Nominal Unification

Guilherme Borges Brandão[1] Thomas Ammer[2]
Daniele Nantes Sobrinho[1] Mauricio Ayala-Rinción[1]
Christian Urban[2] Maribel Fernández[2]
Mohammad Abdulaziz[2]

[1] Universidade de Brasília, Brasília D.F., Brazil
[2]King's College London, London, U.K.

April 24, 2026

Abstract

This is an improved and updated formalisation of the nominal unification algorithm. Nominal unification is a generalisation of the classic first-order unification to the practically important case of equations between terms involving binding operations. A simple, possibly capturing substitution of terms for variables solves such equations if it makes the equated terms alpha-equivalent, i.e. equal up to renaming bound names. While nominal unification can be seen as equivalent to Miller's higher-order pattern unification (unification problems can be inter-coded), nominal unification has the advantage of involving only first-order syntax. This means in the presented formalisation we only need to reason about standard inductive datatypes and first-order operations. The main improvement of this development is a much simpler proof for establishing the equivalence relation properties for an inductive definition characterising when two terms are alpha-equivalent. In the original literature, this involved a clunky mutual induction over the size of terms involving three properties. Here we can separate the proofs and use just structural inductions. This results in a much smoother formalisation of the nominal unification algorithm.

Contents

1 Introduction

This is an improved and updated formalisation of the nominal unification algorithm given in [5, 6]. The formalisation presented here is improved according to the Rocha Oliveira et al. approach formalised in PVS [2, 4], where the properties of symmetry, transitivity, and equivariance of the α-equivalence relation are proved separately. This makes the proof and formalisation more convenient. The same approach was taken by de Carvalho et al. in the Coq formalisation of nominal α-unification and nominal C-unification [1, 3]. In the PVS formalisation, nominal unification is presented as a functional recursive algorithm, proved sound and complete, from which executable code in Lisp can be extracted. In the Coq formalisation, non-deterministic rule-based nominal unification procedures were presented, following the paper by Urban et al, which were proved sound and complete; further, recursive definitions were presented, which were proved equivalent to the inductive procedures, and from which executable code was extracted.

2 Swappings of Pairs of Atoms

fun *swapa* :: $(string \times string) \Rightarrow string \Rightarrow string$
  where
  swapa (b,c) $a = (if\ b = a\ then\ c\ else\ (if\ c = a\ then\ b\ else\ a))$

fun *swapas*:: $(string \times string)\ list \Rightarrow string \Rightarrow string$
  where
  swapas $[]$ $a = a$
| *swapas* $(x\#pi)$ $a = swapa\ x\ (swapas\ pi\ a)$

lemma *swapas-aa*[*simp*]:
shows *swapas* [(*a*,*a*)] *b* = *b*
⟨*proof*⟩

lemma *swapas-ab-ba*:
shows *swapas* [(*a*,*b*)] = *swapas* [(*b*,*a*)]
⟨*proof*⟩

lemma *swapas-append*:
shows *swapas* (*pi1*@*pi2*) *a* = *swapas pi1* (*swapas pi2 a*)
⟨*proof*⟩

lemma *swapas-inv* [*simp*]:
shows *swapas* (*rev pi*) (*swapas pi a*) = *a*
⟨*proof*⟩

lemma *swapas-rev-pi-a*:
assumes *swapas pi a* = *b*
shows *swapas* (*rev pi*) *b* = *a*
⟨*proof*⟩

lemma *swapas-rev-swapas-id* [*simp*]:
shows *swapas pi* (*swapas* (*rev pi*) *a*) = *a*
⟨*proof*⟩

lemma *swapas-rev-pi-b*:
assumes *swapas* (*rev pi*) *a* = *b*
shows *swapas pi b* = *a*
⟨*proof*⟩

lemma *swapas-comm*:
shows (*swapas* (*pi*@[(*a*,*b*)]) *c*) = (*swapas* ([(*swapas pi a*,*swapas pi b*)]@*pi*) *c*)
⟨*proof*⟩

3 Terms

Definions for terms, occurs check and the notion of subterms.

datatype *trm* = *Abst string trm*
| *Susp* (*string* × *string*) *list string*
| *Unit*
| *Atom string*
| *Paar trm trm*
| *Func string trm*

The swapping operation on terms.

fun *swap* :: (*string* × *string*) *list* ⇒ *trm* ⇒ *trm*
where

```
  swap  pi (Unit)         = Unit
| swap  pi (Atom a)       = Atom (swapas pi a)
| swap  pi (Susp pi' X)   = Susp (pi @ pi') X
| swap  pi (Abst a t)     = Abst (swapas pi a) (swap pi t)
| swap  pi (Paar t1 t2)   = Paar (swap pi t1) (swap pi t2)
| swap  pi (Func F t)     = Func F (swap pi t)
```

lemma *swap-id* [*simp*]:
  shows *swap* [] *t* = *t*
  ⟨*proof*⟩

lemma *swap-append*:
  shows *swap* (*pi1*@*pi2*) *t* = *swap pi1* (*swap pi2 t*)
  ⟨*proof*⟩

lemma *swap-empty*:
  assumes *swap pi t* = *Susp* [] *X*
  shows *pi* = []
  ⟨*proof*⟩

The depth of terms used as measure.

```
fun depth :: trm ⇒ nat
  where
  depth (Unit)       = 0
| depth (Atom a)     = 0
| depth (Susp pi X) = 0
| depth (Abst a t)  = 1 + depth t
| depth (Func F t)  = 1 + depth t
| depth (Paar t t') = 1 + (max (depth t) (depth t'))
```

lemma
  Suc-max-left: *Suc*(*max x y*) = *z* ⟶ *x* < *z* **and**
  Suc-max-right: *Suc*(*max x y*) = *z* ⟶ *y* < *z*
⟨*proof*⟩

lemma *swap-depth* [*simp*]:
  shows *depth* (*swap pi t*) = *depth t*
  ⟨*proof*⟩

The occurs predicate and variables inside terms.

```
fun occurs :: string ⇒ trm ⇒ bool
  where
  occurs X (Unit)       = False
| occurs X (Atom a)     = False
| occurs X (Susp pi Y)  = (if X = Y then True else False)
| occurs X (Abst a t)   = occurs X t
| occurs X (Paar t1 t2) = (if (occurs X t1) then True else (occurs X t2))
| occurs X (Func F t)   = occurs X t
```

fun *vars-trm* :: *trm* ⇒ *string set*
  where
  vars-trm (*Unit*) = {}
| *vars-trm* (*Atom a*) = {}
| *vars-trm* (*Susp pi X*) = {*X*}
| *vars-trm* (*Paar t1 t2*) = (*vars-trm t1*)∪(*vars-trm t2*)
| *vars-trm* (*Abst a t*) = *vars-trm t*
| *vars-trm* (*Func F t*) = *vars-trm t*

lemma *vars-swap*:
  shows *vars-trm* (*swap pi t*) = *vars-trm t*
  ⟨*proof*⟩

Subterms and proper subterms.

