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

Andrei Popescu

March 17, 2025

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 **2**
1.1 Variables . . . 2
1.2 Pre-terms and operators on them . . . 3
1.3 Terms via quotienting pre-terms . . . 9
1.4 Fresh induction . . . 13
1.5 Substitution . . . 14
1.6 Renaming (a.k.a. variable-for-variable substitution) . . . 20
1.7 Syntactic environments . . . 21

2 Renaming-Enriched Sets (Rensets) **22**
2.1 Rensets . . . 22
2.2 Finitely supported rensets . . . 24
2.3 Morphisms between rensets . . . 25

3 Nominal sets **25**
3.1 From Rensets to Nominal Sets . . . 26

4 Renset-based Recursion **30**

5 Full-fledged, Barendregt-constrctor-enriched recursion **30**
5.1 The relational version of the recursor 33
5.2 The functional version of the recursor 36
5.3 The full-fledged recursion theorem 36
5.4 The particular case of iteration 37

6 Substitutive Sets **39**
6.1 Substitutive Sets 39
6.2 Constructor-Enriched (CE) Substitutive Sets 40
6.3 The recursion theorem for substitutive sets 41

7 Examples of Rensets and Renaming-Based Recursion **42**
7.1 Variables and terms as rensets 42
7.2 Interpretation in semantic domains 43
7.3 Closure of rensets under functors 43
7.4 The length of a term via renaming-based recursion 44
7.5 Counting the lambda-abstractions in a term via renaming-based recursion 44
7.6 Counting free occurences of a variable in a term via renaming-based recursion 44
7.7 Substitution via renaming-based recursion 45
7.8 Parrallel substitution via renaming-based recursion 45
7.9 Counting bound variables via renaming-based recursion . . . 46
7.10 Testing eta-reducibility via renaming-based recursion 46

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*
  using *pfresh.cases* **by** *blast+*

lemma *inj-Variable*: *inj Variable*
  unfolding *inj-def* **by** *auto*

lemma *infinite-var*: *infinite* (*UNIV*::*var set*)
  using *infinite-iff-countable-subset inj-Variable* **by** *auto*

lemma *exists-var*:
  assumes *finite X*
  shows ∃ *x*::*var*. $x \notin X$
  by (*simp add*: *assms ex-new-if-finite infinite-var*)

lemma *finite-neg-imp*:
  assumes *finite* {*x*. ¬ φ *x*} **and** *finite* {*x*. χ *x*}
  shows *finite* {*x*. φ *x* ⟶ χ *x*}
  using *finite-UnI*[*OF assms*] **by** (*simp add*: *Collect-imp-eq Collect-neg-eq*)

lemma *cofinite-pfresh*: *finite* {*x* . ¬ *pfresh x t*}
  by (*induct t*) (*simp-all add*: *finite-neg-imp*)

lemma *cofinite-list-ptrm*: *finite* {*x* . ∃ *t* ∈ *set ts*. ¬ *pfresh x t*}
proof (*induct ts*)
  case *Nil*
  then show *?case* **using** *infinite-var* **by** *auto*
next
  case (*Cons t ts*)
  have {*x*. ∃ *t*∈*set* (*t* # *ts*). ¬ *pfresh x t*} ⊆
   {*x*. ¬ *pfresh x t*} ∪ {*x*. ∃ *s*∈*set ts*. ¬ *pfresh x s*} **by** *auto*
  then show *?case* **using** *Cons*
    by (*simp add*: *cofinite-pfresh finite-subset*)
qed

lemma *exists-pfresh-set*:
  assumes *finite X*
  shows $\exists\ z.\ z \notin X \wedge z \notin set\ xs \wedge (\forall t \in set\ ts.\ pfresh\ z\ t)$
proof−
  have *0*: *finite* $(X \cup set\ xs \cup \{x.\ \exists s{\in}set\ ts.\ \neg\ pfresh\ x\ s\})$
    using *assms* **by** (*simp add*: *cofinite-list-ptrm*)
  show *?thesis* **using** *exists-var*[*OF 0*] **by** *simp*
qed

lemma *exists-pfresh*:
  $\exists\ z.\ z \notin set\ xs \wedge (\forall t \in set\ ts.\ pfresh\ z\ t)$
  using *exists-pfresh-set* **by** *blast*

definition *pickFreshS* :: $var\ set \Rightarrow var\ list \Rightarrow ptrm\ list \Rightarrow var$ **where**
  $pickFreshS\ X\ xs\ ts \equiv SOME\ z.\ z \notin X \wedge z \notin set\ xs \wedge (\forall t \in set\ ts.\ pfresh\ z\ t)$

lemma *pickFreshS*:
  assumes *finite X*
  shows $pickFreshS\ X\ xs\ ts \notin X \wedge pickFreshS\ X\ xs\ ts \notin set\ xs \wedge$
      $(\forall t \in set\ ts.\ pfresh\ (pickFreshS\ X\ xs\ ts)\ t)$
  using *exists-pfresh-set*[*OF assms*] **unfolding** *pickFreshS-def*
  by (*rule someI-ex*)

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* :: $var \Rightarrow var \Rightarrow var \Rightarrow var$ **where**
  $sw\ x\ y\ z \equiv if\ x = y\ then\ z\ else\ if\ x = z\ then\ y\ else\ x$

lemma *sw-eqL*[*simp*,*intro*!]: $\bigwedge x\ y\ z.\ sw\ x\ x\ y = y$
  and *sw-eqR*[*simp*,*intro*!]: $\bigwedge x\ y\ z.\ sw\ x\ y\ x = y$
  and *sw-diff*[*simp*]: $\bigwedge x\ y\ z.\ x \neq y \Longrightarrow x \neq z \Longrightarrow sw\ x\ y\ z = x$
  unfolding *sw-def* **by** *auto*

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*]: $sw\ (sw\ x\ y\ z)\ y\ z = x$

unfolding *sw-def* **by** *auto*

lemma *sw-invol2*: *sw* (*sw x y z*) *z y* = *x*
by (*simp add*: *sw-sym*)

lemma *sw-inj*[*iff*]: *sw x z1 z2* = *sw y z1 z2* ⟷ *x* = *y*
unfolding *sw-def* **by** *auto*

lemma *sw-surj*: ∃ *y*. *x* = *sw y z1 z2*
by (*metis sw-invol*)

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*
by (*induct s*) (*auto simp*: *sw-sym*)

lemma *pswap-id*[*simp*]: *pswap s y y* = *s*
by (*induct s*) *auto*

lemma *pswap-pswap*:
pswap (*pswap s y z*) *y1 z1* = *pswap* (*pswap s y1 z1*) (*sw y y1 z1*) (*sw z y1 z1*)
using *sw-sw* **by** (*induct s*) *auto*

lemma *pswap-invol*[*simp*]: *pswap* (*pswap s y z*) *y z* = *s*
by (*induct s*) *auto*

lemma *pswap-invol2*: *pswap* (*pswap s y z*) *z y* = *s*
by (*simp add*: *pswap-sym*)

lemma *pswap-inj*[*iff*]: *pswap s z1 z2* = *pswap t z1 z2* ⟷ *s* = *t*
by (*metis pswap-invol*)

lemma *pswap-surj*: ∃ *t*. *s* = *pswap t z1 z2*
by (*metis pswap-invol*)

lemma *pswap-pfresh-iff*[*simp*]:
pfresh (*sw x z1 z2*) (*pswap s z1 z2*) ⟷ *pfresh x s*
by (*induct s*) *auto*

lemma *pfresh-pswap-iff*:
pfresh x (*pswap s z1 z2*) = *pfresh* (*sw x z1 z2*) *s*
by (*metis sw-invol pswap-pfresh-iff*)

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*]:
  $(z = x \vee \textit{pfresh } z\ t) \Longrightarrow (z = x' \vee \textit{pfresh } z\ t')$
  $\Longrightarrow \textit{alpha } (\textit{pswap } t\ z\ x)\ (\textit{pswap } t'\ z\ x') \Longrightarrow \textit{alpha } (\textit{PLm } x\ t)\ (\textit{PLm } x'\ t')$

lemma *alpha-PVr-eq*[*simp*]: $\textit{alpha } (\textit{PVr } x)\ t \longleftrightarrow t = \textit{PVr } x$
  by (*auto elim*: *alpha.cases*)

lemma *alpha-eq-PVr*[*simp*]: $\textit{alpha } t\ (\textit{PVr } x) \longleftrightarrow t = \textit{PVr } x$
  by (*auto elim*: *alpha.cases*)

lemma *alpha-PAp-cases*[*elim*, *case-names PApc*]:
  assumes *alpha* (*PAp s1 s2*) *t*
  obtains *t1 t2* **where** $t = \textit{PAp t1 t2}$ **and** *alpha s1 t1* **and** *alpha s2 t2*
  using *assms* **by** (*auto elim*: *alpha.cases*)

lemma *alpha-PAp-cases2*[*elim*, *case-names PApc*]:
  assumes *alpha t* (*PAp s1 s2*)
  obtains *t1 t2* **where** $t = \textit{PAp t1 t2}$ **and** *alpha t1 s1* **and** *alpha t2 s2*
  using *assms* **by** (*auto elim*: *alpha.cases*)

lemma *alpha-PLm-cases*[*elim*, *case-names PLmc*]:
  assumes $\textit{alpha } (\textit{PLm } x\ s)\ t'$
  obtains $x'\ s'\ z$ **where** $t' = \textit{PLm } x'\ s'$
    and $z = x \vee \textit{pfresh } z\ s$ **and** $z = x' \vee \textit{pfresh } z\ s'$
    and $\textit{alpha } (\textit{pswap } s\ z\ x)\ (\textit{pswap } s'\ z\ x')$
  using *assms* **by** *cases auto*

lemma *alpha-pswap*:
  assumes $\textit{alpha } s\ s'$ **shows** $\textit{alpha } (\textit{pswap } s\ z1\ z2)\ (\textit{pswap } s'\ z1\ z2)$
  using *assms* **proof** *induct*
  case $(\textit{PLm } z\ x\ t\ x'\ t')$
  thus *?case*
    by (*auto intro*!: *alpha.PLm*[*of sw z z1 z2*]
        simp: *pswap-pswap*[*of* $t'\ x\ x'$] *pswap-pswap*[*of* $t\ x'\ x$]
        pswap-pswap[*of t z x*] *pswap-pswap*[*of* $t'\ z\ x'$])
qed *auto*

