*⟩

lemma *Lm-sameVar-inj*[*simp*]: *Lm x t* = *Lm x t1* ⟷ *t* = *t1*
⟨*proof*⟩

lemma *Lm-eq-swap*:
assumes *Lm x t* = *Lm x1 t1*
shows *t* = *swap t1 x x1*

⟨*proof*⟩

lemma *alpha-rep-abs-trm*: *alpha* (*rep-trm* (*abs-trm t*)) *t*
⟨*proof*⟩

lemma *swap-fresh-eq*: **assumes** *x*:*fresh x t* **and** *y*:*fresh y t*
shows *swap t x y* = *t*
⟨*proof*⟩

lemma *bij-sw*:*bij* (λ *x*. *sw x z1 z2*)
⟨*proof*⟩

lemma *sw-set*: *x* ∈ *X* = ((*sw x z1 z2*) ∈ (λ *x*. *sw x z1 z2*) ‘ *X*)
⟨*proof*⟩

lemma *ddepth-Vr*[*simp*]: *ddepth* (*Vr x*) = *0*
⟨*proof*⟩

lemma *ddepth-Ap*[*simp*]: *ddepth* (*Ap t1 t2*) = *Suc* (*ddepth t1* + *ddepth t2*)
⟨*proof*⟩

lemma *ddepth-Lm*[*simp*]: *ddepth* (*Lm x t*) = *Suc* (*ddepth t*)
⟨*proof*⟩

lemma *trm-nchotomy*:
(∃ *x*. *tt* = *Vr x*) ∨ (∃ *t1 t2*. *tt* = *Ap t1 t2*) ∨ (∃ *x t*. *tt* = *Lm x t*)
⟨*proof*⟩

lemma *trm-exhaust*[*case-names Vr Ap Lm*, *cases type*: *trm*]:
(⋀*x*. *tt* = *Vr x* ⟹ *P*) ⟹
(⋀*t1 t2*. *tt* = *Ap t1 t2* ⟹ *P*) ⟹ (⋀*x t*. *tt* = *Lm x t* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *Vr-Ap-diff*[*simp*]: *Vr x* ≠ *Ap t1 t2* *Ap t1 t2* ≠ *Vr x*
⟨*proof*⟩

lemma *Vr-Lm-diff*[*simp*]: *Vr x* ≠ *Lm y t* *Lm y t* ≠ *Vr x*
⟨*proof*⟩

lemma *Ap-Lm-diff*[*simp*]: *Ap t1 t2* ≠ *Lm y t* *Lm y t* ≠ *Ap t1 t2*
⟨*proof*⟩

lemma *Vr-inj*[*simp*]: (*Vr x* = *Vr y*) ⟷ *x* = *y*
⟨*proof*⟩

lemma *Ap-inj*[*simp*]: (*Ap t1 t2* = *Ap t1′ t2′*) ⟷ *t1* = *t1′* ∧ *t2* = *t2′*
⟨*proof*⟩

abbreviation *Fvars* :: *ptrm* ⇒ *var set* **where**
Fvars t ≡ {*x*. ¬ *pfresh x t*}
abbreviation *FFvars* :: *trm* ⇒ *var set* **where**
FFvars t ≡ {*x*. ¬ *fresh x t*}

lemma *cofinite-fresh*: *finite* (*FFvars t*)
⟨*proof*⟩

lemma *exists-fresh-set*:
assumes *finite X*
shows ∃ *z*. *z* ∉ *X* ∧ *z* ∉ *set xs* ∧ (∀ *t* ∈ *set ts*. *fresh z t*)
⟨*proof*⟩

definition *ppickFreshS* :: *var set* ⇒ *var list* ⇒ *trm list* ⇒ *var* **where**
ppickFreshS X xs ts ≡ *SOME z*. *z* ∉ *X* ∧ *z* ∉ *set xs* ∧
(∀ *t* ∈ *set ts*. *fresh z t*)

lemma *ppickFreshS*:
assumes *finite X*
shows
ppickFreshS X xs ts ∉ *X* ∧
ppickFreshS X xs ts ∉ *set xs* ∧
(∀ *t* ∈ *set ts*. *fresh* (*ppickFreshS X xs ts*) *t*)
⟨*proof*⟩

lemmas *ppickFreshS-set* = *ppickFreshS*[*THEN conjunct1*]
and *ppickFreshS-var* = *ppickFreshS*[*THEN conjunct2*, *THEN conjunct1*]
and *ppickFreshS-ptrm* = *ppickFreshS*[*THEN conjunct2*, *THEN conjunct2*, *unfolded Ball-def*, *rule-format*]

definition *ppickFresh* ≡ *ppickFreshS* {}

lemmas *ppickFresh-var* = *ppickFreshS-var*[*OF finite.emptyI*, *unfolded ppickFresh-def*[*symmetric*]]
and *ppickFresh-ptrm* = *ppickFreshS-ptrm*[*OF finite.emptyI*, *unfolded ppickFresh-def*[*symmetric*]]

lemma *fresh-swap-nominal-style*:
fresh x t ⟷ *finite* {*y*. *swap t y x* ≠ *t*}
⟨*proof*⟩

1.4 Fresh induction

lemma *swap-induct*[*case-names Vr Ap Lm*]:
assumes *Vr*: ⋀*x*. *φ* (*Vr x*)
and *Ap*: ⋀*t1 t2*. *φ t1* ⟹ *φ t2* ⟹ *φ* (*Ap t1 t2*)
and *Lm*: ⋀*x t*. (∀ *z*. *φ* (*swap t z x*)) ⟹ *φ* (*Lm x t*)
shows *φ t*
⟨*proof*⟩

lemma *fresh-induct*[*consumes 1*, *case-names Vr Ap Lm*]:
assumes *finite X* **and** $\bigwedge x.\ \varphi\ (Vr\ x)$
and $\bigwedge t1\ t2.\ \varphi\ t1 \Longrightarrow \varphi\ t2 \Longrightarrow \varphi\ (Ap\ t1\ t2)$
and $\bigwedge x\ t.\ \varphi\ t \Longrightarrow x \notin X \Longrightarrow \varphi\ (Lm\ x\ t)$
shows $\varphi\ t$
⟨*proof*⟩

lemma *plain-induct*[*case-names Vr Ap Lm*]:
assumes $\bigwedge x.\ \varphi\ (Vr\ x)$
and $\bigwedge t1\ t2.\ \varphi\ t1 \Longrightarrow \varphi\ t2 \Longrightarrow \varphi\ (Ap\ t1\ t2)$
and $\bigwedge x\ t.\ \varphi\ t \Longrightarrow \varphi\ (Lm\ x\ t)$
shows $\varphi\ t$
⟨*proof*⟩