fun *sub-trms* :: *trm* ⇒ *trm set*
  where
  sub-trms (*Unit*) = {*Unit*}
| *sub-trms* (*Atom a*) = {*Atom a*}
| *sub-trms* (*Susp pi Y*) = {*Susp pi Y*}
| *sub-trms* (*Abst a t*) = {*Abst a t*} ∪ *sub-trms t*
| *sub-trms* (*Paar t1 t2*) = {*Paar t1 t2*} ∪ *sub-trms t1* ∪ *sub-trms t2*
| *sub-trms* (*Func F t*) = {*Func F t*} ∪ *sub-trms t*

fun *psub-trms* :: *trm* ⇒ *trm set*
  where
  psub-trms (*Unit*) = {}
| *psub-trms* (*Atom a*) = {}
| *psub-trms* (*Susp pi X*) = {}
| *psub-trms* (*Paar t1 t2*) = *sub-trms t1* ∪ *sub-trms t2*
| *psub-trms* (*Abst a t*) = *sub-trms t*
| *psub-trms* (*Func F t*) = *sub-trms t*

lemma *psub-sub-trms*:
  assumes *t1* ∈ *psub-trms t2*
  shows *t1* ∈ *sub-trms t2*
  ⟨*proof*⟩

lemma *t-sub-trms-t*:
  shows *t* ∈ *sub-trms t*
  ⟨*proof*⟩

lemma *abst-psub-trms*:
  assumes *Abst a t1* ∈ *sub-trms t2*
  shows *t1* ∈ *psub-trms t2*
  ⟨*proof*⟩

lemma *func-psub-trms*:
  assumes *Func F t1* ∈ *sub-trms t2*
  shows *t1* ∈ *psub-trms t2*

⟨*proof*⟩

lemma *paar-psub-trms*:
assumes *Paar t1 t2* ∈ *sub-trms t3*
shows *t1* ∈ *psub-trms t3* **and** *t2* ∈ *psub-trms t3*
⟨*proof*⟩

lemma *t-not-in-psub-trms-t*:
shows *t* ∉ *psub-trms t*
⟨*proof*⟩

lemma *if-sub-not-eq-then-psub*:
assumes *t1* ∈ *sub-trms t2 t1* ≠ *t2*
shows *t1* ∈ *psub-trms t2*
⟨*proof*⟩

lemma *depth-psub-trms*:
assumes *t1* ∈ *psub-trms t2*
shows *depth t1* < *depth t2*
⟨*proof*⟩

4 Facts about Atoms Occurring in Swappings

fun *atms* :: (*string* × *string*) *list* ⇒ *string set* **where**
atms [] = {} |
atms (*x*#*xs*) = ((*atms xs*) ∪ {*fst*(*x*),*snd*(*x*)})

lemma *atms-append*[*simp*]:
shows *atms* (*xs*@*ys*) = *atms xs* ∪ *atms ys*
⟨*proof*⟩

lemma *atms-rev*[*simp*]:
shows *atms* (*rev pi*) = *atms pi*
⟨*proof*⟩

lemma *a-not-in-atms*:
assumes *a* ∉ *atms pi*
shows *a* = *swapas pi a*
⟨*proof*⟩

lemma *swapas-pi-ineq-a*:
assumes *swapas pi a* ≠ *a*
shows *a* ∈ *atms pi*
⟨*proof*⟩

lemma *a-ineq-swapas-pi*:
assumes *a* ≠ *swapas pi a*
shows *a* ∈ *atms pi*
⟨*proof*⟩

lemma *swapas-pi-in-atms*:
assumes $a \in atms\ pi$
shows*swapas pi a* $\in$ *atms pi*
⟨*proof*⟩

5 Disagreement Sets

definition *ds* :: (*string* × *string*) *list* ⇒ (*string* × *string*) *list* ⇒ *string set* **where**
ds-def: *ds xs ys* ≡ { *a* . *a* ∈ (*atms xs* ∪ *atms ys*) ∧ (*swapas xs a* ≠ *swapas ys a*) }

lemma *ds-elem*:
assumes *swapas pi a*≠*a*
shows *a*∈*ds* [] *pi*
⟨*proof*⟩

corollary *ds-elem-cp*:
assumes *a* ∉ *ds* [] *pi*
shows *swapas pi a* = *a*
⟨*proof*⟩

lemma *elem-ds*:
assumes *a*∈*ds* [] *pi*
shows *a*≠*swapas pi a*
⟨*proof*⟩

lemma *ds-sym*:
shows *ds pi1 pi2* = *ds pi2 pi1*
⟨*proof*⟩

lemma *ds-trans*:
assumes *c* ∈ *ds pi1 pi3*
shows *c* ∈ *ds pi1 pi2* ∨ *c* ∈ *ds pi2 pi3*
⟨*proof*⟩

lemma *ds-cancel-pi-left*:
assumes *c*∈ *ds* (*pi1*@*pi*) (*pi2*@*pi*)
shows *swapas pi c* ∈ *ds pi1 pi2*
⟨*proof*⟩

lemma *ds-cancel-pi-right*:
assumes *swapas pi c* ∈ *ds pi1 pi2*
shows *c* ∈ *ds* (*pi1*@*pi*) (*pi2*@*pi*)
⟨*proof*⟩

lemma *ds-equality*:
shows (*ds* [] *pi*)−{*a*,*swapas pi a*} = (*ds* [] ((*a*,*swapas pi a*)#*pi*))−{*swapas pi a*}
⟨*proof*⟩

lemma *ds-7*:
assumes $b \neq swapas\ pi\ b\ a \in ds\ []\ ((b,swapas\ pi\ b)\#pi)$
shows $a \in ds\ []\ pi$
⟨*proof*⟩

lemma *ds-cancel-pi-front*:
shows $ds\ (pi@pi1)\ (pi@pi2) = ds\ pi1\ pi2$
⟨*proof*⟩

lemma *ds-rev-pi-pi*:
shows $ds\ ((rev\ pi1)@pi1)\ pi2 = ds\ []\ pi2$
⟨*proof*⟩

lemma *ds-rev*:
shows $ds\ []\ ((rev\ pi1)@pi2) = ds\ pi1\ pi2$
⟨*proof*⟩

lemma *ds-acabbc*:
assumes $a \neq b\ b \neq c\ c \neq a$
shows $ds\ [(a,\ b),\ (b,\ c)]\ [(a,\ c)] = \{a,\ b\}$
⟨*proof*⟩

lemma *ds-baab*:
assumes $a \neq b$
shows $ds\ [(b,a)]\ [(a,\ b)] = \{\}$
⟨*proof*⟩

lemma *ds-baab-id*:
assumes $a \neq b$
shows $ds\ ([(b,a)]@[(a,\ b)])\ [] = \{\}$
⟨*proof*⟩

lemma *ds-abab*:
shows $ds\ []\ [(a,\ b),\ (a,\ b)] = \{\}$
⟨*proof*⟩

lemma *ds-comm*:
shows $ds\ (pi\ @\ [(a,b)])\ ([(swapas\ pi\ a,\ swapas\ pi\ b)]\ @\ pi) = \{\}$
⟨*proof*⟩

lemma *ds-rev-pi-id*:
shows $ds\ (rev\ pi\ @\ pi)\ [] = \{\}$
⟨*proof*⟩

lemma *ds-pi-rev-id*:
shows $ds\ (pi\ @\ rev\ pi)\ [] = \{\}$
⟨*proof*⟩

lemma *ds-swapas-eq*:
assumes *ds pi1 pi2* = {}
shows *swapas pi1 a* = *swapas pi2 a*
⟨*proof*⟩

Disagreement sets as lists.

fun *flatten* :: (*string* × *string*)*list* ⇒ *string list* **where**
flatten [] = [] |
flatten (*x*#*xs*) = (*fst x*)#(*snd x*)#(*flatten xs*)

definition *ds-list* :: (*string* × *string*)*list* ⇒ (*string* × *string*)*list* ⇒ *string list* **where**
ds-list-def: *ds-list pi1 pi2* ≡ *remdups* ([*x*. *x* <− (*flatten* (*pi1*@*pi2*)), *x*∈*ds pi1 pi2*])

lemma *set-flatten-eq-atms*:
shows *set* (*flatten pi*) = *atms pi*
⟨*proof*⟩

lemma *ds-list-equ-ds*:
set (*ds-list pi1 pi2*) = *ds pi1 pi2*
⟨*proof*⟩

6 Freshness

Defines the freshness relation and shows facts about its behaviour under swappings.