lemma *alpha-refl*[*simp*,*intro*!]: *alpha s s*
  by (*induct s*) *auto*

lemma *alpha-sym*:
  assumes *alpha s t* **shows** *alpha t s*
  using *assms* **by** *induct auto*

lemma *alpha-pfresh-imp*:
  assumes *alpha s t* **and** *pfresh x s* **shows** *pfresh x t*
  using *assms* **apply** *induct*
  by *simp-all* (*metis pfresh-pswap-iff sw-diff sw-eqR*)

```
lemma alpha-pfresh-iff:
  assumes alpha s t
  shows pfresh x s ⟷ pfresh x t
  using alpha-pfresh-imp alpha-sym assms by blast

lemma pswap-pfresh-alpha:
  assumes pfresh z1 t and pfresh z2 t
  shows alpha (pswap t z1 z2) t
  using assms proof(induct t)
  case (PLm z x t)
  thus ?case using alpha.PLm sw-def pswap-sym by fastforce
qed auto



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
  by(induct t1, simp-all)

lemma alpha-same-depth:
  assumes alpha t1 t2 shows depth t1 = depth t2
  using assms pswap-same-depth by induct auto

lemma alpha-trans:
  assumes alpha s t and alpha t u
  shows alpha s u
  using assms proof(induct s arbitrary: t u rule: measure-induct-rule[of depth])
  case (less s)
  show ?case
  proof(cases s)
    case (PVr x1)
    then show ?thesis using less.prems by fastforce
  next
    case (PAp s1 s2)
    then obtain t1 t2 where alpha s1 t1 alpha s2 t2 t = PAp t1 t2
      using less.prems by blast
    moreover then obtain u1 u2 where alpha t1 u1 alpha t2 u2 u = PAp u1 u2
      using less.prems by blast
    ultimately show ?thesis
       by (smt (verit, ccfv-threshold) add.right-neutral add-less-le-mono alpha.PAp
depth.simps(2)
         dual-order.strict-trans2 le-add-same-cancel2 less.hyps less-add-same-cancel1

          PAp neq0-conv zero-le zero-less-one)
```

```
next
  case (PLm x s′)
  obtain t′ z y where t: t = PLm y t′ z = x ∨ pfresh z s′
    z = y ∨ pfresh z t′ alpha (pswap s′ z x) (pswap t′ z y)
    using PLm less.prems by blast
  obtain u′ zz w where u: u = PLm w u′ zz = y ∨ pfresh zz t′
    zz = w ∨ pfresh zz u′ alpha (pswap t′ zz y) (pswap u′ zz w)
    using less.prems t by blast
  obtain zf where zf: zf ≠ x zf ≠ y zf ≠ z zf ≠ w zf ≠ zz
    pfresh zf s′ pfresh zf t′ pfresh zf u′
    using exists-pfresh[of [x,y,z,w,zz] [s′,t′,u′]] by auto

  {have alpha (pswap s′ zf x) (pswap (pswap s′ z x) z zf)
      by (smt (verit, del-insts) alpha-pswap alpha-sym pswap-pfresh-alpha
          pswap-pswap pswap-sym sw-diff sw-eqL t(2) zf(1) zf(6))
    moreover have alpha (pswap (pswap t′ z y) z zf) (pswap t′ zf y)
      by (smt (verit, ccfv-threshold) pswap-pswap alpha-pswap
          sw-diff sw-eqL pswap-pfresh-alpha pswap-sym t(3) zf(2) zf(7))
    ultimately have alpha (pswap s′ zf x) (pswap t′ zf y)
      by (metis alpha-pswap depth.simps(3) less.hyps less-add-same-cancel1
          less-numeral-extra(1) local.PLm pswap-same-depth t(4))
  }
  moreover
  {have alpha (pswap t′ zf y) (pswap (pswap t′ zz y) zz zf)
     by (smt (verit) pswap-pswap alpha-pswap alpha-sym sw-diff sw-eqL pswap-pfresh-alpha
pswap-sym
          u(2) zf(2) zf(7))
    moreover have alpha (pswap (pswap u′ zz w) zz zf) (pswap u′ zf w)
      by (smt (verit, ccfv-threshold) pswap-pswap alpha-pswap sw-diff sw-eqL
          pswap-pfresh-alpha pswap-sym u(3) zf(4) zf(8))
    ultimately have alpha (pswap t′ zf y) (pswap u′ zf w)
     by (smt (verit, best) alpha-same-depth alpha-pswap depth.simps(3) less.hyps
      less.prems less-add-same-cancel1 pswap-same-depth u(1) u(4) zero-less-one)
  }
  ultimately show ?thesis
    by (metis alpha.PLm depth.simps(3) less.hyps less-add-same-cancel1
        local.PLm pswap-same-depth u(1) zero-less-one zf(6) zf(8))
 qed
qed

lemma alpha-PLm-strong-elim:
 assumes alpha (PLm x t) (PLm x′ t′)
   and z = x ∨ pfresh z t and z = x′ ∨ pfresh z t′
 shows alpha (pswap t z x) (pswap t′ z x′)
proof−
 obtain zz where zz: zz = x ∨ pfresh zz t zz = x′ ∨ pfresh zz t′
   alpha (pswap t zz x) (pswap t′ zz x′)
   using alpha-PLm-cases[OF assms(1)] by (smt (verit) ptrm.inject(3))
 have sw1: alpha (pswap t z x) (pswap (pswap t zz x) zz z)
```

unfolding *pswap-pswap*[*of t zz x*]
by (*metis alpha-refl alpha-pswap assms*(*2*)
sw-diff sw-eqL sw-eqR pswap-pfresh-alpha pswap-id pswap-invol pswap-sym zz(*1*))
have *sw2*: *alpha* (*pswap* (*pswap t′ zz x′*) *zz z*) (*pswap t′ z x′*)
unfolding *pswap-pswap*[*of t′ zz x′*]
by (*metis alpha-refl alpha-pswap assms*(*3*) *sw-diff sw-eqL sw-eqR*
pswap-pfresh-alpha pswap-id pswap-invol pswap-sym zz(*2*))
show *?thesis*
by (*meson alpha-pswap alpha-trans sw1 sw2 zz*(*3*))
qed

lemma *pfresh-pswap-alpha*:
assumes $y = x \lor$ *pfresh y t* **and** $z = x \lor$ *pfresh z t*
shows *alpha* (*pswap* (*pswap t y x*) *z y*) (*pswap t z x*)
by (*smt* (*verit*) *assms pswap-pswap alpha-refl alpha-pswap sw-diff sw-eqR pswap-pfresh-alpha pswap-id pswap-invol2*)

lemma *pfresh-sw-pswap-pswap*:
assumes *sw y′ z1 z2* $\neq y$ **and** $y =$ *sw x z1 z2* $\lor$ *pfresh y* (*pswap t z1 z2*)
and $y' = x \lor$ *pfresh y′ t*
shows *pfresh* (*sw y′ z1 z2*) (*pswap* (*pswap t z1 z2*) *y* (*sw x z1 z2*))
using *assms pfresh-pswap-iff sw-diff sw-eqR sw-invol* **by** (*smt* (*verit*))

1.3 Terms via quotienting pre-terms

quotient-type *trm* = *ptrm* / *alpha*
unfolding *equivp-def fun-eq-iff* **using** *alpha-sym alpha-trans alpha-refl* **by** *blast*

lift-definition *Vr* :: *var* ⇒ *trm* **is** *PVr* **.**
lift-definition *Ap* :: *trm* ⇒ *trm* ⇒ *trm* **is** *PAp* **by** *auto*
lift-definition *Lm* :: *var* ⇒ *trm* ⇒ *trm* **is** *PLm* **by** *auto*
lift-definition *swap* :: *trm* ⇒ *var* ⇒ *var* ⇒ *trm* **is** *pswap*
using *alpha-pswap* **by** *auto*
lift-definition *fresh* :: *var* ⇒ *trm* ⇒ *bool* **is** *pfresh*
using *alpha-pfresh-iff* **by** *blast*
lift-definition *ddepth* :: *trm* ⇒ *nat* **is** *depth*
using *alpha-same-depth* **by** *blast*

lemma *abs-trm-rep-trm*[*simp*]: *abs-trm* (*rep-trm t*) = *t*
by (*meson Quotient3-abs-rep Quotient3-trm*)

lemma *alpha-rep-trm-abs-trm*[*simp,intro*!]: *alpha* (*rep-trm* (*abs-trm t*)) *t*
by (*simp add*: *Quotient3-trm rep-abs-rsp-left*)

lemma *pfresh-rep-trm-abs-trm*[*simp*]: *pfresh z* (*rep-trm* (*abs-trm t*)) ⟷ *pfresh z t*
using *fresh.abs-eq fresh.rep-eq* **by** *auto*

lemma *swap-id*[*simp*]:
swap (*swap t z x*) *z x* = *t*
by *transfer simp*

lemma *fresh-PVr*[*simp*]: *fresh x* (*Vr y*) ⟷ $x \neq y$
by (*simp add*: *Vr-def fresh.abs-eq*)

lemma *fresh-Ap*[*simp*]: *fresh z* (*Ap t1 t2*) ⟷ *fresh z t1* ∧ *fresh z t2*
by *transfer auto*

lemma *fresh-Lm*[*simp*]: *fresh z* (*Lm x t*) ⟷ (*z* = *x* ∨ *fresh z t*)
by *transfer auto*

lemma *Lm-swap-rename*:
assumes *z* = *x* ∨ *fresh z t*
shows *Lm z* (*swap t z x*) = *Lm x t*
using *assms* **apply** *transfer*
using *alpha.PLm* **by** *auto*

lemma *abs-trm-PVr*: *abs-trm* (*PVr x*) = *Vr x*
by (*simp add*: *Vr.abs-eq*)

