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.

May 8, 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*
by *simp*

lemma *swapas-ab-ba*:
shows *swapas* [(*a*,*b*)] = *swapas* [(*b*,*a*)]
by *auto*

lemma *swapas-append*:
shows *swapas* (*pi1*@*pi2*) *a* = *swapas pi1* (*swapas pi2 a*)
by (*induct pi1*) *auto*

lemma *swapas-inv* [*simp*]:
shows *swapas* (*rev pi*) (*swapas pi a*) = *a*
by (*induct-tac pi*) (*auto simp add*: *swapas-append*)

lemma *swapas-rev-pi-a*:
assumes *swapas pi a* = *b*
shows *swapas* (*rev pi*) *b* = *a*
using *assms* **by** *auto*

lemma *swapas-rev-swapas-id* [*simp*]:
shows *swapas pi* (*swapas* (*rev pi*) *a*) = *a*
by (*metis swapas-rev-pi-a*)

lemma *swapas-rev-pi-b*:
assumes *swapas* (*rev pi*) *a* = *b*
shows *swapas pi b* = *a*
using *assms* **by** *auto*

lemma *swapas-comm*:
shows (*swapas* (*pi*@[(*a*,*b*)]) *c*) = (*swapas* ([(*swapas pi a*,*swapas pi b*)]@*pi*) *c*)
by (*metis swapa.simps swapas.simps*(*1*,*2*) *swapas-append swapas-rev-pi-a*)

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*
by (*induct-tac t*) (*auto*)

lemma *swap-append*:
shows *swap* (*pi1*@*pi2*) *t* = *swap pi1* (*swap pi2 t*)
using *swapas-append* **by** (*induct t arbitrary*: *pi1 pi2*) *auto*

lemma *swap-empty*:
assumes *swap pi t* = *Susp* [] *X*
shows *pi* = []
using *assms swap.elims* **by** *blast*

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*
by(*auto*)

lemma *swap-depth* [*simp*]:
shows *depth* (*swap pi t*) = *depth t*
by (*induct t*) (*auto*)

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*
  by (*induct t*) *auto*

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*
  using *assms*
  by(*induct t2*)(*auto*)

lemma *t-sub-trms-t*:
  shows *t* ∈ *sub-trms t*
  by (*induct t*) (*auto*)

lemma *abst-psub-trms*:
  assumes *Abst a t1* ∈ *sub-trms t2*
  shows *t1* ∈ *psub-trms t2*
  using *assms*
  by (*induct t2 arbitrary*: *t1*)
  (*auto simp add*: *t-sub-trms-t intro*: *psub-sub-trms*)

lemma *func-psub-trms*:
assumes *Func F t1* ∈ *sub-trms t2*
shows *t1* ∈ *psub-trms t2*
using *assms*
by (*induct t2*)
(*auto simp add*: *t-sub-trms-t intro*: *psub-sub-trms*)

lemma *paar-psub-trms*:
assumes *Paar t1 t2* ∈ *sub-trms t3*
shows *t1* ∈ *psub-trms t3* **and** *t2* ∈ *psub-trms t3*
using *assms*
by (*induct t3*) (*auto simp add*: *t-sub-trms-t intro*: *psub-sub-trms*)

lemma *t-not-in-psub-trms-t*:
shows *t* ∉ *psub-trms t*
proof(*induct t*)
case (*Abst x1 t*)
then show *?case*
using *abst-psub-trms* **by** *auto*
next
case (*Susp x1 x2*)
then show *?case* **by** *simp*
next
case *Unit*
then show *?case* **by** *simp*
next
case (*Atom x*)
then show *?case* **by** *simp*
next
case (*Paar t1 t2*)
then have *Paar t1 t2* ∉ *sub-trms t1 Paar t1 t2* ∉ *sub-trms t2*
using *paar-psub-trms* **by** *auto*
then show *?case* **by** *simp*
next
case (*Func f t*)
then show *?case*
using *func-psub-trms* **by** *auto*
qed

lemma *if-sub-not-eq-then-psub*:
assumes *t1* ∈ *sub-trms t2 t1* ≠ *t2*
shows *t1* ∈ *psub-trms t2*
using *assms*
by (*induct t2*) *auto*

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

using *assms*
proof(*induct t2 arbitrary*: *t1*)
case (*Abst a t*)
then show *?case*
using *depth.simps*(*4*)[*of a t*]
psub-trms.simps(*5*)[*of a t*] *if-sub-not-eq-then-psub*
by *fastforce*
next
case (*Susp pi X*)
then show *?case* **by** *simp*
next
case *Unit*
then show *?case* **by** *simp*
next
case (*Atom a*)
then show *?case* **by** *simp*
next
case (*Paar t1' t2'*)
then show *?case*
using *depth.simps*(*6*)[*of t1' t2'*]
psub-trms.simps(*4*) *if-sub-not-eq-then-psub* **by** *fastforce*
next
case (*Func f t1'*)
then show *?case*
using *depth.simps*(*5*)[*of f t1'*]
psub-trms.simps(*6*)[*of f t1'*] *if-sub-not-eq-then-psub*
by *fastforce*
qed

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*
by (*induct xs*) *auto*

lemma *atms-rev*[*simp*]:
shows *atms* (*rev pi*) = *atms pi*
by (*induct pi*) *auto*

lemma *a-not-in-atms*:
assumes *a* ∉ *atms pi*
shows *a* = *swapas pi a*
using *assms* **by** (*induct pi*) *auto*

lemma *swapas-pi-ineq-a*:
assumes *swapas pi a* $\neq$ *a*
shows *a* $\in$ *atms pi*
using *a-not-in-atms assms* **by** *force*

lemma *a-ineq-swapas-pi*:
assumes *a* $\neq$ *swapas pi a*
shows *a* $\in$ *atms pi*
using *a-not-in-atms assms* **by** *blast*

lemma *swapas-pi-in-atms*:
assumes *a* $\in$ *atms pi*
shows*swapas pi a* $\in$ *atms pi*
using *assms* **by** (*metis a-not-in-atms swapas-rev-pi-a*)

5 Disagreement Sets

definition *ds* :: (*string* $\times$ *string*) *list* $\Rightarrow$ (*string* $\times$ *string*) *list* $\Rightarrow$ *string set* **where**
ds-def: *ds xs ys* $\equiv$ { *a* . *a* $\in$ (*atms xs* $\cup$ *atms ys*) $\wedge$ (*swapas xs a* $\neq$ *swapas ys a*) }

lemma *ds-elem*:
assumes *swapas pi a*$\neq$*a*
shows *a*$\in$*ds* [] *pi*
using *assms ds-def swapas-pi-ineq-a* **by** *simp*

corollary *ds-elem-cp*:
assumes *a* $\notin$ *ds* [] *pi*
shows *swapas pi a* = *a*
using *assms ds-elem* **by** *blast*

lemma *elem-ds*:
assumes *a*$\in$*ds* [] *pi*
shows *a*$\neq$*swapas pi a*
using *assms ds-def* **by** *simp*

lemma *ds-sym*:
shows *ds pi1 pi2* = *ds pi2 pi1*
using *ds-def* **by** *auto*

lemma *ds-trans*:
assumes *c* $\in$ *ds pi1 pi3*
shows *c* $\in$ *ds pi1 pi2* $\vee$ *c* $\in$ *ds pi2 pi3*
using *assms ds-def a-not-in-atms swapas-pi-ineq-a* **by** *auto*

lemma *ds-cancel-pi-left*:
assumes *c*$\in$ *ds* (*pi1*@*pi*) (*pi2*@*pi*)

shows *swapas pi c* ∈ *ds pi1 pi2*
using *assms ds-def swapas-append a-ineq-swapas-pi a-not-in-atms*
by (*metis* (*mono-tags*, *lifting*) *Un-iff mem-Collect-eq*)

lemma *ds-cancel-pi-right*:
assumes *swapas pi c* ∈ *ds pi1 pi2*
shows *c* ∈ *ds* (*pi1*@*pi*) (*pi2*@*pi*)
using *assms*
unfolding *ds-def*
using *a-not-in-atms swapas-append* **by** *auto*

lemma *ds-equality*:
shows (*ds* [] *pi*)−{*a*,*swapas pi a*} = (*ds* [] ((*a*,*swapas pi a*)#*pi*))−{*swapas pi a*}
using *ds-def* **by** *auto*

lemma *ds-7*:
assumes *b* ≠ *swapas pi b a* ∈ *ds* [] ((*b*,*swapas pi b*)#*pi*)
shows *a*∈*ds* [] *pi*
using *assms ds-def swapas-pi-in-atms a-ineq-swapas-pi swapas-rev-pi-a*
ds-elem elem-ds swapa.simps swapas.simps(*2*)
by *metis*

lemma *ds-cancel-pi-front*:
shows *ds* (*pi*@*pi1*) (*pi*@*pi2*) = *ds pi1 pi2*
unfolding *ds-def*
by (*metis* (*lifting*) *Un-iff a-not-in-atms swapas-append swapas-inv*)

lemma *ds-rev-pi-pi*:
shows *ds* ((*rev pi1*)@*pi1*) *pi2* = *ds* [] *pi2*
unfolding *ds-def*
using *a-not-in-atms swapas-append* **by** *auto*

lemma *ds-rev*:
shows *ds* [] ((*rev pi1*)@*pi2*) = *ds pi1 pi2*
using *ds-cancel-pi-front ds-rev-pi-pi* **by** *blast*

lemma *ds-acabbc*:
assumes *a*≠*b b*≠*c c*≠*a*
shows *ds* [(*a*, *b*), (*b*, *c*)] [(*a*, *c*)] = {*a*, *b*}
using *assms ds-def* **by** *auto*

lemma *ds-baab*:
assumes *a*≠*b*
shows *ds* [(*b*,*a*)] [(*a*, *b*)] = {}
using *assms ds-def* **by** *auto*

lemma *ds-baab-id*:
assumes *a*≠*b*
shows *ds* ([(*b*,*a*)]@[(*a*, *b*)]) [] = {}

using *assms ds-def ds-rev ds-baab* **by** *simp*

lemma *ds-abab*:
shows *ds* [] [(*a*, *b*), (*a*, *b*)] = {}
using *ds-def* **by** *auto*

lemma *ds-comm*:
shows *ds* (*pi* @ [(*a*,*b*)]) ([(*swapas pi a*, *swapas pi b*)] @ *pi*) = {}
using *swapas-comm ds-def* **by** *simp*

lemma *ds-rev-pi-id*:
shows *ds* (*rev pi* @ *pi*) [] = {}
using *ds-rev-pi-pi*[*of pi* ‹[]›] *elem-ds*
ds-sym[*of* ‹(*rev pi* @ *pi*)› ‹[]›] **by** *fastforce*

lemma *ds-pi-rev-id*:
shows *ds* (*pi* @ *rev pi*) [] = {}
using *ds-rev-pi-id*[*of* ‹*rev pi*›] *rev-rev-ident*[*of pi*]
by *simp*

lemma *ds-swapas-eq*:
assumes *ds pi1 pi2* = {}
shows *swapas pi1 a* = *swapas pi2 a*
using *assms*
by (*metis ds-elem-cp ds-pi-rev-id ds-rev ds-sym emptyE swapas-append*)

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*
by (*induct pi*) *auto*

lemma *ds-list-equ-ds*:
set (*ds-list pi1 pi2*) = *ds pi1 pi2*
using *ds-list-def ds-def set-flatten-eq-atms* **by** *auto*

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*
using *assms*
by (*induct arbitrary*: *pi rule*: *fresh.induct*)
(*auto simp add*: *swapas-append dest*: *swapas-rev-pi-a*)

lemma *fresh-swap-left*:
assumes *nabla* ⊢ *a* ♯ *swap pi t*
shows *nabla* ⊢ *swapas* (*rev pi*) *a* ♯ *t*
using *assms*
proof(*induct t arbitrary*: *a pi*)
case (*Abst x1 t*)
have *nabla* ⊢ *a* ♯ *swap pi* (*Abst x1 t*) **by** *fact*
then have (*nabla* ⊢ *a* ♯ *swap pi t* ∧ *a* ≠ *swapas pi x1*) ∨ (*a* = *swapas pi x1*)
by (*metis Fresh-elims*(*1*) *swap.simps*(*4*))
moreover {
assume *as*: *nabla* ⊢ *a* ♯ *swap pi t* ∧ *a* ≠ *swapas pi x1*
have *nabla* ⊢ *swapas* (*rev pi*) *a* ♯ *Abst x1 t*
using *Abst.hyps as* **by** *auto*
}
moreover {
assume *as*: *a* = *swapas pi x1*
then have *nabla* ⊢ *swapas* (*rev pi*) *a* ♯ *Abst x1 t*

```
    by (simp add: fresh-abst-aa)
  }
  ultimately show nabla ⊢ swapas (rev pi) a ♯ Abst x1 t
    by argo
next
  case (Susp x1 x2)
  have nabla ⊢ a ♯ swap pi (Susp x1 x2) by fact
  have (swapas (rev x1) (swapas (rev pi) a), x2) ∈ nabla
    by (metis Fresh-elims(4) Susp rev-append swap.simps(3) swapas-append)
  then show nabla ⊢ swapas (rev pi) a ♯ Susp x1 x2
    by (simp add: fresh-susp)
next
  case (Atom x)
  have nabla ⊢ a ♯ swap pi (Atom x) by fact
  then have swapas pi x ≠ a by (auto elim: Fresh-elims)
  then have x ≠ swapas (rev pi) a using swapas-rev-pi-b by blast
  then show nabla ⊢ swapas (rev pi) a ♯ Atom x
    by (simp add: fresh-atom)
qed (auto elim: Fresh-elims)

lemma fresh-swap-right:
  assumes nabla ⊢ swapas (rev pi) a ♯ t
  shows nabla ⊢ a ♯ swap pi t
  using assms fresh-swap-eqvt[of nabla ‹swapas (rev pi) a› t pi]
    swapas-rev-swapas-id[of pi a] by simp

lemma fresh-weak:
  assumes nabla1 ⊢ a ♯ t
  shows (nabla1 ∪ nabla2) ⊢ a ♯ t
  using assms
  by(induct rule: fresh.induct)(auto)

lemma ds-empty-fresh-1:
  assumes ds pi1 pi2 = {}
  shows nabla ⊢ swapas pi1 a ♯ t ⟹ nabla ⊢ swapas pi2 a ♯ t
  using assms ds-swapas-eq by simp

lemma ds-empty-fresh-2:
  assumes ds pi1 pi2 = {}
  shows nabla ⊢ a ♯ swap pi1 t ⟹ nabla ⊢ a ♯ swap pi2 t
  using assms
proof−
  assume assm1: nabla ⊢ a ♯ swap pi1 t
  have i: swapas (rev pi1) a = swapas (rev pi2) a
    using assms ds-swapas-eq swapas-rev-pi-a by metis
  with assms have ii: nabla ⊢ swapas (rev pi2) a ♯ t
    using fresh-swap-left[OF assm1] i by simp
  show ?thesis
    using fresh-swap-right[OF ii] by simp
```

qed

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 ⊢ *Unit* ≈ *Unit*
nabla ⊢ *Susp pi1 X* ≈ *Susp pi2 X*
nabla ⊢ *Paar s1 t1* ≈ *Paar s2 t2*
nabla ⊢ *Func F t1* ≈ *Func F t2*
nabla ⊢ *Abst a t1* ≈ *Abst a t2*
nabla ⊢ *Abst a t1* ≈ *Abst b t2*

lemma *equ-depth*:
assumes *nabla* ⊢ *t1* ≈ *t2*
shows *depth t1* = *depth t2*
using *assms* **by** (*rule equ.induct*, *simp-all*)

lemma *rev-pi-pi-equ*:
shows (*nabla* ⊢ *swap* (*rev pi*) (*swap pi t*) ≈ *t*)
proof(*induct t*)
case (*Susp pi′ X*)
then show *?case*
using *ds-cancel-pi-left*[*of* - ‹*rev pi* @ *pi*› - ‹[]›]
ds-rev-pi-pi elem-ds ds-rev-pi-id **by** *auto*
qed (*auto*)

lemma *equ-pi-right*:
assumes ∀ *a* ∈ *ds* [] *pi*. *nabla* ⊢ *a* ♯ *t*
shows *nabla* ⊢ *t* ≈ *swap pi t*
using *assms*
proof(*induct t arbitrary*: *pi*)

```
 case (Abst b t')
 have swapas pi b = b ∨ swapas pi b ≠ b by simp
moreover
{ assume eq: swapas pi b = b
 have nabla ⊢ Abst b t' ≈ Abst b (swap pi t')
   by (metis Abst.hyps Abst.prems Fresh-elims(1) elem-ds eq equ-abst-aa)
 then have nabla ⊢ Abst b t' ≈ swap pi (Abst b t') using eq by simp
}
moreover
{ assume neq: b ≠ swapas pi b
 obtain c where c-def: c = swapas pi b by simp
 have c ∈ ds [] pi
  by (metis append.right-neutral c-def ds-cancel-pi-front ds-cancel-pi-left ds-elem
neq)
 then have one: nabla ⊢ c ♯ Abst b t' using assms Abst.prems by auto
 have two: b ≠ c using neq c-def by blast
 have nabla ⊢ c ♯ t' using one two by cases auto
 have nabla ⊢ Abst b t' ≈ Abst c (swap pi t')
 proof(rule equ-abst-ab)
  show b ≠ c using neq c-def by blast
  have b-is-swap: b = swapas (rev pi) c using c-def by simp
  show nabla ⊢ b ♯ swap pi t'
  using Abst.prems Fresh-elims(1) ds-elem fresh-swap-right neq swapas-rev-pi-a
by metis
  show nabla ⊢ t' ≈ swap [(b, c)] (swap pi t')
  proof −
    have fresh-ext:
     ∀ a ∈ ds [] ((b,c) # pi). nabla ⊢ a ♯ t'
   proof (rule ballI)
     fix a assume A: a ∈ ds [] ((b,c)#pi)
     then have a = c ∨ a ∈ ds [] pi
       using c-def ds-7 neq by auto
     then show nabla ⊢ a ♯ t'
     proof
       assume a = c
       with ‹nabla ⊢ c ♯ t'› show ?thesis by simp
     next
       assume a ∈ ds [] pi
       show ?thesis
       proof(rule Fresh-elims(1)[of nabla a b t'], goal-cases)
         case 1
         then show ?case
           using Abst.prems ‹a ∈ ds [] pi›
         by simp
       next
         case 2
         then show ?case by simp
       next
         case 3
```

```
            have a ≠ swapas ((b, c) # pi) a
              using A elem-ds
              by blast
            hence a ≠ b using c-def A
              by (auto simp add: if-split)
            hence False using 3 by simp
            then show ?case by simp
          qed
        qed
      qed
        from Abst.hyps[OF fresh-ext]
        have nabla ⊢ t' ≈ swap ([(b,c)]@pi) t'
          by simp
        moreover have swap ([(b,c)]@pi) t' = swap [(b,c)] (swap pi t')
          using swap-append by blast
        ultimately show ?thesis by simp
      qed
    qed
   then have nabla ⊢ Abst b t' ≈ swap pi (Abst b t') using neq c-def by force
 }
  ultimately show ?case by argo
next
  case (Susp pi' X)
  have ∀ a∈ds [] pi. nabla ⊢ a ♯ Susp pi' X by fact
  then have nabla ⊢ Susp pi' X ≈ Susp (pi @ pi') X
   using Fresh-elims(4) equ-susp ds-def ds-cancel-pi-left
   by (metis (lifting) append-self-conv2 swapas-rev-pi-a)
  then show ?case by simp
next
  case Unit
  then show ?case
    using equ-unit swap.simps(2) by force
next
  case (Atom b)
    then show ?case
      apply simp
    apply (rule equ-atom)
    using Fresh-elims(3) ds-elem-cp
    by metis
next
  case (Paar t1 t2)
  then have hypsp1: (∀ a∈ds [] pi.  nabla ⊢ a ♯ t1)
    and hypsp2: (∀ a∈ds [] pi.  nabla ⊢ a ♯ t2)
    using Paar.prems Fresh-elims(5) by meson+
  have nabla ⊢ Paar t1 t2 ≈ Paar (swap pi t1) (swap pi t2)
    using Paar.hyps(1,2) hypsp1 hypsp2 by blast
  then show ?case by simp
next
  case (Func f t')
```

then have *hypsf*: $\forall\, a{\in}ds\ []\ pi.\ nabla \vdash a \sharp t'$ **using** *fresh-func*
by (*metis Fresh-elims*(*6*))
have $nabla \vdash Func\ f\ t' \approx Func\ f\ (swap\ pi\ t')$
using *Func.hyps hypsf* **by** *blast*
then show *?case*
by *simp*
qed

lemma *pi-comm*:
shows $nabla \vdash (swap\ (pi\ @\ [(a,b)])\ t) \approx (swap\ ([(swapas\ pi\ a,\ swapas\ pi\ b)]\ @\ pi)\ t)$
proof(*induct t*)
case (*Abst c t*)
then show *?case* **using** *swapas-comm* **by** *force*
next
case (*Susp* π' *X*)
have *rew1*:
$swap\ (pi\ @\ [(a,b)])\ (Susp\ \pi'\ X) = Susp\ (pi\ @\ [(a,b)]\ @\ \pi')\ X$
by *simp*
have *rew2*:
$swap\ ([(swapas\ pi\ a,\ swapas\ pi\ b)]\ @\ pi)\ (Susp\ \pi'\ X)$
$= Susp\ ([(swapas\ pi\ a,\ swapas\ pi\ b)]\ @\ pi\ @\ \pi')\ X$
by *simp*
have *fresh*:
$\forall\, c \in ds\ (pi\ @\ [(a,b)]\ @\ \pi')\ ([(swapas\ pi\ a,\ swapas\ pi\ b)]\ @\ pi\ @\ \pi').\ (c,\ X) \in nabla$
using *swapas-append swapas-comm ds-def* **by** *simp*
from *rew1 rew2 fresh*
show *?case*
using *equ-susp* **by** *simp*
next
case *Unit*
then show *?case* **by** *force*
next
case (*Atom c*)
then show *?case*
using *equ-atom swapas-comm* **by** *auto*
next
case (*Paar t1 t2*)
then show *?case* **by** *force*
next
case (*Func f t*)
then show *?case* **by** *force*
qed