type-synonym *fresh-envs* = (*string* × *string*) *set*

inductive *fresh* :: *fresh-envs* ⇒ *string* ⇒ *trm* ⇒ *bool* (- ⊢ - ♯ - [*80*,*80*,*80*] *80*)
where
fresh-abst-ab[*intro*!]: ⟦*nabla* ⊢ *a* ♯ *t*; *a*≠*b*⟧ ⟹ *nabla* ⊢ *a* ♯ *Abst b t* |
fresh-abst-aa[*intro*!]: *nabla* ⊢ *a* ♯ *Abst a t* |
fresh-unit[*intro*!]: *nabla* ⊢ *a* ♯ *Unit* |
fresh-atom[*intro*!]: *a* ≠ *b* ⟹ *nabla* ⊢ *a* ♯ *Atom b* |
fresh-susp[*intro*!]: (*swapas* (*rev pi*) *a*,*X*) ∈ *nabla* ⟹ *nabla* ⊢ *a* ♯ *Susp pi X* |
fresh-paar[*intro*!]: ⟦*nabla* ⊢ *a* ♯ *t1*; *nabla* ⊢ *a* ♯ *t2*⟧ ⟹ *nabla* ⊢ *a* ♯ *Paar t1 t2* |
fresh-func[*intro*!]: *nabla* ⊢ *a* ♯ *t* ⟹ *nabla* ⊢ *a* ♯ *Func F t*

inductive-cases *Fresh-elims*:
nabla ⊢ *a* ♯ *Abst b t*
nabla ⊢ *a* ♯ *Unit*
nabla ⊢ *a* ♯ *Atom b*
nabla ⊢ *a* ♯ *Susp pi X*
nabla ⊢ *a* ♯ *Paar t1 t2*
nabla ⊢ *a* ♯ *Func F t*

lemma *fresh-swap-eqvt*:

assumes *nabla* ⊢ *a* ♯ *t*
shows *nabla* ⊢ *swapas pi a* ♯ *swap pi t*
⟨*proof*⟩

lemma *fresh-swap-left*:
assumes *nabla* ⊢ *a* ♯ *swap pi t*
shows *nabla* ⊢ *swapas* (*rev pi*) *a* ♯ *t*
⟨*proof*⟩

lemma *fresh-swap-right*:
assumes *nabla* ⊢ *swapas* (*rev pi*) *a* ♯ *t*
shows *nabla* ⊢ *a* ♯ *swap pi t*
⟨*proof*⟩

lemma *fresh-weak*:
assumes *nabla1* ⊢ *a* ♯ *t*
shows (*nabla1* ∪ *nabla2*) ⊢ *a* ♯ *t*
⟨*proof*⟩

lemma *ds-empty-fresh-1*:
assumes *ds pi1 pi2* = {}
shows *nabla* ⊢ *swapas pi1 a* ♯ *t* ⟹ *nabla* ⊢ *swapas pi2 a* ♯ *t*
⟨*proof*⟩

lemma *ds-empty-fresh-2*:
assumes *ds pi1 pi2* = {}
shows *nabla* ⊢ *a* ♯ *swap pi1 t* ⟹ *nabla* ⊢ *a* ♯ *swap pi2 t*
⟨*proof*⟩

7 Pre-Equality

Defines the relation which captures the notion of alpha-equivalence (on open terms) and proves this relation is an equivalence relation.

inductive *equ* :: *fresh-envs* ⇒ *trm* ⇒ *trm* ⇒ *bool* (- ⊢ - ≈ - [*80*,*80*,*80*] *80*) **where**
equ-abst-ab[*intro*!]: ⟦*a*≠*b*;(*nabla* ⊢ *a* ♯ *t2*);(*nabla* ⊢ *t1* ≈ (*swap* [(*a*,*b*)] *t2*))⟧
⟹ (*nabla* ⊢ *Abst a t1* ≈ *Abst b t2*) |
equ-abst-aa[*intro*!]: (*nabla* ⊢ *t1* ≈ *t2*) ⟹ (*nabla* ⊢ *Abst a t1* ≈ *Abst a t2*) |
equ-unit[*intro*!]: (*nabla* ⊢ *Unit* ≈ *Unit*) |
equ-atom[*intro*!]: *a*=*b*⟹*nabla* ⊢ *Atom a* ≈ *Atom b* |
equ-susp[*intro*!]: (∀ *c* ∈ *ds pi1 pi2*. (*c*,*X*) ∈ *nabla*) ⟹ (*nabla* ⊢ *Susp pi1 X* ≈ *Susp pi2 X*) |
equ-paar[*intro*!]: ⟦(*nabla* ⊢ *t1*≈*t2*);(*nabla* ⊢ *s1*≈*s2*)⟧ ⟹ (*nabla* ⊢ *Paar t1 s1* ≈ *Paar t2 s2*) |
equ-func[*intro*!]: (*nabla* ⊢ *t1* ≈ *t2*) ⟹ (*nabla* ⊢ *Func F t1* ≈ *Func F t2*)

inductive-cases *Equ-elims*:
nabla ⊢ *Atom a* ≈ *Atom b*

$nabla \vdash Unit \approx Unit$
$nabla \vdash Susp\ pi1\ X \approx Susp\ pi2\ X$
$nabla \vdash Paar\ s1\ t1 \approx Paar\ s2\ t2$
$nabla \vdash Func\ F\ t1 \approx Func\ F\ t2$
$nabla \vdash Abst\ a\ t1 \approx Abst\ a\ t2$
$nabla \vdash Abst\ a\ t1 \approx Abst\ b\ t2$

lemma *equ-depth*:
assumes $nabla \vdash t1 \approx t2$
shows $depth\ t1 = depth\ t2$
⟨*proof*⟩

lemma *rev-pi-pi-equ*:
shows $(nabla \vdash swap\ (rev\ pi)\ (swap\ pi\ t) \approx t)$
⟨*proof*⟩

lemma *equ-pi-right*:
assumes $\forall a \in ds\ []\ pi.\ nabla \vdash a \sharp t$
shows $nabla \vdash t \approx swap\ pi\ t$
⟨*proof*⟩

lemma *pi-comm*:
shows $nabla \vdash (swap\ (pi\ @\ [(a,b)])\ t) \approx (swap\ ([(swapas\ pi\ a,\ swapas\ pi\ b)]\ @\ pi)\ t)$
⟨*proof*⟩

lemma *l3-jud*:
assumes $(nabla \vdash t1 \approx t2)$
shows $(nabla \vdash a \sharp t1) \longrightarrow (nabla \vdash a \sharp t2)$
⟨*proof*⟩

lemma *ds-empty-equiv-1*:
assumes $ds\ pi1\ pi2 = \{\}$
shows $nabla \vdash swap\ pi1\ t1 \approx t2 \Longrightarrow nabla \vdash swap\ pi2\ t1 \approx t2$
⟨*proof*⟩

lemma *ds-empty-equiv-2*:
assumes $ds\ pi1\ pi2 = \{\}$
shows $nabla \vdash t1 \approx swap\ pi1\ t2 \Longrightarrow nabla \vdash t1 \approx swap\ pi2\ t2$
⟨*proof*⟩

lemma *equ-equivariance*:
assumes $nabla \vdash t1 \approx t2$
shows $nabla \vdash swap\ pi\ t1 \approx swap\ pi\ t2$
⟨*proof*⟩