1.5 Substitution

inductive *substRel* :: $trm \Rightarrow trm \Rightarrow var \Rightarrow trm \Rightarrow bool$ **where**
substRel-Vr-same:
$substRel\ (Vr\ x)\ s\ x\ s$
|*substRel-Vr-diff*:
$x \neq y \Longrightarrow substRel\ (Vr\ x)\ s\ y\ (Vr\ x)$
|*substRel-Ap*:
$substRel\ t1\ s\ y\ t1' \Longrightarrow substRel\ t2\ s\ y\ t2' \Longrightarrow$
$substRel\ (Ap\ t1\ t2)\ s\ y\ (Ap\ t1'\ t2')$
|*substRel-Lm*:
$x \neq y \Longrightarrow fresh\ x\ s \Longrightarrow substRel\ t\ s\ y\ t' \Longrightarrow$
$substRel\ (Lm\ x\ t)\ s\ y\ (Lm\ x\ t')$

lemma *substRel-Vr-invert*:
assumes $substRel\ (Vr\ x)\ t\ y\ t'$
shows $(x = y \wedge t = t') \vee (x \neq y \wedge t' = Vr\ x)$
⟨*proof*⟩

lemma *substRel-Ap-invert*:
assumes $substRel\ (Ap\ t1\ t2)\ s\ y\ t'$
shows $\exists t1'\ t2'.\ t' = Ap\ t1'\ t2' \wedge substRel\ t1\ s\ y\ t1' \wedge substRel\ t2\ s\ y\ t2'$
⟨*proof*⟩

lemma *substRel-Lm-invert-aux*:
assumes $substRel\ (Lm\ x\ t)\ s\ y\ tt'$
shows $\exists x1\ t1\ t1'.$
$x1 \neq y \wedge fresh\ x1\ s \wedge$
$Lm\ x\ t = Lm\ x1\ t1 \wedge tt' = Lm\ x1\ t1' \wedge substRel\ t1\ s\ y\ t1'$
⟨*proof*⟩

lemma *substRel-swap*:
assumes $substRel\ t\ s\ y\ tt$
shows $substRel\ (swap\ t\ z1\ z2)\ (swap\ s\ z1\ z2)\ (sw\ y\ z1\ z2)\ (swap\ tt\ z1\ z2)$

⟨*proof*⟩

lemma *substRel-fresh*:
assumes *substRel t s y t'* **and** *fresh x1 t x1* ≠ *y fresh x1 s*
shows *fresh x1 t'*
⟨*proof*⟩

lemma *substRel-Lm-invert*:
assumes *substRel* (*Lm x t*) *s y tt'* **and** *0*: *x* ≠ *y fresh x s*
shows ∃ *t'*. *tt'* = *Lm x t'* ∧ *substRel t s y t'*
⟨*proof*⟩

lemma *substRel-total*:
∃ *t'*. *substRel t s y t'*
⟨*proof*⟩

lemma *substRel-functional*:
assumes *substRel t s y t'* **and** *substRel t s y tt'*
shows *t'* = *tt'*
⟨*proof*⟩

definition *subst* :: *trm* ⇒ *trm* ⇒ *var* ⇒ *trm* **where**
subst t s x ≡ *SOME tt*. *substRel t s x tt*

lemma *substRel-subst*: *substRel t s x* (*subst t s x*)
⟨*proof*⟩

lemma *substRel-subst-unique*: *substRel t s x tt* ⟹ *tt* = *subst t s x*
⟨*proof*⟩

lemma
subst-Vr[*simp*]: *subst* (*Vr x*) *t z* = (*if x* = *z then t else Vr x*)
and
subst-Ap[*simp*]: *subst* (*Ap s1 s2*) *t z* = *Ap* (*subst s1 t z*) (*subst s2 t z*)
and
subst-Lm[*simp*]:
x ≠ *z* ⟹ *fresh x t* ⟹ *subst* (*Lm x s*) *t z* = *Lm x* (*subst s t z*)
⟨*proof*⟩

lemma *fresh-subst*:
fresh z (*subst s t x*) ⟷ (*z* = *x* ∨ *fresh z s*) ∧ (*fresh x s* ∨ *fresh z t*)
⟨*proof*⟩

lemma *fresh-subst-id*[*simp*]:
assumes *fresh x s* **shows** *subst s t x* = *s*
⟨*proof*⟩

lemma *subst-Vr-id*[*simp*]: *subst s* (*Vr x*) *x* = *s*

⟨*proof*⟩

lemma *Lm-swap-cong*:
assumes $z = x \vee$ *fresh* $z\ s\ z = y \vee$ *fresh* $z\ t$ **and** *swap* $s\ z\ x =$ *swap* $t\ z\ y$
shows *Lm* $x\ s =$ *Lm* $y\ t$
⟨*proof*⟩

lemma *fresh-swap*[*simp*]: *fresh* x (*swap* $t\ z1\ z2$) $\longleftrightarrow$ *fresh* (*sw* $x\ z1\ z2$) t
⟨*proof*⟩

lemma *swap-subst*:
swap (*subst* $s\ t\ x$) $z1\ z2 =$ *subst* (*swap* $s\ z1\ z2$) (*swap* $t\ z1\ z2$) (*sw* $x\ z1\ z2$)
⟨*proof*⟩

lemma *subst-Lm-same*[*simp*]: *subst* (*Lm* $x\ s$) $t\ x =$ *Lm* $x\ s$
⟨*proof*⟩

lemma *fresh-subst-same*:
assumes $y \neq z$ **shows** *fresh* y (*subst* t (*Vr* z) y)
⟨*proof*⟩

lemma *subst-comp-same*:
subst (*subst* $s\ t\ x$) $t1\ x =$ *subst* s (*subst* $t\ t1\ x$) x
⟨*proof*⟩

lemma *subst-comp-diff*:
assumes $x \neq x1$ *fresh* $x\ t1$
shows *subst* (*subst* $s\ t\ x$) $t1\ x1 =$ *subst* (*subst* $s\ t1\ x1$) (*subst* $t\ t1\ x1$) x
⟨*proof*⟩

lemma *subst-comp-diff-var*:
assumes $x \neq x1\ x \neq z1$
shows *subst* (*subst* $s\ t\ x$) (*Vr* $z1$) $x1 =$
subst (*subst* s (*Vr* $z1$) $x1$) (*subst* t (*Vr* $z1$) $x1$) x
⟨*proof*⟩

lemma *subst-chain*:
assumes *fresh* $u\ s$
shows *subst* (*subst* s (*Vr* u) x) $t\ u =$ *subst* $s\ t\ x$
⟨*proof*⟩

lemma *subst-repeated-Vr*:
subst (*subst* t (*Vr* x) y) (*Vr* u) $x =$
subst (*subst* t (*Vr* u) x) (*Vr* u) y
⟨*proof*⟩

lemma *subst-commute-same*:
subst (*subst* d (*Vr* u) x) (*Vr* u) $y =$ *subst* (*subst* d (*Vr* u) y) (*Vr* u) x

⟨*proof*⟩

lemma *subst-commute-diff*:
assumes $x \neq v$ $y \neq u$ $x \neq y$
shows *subst* (*subst t* (*Vr u*) *x*) (*Vr v*) *y* = *subst* (*subst t* (*Vr v*) *y*) (*Vr u*) *x*
⟨*proof*⟩

lemma *subst-same-id*:
assumes *z1* ≠ *y*
shows *subst* (*subst t* (*Vr z1*) *y*) (*Vr z2*) *y* = *subst t* (*Vr z1*) *y*
⟨*proof*⟩

lemma *swap-from-subst*:
assumes *yy*: *yy*∉{*z1*,*z2*} *fresh yy t*
shows *swap t z1 z2* = *subst* (*subst* (*subst t* (*Vr yy*) *z1*) (*Vr z1*) *z2*) (*Vr z2*) *yy*
⟨*proof*⟩