lemma *abs-trm-PAp*: *abs-trm* (*PAp t1 t2*) = *Ap* (*abs-trm t1*) (*abs-trm t2*)
by (*simp add*: *Ap.abs-eq*)

lemma *abs-trm-PLm*: *abs-trm* (*PLm x t*) = *Lm x* (*abs-trm t*)
by (*simp add*: *Lm.abs-eq*)

lemma *abs-trm-pswap*: *abs-trm* (*pswap t z1 z2*) = *swap* (*abs-trm t*) *z1 z2*
by (*simp add*: *swap.abs-eq*)

lemma *swap-Vr*[*simp*]: *swap* (*Vr x*) *z1 z2* = *Vr* (*sw x z1 z2*)
by *transfer simp*

lemma *swap-Ap*[*simp*]: *swap* (*Ap t1 t2*) *z1 z2* = *Ap* (*swap t1 z1 z2*) (*swap t2 z1 z2*)
by *transfer simp*

lemma *swap-Lm*[*simp*]: *swap* (*Lm x t*) *z1 z2* = *Lm* (*sw x z1 z2*) (*swap t z1 z2*)
by *transfer simp*

lemma *Lm-sameVar-inj*[*simp*]: *Lm x t* = *Lm x t1* ⟷ *t* = *t1*
by *transfer* (*metis alpha.PLm alpha-PLm-strong-elim pswap-id*)

lemma *Lm-eq-swap*:
assumes *Lm x t* = *Lm x1 t1*
shows *t* = *swap t1 x x1*
proof(*cases x* = *x1*)

```
  case True
  thus ?thesis using assms Lm-swap-rename by fastforce
next
  case False
  thus ?thesis
    by (metis Lm-sameVar-inj Lm-swap-rename assms fresh-Lm)
qed

lemma alpha-rep-abs-trm: alpha (rep-trm (abs-trm t)) t
  by simp

lemma swap-fresh-eq: assumes x:fresh x t and y:fresh y t
  shows swap t x y = t
  using pswap-pfresh-alpha x y unfolding fresh.rep-eq
 by (metis (full-types) Quotient3-abs-rep Quotient3-trm swap.abs-eq trm.abs-eq-iff)

lemma bij-sw:bij (λ x. sw x z1 z2)
  unfolding sw-def bij-def inj-def surj-def by (smt (verit))

lemma sw-set: x ∈ X = ((sw x z1 z2) ∈ (λ x. sw x z1 z2) ‘ X)
  using bij-sw by blast

lemma ddepth-Vr[simp]: ddepth (Vr x) = 0
  by transfer simp

lemma ddepth-Ap[simp]: ddepth (Ap t1 t2) = Suc (ddepth t1 + ddepth t2)
  by transfer simp

lemma ddepth-Lm[simp]: ddepth (Lm x t) = Suc (ddepth t)
  by transfer simp

lemma trm-nchotomy:
  (∃ x. tt = Vr x) ∨ (∃ t1 t2. tt = Ap t1 t2) ∨ (∃ x t. tt = Lm x t)
  apply transfer using ptrm.nchotomy by (metis alpha-refl ptrm.exhaust)

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
  using trm-nchotomy by blast

lemma Vr-Ap-diff[simp]: Vr x ≠ Ap t1 t2  Ap t1 t2 ≠ Vr x
  by (metis Zero-not-Suc ddepth-Ap ddepth-Vr)+

lemma Vr-Lm-diff[simp]: Vr x ≠ Lm y t  Lm y t ≠ Vr x
  by (metis Zero-not-Suc ddepth-Lm ddepth-Vr)+

lemma Ap-Lm-diff[simp]: Ap t1 t2 ≠ Lm y t  Lm y t ≠ Ap t1 t2
  by (transfer,blast)+
```

lemma *Vr-inj*[*simp*]: (*Vr x* = *Vr y*) ⟷ *x* = *y*
  by *transfer auto*

lemma *Ap-inj*[*simp*]: (*Ap t1 t2* = *Ap t1′ t2′*) ⟷ *t1* = *t1′* ∧ *t2* = *t2′*
  by *transfer auto*

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*)
  unfolding *fresh.rep-eq* **using** *cofinite-pfresh* **by** *simp*

lemma *exists-fresh-set*:
  assumes *finite X*
  shows ∃ *z*. *z* ∉ *X* ∧ *z* ∉ *set xs* ∧ (∀ *t* ∈ *set ts*. *fresh z t*)
  using *assms* **apply** *transfer*
  using *exists-pfresh-set* **by** *presburger*

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*)
  using *exists-fresh-set*[*OF assms*] **unfolding** *ppickFreshS-def*
  by (*rule someI-ex*)

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

assume *fresh x t*
hence {*y*. *swap t y x* ≠ *t*} ⊆ {*y*. ¬ *fresh y t*}
by (*auto*, *meson swap-fresh-eq*)
thus *finite* {*y*. *swap t y x* ≠ *t*}
using *cofinite-fresh rev-finite-subset* **by** *blast*
next
assume *finite* {*y*. *swap t y x* ≠ *t*}
moreover have *finite* {*y*. ¬ *fresh y t*} **using** *cofinite-fresh* **.**
ultimately have *finite* {*y*. ¬ *fresh y t* ∨ *swap t y x* ≠ *t* ∨ *y* = *x*}
by (*metis* (*full-types*) *finite-Collect-disjI finite-insert insert-compr*)
then obtain *y* **where** *fresh y t* **and** *y* ≠ *x* **and** *swap t y x* = *t*
using *exists-var* **by** *auto*
thus *fresh x t* **by** (*metis Lm-swap-rename fresh-Lm*)
qed

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(*induct rule*: *measure-induct*[*of ddepth*])
case (*1 tt*)
show *?case* **using** *trm-nchotomy*[*of tt*] **apply** *safe*
subgoal using *Vr 1* **by** *auto*
subgoal using *Ap 1* **by** *auto*
subgoal by (*metis 1 Lm Lm-swap-rename ddepth-Lm fresh-Lm lessI*
old.nat.inject swap-Lm) **.**
qed

lemma *fresh-induct*[*consumes 1*, *case-names Vr Ap Lm*]:
assumes *finite X* **and** ⋀*x*. φ (*Vr x*)
and ⋀*t1 t2*. φ *t1* ⟹ φ *t2* ⟹ φ (*Ap t1 t2*)
and ⋀*x t*. φ *t* ⟹ *x* ∉ *X* ⟹ φ (*Lm x t*)
shows φ *t*
apply(*induct rule*: *swap-induct*)
using *assms*
by *auto* (*metis Collect-mem-eq Collect-mono Lm-swap-rename cofinite-fresh*
finite-Collect-not infinite-var rev-finite-subset)

lemma *plain-induct*[*case-names Vr Ap Lm*]:
assumes ⋀*x*. φ (*Vr x*)
and ⋀*t1 t2*. φ *t1* ⟹ φ *t2* ⟹ φ (*Ap t1 t2*)
and ⋀*x t*. φ *t* ⟹ φ (*Lm x t*)
shows φ *t*
by (*metis assms fresh-induct finite.emptyI*)

1.5 Substitution

inductive *substRel* :: *trm* ⇒ *trm* ⇒ *var* ⇒ *trm* ⇒ *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 $(x = y \wedge t = t') \vee (x \neq y \wedge t' = Vr\ x)$
using *assms* **by** (*cases rule*: *substRel.cases*) *auto*

lemma *substRel-Ap-invert*:
assumes *substRel* (*Ap t1 t2*) *s y t'*
shows ∃ *t1' t2'*. *t'* = *Ap t1' t2'* ∧ *substRel t1 s y t1'* ∧ *substRel t2 s y t2'*
using *assms* **by** (*cases rule*: *substRel.cases*) *auto*

lemma *substRel-Lm-invert-aux*:
assumes *substRel* (*Lm x t*) *s y tt'*
shows ∃ *x1 t1 t1'*.
x1 ≠ *y* ∧ *fresh x1 s* ∧
Lm x t = *Lm x1 t1* ∧ *tt'* = *Lm x1 t1'* ∧ *substRel t1 s y t1'*
using *assms* **by** (*cases rule*: *substRel.cases*) *auto*

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*)
using *assms* **apply** *induct*
by (*auto intro*: *substRel.intros*) (*simp add*: *fresh.rep-eq substRel-Lm swap-def*)

lemma *substRel-fresh*:
assumes *substRel t s y t'* **and** *fresh x1 t x1* ≠ *y fresh x1 s*
shows *fresh x1 t'*
using *assms* **by** *induct auto*

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'*
using *substRel-Lm-invert-aux*[*OF assms*(*1*)] **proof**(*elim exE conjE*)
fix *x1 t1 t1'*
assume *1*: *x1* ≠ *y fresh x1 s Lm x t* = *Lm x1 t1*
substRel t1 s y t1' tt' = *Lm x1 t1'*
have *2*: *t* = *swap t1 x x1* **by** (*simp add*: *1*(*3*) *Lm-eq-swap*)

hence *3*: $x = x1 \lor$ *fresh x t1*
by (*metis 1*(*3*) *fresh-Lm*)
have *4*: s = *swap s x x1 y* = *sw y x x1*
apply (*simp add*: *1*(*2*) *assms*(*3*) *swap-fresh-eq*)
using *1*(*1*) *assms*(*2*) *sw-def* **by** *presburger*
show *?thesis*
apply(*rule exI*[*of - swap t1′ x x1*], *safe*)
subgoal unfolding *1* **apply**(*rule sym*, *rule Lm-swap-rename*)
using *3 substRel-fresh*[*OF 1*(*4*) *- 0*] **by** *auto*
subgoal unfolding *2* **apply**(*subst 4*(*1*), *subst 4*(*2*))
using *substRel-swap*[*OF 1*(*4*)] **. .**
qed

lemma *substRel-total*:
∃ *t′*. *substRel t s y t′*
proof−
have *finite* ({*y*} ∪ *FFvars s*)
by (*simp add*: *cofinite-fresh*)
thus *?thesis* **apply**(*induct t rule*: *fresh-induct*)
subgoal by (*metis substRel-Vr-diff substRel-Vr-same*)
subgoal by(*auto intro*: *substRel-Ap*)
by(*auto intro*: *substRel-Lm*)
qed

lemma *substRel-functional*:
assumes *substRel t s y t′* **and** *substRel t s y tt′*
shows *t′* = *tt′*
proof−
have *finite* ({*y*} ∪ *FFvars s*)
by (*simp add*: *cofinite-fresh*)
thus *?thesis*
using *assms* **apply**(*induct t arbitrary*: *t′ tt′ rule*: *fresh-induct*)
subgoal using *substRel-Vr-invert* **by** *blast*
subgoal by (*metis substRel-Ap-invert*)
subgoal by (*metis CollectI UnI1 UnI2 singleton-iff substRel-Lm-invert*) **.**
qed