lemma *swap-inv-side*:
shows $nabla \vdash swap\ pi\ t1 \approx t2 = nabla \vdash t1 \approx swap\ (rev\ pi)\ t2$
⟨*proof*⟩

lemma *equ-swap-abba*:
assumes $n = depth\ t1$
shows $nabla \vdash swap\ [(a,b)]\ t1 \approx t2 \Longrightarrow nabla \vdash t1 \approx swap\ [(b,a)]\ t2$
⟨*proof*⟩

lemma *equ-equiv-pi*:
assumes $\forall\, a \in ds\ pi1\ pi2.\ nabla \vdash a \sharp t$
shows $nabla \vdash swap\ pi1\ t \approx swap\ pi2\ t$
⟨*proof*⟩

lemma *equ-symm*:
shows $(nabla \vdash t1 \approx t2) \Longrightarrow (nabla \vdash t2 \approx t1)$
⟨*proof*⟩

lemma *equ-trans*:
assumes $nabla \vdash t1 \approx t2\ nabla \vdash t2 \approx t3$
shows $nabla \vdash t1 \approx t3$
⟨*proof*⟩

lemma *pi-right-equ-help*:
assumes $(n = depth\ t)$
shows $nabla \vdash t \approx swap\ pi\ t \Longrightarrow \forall\ a \in ds\ []\ pi.\ nabla \vdash a \sharp t$
⟨*proof*⟩

8 Equality

Proves various facts about the equivalence relation.

lemma *equ-refl*:
shows $nabla \vdash t \approx t$
⟨*proof*⟩

lemma *equ-dec-pi*:
assumes $nabla \vdash swap\ pi\ t1 \approx swap\ pi\ t2$
shows $nabla \vdash t1 \approx t2$
⟨*proof*⟩

lemma *equ-involutive-left*:
shows $nabla \vdash swap\ (rev\ pi)\ (swap\ pi\ t1) \approx t2 = nabla \vdash t1 \approx t2$
⟨*proof*⟩

lemma *equ-pi-to-left*:
shows $nabla \vdash swap\ (rev\ pi)\ t1 \approx t2 = nabla \vdash t1 \approx swap\ pi\ t2$
⟨*proof*⟩

lemma *equ-pi-to-right*:
shows $nabla \vdash t1 \approx swap\ (rev\ pi)\ t2 = nabla \vdash swap\ pi\ t1 \approx t2$

⟨*proof*⟩

lemma *equ-involutive-right*:
shows *nabla* ⊢ *t1* ≈ *swap* (*rev pi*) (*swap pi t2*) = *nabla* ⊢ *t1* ≈ *t2*
⟨*proof*⟩

lemma *equ-pi1-pi2-add*:
assumes ∀ *a*∈ *ds pi1 pi2*. *nabla*⊢*a*♯*t*
shows *nabla*⊢*swap pi1 t* ≈ *swap pi2 t*
⟨*proof*⟩

lemma *pi-right-equ*:
assumes *nabla* ⊢ *t* ≈ *swap pi t*
shows ∀ *a*∈ *ds* [] *pi*. *nabla* ⊢ *a* ♯ *t*
⟨*proof*⟩

lemma *equ-pi1-pi2-dec*:
assumes *nabla* ⊢ *swap pi1 t* ≈ *swap pi2 t*
shows ∀ *a* ∈ *ds pi1 pi2*. *nabla*⊢*a* ♯ *t*
⟨*proof*⟩

lemma *equ-weak*:
assumes *nabla1* ⊢ *t1* ≈ *t2*
shows (*nabla1* ∪ *nabla2*) ⊢ *t1* ≈ *t2*
⟨*proof*⟩

No term can be equal to one of its proper subterm.

lemma *psub-trm-not-equ*:
shows ∀ *t2* ∈ *psub-trms t1*. (¬(∃ *pi*. (*nabla* ⊢ *t1* ≈ *swap pi t2*)))
⟨*proof*⟩

9 Substitutions

Defines substitutions and composition of substitutions, and establishes some facts of substitution and the equivalence relation.

type-synonym *substs* = (*string* × *trm*) *list*

fun *look-up* :: *string* ⇒ *substs* ⇒ *trm* **where**
look-up X [] = *Susp* [] *X* |
look-up X (*x*#*xs*) = (*if* (*X* = *fst x*) *then* (*snd x*) *else look-up X xs*)

fun *subst* :: *substs* ⇒ *trm* ⇒ *trm* **where**
subst-unit: *subst s* (*Unit*) = *Unit* |
subst-susp: *subst s* (*Susp pi X*) = *swap pi* (*look-up X s*) |
subst-atom: *subst s* (*Atom a*) = *Atom a* |
subst-abst: *subst s* (*Abst a t*) = *Abst a* (*subst s t*) |
subst-paar: *subst s* (*Paar t1 t2*) = *Paar* (*subst s t1*) (*subst s t2*) |
subst-func: *subst s* (*Func F t*) = *Func F* (*subst s t*)

declare *subst-susp* [*simp del*]

The notion of composition of substitutions (adapted from Martin Coen's Classical Computational Logic).

fun *alist-rec* :: *substs* ⇒ *substs* ⇒ (*string*⇒*trm*⇒*substs*⇒*substs*⇒*substs*) ⇒ *substs*
where
alist-rec [] *c d* = *c* |
alist-rec (*p* # *al*) *c d* = *d* (*fst p*) (*snd p*) *al* (*alist-rec al c d*)

definition *comp* :: *substs* ⇒ *substs* ⇒ *substs* (**infixl** ‹•› *81*) **where**
s1 • *s2* ≡ *alist-rec s2 s1* (λ *x y xs g*. (*x*,*subst s1 y*) # *g*)

The domain of substitutions.

definition *domn* :: (*trm* ⇒ *trm*) ⇒ *string set* **where**
domn s ≡ {*X*. (*s* (*Susp* [] *X*)) ≠ (*Susp* [] *X*)}

substitutions extend freshness environments.

definition *ext-subst* :: *fresh-envs* ⇒ (*trm* ⇒ *trm*) ⇒ *fresh-envs* ⇒ *bool* (- ⊨ - - [*80*,*80*,*80*] *80*) **where**
nabla1 ⊨ *s nabla2* ≡ (∀(*a*,*X*)∈*nabla2*. *nabla1*⊢*a*♯ *s* (*Susp* [] *X*))

Alpha-equality for substitutions

definition *subst-equ* :: *fresh-envs* ⇒ (*trm*⇒*trm*) ⇒ (*trm*⇒*trm*) ⇒ *bool* (- ⊨ - ≈ - [*80*,*80*,*80*] *80*)
where
nabla ⊨ *s1* ≈ *s2* ≡ ∀*X*∈(*domn s1*∪*domn s2*). (*nabla* ⊢ *s1* (*Susp* [] *X*) ≈ *s2* (*Susp* [] *X*))

lemma *subst-equ-symm*:
assumes *nabla* ⊨ *s1* ≈ *s2*
shows *nabla* ⊨ *s2* ≈ *s1*
⟨*proof*⟩

lemma *subst-equ-refl*:
shows *nabla* ⊨ *s* ≈ *s*
⟨*proof*⟩

lemma *not-in-domn*:
assumes *X* ∉ (*domn s*)
shows *s* (*Susp* [] *X*) = (*Susp* [] *X*)
⟨*proof*⟩

lemma *subst-swap-comm*:
shows *subst s* (*swap pi t*) = *swap pi* (*subst s t*)
⟨*proof*⟩

lemma *subst-not-occurs*:

assumes $\neg(occurs\ X\ t)$
shows $subst\ [(X,t2)]\ t = t$
⟨*proof*⟩

lemma *id-subst* [*simp*]:
shows $subst\ []\ t = t$
⟨*proof*⟩

lemma *subst-comp-id* [*simp*]:
shows $subst\ (s \cdot []) = subst\ s$
⟨*proof*⟩

lemma *id-comp-subst* [*simp*]:
shows $subst\ ([] \cdot s) = subst\ s$
⟨*proof*⟩

lemma *subst-comp-expand*:
shows $subst\ (s1 \cdot s2)\ t = subst\ s1\ (subst\ s2\ t)$
⟨*proof*⟩

lemma *subst-assoc*:
shows $subst\ (s1 \cdot (s2 \cdot s3)) = subst\ ((s1 \cdot s2) \cdot s3)$
⟨*proof*⟩

lemma *fresh-subst*:
assumes $nabla1 \vdash a \sharp t$
and $nabla2 \models (subst\ s)\ nabla1$
shows $nabla2 \vdash a \sharp subst\ s\ t$
⟨*proof*⟩

lemma *equ-subst*:
assumes $nabla1 \vdash t1 \approx t2$
and $nabla2 \models (subst\ s)\ nabla1$
shows $nabla2 \vdash (subst\ s\ t1) \approx (subst\ s\ t2)$
⟨*proof*⟩

lemma *unif-1*:
assumes $nabla \vdash subst\ s\ (Susp\ pi\ X) \approx subst\ s\ t$
shows $nabla \models subst\ (s \cdot [(X,\ swap\ (rev\ pi)\ t)]) \approx subst\ s$
⟨*proof*⟩

lemma *subst-equ-a*:
assumes $nabla \models (subst\ s1) \approx (subst\ s2)$
and $nabla \vdash (subst\ s2\ t1) \approx t2$
shows $nabla \vdash (subst\ s1\ t1) \approx t2$
⟨*proof*⟩

lemma *unif-2a*:
assumes $nabla \models subst\ s1 \approx subst\ s2$

and (*nabla*⊢*subst s2 t1* ≈ *subst s2 t2*)
shows(*nabla*⊢*subst s1 t1* ≈ *subst s1 t2*)
⟨*proof*⟩

lemma *unif-2b*:
assumes *nabla* ⊨ *subst s1* ≈ *subst s2*
and *nabla* ⊢ *a* ♯ *subst s2 t*
shows *nabla* ⊢ *a* ♯ *subst s1 t*
⟨*proof*⟩

lemma *subst-equ-to-trm*:
assumes *nabla* ⊨ *subst s1* ≈ *subst s2*
shows *nabla*⊢*subst s1 t* ≈ *subst s2 t*
⟨*proof*⟩

lemma *subst-cancel-right*:
assumes *nabla*⊨(*subst s1*)≈(*subst s2*)
shows *nabla*⊨(*subst* (*s1*•*s*))≈(*subst* (*s2*•*s*))
⟨*proof*⟩

lemma *subst-trans*:
assumes *nabla*⊨*subst s1* ≈ *subst s2 nabla* ⊨ *subst s2* ≈ *subst s3*
shows *nabla*⊨*subst s1*≈*subst s3*
⟨*proof*⟩

If occurs holds, then one subterm is equal to (subst s (Susp pi X))

lemma *occurs-sub-trm-equ*:
assumes *occurs X t1*
shows ∃ *t2* ∈ *sub-trms* (*subst s t1*). (∃ *pi*.(*nabla*⊢*subst s* (*Susp pi1 X*)≈(*swap pi t2*)))
⟨*proof*⟩

lemma *ext-subst-strong*:
assumes *nabla1* ⊨ (*subst s*) (*nabla2* ∪ *nabla3*)
shows *nabla1* ⊨ (*subst s*) *nabla2* **and** *nabla1* ⊨ (*subst s*) *nabla3*
⟨*proof*⟩

lemma *ext-subst-id*:
shows *nabla* ⊨ (*subst* []) *nabla*
⟨*proof*⟩

10 Most General Unifiers

Defines the notion of unification problems and reduction rules over sets of such problems; proves that every reduction leading to the empty set produces an mgu.

syntax *equ-prob* :: *trm* ⇒ *trm* ⇒ (*trm* × *trm*) (**infix** ≈*? 81*)
fresh-prob :: *string* ⇒ *trm* ⇒ (*string* × *trm*) (**infix** ♯*? 81*)

translations
$t1 \approx ?\ t2 \rightharpoonup (t1,\ t2)$
$a \sharp ?\ t \rightharpoonup (a,t)$

All solutions for a unification problem.

type-synonym *problem-type* = $((trm \times trm)\ list) \times ((string \times trm)\ list)$

type-synonym *unifier-type* = *fresh-envs* × *substs*

definition U:: *problem-type* ⇒ (*unifier-type set*)
where *all-solutions-def*:
$U\ P \equiv \{(nabla,\ s).$
$(\forall\ (t1,t2) \in set\ (fst\ P).\ nabla \vdash subst\ s\ t1 \approx subst\ s\ t2) \wedge$
$(\forall\ (a,t) \in set\ (snd\ P).\ nabla \vdash a \sharp subst\ s\ t)\}$

The set of variables in unification problems.

type-synonym *eprobs* = $((trm \times trm)\ list)$
type-synonym *fprobs* = $((string \times trm)\ list)$
type-synonym *probs* = *eprobs* × *fprobs*

fun *vars-fprobs*:: $((string \times trm)\ list) \Rightarrow (string\ set)$
where
vars-fprobs [] = {} |
vars-fprobs $(x\#xs) = (vars\text{-}trm\ (snd\ x)) \cup (vars\text{-}fprobs\ xs)$

fun *vars-eprobs* :: $((trm \times trm)\ list) \Rightarrow (string\ set)$
where
vars-eprobs [] = {} |
vars-eprobs $(x\#xs) = (vars\text{-}trm\ (snd\ x)) \cup (vars\text{-}trm\ (fst\ x)) \cup (vars\text{-}eprobs\ xs)$

definition *vars-probs* :: *problem-type* ⇒ *nat*
where *vars-probs* $P \equiv card((vars\text{-}fprobs\ (snd\ P)) \cup (vars\text{-}eprobs\ (fst\ P)))$

Most general unifier

definition *mgu* :: *problem-type* ⇒ *unifier-type* ⇒ *bool*
where *mgu P unif* ≡
$\forall\ (nabla,s1) \in U\ P.\ (\exists\ s2.\ (nabla \models (subst\ s2)\ (fst\ unif)) \wedge$
$(nabla \models subst\ (s2 \cdot (snd\ unif)) \approx subst\ s1))$

Idempotency of a unifier

definition *idem* :: *unifier-type* ⇒ *bool*
where *idem unif* ≡ $(fst\ unif) \models subst\ ((snd\ unif) \cdot (snd\ unif)) \approx subst\ (snd\ unif)$

Application of a substitution to a problem

definition *apply-subst-eprobs* :: *substs* ⇒ *eprobs* ⇒ *eprobs*
where *apply-subst-eprobs s P* ≡ $map\ (\lambda(t1,\ t2).\ (subst\ s\ t1 \approx ?\ subst\ s\ t2))\ P$

definition *apply-subst-fprobs* :: *substs* ⇒ *fprobs* ⇒ *fprobs*

where *apply-subst-fprobs s P* ≡ *map* (λ(*a*, *t*). (*a* ♯*? subst s t*)) *P*

definition *apply-subst* :: *substs* ⇒ *problem-type* ⇒ *problem-type*
where *apply-subst s P* ≡ (*map* (λ(*t1*, *t2*). (*subst s t1* ≈*? subst s t2*)) (*fst P*),
map (λ(*a*, *t*). (*a* ♯*?* (*subst s t*))) (*snd P*))

