

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

Andrei Popescu

April 18, 2024

## Abstract

I formalize the notion of *renaming-enriched sets* (*rensets* for 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 rerset 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 renssets that axiomatize term-for-variable substitution, called *substitutive sets*, which yields a corresponding recursion principle.

## Contents

<b>1</b>	<b>Lambda Terms</b>	<b>2</b>
1.1	Variables . . . . .	2
1.2	Pre-terms and operators on them . . . . .	3
1.3	Terms via quotienting pre-terms . . . . .	7
1.4	Fresh induction . . . . .	10
1.5	Substitution . . . . .	11
1.6	Renaming (a.k.a. variable-for-variable substitution) . . . . .	15
1.7	Syntactic environments . . . . .	16
<b>2</b>	<b>Renaming-Enriched Sets (Rensets)</b>	<b>17</b>
2.1	Rensets . . . . .	17
2.2	Finitely supported renssets . . . . .	18
2.3	Morphisms between renssets . . . . .	19
<b>3</b>	<b>Nominal sets</b>	<b>19</b>
3.1	From Rensets to Nominal Sets . . . . .	20

<b>4</b>	<b>Renset-based Recursion</b>	<b>22</b>
<b>5</b>	<b>Full-fledged, Barendregt-constructor-enriched recursion</b>	<b>22</b>
5.1	The relational version of the recursor . . . . .	24
5.2	The functional version of the recursor . . . . .	25
5.3	The full-fledged recursion theorem . . . . .	25
5.4	The particular case of iteration . . . . .	26
<b>6</b>	<b>Substitutive Sets</b>	<b>27</b>
6.1	Substitutive Sets . . . . .	28
6.2	Constructor-Enriched (CE) Substitutive Sets . . . . .	29
6.3	The recursion theorem for substitutive sets . . . . .	30
<b>7</b>	<b>Examples of Rensets and Renaming-Based Recursion</b>	<b>30</b>
7.1	Variables and terms as renssets . . . . .	31
7.2	Interpretation in semantic domains . . . . .	31
7.3	Closure of renssets under functors . . . . .	31
7.4	The length of a term via renaming-based recursion . . . . .	32
7.5	Counting the lambda-abstractions in a term via renaming-based recursion . . . . .	32
7.6	Counting free occurrences of a variable in a term via renaming-based recursion . . . . .	32
7.7	Substitution via renaming-based recursion . . . . .	33
7.8	Parallel substitution via renaming-based recursion . . . . .	33
7.9	Counting bound variables via renaming-based recursion . . . . .	34
7.10	Testing eta-reducibility via renaming-based recursion . . . . .	34