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*)
by (*simp add*: *someI-ex substRel-total subst-def*)

lemma *substRel-subst-unique*: *substRel t s x tt* ⟹ *tt* = *subst t s x*
by (*simp add*: *substRel-functional substRel-subst*)

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*)
subgoal by (*metis substRel-Vr-invert substRel-subst*)
subgoal by (*metis substRel-Ap substRel-subst substRel-subst-unique*)
subgoal by (*meson substRel-Lm substRel-functional substRel-subst*) **.**

lemma *fresh-subst*:
fresh z (*subst s t x*) ⟷ (*z* = *x* ∨ *fresh z s*) ∧ (*fresh x s* ∨ *fresh z t*)
proof−
have *finite* ({*x*,*z*} ∪ *FFvars t*)
by (*simp add*: *cofinite-fresh*)
thus *?thesis* **apply**(*induct s rule*: *fresh-induct*) **by** *auto*
qed

lemma *fresh-subst-id*[*simp*]:
assumes *fresh x s* **shows** *subst s t x* = *s*
proof−
have *finite* (*FFvars t* ∪ {*x*})
by (*simp add*: *cofinite-fresh*)
thus *?thesis* **using** *assms* **apply**(*induct s rule*: *fresh-induct*) **by** *auto*
qed

lemma *subst-Vr-id*[*simp*]: *subst s* (*Vr x*) *x* = *s*
proof−
have *finite* {*x*} **by** *auto*
thus *?thesis* **by** (*induct s rule*: *fresh-induct*) *auto*
qed

lemma *Lm-swap-cong*:
assumes *z* = *x* ∨ *fresh z s z* = *y* ∨ *fresh z t* **and** *swap s z x* = *swap t z y*
shows *Lm x s* = *Lm y t*
using *assms* **by** (*metis Lm-swap-rename*)

lemma *fresh-swap*[*simp*]: *fresh x* (*swap t z1 z2*) ⟷ *fresh* (*sw x z1 z2*) *t*
apply(*induct t rule*: *plain-induct*) **by** *auto*

lemma *swap-subst*:
swap (*subst s t x*) *z1 z2* = *subst* (*swap s z1 z2*) (*swap t z1 z2*) (*sw x z1 z2*)
proof−
have *finite* (*FFvars t* ∪ {*x*,*z1*,*z2*})
by (*simp add*: *cofinite-fresh*)
thus *?thesis* **apply**(*induct s rule*: *fresh-induct*)
using *fresh-swap subst-Lm sw-def* **by** *auto*
qed

lemma *subst-Lm-same*[*simp*]: *subst* (*Lm x s*) *t x* = *Lm x s*

by *simp*

lemma *fresh-subst-same*:
assumes $y \neq z$ **shows** *fresh y* (*subst t* (*Vr z*) *y*)
proof−
have *finite* ({*y*,*z*})
by (*simp add*: *cofinite-fresh*)
thus *?thesis* **using** *assms* **apply**(*induct t rule*: *fresh-induct*) **by** *auto*
qed

lemma *subst-comp-same*:
subst (*subst s t x*) *t1 x* = *subst s* (*subst t t1 x*) *x*
proof−
have *finite* ({*x*} ∪ *FFvars t* ∪ *FFvars t1*)
by (*simp add*: *cofinite-fresh*)
thus *?thesis* **apply**(*induct s rule*: *fresh-induct*)
using *fresh-subst subst-Lm* **by** *auto*
qed

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−
have *finite* ({*x*,*x1*} ∪ *FFvars t* ∪ *FFvars t1*)
by (*simp add*: *cofinite-fresh*)
thus *?thesis* **using** *assms* **apply**(*induct s rule*: *fresh-induct*)
using *fresh-subst subst-Lm* **by** *auto*
qed

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*
apply(*rule subst-comp-diff*)
using *assms* **by** *auto*

lemma *subst-chain*:
assumes *fresh u s*
shows *subst* (*subst s* (*Vr u*) *x*) *t u* = *subst s t x*
proof−
have *finite* ({*x*,*u*} ∪ *FFvars t* ∪ *FFvars s*)
by (*simp add*: *cofinite-fresh*)
thus *?thesis* **using** *assms* **apply**(*induct s rule*: *fresh-induct*)
by *auto*
qed

lemma *subst-repeated-Vr*:
subst (*subst t* (*Vr x*) *y*) (*Vr u*) *x* =

```
 subst (subst t (Vr u) x) (Vr u) y
proof−
  have finite ({x,y,u} ∪ FFvars t)
    by (simp add: cofinite-fresh)
  thus ?thesis apply(induct t rule: fresh-induct)
    using fresh-subst subst-Lm by auto
qed

lemma subst-commute-same:
  subst (subst d (Vr u) x) (Vr u) y = subst (subst d (Vr u) y) (Vr u) x
  by (metis subst-Vr-id subst-repeated-Vr)

lemma subst-commute-diff:
  assumes x ≠ v y ≠ u x ≠ y
  shows subst (subst t (Vr u) x) (Vr v) y = subst (subst t (Vr v) y) (Vr u) x
proof−
  have finite ({u,v,x,y})
    by (simp add: cofinite-fresh)
  thus ?thesis using assms apply(induct t rule: fresh-induct) by auto
qed

lemma subst-same-id:
  assumes z1 ≠ y
  shows subst (subst t (Vr z1) y) (Vr z2) y = subst t (Vr z1) y
  using assms subst-Vr subst-comp-same by presburger

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−
  have finite ({z1,z2,yy} ∪ FFvars t)
    by (simp add: cofinite-fresh)
  thus ?thesis using assms apply(induct t rule: fresh-induct) by auto
qed

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−
  have ?L = swap tt z1 z2
    apply(rule sym, rule swap-from-subst)
    using assms fresh-PVr fresh-subst by auto
  also have ... = ?R apply(rule swap-from-subst)
    using assms fresh-PVr fresh-subst by auto
  finally show ?thesis .
```

qed

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−
  have *?L* = *subst* (*swap* (*subst t* (*Vr xx*) *x*) *z1 z2*) (*Vr vv*) *xx*
    by (*metis assms*(*1*,*2*) *fresh-PVr fresh-subst insertCI swap-from-subst*)
  also have ... = *?R*
    using *assms*(*1*,*3*) *subst-chain sw-diff swap-from-subst swap-subst* **by** *auto*
  finally show *?thesis* **.**
qed

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*)
  by (*metis assms fresh-PVr fresh-subst insertCI subst-two-ways''*)

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*
  by (*metis fresh-Ap fresh-Lm fresh-PVr trm-nchotomy*)

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−
have *finite* ({*z*,*x*})
by *simp*
thus *?thesis* **using** *assms* **by** (*induct t rule*: *fresh-induct*) *auto*
qed

lemma *Lm-subst-rename*:
assumes *z* = *x* ∨ *fresh z t*
shows *Lm z* (*subst t* (*Vr z*) *x*) = *Lm x t*
using *Lm-swap-rename assms fresh-subst-eq-swap subst-Vr-id* **by** *presburger*

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*
by (*metis Lm-subst-rename*)

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*
by (*simp add*: *Lm-eq-swap Lm-swap-rename*)

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*
by (*smt* (*verit*, *ccfv-threshold*) *Lm-eq-elim Lm-subst-rename swap-id*)

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′* ⟹ *substConnect t* (*vsubst t′ z x*)

lemma *ddepth-swap*:
ddepth (*swap t z x*) = *ddepth t*
by (*metis ddepth.abs-eq ddepth.rep-eq map-fun-apply swap-def pswap-same-depth*)

lemma *ddepth-subst-Vr*[*simp*]:
ddepth (*vsubst t z x*) = *ddepth t*
proof−
have *finite* ({*z*,*x*})
by *simp*
thus *?thesis* **by** (*induct t rule*: *fresh-induct*) *auto*
qed

lemma *substConnect-depth*:
  assumes *substConnect t t′* **shows** *ddepth t = ddepth t′*
  using *assms* **by** (*induct*, *auto*)

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(*induct rule*: *measure-induct*[*of ddepth*])
  case (*1 tt*)
  show *?case* **using** *trm-nchotomy*[*of tt*]
    using *1 Ap Lm Vr substConnect-depth* **by** *auto*
qed

1.7 Syntactic environments

typedef *fenv* = $\{f :: var \Rightarrow trm\ .\ finite\ \{x.\ f\ x \neq Vr\ x\}\}$
  using *not-finite-existsD* **by** *auto*

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 ≡ $\{x.\ get\ f\ x \neq Vr\ x\}$

lemma *finite-supp*: *finite* (*supp f*)
  using *Rep-fenv get-def supp-def* **by** *auto*

lemma *finite-upd*:
  assumes *finite* $\{x.\ f\ x \neq Vr\ x\}$
  shows *finite* $\{x.\ (f(y{:=}t))\ x \neq Vr\ x\}$
proof−
  have $\{x.\ (f(y{:=}t))\ x \neq Vr\ x\} \subseteq \{x.\ f\ x \neq Vr\ x\} \cup \{y\}$
    by *auto*
  thus *?thesis*
   by (*metis* (*full-types*) *assms finite-insert insert-is-Un rev-finite-subset sup.commute*)
qed

lemma *get-upd-same*[*simp*]: *get* (*upd f x t*) *x* = *t*
  and *get-upd-diff*[*simp*]: $x \neq y \Longrightarrow$ *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*: $x \neq y \Longrightarrow$ *upd* (*upd f x t*) *y s* = *upd* (*upd f y s*) *x t*
  and *supp-get*[*simp*]: $x \notin supp\ \varrho \Longrightarrow get\ \varrho\ x = Vr\ x$
  unfolding *get-def upd-def* **using** *Rep-fenv finite-upd*
  by (*auto simp*: *fun-upd-twist Abs-fenv-inverse get-def supp-def*)