Equality reductions

inductive *s-red* :: *problem-type* ⇒ *substs* ⇒ *problem-type* ⇒ *bool* (- ⊢ - ⇝ - [*80*,*80*,*80*] *80*)
where
unit-sred[*intro*!]: ((*Unit* ≈*? Unit*) # *xs*, *ys*) ⊢ [] ⇝ (*xs*, *ys*) |
paar-sred[*intro*!]: ((*Paar t1 t2* ≈*? Paar s1 s2*)#*xs*,*ys*) ⊢[]⇝ ((*t1*≈*?s1*)#(*t2*≈*?s2*)#*xs*,*ys*)
|
func-sred[*intro*!]: ((*Func F t1* ≈*? Func F t2*)#*xs*,*ys*) ⊢[]⇝ ((*t1*≈*?t2*)#*xs*,*ys*) |
abst-aa-sred[*intro*!]: ((*Abst a t1* ≈*? Abst a t2*)#*xs*,*ys*) ⊢[]⇝ ((*t1*≈*?t2*)#*xs*,*ys*) |
abst-ab-sred[*intro*!]: *a*≠*b* ⟹
((*Abst a t1*≈*?Abst b t2*)#*xs*,*ys*) ⊢[]⇝ ((*t1*≈*?swap* [(*a*,*b*)] *t2*)#*xs*,(*a*♯*?t2*)#*ys*) |
atom-sred[*intro*!]: ((*Atom a*≈*?Atom a*)#*xs*,*ys*) ⊢[]⇝ (*xs*,*ys*) |
susp-sred[*intro*!]: ((*Susp pi1 X*≈*?Susp pi2 X*)#*xs*,*ys*)
⊢[]⇝ (*xs*,(*map* (λ*a*. *a*♯*? Susp* [] *X*) (*ds-list pi1 pi2*))@*ys*) |
var-1-sred[*intro*!]: ¬(*occurs X t*) ⟹ ((*Susp pi X*≈*?t*)#*xs*,*ys*)
⊢[(*X*,*swap* (*rev pi*) *t*)]⇝ *apply-subst* [(*X*,*swap* (*rev pi*) *t*)] (*xs*,*ys*) |
var-2-sred[*intro*!]: ¬(*occurs X t*) ⟹ ((*t*≈*?Susp pi X*)#*xs*,*ys*)
⊢[(*X*,*swap* (*rev pi*) *t*)]⇝ *apply-subst* [(*X*,*swap* (*rev pi*) *t*)] (*xs*,*ys*)

inductive-cases *s-red-elims*:
((*Unit* ≈*? Unit*) # *xs*, *ys*) ⊢[] ⇝ (*xs*, *ys*)
((*Paar t1 t2* ≈*? Paar s1 s2*)#*xs*,*ys*) ⊢[]⇝ ((*t1*≈*?s1*)#(*t2*≈*?s2*)#*xs*,*ys*)
((*Func F t1* ≈*? Func F t2*)#*xs*,*ys*) ⊢[]⇝ ((*t1*≈*?t2*)#*xs*,*ys*)
((*Abst a t1* ≈*? Abst a t2*)#*xs*,*ys*) ⊢[]⇝ ((*t1*≈*?t2*)#*xs*,*ys*)
((*Abst a t1*≈*?Abst b t2*)#*xs*,*ys*) ⊢[]⇝ ((*t1*≈*?swap* [(*a*,*b*)] *t2*)#*xs*,(*a*♯*?t2*)#*ys*)
((*Atom a*≈*?Atom a*)#*xs*,*ys*) ⊢[]⇝ (*xs*,*ys*)
((*Susp pi1 X*≈*?Susp pi2 X*)#*xs*,*ys*) ⊢[]⇝ (*xs*,(*map* (λ*a*. *a*♯*? Susp* [] *X*) (*ds-list pi1 pi2*))@*ys*)
((*Susp pi X*≈*?t*)#*xs*,*ys*) ⊢[(*X*,*swap* (*rev pi*) *t*)]⇝ *apply-subst* [(*X*,*swap* (*rev pi*) *t*)] (*xs*,*ys*)
((*t*≈*?Susp pi X*)#*xs*,*ys*) ⊢[(*X*,*swap* (*rev pi*) *t*)]⇝ *apply-subst* [(*X*,*swap* (*rev pi*) *t*)] (*xs*,*ys*)

lemma *sred-symm*:
assumes ((*t1* ≈*? t2*) # *xs*, *ys*) ⊢ *s* ⇝ *P2*
shows ∃ *P3*. ((*t2* ≈*? t1*) # *xs*, *ys*) ⊢ *s* ⇝ *P3*
⟨*proof*⟩

Weakening of freshness

lemma *solution-weak*:
assumes $(nabla1,\ s) \in U\ P$
shows $(nabla1 \cup nabla3,\ s) \in U\ P$
⟨*proof*⟩

lemma *solution-comp-id*:
shows $((nabla,\ s) \in U\ P) = ((nabla,\ s \cdot []) \in U\ P)$ **and**
$((nabla,\ s) \in U\ P) = ((nabla,\ [] \cdot s) \in U\ P)$
⟨*proof*⟩

lemma *solutions-subst-assoc*:
$((nabla,\ s1 \cdot (s2 \cdot s3)) \in U\ P) = ((nabla,\ (s1 \cdot s2) \cdot s3) \in U\ P)$
⟨*proof*⟩

Freshness reductions

inductive *c-red* :: *problem-type* ⇒ *fresh-envs* ⇒ *problem-type* ⇒ *bool* (- ⊢ - → - [*80*,*80*,*80*] *80*)
where
unit-cred[*intro*!]: ([],(*a* ♯? *Unit*)#*xs*) ⊢{}→ ([],*xs*) |
paar-cred[*intro*!]: ([],(*a* ♯? *Paar t1 t2*)#*xs*) ⊢{}→ ([],(*a*♯?*t1*)#(*a*♯?*t2*)#*xs*) |
func-cred[*intro*!]: ([],(*a* ♯? *Func F t*)#*xs*) ⊢{}→ ([],(*a*♯?*t*)#*xs*) |
abst-aa-cred[*intro*!]: ([],(*a* ♯? *Abst a t*)#*xs*) ⊢{}→ ([],*xs*) |
abst-ab-cred[*intro*!]: *a*≠*b* ⟹([],(*a* ♯? *Abst b t*)#*xs*) ⊢{}→ ([],(*a*♯?*t*)#*xs*) |
atom-cred[*intro*!]: *a*≠*b* ⟹([],(*a* ♯? *Atom b*)#*xs*) ⊢{}→ ([],*xs*) |
susp-cred[*intro*!]: ([],(*a* ♯? *Susp pi X*)#*xs*) ⊢ {((*swapas* (*rev pi*) *a*),*X*)}→ ([],*xs*)