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

proof(*induction rule*: *equ.induct*)
case (*equ-abst-ab b c nabla t2 t1*)
then show *?case*
using *fresh-swap-left Fresh-elims*(*1*) *fresh-abst-aa fresh-abst-ab rev-singleton-conv swapa.simps swapas.simps*(*1*,*2*)
by *metis*
next
case (*equ-abst-aa nabla t1 t2 b*)
then show *?case*
using *fresh-swap-left Fresh-elims*(*1*) *fresh-abst-aa fresh-abst-ab*
by *metis*
next
case (*equ-unit nabla*)
then show *?case*
by *blast*
next
case (*equ-atom b c nabla*)
then show *?case*
by *simp*
next
case (*equ-susp pi1 pi2 X nabla*)
then show *?case*
using *Fresh-elims*(*4*) *ds-elem-cp ds-rev fresh-susp swapas-append swapas-rev-pi-a*
by *metis*
next
case (*equ-paar nabla t1 t2 s1 s2*)
then show *?case*
using *Fresh-elims*(*5*) **by** *blast*
next
case (*equ-func nabla t1 t2 f*)
then show *?case*
using *Fresh-elims*(*6*) **by** *blast*
qed

lemma *ds-empty-equiv-1*:
assumes *ds pi1 pi2* = {}
shows *nabla* ⊢ *swap pi1 t1* ≈ *t2* ⟹ *nabla* ⊢ *swap pi2 t1* ≈ *t2*
using *assms*
proof(*induction t1 arbitrary*: *t2*)
case (*Abst a t1'*)
then obtain *b t2'*
where *t2*: *t2* = *Abst b t2'*
by(*auto elim*: *equ.cases*)
with *Abst* **have** *i*: *nabla* ⊢ *Abst* (*swapas pi1 a*) (*swap pi1 t1'*) ≈ *Abst b t2'*
using *swap.simps*(*4*) **by** *simp*
then show *?case*
proof(*cases* ‹*swapas pi1 a* = *b*›)
case *True*
then have *nabla* ⊢ *swap pi1 t1'* ≈ *t2'*

```
      using Abst i Equ-elims(6) by blast
    then have ii: nabla ⊢ swap pi2 t1' ≈ t2'
      using Abst by simp
    then have nabla ⊢ Abst (swapas pi2 a) (swap pi2 t1') ≈ Abst b t2'
      using True ds-swapas-eq[OF Abst(3), of a] equ-abst-aa[OF ii, of ‹swapas pi2
a›]
      by simp
    then show ?thesis
      using t2 by simp
  next
    case False
    then have equ-ab-pi1: nabla ⊢ swap pi1 t1' ≈ swap [(swapas pi1 a, b)] t2'
      and fresh-ab-pi1: nabla ⊢ swapas pi1 a ♯ t2'
      using i Equ-elims(7)[of nabla ‹swapas pi1 a› ‹swap pi1 t1'› b t2']
      by blast+
    have equ-ab-pi2: nabla ⊢ swap pi2 t1' ≈ swap [(swapas pi2 a, b)] t2'
      using equ-ab-pi1 ds-swapas-eq[OF Abst(3), of a]
        Abst(3) Abst(1)[of ‹swap [(swapas pi1 a, b)] t2'›] by auto
    have fresh-ab-pi2: nabla ⊢ swapas pi2 a ♯ t2'
      using fresh-ab-pi1 ds-swapas-eq[OF Abst(3), of a] by simp
    with equ-ab-pi2 have nabla ⊢ Abst (swapas pi2 a) (swap pi2 t1') ≈ Abst b t2'
      using equ-abst-ab[OF False, of nabla t2'] ds-swapas-eq[OF Abst(3), of a] by
simp
    then show ?thesis using t2 by simp
  qed
next
  case (Susp pi X)
  then obtain pi'
    where t2: t2 = Susp pi' X
    by (auto elim:equ.cases)
  with Susp have i: nabla ⊢ Susp (pi1 @ pi) X ≈ Susp pi' X
    by simp
  then have ∀ c ∈ ds (pi1@pi) pi'. (c,X) ∈ nabla
    using Equ-elims(3)[OF i] by auto
  with Susp have ∀ c ∈ ds (pi2@pi) pi'. (c,X) ∈ nabla
    using ds-cancel-pi-left ds-sym ds-trans
    by blast
  then have nabla ⊢ Susp (pi2 @ pi) X ≈ Susp pi' X
    using equ-susp by simp
  then show ?case using t2 swap.simps(3) by simp
next
  case (Atom a)
  then show ?case
    using ds-swapas-eq[OF Atom(2), of a]
    by (auto elim: equ.cases)
qed (auto elim: equ.cases)

lemma ds-empty-equiv-2:
  assumes ds pi1 pi2 = {}
```

```
  shows nabla ⊢ t1 ≈ swap pi1 t2 ⟹ nabla ⊢ t1 ≈ swap pi2 t2
using assms
proof(induction t2 arbitrary: pi1 pi2 t1)
  case (Abst b t2')
  then obtain a t1'
    where t1: t1 = Abst a t1'
    by(auto elim: equ.cases)
  with Abst have i: nabla ⊢ Abst a t1' ≈ Abst (swapas pi1 b) (swap pi1 t2')
    using swap.simps(4) by simp
  then show ?case
  proof(cases ‹swapas pi1 b = a›)
    case True
    then have nabla ⊢ t1' ≈ swap pi1 t2'
      using Abst i Equ-elims(6) by blast
    then have ii: nabla ⊢ t1' ≈ swap pi2 t2'
      using Abst by simp
    then have nabla ⊢ Abst a t1' ≈ Abst (swapas pi2 b) (swap pi2 t2')
      using True ds-swapas-eq[OF Abst(3), of b] equ-abst-aa[OF ii, of ‹swapas pi2
b›]
      by simp
    then show ?thesis
      using t1 by simp
  next
    case False
    then have equ-ab: nabla ⊢ t1' ≈ swap [(a, swapas pi1 b)] (swap pi1 t2')
      and fresh-ab-pi1: nabla ⊢ a ♯ swap pi1 t2'
      using i Equ-elims(7) by blast+

    then have equ-ab-pi1: nabla ⊢ t1' ≈ swap ([(a, swapas pi1 b)]@ pi1) t2'
      using swap-append[of ‹[(a, swapas pi1 b)]› pi1 t2'] by simp

    have ds-empty: ds ([(a, swapas pi1 b)] @ pi1) ([(a, swapas pi2 b)] @ pi2) = {}
      using Abst(3) ds-swapas-eq[OF Abst(3), of b]
        ds-cancel-pi-front[of ‹[(a, swapas pi1 b)]› pi1 pi2] by simp

    hence nabla ⊢ t1' ≈ swap ([(a, swapas pi2 b)]@ pi2) t2'
      using Abst(1)[OF equ-ab-pi1 ds-empty] by simp
    hence equ-ab-pi2: nabla ⊢ t1' ≈ swap [(a, swapas pi2 b)] (swap pi2 t2')
      using swap-append[of ‹[(a, swapas pi2 b)]› pi2 t2'] by simp

    have fresh-ab-pi2: nabla ⊢ a ♯ swap pi2 t2'
      using ds-empty-fresh-2 Abst(3) fresh-ab-pi1 by auto
    with equ-ab-pi2 have nabla ⊢ Abst a t1' ≈ Abst (swapas pi2 b) (swap pi2 t2')
      using equ-abst-ab ds-swapas-eq False assms Abst(3) by auto
    then show ?thesis using t1 by simp
  qed
next
  case (Susp pi' X)
  then obtain pi
```

```
    where t1: t1 = Susp pi X
    by (auto elim:equ.cases)
  with Susp have i: nabla ⊢ Susp pi X ≈ Susp (pi1 @ pi') X
    by simp
  then have ∀ c ∈ ds pi (pi1 @ pi'). (c,X) ∈ nabla
    using Equ-elims(3)[OF i] by auto
  with Susp have ∀ c ∈ ds pi (pi2 @ pi'). (c,X) ∈ nabla
    using ds-cancel-pi-left ds-sym ds-trans
    by blast
  then have nabla ⊢ Susp pi X ≈ Susp (pi2 @ pi') X
    using equ-susp by simp
  then show ?case using t1 swap.simps(3) by simp
next
  case (Atom a)
  then show ?case
    using ds-swapas-eq by (auto elim: equ.cases)
qed (auto elim: equ.cases)

lemma equ-equivariance:
  assumes nabla ⊢ t1 ≈ t2
  shows nabla ⊢ swap pi t1 ≈ swap pi t2
  using assms
proof(induct rule: equ.induct)
  case (equ-abst-ab a b nabla t2' t1')
  have i: nabla ⊢ swapas pi a ♯ swap pi t2'
    using fresh-swap-eqvt[OF equ-abst-ab(2), of pi] by simp

  have nabla ⊢ swap pi t1' ≈ swap (pi @ [(a,b)]) t2'
    using equ-abst-ab(4) swap-append[of pi ‹[(a,b)]› t2'] by simp
  then have nabla ⊢ swap pi t1' ≈ swap ([(swapas pi a, swapas pi b)] @ pi) t2'
    using ds-empty-equiv-2[OF ds-comm[of pi a b]] by auto
  then have ii: nabla ⊢ swap pi t1' ≈ swap [(swapas pi a, swapas pi b)] (swap pi
t2')
    using ds-empty-equiv-2[OF ds-comm[of pi a b]]
      swap-append[of ‹[(swapas pi a, swapas pi b)]› pi t2'] by simp

  have iii: swapas pi a ≠ swapas pi b
    using equ-abst-ab.hyps(1) swapas-rev-pi-a by blast

  with i ii
  have nabla ⊢ Abst (swapas pi a) (swap pi t1') ≈ Abst (swapas pi b) (swap pi t2')
    using equ.equ-abst-ab by auto
  hence nabla ⊢ swap pi (Abst a t1') ≈ swap pi (Abst b t2')
    using swap.simps(4) by simp
  then show ?case by simp
next
  case (equ-abst-aa nabla t1' t2' a)
  then show ?case
    using equ.equ-abst-aa by simp
```

next
case (*equ-susp pi1 pi2 X nabla*)
then show *?case* **using** *ds-cancel-pi-front*
by *auto*
qed (*auto*)

lemma *swap-inv-side*:
shows *nabla* ⊢ *swap pi t1* ≈ *t2* = *nabla* ⊢ *t1* ≈ *swap* (*rev pi*) *t2*
proof

{**assume** *assm1*: *nabla* ⊢ *swap pi t1* ≈ *t2*
from *equ-equivariance*[*OF assm1*, *of* ‹*rev pi*›]
have *nabla* ⊢ *swap* (*rev pi* @ *pi*) *t1* ≈ *swap* (*rev pi*) *t2*
using *swap-append*[*of* ‹*rev pi*› *pi t1*] **by** *simp*
then show *nabla* ⊢ *t1* ≈ *swap* (*rev pi*) *t2*
using *ds-empty-equiv-1*[*OF ds-rev-pi-id*[*of pi*], *of nabla t1* ‹*swap* (*rev pi*) *t2*›]
swap-id[*of t1*] **by** *simp*}

{**assume** *assm2*: *nabla* ⊢ *t1* ≈ *swap* (*rev pi*) *t2*
from *equ-equivariance*[*OF assm2*, *of pi*]
have *nabla* ⊢ *swap pi t1* ≈ *swap* (*pi* @ *rev pi*) *t2*
using *swap-append*[*of pi* ‹*rev pi*› *t2*] **by** *simp*
then show *nabla* ⊢ *swap pi t1* ≈ *t2*
using *ds-empty-equiv-2*[*OF ds-pi-rev-id*[*of pi*], *of nabla* ‹*swap pi t1*› *t2*]
swap-id[*of t2*] **by** *simp*}
qed

lemma *equ-swap-abba*:
assumes *n* = *depth t1*
shows *nabla* ⊢ *swap* [(*a*,*b*)] *t1* ≈ *t2* ⟹ *nabla* ⊢ *t1* ≈ *swap* [(*b*,*a*)] *t2*
proof−
assume *nabla* ⊢ *swap* [(*a*,*b*)] *t1* ≈ *t2*
with *ds-baab*[*of b a*]
have *nabla* ⊢ *swap* [(*b*,*a*)] *t1* ≈ *t2*
using *ds-empty-equiv-1* **by** *blast*
then show *?thesis*
using *swap-inv-side*[*of nabla* ‹[(*b*,*a*)]›] **by** *simp*
qed

lemma *equ-equiv-pi*:
assumes ∀ *a* ∈ *ds pi1 pi2*. *nabla* ⊢ *a* ♯ *t*
shows *nabla* ⊢ *swap pi1 t* ≈ *swap pi2 t*
using *assms equ-pi-right*[*of* ‹*rev pi1* @ *pi2*› *nabla t*]
swap-inv-side[*of nabla pi1 t* ‹*swap pi2 t*›] *ds-rev swap-append* **by** *simp*

lemma *equ-symm*:
shows (*nabla* ⊢ *t1* ≈ *t2*) ⟹ (*nabla* ⊢ *t2* ≈ *t1*)
proof(*induction rule*: *equ.induct*)
case (*equ-abst-ab a b nabla t2′ t1′*)

```
  have i: nabla ⊢ b ♯ swap [(a, b)] t2'
    using fresh-swap-eqvt[of nabla a t2' [(a,b)]] equ-abst-ab(2) by auto
  with equ-abst-ab(4)
  have b-fresh: nabla ⊢ b ♯ t1'
    using l3-jud by blast
  from equ-abst-ab(4)
  have ii: nabla ⊢ t2' ≈ swap [(b, a)] t1'
    using equ-swap-abba by simp
  show ?case
    using equ.equ-abst-ab[OF equ-abst-ab(1)[symmetric] b-fresh ii] by simp
next
  case (equ-abst-aa nabla t1' t2' a)
  then show ?case
    using equ.equ-abst-aa[OF equ-abst-aa(2), of a] by simp
next
  case (equ-unit nabla)
  then show ?case by auto
next
  case (equ-atom a b nabla)
  then show ?case by auto
next
  case (equ-susp pi1 pi2 X nabla)
  then show ?case
    using ds-sym by auto
next
  case (equ-paar nabla t1' t2' s1' s2')
  then show ?case by auto
next
  case (equ-func nabla t1' t2' f)
  then show ?case
    using equ.equ-func[OF equ-func(2), of f] by simp
qed

lemma equ-trans:
  assumes nabla ⊢ t1 ≈ t2 nabla ⊢ t2 ≈ t3
  shows nabla ⊢ t1 ≈ t3
  using assms
proof(induction ‹depth t1› arbitrary: t1 t2 t3 rule: nat-less-induct)
  case 1
  note IH = this
  have IH-usable:
    depth t1' < depth t1 ⟹ nabla ⊢ t1' ≈ t2' ⟹  nabla ⊢ t2' ≈ t3' ⟹ nabla
⊢ t1' ≈ t3'
    for t1' t2' t3' using IH by blast
  have nabla ⊢ t1 ≈ t2 ⟹ nabla ⊢ t2 ≈ t3 ⟹  nabla ⊢ t1 ≈ t3
  proof−
    assume t12: nabla ⊢ t1 ≈ t2 and t23: nabla ⊢ t2 ≈ t3
    show ?thesis
     proof(cases rule: equ.cases[OF ‹nabla ⊢ t1 ≈ t2›])
```

```
case (1 a b nabla t2' t1')
from 1(2) have depth1: depth t1' < depth t1
  using depth.simps(4) by auto
from t23 show ?thesis
proof(cases rule: equ.cases)
  case (equ-abst-ab b c t3' t2')
  then show ?thesis
  proof(cases a = c)
    case True
    have i: nabla ⊢ swap [(c,b)] t2' ≈ swap ([(c,b)]@[(b,c)]) t3'
      using equ-equivariance[OF equ-abst-ab(5), of ‹[(c,b)]›] 1(1)
        swap-append[of ‹[(c,b)]› ‹[(b,c)]› t3'] by simp
    have ii: nabla ⊢ swap [(c,b)] t2' ≈ t3'
      using ds-empty-equiv-2[OF ds-baab-id[OF equ-abst-ab(3)] i] by simp
    with equ-abst-ab 1 True
    have iii: nabla ⊢ t1' ≈ swap [(c, b)] t2' by blast
    with ii have nabla ⊢ t1' ≈ t3'
      using IH-usable[OF depth1, of ‹swap [(c, b)] t2'› t3'] 1(1)
      by simp
    then show ?thesis using True equ-abst-ab(2) 1(1,2) by auto
  next
    case False

    have depth2: depth (swap [(a,b)] t2') < depth t1
      using depth.simps(4) equ-depth[OF 1(6)] swap-depth 1(3) depth1
      equ-abst-ab(1) by simp

    from equ-abst-ab(1,4,5) 1(1,3,5)
      have ii: nabla ⊢ a ♯ swap [(b,c)] t3'
        using l3-jud by simp

    then have a-fresh-t3: nabla ⊢ a ♯ t3'
      using fresh-swap-left[OF ii] False equ-abst-ab 1 by simp
    have b-fresh-t3: nabla ⊢ b ♯ t3'
      using equ-abst-ab(4) 1(1) by simp

    from 1(1) equ-abst-ab(5)
    have nabla ⊢ swap [(a,b)] t2' ≈ swap ([(a,b)]@[(b,c)]) t3'
      using equ-equivariance[OF equ-abst-ab(5)]
     swap-append[of ‹[(a, b)]› ‹[(b,c)]› t3'] by simp
    then have iii: nabla ⊢ swap [(a,b)] t2' ≈ swap ([(a,b),(b,c)]) t3'
      by simp

    have ds [(a,b), (b,c)] [(a,c)] = {a,b}
      using ds-acabbc equ-abst-ab 1 False by simp
    with a-fresh-t3 b-fresh-t3
    have iv: nabla ⊢ swap [(a,b),(b,c)] t3' ≈ swap [(a,c)] t3'
      using equ-equiv-pi by simp
```

```
        with 1(1) iii have nabla ⊢ swap [(a,b)] t2' ≈ swap [(a,c)] t3'
          using IH-usable[OF deptht2, of ‹swap ([(a,b),(b,c)]) t3'› ‹swap [(a,c)]
t3'›]
          by simp

        with 1(1,3,6) have nabla ⊢ t1' ≈ swap [(a,c)] t3'
          using IH-usable[OF deptht1, of ‹swap [(a, b)] t2'› ‹swap [(a,c)] t3'›]
          equ-abst-ab(1) by simp

        with 1(1,2) equ-abst-ab(2)
        show ?thesis using equ.equ-abst-ab[OF False a-fresh-t3] by simp
      qed
    next
      case (equ-abst-aa t2' t3' b)
      from this(3) 1(1)
      have i: nabla ⊢ swap [(a,b)] t2' ≈ swap [(a,b)] t3'
        using equ-equivariance[of nabla t2' t3' ‹[(a,b)]›] by simp

      have nabla ⊢ b ♯ swap [(a,b)] t2'
        using 1(1,3,4,5) equ-abst-aa(1) fresh-swap-eqvt[of nabla a t2' ‹[(a,b)]›]
        by auto
      then have ii: nabla ⊢ b ♯ swap [(a,b)] t3'
        using l3-jud[OF i, of b] by simp
      have nabla ⊢ swapas [(a,b)] b ♯ t3'
        using fresh-swap-left[OF ii] by simp
      then have a-fresh-t3: nabla ⊢ a ♯ t3'
        using 1(3,4) equ-abst-aa(1) by simp

      have is-equ: nabla ⊢ t1' ≈ swap [(a,b)] t3'
        using 1 i IH-usable[OF deptht1, of ‹swap [(a,b)] t2'› ‹swap [(a,b)] t3'›]
        equ-abst-aa(1) by auto

      from a-fresh-t3 is-equ show ?thesis
        using 1 equ-abst-aa(1,2) equ.equ-abst-ab by simp
    qed (auto simp add: 1 t12)
  next
    case (2 nabla t1' t2' a)
    have deptht1: depth t1' < depth t1
      using depth.simps(4) 2(2) by simp
      from t23 show ?thesis
      proof(cases rule: equ.cases)
        case (equ-abst-ab a c t3' t2')
        have nabla ⊢ t1' ≈ swap [(a,c)] t3'
          using 2(1,2,3,4) IH-usable[OF deptht1, of ‹t2'› ‹swap [(a,c)] t3'›]
         equ-abst-ab(1,4,5) by simp
        then show ?thesis
          using equ.equ-abst-ab[OF equ-abst-ab(3)]
            2(1,2,3) equ-abst-ab(1,2,4) by auto
      next
```

```
      case (equ-abst-aa t2' t3' a)
      then show ?thesis
        using 1 2(1,2,3,4) by auto
    qed (auto simp add: 2)
next
  case (3 nabla)
   from t23 show ?thesis
   proof(cases rule: equ.cases)
     case equ-unit
     then show ?thesis using t12 by auto
   qed (auto simp add: 3)
next
  case (4 a b nabla)
   from t23 show ?thesis
   proof(cases rule: equ.cases)
     case (equ-atom a b)
     then show ?thesis
       using t12 by auto
   qed (auto simp add: 4)
next
  case (5 pi1 pi2 X nabla)
   from t23 show ?thesis
   proof(cases rule: equ.cases)
     case (equ-susp pi1 pi2 X)
     then show ?thesis
       using ds-trans 5 by blast
   qed (auto simp add: 5)
next
  case (6 nabla t1' t2' s1' s2')
   from t23 show ?thesis
   proof(cases rule: equ.cases)
     case (equ-paar t2' t3' s2' s3')
     have depth t1': depth t1' < depth t1
       using 6(2) by simp
    have depths1: depth s1' < depth t1
      using 6(2) by simp
    have a: nabla ⊢ t1' ≈ t2'
      using 6(3,4) equ-paar by auto
    have b: nabla ⊢ s1' ≈ s2'
      using 6 equ-paar by auto
    from a have t1-equal-t3: nabla ⊢ t1' ≈ t3'
      using IH-usable[of t1' t2' t3'] equ-paar(3) 6(1) depth t1
      by simp
    from b have s1-equal-s3: nabla ⊢ s1' ≈ s3'
      using IH-usable[of s1' s2' s3'] equ-paar(4) 6(1) depths1
      by simp
    from t1-equal-t3 s1-equal-s3 show ?thesis
      using equ.equ-paar[of nabla t1' t3' s1' s3'] 6(1,2) equ-paar(2)
      by simp
```

```
      qed (auto simp add: 6)
    next
      case (7 nabla t1 t2 F)
       from t23 show ?thesis
      proof(cases rule: equ.cases)
        case (equ-func t1 t2 F)
       then show ?thesis
         using 1 7(1,2,3,4) by auto
      qed (auto simp add: 7)
    qed
  qed
  then show ?case
    using IH(2,3) by blast
qed

lemma pi-right-equ-help:
  assumes (n = depth t)
  shows nabla ⊢ t ≈ swap pi t ⟹ ∀ a ∈ ds [] pi. nabla ⊢ a ♯ t
  using assms
proof(induction n arbitrary: t pi rule: nat-less-induct)
  case (1 n)
  note IH = this
  then show ?case
    proof(cases rule: equ.cases[OF ‹nabla ⊢ t ≈ swap pi t›])
      case (1 b c nabla t2 t1)
        have depth t1: depth t1 < n
          using 1 depth.simps(4) IH by simp
        have swap-pi-t1-is-t2: swap pi t1 = t2
          using 1(2,3) by auto
        have swap-pi-b-is-c: swapas pi b = c
          using 1(2,3) by auto
        with swap-pi-t1-is-t2 have nabla ⊢ t1 ≈ swap [(b, swapas pi b)] (swap pi
t1)
          using 1(1,6) by simp
        then have nabla ⊢ t1 ≈ swap ([(b, swapas pi b)] @ pi) t1
          using swap-append[of ‹[(b, swapas pi b)]› pi t1] by simp
        with depth t1 have ∀ a ∈ ds [] ((b, swapas pi b)#pi). nabla ⊢ a ♯ t1
          using 1.IH 1 by auto
        then have ds-minus-bc: ∀ a ∈ ds [] pi − {b, c}. nabla ⊢ a ♯ t1
          using ds-equality swap-pi-b-is-c by auto
        have c-fresh-t1: nabla ⊢ c ♯ t1
          using 1 equ-symm l3-jud fresh-swap-eqvt Equ-elims equ-abst-ab by meson
        with ds-minus-bc have ∀ a ∈ ds [] pi − {b}. nabla ⊢ a ♯ t1
          by auto
        then have ds-minus-b: ∀ a ∈ ds [] pi − {b}. nabla ⊢ a ♯ Abst b t1
          using fresh-abst-ab by blast
        have b-fresh-abst-t1: nabla ⊢ b ♯ Abst b t1
          using fresh-abst-aa by simp
        have ∀ a ∈ ds [] pi. nabla ⊢ a ♯ Abst b t1
```

```
    proof(rule, goal-cases)
      case (1 a)
      then show ?case
        using b-fresh-abst-t1 ds-minus-b
        by (cases a=b, auto)
    qed
    with 1 show ?thesis
      by auto
next
  case (2 nabla t1 t2 b)
  have depth t1 < n
    using 2 depth.simps(4) IH by simp
  have swap-pi-t1-is-t2: swap pi t1 = t2
    using 2(2,3) by auto
  from swap-pi-t1-is-t2 depth t1
  have ∀ a ∈ ds [] pi. nabla ⊢ a ♯ t1
    using 1.IH 2(1,4) by auto
  then show ?thesis
    using 2(1,2,3) elem-ds by fastforce
next
  case (3 nabla)
  then show ?thesis
    by blast
next
  case (4 a b nabla)
  then show ?thesis
    using elem-ds by force
next
  case (5 pi1 pi2 X nabla)
  have swap pi t = Susp (pi@pi1) X
    using 5(2) by auto
  also have ... = Susp pi2 X
    using 5(3) calculation by simp
  then have pi@pi1 = pi2 by simp
  hence ∀ c∈ds pi1 (pi@pi1). (c, X) ∈ nabla
    using 5(4) by simp
  then show ?thesis
    using 5(1,2) fresh-susp ds-cancel-pi-right[of - - [] -]
    by simp
next
  case (6 nabla t1 t2 s1 s2)
  have depth t1 < n
    using 6 depth.simps(5) IH
    by simp
  have depth s1 < n
    using 6 depth.simps(5) IH
    by simp
  from depth t1 depth s1 show ?thesis
    using 1.IH 6 by auto
```

```
  next
    case (7 nabla t1 t2 f)
    have depptht1: depth t1 < n
      using 7(2) depth.simps(5) IH
      by auto
    have nabla ⊢ t1 ≈ swap pi t1
      using 7 by simp
    then have ∀ a ∈ ds [] pi. nabla ⊢ a ♯ t1
      using 7(1) IH depptht1 by simp
    then show ?thesis
      using 7(1,2) fresh-func[of nabla - t1 f]
      by simp
  qed
qed
```