lemma *subst-two-ways′*:
fixes *t yy x*
assumes *yy*: *yy*∉{*z1*,*z2*} *yy′*∉{*z1*,*z2*} *x* ∉ {*yy*,*yy′*}
defines *tt* ≡ *subst* (*subst t* (*Vr x*) *yy*) (*Vr x*) *yy′*
shows *subst* (*subst* (*subst tt* (*Vr yy*) *z1*) (*Vr z1*) *z2*) (*Vr z2*) *yy* =
subst (*subst* (*subst tt* (*Vr yy′*) *z1*) (*Vr z1*) *z2*) (*Vr z2*) *yy′*
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *subst-two-ways′′*:
assumes *xx* ∉ {*x*, *z1*, *z2*, *uu*, *vv*} ∧ *fresh xx t*
vv ∉ {*x*, *z1*, *z2*} ∧ *fresh vv t*
yy ∉ {*z1*, *z2*} ∧ *fresh yy t*
shows
subst (*subst* (*subst* (*subst* (*subst t* (*Vr xx*) *x*) (*Vr vv*) *z1*) (*Vr z1*) *z2*) (*Vr z2*) *vv*) (*Vr vv*) *xx* =
subst (*subst* (*subst* (*subst t* (*Vr yy*) *z1*) (*Vr z1*) *z2*) (*Vr z2*) *yy*) (*Vr vv*) (*sw x z1 z2*)
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *subst-two-ways′′-aux*:
fixes *t z1 xx z2 vv*
assumes *xx* ∉ {*x*, *z1*, *z2*, *uu*, *vv*}
vv ∉ {*x*, *z1*, *z2*}
yy ∉ {*z1*, *z2*}
defines *tt* ≡ *subst* (*subst* (*subst t* (*Vr z1*) *xx*) (*Vr z1*) *yy*) (*Vr z1*) *vv*
shows
subst (*subst* (*subst* (*subst* (*subst tt* (*Vr xx*) *x*) (*Vr vv*) *z1*) (*Vr z1*) *z2*) (*Vr z2*) *vv*) (*Vr vv*) *xx* =
subst (*subst* (*subst* (*subst tt* (*Vr yy*) *z1*) (*Vr z1*) *z2*) (*Vr z2*) *yy*) (*Vr vv*) (*sw x z1*

z2)
⟨*proof*⟩

lemma *fresh-cases*[*cases pred*: *fresh*, *case-names Vr Ap Lm*]:
fresh a1 a2 ⟹
(⋀*z x*. *a1* = *z* ⟹ *a2* = *Vr x* ⟹ *z* ≠ *x* ⟹ *P*) ⟹
(⋀*z t1 t2*. *a1* = *z* ⟹ *a2* = *Ap t1 t2* ⟹ *fresh z t1* ⟹ *fresh z t2* ⟹ *P*) ⟹
(⋀*z x t*. *a1* = *z* ⟹ *a2* = *Lm x t* ⟹ *z* = *x* ∨ *fresh z t* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

definition *vss* :: *var* ⇒ *var* ⇒ *var* ⇒ *var* **where**
vss x y z = (*if x* = *z then y else x*)

lemma *fresh-subst-eq-swap*:
assumes *fresh z t*
shows *subst t* (*Vr z*) *x* = *swap t z x*
⟨*proof*⟩

lemma *Lm-subst-rename*:
assumes *z* = *x* ∨ *fresh z t*
shows *Lm z* (*subst t* (*Vr z*) *x*) = *Lm x t*
⟨*proof*⟩

lemma *Lm-subst-cong*:
z = *x* ∨ *fresh z s* ⟹ *z* = *y* ∨ *fresh z t* ⟹
subst s (*Vr z*) *x* = *subst t* (*Vr z*) *y* ⟹ *Lm x s* = *Lm y t*
⟨*proof*⟩

lemma *Lm-eq-elim*:
Lm x s = *Lm y t* ⟹ *z* = *x* ∨ *fresh z s* ⟹ *z* = *y* ∨ *fresh z t*
⟹ *swap s z x* = *swap t z y*
⟨*proof*⟩

lemma *Lm-eq-elim-subst*:
Lm x s = *Lm y t* ⟹ *z* = *x* ∨ *fresh z s* ⟹ *z* = *y* ∨ *fresh z t*
⟹
subst s (*Vr z*) *x* = *subst t* (*Vr z*) *y*
⟨*proof*⟩

1.6 Renaming (a.k.a. variable-for-variable substitution)

abbreviation *vsubst* **where** *vsubst* ≡ λ*t x y*. *subst t* (*Vr x*) *y*

inductive *substConnect* :: *trm* ⇒ *trm* ⇒ *bool* **where**
Refl: *substConnect t t*

| *Step*: *substConnect t t'* $\Longrightarrow$ *substConnect t* (*vsubst t' z x*)

lemma *ddepth-swap*:
ddepth (*swap t z x*) = *ddepth t*
⟨*proof*⟩

lemma *ddepth-subst-Vr*[*simp*]:
ddepth (*vsubst t z x*) = *ddepth t*
⟨*proof*⟩

lemma *substConnect-depth*:
assumes *substConnect t t'* **shows** *ddepth t* = *ddepth t'*
⟨*proof*⟩

lemma *substConnect-induct*[*case-names Vr Ap Lm*]:
assumes *Vr*: $\bigwedge x.\ \varphi\ (Vr\ x)$
and *Ap*: $\bigwedge t1\ t2.\ \varphi\ t1 \Longrightarrow \varphi\ t2 \Longrightarrow \varphi\ (Ap\ t1\ t2)$
and *Lm*: $\bigwedge x\ t.\ (\forall t'.\ substConnect\ t\ t' \longrightarrow \varphi\ t') \Longrightarrow \varphi\ (Lm\ x\ t)$
shows $\varphi\ t$
⟨*proof*⟩

1.7 Syntactic environments

typedef *fenv* = {*f* :: *var* ⇒ *trm* . *finite* {*x*. *f x* ≠ *Vr x*}}
⟨*proof*⟩

definition *get* :: *fenv* ⇒ *var* ⇒ *trm* **where**
get f x ≡ *Rep-fenv f x*

definition *upd* :: *fenv* ⇒ *var* ⇒ *trm* ⇒ *fenv* **where**
upd f x t = *Abs-fenv* ((*Rep-fenv f*)(*x*:=*t*))

definition *supp* :: *fenv* ⇒ *var set* **where**
supp f ≡ {*x*. *get f x* ≠ *Vr x*}

lemma *finite-supp*: *finite* (*supp f*)
⟨*proof*⟩

lemma *finite-upd*:
assumes *finite* {*x*. *f x* ≠ *Vr x*}
shows *finite* {*x*. (*f*(*y*:=*t*)) *x* ≠ *Vr x*}
⟨*proof*⟩

lemma *get-upd-same*[*simp*]: *get* (*upd f x t*) *x* = *t*
and *get-upd-diff*[*simp*]: *x* ≠ *y* ⟹ *get* (*upd f x t*) *y* = *get f y*
and *upd-upd-same*[*simp*]: *upd* (*upd f x t*) *x s* = *upd f x s*
and *upd-upd-diff*: *x* ≠ *y* ⟹ *upd* (*upd f x t*) *y s* = *upd* (*upd f y s*) *x t*
and *supp-get*[*simp*]: *x* ∉ *supp* ϱ ⟹ *get* ϱ *x* = *Vr x*
⟨*proof*⟩

end

2 Renaming-Enriched Sets (Rensets)

theory *Rensets*
imports *Lambda-Terms*
begin

This theory defines rensets and proves their basic properties.