## 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  $\equiv$  undefined

```

### 1.1 Variables

```

datatype var = Variable nat

```

## 1.2 Pre-terms and operators on them

**datatype**  $ptrm = PVr\ var \mid PAp\ ptrm\ ptrm \mid PLm\ var\ ptrm$

**inductive**  $pfresh :: var \Rightarrow ptrm \Rightarrow bool$  **where**

$PVr[intro]: z \neq x \Longrightarrow pfresh\ z\ (PVr\ x)$

$PAp[intro]: pfresh\ z\ t1 \Longrightarrow pfresh\ z\ t2 \Longrightarrow pfresh\ z\ (PAp\ t1\ t2)$

$PLm[intro]: z = x \vee pfresh\ z\ t \Longrightarrow pfresh\ z\ (PLm\ x\ t)$

**lemma**  $pfresh-simps[simp]:$

$pfresh\ z\ (PVr\ x) \longleftrightarrow z \neq x$

$pfresh\ z\ (PAp\ t1\ t2) \longleftrightarrow pfresh\ z\ t1 \wedge pfresh\ z\ t2$

$pfresh\ z\ (PLm\ x\ t) \longleftrightarrow z = x \vee pfresh\ z\ t$

$\langle proof \rangle$

**lemma**  $inj\text{-}Variable: inj\ Variable$

$\langle proof \rangle$

**lemma**  $infinite\text{-}var: infinite\ (UNIV::var\ set)$

$\langle proof \rangle$

**lemma**  $exists\text{-}var:$

**assumes**  $finite\ X$

**shows**  $\exists x::var. x \notin X$

$\langle proof \rangle$

**lemma**  $finite\text{-}neg\text{-}imp:$

**assumes**  $finite\ \{x. \neg \varphi\ x\}$  **and**  $finite\ \{x. \chi\ x\}$

**shows**  $finite\ \{x. \varphi\ x \longrightarrow \chi\ x\}$

$\langle proof \rangle$

**lemma**  $cofinite\text{-}pfresh: finite\ \{x. \neg pfresh\ x\ t\}$

$\langle proof \rangle$

**lemma**  $cofinite\text{-}list\text{-}ptrm: finite\ \{x. \exists t \in set\ ts. \neg pfresh\ x\ t\}$

$\langle proof \rangle$

**lemma**  $exists\text{-}pfresh\text{-}set:$

**assumes**  $finite\ X$

**shows**  $\exists z. z \notin X \wedge z \notin set\ xs \wedge (\forall t \in set\ ts. pfresh\ z\ t)$

$\langle proof \rangle$

**lemma**  $exists\text{-}pfresh:$

$\exists z. z \notin set\ xs \wedge (\forall t \in set\ ts. pfresh\ z\ t)$

$\langle proof \rangle$

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

**lemma**  $pickFreshS$ :  
**assumes**  $finite X$   
**shows**  $pickFreshS X xs ts \notin X \wedge pickFreshS X xs ts \notin set xs \wedge$   
 $(\forall t \in set ts. pfresh (pickFreshS X xs ts) t)$   
 $\langle proof \rangle$

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

**definition**  $pickFresh \equiv pickFreshS \{\}$

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

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

**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**  $sw-diff[simp]: \bigwedge x y z. x \neq y \implies x \neq z \implies sw x y z = x$   
 $\langle proof \rangle$

**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**  $sw-invol[simp]: sw (sw x y z) y z = x$   
 $\langle proof \rangle$

**lemma**  $sw-invol2: sw (sw x y z) z y = x$   
 $\langle proof \rangle$

**lemma**  $sw-inj[iff]: sw x z1 z2 = sw y z1 z2 \iff x = y$   
 $\langle proof \rangle$

**lemma**  $sw-surj: \exists y. x = sw y z1 z2$   
 $\langle proof \rangle$

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

|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  $\longleftrightarrow$  s = t  
⟨proof⟩

**lemma** pswap-surj:  $\exists t. s = \text{pswap } t \ z1 \ z2$   
⟨proof⟩

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

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

**inductive** alpha :: ptrm  $\Rightarrow$  ptrm  $\Rightarrow$  bool **where**

PVr[intro]: alpha (PVr x) (PVr x)

PAP[intro]: alpha s s'  $\Longrightarrow$  alpha t t'  $\Longrightarrow$  alpha (PAP s t) (PAP s' t')

PLM[intro]:

(z = x  $\vee$  pfresh z t)  $\Longrightarrow$  (z = x'  $\vee$  pfresh z t')

$\Longrightarrow$  alpha (pswap t z x) (pswap t' z x')  $\Longrightarrow$  alpha (PLM x t) (PLM x' t')

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

**lemma** alpha-eq-PVr[simp]: alpha t (PVr x)  $\longleftrightarrow$  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-pfresch-imp*:  
**assumes** *alpha s t and pfresh x s* **shows** *pfresh x t*  
*<proof>*

**lemma** *alpha-pfresch-iff*:  
**assumes** *alpha s t*  
**shows** *pfresh x s ⟷ pfresh x t*  
*<proof>*

**lemma** *pswap-pfresch-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 ∨ pfresh z t* and *z = x' ∨ pfresh z t'*  
 shows *alpha (pswap t z x) (pswap t' z x')*  
 ⟨*proof*⟩

**lemma** *pfresh-pswap-alpha*:  
 assumes *y = x ∨ pfresh y t* and *z = x ∨ 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 ≠ y* and *y = sw x z1 z2 ∨ pfresh y (pswap t z1 z2)*  
 and *y' = x ∨ 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*

$t$   
 $\langle proof \rangle$

**lemma** *swap-id[simp]*:  
 $swap (swap t z x) z x = t$   
 $\langle proof \rangle$

**lemma** *fresh-PVr[simp]*:  $fresh\ x\ (Vr\ y) \longleftrightarrow x \neq y$   
 $\langle proof \rangle$

**lemma** *fresh-Ap[simp]*:  $fresh\ z\ (Ap\ t1\ t2) \longleftrightarrow fresh\ z\ t1 \wedge fresh\ z\ t2$   
 $\langle proof \rangle$

**lemma** *fresh-Lm[simp]*:  $fresh\ z\ (Lm\ x\ t) \longleftrightarrow (z = x \vee fresh\ z\ t)$   
 $\langle proof \rangle$