8 Equality

Proves various facts about the equivalence relation.

```
lemma equ-refl:
  shows nabla ⊢ t ≈ t
  by(induct t, auto simp add: ds-def)


lemma equ-dec-pi:
  assumes nabla ⊢ swap pi t1 ≈ swap pi t2
  shows nabla ⊢ t1 ≈ t2
proof−
  have i: nabla ⊢ swap (rev pi) (swap pi t1) ≈ t1
    nabla ⊢ swap (rev pi) (swap pi t2) ≈ t2
    using rev-pi-pi-equ by auto
  then have nabla ⊢ swap (rev pi) (swap pi t1) ≈ swap (rev pi) (swap pi t2)
    using equ-equivariance assms by simp
  then show ?thesis using i equ-symm equ-trans by meson
qed

lemma equ-involutive-left:
  shows nabla ⊢ swap (rev pi) (swap pi t1) ≈ t2 = nabla ⊢ t1 ≈ t2
proof(auto)
  have i: nabla ⊢ t1 ≈ swap (rev pi) (swap pi t1)
    using rev-pi-pi-equ equ-symm by blast
  show nabla ⊢ swap (rev pi) (swap pi t1) ≈ t2 ⟹ nabla ⊢ t1 ≈ t2
    using i equ-trans by blast
  show nabla ⊢ t1 ≈ t2 ⟹ nabla ⊢ swap (rev pi) (swap pi t1) ≈ t2
    using i equ-trans equ-symm by blast
qed

lemma equ-pi-to-left:
```

shows *nabla* ⊢ *swap* (*rev pi*) *t1* ≈ *t2* = *nabla* ⊢ *t1* ≈ *swap pi t2*
proof

{assume *i*: *nabla* ⊢ *swap* (*rev pi*) *t1* ≈ *t2*
have *nabla* ⊢ *swap pi* (*swap* (*rev pi*) *t1*) ≈ *swap pi t2*
using *equ-equivariance*[*OF i*, *of pi*] **by** *simp*
then show *nabla* ⊢ *t1* ≈ *swap pi t2*
using *equ-involutive-left*[*of nabla* ‹*rev pi*› *t1* ‹*swap pi t2*›] *rev-rev-ident*[*of pi*]
by *simp***}**

{assume *i*: *nabla* ⊢ *t1* ≈ *swap pi t2*
have *ii*: *nabla* ⊢ *swap* (*rev pi*) *t1* ≈ *swap* (*rev pi*) (*swap pi t2*)
using *equ-equivariance*[*OF i*, *of* ‹*rev pi*›] **by** *simp*
then have *iii*: *nabla* ⊢ *swap* (*rev pi*) (*swap pi t2*) ≈ *swap* (*rev pi*) *t1*
using *equ-symm*[*OF ii*] **by** *simp*
then have *iv*: *nabla* ⊢ *t2* ≈ *swap* (*rev pi*) *t1*
using *equ-involutive-left*[*of nabla pi t2* ‹*swap* (*rev pi*) *t1*›] **by** *simp*
then show *nabla* ⊢ *swap* (*rev pi*) *t1* ≈ *t2*
using *equ-symm*[*OF iv*] **by** *simp***}**

qed

lemma *equ-pi-to-right*:
shows *nabla*⊢*t1* ≈ *swap* (*rev pi*) *t2* = *nabla*⊢*swap pi t1*≈*t2*
proof
{assume *i*: *nabla* ⊢ *t1* ≈ *swap* (*rev pi*) *t2*
then show *nabla* ⊢ *swap pi t1* ≈ *t2*
using *equ-involutive-left equ-dec-pi* **by** *blast***}**
{assume *ii*: *nabla* ⊢ *swap pi t1* ≈ *t2*
then show *nabla* ⊢ *t1* ≈ *swap* (*rev pi*) *t2*
using *equ-involutive-left equ-equivariance* **by** *blast***}**
qed

lemma *equ-involutive-right*:
shows *nabla* ⊢ *t1* ≈ *swap* (*rev pi*) (*swap pi t2*) = *nabla* ⊢ *t1* ≈ *t2*
using *equ-dec-pi*[*of nabla pi t1 t2*] *equ-equivariance*[*of nabla t1 t2 pi*]
equ-pi-to-right[*of nabla t1 pi* ‹*swap pi t2*›]
by *auto*

lemma *equ-pi1-pi2-add*:
assumes ∀ *a*∈ *ds pi1 pi2*. *nabla*⊢*a*♯*t*
shows *nabla*⊢*swap pi1 t* ≈ *swap pi2 t*
proof−
have *nabla* ⊢ *t* ≈ *swap* (*rev pi1* @ *pi2*) *t*
using *assms ds-rev equ-pi-right* **by** *simp*
hence *nabla* ⊢ *t* ≈ *swap* (*rev pi1*) (*swap pi2 t*)
using *swap-append* **by** *auto*
then show *nabla*⊢*swap pi1 t* ≈ *swap pi2 t*
using *equ-pi-to-right* **by** *simp*

qed

lemma *pi-right-equ*:
assumes *nabla* ⊢ *t* ≈ *swap pi t*
shows ∀ *a*∈ *ds* [] *pi*. *nabla* ⊢ *a* ♯ *t*
using *assms pi-right-equ-help* **by** *blast*

lemma *equ-pi1-pi2-dec*:
assumes *nabla* ⊢ *swap pi1 t* ≈ *swap pi2 t*
shows ∀ *a* ∈ *ds pi1 pi2*. *nabla*⊢*a* ♯ *t*
proof−
have *nabla* ⊢ *t* ≈ *swap* ((*rev pi1*) @ *pi2*) *t*
using *assms equ-pi-to-right swap-append* **by** *simp*
then show ∀ *a* ∈ *ds pi1 pi2*. *nabla*⊢*a* ♯ *t*
using *pi-right-equ ds-rev*[*of pi1 pi2*] **by** *simp*
qed

lemma *equ-weak*:
assumes *nabla1* ⊢ *t1* ≈ *t2*
shows (*nabla1* ∪ *nabla2*) ⊢ *t1* ≈ *t2*
using *assms* **by** (*erule-tac equ.induct*) (*auto simp add*: *fresh-weak*)

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
fix *t2*
assume *i*: *t2* ∈ *psub-trms t1*
show ¬ (∃ *pi*. *nabla* ⊢ *t1* ≈ *swap pi t2*)
proof
assume ∃ *pi*. *nabla* ⊢ *t1* ≈ *swap pi t2*
then obtain *pi* **where** *H*:
nabla ⊢ *t1* ≈ *swap pi t2* **by** *blast*

from *equ-depth*[*OF H*]
have *depth t1* = *depth* (*swap pi t2*)
by *simp*
hence *depth t1* = *depth t2*
by *simp*
moreover have *depth t2* < *depth t1*
using *i depth-psub-trms*
by *simp*

ultimately show *False* **by** *auto*
qed
qed

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* $\models$ *s1* $\approx$ *s2*
shows *nabla* $\models$ *s2* $\approx$ *s1*
using *assms equ-symm subst-equ-def*[*of nabla s1 s2*]
subst-equ-def[*of nabla s2 s1*] *Un-commute*[*of* ‹*domn s1*› ‹*domn s2*›] **by** *blast*

lemma *subst-equ-refl*:
shows *nabla* $\models$ *s* $\approx$ *s*
using *equ-refl subst-equ-def Un-absorb* **by** *simp*

lemma *not-in-domn*:
assumes *X* $\notin$ (*domn s*)
shows *s* (*Susp* [] *X*) = (*Susp* [] *X*)
using *assms domn-def* **by** *simp*

lemma *subst-swap-comm*:
shows *subst s* (*swap pi t*) = *swap pi* (*subst s t*)
using *swap-append subst-susp* **by** (*induct t*) *auto*

lemma *subst-not-occurs*:
assumes ¬(*occurs X t*)
shows *subst* [(*X*,*t2*)] *t* = *t*
proof−
have *i*: ¬(*occurs X t*) ⟶ *subst* [(*X*,*t2*)] *t* = *t*
using *subst-susp* **by** (*induct t*) *auto*
then show *?thesis* **using** *assms* **by** *simp*
qed

lemma *id-subst* [*simp*]:
shows *subst* [] *t* = *t*
using *subst-susp* **by** (*induct t*) *auto*

lemma *subst-comp-id* [*simp*]:
shows *subst* (*s* • []) = *subst s*
using *comp-def* **by** *simp*

lemma *id-comp-subst* [*simp*]:
shows *subst* ([] • *s*) = *subst s*
proof(*rule ext*)
fix *t*
show *subst* ([] • *s*) *t* = *subst s t*
proof(*induction t arbitrary*: *s*)
case (*Susp pi X*)
then show *?case* **using** *comp-def subst-susp*
proof (*induction s*)
case *Nil*
then show *?case*
using *comp-def subst-susp* **by** *simp*
next
case (*Cons a s*)

```
    then show ?case
      using comp-def subst-susp by simp
   qed
  qed (simp-all)
qed

lemma subst-comp-expand:
  shows subst (s1 · s2) t = subst s1 (subst s2 t)
proof(induct t)
  case (Susp x1 x2)
  then show ?case
   proof(induct s2)
     case Nil
     then show ?case using comp-def subst-susp subst-swap-comm by simp
   next
    case (Cons a s2)
    then show ?case using comp-def subst-susp subst-swap-comm by simp
  qed
qed (simp-all)

lemma subst-assoc:
  shows subst (s1 · (s2 · s3)) = subst ((s1 · s2) · s3)
proof(rule ext)
  fix t
  show subst (s1 · (s2 · s3)) t = subst (s1 · s2 · s3) t
  proof(induct t)
    case (Susp pi X)
    then show ?case using subst-comp-expand by simp
  qed (simp-all)
qed

lemma fresh-subst:
  assumes nabla1 ⊢ a ♯ t
    and nabla2 ⊨ (subst s) nabla1
  shows nabla2 ⊢ a ♯ subst s t
  using assms
proof(induction rule: fresh.induct)
  case (fresh-susp pi a X nabla)
  then show ?case
    using ext-subst-def subst-susp fresh-swap-right
    by auto
qed (auto)

lemma equ-subst:
  assumes nabla1⊢ t1 ≈t2
    and nabla2 ⊨ (subst s) nabla1
  shows nabla2 ⊢(subst s t1)≈(subst s t2)
  using assms
proof(induction rule: equ.induct)
```

case (*equ-abst-ab a b nabla t2 t1*)
then show *?case*
using *subst-swap-comm fresh-subst equ.equ-abst-ab* **by** *simp*
next
case (*equ-susp pi1 pi2 X nabla*)
then show *?case*
using *subst-susp equ-pi1-pi2-add ext-subst-def subst-susp*
by *fastforce*
qed (*auto*)

lemma *unif-1*:
assumes *nabla* ⊢ *subst s* (*Susp pi X*) ≈ *subst s t*
shows *nabla* ⊨ *subst* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)]) ≈ *subst s*
using *assms*
apply(*simp only*: *subst-equ-def*)
proof
fix *Xa*
assume *hyps*: *nabla* ⊢ *subst s* (*Susp pi X*) ≈ *subst s t*
Xa ∈ *domn* (*subst* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)])) ∪ *domn* (*subst s*)
show *nabla* ⊢ *subst* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)]) (*Susp* [] *Xa*) ≈ *subst s* (*Susp* [] *Xa*)
proof−
have *one*:
nabla ⊢ *subst* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)]) (*Susp pi Xa*) ≈
subst s (*subst* [(*X*, *swap* (*rev pi*) *t*)] (*Susp pi Xa*))
using *subst-comp-expand equ-refl* **by** *simp*
have *two*:
nabla ⊢ *subst s* (*subst* [(*X*, *swap* (*rev pi*) *t*)] (*Susp pi Xa*)) ≈
subst s (*swap pi* (*look-up Xa* [(*X*, *swap* (*rev pi*) *t*)]))
using *equ-refl subst-susp*[*of* ‹[(*X*, *swap* (*rev pi*)*t*)]› *pi Xa*]
by *simp*
have *nabla* ⊢ *subst* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)]) (*Susp pi Xa*) ≈
subst s (*swap pi* (*look-up Xa* [(*X*, *swap* (*rev pi*) *t*)]))
using *equ-trans*[*OF one two*] **by** *simp*
hence *nabla* ⊢ *swap pi* (*look-up Xa* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)])) ≈
subst s (*swap pi* (*look-up Xa* [(*X*, *swap* (*rev pi*) *t*)]))
using *subst-susp*[*of* ‹(*s* ∙ [(*X*, *swap* (*rev pi*) *t*)])›] **by** *simp*
hence *swap-pi-outside*: *nabla* ⊢ *swap pi* (*look-up Xa* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)])) ≈
swap pi (*subst s* (*look-up Xa* [(*X*, *swap* (*rev pi*) *t*)]))
using *subst-swap-comm* **by** *simp*
hence *nabla* ⊢ (*look-up Xa* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)])) ≈
(*subst s* (*look-up Xa* [(*X*, *swap* (*rev pi*) *t*)]))
using *equ-dec-pi*[*OF swap-pi-outside*] **by** *simp*
hence *three*:
nabla ⊢ *subst* (*s* ∙ [(*X*, *swap* (*rev pi*) *t*)]) (*Susp* [] *Xa*) ≈
(*subst s* (*look-up Xa* [(*X*, *swap* (*rev pi*) *t*)]))
using *subst-susp* **by** *simp*
then show *?thesis*

```
      proof(cases Xa = X)
        case True
       hence subst s (look-up Xa [(X, swap (rev pi) t)]) = subst s (swap (rev pi) t)
          by simp
        with three have
            nabla ⊢ subst (s • [(X, swap (rev pi) t)]) (Susp [] Xa) ≈ swap (rev pi)
(subst s t)
          using subst-swap-comm by auto
        hence i: nabla ⊢ swap pi (subst (s • [(X, swap (rev pi) t)]) (Susp [] Xa))
          ≈ subst s t using swap-inv-side by simp
        hence nabla ⊢ swap pi (subst (s • [(X, swap (rev pi) t)]) (Susp [] Xa))
          ≈ subst s (Susp pi Xa)
          using True equ-trans[OF i equ-symm[OF assms(1)]] by simp
        hence nabla ⊢ swap pi (subst (s • [(X, swap (rev pi) t)]) (Susp [] Xa))
          ≈ swap pi (look-up Xa s)
          using subst-susp by simp
        then show nabla ⊢ (subst (s • [(X, swap (rev pi) t)]) (Susp [] Xa))
          ≈ subst s (Susp [] Xa)
          using equ-dec-pi subst-susp by simp
      next
        case False
        hence subst s (look-up Xa [(X, swap (rev pi) t)]) = subst s (Susp [] Xa)
          by simp
        with three show ?thesis by auto
      qed
    qed
  qed

lemma subst-equ-a:
  assumes nabla ⊨ (subst s1) ≈ (subst s2)
    and nabla ⊢ (subst s2 t1) ≈ t2
  shows nabla ⊢ (subst s1 t1) ≈ t2
  using assms
proof(induct t1 arbitrary: t2)
  case (Abst a t1')
  then obtain t2' b
    where t2: t2 = Abst b t2'
    by(auto elim: equ.cases)
  with Abst(3) have
    i: nabla ⊢ Abst a (subst s2 t1') ≈ Abst b t2'
    using subst-abst by simp
  then show ?case
  proof(cases a=b)
    case True
    hence nabla ⊢ (subst s2 t1') ≈ t2'
      using Equ-elims(7)[OF i] by auto
    with Abst(1)[OF Abst(2), of t2']
    have nabla ⊢ subst s1 t1' ≈ t2'
      by simp
```

```
    hence nabla ⊢ Abst a (subst s1 t1') ≈ Abst b t2'
      using True equ-abst-aa by simp
    with t2 show ?thesis
      using subst-abst[of s1 a t1'] by simp
  next
    case False
    hence nabla ⊢ (subst s2 t1') ≈ swap [(a,b)] t2'
        and fresh: nabla ⊢ a ♯ t2'
      using Equ-elims(7)[OF i] by auto
     with Abst(1)[OF Abst(2), of ‹swap [(a,b)] t2'›]
    have nabla ⊢ subst s1 t1' ≈  swap [(a,b)] t2'
      by simp
    hence nabla ⊢ Abst a (subst s1 t1') ≈ Abst b t2'
      using equ-abst-ab[OF False fresh] by simp
    with t2 show ?thesis
      using subst-abst[of s1 a t1'] by simp
  qed
next
  case (Susp pi X)
  hence nabla ⊢ swap pi (look-up X s2) ≈ t2
    using subst-susp by simp
  hence nabla ⊢ (look-up X s2) ≈ swap (rev pi) t2
    using subst-susp swap-inv-side by simp
  hence i: nabla ⊢ subst s2 (Susp [] X) ≈ swap (rev pi) t2
    using subst-susp[of s2 ‹[]›] by simp

  with Susp(1) subst-equ-def[of nabla ‹subst s1› ‹subst s2›]
  have nabla ⊢ subst s1 (Susp [] X) ≈ subst s2 (Susp [] X)
    using UnCI equ-refl not-in-domn by metis

  with i have nabla ⊢ subst s1 (Susp [] X) ≈ swap (rev pi) t2
    using equ-trans by blast
  hence nabla ⊢ (look-up X s1) ≈ swap (rev pi) t2
    using subst-susp by simp
  hence nabla ⊢ swap pi (look-up X s1) ≈  t2
    using swap-inv-side by simp
    then show ?case
      using subst-susp by simp
next
  case (Paar t11 t12)
  then obtain t21 t22
    where t2: t2 = Paar t21 t22
    by(auto elim: equ.cases)
  with Paar(4)
  have paar-i: nabla ⊢ subst s2 t11 ≈ t21
    and paar-ii: nabla ⊢ subst s2 t12 ≈ t22
    by (auto elim: equ.cases)
  from t2 show ?case
   using equ-paar[OF Paar(1)[OF Paar(3) paar-i] Paar(2)[OF Paar(3) paar-ii]]
```

```
    subst-paar by simp
next
  case (Func f t1')
  then obtain t2'
    where t2: t2 = Func f t2'
    by(auto elim: equ.cases)
   with Func(3)
   have func: nabla ⊢ subst s2 t1' ≈ t2'
   by (auto elim: equ.cases)
  from t2 show ?case
    using equ-func[OF Func(1)[OF Func(2) func], of f] by simp
qed (simp-all)

lemma unif-2a:
  assumes nabla⊨subst s1≈subst s2
  and (nabla⊢subst s2 t1 ≈ subst s2 t2)
shows(nabla⊢subst s1 t1 ≈ subst s1 t2)
proof−
  have i: (nabla⊢subst s1 t1 ≈ subst s2 t2)
    using subst-equ-a[OF assms(1,2)] by simp
  show (nabla⊢subst s1 t1 ≈ subst s1 t2)
    using subst-equ-a[OF assms(1) equ-symm[OF i]] equ-symm by auto
qed

lemma unif-2b:
  assumes nabla ⊨ subst s1 ≈ subst s2
  and nabla ⊢ a ♯ subst s2 t
shows nabla ⊢ a ♯ subst s1 t
  using assms
proof(induct t)
  case (Abst b t')
  then show ?case
  proof(cases a=b)
    case True
    then show ?thesis
      using fresh-abst-aa by simp
  next
    case False
    with Abst show ?thesis
      using Fresh-elims(1) subst-abst fresh-abst-ab
      by metis
  qed
next
  case (Susp pi X)
  then show ?case
    using equ-refl equ-symm l3-jud subst-equ-a by meson
next
  case (Paar t1 t2)
  then show ?case
```