2.1 Rensets

locale *Renset* =
fixes *vsubstA* :: *'A* ⇒ *var* ⇒ *var* ⇒ *'A*
assumes
vsubstA-id[*simp*]: ⋀*x a*. *vsubstA a x x* = *a*
and
vsubstA-idem[*simp*]: ⋀*x y1 y2 a*. *y1* ≠ *x* ⟹ *vsubstA* (*vsubstA a y1 x*) *y2 x* = *vsubstA a y1 x*
and
vsubstA-chain: ⋀*u x1 x2 x3 a*.
u ≠ *x2* ⟹
vsubstA (*vsubstA* (*vsubstA a u x2*) *x2 x1*) *x3 x2* =
vsubstA (*vsubstA a u x2*) *x3 x1*
and
vsubstA-commute-diff:
⋀ *x y u a v*. *x* ≠ *v* ⟹ *y* ≠ *u* ⟹ *x* ≠ *y* ⟹
vsubstA (*vsubstA a u x*) *v y* = *vsubstA* (*vsubstA a v y*) *u x*
begin

definition *freshA* **where** *freshA x a* ≡ *finite* {*y*. *vsubstA a y x* ≠ *a*}

lemma *freshA-vsubstA-idle*:
assumes *n*: *freshA x a* **and** *xy*: *x* ≠ *y*
shows *vsubstA a y x* = *a*
⟨*proof*⟩

lemma *vsubstA-chain-freshA*:
assumes *freshA x2 a*
shows *vsubstA* (*vsubstA a x2 x1*) *x3 x2* = *vsubstA a x3 x1*
⟨*proof*⟩

lemma *freshA-vsubstA*:
assumes *freshA u a* **and** *u* ≠ *y*
shows *freshA u* (*vsubstA a y x*)

⟨*proof*⟩

lemma *freshA-vsubstA2*:
assumes *freshA z a* ∨ *z* = *x* **and** *freshA x a* ∨ *z* ≠ *y*
shows *freshA z* (*vsubstA a y x*)
⟨*proof*⟩

lemma *vsubstA-idle-freshA*:
assumes *vsubstA a y x* = *a* **and** *xy*: *x* ≠ *y*
shows *freshA x a*
⟨*proof*⟩

lemma *freshA-iff-ex-vvsubstA-idle*:
freshA x a ⟷ (∃ *y*. *y*≠*x* ∧ *vsubstA a y x* = *a*)
⟨*proof*⟩

lemma *freshA-iff-all-vvsubstA-idle*:
freshA x a ⟷ (∀ *y*. *y*≠*x* ⟶ *vsubstA a y x* = *a*)
⟨*proof*⟩

end

2.2 Finitely supported rensets

locale *Renset-FinSupp* = *Renset vsubstA*
for *vsubstA* :: *'A* ⇒ *var* ⇒ *var* ⇒ *'A*
+
assumes *cofinite-freshA*: ⋀*a*. *finite* {*x*. ¬ *freshA x a*}
begin

definition *pickFreshSA* :: *var set* ⇒ *var list* ⇒ *'A list* ⇒ *var* **where**
pickFreshSA X xs ds ≡ *SOME z*. *z* ∉ *X* ∧ *z* ∉ *set xs* ∧ (∀ *a* ∈ *set ds*. *freshA z a*)

lemma *exists-freshA-set*:
assumes *finite X*
shows ∃ *z*. *z* ∉ *X* ∧ *z* ∉ *set xs* ∧ (∀ *a* ∈ *set ds*. *freshA z a*)
⟨*proof*⟩

lemma *exists-freshA*:
∃ *z*. *z* ∉ *set xs* ∧ (∀ *a* ∈ *set ds*. *freshA z a*)
⟨*proof*⟩

lemma *pickFreshSA*:
assumes *finite X*
shows
pickFreshSA X xs ds ∉ *X* ∧

pickFreshSA X xs ds $\notin$ *set xs* $\wedge$
$(\forall\, a \in$ *set ds. freshA* (*pickFreshSA X xs ds*) *a*)
⟨*proof*⟩

lemmas *pickFreshSA-set* = *pickFreshSA*[*THEN conjunct1*]
and *pickFreshSA-var* = *pickFreshSA*[*THEN conjunct2*, *THEN conjunct1*]
and *pickFreshSA-freshA* = *pickFreshSA*[*THEN conjunct2*, *THEN conjunct2*, *unfolded Ball-def*, *rule-format*]

definition *pickFreshA* ≡ *pickFreshSA* {}

lemmas *pickFreshA* = *pickFreshSA*[*OF finite.emptyI*, *unfolded pickFreshA-def*[*symmetric*], *simplified*]
lemmas *pickFreshA-var* = *pickFreshSA-var*[*OF finite.emptyI*, *unfolded pickFreshA-def*[*symmetric*]]
and *pickFreshA-freshA* = *pickFreshSA-freshA*[*OF finite.emptyI*, *unfolded pickFreshA-def*[*symmetric*]]

end

2.3 Morphisms between rensets

locale *Renset-Morphism* =
A: *Renset-FinSupp substA* + *B*: *Renset-FinSupp substB*
for *substA* :: *'A* ⇒ *var* ⇒ *var* ⇒ *'A* **and** *substB* :: *'B* ⇒ *var* ⇒ *var* ⇒ *'B*
+
fixes *f* :: *'A* ⇒ *'B*
assumes *f-substA-substB*: $\bigwedge a\ y\ z.$ *f* (*substA a y z*) = *substB* (*f a*) *y z*

end

3 Nominal sets

theory *Nominal-Sets*
imports *Lambda-Terms*
begin

This theory introduces pre-nominal sets, and then nominal sets as pre-nominal sets of finite support.

locale *Pre-Nominal-Set* =
fixes *swapA* :: *'A* ⇒ *var* ⇒ *var* ⇒ *'A*
assumes
swapA-id: $\bigwedge a\ x.$ *swapA a x x* = *a*
and
swapA-invol: $\bigwedge a\ x\ y.$ *swapA* (*swapA a x y*) *x y* = *a*
and
swapA-cmp:

$\bigwedge x\ y\ a\ z1\ z2.\ swapA\ (swapA\ a\ x\ y)\ z1\ z2 =$
$swapA\ (swapA\ a\ z1\ z2)\ (sw\ x\ z1\ z2)\ (sw\ y\ z1\ z2)$
begin

definition *freshA* **where** $freshA\ x\ a \equiv finite\ \{y.\ swapA\ a\ y\ x \neq a\}$

end

locale *Nominal-Set* = *Pre-Nominal-Set swapA*
for $swapA :: 'A \Rightarrow var \Rightarrow var \Rightarrow 'A$
+
assumes *cofinite-freshA*: $\bigwedge a.\ finite\ \{x.\ \neg\ freshA\ x\ a\}$

locale *Nominal-Morphism* =
A: *Nominal-Set swapA* + *B*: *Nominal-Set swapB*
for $swapA :: 'A \Rightarrow var \Rightarrow var \Rightarrow 'A$ **and** $swapB :: 'B \Rightarrow var \Rightarrow var \Rightarrow 'B$
+
fixes $f :: 'A \Rightarrow 'B$
assumes *f-swapA-swapB*: $\bigwedge a\ z1\ z2.\ f\ (swapA\ a\ z1\ z2) = swapB\ (f\ a)\ z1\ z2$

end

3.1 From Rensets to Nominal Sets

theory *Rensets-to-Nominal-Sets*
imports *Rensets Nominal-Sets*
begin

This theory shows that any finitely supported rensets gives rise to a nominal set. This is done by defining swapping from renaming.