**lemma** *Lm-swap-rename*:  
**assumes**  $z = x \vee fresh\ z\ t$   
**shows**  $Lm\ z\ (swap\ t\ z\ x) = Lm\ x\ t$   
 $\langle proof \rangle$

**lemma** *abs-trm-PVr*:  $abs-trm\ (PVr\ x) = Vr\ x$   
 $\langle proof \rangle$

**lemma** *abs-trm-PAp*:  $abs-trm\ (PAp\ t1\ t2) = Ap\ (abs-trm\ t1)\ (abs-trm\ t2)$   
 $\langle proof \rangle$

**lemma** *abs-trm-PLm*:  $abs-trm\ (PLm\ x\ t) = Lm\ x\ (abs-trm\ t)$   
 $\langle proof \rangle$

**lemma** *abs-trm-pswap*:  $abs-trm\ (pswap\ t\ z1\ z2) = swap\ (abs-trm\ t)\ z1\ z2$   
 $\langle proof \rangle$

**lemma** *swap-Vr[simp]*:  $swap\ (Vr\ x)\ z1\ z2 = Vr\ (sw\ x\ z1\ z2)$   
 $\langle proof \rangle$

**lemma** *swap-Ap[simp]*:  $swap\ (Ap\ t1\ t2)\ z1\ z2 = Ap\ (swap\ t1\ z1\ z2)\ (swap\ t2\ z1\ z2)$   
 $\langle proof \rangle$

**lemma** *swap-Lm[simp]*:  $swap\ (Lm\ x\ t)\ z1\ z2 = Lm\ (sw\ x\ z1\ z2)\ (swap\ t\ z1\ z2)$   
 $\langle proof \rangle$

**lemma** *Lm-sameVar-inj[simp]*:  $Lm\ x\ t = Lm\ x\ t1 \longleftrightarrow t = t1$   
 $\langle proof \rangle$

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



$\langle proof \rangle$

**lemma** *alpha-rep-abs-trm*:  $alpha (rep-trm (abs-trm t)) t$   
 $\langle proof \rangle$

**lemma** *swap-fresh-eq*: **assumes**  $x: fresh\ x\ t$  **and**  $y: fresh\ y\ t$   
**shows**  $swap\ t\ x\ y = t$   
 $\langle proof \rangle$

**lemma** *bij-sw:bij*  $(\lambda x. sw\ x\ z1\ z2)$   
 $\langle proof \rangle$

**lemma** *sw-set*:  $x \in X = ((sw\ x\ z1\ z2) \in (\lambda x. sw\ x\ z1\ z2) ' X)$   
 $\langle proof \rangle$

**lemma** *ddepth-Vr[simp]*:  $ddepth (Vr\ x) = 0$   
 $\langle proof \rangle$

**lemma** *ddepth-Ap[simp]*:  $ddepth (Ap\ t1\ t2) = Suc (ddepth\ t1 + ddepth\ t2)$   
 $\langle proof \rangle$

**lemma** *ddepth-Lm[simp]*:  $ddepth (Lm\ x\ t) = Suc (ddepth\ t)$   
 $\langle proof \rangle$

**lemma** *trm-nchotomy*:  
 $(\exists x. tt = Vr\ x) \vee (\exists t1\ t2. tt = Ap\ t1\ t2) \vee (\exists x\ t. tt = Lm\ x\ t)$   
 $\langle proof \rangle$

**lemma** *trm-exhaust*[*case-names Vr Ap Lm, cases type: trm*]:  
 $(\bigwedge x. tt = Vr\ x \implies P) \implies$   
 $(\bigwedge t1\ t2. tt = Ap\ t1\ t2 \implies P) \implies (\bigwedge x\ t. tt = Lm\ x\ t \implies P) \implies P$   
 $\langle proof \rangle$

**lemma** *Vr-Ap-diff[simp]*:  $Vr\ x \neq Ap\ t1\ t2 \iff Ap\ t1\ t2 \neq Vr\ x$   
 $\langle proof \rangle$

**lemma** *Vr-Lm-diff[simp]*:  $Vr\ x \neq Lm\ y\ t \iff Lm\ y\ t \neq Vr\ x$   
 $\langle proof \rangle$

**lemma** *Ap-Lm-diff[simp]*:  $Ap\ t1\ t2 \neq Lm\ y\ t \iff Lm\ y\ t \neq Ap\ t1\ t2$   
 $\langle proof \rangle$

**lemma** *Vr-inj[simp]*:  $(Vr\ x = Vr\ y) \iff x = y$   
 $\langle proof \rangle$

**lemma** *Ap-inj[simp]*:  $(Ap\ t1\ t2 = Ap\ t1'\ t2') \iff t1 = t1' \wedge t2 = t2'$   
 $\langle proof \rangle$

**abbreviation**  $Fvars :: ptrm \Rightarrow var\ set$  **where**

$Fvars\ t \equiv \{x. \neg\ pfresh\ x\ t\}$

**abbreviation**  $FFvars :: trm \Rightarrow var\ set$  **where**

$FFvars\ t \equiv \{x. \neg\ fresh\ x\ t\}$

**lemma** *cofinite-fresh*:  $finite\ (FFvars\ t)$

$\langle proof \rangle$

**lemma** *exists-fresh-set*:

**assumes**  $finite\ X$

**shows**  $\exists\ z. z \notin X \wedge z \notin set\ xs \wedge (\forall t \in set\ ts. fresh\ z\ t)$

$\langle proof \rangle$

**definition**  $ppickFreshS :: var\ set \Rightarrow var\ list \Rightarrow trm\ list \Rightarrow var$  **where**

$ppickFreshS\ X\ xs\ ts \equiv SOME\ z. z \notin X \wedge z \notin set\ xs \wedge$   
 $(\forall t \in set\ ts. fresh\ z\ t)$

**lemma** *ppickFreshS*:

**assumes**  $finite\ X$

**shows**

$ppickFreshS\ X\ xs\ ts \notin X \wedge$

$ppickFreshS\ X\ xs\ ts \notin set\ xs \wedge$

$(\forall t \in set\ ts. fresh\ (ppickFreshS\ X\ xs\ ts)\ t)$

$\langle proof \rangle$

**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**  $ppickFresh \equiv ppickFreshS\ \{\}$

**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*:

$fresh\ x\ t \longleftrightarrow finite\ \{y. swap\ t\ y\ x \neq t\}$

$\langle proof \rangle$

## 1.4 Fresh induction

**lemma** *swap-induct*[*case-names Vr Ap Lm*]:

**assumes**  $Vr: \bigwedge x. \varphi\ (Vr\ x)$

**and**  $Ap: \bigwedge t1\ t2. \varphi\ t1 \implies \varphi\ t2 \implies \varphi\ (Ap\ t1\ t2)$

**and**  $Lm: \bigwedge x\ t. (\forall z. \varphi\ (swap\ t\ z\ x)) \implies \varphi\ (Lm\ x\ t)$

**shows**  $\varphi\ t$

$\langle proof \rangle$

**lemma** *fresh-induct*[*consumes 1, case-names Vr Ap Lm*]:

**assumes** *finite X and*  $\bigwedge x. \varphi (Vr\ x)$   
**and**  $\bigwedge t1\ t2. \varphi\ t1 \implies \varphi\ t2 \implies \varphi\ (Ap\ t1\ t2)$   
**and**  $\bigwedge x\ t. \varphi\ t \implies x \notin X \implies \varphi\ (Lm\ x\ t)$   
**shows**  $\varphi\ t$   
 $\langle proof \rangle$

**lemma** *plain-induct*[*case-names Vr Ap Lm*]:

**assumes**  $\bigwedge x. \varphi (Vr\ x)$   
**and**  $\bigwedge t1\ t2. \varphi\ t1 \implies \varphi\ t2 \implies \varphi\ (Ap\ t1\ t2)$   
**and**  $\bigwedge x\ t. \varphi\ t \implies \varphi\ (Lm\ x\ t)$   
**shows**  $\varphi\ t$   
 $\langle proof \rangle$