using *Fresh-elims*(*5*) *subst-paar fresh-paar*
by *metis*
next
case (*Func f t'*)
then show *?case*
using *Fresh-elims*(*6*) *subst-func fresh-func*
by *metis*
qed (*simp-all*)

lemma *subst-equ-to-trm*:
assumes *nabla* ⊨ *subst s1* ≈ *subst s2*
shows *nabla*⊢*subst s1 t* ≈ *subst s2 t*
using *assms equ-refl subst-equ-a* **by** *simp*

lemma *subst-cancel-right*:
assumes *nabla*⊨(*subst s1*)≈(*subst s2*)
shows *nabla*⊨(*subst* (*s1*•*s*))≈(*subst* (*s2*•*s*))
using *assms subst-comp-expand subst-equ-def subst-equ-to-trm*
by *simp*

lemma *subst-trans*:
assumes *nabla*⊨*subst s1* ≈ *subst s2 nabla* ⊨ *subst s2* ≈ *subst s3*
shows *nabla*⊨*subst s1*≈*subst s3*
using *assms equ-trans subst-equ-def subst-equ-to-trm* **by** *meson*

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*)))
using *assms*
proof(*induct t1*)
case (*Susp pi' X*)
then show *?case*
using *equ-pi-to-left equ-involutive-left*
equ-refl subst-susp swap-append t-sub-trms-t
occurs.simps(*3*) **by** *metis*
next
case (*Paar t11 t12*)
then show *?case*
using *subst-paar Un-iff occurs.simps*(*5*) *psub-sub-trms*
psub-trms.simps(*4*) **by** (*metis*)
qed(*auto*)

lemma *ext-subst-strong*:
assumes *nabla1* ⊨ (*subst s*) (*nabla2* ∪ *nabla3*)
shows *nabla1* ⊨ (*subst s*) *nabla2* **and** *nabla1* ⊨ (*subst s*) *nabla3*
using *assms*
unfolding *ext-subst-def* **by** *auto*

lemma *ext-subst-id*:
shows *nabla* $\models$ (*subst* []) *nabla*
unfolding *ext-subst-def id-subst* **by** *auto*

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 ≈*?* *t2* ⇀ (*t1*, *t2*)
a ♯*?* *t* ⇀ (*a*,*t*)

All solutions for a unification problem.

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

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

definition *U*:: *problem-type* ⇒ (*unifier-type set*)
where *all-solutions-def*:
U P ≡ {(*nabla*, *s*).
(∀ (*t1*,*t2*) ∈ *set* (*fst P*). *nabla* ⊢ *subst s t1* ≈ *subst s t2*) ∧
(∀ (*a*,*t*) ∈ *set* (*snd P*). *nabla* ⊢ *a* ♯ *subst s t*)}

The set of variables in unification problems.

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

fun *vars-fprobs*:: ((*string* × *trm*) *list*) ⇒ (*string set*)
where
vars-fprobs [] = {} |
vars-fprobs (*x*#*xs*) = (*vars-trm* (*snd x*))∪(*vars-fprobs xs*)

fun *vars-eprobs* :: ((*trm* × *trm*) *list*) ⇒ (*string set*)
where
vars-eprobs [] = {} |
vars-eprobs (*x*#*xs*) = (*vars-trm* (*snd x*))∪(*vars-trm* (*fst x*))∪(*vars-eprobs xs*)

definition *vars-probs* :: *problem-type* ⇒ *nat*
where *vars-probs P* ≡ *card*((*vars-fprobs* (*snd P*))∪(*vars-eprobs* (*fst P*)))

Most general unifier

definition *mgu* :: *problem-type* ⇒ *unifier-type* ⇒ *bool*
where *mgu P unif* ≡
∀ (*nabla*,*s1*)∈ *U P*. (∃ *s2*. (*nabla* ⊨ (*subst s2*) (*fst unif*)) ∧
(*nabla* ⊨ *subst* (*s2* ∘(*snd unif*)) ≈ *subst s1*))

Idempotency of a unifier

definition *idem* :: *unifier-type* ⇒ *bool*
where *idem unif* ≡ (*fst unif*)⊨ *subst* ((*snd unif*)∘(*snd unif*)) ≈ *subst* (*snd unif*)

Application of a substitution to a problem

definition *apply-subst-eprobs* :: *substs* ⇒ *eprobs* ⇒ *eprobs*
where *apply-subst-eprobs s P* ≡ *map* (λ(*t1*, *t2*). (*subst s t1* ≈*? 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*
  using *assms*
proof(*cases rule*: *s-red.cases*)
  case (*var-1-sred X pi*)
  have ((*t2*, *Susp pi X*) # *xs*,
    ys) ⊢ [(*X*, *swap* (*rev pi*) *t2*)] ⇝ *apply-subst* [(*X*, *swap* (*rev pi*) *t2*)] (*xs*, *ys*)
    using *var-2-sred*[*OF* ‹¬ *occurs X t2*›] **by** *simp*
  then show *?thesis* **using** *var-1-sred* **by** *blast*
next
  case (*var-2-sred X pi*)
  have ((*Susp pi X*, *t1*) # *xs*,
    ys) ⊢ [(*X*, *swap* (*rev pi*) *t1*)] ⇝ *apply-subst* [(*X*, *swap* (*rev pi*) *t1*)] (*xs*, *ys*)
    using *var-1-sred*[*OF* ‹¬ *occurs X t1*›] **by** *simp*
  then show *?thesis* **using** *var-2-sred* **by** *blast*
qed (*auto*)

Weakening of freshness

lemma *solution-weak*:
  assumes (*nabla1*, *s*) ∈ *U P*
  shows (*nabla1* ∪ *nabla3*, *s*) ∈ *U P*
  using *assms*
  unfolding *all-solutions-def* **using** *fresh-weak equ-weak* **by** *auto*

lemma *solution-comp-id*:
  shows ((*nabla*, *s*) ∈ *U P*) = ((*nabla*, *s* · []) ∈ *U P*) **and**
    ((*nabla*, *s*) ∈ *U P*) = ((*nabla*, [] · *s*) ∈ *U P*)
  unfolding *all-solutions-def* **by** *auto*

lemma *solutions-subst-assoc*:
   ((*nabla*, *s1* · (*s2* · *s3*)) ∈ *U P*) = ((*nabla*, (*s1* · *s2*) · *s3*) ∈ *U P*)
  unfolding *all-solutions-def* **using** *subst-assoc* **by** *simp*

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* = []
using *assms* **by** (*auto elim*: *c-red.cases*)

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*
using *assms*
proof (*induction arbitrary*: *nabla s rule*: *red-plus.induct*[*OF assms*(*2*)])
case (*1 P1 s1 P*)
hence (((*t1*, *t2*) # *xs*, *ys*)) ⊢ *s1* ⇝ *P* **by** *simp*
hence ∃ *P3*. ((*t2* ≈? *t1*) # *xs*, *ys*) ⊢ *s1* ⇝ *P3*
using *sred-symm* **by** *simp*
hence ∃ *P3*. ((*t2* ≈? *t1*) # *xs*, *ys*) ⊨ ({}, *s1*) ⇒ *P3*
using *sred-single* **by** *auto*
then show ∃*nabla1 s1 P3*. ((*t2*, *t1*) # *xs*, *ys*) ⊨ (*nabla1*, *s1*) ⇒ *P3* **by** *auto*
next
case (*2 P1 nabla1 P2*)
hence *fst P1* ≠ [] **by** *simp*
moreover from *2*(*1*) **have** *fst P1* = []
using *c-red-eqs-empty* **by** *simp*
ultimately have *False* **by** *simp*
then show *?case* **by** *simp*
next
case (*3 P1 s1 P' nabla2 s2 P3*)
hence (((*t1*, *t2*) # *xs*, *ys*)) ⊢ *s1* ⇝ *P'* **by** *simp*
hence ∃ *P3*. ((*t2* ≈? *t1*) # *xs*, *ys*) ⊢ *s1* ⇝ *P3*
using *sred-symm* **by** *simp*
hence ∃ *P3*. ((*t2* ≈? *t1*) # *xs*, *ys*) ⊨ ({}, *s1*) ⇒ *P3*
using *sred-single* **by** *auto*

then show $\exists$ *nabla1 s1 P3*. ((*t2*, *t1*) # *xs*, *ys*) $\models$ (*nabla1*, *s1*) $\Rightarrow$ *P3* **by** *auto*
next
case (*4 P1 nabla1 P' nabla2 P3*)
hence *fst P1* $\neq$ [] **by** *simp*
moreover from *4*(*1*) **have** *fst P1* = []
using *c-red-eqs-empty* **by** *simp*
ultimately have *False* **by** *simp*
then show *?case* **by** *simp*
qed

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*)
using *assms mgu-def idem-def* **by** *auto*

lemma *problem-subst-comm*:
shows ((*nabla*,*s2*) $\in$ *U* (*apply-subst s1 P*)) = ((*nabla*,(*s2*$\cdot$*s1*))$\in$*U P*)
using *all-solutions-def apply-subst-def subst-comp-expand* **by** *auto*

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*)
using *assms*
proof(*induction arbitrary*: *nabla1 s1 rule*: *s-red.induct*[*OF* ‹*P1* $\vdash$ *s2* $\rightsquigarrow$ *P2*›])
case (*1 xs ys*)
then have *subst*: *nabla1* $\models$ *subst* (*s1* $\cdot$ []) $\approx$ *subst s1*
using *equ-refl subst-equ-def* **by** *simp*
from *1*
have *eqs*: $\forall$ (*t1*,*t2*) $\in$ *set xs*. *nabla1* $\vdash$ *subst s1 t1* $\approx$ *subst s1 t2*
and *fresh*: $\forall$ (*a*,*t*) $\in$ *set ys*. *nabla1* $\vdash$ *a* $\sharp$ *subst s1 t*
using *all-solutions-def* **by** *simp-all*
with *subst* **show** *?case*
using *all-solutions-def* **by** *auto*
next
case (*2 t1 t2 t3 t4 xs ys*)
then have *subst*: *nabla1* $\models$ *subst* (*s1*$\cdot$ []) $\approx$ *subst s1*
using *equ-refl subst-equ-def* **by** *simp*

from *2*
have *fresh*: $\forall$ (*a*,*t*) $\in$ *set ys*. *nabla1* $\vdash$ *a* $\sharp$ *subst s1 t*
using *all-solutions-def* **by** *simp*

from *2*
have *i*: $\forall$ (*t1*,*t2*) $\in$ *set* ((*Paar t1 t2*, *Paar t3 t4*) # *xs*). *nabla1* $\vdash$ *subst s1 t1* $\approx$ *subst s1 t2*

```
    using all-solutions-def by simp
  hence nabla1 ⊢ subst s1 t1 ≈ subst s1 t3
   nabla1 ⊢ subst s1 t2 ≈ subst s1 t4
    using Equ-elims(4)[of nabla1 ‹(subst s1 t1)› ‹(subst s1 t2)› ‹(subst s1 t3)›
‹(subst s1 t4)›]
     subst-paar[of s1] by auto+
   with i have eqs:
   ∀ (t1,t2) ∈ set ((t1,t3)#(t2, t4) # xs). nabla1 ⊢ subst s1 t1 ≈ subst s1 t2
    by auto

   from fresh eqs subst
   show ?case using all-solutions-def by simp
 next
  case (3 F t1 t2 xs ys)
  then have subst: nabla1 ⊨ subst (s1 • []) ≈ subst s1
    using equ-refl subst-equ-def by simp

  from 3
  have fresh: ∀ (a,t) ∈ set ys. nabla1 ⊢ a ♯ subst s1 t
    using all-solutions-def by simp

  from 3
  have i: ∀ (t1,t2) ∈ set ((Func F t1, Func F t2) # xs). nabla1 ⊢ subst s1 t1 ≈
subst s1 t2
    using all-solutions-def by simp
  hence nabla1 ⊢ subst s1 t1 ≈ subst s1 t2
    using Equ-elims(5)[of nabla1 F ‹subst s1 t1› ‹subst s1 t2›]
     subst-func by auto
  with i have eqs:
    ∀ (t1,t2) ∈ set ((t1, t2) # xs). nabla1 ⊢ subst s1 t1 ≈ subst s1 t2
    by auto

  from subst fresh eqs
  show ?case using all-solutions-def by simp
 next
  case (4 a t1 t2 xs ys)
  then have subst: nabla1 ⊨ subst (s1 • []) ≈ subst s1
    using equ-refl subst-equ-def by simp

  from 4
  have fresh: ∀ (a,t) ∈ set ys. nabla1 ⊢ a ♯ subst s1 t
    using all-solutions-def by simp

  from 4
  have i: ∀ (t1,t2) ∈ set ((Abst a t1, Abst a t2) # xs). nabla1 ⊢ subst s1 t1 ≈
subst s1 t2
    using all-solutions-def by simp
  hence nabla1 ⊢ subst s1 t1 ≈ subst s1 t2
    using Equ-elims(6)[of nabla1 a ‹subst s1 t1› ‹subst s1 t2›]
```

subst-abst **by** *auto*
with *i* **have** *eqs*:
∀ (*t1*,*t2*) ∈ *set* ((*t1*, *t2*) # *xs*). *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
by *auto*

from *subst fresh eqs*
show *?case* **using** *all-solutions-def* **by** *simp*
next
case (*5 a b t1 t2 xs ys*)
then have *subst*: *nabla1* ⊨ *subst* (*s1* • []) ≈ *subst s1*
using *equ-refl subst-equ-def* **by** *simp*

from *5*
have *eqsP1*: ∀ (*t1*,*t2*) ∈ *set* ((*Abst a t1*, *Abst b t2*) # *xs*). *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *simp*
hence *i*: *nabla1* ⊢ *Abst a* (*subst s1 t1*) ≈ *Abst b* (*subst s1 t2*)
using *subst-abst*[*of s1*] **by** *simp*

from *5*
have *ii*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst s1 t*
using *all-solutions-def* **by** *auto*
from *5 i*
have *nabla1* ⊢ *a* ♯ *subst s1 t2*
using *Equ-elims*(*7*) **by** *blast*
with *ii*
have *fresh*: ∀ (*a*,*t*) ∈ *set* ((*a*,*t2*) # *ys*). *nabla1* ⊢ *a* ♯ *subst s1 t*
by *simp*

from *i 5*
have *nabla1* ⊢ *subst s1 t1* ≈ *subst s1* (*swap* [(*a*, *b*)] *t2*)
using *5 Equ-elims*(*7*)[*of nabla1 a* ‹*subst s1 t1*› *b* ‹*subst s1 t2*›]
subst-swap-comm **by** *simp*
with *eqsP1* **have** *eqs*:
∀ (*t1*,*t2*) ∈ *set* ((*t1*, *swap* [(*a*,*b*)] *t2*) # *xs*). *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
by *auto*

from *subst fresh eqs*
show *?case* **using** *all-solutions-def* **by** *simp*
next
case (*6 a xs ys*)
then have *subst*: *nabla1* ⊨ *subst* (*s1* • []) ≈ *subst s1*
using *equ-refl subst-equ-def* **by** *simp*

from *6*
have *fresh*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst s1 t*
using *all-solutions-def* **by** *auto*

from *6*

```
  have eqs: ∀ (t1,t2) ∈ set xs. nabla1 ⊢ subst s1 t1 ≈ subst s1 t2
    using all-solutions-def by auto

  from subst fresh eqs
  show ?case using all-solutions-def by simp
next
  case (7 pi1 X pi2 xs ys)
  then have subst: nabla1 ⊨ subst (s1 • []) ≈ subst s1
    using equ-refl subst-equ-def by simp

  from 7
  have i: ∀ (t1,t2) ∈ set ((Susp pi1 X, Susp pi2 X) # xs). nabla1 ⊢ subst s1 t1
≈ subst s1 t2
     ∀ (a,t) ∈ set ys. nabla1 ⊢ a ♯ subst s1 t
    using all-solutions-def by simp+


  from 7(1)
  have nabla1 ⊢ swap pi1 (look-up X s1) ≈ swap pi2 (look-up X s1)
    using all-solutions-def subst-susp by simp
  hence ∀ a∈ds pi1 pi2.  nabla1 ⊢ a ♯ (look-up X s1)
    using equ-pi1-pi2-dec by simp
  hence ∀ a∈ set (ds-list pi1 pi2). nabla1 ⊢ a ♯ subst s1 (Susp [] X)
    using subst-susp ds-list-equ-ds[of pi1 pi2] by simp
  hence ∀ (a,t) ∈ set (map (λa. a ♯? Susp [] X) (ds-list pi1 pi2)). nabla1 ⊢ a ♯
subst s1 t
   by (auto split: prod.splits)
  with 7 i(2)
  have fresh:
    ∀ (a,t) ∈ set (map (λa. (a, Susp [] X)) (ds-list pi1 pi2) @ ys). nabla1 ⊢ a ♯
subst s1 t
   by auto

  from i(1)
  have eqs:
   ∀ (t1,t2) ∈ set xs. nabla1 ⊢ subst s1 t1 ≈ subst s1 t2
    by simp

  from subst fresh eqs
  show ?case using all-solutions-def by simp
next
  case (8 X t' pi xs ys)
  hence nabla1 ⊢ subst s1 (Susp pi X) ≈ subst s1 t'
    using all-solutions-def[of ‹((Susp pi X, t') # xs, ys)›] by auto
  hence subst: nabla1 ⊨ subst (s1 • [(X, swap (rev pi) t')]) ≈ subst s1
    using unif-1 by simp

  from 8
  have eqs-old:
```