context *Renset-FinSupp*
begin

definition $swapA :: 'A \Rightarrow var \Rightarrow var \Rightarrow 'A$ **where**
$swapA\ a\ z1\ z2 \equiv$
$let\ yy = pickFreshA\ [z1,z2]\ [a]\ in$
$vsubstA\ (vsubstA\ (vsubstA\ a\ yy\ z1)\ z1\ z2)\ z2\ yy$

lemma *swapA*:
$\exists yy.\ yy \notin \{z1,z2\} \wedge freshA\ yy\ a \wedge$
$swapA\ a\ z1\ z2 = vsubstA\ (vsubstA\ (vsubstA\ a\ yy\ z1)\ z1\ z2)\ z2\ yy$
⟨*proof*⟩

lemma *swapA-id*[*simp*]:
swapA a z z = *a*
⟨*proof*⟩

lemma *vvsubstA-twoWays*:
assumes *uu* ≠ *x* ∧ *uu* ≠ *y* ∧ *freshA uu a vv* ≠ *x* ∧ *vv* ≠ *y* ∧ *freshA vv a*
shows *vsubstA* (*vsubstA* (*vsubstA a uu x*) *x y*) *y uu* =
vsubstA (*vsubstA* (*vsubstA a vv x*) *x y*) *y vv*
⟨*proof*⟩

lemma *swapA-any*:
assumes *uu* ≠ *x* ∧ *uu* ≠ *y* ∧ *freshA uu a*
shows *swapA a x y* = *vsubstA* (*vsubstA* (*vsubstA a uu x*) *x y*) *y uu*
⟨*proof*⟩

lemma *swapA-invol*[*simp*]: *swapA* (*swapA a x y*) *x y* = *a*
⟨*proof*⟩

lemma *swapA-cmp*:
swapA (*swapA a x y*) *z1 z2* = *swapA* (*swapA a z1 z2*) (*sw x z1 z2*) (*sw y z1 z2*)
⟨*proof*⟩

lemma *freshA-swapA-vsubstA*:
assumes *freshA y a*
shows *swapA a y x* = *vsubstA a y x*
⟨*proof*⟩

end

sublocale *Renset-FinSupp* < *Sw*: *Pre-Nominal-Set* **where** *swapA* = *swapA*
⟨*proof*⟩

context *Renset-FinSupp*
begin

lemma *freshA-swapA*: *freshA x a* ⟷ *Sw.freshA x a*
⟨*proof*⟩

end

The statement that any finitely supported renset produces a nominal set is written as sublocale inclusions.

... the object component:

sublocale *Renset-FinSupp* < *Sw*: *Nominal-Set* **where** *swapA* = *swapA*
⟨*proof*⟩

... the morphism component:

sublocale *Renset-Morphism* < *F*: *Nominal-Morphism* **where**
swapA = *A.swapA* **and** *swapB* = *B.swapA* **and** *f* = *f*
⟨*proof*⟩

end

4 Renset-based Recursion

theory *FRBCE-Rensets*
imports *Rensets*
begin

In this theory we prove that lambda-terms (modulo alpha) form the initial renset. This gives rise to a recursion principle, which we further enhance with support for the Barendregt variable convention (similarly to the nominal recursion).

5 Full-fledged, Barendregt-constrctor-enriched recursion

locale *FR-BCE-Renset* = *Renset vsubstA*
for *vsubstA* :: *'A* ⇒ *var* ⇒ *var* ⇒ *'A*
+
fixes
X :: *var set*

and *VrA* :: *var* ⇒ *'A*
and *ApA* :: *trm* ⇒ *'A* ⇒ *trm* ⇒ *'A* ⇒ *'A*
and *LmA* :: *var* ⇒ *trm* ⇒ *'A* ⇒ *'A*
assumes
finite-X[*simp*,*intro*!]: *finite X*
and
vsubstA-VrA: ⋀ *x y z*. {*y*,*z*} ∩ *X* = {} ⟹
vsubstA (*VrA x*) *y z* = (*if x* = *z then VrA y else VrA x*)
and
vsubstA-ApA: ⋀ *y z t1 a1 t2 a2*. {*y*,*z*} ∩ *X* = {} ⟹
vsubstA (*ApA t1 a1 t2 a2*) *y z* =
ApA (*vsubst t1 y z*) (*vsubstA a1 y z*)
(*vsubst t2 y z*) (*vsubstA a2 y z*)
and
vsubstA-LmA: ⋀ *t a z x y*. {*x*,*y*,*z*} ∩ *X* = {} ⟹

$x \neq y \Longrightarrow$
$vsubstA\ (LmA\ x\ t\ a)\ y\ z =$
$(if\ x = z\ then\ LmA\ x\ t\ a\ else\ LmA\ x\ (vsubst\ t\ y\ z)\ (vsubstA\ a\ y\ z))$
and
LmA-rename: $\bigwedge x\ y\ z\ t\ a.\ \{x,y,z\} \cap X = \{\} \Longrightarrow$
$z \neq y \Longrightarrow$
$LmA\ x\ (vsubst\ t\ z\ y)\ (vsubstA\ a\ z\ y) =$
$LmA\ y\ (vsubst\ (vsubst\ t\ z\ y)\ y\ x)\ (vsubstA\ (vsubstA\ a\ z\ y)\ y\ x)$
begin

lemma *LmA-cong*:
$\{u,z,x,x'\} \cap X = \{\} \Longrightarrow$
$z \neq u \Longrightarrow$
$z \neq x \Longrightarrow z \neq x' \Longrightarrow$
$vsubst\ (vsubst\ t\ u\ z)\ z\ x = vsubst\ (vsubst\ t'\ u\ z)\ z\ x' \Longrightarrow$
$vsubstA\ (vsubstA\ a\ u\ z)\ z\ x = vsubstA\ (vsubstA\ a'\ u\ z)\ z\ x'$
$\Longrightarrow LmA\ x\ (vsubst\ t\ u\ z)\ (vsubstA\ a\ u\ z) =$
$LmA\ x'\ (vsubst\ t'\ u\ z)\ (vsubstA\ a'\ u\ z)$
⟨*proof*⟩

lemma *vsubstA-LmA-same*:
$\{x,y\} \cap X = \{\} \Longrightarrow vsubstA\ (LmA\ x\ t\ a)\ y\ x = LmA\ x\ t\ a$
⟨*proof*⟩

lemma *vsubstA-LmA-diff*:
$\{x,y,z\} \cap X = \{\} \Longrightarrow$
$x \neq y \Longrightarrow x \neq z \Longrightarrow vsubstA\ (LmA\ x\ t\ a)\ y\ z = LmA\ x\ (vsubst\ t\ y\ z)\ (vsubstA\ a\ y\ z)$
⟨*proof*⟩

lemma *freshA-2-vsubstA*:
assumes $freshA\ z\ a\ freshA\ z\ a'$
shows $\exists u.\ u \notin X \wedge u \neq z \wedge vsubstA\ a\ u\ z = a \wedge vsubstA\ a'\ u\ z = a'$
⟨*proof*⟩

lemma *LmA-cong-freshA*:
assumes $\{z,x,x'\} \cap X = \{\}$
and $z \neq x\ fresh\ z\ t\ freshA\ z\ a$
and $z \neq x'\ fresh\ z\ t'\ freshA\ z\ a'$
and $vsubst\ t\ z\ x = vsubst\ t'\ z\ x'$
and $vsubstA\ a\ z\ x = vsubstA\ a'\ z\ x'$
shows $LmA\ x\ t\ a = LmA\ x'\ t'\ a'$
⟨*proof*⟩

lemma *freshA-VrA*: $z \notin X \Longrightarrow z \neq x \Longrightarrow freshA\ z\ (VrA\ x)$
⟨*proof*⟩

lemma *freshA-ApA*: $z \notin X \Longrightarrow$
$fresh\ z\ t1 \Longrightarrow freshA\ z\ a1 \Longrightarrow$

fresh z t2 $\Longrightarrow$ *freshA z a2* $\Longrightarrow$
freshA z (*ApA t1 a1 t2 a2*)
⟨*proof*⟩

lemma *freshA-LmA-same*:
assumes $x \notin X$
shows *freshA x* (*LmA x t a*)
⟨*proof*⟩

lemma *freshA-LmA'*:
assumes $\{x,z\} \cap X = \{\}$ *fresh z t freshA z a*
shows *freshA z* (*LmA x t a*)
⟨*proof*⟩

lemma *LmA-rename-freshA*:
assumes $\{x,z\} \cap X = \{\}$ $z \neq x$ *fresh z t freshA z a*
shows *LmA x t a* = *LmA z* (*vsubst t z x*) (*vsubstA a z x*)
⟨*proof*⟩

lemma *freshA-LmA*:
$\{x,z\} \cap X = \{\} \Longrightarrow z = x \vee$ (*fresh z t* $\wedge$ *freshA z a*) $\Longrightarrow$ *freshA z* (*LmA x t a*)
⟨*proof*⟩

end

5.1 The relational version of the recursor

context *FR-BCE-Renset*
begin

The recursor is first defined relationally. Then it will be proved to be functional.