## 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*:

$x \neq y \implies \text{substRel } (Vr\ x)\ s\ y\ (Vr\ x)$

|*substRel-Ap*:

$\text{substRel } t1\ s\ y\ t1' \implies \text{substRel } t2\ s\ y\ t2' \implies$

$\text{substRel } (Ap\ t1\ t2)\ s\ y\ (Ap\ t1'\ t2')$

|*substRel-Lm*:

$x \neq y \implies \text{fresh } x\ s \implies \text{substRel } t\ s\ y\ t' \implies$

$\text{substRel } (Lm\ x\ t)\ s\ y\ (Lm\ x\ t')$

**lemma** *substRel-Vr-invert*:

**assumes**  $\text{substRel } (Vr\ x)\ t\ y\ t'$

**shows**  $(x = y \wedge t = t') \vee (x \neq y \wedge t' = Vr\ x)$

$\langle proof \rangle$

**lemma** *substRel-Ap-invert*:

**assumes**  $\text{substRel } (Ap\ t1\ t2)\ s\ y\ t'$

**shows**  $\exists t1'\ t2'. t' = Ap\ t1'\ t2' \wedge \text{substRel } t1\ s\ y\ t1' \wedge \text{substRel } t2\ s\ y\ t2'$

$\langle proof \rangle$

**lemma** *substRel-Lm-invert-aux*:

**assumes**  $\text{substRel } (Lm\ x\ t)\ s\ y\ tt'$

**shows**  $\exists x1\ t1\ t1'.$

$x1 \neq y \wedge \text{fresh } x1\ s \wedge$

$Lm\ x\ t = Lm\ x1\ t1 \wedge tt' = Lm\ x1\ t1' \wedge \text{substRel } t1\ s\ y\ t1'$

$\langle proof \rangle$

**lemma** *substRel-swap*:

**assumes**  $\text{substRel } t\ s\ y\ tt$

**shows**  $\text{substRel } (\text{swap } t\ z1\ z2)\ (\text{swap } s\ z1\ z2)\ (\text{sw } y\ z1\ z2)\ (\text{swap } tt\ z1\ z2)$

*<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**  $\exists t'. tt' = Lm x t' \wedge \text{substRel } t s y t'$

*<proof>*

**lemma** *substRel-total*:

$\exists t'. \text{substRel } t s y t'$

*<proof>*

**lemma** *substRel-functional*:

**assumes** *substRel t s y t'* **and** *substRel t s y tt'*  
**shows**  $t' = tt'$

*<proof>*

**definition** *subst* :: *trm*  $\Rightarrow$  *trm*  $\Rightarrow$  *var*  $\Rightarrow$  *trm* **where**

*subst t s x*  $\equiv$  *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*  $\implies$   $tt = \text{subst } t s x$

*<proof>*

**lemma**

*subst-Vr[simp]*: *subst (Vr x) t z* = (if  $x = z$  then  $t$  else  $\text{Vr } x$ )

**and**

*subst-Ap[simp]*: *subst (Ap s1 s2) t z* =  $\text{Ap } (\text{subst } s1 t z) (\text{subst } s2 t z)$

**and**

*subst-Lm[simp]*:

$x \neq z \implies \text{fresh } x t \implies \text{subst } (Lm x s) t z = Lm x (\text{subst } s t z)$

*<proof>*

**lemma** *fresh-subst*:

*fresh z (subst s t x)*  $\iff (z = x \vee \text{fresh } z s) \wedge (\text{fresh } x s \vee \text{fresh } z t)$

*<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* =  $s$

$\langle proof \rangle$

**lemma** *Lm-swap-cong*:

**assumes**  $z = x \vee \text{fresh } z \text{ } s \text{ } z = y \vee \text{fresh } z \text{ } t$  **and**  $\text{swap } s \text{ } z \text{ } x = \text{swap } t \text{ } z \text{ } y$   
**shows**  $Lm \text{ } x \text{ } s = Lm \text{ } y \text{ } t$

$\langle proof \rangle$

**lemma** *fresh-swap[simp]*:  $\text{fresh } x \text{ } (\text{swap } t \text{ } z1 \text{ } z2) \longleftrightarrow \text{fresh } (sw \text{ } x \text{ } z1 \text{ } z2) \text{ } t$

$\langle proof \rangle$

**lemma** *swap-subst*:

$\text{swap } (\text{subst } s \text{ } t \text{ } x) \text{ } z1 \text{ } z2 = \text{subst } (\text{swap } s \text{ } z1 \text{ } z2) \text{ } (\text{swap } t \text{ } z1 \text{ } z2) \text{ } (sw \text{ } x \text{ } z1 \text{ } z2)$

$\langle proof \rangle$

**lemma** *subst-Lm-same[simp]*:  $\text{subst } (Lm \text{ } x \text{ } s) \text{ } t \text{ } x = Lm \text{ } x \text{ } s$

$\langle proof \rangle$

**lemma** *fresh-subst-same*:

**assumes**  $y \neq z$  **shows**  $\text{fresh } y \text{ } (\text{subst } t \text{ } (Vr \text{ } z) \text{ } y)$

$\langle proof \rangle$

**lemma** *subst-comp-same*:

$\text{subst } (\text{subst } s \text{ } t \text{ } x) \text{ } t1 \text{ } x = \text{subst } s \text{ } (\text{subst } t \text{ } t1 \text{ } x) \text{ } x$

$\langle proof \rangle$

**lemma** *subst-comp-diff*:

**assumes**  $x \neq x1$   $\text{fresh } x \text{ } t1$

**shows**  $\text{subst } (\text{subst } s \text{ } t \text{ } x) \text{ } t1 \text{ } x1 = \text{subst } (\text{subst } s \text{ } t1 \text{ } x1) \text{ } (\text{subst } t \text{ } t1 \text{ } x1) \text{ } x$

$\langle proof \rangle$

**lemma** *subst-comp-diff-var*:

**assumes**  $x \neq x1$   $x \neq z1$

**shows**  $\text{subst } (\text{subst } s \text{ } t \text{ } x) \text{ } (Vr \text{ } z1) \text{ } x1 =$

$\text{subst } (\text{subst } s \text{ } (Vr \text{ } z1) \text{ } x1) \text{ } (\text{subst } t \text{ } (Vr \text{ } z1) \text{ } x1) \text{ } x$

$\langle proof \rangle$

**lemma** *subst-chain*:

**assumes**  $\text{fresh } u \text{ } s$

**shows**  $\text{subst } (\text{subst } s \text{ } (Vr \text{ } u) \text{ } x) \text{ } t \text{ } u = \text{subst } s \text{ } t \text{ } x$

$\langle proof \rangle$

**lemma** *subst-repeated-Vr*:

$\text{subst } (\text{subst } t \text{ } (Vr \text{ } x) \text{ } y) \text{ } (Vr \text{ } u) \text{ } x =$