$\forall (t1,t2) \in set\ xs.\ nabla1 \vdash subst\ s1\ t1 \approx subst\ s1\ t2$
and *fresh-old*:
$\forall (a,t) \in set\ ys.\ nabla1 \vdash a \sharp subst\ s1\ t$
using *all-solutions-def* **by** *auto*

from *eqs-old*
have *eqs*: $\forall (t1,t2) \in set\ xs.$
$nabla1 \vdash subst\ (s1 \cdot [(X, swap\ (rev\ pi)\ t')])\ t1 \approx subst\ (s1 \cdot [(X, swap\ (rev\ pi)\ t')])\ t2$
using *unif-2a*[*OF subst*] **by** *auto*

from *fresh-old*
have *fresh*: $\forall (a,t) \in set\ ys.\ nabla1 \vdash a \sharp subst\ (s1 \cdot [(X, swap\ (rev\ pi)\ t')])\ t$
using *unif-2b*[*OF subst*] **by** *auto*

from *eqs fresh*
have $(nabla1, (s1 \cdot [(X, swap\ (rev\ pi)\ t')])) \in U\ (xs, ys)$
using *all-solutions-def* **by** *simp*
hence *U*: $(nabla1, s1) \in U\ (apply\text{-}subst\ [(X, swap\ (rev\ pi)\ t')]\ (xs, ys))$
using *problem-subst-comm* **by** *simp*

from *subst U*
show *?case* **by** *simp*
next
case (*9 X t' pi xs ys*)
hence $nabla1 \vdash subst\ s1\ t' \approx subst\ s1\ (Susp\ pi\ X)$
using *all-solutions-def*[*of* $\langle((t', Susp\ pi\ X) \#\ xs, ys)\rangle$] **by** *simp*
hence *subst*: $nabla1 \models subst\ (s1 \cdot [(X, swap\ (rev\ pi)\ t')]) \approx subst\ s1$
using *unif-1*[*OF equ-symm*] **by** *blast*

from *9*
have *eqs-old*:
$\forall (t1,t2) \in set\ xs.\ nabla1 \vdash subst\ s1\ t1 \approx subst\ s1\ t2$
and *fresh-old*:
$\forall (a,t) \in set\ ys.\ nabla1 \vdash a \sharp subst\ s1\ t$
using *all-solutions-def* **by** *auto*

from *eqs-old*
have *eqs*: $\forall (t1,t2) \in set\ xs.$
$nabla1 \vdash subst\ (s1 \cdot [(X, swap\ (rev\ pi)\ t')])\ t1 \approx subst\ (s1 \cdot [(X, swap\ (rev\ pi)\ t')])\ t2$
using *unif-2a*[*OF subst*] **by** *auto*

from *fresh-old*
have *fresh*: $\forall (a,t) \in set\ ys.\ nabla1 \vdash a \sharp subst\ (s1 \cdot [(X, swap\ (rev\ pi)\ t')])\ t$
using *unif-2b*[*OF subst*] **by** *auto*

from *eqs fresh*
have $(nabla1, (s1 \cdot [(X, swap\ (rev\ pi)\ t')])) \in U\ (xs, ys)$

using *all-solutions-def* **by** *simp*
hence *U*: (*nabla1*, *s1*) ∈ *U* (*apply-subst* [(*X*, *swap* (*rev pi*) *t'*)] (*xs*, *ys*))
using *problem-subst-comm* **by** *simp*

from *subst U*
show *?case* **by** *simp*
qed

Auxiliary lemma for completeness

lemma *P1-from-P2-sred*:
assumes (*nabla1*,*s1*)∈ *U P2* **and** *P1*⊢*s2*⇝*P2*
shows (*nabla1*,*s1*•*s2*)∈ *U P1*
using *assms*
proof(*induction arbitrary*: *nabla1 s1 rule*: *s-red.induct*[*OF* ‹*P1* ⊢ *s2* ⇝ *P2*›])
case (*1 xs ys*)
hence ∀ (*t1*,*t2*) ∈ *set xs*. *nabla1* ⊢ *subst* (*s1* • []) *t1* ≈ *subst* (*s1* • []) *t2*
and *fresh*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst* (*s1* • []) *t*
using *all-solutions-def* **by** *simp-all*
hence *eqs*:
∀ (*t1*,*t2*) ∈ *set* ((*Unit*, *Unit*) # *xs*). *nabla1* ⊢ *subst* (*s1* • []) *t1* ≈ *subst* (*s1* • []) *t2*
using *subst-unit*[*of* ‹(*s1* • [])›] *equ-refl*[*of nabla1 Unit*] **by** *auto*
from *fresh eqs* **show** *?case*
using *all-solutions-def* **by** *simp*
next
case (*2 t1 t2 t3 t4 xs ys*)
hence *eqs-old*: ∀ (*t1*,*t2*) ∈ *set* ((*t1*, *t3*) # (*t2*, *t4*) # *xs*). *nabla1* ⊢ *subst* (*s1* • []) *t1* ≈ *subst* (*s1* • []) *t2*
and *fresh*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst* (*s1* • []) *t*
using *all-solutions-def* **by** *simp-all*
hence *nabla1* ⊢ *subst* (*s1* • []) *t1* ≈ *subst* (*s1* • []) *t3*
nabla1 ⊢ *subst* (*s1* • []) *t2* ≈ *subst* (*s1* • []) *t4* **by** *simp-all*
hence *nabla1* ⊢ *subst* (*s1* • []) (*Paar t1 t2*) ≈ *subst* (*s1* • []) (*Paar t3 t4*)
using *equ-paar* **by** *simp*
with *eqs-old* **have** *eqs*:
∀ (*t1*,*t2*) ∈ *set* ((*Paar t1 t2*, *Paar t3 t4*) # *xs*). *nabla1* ⊢ *subst* (*s1* • []) *t1* ≈ *subst* (*s1* • []) *t2*
by *auto*
from *fresh eqs* **show** *?case*
using *all-solutions-def* **by** *simp*
next
case (*3 f t1 t2 xs ys*)
hence *eqs-old*: ∀ (*t1*,*t2*) ∈ *set* ((*t1*, *t2*) # *xs*). *nabla1* ⊢ *subst* (*s1* • []) *t1* ≈ *subst* (*s1* • []) *t2*
and *fresh*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst* (*s1* • []) *t*
using *all-solutions-def* **by** *simp-all*
hence *nabla1* ⊢ *subst* (*s1* • []) (*Func f t1*) ≈ *subst* (*s1* • []) (*Func f t2*)
using *equ-func* **by** *simp*
with *eqs-old* **have** *eqs*:

```
    ∀ (t1,t2) ∈ set ((Func f t1, Func f t2) # xs). nabla1 ⊢ subst (s1 · []) t1 ≈
subst (s1 · []) t2
    by auto
  from fresh eqs show ?case
    using all-solutions-def by simp
next
  case (4 a t1 t2 xs ys)
  hence eqs-old: ∀ (t1,t2) ∈ set ((t1, t2) # xs). nabla1 ⊢ subst (s1 · []) t1 ≈
subst (s1 · []) t2
    and fresh: ∀ (a,t) ∈ set ys. nabla1 ⊢ a ♯ subst (s1 · []) t
    using all-solutions-def by simp-all
  hence nabla1 ⊢ subst (s1 · []) (Abst a t1) ≈ subst (s1 · []) (Abst a t2)
    using equ-abst-aa by simp
  with eqs-old have eqs:
    ∀ (t1,t2) ∈ set ((Abst a t1, Abst a t2) # xs). nabla1 ⊢ subst (s1 · []) t1 ≈
subst (s1 · []) t2
    by auto
  from fresh eqs show ?case
    using all-solutions-def by simp
next
  case (5 a b t1 t2 xs ys)
  hence eqs-old: ∀ (t1,t2) ∈ set ((t1, swap [(a, b)] t2) # xs). nabla1 ⊢ subst (s1
· []) t1 ≈ subst (s1 · []) t2
    and fresh: ∀ (a,t) ∈ set ((a, t2) # ys). nabla1 ⊢ a ♯ subst (s1 · []) t
    using all-solutions-def by simp-all
  hence nabla1 ⊢ subst (s1 · []) t1 ≈ swap [(a, b)] (subst (s1 · []) t2)
      nabla1 ⊢ a ♯ subst (s1 · []) t2
    using subst-swap-comm[of ‹(s1 · [])› ‹[(a, b)]› t2] by simp-all
  hence nabla1 ⊢ subst (s1 · []) (Abst a t1) ≈ subst (s1 · []) (Abst b t2)
   using equ-abst-ab[OF 5(1) ‹nabla1 ⊢ a ♯ subst (s1 · []) t2›] subst-abst by simp
  with eqs-old have eqs:
    ∀ (t1,t2) ∈ set ((Abst a t1, Abst b t2) # xs). nabla1 ⊢ subst (s1 · []) t1 ≈
subst (s1 · []) t2
    by auto
  from fresh eqs show ?case
    using all-solutions-def by simp
next
  case (6 a xs ys)
  hence ∀ (t1,t2) ∈ set xs. nabla1 ⊢ subst (s1 · []) t1 ≈ subst (s1 · []) t2
    and fresh: ∀ (a,t) ∈ set ys. nabla1 ⊢ a ♯ subst (s1 · []) t
    using all-solutions-def by simp-all
  hence eqs:
    ∀ (t1,t2) ∈ set ((Atom a, Atom a) # xs). nabla1 ⊢ subst (s1 · []) t1 ≈ subst
(s1 · []) t2
    using subst-unit[of ‹(s1 · [])›] equ-refl[of nabla1 Unit] by auto
  from fresh eqs show ?case
    using all-solutions-def by simp
next
  case (7 pi1 X pi2 xs ys)
```

hence *eqs-old*: ∀ (*t1*,*t2*) ∈ *set xs*. *nabla1* ⊢ *subst* (*s1* · []) *t1* ≈ *subst* (*s1* · []) *t2*
and *fresh-old*:
∀ (*a*,*t*) ∈ *set* (*map* (λ*a*. (*a*, *Susp* [] *X*)) (*ds-list pi1 pi2*) @ *ys*). *nabla1* ⊢ *a* ♯ *subst* (*s1* · []) *t*
using *all-solutions-def* **by** *simp-all*
hence ∀ *c*∈ *set* (*ds-list pi1 pi2*). *nabla1* ⊢ *c* ♯ *subst* (*s1* · []) (*Susp* [] *X*)
by (*auto split*: *prod.splits*)
hence ∀ *c* ∈ *ds pi1 pi2*. *nabla1* ⊢ *c* ♯ *subst* (*s1* · []) (*Susp* [] *X*)
using *ds-list-equ-ds* **by** *simp*
hence ∀ *c* ∈ *ds pi1 pi2*. *nabla1* ⊢ *c* ♯ (*look-up X* (*s1* · []))
using *subst-susp*[*of* ‹(*s1* · [])› ‹[]› *X*] *swap-id* **by** *simp*
hence *nabla1* ⊢ *subst* (*s1* · []) (*Susp pi1 X*) ≈ *subst* (*s1* · []) (*Susp pi2 X*)
using *equ-equiv-pi subst-susp*[*of* ‹(*s1* · [])› - *X*] **by** *auto*
with *eqs-old fresh-old* **have**
eqs: ∀ (*t1*,*t2*) ∈ *set* ((*Susp pi1 X*, *Susp pi2 X*)#*xs*).
nabla1 ⊢ *subst* (*s1* · []) *t1* ≈ *subst* (*s1* · []) *t2*
and *fresh*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst* (*s1* · []) *t*
by *simp-all*
then show *?case*
using *all-solutions-def* **by** *simp*
next
case (*8 X t′ pi xs ys*)
hence (*nabla1*, *s1* · [(*X*, *swap* (*rev pi*) *t′*)]) ∈ *U* (*xs*, *ys*)
using *problem-subst-comm* **by** *simp*
hence *eqs-old*:
∀ (*t1*,*t2*) ∈ *set xs*.
nabla1 ⊢ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t1* ≈ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t2*
and *fresh*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t*
using *all-solutions-def* **by** *simp-all*

have *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) (*Susp pi X*) = *subst s1* (*swap pi* (*swap* (*rev pi*) *t′*))
using *subst-susp subst-swap-comm subst-comp-expand eq-fst-iff look-up.simps*(*2*) *snd-eqD* **by** *metis*
also have ... = *swap pi* (*swap* (*rev pi*) (*subst s1 t′*))
using *subst-swap-comm* **by** *simp*
hence *nabla1* ⊢ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) (*Susp pi X*) ≈ *subst s1 t′*
using *equ-refl calculation rev-pi-pi-equ*[*of nabla1 pi* ‹*subst s1 t′*›]
equ-trans[*of nabla1* ‹*subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) (*Susp pi X*)›
‹*swap* (*rev pi*) (*swap pi* (*subst s1 t′*))› ‹*subst s1 t′*›]
swap-inv-side **by** *auto*

with *8 eqs-old*
have *eqs*: ∀ (*t1*,*t2*) ∈ *set* ((*Susp pi X*, *t′*)#*xs*).
nabla1 ⊢ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t1* ≈ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t2*
using *subst-comp-expand subst-not-occurs* **by** *simp*

from *fresh eqs* **show** *?case*
using *all-solutions-def* **by** *simp*
next
case (*9 X t′ pi xs ys*)
hence (*nabla1*, *s1* · [(*X*, *swap* (*rev pi*) *t′*)]) ∈ *U* (*xs*, *ys*)
using *problem-subst-comm* **by** *simp*
hence *eqs-old*:
∀ (*t1*,*t2*) ∈ *set xs*.
nabla1 ⊢ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t1* ≈ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t2*
and *fresh*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t*
using *all-solutions-def* **by** *simp-all*

have *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) (*Susp pi X*) = *subst s1* (*swap pi* (*swap* (*rev pi*) *t′*))
using *subst-susp subst-swap-comm subst-comp-expand eq-fst-iff look-up.simps*(*2*) *snd-eqD* **by** *metis*
also have ... = *swap pi* (*swap* (*rev pi*) (*subst s1 t′*))
using *subst-swap-comm* **by** *simp*
hence *nabla1* ⊢ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) (*Susp pi X*) ≈ *subst s1 t′*
using *equ-refl calculation rev-pi-pi-equ*[*of nabla1 pi* ‹*subst s1 t′*›]
equ-trans[*of nabla1* ‹*subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) (*Susp pi X*)›
‹*swap* (*rev pi*) (*swap pi* (*subst s1 t′*))› ‹*subst s1 t′*›]
swap-inv-side **by** *auto*
hence *nabla1* ⊢ *subst s1 t′* ≈ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) (*Susp pi X*)
using *equ-symm* **by** *simp*

with *9 eqs-old*
have *eqs*: ∀ (*t1*,*t2*) ∈ *set* ((*t′*, *Susp pi X*)#*xs*).
nabla1 ⊢ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t1* ≈ *subst* (*s1* · [(*X*, *swap* (*rev pi*) *t′*)]) *t2*
using *subst-comp-expand subst-not-occurs* **by** *simp*

from *fresh eqs* **show** *?case*
using *all-solutions-def* **by** *simp*
qed

lemma *P1-to-P2-cred*:
assumes (*nabla1*,*s1*)∈ *U P1*
and *P1* ⊢*nabla2* → *P2*
shows (*nabla1*,*s1*)∈ *U P2* ∧ (*nabla1*⊨(*subst s1*) *nabla2*)
using *assms*
proof(*induction arbitrary*: *nabla1 s1 rule*: *c-red.induct*[*OF* ‹*P1* ⊢*nabla2*→ *P2*›])
case (*2 a t1 t2 ys*)
hence *fresh-old*: ∀ (*a*,*t*) ∈ *set* ((*a*, *Paar t1 t2*) # *ys*). *nabla1* ⊢ *a* ♯ *subst s1 t*
and *eqs*: ∀ (*t1*,*t2*) ∈ *set* []. *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *simp-all*
hence *nabla1* ⊢ *a* ♯ *subst s1* (*Paar t1 t2*)
by *simp*

hence *nabla1* ⊢ *a* ♯ *subst s1 t1 nabla1* ⊢ *a* ♯ *subst s1 t2*
using *Fresh-elims*(*5*)[*of nabla1 a* ‹*subst s1 t1*›
‹*subst s1 t2*›] **by** *auto*
with *fresh-old* **have**
fresh: ∀ (*a*,*t*) ∈ *set* ((*a*, *t1*) # (*a*, *t2*) # *ys*). *nabla1* ⊢ *a* ♯ *subst s1 t*
by *auto*
from *fresh eqs*
show *?case*
using *all-solutions-def ext-subst-def* **by** *simp*
next
case (*3 a f t ys*)
hence *fresh-old*: ∀ (*a*,*t*) ∈ *set* ((*a*, *Func f t*) # *ys*). *nabla1* ⊢ *a* ♯ *subst s1 t*
and *eqs*: ∀ (*t1*,*t2*) ∈ *set* []. *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *simp-all*
hence *nabla1* ⊢ *a* ♯ *subst s1* (*Func f t*)
by *simp*
hence *nabla1* ⊢ *a* ♯ *subst s1 t*
using *Fresh-elims*(*6*) **by** *auto*
with *fresh-old* **have**
fresh: ∀ (*a*,*t*) ∈ *set* ((*a*, *t*) # *ys*). *nabla1* ⊢ *a* ♯ *subst s1 t*
by *auto*
from *fresh eqs*
show *?case*
using *all-solutions-def ext-subst-def* **by** *simp*
next
case (*5 a b t ys*)
hence *fresh-old*: ∀ (*a*,*t'*) ∈ *set* ((*a*, *Abst b t*) # *ys*). *nabla1* ⊢ *a* ♯ *subst s1 t'*
and *eqs*: ∀ (*t1*,*t2*) ∈ *set* []. *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *simp-all*
hence *nabla1* ⊢ *a* ♯ *subst s1* (*Abst b t*)
by *simp*
hence *nabla1* ⊢ *a* ♯ *subst s1 t*
using *5 Fresh-elims*(*1*)[*of nabla1 a b* ‹*subst s1 t*›] **by** *auto*
with *fresh-old* **have**
fresh: ∀ (*a*,*t'*) ∈ *set* ((*a*, *t*) # *ys*). *nabla1* ⊢ *a* ♯ *subst s1 t'*
by *auto*
from *fresh eqs*
show *?case*
using *all-solutions-def ext-subst-def* **by** *simp*
next
case (*7 b pi X ys*)
hence *fresh-old*: ∀ (*a*,*t*) ∈ *set* ((*b*, *Susp pi X*) # *ys*). *nabla1* ⊢ *a* ♯ *subst s1 t*
and *eqs*: ∀ (*t1*,*t2*) ∈ *set* []. *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *simp-all*
hence *fresh*: ∀ (*a*,*t*) ∈ *set ys*. *nabla1* ⊢ *a* ♯ *subst s1 t* **by** *simp*
from *fresh-old* **have**
nabla1 ⊢ *b* ♯ *subst s1* (*Susp pi X*) **by** *simp*
hence *nabla1* ⊢ *swapas* (*rev pi*) *b* ♯ *subst s1* (*Susp* [] *X*)
using *fresh-swap-left subst-susp* **by** *simp*

hence *ext*: *nabla1* $\models$ (*subst s1*) {(*swapas* (*rev pi*) *b*, *X*)}
using *ext-subst-def* **by** *simp*
from *fresh eqs ext*
show *?case* **using** *all-solutions-def* **by** *simp*
qed (*auto simp add*: *all-solutions-def ext-subst-def*)

lemma *P1-from-P2-cred*:
assumes (*nabla1*,*s1*)∈*U P2*
and *P1* ⊢*nabla2*→ *P2* **and** *nabla3*⊨(*subst s1*) *nabla2*
shows (*nabla1*∪*nabla3*,*s1*)∈ *U P1*
using *assms*
proof(*induction arbitrary*: *nabla1 s1 rule*: *c-red.induct*[*OF* ‹*P1* ⊢*nabla2*→ *P2*›])
case (*2 a t1 t2 xs*)
hence *fresh-old*:
∀(*b*,*t*)∈*set* ((*a*,*t1*)#(*a*,*t2*)#*xs*). *nabla1* ⊢ *b* ♯ *subst s1 t*
and ∀ (*t1*, *t2*) ∈ *set* []. *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *auto*
hence *weak1*: ∀(*b*,*t*)∈*set xs*. (*nabla1* ∪ *nabla3*) ⊢ *b* ♯ *subst s1 t*
∀ (*t1*, *t2*) ∈ *set* []. (*nabla1* ∪ *nabla3*) ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *fresh-weak* **by** *auto*
from *fresh-old* **have** *nabla1* ⊢ *a* ♯ *subst s1 t1 nabla1* ⊢ *a* ♯ *subst s1 t2*
by *simp*+
hence *nabla1* ⊢ *a* ♯ *subst s1* (*Paar t1 t2*)
using *fresh-paar subst-paar* **by** *auto*
hence (*nabla1* ∪ *nabla3*) ⊢ *a* ♯ *subst s1* (*Paar t1 t2*)
using *fresh-weak* **by** *auto*
with *weak1*
have *fresh*: ∀(*b*,*t*)∈*set* ((*a*,*Paar t1 t2*)#*xs*). (*nabla1* ∪ *nabla3*) ⊢ *b* ♯ *subst s1 t*
by *simp*
with *weak1*(*2*) **show** *?case*
using *all-solutions-def ext-subst-def* **by** *simp*
next
case (*3 a f t′ xs*)
hence *fresh-old*:
∀(*b*,*t*)∈*set* ((*a*,*t′*)#*xs*). *nabla1* ⊢ *b* ♯ *subst s1 t*
and ∀ (*t1*, *t2*) ∈ *set* []. *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *auto*
hence *weak1*: ∀(*b*,*t*)∈*set xs*. (*nabla1* ∪ *nabla3*) ⊢ *b* ♯ *subst s1 t*
∀ (*t1*, *t2*) ∈ *set* []. (*nabla1* ∪ *nabla3*) ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *fresh-weak* **by** *auto*
from *fresh-old* **have** *nabla1* ⊢ *a* ♯ *subst s1 t′*
by *simp*
hence *nabla1* ⊢ *a* ♯ *subst s1* (*Func f t′*)
using *fresh-paar subst-paar* **by** *auto*
hence (*nabla1* ∪ *nabla3*) ⊢ *a* ♯ *subst s1* (*Func f t′*)
using *fresh-weak* **by** *auto*
with *weak1*
have *fresh*: ∀(*b*,*t*)∈*set* ((*a*, *Func f t′*)#*xs*). (*nabla1* ∪ *nabla3*) ⊢ *b* ♯ *subst s1 t*
by *simp*