inductive *R* :: *trm* $\Rightarrow$ *'A* $\Rightarrow$ *bool* **where**
Vr: *R* (*Vr x*) (*VrA x*)
|
Ap: *R t1 a1* $\Longrightarrow$ *R t2 a2* $\Longrightarrow$ *R* (*Ap t1 t2*) (*ApA t1 a1 t2 a2*)
|
Lm: *R t a* $\Longrightarrow$ $x \notin X \Longrightarrow$ *R* (*Lm x t*) (*LmA x t a*)

lemma *F-Vr-elim*[*simp*]: *R* (*Vr x*) *a* $\longleftrightarrow$ *a* = *VrA x*
⟨*proof*⟩

lemma *F-Ap-elim*:
assumes *R* (*Ap t1 t2*) *a*
shows $\exists$ *a1 a2*. *R t1 a1* $\wedge$ *R t2 a2* $\wedge$ *a* = *ApA t1 a1 t2 a2*
⟨*proof*⟩

lemma *F-Lm-elim*:
assumes *R* (*Lm x t*) *a*

shows $\exists x'\ t'\ e.\ R\ t'\ e \wedge x' \notin X \wedge Lm\ x\ t = Lm\ x'\ t' \wedge a = LmA\ x'\ t'\ e$
⟨*proof*⟩

lemma *F-total*: $\exists a.\ R\ t\ a$
⟨*proof*⟩

The main lemma needed in the recursion theorem: It states that the relational version of the recursor is (1) functional, (2) preserves freshness and (3) preserves renaming. These three facts must be proved mutually recursively.

lemma *F-main*:
$(\forall a\ a'.\ R\ t\ a \longrightarrow R\ t\ a' \longrightarrow a = a') \wedge$
$(\forall a\ x.\ x \notin X \wedge fresh\ x\ t \wedge R\ t\ a \longrightarrow freshA\ x\ a) \wedge$
$(\forall a\ x\ y.\ x \notin X \wedge y \notin X \longrightarrow R\ t\ a \longrightarrow R\ (vsubst\ t\ y\ x)\ (vsubstA\ a\ y\ x))$
⟨*proof*⟩

lemmas *F-functional* = *F-main*[*THEN conjunct1*]
lemmas *F-fresh* = *F-main*[*THEN conjunct2*, *THEN conjunct1*]
lemmas *F-subst* = *F-main*[*THEN conjunct2*, *THEN conjunct2*]

5.2 The functional version of the recursor

definition $f :: trm \Rightarrow 'A$ **where** $f\ t \equiv SOME\ a.\ R\ t\ a$

lemma *F-f*: $R\ t\ (f\ t)$
⟨*proof*⟩

lemma *f-eq-F*: $f\ t = a \longleftrightarrow R\ t\ a$
⟨*proof*⟩

5.3 The full-fledged recursion theorem

theorem *f-Vr*[*simp*]: $f\ (Vr\ x) = VrA\ x$
⟨*proof*⟩

theorem *f-Ap*[*simp*]: $f\ (Ap\ t1\ t2) = ApA\ t1\ (f\ t1)\ t2\ (f\ t2)$
⟨*proof*⟩

theorem *f-Lm*[*simp*]:
$x \notin X \Longrightarrow f\ (Lm\ x\ t) = LmA\ x\ t\ (f\ t)$
⟨*proof*⟩

theorem *f-subst*:
$y \notin X \Longrightarrow z \notin X \Longrightarrow f\ (subst\ t\ (Vr\ y)\ z) = vsubstA\ (f\ t)\ y\ z$
⟨*proof*⟩

theorem *f-fresh*:
assumes $z \notin X\ fresh\ z\ t$
shows $freshA\ z\ (f\ t)$

⟨*proof*⟩

theorem *f-unique*:
assumes [*simp*]: $\bigwedge x.\ g\ (Vr\ x) = VrA\ x$
$\bigwedge t1\ t2.\ g\ (Ap\ t1\ t2) = ApA\ t1\ (g\ t1)\ t2\ (g\ t2)$
$\bigwedge x\ t.\ x \notin X \Longrightarrow g\ (Lm\ x\ t) = LmA\ x\ t\ (g\ t)$
shows $g = f$
⟨*proof*⟩

end

5.4 The particular case of iteration

locale *BCE-Renset* = *Renset vsubstA*
for *vsubstA* :: $'A \Rightarrow var \Rightarrow var \Rightarrow 'A$
+
fixes
X :: *var set*

and *VrA* :: $var \Rightarrow 'A$
and *ApA* :: $'A \Rightarrow 'A \Rightarrow 'A$
and *LmA* :: $var \Rightarrow 'A \Rightarrow 'A$
assumes
finite-X'[*simp*,*intro*!]: *finite X*
and
vsubstA-VrA': $\bigwedge x\ y\ z.\ \{y,z\} \cap X = \{\} \Longrightarrow$
$vsubstA\ (VrA\ x)\ y\ z = (if\ x = z\ then\ VrA\ y\ else\ VrA\ x)$
and
vsubstA-ApA': $\bigwedge y\ z\ a1\ a2.\ \{y,z\} \cap X = \{\} \Longrightarrow$
$vsubstA\ (ApA\ a1\ a2)\ y\ z =$
$ApA\ (vsubstA\ a1\ y\ z)$
$(vsubstA\ a2\ y\ z)$
and
vsubstA-LmA': $\bigwedge a\ z\ x\ y.\ \{x,y,z\} \cap X = \{\} \Longrightarrow$
$x \neq y \Longrightarrow$
$vsubstA\ (LmA\ x\ a)\ y\ z = (if\ x = z\ then\ LmA\ x\ a\ else\ LmA\ x\ (vsubstA\ a\ y\ z))$
and
LmA-rename': $\bigwedge x\ y\ z\ a.\ \{x,y,z\} \cap X = \{\} \Longrightarrow$
$z \neq y \Longrightarrow LmA\ x\ (vsubstA\ a\ z\ y) = LmA\ y\ (vsubstA\ (vsubstA\ a\ z\ y)\ y\ x)$
begin

sublocale *FR-BCE-Renset* **where**
$VrA = VrA$ **and**
$ApA = \lambda t1\ a1\ t2\ a2.\ ApA\ a1\ a2$ **and**
$LmA = \lambda x\ t\ a.\ LmA\ x\ a$
⟨*proof*⟩

lemmas *f-clauses* = *f-Vr f-Ap f-Lm f-subst f-unique*

end

locale *CE-Renset* = *Renset vsubstA*
for *vsubstA* :: *'A* ⇒ *var* ⇒ *var* ⇒ *'A*
+
fixes