$\text{subst } (\text{subst } t \text{ } (Vr \text{ } u) \text{ } x) \text{ } (Vr \text{ } u) \text{ } y$

$\langle proof \rangle$

**lemma** *subst-commute-same*:

$\text{subst } (\text{subst } d \text{ } (Vr \text{ } u) \text{ } x) \text{ } (Vr \text{ } u) \text{ } y = \text{subst } (\text{subst } d \text{ } (Vr \text{ } u) \text{ } y) \text{ } (Vr \text{ } u) \text{ } x$

$\langle proof \rangle$

**lemma** *subst-commute-diff*:

**assumes**  $x \neq v \ y \neq u \ x \neq y$

**shows**  $subst (subst t (Vr u) x) (Vr v) y = subst (subst t (Vr v) y) (Vr u) x$

$\langle proof \rangle$

**lemma** *subst-same-id*:

**assumes**  $z1 \neq y$

**shows**  $subst (subst t (Vr z1) y) (Vr z2) y = subst t (Vr z1) y$

$\langle proof \rangle$

**lemma** *swap-from-subst*:

**assumes**  $yy: yy \notin \{z1, z2\}$  *fresh*  $yy \ t$

**shows**  $swap \ t \ z1 \ z2 = subst (subst (subst t (Vr yy) z1) (Vr z1) z2) (Vr z2) yy$

$\langle proof \rangle$

**lemma** *subst-two-ways'*:

**fixes**  $t \ yy \ x$

**assumes**  $yy: yy \notin \{z1, z2\}$   $yy' \notin \{z1, z2\}$   $x \notin \{yy, yy'\}$

**defines**  $tt \equiv subst (subst t (Vr x) yy) (Vr x) yy'$

**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**  $?L = ?R$ )

$\langle proof \rangle$

**lemma** *subst-two-ways''*:

**assumes**  $xx \notin \{x, z1, z2, uu, vv\} \wedge fresh \ xx \ t$

$vv \notin \{x, z1, z2\} \wedge fresh \ vv \ t$

$yy \notin \{z1, z2\} \wedge fresh \ yy \ t$

**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**  $?L = ?R$ )

$\langle proof \rangle$

**lemma** *subst-two-ways''-aux*:

**fixes**  $t \ z1 \ xx \ z2 \ vv$

**assumes**  $xx \notin \{x, z1, z2, uu, vv\}$

$vv \notin \{x, z1, z2\}$

$yy \notin \{z1, z2\}$

**defines**  $tt \equiv subst (subst (subst t (Vr z1) xx) (Vr z1) yy) (Vr z1) vv$

**shows**

$subst (subst (subst (subst (subst tt (Vr xx) x) (Vr vv) z1) (Vr z1) z2) (Vr z2) vv) (Vr vv) xx =$

$subst (subst (subst (subst tt (Vr yy) z1) (Vr z1) z2) (Vr z2) yy) (Vr vv) (sw \ x \ z1 \ z2)$

$z2$ )  
(proof)

**lemma** *fresh-cases*[cases pred: *fresh*, case-names *Vr Ap Lm*]:

*fresh a1 a2*  $\implies$   
( $\bigwedge z x. a1 = z \implies a2 = Vr\ x \implies z \neq x \implies P$ )  $\implies$   
( $\bigwedge z t1\ t2. a1 = z \implies a2 = Ap\ t1\ t2 \implies fresh\ z\ t1 \implies fresh\ z\ t2 \implies P$ )  $\implies$   
( $\bigwedge z x\ t. a1 = z \implies a2 = Lm\ x\ t \implies z = x \vee fresh\ z\ t \implies P$ )  $\implies P$   
(proof)

**definition** *vss* :: *var*  $\Rightarrow$  *var*  $\Rightarrow$  *var*  $\Rightarrow$  *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*  $\vee$  *fresh z t*  
**shows** *Lm z (subst t (Vr z) x)* = *Lm x t*  
(proof)

**lemma** *Lm-subst-cong*:

*z = x*  $\vee$  *fresh z s*  $\implies$  *z = y*  $\vee$  *fresh z t*  $\implies$   
*subst s (Vr z) x* = *subst t (Vr z) y*  $\implies$  *Lm x s* = *Lm y t*  
(proof)

**lemma** *Lm-eq-elim*:

*Lm x s* = *Lm y t*  $\implies$  *z = x*  $\vee$  *fresh z s*  $\implies$  *z = y*  $\vee$  *fresh z t*  
 $\implies$  *swap s z x* = *swap t z y*  
(proof)

**lemma** *Lm-eq-elim-subst*:

*Lm x s* = *Lm y t*  $\implies$  *z = x*  $\vee$  *fresh z s*  $\implies$  *z = y*  $\vee$  *fresh z t*  
 $\implies$   
*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*  $\equiv$   $\lambda t\ x\ y. subst\ t\ (Vr\ x)\ y$

**inductive** *substConnect* :: *trm*  $\Rightarrow$  *trm*  $\Rightarrow$  *bool* **where**

*Refl*: *substConnect t t*

| *Step*:  $\text{substConnect } t \ t' \implies \text{substConnect } t \ (\text{vsubst } t' \ z \ x)$

**lemma** *ddepth-swap*:

$\text{ddepth } (\text{swap } t \ z \ x) = \text{ddepth } t$   
*<proof>*

**lemma** *ddepth-subst-Vr[simp]*:

$\text{ddepth } (\text{vsubst } t \ z \ x) = \text{ddepth } t$   
*<proof>*

**lemma** *substConnect-depth*:

**assumes**  $\text{substConnect } t \ t'$  **shows**  $\text{ddepth } t = \text{ddepth } t'$   
*<proof>*

**lemma** *substConnect-induct[case-names Vr Ap Lm]*:

**assumes**  $Vr: \bigwedge x. \varphi (Vr \ x)$   
**and**  $Ap: \bigwedge t1 \ t2. \varphi \ t1 \implies \varphi \ t2 \implies \varphi (Ap \ t1 \ t2)$   
**and**  $Lm: \bigwedge x \ t. (\forall t'. \text{substConnect } t \ t' \longrightarrow \varphi \ t') \implies \varphi (Lm \ x \ t)$   
**shows**  $\varphi \ t$   
*<proof>*

## 1.7 Syntactic environments

**typedef**  $\text{fenv} = \{f :: \text{var} \Rightarrow \text{trm} . \text{finite } \{x. f \ x \neq Vr \ x\}\}$   
*<proof>*

**definition**  $\text{get} :: \text{fenv} \Rightarrow \text{var} \Rightarrow \text{trm}$  **where**

$\text{get } f \ x \equiv \text{Rep-fenv } f \ x$

**definition**  $\text{upd} :: \text{fenv} \Rightarrow \text{var} \Rightarrow \text{trm} \Rightarrow \text{fenv}$  **where**

$\text{upd } f \ x \ t = \text{Abs-fenv } ((\text{Rep-fenv } f)(x:=t))$

**definition**  $\text{supp} :: \text{fenv} \Rightarrow \text{var set}$  **where**

$\text{supp } f \equiv \{x. \text{get } f \ x \neq Vr \ x\}$

**lemma** *finite-supp*:  $\text{finite } (\text{supp } f)$

*<proof>*