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−
  obtain *yy* **where** *yy*: *vsubstA a yy x* = *a yy* ≠ *x*
    using *n* **unfolding** *freshA-def* **using** *exists-var* **by** *force*
  hence *vsubstA a y x* = *vsubstA* (*vsubstA a yy x*) *y x* **by** *simp*
  also have ... = *vsubstA a yy x* **by** (*simp add*: *yy*(*2*))
  also have ... = *a* **using** *yy*(*1*) **.**
  finally show *?thesis* **.**
qed

lemma *vsubstA-chain-freshA*:

assumes *freshA x2 a*
shows *vsubstA (vsubstA a x2 x1) x3 x2 = vsubstA a x3 x1*
proof−
obtain *yy* **where** *yy*: *yy ≠ x2*
by (*metis*(*full-types*) *list.set-intros*(*1*) *pickFresh-var*)
have *0*: *a = vsubstA a yy x2*
using *assms freshA-vsubstA-idle yy* **by** *presburger*
show *?thesis*
by (*metis 0 vsubstA-chain yy*)
qed

lemma *freshA-vsubstA*:
assumes *freshA u a* **and** *u ≠ y*
shows *freshA u* (*vsubstA a y x*)
proof−
have {*ya. vsubstA* (*vsubstA a y x*) *ya u ≠ vsubstA a y x*} ⊆ {*y. vsubstA a y u ≠ a*} ∪ {*x*,*y*,*u*} ∪ {*y.* ¬ *freshA y a*}
using *assms* **by** *auto* (*metis vsubstA-commute-diff vsubstA-idem*)
show *?thesis* **using** *assms* **unfolding** *freshA-def*
by (*smt* (*verit*, *best*) *Collect-mono-iff finite-subset vsubstA-commute-diff vsubstA-id vsubstA-idem*)
qed

lemma *freshA-vsubstA2*:
assumes *freshA z a ∨ z = x* **and** *freshA x a ∨ z ≠ y*
shows *freshA z* (*vsubstA a y x*)
proof(*cases z = y*)
case *True*
thus *?thesis* **using** *assms* **by** (*metis freshA-vsubstA-idle vsubstA-id*)
next
case *False*
hence {*ya. vsubstA* (*vsubstA a y x*) *ya z ≠ vsubstA a y x*} ⊆
{*ya. vsubstA a ya z ≠ a*} ∪ {*ya. vsubstA a ya y ≠ a*} ∪ {*x*}
by *auto* (*metis vsubstA-commute-diff vsubstA-idem*)
thus *?thesis*
using *assms* **unfolding** *freshA-def*
by (*smt* (*verit*, *best*) *False assms*(*1*) *freshA-vsubstA freshA-vsubstA-idle not-finite-existsD vsubstA-idem*)
qed

lemma *vsubstA-idle-freshA*:
assumes *vsubstA a y x = a* **and** *xy*: *x ≠ y*
shows *freshA x a*
by (*smt* (*verit*, *best*) *assms*(*1*) *freshA-def not-finite-existsD vsubstA-idem xy*)

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

by (*smt* (*verit*) *CollectI exists-var finite.insertI insertCI freshA-def vsubstA-idle-freshA*)

lemma *freshA-iff-all-vvsubstA-idle*:
freshA x a ⟷ (∀ *y*. *y*≠*x* ⟶ *vsubstA a y x* = *a*)
by (*metis list.set-intros*(*1*) *freshA-vsubstA-idle pickFresh-var vsubstA-idle-freshA*)

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−
have *1*: {*x*. ∃ *a*∈*set ds*. ¬ *freshA x a*} = ⋃ {{*x*. ¬ *freshA x a*} | *a*. *a*∈*set ds*} **by** *auto*
have *finite* {*x*. ∃ *a*∈*set ds*. ¬ *freshA x a*}
unfolding *1* **apply**(*rule finite-Union*)
using *assms cofinite-freshA* **by** *auto*
hence *0*: *finite* (*X* ∪ *set xs* ∪ {*x*. ∃ *a*∈*set ds*. ¬ *freshA x a*})
using *assms* **by** *blast*
show *?thesis* **using** *exists-var*[*OF 0*] **by** *simp*
qed

lemma *exists-freshA*:
∃ *z*. *z* ∉ *set xs* ∧ (∀ *a* ∈ *set ds*. *freshA z a*)
using *exists-freshA-set* **by** *blast*

lemma *pickFreshSA*:
assumes *finite X*
shows
pickFreshSA X xs ds ∉ *X* ∧
pickFreshSA X xs ds ∉ *set xs* ∧
(∀ *a* ∈ *set ds*. *freshA* (*pickFreshSA X xs ds*) *a*)
using *exists-freshA-set*[*OF assms*] **unfolding** *pickFreshSA-def*
by (*rule someI-ex*)

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* ⇒ *var* ⇒ *var* ⇒ *'A*
+
assumes *cofinite-freshA*: ⋀*a*. *finite* {*x*. ¬ *freshA x a*}

locale *Nominal-Morphism* =
A: *Nominal-Set swapA* + *B*: *Nominal-Set swapB*
for *swapA* :: *'A* ⇒ *var* ⇒ *var* ⇒ *'A* **and** *swapB* :: *'B* ⇒ *var* ⇒ *var* ⇒ *'B*
+
fixes *f* :: *'A* ⇒ *'B*
assumes *f-swapA-swapB*: ⋀*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* ⇒ *var* ⇒ *var* ⇒ *'A* **where**
swapA a z1 z2 ≡
let yy = *pickFreshA* [*z1*,*z2*] [*a*] *in*
vsubstA (*vsubstA* (*vsubstA a yy z1*) *z1 z2*) *z2 yy*

lemma *swapA*:
∃ *yy*. *yy*∉{*z1*,*z2*} ∧ *freshA yy a* ∧
swapA a z1 z2 = *vsubstA* (*vsubstA* (*vsubstA a yy z1*) *z1 z2*) *z2 yy*
proof−
define *yy* **where** *yy*: *yy* = *pickFreshA* [*z1*, *z2*] [*a*]
have *swapA a z1 z2* = *vsubstA* (*vsubstA* (*vsubstA a yy z1*) *z1 z2*) *z2 yy*
unfolding *swapA-def yy* **by** (*simp add*: *Let-def*)
moreover have *yy*∉{*z1*,*z2*} ∧ *freshA yy a* **using** *pickFreshA*[*of* [*z1*,*z2*] [*a*]]
unfolding *yy* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *swapA-id*[*simp*]:
  swapA a z z = *a*
  using *swapA*[*of z z*] **by** (*metis vsubstA-chain-freshA vsubstA-id*)

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*
  by (*smt* (*verit*, *best*) *vsubstA-id vsubstA-idem vsubstA-chain vsubstA-commute-diff*

    assms vsubstA-chain-freshA)

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*
  by (*metis assms insertCI swapA vvsubstA-twoWays*)

lemma *swapA-invol*[*simp*]: *swapA* (*swapA a x y*) *x y* = *a*
proof(*cases x* = *y*)
  case *True*
  thus *?thesis* **by** *simp*
next
  case *False*
  obtain *yy* **where** *yy*: *yy* ≠ *x* ∧ *yy* ≠ *y* ∧ *freshA yy a*
    and *0*: *swapA a x y* = *vsubstA* (*vsubstA* (*vsubstA a yy x*) *x y*) *y yy*
    using *swapA*[*of x y*] **by** *auto*
  define *dd* **where** *dd* ≡ *vsubstA* (*vsubstA* (*vsubstA a yy x*) *x y*) *y yy*

  have *finite* ({*vv*. ¬ *freshA vv a*} ∪ {*yy*,*x*,*y*})
    by (*simp add*: *cofinite-freshA*)
  then obtain *vv* **where** *vv*: *vv* ≠ *yy* ∧ *vv* ≠ *x* ∧ *vv* ≠ *y* ∧ *freshA vv a*
    by (*metis* (*no-types*, *lifting*) *UnCI exists-var insertCI mem-Collect-eq*)

  have *11*: *swapA dd x y* = *vsubstA* (*vsubstA* (*vsubstA dd vv x*) *x y*) *y vv*
    using *swapA-any dd-def freshA-vsubstA vv* **by** *auto*

  show *?thesis* **using** *yy vv* **unfolding** *0 11*[*unfolded dd-def*]
    using *vsubstA-commute-diff*
    by (*smt* (*verit*) *freshA-vsubstA2 vsubstA-chain-freshA vsubstA-id*)
qed

lemma *swapA-cmp*:
  swapA (*swapA a x y*) *z1 z2* = *swapA* (*swapA a z1 z2*) (*sw x z1 z2*) (*sw y z1 z2*)
proof(*cases z1*=*z2* ∨ *x* = *y*)
  case *True*
  thus *?thesis* **by** *auto*

next
  case *False*
  have *finite* ({*uu*. ¬ *freshA uu a*} ∪ {*z1*,*z2*,*x*,*y*})
    by (*simp add*: *cofinite-freshA*)
  then obtain *uu* **where** *uu*: *uu* ≠ *z1* ∧ *uu* ≠ *z2* ∧ *uu* ≠ *x* ∧ *uu* ≠ *y* ∧ *freshA uu a*
    by (*metis* (*no-types*, *lifting*) *UnCI exists-var insertCI mem-Collect-eq*)

  have *finite* ({*uu′*. ¬ *freshA uu′ a*} ∪ {*z1*,*z2*,*x*,*y*,*uu*})
    by (*simp add*: *cofinite-freshA*)
  then obtain *uu′* **where** *uu′*: *uu′* ≠ *z1* ∧ *uu′* ≠ *z2* ∧ *uu′* ≠ *x* ∧ *uu′* ≠ *y* ∧ *uu′* ≠ *uu* ∧ *freshA uu′ a*
    by (*smt* (*verit*, *ccfv-threshold*) *UnCI exists-var insertCI mem-Collect-eq*)