VrA :: *var* ⇒ *'A*
and *ApA* :: *'A* ⇒ *'A* ⇒ *'A*
and *LmA* :: *var* ⇒ *'A* ⇒ *'A*
assumes
vsubstA-VrA'': ⋀ *x y z*.
vsubstA (*VrA x*) *y z* = (*if x* = *z then VrA y else VrA x*)
and
vsubstA-ApA'': ⋀ *y z a1 a2*.
vsubstA (*ApA a1 a2*) *y z* =
ApA (*vsubstA a1 y z*)
(*vsubstA a2 y z*)
and
vsubstA-LmA'': ⋀ *a z x y*.
x ≠ *y* ⟹
vsubstA (*LmA x a*) *y z* = (*if x* = *z then LmA x a else LmA x* (*vsubstA a y z*))
and
LmA-rename'': ⋀ *x y z a*.
z ≠ *y* ⟹ *LmA x* (*vsubstA a z y*) = *LmA y* (*vsubstA* (*vsubstA a z y*) *y x*)
begin

sublocale *BCE-Renset* **where** *X* = {}
⟨*proof*⟩

lemma *triv*: *x*∉{} ⟨*proof*⟩

The initiality theorem

lemmas *f-clauses-init* = *f-Vr f-Ap f-Lm*[*OF triv*] *f-subst*[*OF triv triv*] *f-unique*[*simplified*]

end

end

6 Substitutive Sets

theory *Substitutive-Sets*

imports *FRBCE-Rensets*
begin

This theory describes a variation of the renset algebraic theory, including initiality and recursion principle, but focusing on term-for-variable rather than variable-for-variable substitution. Instead of rensets, we work with what we call substitutive sets.

6.1 Substitutive Sets

locale *Substitutive-Set* =
fixes *substA* :: $'A \Rightarrow 'A \Rightarrow var \Rightarrow 'A$
and *VrA* :: $var \Rightarrow 'A$
assumes *substA-id*[*simp*]: $\bigwedge x\ a.\ substA\ a\ (VrA\ x)\ x = a$
and *substA-idem*: $\bigwedge x\ b1\ b2\ a.\ u \neq x \Longrightarrow$
let $b1' = substA\ b1\ (VrA\ u)\ x$ *in* $substA\ (substA\ a\ b1'\ x)\ b2\ x = substA\ a\ b1'\ x$
and
substA-chain: $\bigwedge\ u\ x1\ x2\ b3\ a.\ u \neq x2 \Longrightarrow$
$substA\ (substA\ (substA\ a\ (VrA\ u)\ x2)\ (VrA\ x2)\ x1)\ b3\ x2 =$
$substA\ (substA\ a\ (VrA\ u)\ x2)\ b3\ x1$
and
substA-commute-diff:
$\bigwedge\ x\ y\ a\ e\ f.\ x \neq y \Longrightarrow u \neq y \Longrightarrow v \neq x \Longrightarrow$
let $e' = substA\ e\ (VrA\ u)\ y$; $f' = substA\ f\ (VrA\ v)\ x$ *in*
$substA\ (substA\ a\ e'\ x)\ f'\ y = substA\ (substA\ a\ f'\ y)\ e'\ x$
and
substA-VrA: $\bigwedge\ x\ a\ z.\ substA\ (VrA\ x)\ a\ z = (if\ x = z\ then\ a\ else\ VrA\ x)$
begin

lemma *substA-idem-var*[*simp*]:
$y1 \neq x \Longrightarrow substA\ (substA\ a\ (VrA\ y1)\ x)\ (VrA\ y2)\ x = substA\ a\ (VrA\ y1)\ x$
⟨*proof*⟩

lemma *substA-commute-diff-var*:
$x \neq v \Longrightarrow y \neq u \Longrightarrow x \neq y \Longrightarrow$
$substA\ (substA\ a\ (VrA\ u)\ x)\ (VrA\ v)\ y = substA\ (substA\ a\ (VrA\ v)\ y)\ (VrA\ u)\ x$
⟨*proof*⟩

end

Any substitutive set is in particular a renset:

sublocale *Substitutive-Set* < *Renset* **where**
$vsubstA = \lambda a\ x.\ substA\ a\ (VrA\ x)$ ⟨*proof*⟩

interpretation *STerm*: *Substitutive-Set* **where** *substA* = *subst* **and** *VrA* = *Vr*
⟨*proof*⟩

6.2 Constructor-Enriched (CE) Substitutive Sets

locale *CE-Substitutive-Set* = *Substitutive-Set substA VrA*
for *substA* :: *'A* ⇒ *'A* ⇒ *var* ⇒ *'A* **and** *VrA*
+
fixes
X :: *'A set*
and

ApA :: *'A* ⇒ *'A* ⇒ *'A*
and *LmA* :: *var* ⇒ *'A* ⇒ *'A*
assumes
substA-ApA: ⋀ *y z a1 a2*.
substA (*ApA a1 a2*) *y z* =
ApA (*substA a1 y z*)
(*substA a2 y z*)
and
substA-LmA: ⋀ *a z x e u*.
let e' = *substA e* (*VrA u*) *x in*
substA (*LmA x a*) *e' z* = (*if x* = *z then LmA x a else LmA x* (*substA a e' z*))
and
LmA-rename: ⋀ *x y z a*.
z ≠ *y* ⟹ *LmA x* (*substA a* (*VrA z*) *y*) = *LmA y* (*substA* (*substA a* (*VrA z*) *y*) (*VrA y*) *x*)
begin

lemma *LmA-cong*: ⋀ *z x x' a a' u*.
z ≠ *u* ⟹
z ≠ *x* ⟹ *z* ≠ *x'* ⟹
substA (*substA a* (*VrA u*) *z*) (*VrA z*) *x* = *substA* (*substA a'* (*VrA u*) *z*) (*VrA z*) *x'*
⟹ *LmA x* (*substA a* (*VrA u*) *z*) = *LmA x'* (*substA a'* (*VrA u*) *z*)
⟨*proof*⟩

lemma *substA-LmA-same*:
substA (*LmA x a*) *e x* = *LmA x a*
⟨*proof*⟩

lemma *substA-LmA-diff*:
freshA x e ⟹ *x* ≠ *z* ⟹ *substA* (*LmA x a*) *e z* = *LmA x* (*substA a e z*)
⟨*proof*⟩

lemma *freshA-2-substA*:
assumes *freshA z a freshA z a'*
shows ∃ *u*. *u* ≠ *z* ∧ *substA a* (*VrA u*) *z* = *a* ∧ *substA a'* (*VrA u*) *z* = *a'*
⟨*proof*⟩

lemma *LmA-cong-freshA*:
assumes *freshA z a freshA z a' substA a* (*VrA z*) *x* = *substA a'* (*VrA z*) *x'*
shows *LmA x a* = *LmA x' a'*

⟨*proof*⟩

lemma *freshA-VrA*: $z \neq x \Longrightarrow$ *freshA z* (*VrA x*)
⟨*proof*⟩

lemma *freshA-ApA*: $\bigwedge$ *z a1 a2*. *freshA z a1* $\Longrightarrow$ *freshA z a2* $\Longrightarrow$ *freshA z* (*ApA a1 a2*)
⟨*proof*⟩

lemma *freshA-LmA-same*:
freshA x (*LmA x a*)
⟨*proof*⟩

lemma *freshA-LmA*:
assumes *freshA z a*
shows *freshA z* (*LmA x a*)
⟨*proof*⟩

end

Any CE substitutive set is in particular a CE renset:

sublocale *CE-Substitutive-Set* < *CE-Renset*
where *vsubstA* = $\lambda a\ x$. *substA a* (*VrA x*)
⟨*proof*⟩