It only reduces the freshness part after the equations list is empty

lemma *c-red-eqs-empty*:
assumes *P1* ⊢ *s* → *P2*
shows *fst P1* = []
⟨*proof*⟩

Unification reduction sequences

inductive *red-plus* :: *problem-type* ⇒ *unifier-type* ⇒ *problem-type* ⇒ *bool* (- ⊨ - ⇒ - [*80*,*80*,*80*] *80*)
where
sred-single[*intro*!]: ⟦*P1*⊢*s1* ⇝ *P2*⟧ ⟹ *P1*⊨({},*s1*)⇒*P2* |
cred-single[*intro*!]: ⟦*P1*⊢*nabla1*→*P2*⟧ ⟹ *P1* ⊨ (*nabla1*,[]) ⇒ *P2* |
sred-step[*intro*!]: ⟦*P1*⊢*s1*⇝*P2*; *P2*⊨(*nabla2*,*s2*)⇒*P3*⟧ ⟹ *P1*⊨(*nabla2*,(*s2*•*s1*))⇒*P3* |
cred-step[*intro*!]: ⟦*P1*⊢*nabla1*→*P2*; *P2*⊨(*nabla2*,[])⇒*P3*⟧ ⟹ *P1*⊨(*nabla2*∪*nabla1*,[])⇒*P3*

Symmetry of the reduction

lemma *red-plus-symm*:
assumes*P1* = ((*t1* ≈? *t2*) # *xs*, *ys*)
and *P1* ⊨ (*nabla*, *s*) ⇒ *P2*
shows ∃ *nabla1 s1 P3*. ((*t2* ≈? *t1*) # *xs*, *ys*) ⊨ (*nabla1*, *s1*) ⇒ *P3*
⟨*proof*⟩

lemma *mgu-idem*:
assumes $(nabla1,s1) \in U\ P$
and $\forall (nabla2,s2) \in U\ P.\ nabla2 \models (subst\ s2)\ nabla1 \wedge nabla2 \models subst(s2 \cdot s1) \approx subst\ s2$
shows $mgu\ P\ (nabla1,s1) \wedge idem\ (nabla1,s1)$
⟨*proof*⟩

lemma *problem-subst-comm*:
shows $((nabla,s2) \in U\ (apply\text{-}subst\ s1\ P)) = ((nabla,(s2 \cdot s1)) \in U\ P)$
⟨*proof*⟩

Preservation of solutions

lemma *P1-to-P2-sred*:
assumes $(nabla1,s1) \in U\ P1$ **and** $P1 \vdash s2 \rightsquigarrow P2$
shows $(nabla1,s1) \in U\ P2 \wedge (nabla1 \models subst\ (s1 \cdot s2) \approx subst\ s1)$
⟨*proof*⟩

Auxiliary lemma for completeness

lemma *P1-from-P2-sred*:
assumes $(nabla1,s1) \in U\ P2$ **and** $P1 \vdash s2 \rightsquigarrow P2$
shows $(nabla1,s1 \cdot s2) \in U\ P1$
⟨*proof*⟩

lemma *P1-to-P2-cred*:
assumes $(nabla1,s1) \in U\ P1$
and $P1 \vdash nabla2 \rightarrow P2$
shows $(nabla1,s1) \in U\ P2 \wedge (nabla1 \models (subst\ s1)\ nabla2)$
⟨*proof*⟩

lemma *P1-from-P2-cred*:
assumes $(nabla1,s1) \in U\ P2$
and $P1 \vdash nabla2 \rightarrow P2$ **and** $nabla3 \models (subst\ s1)\ nabla2$
shows $(nabla1 \cup nabla3,s1) \in U\ P1$
⟨*proof*⟩

lemma *P1-to-P2-red-plus1*:
assumes $P1 \models (nabla,s) \Rightarrow P2$
and $(nabla1,s1) \in U\ P1$
shows $(nabla1,s1) \in U\ P2$
⟨*proof*⟩

lemma *P1-to-P2-red-plus3*:
assumes $P1 \models (nabla,s) \Rightarrow P2$
and $(nabla1,s1) \in U\ P1$
shows $nabla1 \models (subst\ s1)\ nabla$
⟨*proof*⟩

lemma *P1-to-P2-red-plus2*:
assumes *P1* $\models$ (*nabla*, *s*) $\Rightarrow$ *P2*
and (*nabla1*, *s1*) $\in$ *U P1*
shows *nabla1* $\models$ *subst* (*s1* $\cdot$ *s*) $\approx$ *subst s1*
⟨*proof*⟩

lemma *P1-from-P2-red-plus*:
assumes *P1* $\models$(*nabla*,*s*)$\Rightarrow$*P2*
and (*nabla1*,*s1*)$\in$ *U P2* **and** *nabla3*$\models$(*subst s1*)(*nabla*)
shows (*nabla1*$\cup$*nabla3*,(*s1* $\cdot$ *s*))$\in$ *U P1*
⟨*proof*⟩

lemma *mgu*:
assumes *P* $\models$(*nabla*,*s*)$\Rightarrow$([],[])
shows *mgu P* (*nabla*,*s*) $\wedge$ *idem* (*nabla*,*s*)
⟨*proof*⟩

11 Termination

Defines a lexicographic termination measure and proves that all the unification reductions decrease this measure.

lemma *apply-subst-equivalence*:
shows *apply-subst s P* = (*apply-subst-eprobs s* (*fst P*), *apply-subst-fprobs s* (*snd P*))
⟨*proof*⟩

fun *size-trm* :: *trm* $\Rightarrow$ *nat*
where
size-trm (*Unit*) = *1* |
size-trm (*Atom a*) = *1* |
size-trm (*Susp pi X*) = *1* |
size-trm (*Abst a t*) = *1* + *size-trm t* |
size-trm (*Func F t*) = *1* + *size-trm t* |
size-trm (*Paar t t'*) = *1* + (*size-trm t*) + (*size-trm t'*)

fun *size-fprobs* :: *fprobs* $\Rightarrow$ *nat*
where
size-fprobs [] = *0* |
size-fprobs (*x*#*xs*) = (*size-trm* (*snd x*))+(*size-fprobs xs*)

fun *size-eprobs* :: *eprobs* $\Rightarrow$ *nat*
where
size-eprobs [] = *0* |
size-eprobs (*x*#*xs*) = (*size-trm* (*fst x*))+(*size-trm* (*snd x*))+(*size-eprobs xs*)

lemma *size-swap* [*simp*]: *size-trm* (*swap pi t*) = *size-trm t*

⟨*proof*⟩

definition *rank-r*
where
rank-r = *measures* [λ(*eprobs*, *fprobs*). *card* (*vars-eprobs eprobs*),
λ(*eprobs*, *fprobs*). *size-eprobs eprobs*,
λ(*eprobs*, *fprobs*). *size-fprobs fprobs*]

lemma *vars-term-finite* [*simp*]: *finite* (*vars-trm t*)
⟨*proof*⟩

lemma *vars-eprobs-finite* [*simp*]: *finite* (*vars-eprobs P*)
⟨*proof*⟩

lemma *not-occurs-trm*: ¬*occurs X t* ⟶ *X*∉ *vars-trm t*
⟨*proof*⟩

lemma *not-occurs-subst*: ¬*occurs X t1*⟶ *X*∉ *vars-trm* (*subst* [(*X*,*swap pi2 t1*)] *t2*)
⟨*proof*⟩

lemma *not-occurs-list*: ¬ *occurs X t* ⟶
X ∉ *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] *xs*)
⟨*proof*⟩

lemma *vars-equ*:
assumes ¬*occurs X t1* **and** *occurs X t2*
shows *vars-trm* (*subst* [(*X*, *swap pi t1*)] *t2*) = (*vars-trm t1* ∪ *vars-trm t2*)−{*X*}
⟨*proof*⟩

lemma *vars-subseteq*:
assumes ¬*occurs X t*
shows *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] *xs*) ⊆ (*vars-trm t* ∪ *vars-eprobs xs*)
⟨*proof*⟩

lemma *vars-decrease*:
assumes ¬*occurs X t*
shows *card* (*vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] *xs*))
< *card* (*insert X* (*vars-trm t* ∪ *vars-eprobs xs*))
⟨*proof*⟩

lemma *rank-r-cred*:
assumes *P1*⊢(*nabla*)→*P2*
shows (*P2*, *P1*) ∈ *rank-r*
⟨*proof*⟩

lemma *rank-r-sred*:
assumes *P1* ⊢ *s* ⇝*P2*
shows (*P2*,*P1*) ∈ *rank-r*
⟨*proof*⟩

lemma *rank-r-trans*: ⟦(*P1*,*P2*) ∈ *rank-r*; (*P2*,*P3*) ∈ *rank-r*⟧⟹ (*P1*,*P3*)∈ *rank-r*
⟨*proof*⟩

lemma *rank-r-red-plus*:
assumes *P1*⊨ (*s*,*nabla*)⇒*P2*
shows (*P2*, *P1*) ∈ *rank-r*
⟨*proof*⟩

lemma *wf-rank-r*:
shows *wf* (*rank-r*)
⟨*proof*⟩

12 Unification

Proves that all problems that are stuck and fail, have no unifier.