  show *?thesis* **apply**(*subst swapA-any*[*of uu*])
    subgoal using *uu* **by** *auto*
    subgoal apply(*subst swapA-any*[*of uu′*])
      subgoal using *uu′* **by** (*simp add*: *freshA-vsubstA2*)
      subgoal apply(*subst swapA-any*[*of uu*])
        subgoal using *uu* **by** (*simp add*: *freshA-vsubstA2*)
        subgoal apply(*subst swapA-any*[*of uu′*])
          subgoal using *uu′* **by** (*simp add*: *freshA-vsubstA2 sw-def*)
          subgoal apply(*cases x* = *z1*, *simp-all*)
            subgoal apply(*cases y* = *z1*, *simp-all*)
              subgoal
                by (*smt* (*verit*, *ccfv-threshold*) *vsubstA-id vsubstA-idem*
                    vsubstA-chain vsubstA-commute-diff swapA-any uu uu′)
              subgoal apply(*cases y* = *z2*, *simp-all*)
                  subgoal by (*smt* (*verit*, *ccfv-threshold*) *uu uu′ vsubstA-chain vsubstA-chain-freshA vsubstA-commute-diff*)
                subgoal by (*smt* (*verit*, *ccfv-threshold*) *freshA-vsubstA2 uu uu′ vsubstA-chain-freshA vsubstA-commute-diff*) **. .**
            subgoal apply(*cases x* = *z2*, *simp-all*)
              subgoal apply(*cases y* = *z1*, *simp-all*)
                  subgoal by (*smt* (*verit*, *ccfv-threshold*) *uu uu′ vsubstA-chain vsubstA-chain-freshA vsubstA-commute-diff*)
                subgoal apply(*cases y* = *z2*, *simp-all*)
                  subgoal by (*smt* (*verit*, *ccfv-threshold*) *uu uu′ vsubstA-chain vsubstA-chain-freshA vsubstA-commute-diff*)
                  subgoal by (*smt* (*verit*, *ccfv-threshold*) *uu uu′ vsubstA-chain vsubstA-chain-freshA vsubstA-commute-diff*) **. .**
              subgoal apply(*cases y* = *z1*, *simp-all*)
                subgoal by (*smt* (*verit*, *ccfv-threshold*) *freshA-vsubstA2 uu uu′ vsubstA-chain-freshA vsubstA-commute-diff*)
                subgoal apply(*cases y* = *z2*, *simp-all*)
                  subgoal by (*smt* (*verit*, *ccfv-threshold*) *uu uu′ vsubstA-chain vsubstA-chain-freshA vsubstA-commute-diff*)
                  subgoal

```
          by (smt (verit, ccfv-threshold) freshA-vsubstA2 swapA-any uu uu'
vsubstA-commute-diff) ........
qed



lemma freshA-swapA-vsubstA:
  assumes freshA y a
  shows swapA a y x = vsubstA a y x
proof−
  have finite ({uu. ¬ freshA uu a} ∪ {x,y})
    by (simp add: cofinite-freshA)
  then obtain uu where uu: uu ≠ x ∧ uu ≠ y ∧ freshA uu a
    by (metis (no-types, lifting) UnCI exists-var insertCI mem-Collect-eq)
  show ?thesis apply(subst swapA-any[of uu])
    subgoal using uu by simp
    subgoal using assms freshA-vsubstA2 freshA-vsubstA-idle uu by force .
qed

end

sublocale Renset-FinSupp < Sw: Pre-Nominal-Set where swapA = swapA
  using Pre-Nominal-Set-def swapA-cmp swapA-id swapA-invol by blast

context Renset-FinSupp
begin


lemma freshA-swapA: freshA x a ⟷ Sw.freshA x a
proof−
  have 0: {y. swapA a y x ≠ a} ⊆ {y. vsubstA a y x ≠ a} ∪ {y. ¬ freshA y a}
    using freshA-swapA-vsubstA by auto
  have
    1: {y. vsubstA a y x ≠ a} ⊆ {y. swapA a y x ≠ a} ∪ {y. ¬ freshA y a}
    using freshA-swapA-vsubstA by auto
  show ?thesis unfolding freshA-def using cofinite-freshA
    unfolding Sw.freshA-def by (metis 0 1 finite-Un rev-finite-subset)
qed

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
  apply standard unfolding freshA-swapA[symmetric]
  by (simp add: cofinite-freshA)
```

... the morphism component:

sublocale *Renset-Morphism* < *F*: *Nominal-Morphism* **where**
swapA = *A.swapA* **and** *swapB* = *B.swapA* **and** *f* = *f*
apply *standard*
by (*metis* (*no-types*, *opaque-lifting*) *A.Renset-axioms A.swapA*
B.Renset-FinSupp-axioms B.Renset-axioms
Renset.freshA-iff-ex-vvsubstA-idle
Renset-FinSupp.swapA-any f-substA-substB insertCI)

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* :: $'A \Rightarrow var \Rightarrow var \Rightarrow 'A$
+
fixes
X :: *var set*

and *VrA* :: $var \Rightarrow 'A$
and *ApA* :: $trm \Rightarrow 'A \Rightarrow trm \Rightarrow 'A \Rightarrow 'A$
and *LmA* :: $var \Rightarrow trm \Rightarrow 'A \Rightarrow 'A$
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: $\bigwedge y\ z\ t1\ a1\ t2\ a2.\ \{y,z\} \cap X = \{\} \Longrightarrow$
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: $\bigwedge$ *t a z x y*. $\{x,y,z\} \cap X = \{\} \Longrightarrow$
$x \neq y \Longrightarrow$
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*)
using *LmA-rename* **using** *Int-commute disjoint-insert*(*2*) **by** *metis*

lemma *vsubstA-LmA-same*:
$\{x,y\} \cap X = \{\} \Longrightarrow$ *vsubstA* (*LmA x t a*) *y x* = *LmA x t a*
by (*metis insert-disjoint*(*1*) *vsubstA-LmA vsubstA-id*)

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*)
using *vsubstA-LmA* **by** *meson*

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'
using *assms* **unfolding** *freshA-def*
by (*metis assms exists-var finite.insertI finite-X insertCI freshA-vsubstA-idle*)

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−
obtain *u* **where** *1*: $u \notin X \cup \{z\}$
by (*metis Un-insert-right boolean-algebra-cancel.sup0 exists-var finite-X finite-insert*)
hence *0*: *t* = *vsubst t u z* *a* = *vsubstA a u z*
t' = *vsubst t' u z* *a'* = *vsubstA a' u z*

using *assms freshA-iff-all-vvsubstA-idle* **by** *auto*
show *?thesis* **apply**(*subst 0*(*1*), *subst 0*(*2*), *subst 0*(*3*), *subst 0*(*4*))
apply(*rule LmA-cong*) **using** *assms 1 0* **by** *auto*
qed

lemma *freshA-VrA*: $z \notin X \Longrightarrow z \neq x \Longrightarrow$ *freshA z* (*VrA x*)
using *freshA-def vsubstA-VrA*
by (*metis Int-commute Int-insert-right-if0*
freshA-iff-ex-vvsubstA-idle exists-fresh-set finite-X inf-bot-right insertI1 list.set(*2*))

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*)
using *freshA-2-vsubstA*[*of z a1 a2*] *freshA-vsubstA2 vsubstA-ApA*
by (*metis Diff-disjoint Diff-insert-absorb Int-insert-left-if0 fresh-subst-id*)

lemma *freshA-LmA-same*:
assumes $x \notin X$
shows *freshA x* (*LmA x t a*)
proof−
have {*y. vsubstA* (*LmA x t a*) *y x* ≠ *LmA x t a*} ⊆ *X*
using *assms vsubstA-LmA-same*[*of x - t a*] **by** *blast*
thus *?thesis* **unfolding** *freshA-def finite-X*
by (*meson finite-X rev-finite-subset*)
qed

lemma *freshA-LmA′*:
assumes {*x*,*z*} ∩ *X* = {} *fresh z t* *freshA z a*
shows *freshA z* (*LmA x t a*)
proof(*cases x* = *z*)
case *True*
thus *?thesis*
using *assms*(*1*) *freshA-LmA-same* **by** *auto*
next
case *False*
hence {*y. vsubstA* (*LmA x t a*) *y z* ≠ *LmA x t a*} ⊆
{*y. vsubst t y z* ≠ *t*} ∪ {*y. vsubstA a y z* ≠ *a*} ∪ {*x*} ∪ *X*
using *assms*(*1*) *vsubstA-LmA* **by** *force*
hence *finite* {*y. vsubstA* (*LmA x t a*) *y z* ≠ *LmA x t a*}
by (*smt* (*verit*, *best*) *Collect-empty-eq assms*(*2*) *assms*(*3*)
fresh-subst-id finite.simps finite-UnI finite-X freshA-2-vsubstA rev-finite-subset
vsubstA-idem)
thus *?thesis* **unfolding** *freshA-def* **by** *auto*
qed

lemma *LmA-rename-freshA*:
assumes {*x*,*z*} ∩ *X* = {} *z* ≠ *x fresh z t* *freshA z a*
shows *LmA x t a* = *LmA z* (*vsubst t z x*) (*vsubstA a z x*)

using *assms*
by *simp* (*smt* (*verit*, *ccfv-SIG*) *Int-insert-left LmA-rename assms*(*1*)
freshA-iff-ex-vvsubstA-idle subst-Vr-id subst-chain
vsubstA-chain vsubstA-id)

lemma *freshA-LmA*:
$\{x,z\} \cap X = \{\} \Longrightarrow z = x \vee (fresh\ z\ t \wedge freshA\ z\ a) \Longrightarrow freshA\ z\ (LmA\ x\ t\ a)$
using *freshA-LmA′ freshA-LmA-same* **by** (*meson insert-disjoint*(*1*))