with *weak1*(*2*) **show** *?case*
using *all-solutions-def ext-subst-def* **by** *simp*
next
case (*5 a b t′ xs*)
hence *fresh-old*:
∀(*c*,*t*)∈*set* ((*a*,*t′*)#*xs*). *nabla1* ⊢ *c* ♯ *subst s1 t*
and ∀ (*t1*, *t2*) ∈ *set* []. *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *auto*
hence *weak1*: ∀(*c*,*t*)∈*set xs*. (*nabla1* ∪ *nabla3*) ⊢ *c* ♯ *subst s1 t*
∀ (*t1*, *t2*) ∈ *set* []. (*nabla1* ∪ *nabla3*) ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *fresh-weak* **by** *auto*
from *fresh-old* **have** *nabla1* ⊢ *a* ♯ *subst s1 t′*
by *simp*
hence *nabla1* ⊢ *a* ♯ *subst s1* (*Abst b t′*)
using *fresh-abst-ab*[*OF* ‹*nabla1* ⊢ *a* ♯ *subst s1 t′*› *5*(*1*)] *subst-abst* **by** *auto*
hence (*nabla1* ∪ *nabla3*) ⊢ *a* ♯ *subst s1* (*Abst b t′*)
using *fresh-weak* **by** *auto*
with *weak1*
have *fresh*: ∀(*b*,*t*)∈*set* ((*a*, *Abst b t′*)#*xs*). (*nabla1* ∪ *nabla3*) ⊢ *b* ♯ *subst s1 t*
by *simp*
with *weak1*(*2*) **show** *?case*
using *all-solutions-def ext-subst-def* **by** *simp*
next
case (*7 b pi X xs*)
hence *fresh-old*: ∀ (*a*,*t*) ∈ *set xs*. *nabla1* ⊢ *a* ♯ *subst s1 t*
and *eqs*: ∀ (*t1*,*t2*) ∈ *set* []. *nabla1* ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *all-solutions-def* **by** *simp-all*
hence *weak1*: ∀(*a*,*t*)∈*set xs*. (*nabla1* ∪ *nabla3*) ⊢ *a* ♯ *subst s1 t*
∀ (*t1*, *t2*) ∈ *set* []. (*nabla1* ∪ *nabla3*) ⊢ *subst s1 t1* ≈ *subst s1 t2*
using *fresh-weak* **by** *auto*
from *7* **have** *nabla3* ⊢ *swapas* (*rev pi*) *b* ♯ *subst s1* (*Susp* [] *X*)
using *ext-subst-def* **by** *auto*
hence *nabla3* ⊢ *b* ♯ *subst s1* (*Susp pi X*)
using *fresh-swap-right subst-susp* **by** *auto*
hence *weak2*: (*nabla1* ∪ *nabla3*) ⊢ *b* ♯ *subst s1* (*Susp pi X*)
using *fresh-weak*[*of nabla3 b - nabla1*] *Un-commute*[*of nabla3 nabla1*] **by** *auto*
with *weak1*(*1*)
have ∀(*a*,*t*)∈*set* ((*b*,*Susp pi X*)#*xs*). (*nabla1* ∪ *nabla3*) ⊢ *a* ♯ *subst s1 t*
by *simp*
with *weak2* **show** *?case*
using *ext-subst-def all-solutions-def* **by** *simp*
qed (*auto simp add*: *ext-subst-def all-solutions-def fresh-weak*)

lemma *P1-to-P2-red-plus1*:
assumes *P1* ⊨ (*nabla*,*s*) ⇒ *P2*
and (*nabla1*,*s1*) ∈ *U P1*
shows (*nabla1*,*s1*) ∈ *U P2*
using *assms*
proof(*induction P1*≡*P1 nabla*≡(*nabla*, *s*) *P2*≡*P2 arbitrary*: *s nabla1 nabla s1 P1*

P2 rule: *red-plus.induct*)
case (*sred-single P1 s P2 nabla1 s1*)
have *P1* ⊢ *s* ⇝ *P2* (*nabla1*, *s1*) ∈ *U P1* **by** *fact+*
then show (*nabla1*, *s1*) ∈ *U P2*
using *P1-to-P2-sred* **by** *blast*
next
case (*cred-single P1 nabla1 P2 nabla2 s1*)
have *P1* ⊢ *nabla1* →*P2* (*nabla2*, *s1*) ∈ *U P1* **by** *fact+*
then show (*nabla2*, *s1*) ∈ *U P2*
using *P1-to-P2-cred* **by** *blast*
next
case (*sred-step P1 s P2 nabla2 s2 P3 nabla1 s1*)
have *P1* ⊢ *s* ⇝ *P2* **and** (*nabla1*, *s1*) ∈ *U P1* **by** *fact+*
then have (*nabla1*, *s1*) ∈ *U P2* **using** *P1-to-P2-sred* **by** *blast*
moreover
have *IH*: (*nabla1*, *s1*) ∈ *U P2* ⟹ (*nabla1*, *s1*) ∈ *U P3* **by** *fact*
ultimately
show (*nabla1*, *s1*) ∈ *U P3* **by** *simp*
next
case (*cred-step P1 nabla1 P2 nabla2 P3 nabla3 s1*)
have *P1* ⊢ *nabla1* → *P2* **and** (*nabla3*, *s1*) ∈ *U P1* **by** *fact+*
then have (*nabla3*, *s1*) ∈ *U P2* **using** *P1-to-P2-cred* **by** *blast*
moreover
have *IH*: (*nabla3*, *s1*) ∈ *U P2* ⟹ (*nabla3*, *s1*) ∈ *U P3* **by** *fact*
ultimately show (*nabla3*, *s1*) ∈ *U P3* **by** *simp*
qed

lemma *P1-to-P2-red-plus3*:
assumes *P1* ⊨ (*nabla*,*s*) ⇒ *P2*
and (*nabla1*,*s1*)∈ *U P1*
shows *nabla1*⊨(*subst s1*) *nabla*
using *assms*
proof(*induction P1*≡*P1 nabla*≡(*nabla*, *s*) *P2*≡*P2 arbitrary*: *s nabla1 nabla s1 P1 P2 rule*: *red-plus.induct*)
case (*sred-single P1 s P2 nabla1 s1*)
then show *nabla1* ⊨ (*subst s1*) {} **by** (*simp add*: *ext-subst-def*)
next
case (*cred-single P1 nabla1 P2 nabla2 s1*)
have *P1* ⊢ *nabla1* → *P2* (*nabla2*, *s1*) ∈ *U P1* **by** *fact+*
then show *nabla2* ⊨ (*subst s1*) *nabla1* **using** *P1-to-P2-cred* **by** *blast*
next
case (*sred-step P1 s P2 nabla2 s2 P3 nabla1 s1*)
have *P1* ⊢ *s* ⇝ *P2* (*nabla1*, *s1*)∈ *U P1* **by** *fact+*
then have (*nabla1*, *s1*)∈ *U P2* **using** *P1-to-P2-sred* **by** *blast*
moreover have *IH*: (*nabla1*, *s1*)∈ *U P2* ⟹ *nabla1* ⊨ (*subst s1*) *nabla2* **by** *fact*
ultimately show *nabla1* ⊨ (*subst s1*) *nabla2* **by** *simp*
next
case (*cred-step P1 nabla1 P2 nabla2 P3 nabla3 s1*)

have *IH*: (*nabla3*, *s1*) ∈ *U P2* ⟹ *nabla3* ⊨ (*subst s1*) *nabla2* **by** *fact*
have *as*: *P1* ⊢ *nabla1* → *P2* (*nabla3*, *s1*) ∈ *U P1* **by** *fact+*

from *as* **have** *nabla3* ⊨ (*subst s1*) *nabla1* **using** *P1-to-P2-cred* **by** *blast*
moreover
from *as* **have** (*nabla3*, *s1*) ∈ *U P2* **using** *P1-to-P2-cred* **by** *blast*
then have *nabla3* ⊨ (*subst s1*) *nabla2* **using** *IH* **by** *blast*
ultimately show *nabla3* ⊨ (*subst s1*) (*nabla2* ∪ *nabla1*)
by(*auto simp add*: *ext-subst-def*)
qed

lemma *P1-to-P2-red-plus2*:
assumes *P1* ⊨ (*nabla*, *s*) ⇒ *P2*
and (*nabla1*, *s1*) ∈ *U P1*
shows *nabla1* ⊨ *subst* (*s1* · *s*) ≈ *subst s1*
using *assms*
proof(*induction P1*≡*P1 nablas*≡(*nabla*, *s*) *P2*≡*P2 arbitrary*: *s nabla1 nabla s1 P1 P2 rule*: *red-plus.induct*)
case (*sred-single P1 s P2 nabla1 s1*)
have *P1* ⊢*s* ⇝*P2* (*nabla1*, *s1*) ∈ *U P1* **by** *fact+*
then show *nabla1* ⊨ *subst* (*s1* · *s*) ≈*subst s1*
using *P1-to-P2-sred* **by** *blast*
next
case (*cred-single P1 nabla1 P2 nabla2 s1*)
show *nabla2* ⊨ *subst* (*s1* · []) ≈*subst s1*
by (*simp add*: *subst-equ-refl*)
next
case (*sred-step P1 s P2 nabla2 s2 P3 nabla1 s1*)
have *IH*: (*nabla1*, *s1*) ∈ *U P2* ⟹ *nabla1* ⊨ *subst* (*s1* · *s2*) ≈*subst s1* **by** *fact*
have *as*: *P1* ⊢*s* ⇝*P2* (*nabla1*, *s1*) ∈ *U P1* **by** *fact+*

from *as* **have** (*nabla1*, *s1*) ∈ *U P2* **using** *P1-to-P2-sred* **by** *blast*
with *IH* **have** *nabla1* ⊨ *subst* (*s1* · *s2*) ≈ *subst s1* **by** *blast*
then have *nabla1* ⊨ *subst* ((*s1* · *s2*) · *s*) ≈ *subst* (*s1* · *s*)
by (*simp add*: *subst-cancel-right*)
moreover
from *as* **have** *nabla1* ⊨ *subst* (*s1* · *s*) ≈*subst s1*
using *P1-to-P2-sred* **by** *blast*
ultimately
show *nabla1* ⊨ *subst* (*s1* · (*s2* · *s*)) ≈ *subst s1*
using *subst-assoc subst-trans* **by** *metis*
next
case (*cred-step P1 nabla1 P2 nabla2 P3*)
show *nabla1* ⊨ *subst* (*s1* · []) ≈ *subst s1*
by (*simp add*: *subst-equ-refl*)
qed

lemma *P1-from-P2-red-plus*:
assumes *P1* ⊨(*nabla*,*s*)⇒*P2*

and (*nabla1*,*s1*)∈ *U P2* **and** *nabla3*⊨(*subst s1*)(*nabla*)
shows (*nabla1*∪*nabla3*,(*s1* · *s*))∈ *U P1*
using *assms*
proof(*induction P1* ≡ *P1 nablas*≡(*nabla*, *s*) *P2*≡*P2 arbitrary*: *s nabla1 nabla nabla3 s1 P1 P2 rule*: *red-plus.induct*)
case (*sred-single P1 s P2 nabla1 nabla3*)
have *P1* ⊢ *s* ⇝ *P2* (*nabla1*, *s1*) ∈ *U P2* **by** *fact+*
then have (*nabla1*, *s1* · *s*) ∈ *U P1*
using *P1-from-P2-sred* **by** *simp*
then show (*nabla1* ∪ *nabla3*, *s1* · *s*) ∈ *U P1*
using *P1-from-P2-sred solution-weak* **by** *simp*
next
case (*cred-single P1 nabla P2 nabla1 nabla3*)
have *P1* ⊢ *nabla* → *P2*
(*nabla1*, *s1*) ∈ *U P2*
nabla3 ⊨ (*subst s1*) *nabla* **by** *fact+*
hence (*nabla1* ∪ *nabla3*, *s1*) ∈ *U P1*
using *P1-from-P2-cred* **by** *simp*
then show (*nabla1* ∪ *nabla3*, *s1* · []) ∈ *U P1*
using *solution-comp-id* **by** *auto*
next
case (*sred-step P1 s P2 nabla2 s2 P3 nabla1 nabla3*)
have *IH*: (*nabla1*, *s1*) ∈ *U P3* ⟹
nabla3 ⊨ (*subst s1*) *nabla2* ⟹ (*nabla1* ∪ *nabla3*, *s1* · *s2*) ∈ *U P2* **by** *fact+*
have *as*: (*nabla1*, *s1*) ∈ *U P3*
nabla3 ⊨ (*subst s1*) *nabla2*
P1 ⊢ *s* ⇝ *P2* **by** *fact+*
hence (*nabla1* ∪ *nabla3*, *s1* · *s2*) ∈ *U P2*
using *IH* **by** *simp*
hence (*nabla1* ∪ *nabla3*, (*s1* · *s2*) · *s*) ∈ *U P1*
using *as*(*3*)
by (*simp add*: *P1-from-P2-sred*)
then show (*nabla1* ∪ *nabla3*, *s1* · (*s2* · *s*)) ∈ *U P1*
using *solutions-subst-assoc* **by** *simp*
next
case (*cred-step P1 nabla P2 nabla2 P3*)
have *IH*: (*nabla1*, *s1*) ∈ *U P3* ⟹
nabla3 ⊨ (*subst s1*) *nabla2* ⟹ (*nabla1* ∪ *nabla3*, *s1* · []) ∈ *U P2* **by** *fact*
have *as*: *P1* ⊢ *nabla* → *P2*
(*nabla1*, *s1*) ∈ *U P3*
nabla3 ⊨ (*subst s1*) (*nabla2* ∪ *nabla*) **by** *fact+*
have *ext-substs*: *nabla3* ⊨ (*subst s1*) (*nabla2*)
nabla3 ⊨ (*subst s1*) *nabla*
using *ext-subst-strong*[*OF as*(*3*)] **by** *simp+*
hence *a*: (*nabla1* ∪ *nabla3*, *s1* · []) ∈ *U P2*
using *IH as*(*2*) **by** *simp*
hence (*nabla1* ∪ *nabla3* ∪ *nabla3*, *s1* · []) ∈ *U P1*
using *as*(*1*) *ext-substs*(*2*) *P1-from-P2-cred solution-comp-id* **by** *blast*
then show (*nabla1* ∪ *nabla3*, *s1* · []) ∈ *U P1*

unfolding *all-solutions-def* **using** *Un-absorb* **by** *simp*
qed

lemma *mgu*:
assumes $P \models(nabla,s)\Rightarrow([],[])$
shows *mgu P* (*nabla*,*s*) $\wedge$ *idem* (*nabla*,*s*)
proof(*rule mgu-idem*)
have *i*: ({},[]) $\in$ *U* ([],[])
unfolding *all-solutions-def* **by** *simp*
then show (*nabla*, *s*) $\in$ *U P*
using *P1-from-P2-red-plus*[*OF assms i*]
ext-subst-id solution-comp-id(*2*) **by** *simp*

show $\forall$ (*nabla2*, *s2*)$\in$ *U P*. *nabla2* $\models$ (*subst s2*) *nabla* $\wedge$
nabla2 $\models$ *subst* (*s2* $\cdot$ *s*) $\approx$ *subst s2*
proof
fix *x*
assume *x* $\in$ *U P*
then show *case x of* (*nabla2*, *s2*) $\Rightarrow$
nabla2 $\models$ (*subst s2*) *nabla* $\wedge$
nabla2 $\models$ *subst* (*s2* $\cdot$ *s*) $\approx$ *subst s2*
proof (*cases x*)
case (*Pair nabla2 s2*)
hence (*nabla2*,*s2*) $\in$ *U P*
using ‹*x* $\in$ *U P*› **by** *auto*
hence *nabla2* $\models$ (*subst s2*) *nabla* $\wedge$ *nabla2* $\models$ *subst* (*s2* $\cdot$ *s*) $\approx$ *subst s2*
using *assms P1-to-P2-red-plus2 P1-to-P2-red-plus3* **by** *auto*+
with *Pair* **show** *?thesis* **by** *simp*
qed
qed
qed

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*))
unfolding *apply-subst-def apply-subst-eprobs-def apply-subst-fprobs-def* **by** *simp*

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* ⇒ *nat*
where
size-fprobs [] = *0* |
size-fprobs (*x*#*xs*) = (*size-trm* (*snd x*))+(*size-fprobs xs*)

fun *size-eprobs* :: *eprobs* ⇒ *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*
by (*induct t*) *auto*

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*)
by (*induct t*) *auto*

lemma *vars-eprobs-finite* [*simp*]: *finite* (*vars-eprobs P*)
by (*induct P*) *auto*

lemma *not-occurs-trm*: ¬*occurs X t* ⟶ *X*∉ *vars-trm t*
by (*induct t*) *auto*

lemma *not-occurs-subst*: ¬*occurs X t1*⟶ *X*∉ *vars-trm* (*subst* [(*X*,*swap pi2 t1*)] *t2*)
using *subst-susp not-occurs-trm* **by** (*induct t2*) (*auto simp add*: *vars-swap*)

lemma *not-occurs-list*: ¬ *occurs X t* ⟶
X ∉ *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] *xs*)
using *not-occurs-subst apply-subst-eprobs-def* **by** (*induct xs*) *auto*

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*}
using *assms*
proof(*induct t2*)
case (*Susp pi′ Y*)
hence *eq*: *X*=*Y*
unfolding *occurs.simps*(*3*) **by** *argo*
have *subst-i*: *subst* [(*X*, *swap pi t1*)] (*Susp pi′ Y*) = *swap pi′*(*look-up X* [(*X*, *swap pi t1*)])
using *eq subst-susp* **by** *simp*
also have ... = *swap pi′*(*swap pi t1*) **by** *simp*
hence *vars-trm* (*subst* [(*X*, *swap pi t1*)] (*Susp pi′ Y*)) = *vars-trm t1*
using *subst-i swap-append vars-swap* **by** *simp*
thus
vars-trm (*subst* [(*X*, *swap pi t1*)] (*Susp pi′ Y*)) =
vars-trm t1 ∪ *vars-trm* (*Susp pi′ Y*) − {*X*}
using *eq assms*(*1*) *not-occurs-subst*[*of X t1 pi* ‹*Susp pi′ Y*›] **by** *auto*
next
case (*Paar t21 t22*)
hence *occurs X t21* ∨ *occurs X t22*
unfolding *occurs.simps*(*5*) **by** *argo*
then show *?case*
proof
assume *occurs X t21*
hence *vars-trm* (*subst* [(*X*, *swap pi t1*)] *t21*) = *vars-trm t1* ∪ *vars-trm t21* − {*X*}
using *assms*(*1*) *Paar* **by** *auto*
hence *vars-trm* (*subst* [(*X*, *swap pi t1*)] (*Paar t21 t22*)) =
(*vars-trm t1* ∪ *vars-trm t21* − {*X*}) ∪ (*vars-trm t1* ∪ *vars-trm t22* − {*X*})
using *Paar.hyps*(*2*) *assms*(*1*) *subst-not-occurs*[*of X t22* ‹*swap pi t1*›] *not-occurs-trm*
by (*cases occurs X t22*) *auto*
thus *vars-trm* (*subst* [(*X*, *swap pi t1*)] (*Paar t21 t22*)) =
vars-trm t1 ∪ *vars-trm* (*Paar t21 t22*) − {*X*} **by** *auto*
next
assume *occurs X t22*
hence *vars-trm* (*subst* [(*X*, *swap pi t1*)] *t22*) = *vars-trm t1* ∪ *vars-trm t22* − {*X*}
using *assms*(*1*) *Paar* **by** *auto*
hence *vars-trm* (*subst* [(*X*, *swap pi t1*)] (*Paar t21 t22*)) =
(*vars-trm t1* ∪ *vars-trm t21* − {*X*}) ∪ (*vars-trm t1* ∪ *vars-trm t22* − {*X*})
using *Paar.hyps*(*1*) *assms*(*1*) *subst-not-occurs*[*of X t21* ‹*swap pi t1*›] *not-occurs-trm*
by (*cases occurs X t21*) *auto*
thus *vars-trm* (*subst* [(*X*, *swap pi t1*)] (*Paar t21 t22*)) =
vars-trm t1 ∪ *vars-trm* (*Paar t21 t22*) − {*X*} **by** *auto*
qed
qed (*simp-all*)