definition *stuck* :: *problem-type set* **where**
stuck-def: *stuck* ≡ { *P1*. ¬(∃ *P2 nabla s*. *P1* ⊨(*nabla*,*s*)⇒*P2*)}

inductive *fail* :: *problem-type* ⇒ *bool* **where**
fail-occur-abst [*intro*!]: ⟦*occurs X t*⟧⟹ *fail* ((*Susp pi X* ≈*? Abst a t*) # *xs*, *ys*) |
fail-occur-func [*intro*!]: ⟦*occurs X t*⟧⟹ *fail* (*Susp pi X* ≈*?Func F t*#*xs*,*ys*) |
fail-occur-paar [*intro*!]: ⟦*occurs X t1*∨*occurs X t2*⟧⟹ *fail* (*Susp pi X*≈*?Paar t1 t2*#*xs*,*ys*) |
fail-fresh-atom [*intro*!]: *fail* ([],*a*♯*? Atom a*#*ys*)|
fail-diff-atoms [*intro*!]: *a*≠*b*⟹ *fail* (*Atom a*≈*? Atom b*#*xs*,*ys*) |
fail-abst-unit [*intro*!]: *fail* (*Abst a t*≈*?Unit*#*xs*,*ys*) |
fail-abst-atom [*intro*!]: *fail* (*Abst a t*≈*?Atom b*#*xs*,*ys*) |
fail-abst-paar [*intro*!]: *fail* (*Abst a t*≈*?Paar t1 t2*#*xs*,*ys*) |
fail-func-abst [*intro*!]: *fail* (*Func F t1*≈*?Abst a t*#*xs*,*ys*) |
fail-atom-unit [*intro*!]: *fail* (*Atom b*≈*?Unit*#*xs*,*ys*) |
fail-paar-unit [*intro*!]: *fail* (*Paar t1 t2*≈*?Unit*#*xs*,*ys*) |
fail-func-unit [*intro*!]: *fail* (*Func F t1*≈*?Unit*#*xs*,*ys*) |
fail-atom-paar [*intro*!]: *fail* (*Atom a*≈*?Paar t1 t2*#*xs*,*ys*) |
fail-func-atom [*intro*!]: *fail* (*Func F t1*≈*?Atom a*#*xs*,*ys*) |
fail-func-paar [*intro*!]: *fail* (*Func F t*≈*?Paar t1 t2*#*xs*,*ys*) |
fail-diff-func [*intro*!]: ⟦*F1*≠*F2*⟧⟹ *fail* (*Func F1 t1*≈*?Func F2 t2*#*xs*,*ys*) |
fail-sym [*intro*!]: *fail* (*s* ≈*? t* # *xs*, *ys*) ⟹ *fail* (*t* ≈*? s* # *xs*, *ys*)

definition
normal-form :: *problem-type* ⇒ *problem-type set* **where**
normal-form P1 ≡ *if P1* ∈ *stuck then* {*P1*} *else* {*P2*. ∃ *nabla s*. *P1*⊨(*nabla*,*s*)⇒*P2* ∧ *P2*∈*stuck*}

lemma *U-equ-symm*:
shows *U* (*s*≈?*t*#*xs*, *ys*) = *U* (*t*≈?*s*#*xs*, *ys*)
⟨*proof*⟩

lemma *fail-then-empty*:
assumes *fail P1*
shows *U P1* = {}
⟨*proof*⟩

lemma *not-reduce-then-fail*:
assumes ¬ (∃ *nabla s P'*. ((*t1* ≈? *t2*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P'*)
shows *fail* ((*t1* ≈? *t2*) # *xs*, *ys*)
⟨*proof*⟩

lemma *fresh-reduces-if-not-atom*:
assumes *t* ≠ *Atom a*
shows ∃ *P2 nabla s*. ([], (*a* ♯? *t*) # *xs*) ⊨ (*nabla*,*s*) ⇒ *P2*
⟨*proof*⟩

lemma *empty-stuck*:
shows ([],[]) ∈ *stuck*
⟨*proof*⟩

lemma *fail-is-stuck*:
assumes *fail P*
shows *P* ∈ *stuck*
⟨*proof*⟩

lemma *stuck-equiv*:
shows *stuck* = {([],[])} ∪ {*P1*. *fail P1*}
⟨*proof*⟩

lemma *u-empty-sred*:
assumes *P1*⊢*s*⇝*P2* **and** *U P2* ={}
shows *U P1* = {}
⟨*proof*⟩

lemma *u-empty-cred*:
assumes *P1*⊢*nabla*→*P2* **and** *U P2* ={}
shows *U P1*={}
⟨*proof*⟩

lemma *u-empty-red-plus*:
assumes *P1*⊨(*nabla*,*s*)⇒*P2* **and** *U P2* ={}
shows *U P1*={}
⟨*proof*⟩

lemma *empty-then-fail*:
assumes *U P1*={}
shows (∀ *P* ∈ *normal-form P1*. *fail P*)
⟨*proof*⟩

lemma *not-empty-then-not-fail*:
assumes *U P1*≠{}
shows ¬(∃ *P*∈ *normal-form P1*. *fail P*)
⟨*proof*⟩

References

[1] M. Ayala-Rincón, W. de Carvalho Segundo, M. Fernández, D. Nantes-Sobrinho, and A. C. R. Oliveira. A formalisation of nominal α-equivalence with a, c, and AC function symbols. *Theor. Comput. Sci.*, 781:3–23, 2019.

[2] M. Ayala-Rincón, M. Fernández, and A. C. R. Oliveira. Completeness

in PVS of a nominal unification algorithm. In M. R. F. Benevides and R. Thiemann, editors, *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2015, Natal, Brazil, August 31 - September 1, 2015*, volume 323 of *Electronic Notes in Theoretical Computer Science*, pages 57–74. Elsevier, 2015.

[3] W. L. R. de Carvalho. *Nominal Equational Problems Modulo Associativity, Commutativity and Associativity-Commutativity.* PhD thesis, Graduate Program in Informatics, University of Brasília, 2019. in English.

[4] A. C. R. Oliveira. *Unification, Confluence, and Intersection Types for Nominal Rewriting Systems.* PhD thesis, Graduate Program in Informatics, University of Brasília, 2016. in English.

[5] C. Urban. Nominal unification revisited. In M. Fernández, editor, *Proceedings 24th International Workshop on Unification, UNIF 2010, Edinburgh, United Kingdom, 14th July 2010*, volume 42 of *EPTCS*, pages 1–11, 2010.

[6] C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.