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*]: $R\ (Vr\ x)\ a \longleftrightarrow a = VrA\ x$
apply *safe*
subgoal using *R.cases* **by** *fastforce*
subgoal by (*auto intro*: *R.intros*) **.**

lemma *F-Ap-elim*:
assumes $R\ (Ap\ t1\ t2)\ a$
shows $\exists\, a1\ a2.\ R\ t1\ a1 \wedge R\ t2\ a2 \wedge a = ApA\ t1\ a1\ t2\ a2$
by (*metis Ap-Lm-diff*(*1*) *Ap-inj R.cases Vr-Ap-diff*(*1*) *assms*)

lemma *F-Lm-elim*:
assumes $R\ (Lm\ x\ t)\ a$
shows $\exists\, x'\ t'\ e.\ R\ t'\ e \wedge x' \notin X \wedge Lm\ x\ t = Lm\ x'\ t' \wedge a = LmA\ x'\ t'\ e$
using *assms* **by** (*cases rule*: *R.cases*) *auto*

lemma *F-total*: $\exists\, a.\ R\ t\ a$
using *finite-X* **apply**(*induct rule*: *fresh-induct*) **by** (*auto intro*: *R.intros*)

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*:
$(\forall\, a\ a'.\ R\ t\ a \longrightarrow R\ t\ a' \longrightarrow a = a') \wedge$

```
 (∀ a x. x ∉ X ∧ fresh x t ∧ R t a ⟶ freshA x a) ∧
 (∀ a x y. x ∉ X ∧ y ∉ X ⟶ R t a ⟶ R (vsubst t y x) (vsubstA a y x))
proof(induct t rule: substConnect-induct)
  case (Vr x)
  then show ?case by (auto simp: freshA-VrA vsubstA-VrA)
next
  case (Ap t1 t2)
  then show ?case apply safe
      apply (metis F-Ap-elim)
     apply (metis F-Ap-elim freshA-ApA fresh-Ap)
     by (smt (verit, ccfv-SIG) Diff-disjoint Diff-insert-absorb R.simps F-Ap-elim
Int-commute
        Int-insert-right-if0 subst-Ap vsubstA-ApA)
next
  case (Lm x t)
  show ?case
  proof safe
    fix a1 a2 assume R (Lm x t) a1 R (Lm x t) a2
    then obtain x1' t1' a1' x2' t2' a2'
      where 1: x1' ∉ X R t1' a1' Lm x t = Lm x1' t1' a1 = LmA x1' t1' a1'
        and   2: x2' ∉ X  R t2' a2' Lm x t = Lm x2' t2' a2 = LmA x2' t2' a2'
      using F-Lm-elim by metis

    define z where z = ppickFreshS X [x,x1',x2'] [t,t1',t2']
    have z: z ∉ {x,x1',x2'} fresh z t fresh z t1' fresh z t2' z ∉ X
      unfolding z-def using ppickFreshS[of X [x,x1',x2'] [t,t1',t2']] by auto

    have 11: vsubst t z x = vsubst t1' z x1'
      using 1(3) Lm-eq-elim-subst z by blast
    hence tt1': substConnect t t1'
     by (metis substConnect.simps subst-Vr-id subst-chain z(3))
    have 22: subst t (Vr z) x = subst t2' (Vr z) x2'
      using 2(3) Lm-eq-elim-subst z by blast
    hence tt2': substConnect t t2'
      by (metis Refl Step subst-Vr-id subst-chain z(4))

    show a1 = a2 unfolding 1 2 apply(rule LmA-cong-freshA[of z])
      subgoal using z by (simp add: 1(1) 2(1))
      subgoal using z(1) by force
      subgoal by (simp add: z)
      subgoal apply(rule Lm[rule-format, OF tt1',
            THEN conjunct2, THEN conjunct1, rule-format])
        using 1 z by auto
      subgoal using z by simp
      subgoal by (simp add: z)
      subgoal apply(rule Lm[rule-format, OF tt2',
            THEN conjunct2, THEN conjunct1, rule-format])
        using 2 z by auto
      subgoal using 11 22 by presburger
```

subgoal by (*metis 1*(*1*) *1*(*2*) *11 2*(*1*) *2*(*2*) *22 Lm.hyps Step tt1′ tt2′ z*(*5*)) **.**

next
fix *a y* **assume** *yX*: $y \notin X$
and *fr*: *fresh y* (*Lm x t*) *R* (*Lm x t*) *a*
then obtain x' t' a'
where *0*: $x' \notin X$ *R* t' a' *Lm x t* = *Lm* x' t' *a* = *LmA* x' t' a'
using *F-Lm-elim* **by** *metis*

define *z* **where** *z* = *ppickFreshS X* [*x*,*x*′] [*t*,*t*′]
have *z*: $z \notin \{x,x'\}$ *fresh z t fresh z* t' $z \notin X$
unfolding *z-def* **using** *ppickFreshS*[*of X* [*x*,*x*′] [*t*,*t*′]] **by** *auto*

have *00*: *subst t* (*Vr z*) *x* = *subst* t' (*Vr z*) x'
using *0*(*3*) *Lm-eq-elim-subst z* **by** *blast*
hence *tt1′*: *substConnect t* t'
by (*metis substConnect.simps subst-Vr-id subst-chain z*(*3*))

show *freshA y a* **unfolding** *0*
apply(*rule freshA-LmA*)
apply (*simp add*: *0*(*1*) *yX*)
apply(*subst disj-commute*, *safe*)
apply (*metis 0*(*3*) *fresh-Lm fr*(*1*))
apply(*rule Lm*[*rule-format*,*THEN conjunct2*, *THEN conjunct1*, *rule-format*, *of t*′])
subgoal using *tt1′* **.**
subgoal apply *safe*
subgoal using *yX* **by** *blast*
subgoal using *fr*(*1*) **unfolding** *0* **by** *simp*
subgoal using *0*(*2*) **. . .**

next
fix *a yy y* **assume** *yy-y*: $yy \notin X$ $y \notin X$ **and** *R* (*Lm x t*) *a*
then obtain x' t' a'
where *0*: $x' \notin X$ *R* t' a' *Lm x t* = *Lm* x' t' *a* = *LmA* x' t' a'
using *F-Lm-elim* **by** *metis*

define *z* **where** *z* = *ppickFreshS X* [*x*,*x*′,*y*,*yy*] [*t*,*t*′]
have *z*: $z \notin \{x,x',y,yy\}$ *fresh z t fresh z* t' $z \notin X$
unfolding *z-def* **using** *ppickFreshS*[*of X* [*x*,*x*′,*y*,*yy*] [*t*,*t*′]] **by** *auto*

have *00*: *subst t* (*Vr z*) *x* = *subst* t' (*Vr z*) x'
using *0*(*3*) *Lm-eq-elim-subst z* **by** *blast*
hence *tt1′*: *substConnect t* t'
by (*metis substConnect.simps subst-Vr-id subst-chain z*(*3*))

define t'' **where** $t'' \equiv$ *subst* t' (*Vr z*) x'

```
    have 1: Lm x′ t′ = Lm z t′′ unfolding t′′-def
     by (simp add: Lm-subst-rename z(3))

    have tt′′: substConnect t t′′
     using Step t′′-def tt1′ by blast

    define a′′ where a′′ ≡ vsubstA a′ z x′

    have 11: LmA x′ t′ a′ = LmA z t′′ a′′ unfolding a′′-def
     using 0(1,2) LmA-rename-freshA Lm.hyps tt1′ z(1) z(3,4)
     unfolding t′′-def by auto

    show R (subst (Lm x t) (Vr y) yy) (vsubstA a y yy)
      unfolding 0 1 11 using z apply (simp add: yy-y vsubstA-LmA)
      apply(rule R.Lm)
      apply(rule Lm[rule-format,THEN conjunct2, THEN conjunct2, rule-format])
      subgoal using tt′′ .
      subgoal using yy-y(1) yy-y(2) by auto
      subgoal unfolding t′′-def a′′-def
      apply(rule Lm[rule-format,THEN conjunct2, THEN conjunct2, rule-format])
        subgoal using tt1′ .
        subgoal using 0(1) by blast
        subgoal using 0(2) . .
      subgoal using z(4) . .
  qed
qed

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 ⇒ ′A where f t ≡ SOME a. R t a

lemma F-f: R t (f t)
  by (simp add: F-total f-def someI-ex)

lemma f-eq-F: f t = a ⟷ R t a
  by (meson F-f F-functional)
```

5.3 The full-fledged recursion theorem

```
theorem f-Vr[simp]: f (Vr x) = VrA x
  unfolding f-eq-F by (auto simp: F-f intro: R.intros)

theorem f-Ap[simp]: f (Ap t1 t2) = ApA t1 (f t1) t2 (f t2)
  unfolding f-eq-F by (auto simp: F-f intro: R.intros)
```

theorem *f-Lm*[*simp*]:
  $x \notin X \Longrightarrow f\ (Lm\ x\ t) = LmA\ x\ t\ (f\ t)$
  unfolding *f-eq-F* **by** (*auto simp*: *F-f intro*: *R.intros*)

theorem *f-subst*:
  $y \notin X \Longrightarrow z \notin X \Longrightarrow f\ (subst\ t\ (Vr\ y)\ z) = vsubstA\ (f\ t)\ y\ z$
  using *F-subst f-eq-F* **by** *blast*

theorem *f-fresh*:
  assumes $z \notin X$ *fresh z t*
  shows *freshA z* (*f t*)
  using *F-f F-fresh assms* **by** *blast*

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 $g = f$
  apply(*rule ext*)
  subgoal for *t* **using** *finite-X* **by** (*induct t rule*: *fresh-induct*, *auto*) **.**

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′: ⋀ *x y z a*. {*x*,*y*,*z*} ∩ *X* = {} ⟹
z ≠ *y* ⟹ *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 = λ*t1 a1 t2 a2*. *ApA a1 a2* **and**
LmA = λ*x t a*. *LmA x a*
apply *standard* **by** (*auto simp*: *vsubstA-VrA′ vsubstA-ApA′ vsubstA-LmA′ LmA-rename′*)

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* = {}
apply *standard* **by** (*auto simp*: *vsubstA-VrA′′ vsubstA-ApA′′ vsubstA-LmA′′ LmA-rename′′*)

lemma *triv*: *x*∉{} **by** *simp*

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*]:
$y1 \neq x \Longrightarrow substA\ (substA\ a\ (VrA\ y1)\ x)\ (VrA\ y2)\ x = substA\ a\ (VrA\ y1)\ x$
by (*metis substA-VrA substA-idem*)