lemma *vars-subseteq*:
assumes ¬*occurs X t*

shows *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] *xs*) ⊆ (*vars-trm t* ∪ *vars-eprobs xs*)
using *assms*
proof(*induct xs*)
case *Nil*
have *apply-subst-eprobs* [(*X*, *swap pi t*)] [] = []
unfolding *apply-subst-eprobs-def* **by** *simp*
hence *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] []) = {}
by *simp*
thus *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] []) ⊆ *vars-trm t* ∪ *vars-eprobs* []
by *simp*
next
case (*Cons a xs*)
have *apply-subst-eprobs* [(*X*, *swap pi t*)] (*a* # *xs*) =
(*subst* [(*X*, *swap pi t*)] (*fst a*), *subst* [(*X*, *swap pi t*)] (*snd a*)) # (*apply-subst-eprobs*[(*X*, *swap pi t*)] *xs*)
unfolding *apply-subst-eprobs-def case-prod-beta* **by** *simp*
hence *A*: *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] (*a* # *xs*)) =
(*vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)))∪(*vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)))
∪(*vars-eprobs* (*apply-subst-eprobs*[(*X*, *swap pi t*)] *xs*))
by *auto*
then show *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] (*a* # *xs*))
⊆ *vars-trm t* ∪ *vars-eprobs* (*a* # *xs*)
proof(*cases occurs X* (*fst a*))
case *True*
have *X-in-fst*:
occurs X (*fst a*) **by** *fact*
hence *vars-fst*:
vars-trm (*subst* [(*X*, *swap pi t*)] (*fst a*)) = (*vars-trm t* ∪ *vars-trm* (*fst a*))−{*X*}
using *vars-equ*[*OF assms X-in-fst*] **by** *simp*
then show *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] (*a* # *xs*))
⊆ *vars-trm t* ∪ *vars-eprobs* (*a* # *xs*)
proof(*cases occurs X* (*snd a*))
case *True*
have *X-in-snd*: *occurs X* (*snd a*) **by** *fact*
hence *vars-snd*:
vars-trm (*subst* [(*X*, *swap pi t*)] (*snd a*)) = (*vars-trm t* ∪ *vars-trm* (*snd a*))−{*X*}
using *vars-equ*[*OF assms X-in-snd*] **by** *simp*
with *vars-fst*
have *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) ∪ *vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)) =
vars-trm t ∪ *vars-trm* (*snd a*) ∪ *vars-trm* (*fst a*) − {*X*}
by *auto*
hence *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) ∪ *vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)) ⊆
vars-trm t ∪ *vars-eprobs* (*a* # *xs*) **by** *auto*

thus *?thesis* **using** *A Cons* **by** *auto*
next
case *False*
hence *vars-snd*: *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) = *vars-trm* (*snd a*)
using *subst-not-occurs* **by** *auto*
with *vars-fst*
have *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) ∪ *vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)) =
vars-trm t ∪ *vars-trm* (*snd a*) ∪ *vars-trm* (*fst a*) − {*X*}
using *False not-occurs-trm* **by** *auto*
hence *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) ∪ *vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)) ⊆
vars-trm t ∪ *vars-eprobs* (*a* # *xs*) **by** *auto*
thus *?thesis* **using** *A Cons* **by** *auto*
qed
next
case *False*
hence *vars-fst*:
vars-trm (*subst* [(*X*, *swap pi t*)] (*fst a*)) = *vars-trm* (*fst a*)
using *subst-not-occurs* **by** *simp*
then show *vars-eprobs* (*apply-subst-eprobs* [(*X*, *swap pi t*)] (*a* # *xs*))
⊆ *vars-trm t* ∪ *vars-eprobs* (*a* # *xs*)
proof(*cases occurs X* (*snd a*))
case *True*
have *X-in-snd*: *occurs X* (*snd a*) **by** *fact*
hence *vars-snd*:
vars-trm (*subst* [(*X*, *swap pi t*)] (*snd a*)) = (*vars-trm t* ∪ *vars-trm* (*snd a*))−{*X*}
using *vars-equ*[*OF assms X-in-snd*] **by** *simp*
with *vars-fst*
have *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) ∪ *vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)) =
vars-trm t ∪ *vars-trm* (*snd a*) ∪ *vars-trm* (*fst a*) − {*X*}
using *False not-occurs-trm* **by** *auto*
hence *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) ∪ *vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)) ⊆
vars-trm t ∪ *vars-eprobs* (*a* # *xs*) **by** *auto*
thus *?thesis* **using** *A Cons* **by** *auto*
next
case *False*
hence *vars-snd*: *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) = *vars-trm* (*snd a*)
using *subst-not-occurs* **by** *auto*
with *vars-fst*
have *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) ∪ *vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)) =
vars-trm (*snd a*) ∪ *vars-trm* (*fst a*)
by *simp*
hence *vars-trm* (*subst* [(*X*, *swap pi t*)] (*snd a*)) ∪ *vars-trm* (*subst* [(*X*, *swap pi t*)] (*fst a*)) ⊆

```
    vars-trm t ∪ vars-eprobs (a # xs) by auto
    thus ?thesis using A Cons by auto
   qed
  qed
qed


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))
  using assms
proof(cases X ∈ (vars-trm t ∪ vars-eprobs xs))
  case True
  hence insert X (vars-trm t ∪ vars-eprobs xs) = vars-trm t ∪ vars-eprobs xs
    using  insert-absorb by auto
  moreover have subset:
    vars-eprobs (apply-subst-eprobs [(X, swap pi t)] xs) ⊆ vars-trm t ∪ vars-eprobs
xs
    using assms vars-subseteq by simp
  hence
    vars-eprobs (apply-subst-eprobs [(X, swap pi t)] xs) ⊆ vars-trm t ∪ vars-eprobs
xs − {X}
    using not-occurs-list assms subset-Diff-insert by auto
  ultimately have
  card (vars-eprobs (apply-subst-eprobs [(X, swap pi t)] xs))
              ≤ card (vars-trm t ∪ vars-eprobs xs − {X})
    using Finite-Set.card-mono
    by (metis finite-Diff finite-Un vars-eprobs-finite vars-term-finite)
  thus card (vars-eprobs (apply-subst-eprobs [(X, swap pi t)] xs))
              < card (insert X (vars-trm t ∪ vars-eprobs xs))
    by (simp add: card.insert-remove)
next
  case False
 hence card (vars-trm t ∪ vars-eprobs xs) < card (insert X (vars-trm t ∪ vars-eprobs
xs))
    by auto
  moreover have subset:
    vars-eprobs (apply-subst-eprobs [(X, swap pi t)] xs) ⊆ vars-trm t ∪ vars-eprobs
xs
    using assms vars-subseteq by simp
  hence card (vars-eprobs (apply-subst-eprobs [(X, swap pi t)] xs))
              ≤ card (vars-trm t ∪ vars-eprobs xs)
    using Finite-Set.card-mono
    by (metis finite-Un vars-eprobs-finite vars-term-finite)
  ultimately show
    card (vars-eprobs (apply-subst-eprobs [(X, swap pi t)] xs))
    < card (insert X (vars-trm t ∪ vars-eprobs xs))
    by simp
```

qed

lemma *rank-r-cred*:
assumes *P1*⊢(*nabla*)→*P2*
shows (*P2*, *P1*) ∈ *rank-r*
unfolding *rank-r-def* **by** (*cases rule*: *c-red.cases*[*OF* ‹*P1* ⊢ *nabla* → *P2*›], *simp-all*)

lemma *rank-r-sred*:
assumes *P1* ⊢ *s* ⇝*P2*
shows (*P2*,*P1*) ∈ *rank-r*
proof(*cases rule*: *s-red.cases*[*OF* ‹*P1*⊢ *s* ⇝*P2*›])
case (*2 t1 t2 s1 s2 xs ys*)
case (*2 t1 t2 s1 s2 xs ys*)
let *?union* = *vars-trm s1* ∪ *vars-trm s2* ∪ *vars-trm t1* ∪ *vars-trm t2* ∪ *vars-eprobs xs*
and *?size* = *size-trm t1* + *size-trm t2* + *size-trm s1* + *size-trm s2* + *size-eprobs xs*
from *2* **have** *vars-eprobs* ((*Paar t1 t2*, *Paar s1 s2*) # *xs*) = *?union*
unfolding *vars-eprobs.simps vars-trm.simps* **by** *auto*
moreover from *2* **have**
size-eprobs ((*Paar t1 t2*, *Paar s1 s2*) # *xs*) = *2* + *?size*
unfolding *size-eprobs.simps* **using** *size-trm.simps*(*6*) **by** *auto*
from *2* **have**
vars-eprobs ((*t1*, *s1*) # (*t2*, *s2*) # *xs*) = *?union*
unfolding *vars-eprobs.simps* **by** *auto*
moreover from *2* **have**
size-eprobs ((*t1*, *s1*) # (*t2*, *s2*) # *xs*) = *?size*
unfolding *size-eprobs.simps* **by** *simp*
ultimately have
size-eprobs ((*t1*, *s1*) # (*t2*, *s2*) # *xs*) < *size-eprobs* ((*Paar t1 t2*, *Paar s1 s2*) # *xs*)
card (*vars-eprobs* ((*t1*, *s1*) # (*t2*, *s2*) # *xs*)) = *card* (*vars-eprobs* ((*Paar t1 t2*, *Paar s1 s2*) # *xs*))
by *simp*+
with *2* **show** (*P2*, *P1*) ∈ *rank-r*
unfolding *rank-r-def* **by** *simp*
next
case (*7 pi1 X pi2 xs ys*)
let *?mapds* = *map* (λ*a*. (*a*, *Susp* [] *X*)) (*ds-list pi1 pi2*) @ *ys*
let *?union* = {*X*} ∪ *vars-eprobs xs*
from *7* **have**
vars: *vars-eprobs* ((*Susp pi1 X*, *Susp pi2 X*) # *xs*) = *?union* **and**
size: *size-eprobs* ((*Susp pi1 X*, *Susp pi2 X*) # *xs*) = *2* + *size-eprobs xs*
unfolding *vars-eprobs.simps size-eprobs.simps* **by** *simp*+
then show (*P2*, *P1*) ∈ *rank-r*
proof(*cases X* ∈ *vars-eprobs xs*)
case *True*

```
    hence card ?union = card (vars-eprobs xs)
      by (simp add: insert-absorb)
    with 7 show (P2, P1) ∈ rank-r
      unfolding rank-r-def by simp
  next
    case False
    hence card ?union = 1 + card (vars-eprobs xs)
      by auto
    with 7 show (P2, P1) ∈ rank-r
      unfolding rank-r-def by simp
  qed
next
  case (8 X t pi xs ys)
   let ?union = insert X (vars-trm t ∪ vars-eprobs xs)
    and ?size = size-trm t + size-eprobs xs
   from 8 have
     vars: vars-eprobs ((Susp pi X, t) # xs) = ?union and
     size: size-eprobs ((Susp pi X, t) # xs) = 1 + ?size
    unfolding vars-eprobs.simps size-eprobs.simps by simp+
   moreover from 8 have
    P2 = (apply-subst-eprobs [(X, swap (rev pi) t)] xs,
    apply-subst-fprobs [(X, swap (rev pi) t)] ys)
     using apply-subst-equivalence by auto
   ultimately show ?thesis
    using 8 vars-decrease[OF 8(4)] unfolding rank-r-def by simp
next
  case (9 X t pi xs ys)
   let ?union = insert X (vars-trm t ∪ vars-eprobs xs)
    and ?size = size-trm t + size-eprobs xs
  from 9 have
     vars: vars-eprobs ((t, Susp pi X) # xs) = ?union and
     size: size-eprobs ((t, Susp pi X) # xs) = 1 + ?size
    unfolding vars-eprobs.simps size-eprobs.simps by simp+
  moreover from 9 have
    P2 = (apply-subst-eprobs [(X, swap (rev pi) t)] xs,
    apply-subst-fprobs [(X, swap (rev pi) t)] ys)
     using apply-subst-equivalence by auto
   ultimately show ?thesis
    using 9 vars-decrease[OF 9(4)] unfolding rank-r-def by simp
qed (unfold rank-r-def, auto simp add: vars-swap)

lemma rank-r-trans: ⟦(P1,P2) ∈ rank-r; (P2,P3) ∈ rank-r⟧⟹ (P1,P3)∈ rank-r
  unfolding rank-r-def by auto

lemma rank-r-red-plus:
  assumes P1⊨ (s,nabla)⇒P2
  shows (P2, P1) ∈ rank-r
  using assms
by(induct rule: red-plus.induct)(auto dest: rank-r-sred rank-r-cred rank-r-trans)
```

lemma *wf-rank-r*:
shows *wf* (*rank-r*)
unfolding *rank-r-def* **by** *simp*

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*)
by(*auto simp add*: *all-solutions-def equ-symm*)

lemma *fail-then-empty*:
assumes *fail P1*
shows *U P1* = {}
using *assms*
proof(*induct rule*: *fail.induct*)
case (*fail-occur-abst X t pi a xs ys*)
let *?P* = (*Susp pi X* ≈*? Abst a t* # *xs*, *ys*)
{ assume *U* ((*Susp pi X*, *Abst a t*) # *xs*, *ys*) ≠ {}
then obtain *s nabla* **where** *eq1*: *nabla* ⊢ *subst s* (*Susp pi X*) ≈ *Abst a* (*subst s t*)
by (*auto simp add*: *all-solutions-def*)
moreover
have *occurs X t* **by** *fact*
then obtain *t′ pi′* **where**
eq2: *nabla* ⊢ *subst s* (*Susp pi X*) ≈ *swap pi′ t′ t′*∈*sub-trms* (*subst s t*)
using *occurs-sub-trm-equ* **by** *blast*
moreover
have *eq3*: ¬ *nabla* ⊢ (*Abst a* (*subst s t*)) ≈ *swap pi′ t′*
using *eq2 psub-trm-not-equ* **by** *auto*
then have *False* **using** *eq1 eq2 eq3*
by (*metis equ-symm equ-trans*)
}
then show *U ?P* = {} **by** *auto*
next
case (*fail-occur-func X t pi F xs ys*)
let *?P* = (*Susp pi X* ≈*? Func F t* # *xs*, *ys*)
{ assume *U* ((*Susp pi X*, *Func F t*) # *xs*, *ys*) ≠ {}
then obtain *s nabla* **where** *eq1*: *nabla* ⊢ *subst s* (*Susp pi X*) ≈ *Func F* (*subst s t*)
by (*auto simp add*: *all-solutions-def*)
moreover
have *occurs X t* **by** *fact*
then obtain *t′ pi′* **where**
eq2: *nabla* ⊢ *subst s* (*Susp pi X*) ≈ *swap pi′ t′ t′*∈*sub-trms* (*subst s t*)
using *occurs-sub-trm-equ* **by** *blast*
moreover
have *eq3*: ¬ *nabla* ⊢ (*Func F* (*subst s t*)) ≈ *swap pi′ t′*
using *eq2 psub-trm-not-equ* **by** *auto*
then have *False* **using** *eq1 eq2 eq3*
by (*metis equ-symm equ-trans*)
}
then show *U ?P* = {} **by** *auto*
next
case (*fail-occur-paar X t1 t2 pi xs ys*)
let *?P* = (*Susp pi X* ≈*? Paar t1 t2* # *xs*, *ys*)
have *occurs X t1* ∨ *occurs X t2* **by** *fact*

```
  then show U ?P = {}
  proof
    {assume occurs X t1
     {assume U ((Susp pi X, Paar t1 t2) # xs, ys) ≠ {}
    then obtain s nabla where eq1: nabla ⊢ subst s (Susp pi X) ≈ Paar (subst s
t1) (subst s t2)
      by (auto simp add: all-solutions-def)
    moreover
    have occurs X t1 by fact
     then obtain t' pi' where
      eq2: nabla ⊢ subst s (Susp pi X) ≈ swap pi' t' t'∈sub-trms (subst s t1)
      using occurs-sub-trm-equ by blast
    moreover
    have eq3: ¬ nabla ⊢ (Paar (subst s t1) (subst s t2)) ≈ swap pi' t'
      using eq2 psub-trm-not-equ by auto
    then have False using eq1 eq2 eq3
      by (metis equ-symm equ-trans)}
  then show U ?P = {} by auto}

    {assume occurs X t2
    {assume U ((Susp pi X, Paar t1 t2) # xs, ys) ≠ {}
      then obtain s nabla where eq1: nabla ⊢ subst s (Susp pi X) ≈ Paar (subst
s t1) (subst s t2)
        by (auto simp add: all-solutions-def)
      moreover
      have occurs X t2 by fact
      then obtain t' pi' where
        eq2: nabla ⊢ subst s (Susp pi X) ≈ swap pi' t' t'∈sub-trms (subst s t2)
        using occurs-sub-trm-equ by blast
      moreover
      have eq3: ¬ nabla ⊢ (Paar (subst s t1) (subst s t2)) ≈ swap pi' t'
        using eq2 psub-trm-not-equ by auto
      then have False using eq1 eq2 eq3
        by (metis equ-symm equ-trans)
    }
    then show U ?P = {} by auto}
  qed
next
  case (fail-fresh-atom a ys)
  let ?P = ([], a ♯? Atom a # ys)
  have ∄ nabla s. nabla ⊢ a ♯ subst s (Atom a)
    using subst-atom Fresh-elims(3) by auto
  thus U ?P = {}
    using all-solutions-def by simp
next
  case (fail-diff-atoms a b xs ys)
  let ?P = (Atom a ≈? Atom b # xs, ys)
  from ‹a ≠ b› have ∄ nabla s. nabla ⊢ subst s (Atom a) ≈ subst s (Atom b)
    using Equ-elims(1) by auto
```

```
  thus U ?P = {}
    using all-solutions-def by simp
next
  case (fail-abst-unit a t xs ys)
  let ?P = (Abst a t ≈? Unit # xs, ys)
  have ∄ nabla s. nabla ⊢ subst s (Abst a t) ≈ subst s Unit
    by (auto elim: equ.cases)
  thus U ?P = {}
    using all-solutions-def by simp
next
  case (fail-abst-atom a t b xs ys)
  let ?P = (Abst a t ≈? Atom b # xs, ys)
  have ∄ nabla s. nabla ⊢ subst s (Abst a t) ≈ subst s (Atom b)
    by (auto elim: equ.cases)
  thus U ?P = {}
    using all-solutions-def by simp
next
  case (fail-abst-paar a t t1 t2 xs ys)
  let ?P = (Abst a t ≈? Paar t1 t2 # xs, ys)
  have ∄ nabla s. nabla ⊢ subst s (Abst a t) ≈ subst s (Paar t1 t2)
    by (auto elim: equ.cases)
  thus U ?P = {}
    using all-solutions-def by simp
next
  case (fail-func-abst F t1 a t xs ys)
  let ?P = (Func F t1 ≈? Abst a t # xs, ys)
  have ∄ nabla s. nabla ⊢ subst s (Func F t1) ≈ subst s (Abst a t)
    by (auto elim: equ.cases)
  thus U ?P = {}
    using all-solutions-def by simp
next
  case (fail-atom-unit b xs ys)
  let ?P = (Atom b ≈? Unit # xs, ys)
  have ∄ nabla s. nabla ⊢ subst s (Atom b) ≈ subst s (Unit)
    by (auto elim: equ.cases)
  thus U ?P = {}
    using all-solutions-def by simp
next
  case (fail-paar-unit t1 t2 xs ys)
  let ?P = (Paar t1 t2 ≈? Unit # xs, ys)
  have ∄ nabla s. nabla ⊢ subst s (Paar t1 t2) ≈ subst s (Unit)
    by (auto elim: equ.cases)
  thus U ?P = {}
    using all-solutions-def by simp
next
  case (fail-func-unit F t1 xs ys)
  let ?P = (Func F t1≈? Unit # xs, ys)
  have ∄ nabla s. nabla ⊢ subst s (Func F t1) ≈ subst s (Unit)
    by (auto elim: equ.cases)
```

thus *U ?P = {}*
using *all-solutions-def* **by** *simp*
next
case (*fail-atom-paar b t1 t2 xs ys*)
let *?P = (Atom b ≈? Paar t1 t2 # xs, ys)*
have ♯ *nabla s. nabla ⊢ subst s (Atom b) ≈ subst s (Paar t1 t2)*
by (*auto elim*: *equ.cases*)
thus *U ?P = {}*
using *all-solutions-def* **by** *simp*
next
case (*fail-func-atom F t1 b xs ys*)
let *?P = (Func F t1 ≈? Atom b # xs, ys)*
have ♯ *nabla s. nabla ⊢ subst s (Func F t1) ≈ subst s (Atom b)*
by (*auto elim*: *equ.cases*)
thus *U ?P = {}*
using *all-solutions-def* **by** *simp*
next
case (*fail-func-paar F t t1 t2 xs ys*)
let *?P = (Func F t ≈? Paar t1 t2 # xs, ys)*
have ♯ *nabla s. nabla ⊢ subst s (Func F t) ≈ subst s (Paar t1 t2)*
by (*auto elim*: *equ.cases*)
thus *U ?P = {}*
using *all-solutions-def* **by** *simp*
next
case (*fail-diff-func F1 F2 t1 t2 xs ys*)
let *?P = (Func F1 t1 ≈? Func F2 t2 # xs, ys)*
from ‹*F1 ≠ F2*› **have** ♯ *nabla s. nabla ⊢ subst s (Func F1 t1) ≈ subst s (Func F2 t2)*
by (*auto elim*: *equ.cases*)
thus *U ?P = {}*
using *all-solutions-def* **by** *simp*
next
case (*fail-sym s t xs ys*)
let *?P = (t ≈? s # xs, ys)*
have *fail* ((*s*, *t*) # *xs*, *ys*)
U ((*s*, *t*) # *xs*, *ys*) = {} **by** *fact+*
thus *U ?P = {}*
using *all-solutions-def U-equ-symm* **by** *simp*
qed