6.3 The recursion theorem for substitutive sets

context *CE-Substitutive-Set*
begin

lemmas *f-clauses′* = *f-Vr f-Ap f-Lm f-fresh f-subst f-unique*

theorem *f-subst-strong*:
f (*subst t s z*) = *substA* (*f t*) (*f s*) *z*
⟨*proof*⟩

end

end

7 Examples of Rensets and Renaming-Based Recursion

theory *Examples*
imports *FRBCE-Rensets Rensets*
begin

7.1 Variables and terms as rensets

Variables form a renset:

interpretation *Var*: *Renset* **where** *vsubstA* = *vss*
⟨*proof*⟩

Terms form a renset:

interpretation *Term*: *Renset* **where** *vsubstA* = λ*t x*. *vsubst t x*
⟨*proof*⟩

... and a CE renset:

interpretation *Term*: *CE-Renset*
where *vsubstA* = λ*t x*. *subst t* (*Vr x*)
and *VrA* = *Vr* **and** *ApA* = *Ap* **and** *LmA* = *Lm*
⟨*proof*⟩

7.2 Interpretation in semantic domains

type-synonym *'A I* = (*var* ⇒ *'A*) ⇒ *'A*

locale *Sem-Int* =
fixes *ap* :: *'A* ⇒ *'A* ⇒ *'A* **and** *lm* :: (*'A* ⇒ *'A*) ⇒ *'A*
begin

sublocale *CE-Renset*
where *vsubstA* = λ*s x y* ξ. *s* (ξ (*y* := ξ *x*))
and *VrA* = λ*x* ξ. ξ *x*
and *ApA* = λ*i1 i2* ξ. *ap* (*i1* ξ) (*i2* ξ)
and *LmA* = λ*x i* ξ. *lm* (λ*d*. *i* (ξ(*x*:=*d*)))
⟨*proof*⟩

lemmas *sem-f-clauses* = *f-Vr f-Ap f-Lm f-subst f-unique*

end

7.3 Closure of rensets under functors

A functor applied to a renset yields a renset – actually, a "local functor", i.e., one that is functorial w.r.t. functions on the substitutive set's carrier only, suffices.

locale *Local-Functor* =
fixes *Fmap* :: (*'A* ⇒ *'A*) ⇒ *'FA* ⇒ *'FA*
assumes *Fmap-id*: *Fmap id* = *id*
and *Fmap-comp*: *Fmap* (*g o f*) = *Fmap g o Fmap f*
begin

lemma *Fmap-comp'*: *Fmap* (*g o f*) *k* = *Fmap g* (*Fmap f k*)

⟨*proof*⟩

end

locale *Renset-plus-Local-Functor* =
Renset vsubstA + *Local-Functor Fmap*
for *vsubstA* :: $'A \Rightarrow var \Rightarrow var \Rightarrow 'A$
and *Fmap* :: $('A \Rightarrow 'A) \Rightarrow 'FA \Rightarrow 'FA$
begin

sublocale *F*: *Renset* **where** *vsubstA* =
$\lambda k\ x\ y.\ Fmap\ (\lambda a.\ vsubstA\ a\ x\ y)\ k$
⟨*proof*⟩

end

7.4 The length of a term via renaming-based recursion

interpretation *length* : *CE-Renset*
where *vsubstA* = $\lambda n\ x\ y.\ n$
and *VrA* = $\lambda x.\ 1$
and *ApA* = $\lambda n1\ n2.\ max\ n1\ n2 + 1$
and *LmA* = $\lambda x\ n.\ n + 1$
⟨*proof*⟩

lemmas *length-f-clauses* = *length.f-Vr length.f-Ap length.f-Lm length.f-subst length.f-unique*

7.5 Counting the lambda-abstractions in a term via renaming-based recursion

interpretation *clam* : *CE-Renset*
where *vsubstA* = $\lambda n\ x\ y.\ n$
and *VrA* = $\lambda x.\ 0$
and *ApA* = $\lambda n1\ n2.\ n1 + n2$
and *LmA* = $\lambda x\ n.\ n + 1$
⟨*proof*⟩

lemmas *clam-f-clauses* = *clam.f-Vr clam.f-Ap clam.f-Lm clam.f-subst clam.f-unique*

7.6 Counting free occurences of a variable in a term via renaming-based recursion

interpretation *cfv* : *CE-Renset*
where *vsubstA* =
$\lambda f\ z\ y.\ \lambda x.\ if\ x \notin \{y,z\}$

then f x
else if x = z ∧ x ≠ y then f x + f y
else if x = y ∧ x ≠ z then (0::nat)
else f y
and *VrA = λy. λx. if x = y then 1 else 0*
and *ApA = λf1 f2. λx. f1 x + f2 x*
and *LmA = λy f. λx. if x = y then 0 else f x*
⟨*proof*⟩

lemmas *cfv-f-clauses = cfv.f-Vr cfv.f-Ap cfv.f-Lm cfv.f-subst cfv.f-unique*

7.7 Substitution via renaming-based recursion

locale *Subst* =
fixes *s* :: *trm* **and** *x* :: *var*
begin

sublocale *ssb* : *BCE-Renset*
where *vsubstA = vsubst*
and *VrA = λy. if y = x then s else Vr y*
and *ApA = Ap*
and *LmA = Lm*
and *X = FFvars s ∪ {x}*
⟨*proof*⟩

lemmas *ssb-f-clauses = ssb.f-Vr ssb.f-Ap ssb.f-Lm ssb.f-subst ssb.f-unique*

lemma *subst-eq-ssb*:
subst t s x = ssb.f t
⟨*proof*⟩

end

7.8 Parrallel substitution via renaming-based recursion

locale *PSubst* =
fixes *ϱ* :: *fenv*
begin

definition *X* **where**
X = supp ϱ ∪ ⋃ {FFvars (get ϱ x) | x . x ∈ supp ϱ}

lemma *finite-Supp*: *finite X*
⟨*proof*⟩

sublocale *canEta′* : *BCE-Renset*

where *vsubstA* = *vsubst*
and *VrA* = $\lambda y.$ *get* ϱ *y*
and *ApA* = *Ap*
and *LmA* = *Lm*
and *X* = *X*
⟨*proof*⟩

lemmas *canEta′-f-clauses* = *canEta′.f-Vr canEta′.f-Ap canEta′.f-Lm canEta′.f-subst canEta′.f-unique*

end

7.9 Counting bound variables via renaming-based recursion

interpretation *cbvs*: *Sem-Int* **where** *ap* = (+) **and** *lm* = $\lambda d.$ *d* (*1*::*nat*) ⟨*proof*⟩

lemmas *cbvs-f-clauses* = *cbvs.f-Vr cbvs.f-Ap cbvs.f-Lm cbvs.f-subst cbvs.f-unique*

definition *cbv* :: *trm* ⇒ *nat* **where**
cbv t ≡ *cbvs.f t* (λ-. *0*)

7.10 Testing eta-reducibility via renaming-based recursion

interpretation *canEta′*: *Sem-Int* **where** *ap* = (∧) **and** *lm* = $\lambda d.$ *d True* ⟨*proof*⟩

lemmas *canEta′-f-clauses* = *canEta′.f-Vr canEta′.f-Ap canEta′.f-Lm canEta′.f-subst canEta′.f-unique*

definition *canEta* :: *trm* ⇒ *bool* **where**
canEta t ≡ ∃ *x s. t* = *Lm x* (*Ap s* (*Vr x*)) ∧ *canEta′.f s* ((λ-. *True*)(*x*:=*False*))

end
theory *All*

imports *Rensets-to-Nominal-Sets FRBCE-Rensets Substitutive-Sets Examples*

begin

end

References

[1] M. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *Logic in Computer Science (LICS) 1999*, pages 214–224. IEEE Computer Society, 1999.

[2] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2013.

[3] A. Popescu. Rensets and renaming-based recursion for syntax with bindings. In J. Blanchette, L. Kovács, and D. Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 618–639. Springer, 2022.