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*
by (*metis* *substA-VrA* *substA-commute-diff*)

end

Any substitutive set is in particular a renset:

sublocale *Substitutive-Set* < *Renset* **where**
vsubstA = λ*a* *x*. *substA* *a* (*VrA* *x*) **apply** *standard*
using *substA-chain* *substA-commute-diff-var* *substA-VrA* **by** *auto*

interpretation *STerm*: *Substitutive-Set* **where** *substA* = *subst* **and** *VrA* = *Vr*
unfolding *Substitutive-Set-def*
using *fresh-subst-same* *subst-chain* *fresh-subst*
using *fresh-subst-id* *subst-comp-diff* **by** *auto*

6.2 Constructor-Enriched (CE) Substitutive Sets

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

ApA :: $'A \Rightarrow 'A \Rightarrow 'A$
and *LmA* :: $var \Rightarrow 'A \Rightarrow 'A$
assumes
substA-ApA: $\bigwedge$ *y* *z* *a1* *a2*.
substA (*ApA* *a1* *a2*) *y* *z* =
ApA (*substA* *a1* *y* *z*)
(*substA* *a2* *y* *z*)
and
substA-LmA: $\bigwedge$ *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: $\bigwedge$ *x* *y* *z* *a*.
$z \neq y \Longrightarrow$ *LmA* *x* (*substA* *a* (*VrA* *z*) *y*) = *LmA* *y* (*substA* (*substA* *a* (*VrA* *z*) *y*) (*VrA* *y*) *x*)
begin

lemma *LmA-cong*: $\bigwedge$ *z* *x* x' *a* a' *u*.
$z \neq u \Longrightarrow$
$z \neq x \Longrightarrow z \neq x' \Longrightarrow$
substA (*substA* *a* (*VrA* *u*) *z*) (*VrA* *z*) *x* = *substA* (*substA* a' (*VrA* *u*) *z*) (*VrA* *z*) x'
$\Longrightarrow$ *LmA* *x* (*substA* *a* (*VrA* *u*) *z*) = *LmA* x' (*substA* a' (*VrA* *u*) *z*)
by (*metis* *LmA-rename*)

lemma *substA-LmA-same*:
substA (*LmA x a*) *e x* = *LmA x a*
by (*metis vsubstA-id substA-LmA*)

lemma *substA-LmA-diff*:
freshA x e $\Longrightarrow$ *x* $\neq$ *z* $\Longrightarrow$ *substA* (*LmA x a*) *e z* = *LmA x* (*substA a e z*)
using *vsubstA-id* **by** (*metis substA-LmA*)

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′*
using *assms* **unfolding** *freshA-def* **by** (*meson assms*(*1*) *assms*(*2*) *local.freshA-iff-all-vvsubstA-idle freshA-iff-ex-vvsubstA-idle*)

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′*
by (*metis LmA-rename assms freshA-2-substA*)

lemma *freshA-VrA*: *z* ≠ *x* $\Longrightarrow$ *freshA z* (*VrA x*)
using *freshA-def substA-VrA* **by** *auto*

lemma *freshA-ApA*: ⋀ *z a1 a2*. *freshA z a1* $\Longrightarrow$ *freshA z a2* $\Longrightarrow$ *freshA z* (*ApA a1 a2*)
by (*simp add*: *freshA-iff-all-vvsubstA-idle substA-ApA*)

lemma *freshA-LmA-same*:
freshA x (*LmA x a*)
using *freshA-iff-all-vvsubstA-idle substA-LmA-same* **by** *presburger*

lemma *freshA-LmA*:
assumes *freshA z a*
shows *freshA z* (*LmA x a*)
by (*metis LmA-rename assms freshA-2-substA freshA-iff-all-vvsubstA-idle substA-LmA-same*)

end

Any CE substitutive set is in particular a CE renset:

sublocale *CE-Substitutive-Set* < *CE-Renset*
where *vsubstA* = λ*a x*. *substA a* (*VrA x*)
by (*simp add*: *CE-Renset-axioms-def CE-Renset-def LmA-rename Renset-axioms freshA-VrA substA-ApA substA-LmA-diff substA-LmA-same substA-VrA*)

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−
  have *finite* ({*z*} ∪ *FFvars s*)
   by (*simp add*: *cofinite-fresh*)
  thus *?thesis*
  proof(*induct t rule*: *fresh-induct*)
    case (*Vr x*)
    then show *?case*
      by (*simp add*: *substA-VrA*)
  next
    case (*Ap t1 t2*)
    then show *?case*
      using *substA-ApA* **by** *force*
  next
    case (*Lm x t*)
    then show *?case*
      by (*simp add*: *f-fresh substA-LmA-diff*)
  qed
qed

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*
  unfolding *Renset-def vss-def* **by** *auto*

Terms form a renset:

interpretation *Term*: *Renset* **where** *vsubstA* = *λt x. vsubst t x*
  unfolding *Renset-def*
 using *subst-Vr subst-comp-same*
 using *fresh-subst-same subst-chain*
  using *subst-commute-diff* **by** *auto*

... and a CE renset:

interpretation *Term*: *CE-Renset*
  where *vsubstA* = $\lambda t\ x.\ subst\ t\ (Vr\ x)$
    and *VrA* = *Vr* **and** *ApA* = *Ap* **and** *LmA* = *Lm*
  apply *standard*
  by (*auto simp add*: *Lm-subst-rename fresh-subst-same*)

7.2 Interpretation in semantic domains

type-synonym $'A\ I = (var \Rightarrow 'A) \Rightarrow 'A$

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

sublocale *CE-Renset*
  where $vsubstA = \lambda s\ x\ y\ \xi.\ s\ (\xi\ (y := \xi\ x))$
    and $VrA = \lambda x\ \xi.\ \xi\ x$
    and $ApA = \lambda i1\ i2\ \xi.\ ap\ (i1\ \xi)\ (i2\ \xi)$
    and $LmA = \lambda x\ i\ \xi.\ lm\ (\lambda d.\ i\ (\xi(x:=d)))$
  by *standard* (*auto simp*: *fun-eq-iff fun-upd-twist intro*!: *arg-cong*[*of* - - *lm*])

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 $Fmap :: ('A \Rightarrow 'A) \Rightarrow 'FA \Rightarrow 'FA$
  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*)
  using *Fmap-comp* **by** *auto*

end

locale *Renset-plus-Local-Functor* =
  Renset vsubstA + *Local-Functor Fmap*
  for $vsubstA :: 'A \Rightarrow var \Rightarrow var \Rightarrow 'A$

  and *Fmap* :: ('*A* ⇒ '*A*) ⇒ '*FA* ⇒ '*FA*
begin

sublocale *F*: *Renset* **where** *vsubstA* =
 λ*k x y*. *Fmap* (λ*a*. *vsubstA a x y*) *k*
 apply *standard*
 subgoal by (*metis Fmap-id eq-id-iff vsubstA-id*)
 subgoal unfolding *Fmap-comp'*[*symmetric*] *o-def* **by** *simp*
 subgoal unfolding *Fmap-comp'*[*symmetric*] *o-def*
  by (*simp add*: *vsubstA-chain*)
 subgoal unfolding *Fmap-comp'*[*symmetric*] *o-def*
  using *vsubstA-commute-diff* **by** *force* **.**

end

7.4 The length of a term via renaming-based recursion

interpretation *length* : *CE-Renset*
 where *vsubstA* = λ*n x y*. *n*
  and *VrA* = λ*x*. *1*
  and *ApA* = λ*n1 n2*. *max n1 n2* + *1*
  and *LmA* = λ*x n*. *n* + *1*
 apply *standard* **by** *auto*

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* = λ*n x y*. *n*
  and *VrA* = λ*x*. *0*
  and *ApA* = λ*n1 n2*. *n1* + *n2*
  and *LmA* = λ*x n*. *n* + *1*
 apply *standard* **by** *auto*

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* =
  λ*f z y*. λ*x*. *if x* ∉ {*y*,*z*}
   then f x
   else if x = *z* ∧ *x* ≠ *y then f x* + *f y*

else if $x = y \wedge x \neq z$ *then* $(0::nat)$
else f y
and $VrA = \lambda y.\ \lambda x.$ *if* $x = y$ *then 1 else 0*
and $ApA = \lambda f1\ f2.\ \lambda x.\ f1\ x + f2\ x$
and $LmA = \lambda y\ f.\ \lambda x.$ *if* $x = y$ *then 0 else f x*
apply *standard* **by** (*auto simp*: *fun-eq-iff*)

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 $VrA = \lambda y.$ *if y = x then s else Vr y*
and *ApA* = *Ap*
and *LmA* = *Lm*
and $X = FFvars\ s \cup \{x\}$
apply *standard* **by** (*auto simp*: *fun-eq-iff cofinite-fresh Term.LmA-rename*)

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−
have ($\lambda t.$ *subst t s x*) = *ssb.f*
apply(*rule ssb.f-unique*) **by** *auto*
thus *?thesis* **unfolding** *fun-eq-iff* **by** *auto*
qed

end

7.8 Parrallel substitution via renaming-based recursion

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

definition *X* **where**
$X = supp\ \varrho \cup \bigcup\ \{FFvars\ (get\ \varrho\ x) \mid x\ .\ x \in supp\ \varrho\}$

lemma *finite-Supp*: *finite X*
unfolding *X-def* **unfolding** *finite-Un* **apply** *safe*

by (*auto simp*: *finite-supp cofinite-fresh*)

sublocale *canEta′* : *BCE-Renset*
where *vsubstA* = *vsubst*
and *VrA* = λy. *get* ϱ *y*
and *ApA* = *Ap*
and *LmA* = *Lm*
and *X* = *X*
apply *standard*
by (*auto simp*: *fun-eq-iff cofinite-fresh finite-Supp Term.LmA-rename X-def finite-supp*)
(*metis fresh-subst-id mem-Collect-eq subst-Vr supp-get*)

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*) **.**

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* **.**

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.