**lemma** *finite-upd*:

**assumes**  $\text{finite } \{x. f \ x \neq Vr \ x\}$   
**shows**  $\text{finite } \{x. (f(y:=t)) \ x \neq Vr \ x\}$

*<proof>*

**lemma** *get-upd-same[simp]*:  $\text{get } (\text{upd } f \ x \ t) \ x = t$

**and** *get-upd-diff[simp]*:  $x \neq y \implies \text{get } (\text{upd } f \ x \ t) \ y = \text{get } f \ y$

**and** *upd-upd-same[simp]*:  $\text{upd } (\text{upd } f \ x \ t) \ x \ s = \text{upd } f \ x \ s$

**and** *upd-upd-diff*:  $x \neq y \implies \text{upd } (\text{upd } f \ x \ t) \ y \ s = \text{upd } (\text{upd } f \ y \ s) \ x \ t$

**and** *supp-get[simp]*:  $x \notin \text{supp } \varrho \implies \text{get } \varrho \ x = Vr \ x$

*<proof>*



end

## 2 Renaming-Enriched Sets (Rensets)

**theory** *Rensets*  
  **imports** *Lambda-Terms*  
**begin**

This theory defines renssets and proves their basic properties.

### 2.1 Rensets

**locale** *Renset* =  
  **fixes** *vsubstA* :: 'A  $\Rightarrow$  var  $\Rightarrow$  var  $\Rightarrow$  'A  
  **assumes**  
    *vsubstA-id[simp]*:  $\bigwedge x a. \text{vsubstA } a \ x \ x = a$   
    **and**  
    *vsubstA-idem[simp]*:  $\bigwedge x \ y1 \ y2 \ a. \ y1 \neq x \Longrightarrow \text{vsubstA } (\text{vsubstA } a \ y1 \ x) \ y2 \ x = \text{vsubstA } a \ y1 \ x$   
    **and**  
    *vsubstA-chain*:  $\bigwedge u \ x1 \ x2 \ x3 \ a. \ u \neq x2 \Longrightarrow \text{vsubstA } (\text{vsubstA } (\text{vsubstA } a \ u \ x2) \ x2 \ x1) \ x3 \ x2 = \text{vsubstA } (\text{vsubstA } a \ u \ x2) \ x3 \ x1$   
    **and**  
    *vsubstA-commute-diff*:  
     $\bigwedge x \ y \ u \ a \ v. \ x \neq v \Longrightarrow y \neq u \Longrightarrow x \neq y \Longrightarrow \text{vsubstA } (\text{vsubstA } a \ u \ x) \ v \ y = \text{vsubstA } (\text{vsubstA } a \ v \ y) \ u \ x$   
**begin**

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

**lemma** *freshA-vsubstA-idle*:  
  **assumes**  $n: \text{freshA } x \ a$  **and**  $xy: x \neq y$   
  **shows**  $\text{vsubstA } a \ y \ x = a$   
*<proof>*

**lemma** *vsubstA-chain-freshA*:  
  **assumes**  $\text{freshA } x2 \ a$   
  **shows**  $\text{vsubstA } (\text{vsubstA } a \ x2 \ x1) \ x3 \ x2 = \text{vsubstA } a \ x3 \ x1$   
*<proof>*

**lemma** *freshA-vsubstA*:  
  **assumes**  $\text{freshA } u \ a$  **and**  $u \neq y$   
  **shows**  $\text{freshA } u \ (\text{vsubstA } a \ y \ x)$

*<proof>*

**lemma** *freshA-vsubstA2*:

**assumes** *freshA z a*  $\vee$  *z = x* **and** *freshA x a*  $\vee$  *z  $\neq$  y*  
**shows** *freshA z (vsubstA a y x)*

*<proof>*

**lemma** *vsubstA-idle-freshA*:

**assumes** *vsubstA a y x = a* **and** *xy: x  $\neq$  y*  
**shows** *freshA x a*

*<proof>*

**lemma** *freshA-iff-ex-vsubstA-idle*:

*freshA x a*  $\longleftrightarrow$   $(\exists y. y \neq x \wedge vsubstA a y x = a)$

*<proof>*

**lemma** *freshA-iff-all-vsubstA-idle*:

*freshA x a*  $\longleftrightarrow$   $(\forall y. y \neq x \longrightarrow vsubstA a y x = a)$

*<proof>*

**end**

## 2.2 Finitely supported renssets

**locale** *Renset-FinSupp = Renset vsubstA*

**for** *vsubstA* :: *'A*  $\Rightarrow$  *var*  $\Rightarrow$  *var*  $\Rightarrow$  *'A*

+

**assumes** *cofinite-freshA*:  $\bigwedge a. finite \{x. \neg freshA x a\}$

**begin**

**definition** *pickFreshSA* :: *var set*  $\Rightarrow$  *var list*  $\Rightarrow$  *'A list*  $\Rightarrow$  *var* **where**

*pickFreshSA X xs ds*  $\equiv$  *SOME z. z  $\notin$  X*  $\wedge$  *z  $\notin$  set xs*  $\wedge$   $(\forall a \in set ds. freshA z a)$

**lemma** *exists-freshA-set*:

**assumes** *finite X*

**shows**  $\exists z. z \notin X \wedge z \notin set xs \wedge (\forall a \in set ds. freshA z a)$

*<proof>*

**lemma** *exists-freshA*:

$\exists z. z \notin set xs \wedge (\forall a \in set ds. freshA z a)$

*<proof>*

**lemma** *pickFreshSA*:

**assumes** *finite X*

**shows**

*pickFreshSA X xs ds*  $\notin X \wedge$

*pickFreshSA*  $X$   $xs$   $ds \notin \text{set } xs \wedge$   
 $(\forall a \in \text{set } ds. \text{freshA } (\text{pickFreshSA } X \text{ } xs \text{ } ds) \ a)$   
 $\langle \text{proof} \rangle$

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

**definition** *pickFreshA*  $\equiv$  *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 renses

**locale** *Renset-Morphism* =

*A*: *Renset-FinSupp* *substA* + *B*: *Renset-FinSupp* *substB*

**for** *substA* :: '*A*  $\Rightarrow$  *var*  $\Rightarrow$  *var*  $\Rightarrow$  '*A* **and** *substB* :: '*B*  $\Rightarrow$  *var*  $\Rightarrow$  *var*  $\Rightarrow$  '*B*  
+

**fixes** *f* :: '*A*  $\Rightarrow$  '*B*

**assumes** *f-substA-substB*:  $\bigwedge a \ y \ z. f \ (\text{substA } a \ y \ z) = \text{substB } (f \ a) \ y \ z$

**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*  $\Rightarrow$  *var*  $\Rightarrow$  *var*  $\Rightarrow$  '*A*

**assumes**

*swapA-id*:  $\bigwedge a \ x. \text{swapA } a \ x \ x = a$

**and**

*swapA-invol*:  $\bigwedge a \ x \ y. \text{swapA } (\text{swapA } a \ x \ y) \ x \ y = a$

**and**

*swapA-cmp*:

```

 $\bigwedge x y a z1 z2. \text{swap}A (\text{swap}A a x y) z1 z2 =$ 
 $\text{swap}A (\text{swap}A a z1 z2) (\text{sw } x z1 z2) (\text{sw } y z1 z2)$ 
begin

```

```

definition freshA where freshA  $x a \equiv \text{finite } \{y. \text{swap}A a y x \neq a\}$ 

```

```

end

```

```

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

```

```

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:  $\bigwedge a z1 z2. f (\text{swap}A a z1 z2) = \text{swap}B (f a) z1 z2$ 

```

```

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 renssets 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
 $\text{swap}A a z1 z2 \equiv$ 
 $\text{let } yy = \text{pickFresh}A [z1, z2] [a] \text{ in}$ 
 $\text{vsubst}A (\text{vsubst}A (\text{vsubst}A a yy z1) z1 z2) z2 yy$ 

```

```

lemma swapA:
 $\exists yy. yy \notin \{z1, z2\} \wedge \text{fresh}A yy a \wedge$ 
 $\text{swap}A a z1 z2 = \text{vsubst}A (\text{vsubst}A (\text{vsubst}A a yy z1) z1 z2) z2 yy$ 
<proof>

```

**lemma** *swapA-id[simp]*:

*swapA a z z = a*

*<proof>*

**lemma** *vsubstA-two Ways*:

**assumes**  $uu \neq x \wedge uu \neq y \wedge \text{freshA } uu \ a \ vv \neq x \wedge vv \neq y \wedge \text{freshA } vv \ a$

**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**  $uu \neq x \wedge uu \neq y \wedge \text{freshA } uu \ a$

**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**  $\text{freshA } y \ a$

**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*:  $\text{freshA } x \ a \longleftrightarrow Sw.\text{freshA } x \ a$

*<proof>*

**end**

The statement that any finitely supported renet 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 rensset. 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-constructor-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:  $\bigwedge x y z. \{y,z\} \cap X = \{\} \implies$*   
*vsubstA (VrA x) y z = (if x = z then VrA y else VrA x)*  
**and**  
*vsubstA-ApA:  $\bigwedge y z t1 a1 t2 a2. \{y,z\} \cap X = \{\} \implies$*   
*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:  $\bigwedge t a z x y. \{x,y,z\} \cap X = \{\} \implies$*

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

**lemma** *LmA-cong*:

$\{u,z,x,x'\} \cap X = \{\} \implies$   
 $z \neq u \implies$   
 $z \neq x \implies z \neq x' \implies$   
 $vsubst\ (vsubst\ t\ u\ z)\ z\ x = vsubst\ (vsubst\ t'\ u\ z)\ z\ x' \implies$   
 $vsubstA\ (vsubstA\ a\ u\ z)\ z\ x = vsubstA\ (vsubstA\ a'\ u\ z)\ z\ x'$   
 $\implies LmA\ x\ (vsubst\ t\ u\ z)\ (vsubstA\ a\ u\ z) =$   
 $LmA\ x'\ (vsubst\ t'\ u\ z)\ (vsubstA\ a'\ u\ z)$   
 $\langle proof \rangle$

**lemma** *vsubstA-LmA-same*:

$\{x,y\} \cap X = \{\} \implies vsubstA\ (LmA\ x\ t\ a)\ y\ x = LmA\ x\ t\ a$   
 $\langle proof \rangle$

**lemma** *vsubstA-LmA-diff*:

$\{x,y,z\} \cap X = \{\} \implies$   
 $x \neq y \implies x \neq z \implies vsubstA\ (LmA\ x\ t\ a)\ y\ z = LmA\ x\ (vsubst\ t\ y\ z)\ (vsubstA\ a\ y\ z)$   
 $\langle proof \rangle$

**lemma** *freshA-2-vsubstA*:

**assumes**  $freshA\ z\ a\ freshA\ z\ a'$   
**shows**  $\exists u.\ u \notin X \wedge u \neq z \wedge vsubstA\ a\ u\ z = a \wedge vsubstA\ a'\ u\ z = a'$   
 $\langle proof \rangle$

**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**  $LmA\ x\ t\ a = LmA\ x'\ t'\ a'$   
 $\langle proof \rangle$

**lemma** *freshA-VrA*:  $z \notin X \implies z \neq x \implies freshA\ z\ (VrA\ x)$

$\langle proof \rangle$

**lemma** *freshA-ApA*:  $z \notin X \implies$

$fresh\ z\ t1 \implies freshA\ z\ a1 \implies$

$fresh\ z\ t2 \implies freshA\ z\ a2 \implies$   
 $freshA\ z\ (ApA\ t1\ a1\ t2\ a2)$   
 $\langle proof \rangle$

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

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

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

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

**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**  
 $Vr: R\ (Vr\ x)\ (VrA\ x)$   
 $|$   
 $Ap: R\ t1\ a1 \implies R\ t2\ a2 \implies R\ (Ap\ t1\ t2)\ (ApA\ t1\ a1\ t2\ a2)$   
 $|$   
 $Lm: R\ t\ a \implies x \notin X \implies R\ (Lm\ x\ t)\ (LmA\ x\ t\ a)$

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

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

**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*:  $f t = a \iff R t a$   
 ⟨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]*:  
 $x \notin X \implies f (Lm x t) = LmA x t (f t)$   
 ⟨proof⟩

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

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

*<proof>*

**theorem** *f-unique*:

**assumes** [*simp*]:  $\bigwedge x. g (Vr x) = VrA x$   
 $\bigwedge t1 t2. g (Ap t1 t2) = ApA t1 (g t1) t2 (g t2)$   
 $\bigwedge x t. x \notin X \implies g (Lm x t) = LmA x t (g t)$   
**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'*:  $\bigwedge x y z. \{y,z\} \cap X = \{\} \implies$

*vsubstA* (*VrA* *x*) *y z* = (*if*  $x = z$  *then* *VrA* *y* *else* *VrA* *x*)

**and**

*vsubstA-ApA'*:  $\bigwedge y z a1 a2. \{y,z\} \cap X = \{\} \implies$

*vsubstA* (*ApA* *a1* *a2*) *y z* =

*ApA* (*vsubstA* *a1* *y z*)

(*vsubstA* *a2* *y z*)

**and**

*vsubstA-LmA'*:  $\bigwedge a z x y. \{x,y,z\} \cap X = \{\} \implies$

$x \neq y \implies$

*vsubstA* (*LmA* *x* *a*) *y z* = (*if*  $x = z$  *then* *LmA* *x* *a* *else* *LmA* *x* (*vsubstA* *a* *y z*))

**and**

*LmA-rename'*:  $\bigwedge x y z a. \{x,y,z\} \cap X = \{\} \implies$

$z \neq y \implies LmA x (vsubstA a z y) = LmA y (vsubstA (vsubstA a z y) y x)$

**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''*:  $\bigwedge x y z.$

*vsubstA (VrA x) y z* = (*if*  $x = z$  *then* *VrA y* *else* *VrA x*)

**and**

*vsubstA-ApA''*:  $\bigwedge y z a1 a2.$

*vsubstA (ApA a1 a2) y z* =

*ApA (vsubstA a1 y z)*

(*vsubstA a2 y z*)

**and**

*vsubstA-LmA''*:  $\bigwedge a z x y.$

$x \neq y \implies$

*vsubstA (LmA x a) y z* = (*if*  $x = z$  *then* *LmA x a* *else* *LmA x (vsubstA a y z)*)

**and**

*LmA-rename''*:  $\bigwedge x y z a.$

$z \neq y \implies$  *LmA x (vsubstA a z y)* = *LmA y (vsubstA (vsubstA a z y) y x)*

**begin**

**sublocale** *BCE-Renset* **where**  $X = \{\}$

*<proof>*

**lemma** *triv*:  $x \notin \{\}$  *<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 rerset algebraic theory, including initiality and recursion principle, but focusing on term-for-variable rather than variable-for-variable substitution. Instead of rersets, 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]*:  $\bigwedge x a. \text{substA } a \text{ (VrA } x) x = a$   
**and** *substA-idem*:  $\bigwedge x b1 b2 a. u \neq x \implies$   
*let*  $b1' = \text{substA } b1 \text{ (VrA } u) x$  *in*  $\text{substA } (\text{substA } a \ b1' \ x) \ b2 \ x = \text{substA } a \ b1' \ x$   
**and**  
*substA-chain*:  $\bigwedge u x1 x2 b3 a. u \neq x2 \implies$   
 $\text{substA } (\text{substA } (\text{substA } a \text{ (VrA } u) \ x2) \text{ (VrA } x2) \ x1) \ b3 \ x2 =$   
 $\text{substA } (\text{substA } a \text{ (VrA } u) \ x2) \ b3 \ x1$   
**and**  
*substA-commute-diff*:  
 $\bigwedge x y a e f. x \neq y \implies u \neq y \implies v \neq x \implies$   
*let*  $e' = \text{substA } e \text{ (VrA } u) \ y; f' = \text{substA } f \text{ (VrA } v) \ x$  *in*  
 $\text{substA } (\text{substA } a \ e' \ x) \ f' \ y = \text{substA } (\text{substA } a \ f' \ y) \ e' \ x$   
**and**  
*substA-VrA*:  $\bigwedge x a z. \text{substA } (\text{VrA } x) \ a \ z = (\text{if } x = z \text{ then } a \text{ else } \text{VrA } x)$   
**begin**

**lemma** *substA-idem-var[simp]*:  
 $y1 \neq x \implies \text{substA } (\text{substA } a \text{ (VrA } y1) \ x) \text{ (VrA } y2) \ x = \text{substA } a \text{ (VrA } y1) \ x$   
 $\langle \text{proof} \rangle$

**lemma** *substA-commute-diff-var*:  
 $x \neq v \implies y \neq u \implies x \neq y \implies$   
 $\text{substA } (\text{substA } a \text{ (VrA } u) \ x) \text{ (VrA } v) \ y = \text{substA } (\text{substA } a \text{ (VrA } v) \ y) \text{ (VrA } u) \ x$   
 $\langle \text{proof} \rangle$

**end**

Any substitutive set is in particular a rerset:

**sublocale** *Substitutive-Set* < *Rerset* **where**  
 $v\text{substA} = \lambda a x. \text{substA } a \text{ (VrA } x) \langle \text{proof} \rangle$

**interpretation** *STerm*: *Substitutive-Set* **where**  $\text{substA} = \text{subst}$  **and**  $\text{VrA} = \text{Vr}$   
 $\langle \text{proof} \rangle$

## 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*  $\implies$  *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*  $\implies$   
*z*  $\neq$  *x*  $\implies$  *z*  $\neq$  *x'*  $\implies$   
*substA* (*substA* *a* (*VrA* *u*) *z*) (*VrA* *z*) *x* = *substA* (*substA* *a'* (*VrA* *u*) *z*) (*VrA* *z*)  
*x'*  
 $\implies$  *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*  $\implies$  *x*  $\neq$  *z*  $\implies$  *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**  $\exists u. u \neq z \wedge \text{substA } a \text{ (VrA } u) z = a \wedge \text{substA } a' \text{ (VrA } u) z = a'$   
 <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*:  $z \neq x \implies \text{freshA } z \text{ (VrA } x)$   
*<proof>*

**lemma** *freshA-ApA*:  $\bigwedge z a1 a2. \text{freshA } z a1 \implies \text{freshA } z a2 \implies \text{freshA } z \text{ (ApA } a1 a2)$   
*<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 rerset:

**sublocale** *CE-Substitutive-Set < CE-Rerset*  
**where** *vsubstA =  $\lambda a x. \text{substA } a \text{ (VrA } x)$*   
*<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 rerset:

**interpretation** *Var: Rerset* **where**  $vsubstA = vss$   
 $\langle proof \rangle$

Terms form a rerset:

**interpretation** *Term: Rerset* **where**  $vsubstA = \lambda t x. vsubst t x$   
 $\langle proof \rangle$

... and a CE rerset:

**interpretation** *Term: CE-Rerset*  
**where**  $vsubstA = \lambda t x. subst t (Vr x)$   
**and**  $VrA = Vr$  **and**  $ApA = Ap$  **and**  $LmA = Lm$   
 $\langle proof \rangle$

## 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-Rerset*

**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**  $LmA = \lambda x i \xi. lm (\lambda d. i (\xi(x:=d)))$   
 $\langle proof \rangle$

**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 rerset yields a rerset – 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'*:  $Fmap (g o f) k = Fmap g (Fmap f k)$

```

    <proof>

end

locale Renset-plus-Local-Functor =
  Renset vsubstA + Local-Functor Fmap
  for vsubstA :: 'A ⇒ var ⇒ var ⇒ 'A
    and Fmap :: ('A ⇒ 'A) ⇒ 'FA ⇒ '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 occurrences of a variable in a term via renaming-based recursion

```

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

```



```

    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  $VrA = \lambda y. get\ \varrho\ y$ 
and  $ApA = Ap$ 
and  $LmA = Lm$ 
and  $X = X$ 
 $\langle proof \rangle$ 

```

```

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  $lm = \lambda d. d\ (1::nat)$   $\langle proof \rangle$ 

```

```

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

```

```

definition  $cbv :: trm \Rightarrow nat$  where
 $cbv\ t \equiv cbvs.f\ t\ (\lambda-. 0)$ 

```

## 7.10 Testing eta-reducibility via renaming-based recursion

```

interpretation  $canEta': Sem-Int$  where  $ap = (\wedge)$  and  $lm = \lambda d. d\ True$   $\langle proof \rangle$ 

```

```

lemmas  $canEta'.f-clauses = canEta'.f-Vr\ canEta'.f-Ap\ canEta'.f-Lm\ canEta'.f-subst$ 
 $canEta'.f-unique$ 

```

```

definition  $canEta :: trm \Rightarrow bool$  where
 $canEta\ t \equiv \exists x\ s. t = Lm\ x\ (Ap\ s\ (Vr\ x)) \wedge canEta'.f\ s\ ((\lambda-. True)(x:=False))$ 

```

```

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.