lemma *not-reduce-then-fail*:
assumes ¬ (∃ *nabla s P′*. ((*t1 ≈? t2*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P′*)
shows *fail* ((*t1 ≈? t2*) # *xs*, *ys*)
using *assms*
proof(*cases t1*)
case (*Abst a t1′*)

```
  have t1-def: t1 = Abst a t1' by fact
  then show fail ((t1, t2) # xs, ys)
  proof(cases t2)
    case (Abst b t2')
    with t1-def
    have ((t1 ≈? t2)#xs,ys) ⊢[]⇝ ((t1'≈?t2')#xs,ys) ∨
    ((t1≈? t2)#xs,ys) ⊢[]⇝ ((t1'≈?swap [(a,b)] t2')#xs,(a♯?t2')#ys)
      using abst-aa-sred abst-ab-sred by (cases a=b) auto
    hence ∃ P2. ((t1 ≈? t2)#xs,ys) ⊨({},[])⇒P2
      using sred-single by blast
    thus fail ((t1, t2) # xs, ys)
      using assms by simp
  next
    case (Susp pi X)
    have t2-def: t2 = Susp pi X by fact
    with t1-def
    show fail ((t1, t2) # xs, ys)
    proof(cases occurs X t1')
      case True
      with t1-def t2-def
      show fail ((t1, t2) # xs, ys)
      using fail-sym[OF fail-occur-abst[OF True]] by simp
    next
      case False
      with t1-def
      have not-occurs: ¬ occurs X t1 by simp
      hence ((t1≈? Susp pi X)#xs,ys)
                ⊢[(X,swap (rev pi) t1)]⇝ apply-subst [(X,swap (rev pi) t1)] (xs,ys)
        using t1-def var-2-sred[OF not-occurs] by simp
       hence ∃ P2 s. ((t1 ≈? t2)#xs,ys) ⊨({},s)⇒P2
         using t1-def t2-def sred-single by blast
       thus fail ((t1, t2) # xs, ys)
         using assms by simp
    qed
  qed (auto)
next
  case (Susp pi X)
  have t1-def: t1 = Susp pi X by fact
  then show fail ((t1, t2) # xs, ys)
  proof(cases occurs X t2)
    case True
    then show fail ((t1, t2) # xs, ys)
    proof(cases t2)
      case (Abst a t2')
      have t2-def: t2 = Abst a t2' by fact
      with True
      have occurs X t2' unfolding occurs.simps(4) t2-def by argo
      thus fail ((t1, t2) # xs, ys)
        using t1-def t2-def fail-occur-abst by simp
```

```
    next
      case (Susp pi' Y)
      have t2-def: t2 = Susp pi' Y by fact
      have X = Y
        using True unfolding t2-def occurs.simps(3)
        by argo
      hence ((Susp pi X≈?Susp pi' Y)#xs,ys)
                              ⊢[]⇝ (xs,(map (λa. a♯? Susp [] X) (ds-list pi pi'))@ys)
        using susp-sred by simp
      hence ∃ P2. ((t1 ≈? t2)#xs,ys) ⊨({},[])⇒P2
        using sred-single t1-def t2-def by blast
      thusfail ((t1, t2) # xs, ys)
        using assms by simp
    next
      case (Paar t21 t22)
      have t2-def: t2 = Paar t21 t22 by fact
      with True
     have occurs X t21 ∨ occurs X t22 unfolding occurs.simps(5) t2-def by argo
      thus fail ((t1, t2) # xs, ys)
        using t1-def t2-def fail-occur-paar by simp
    next
      case (Func f t2')
      have t2-def: t2 = Func f t2' by fact
      with True
      have occurs X t2' unfolding occurs.simps(6) t2-def by argo
      thus fail ((t1, t2) # xs, ys)
        using t1-def t2-def fail-occur-func by simp
    qed (auto simp add: True)
  next
    case False
    hence ((Susp pi X, t2) # xs, ys) ⊢
    [(X, swap (rev pi) t2)] ⇝ apply-subst [(X, swap (rev pi) t2)] (xs, ys)
      using var-1-sred[OF False] by simp
    hence ∃ P2 s. ((t1 ≈? t2)#xs,ys) ⊨({},s)⇒P2
      using t1-def sred-single by blast
    thus fail ((t1, t2) # xs, ys)
      using assms by simp
  qed
next
  case Unit
  have t1-def: t1 = Unit by fact
  then show fail ((t1, t2) # xs, ys)
  proof(cases t2)
    case (Susp pi X)
    have t2-def: t2 =  Susp pi X by fact
    with t1-def have not-occurs: ¬ occurs X t1 by simp
    hence ((t1≈? Susp pi X)#xs,ys)
                ⊢[(X,swap (rev pi) t1)]⇝ apply-subst [(X,swap (rev pi) t1)] (xs,ys)
      using t1-def var-2-sred[OF not-occurs] by simp
```

```
    hence ∃ P2 s. ((t1 ≈? t2)#xs,ys) ⊨({},s)⇒P2
      using t1-def t2-def sred-single by blast
    thus fail ((t1, t2) # xs, ys)
      using assms by simp
  next
    case Unit
    with t1-def
    have ((t1 ≈? t2)#xs,ys) ⊢[]⇝ (xs,ys)
      using unit-sred by auto
    hence ∃ P2. ((t1 ≈? t2)#xs,ys) ⊨({},[])⇒P2
      using sred-single by blast
    thus fail ((t1, t2) # xs, ys)
      using assms by simp
  qed (auto)
next
  case (Atom a)
  have t1-def: t1 = Atom a by fact
  then show fail ((t1, t2) # xs, ys)
  proof(cases t2)
    case (Susp pi X)
    have t2-def: t2 =  Susp pi X by fact
    with t1-def have not-occurs: ¬ occurs X t1 by simp
    hence ((t1≈? Susp pi X)#xs,ys)
                ⊢[(X,swap (rev pi) t1)]⇝ apply-subst [(X,swap (rev pi) t1)] (xs,ys)
      using t1-def var-2-sred[OF not-occurs] by simp
    hence ∃ P2 s. ((t1 ≈? t2)#xs,ys) ⊨({},s)⇒P2
      using t1-def t2-def sred-single by blast
    thus fail ((t1, t2) # xs, ys)
      using assms by simp
  next
    case (Atom b)
    have t2-def: t2 = Atom b by fact
    then show fail ((t1, t2) # xs, ys)
    proof(cases a=b)
      case True
      hence ((t1 ≈? t2)#xs,ys) ⊢[]⇝ (xs,ys)
        using t2-def t1-def atom-sred by simp
      hence ∃ P2. ((t1 ≈? t2)#xs,ys) ⊨({},[])⇒P2
        using sred-single by blast
      thus fail ((t1, t2) # xs, ys)
        using assms by simp
    qed (simp add: t1-def t2-def fail-diff-atoms)
  qed(auto)
next
  case (Paar t11 t12)
  have t1-def: t1 = Paar t11 t12 by fact
  then show fail ((t1, t2) # xs, ys)
  proof(cases t2)
  next
```

```
    case (Susp pi X)
    have t2-def: t2 = Susp pi X by fact
    then show fail ((t1, t2) # xs, ys)
    proof(cases occurs X t11 ∨ occurs X t12)
      case True
      then show fail ((t1, t2) # xs, ys)
        using t1-def t2-def fail-sym[OF fail-occur-paar[OF True]] by simp
    next
      case False
      hence ¬ occurs X t1
        using t1-def by simp
      hence ((t1≈?Susp pi X)#xs,ys)
                                ⊢[(X,swap (rev pi) t1)]⇝ apply-subst [(X,swap (rev pi)
t1)] (xs,ys)
        by auto
      hence ∃ P2 s. ((t1 ≈? t2)#xs,ys) ⊨({},s)⇒P2
        using t1-def t2-def sred-single by blast
      thus fail ((t1, t2) # xs, ys)
        using assms by simp
    qed
  next
    case (Paar t21 t22)
    have t2-def: t2 = Paar t21 t22 by fact
    with t1-def have
    ((t1 ≈? t2)#xs,ys) ⊢[]⇝ ((t11≈?t21)#(t12≈?t22)#xs,ys)
      using paar-sred by simp
    hence ∃ P2. ((t1 ≈? t2)#xs,ys) ⊨({},[])⇒P2
      using sred-single by blast
    thus fail ((t1, t2) # xs, ys)
      using assms by simp
  qed(auto)
next
  case (Func f t1')
  have t1-def: t1 = Func f t1' by fact
  then show fail ((t1, t2) # xs, ys)
  proof(cases t2)
    case (Susp pi X)
     have t2-def: t2 = Susp pi X by fact
     with t1-def
    show fail ((t1, t2) # xs, ys)
    proof(cases occurs X t1')
      case True
      with t1-def t2-def
      show fail ((t1, t2) # xs, ys)
      using fail-sym[OF fail-occur-func[OF True]] by simp
    next
      case False
      with t1-def
      have not-occurs: ¬ occurs X t1 by simp
```

```
      hence ((t1≈? Susp pi X)#xs,ys)
              ⊢[(X,swap (rev pi) t1)]⇝ apply-subst [(X,swap (rev pi) t1)] (xs,ys)
       using t1-def var-2-sred[OF not-occurs] by simp
      hence ∃ P2 s. ((t1 ≈? t2)#xs,ys) ⊨({},s)⇒P2
       using t1-def t2-def sred-single by blast
      thus fail ((t1, t2) # xs, ys)
       using assms by simp
    qed
  next
    case (Func g t2')
     have t2-def: t2 = Func g t2' by fact
     then show fail ((t1, t2) # xs, ys)
     proof(cases f=g)
      case True
      hence ((t1 ≈? t2)#xs,ys) ⊢[]⇝ ((t1'≈?t2')#xs,ys)
       using t1-def t2-def func-sred by simp
      hence ∃ P2. ((t1 ≈? t2)#xs,ys) ⊨({},[])⇒P2
       using sred-single by blast
      thus fail ((t1, t2) # xs, ys)
       using assms by simp
     next
      case False
      then show fail ((t1, t2) # xs, ys)
       using t1-def t2-def fail-diff-func[OF False] by simp
     qed
  qed(auto)
qed


lemma fresh-reduces-if-not-atom:
  assumes t ≠ Atom a
  shows ∃P2 nabla s. ([], (a ♯? t) # xs) ⊨ (nabla,s) ⇒ P2
  using assms cred-single
proof(cases t)
  case (Abst b t')
  then show ∃P2 nabla s. ([], (a ♯? t) # xs) ⊨ (nabla, s) ⇒ P2
  proof(cases a=b)
    case True
    hence ([], (a ♯? t) # xs) ⊢{}→ ([],xs)
     unfolding Abst using abst-aa-cred by simp
    then show ∃P2 nabla s. ([], (a ♯? t) # xs) ⊨ (nabla, s) ⇒ P2
     using cred-single by blast
  next
    case False
    hence ([], (a ♯? t) # xs) ⊢{}→ ([], (a♯? t') # xs)
     unfolding Abst using abst-ab-cred by simp
    then show ∃P2 nabla s. ([], (a ♯? t) # xs) ⊨ (nabla, s) ⇒ P2
     using cred-single by blast
  qed
```

next
case (*Atom b*)
with *assms*
have $a \neq b$ **by** *simp*
hence ([], (a ♯? t) # *xs*) ⊢ {}→ ([],*xs*)
unfolding *Atom* **using** *atom-cred* **by** *simp*
then show ∃ *P2 nabla s*. ([], (a ♯? t) # *xs*) ⊨ (*nabla*, *s*) ⇒ *P2*
using *cred-single* **by** *blast*
qed (*simp add*: *cred-single*, *blast*+)

lemma *empty-stuck*:
shows ([],[]) ∈ *stuck*
proof−
have ¬ (∃ *P2 nabla s*. ([],[]) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
thus ([],[]) ∈ *stuck*
unfolding *stuck-def* **by** *auto*
qed

lemma *fail-is-stuck*:
assumes *fail P*
shows $P \in$ *stuck*
using *assms*
proof(*induct rule*: *fail.induct*)
case (*fail-occur-abst X t pi a xs ys*)
have *t-occurs*: *occurs X t* **by** *fact*
moreover have ¬ (∃ *P2 nabla s*. ((*Susp pi X* ≈? *Abst a t*) # *xs*, *ys*) ⊨ (*nabla*,*s*)
⇒ *P2*)
proof
assume ∃ *P2 nabla s*. ((*Susp pi X* ≈? *Abst a t*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*
then obtain *P2 nabla s* **where**
red: ((*Susp pi X* ≈? *Abst a t*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*
by *auto*
thus *False*
proof (*cases rule*: *red-plus.cases*)
case *sred-single*
have ((*Susp pi X*, *Abst a t*) # *xs*, *ys*) ⊢ *s* ⇝ *P2* **by** *fact*
hence ¬ *occurs X t*
by (*auto elim*: *s-red.cases*)
with *t-occurs* **show** *False* **by** *simp*
next
case *cred-single*
have ((*Susp pi X*, *Abst a t*) # *xs*, *ys*) ⊢ *nabla* → *P2* **by** *fact*
moreover have *fst* ((*Susp pi X*, *Abst a t*) # *xs*, *ys*) ≠ []
by *simp*
ultimately show *False*
using *c-red-eqs-empty* **by** *blast*
next

```
    case (sred-step s1 P3 s2)
    have ((Susp pi X, Abst a t) # xs, ys) ⊢ s1 ⇝ P3 by fact
    hence ¬ occurs X t
      by (auto elim: s-red.cases)
    with t-occurs show False by simp
  next
    case (cred-step nabla1 P3 nabla2)
    have ((Susp pi X, Abst a t) # xs, ys) ⊢ nabla1 → P3 by fact
     moreover have fst ((Susp pi X, Abst a t) # xs, ys) ≠ []
      by simp
    ultimately show False
      using c-red-eqs-empty by blast
  qed
 qed
 then show ((Susp pi X, Abst a t) # xs, ys) ∈ stuck
  unfolding stuck-def by simp
next
 case (fail-occur-func X t pi F xs ys)
 have t-occurs: occurs X t by fact
 moreover have ¬ (∃ P2 nabla s. ((Susp pi X ≈? Func F t) # xs, ys) ⊨ (nabla,s)
⇒ P2)
 proof
  assume ∃ P2 nabla s. ((Susp pi X ≈? Func F t) # xs, ys) ⊨ (nabla,s) ⇒ P2
  then obtain P2 nabla s where
  red: ((Susp pi X ≈? Func F t) # xs, ys) ⊨ (nabla,s) ⇒ P2
    by auto
  thus False
  proof (cases rule: red-plus.cases)
    case sred-single
    have ((Susp pi X, Func F t) # xs, ys) ⊢ s ⇝ P2 by fact
    hence ¬ occurs X t
      by (auto elim: s-red.cases)
    with t-occurs show False by simp
  next
    case cred-single
    have ((Susp pi X, Func F t) # xs, ys) ⊢ nabla → P2 by fact
    moreover have fst ((Susp pi X, Func F t) # xs, ys) ≠ []
      by simp
    ultimately show False
      using c-red-eqs-empty by blast
  next
    case (sred-step s1 P3 s2)
    have ((Susp pi X, Func F t) # xs, ys) ⊢ s1 ⇝ P3 by fact
    hence ¬ occurs X t
      by (auto elim: s-red.cases)
    with t-occurs show False by simp
  next
    case (cred-step nabla1 P3 nabla2)
    have ((Susp pi X, Func F t) # xs, ys) ⊢ nabla1 → P3 by fact
```

```
      moreover have fst ((Susp pi X, Func F t) # xs, ys) ≠ []
       by simp
      ultimately show False
       using c-red-eqs-empty by blast
    qed
  qed
  then show ((Susp pi X, Func F t) # xs, ys) ∈ stuck
    unfolding stuck-def by simp
next
  case (fail-occur-paar X t1 t2 pi xs ys)
  have occurs X t1 ∨ occurs X t2 by fact
  hence t-occurs: occurs X (Paar t1 t2)
    using occurs.simps(5) by auto
  moreover have ¬ (∃ P2 nabla s. ((Susp pi X ≈? Paar t1 t2) # xs, ys) ⊨
(nabla,s) ⇒ P2)
  proof
    assume ∃ P2 nabla s. ((Susp pi X ≈? Paar t1 t2) # xs, ys) ⊨ (nabla,s) ⇒ P2
    then obtain P2 nabla s where
    red: ((Susp pi X ≈? Paar t1 t2) # xs, ys) ⊨ (nabla,s) ⇒ P2
      by auto
    thus False
    proof (cases rule: red-plus.cases)
      case sred-single
      have ((Susp pi X, Paar t1 t2) # xs, ys) ⊢ s ⇝ P2 by fact
      hence ¬ occurs X (Paar t1 t2)
        by (auto elim: s-red.cases)
      with t-occurs show False by simp
    next
      case cred-single
      have ((Susp pi X, Paar t1 t2) # xs, ys) ⊢ nabla → P2 by fact
      moreover have fst ((Susp pi X, Paar t1 t2) # xs, ys) ≠ []
        by simp
      ultimately show False
        using c-red-eqs-empty by blast
    next
      case (sred-step s1 P3 s2)
      have ((Susp pi X, Paar t1 t2) # xs, ys) ⊢ s1 ⇝ P3 by fact
      hence ¬ occurs X (Paar t1 t2)
        by (auto elim: s-red.cases)
      with t-occurs show False by simp
    next
      case (cred-step nabla1 P3 nabla2)
      have ((Susp pi X, Paar t1 t2) # xs, ys) ⊢ nabla1 → P3 by fact
       moreover have fst ((Susp pi X, Paar t1 t2) # xs, ys) ≠ []
        by simp
      ultimately show False
        using c-red-eqs-empty by blast
    qed
  qed
```

then show ((*Susp pi X*, *Paar t1 t2*) # *xs*, *ys*) ∈ *stuck*
unfolding *stuck-def* **by** *simp*
next
case (*fail-fresh-atom a ys*)
have ¬ (∃ *P2 nabla s*. ([], (*a*, *Atom a*) # *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
then show ([], (*a*, *Atom a*) # *ys*) ∈ *stuck*
unfolding *stuck-def* **by** *simp*
next
case (*fail-diff-atoms a b xs ys*)
hence ¬ (∃ *P2 nabla s*. ((*Atom a*, *Atom b*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
then show ((*Atom a*, *Atom b*) # *xs*, *ys*) ∈ *stuck*
unfolding *stuck-def* **by** *simp*
next
case (*fail-abst-unit a t xs ys*)
hence ¬ (∃ *P2 nabla s*. ((*Abst a t*, *Unit*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
then show ((*Abst a t*, *Unit*) # *xs*, *ys*) ∈ *stuck*
unfolding *stuck-def* **by** *simp*
next
case (*fail-abst-atom a t b xs ys*)
hence ¬ (∃ *P2 nabla s*. ((*Abst a t*, *Atom b*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
then show ((*Abst a t*, *Atom b*) # *xs*, *ys*) ∈ *stuck*
unfolding *stuck-def* **by** *simp*
next
case (*fail-abst-paar a t t1 t2 xs ys*)
hence ¬ (∃ *P2 nabla s*. ((*Abst a t*, *Paar t1 t2*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
then show ((*Abst a t*, *Paar t1 t2*) # *xs*, *ys*) ∈ *stuck*
unfolding *stuck-def* **by** *simp*
next
case (*fail-func-abst F t1 a t xs ys*)
hence ¬ (∃ *P2 nabla s*. ((*Func F t1*, *Abst a t*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
then show ((*Func F t1*, *Abst a t*) # *xs*, *ys*) ∈ *stuck*
unfolding *stuck-def* **by** *simp*
next
case (*fail-atom-unit b xs ys*)
hence ¬ (∃ *P2 nabla s*. ((*Atom b*, *Unit*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
then show ((*Atom b*, *Unit*) # *xs*, *ys*) ∈ *stuck*
unfolding *stuck-def* **by** *simp*
next
case (*fail-paar-unit t1 t2 xs ys*)
hence ¬ (∃ *P2 nabla s*. ((*Paar t1 t2*, *Unit*) # *xs*, *ys*) ⊨ (*nabla*,*s*) ⇒ *P2*)
by (*auto elim*: *red-plus.cases s-red.cases c-red.cases*)
then show ((*Paar t1 t2*, *Unit*) # *xs*, *ys*) ∈ *stuck*

```
    unfolding stuck-def by simp
next
  case (fail-func-unit F t1 xs ys)
  hence ¬ (∃ P2 nabla s. ((Func F t1, Unit) # xs, ys) ⊨ (nabla,s) ⇒ P2)
    by (auto elim: red-plus.cases s-red.cases c-red.cases)
  then show ((Func F t1, Unit) # xs, ys) ∈ stuck
    unfolding stuck-def by simp
next
  case (fail-atom-paar a t1 t2 xs ys)
  hence ¬ (∃ P2 nabla s. ((Atom a, Paar t1 t2) # xs, ys) ⊨ (nabla,s) ⇒ P2)
    by (auto elim: red-plus.cases s-red.cases c-red.cases)
  then show ((Atom a, Paar t1 t2) # xs, ys) ∈ stuck
    unfolding stuck-def by simp
next
  case (fail-func-atom F t1 a xs ys)
  hence ¬ (∃ P2 nabla s. ((Func F t1, Atom a) # xs, ys) ⊨ (nabla,s) ⇒ P2)
    by (auto elim: red-plus.cases s-red.cases c-red.cases)
  then show ((Func F t1, Atom a) # xs, ys) ∈ stuck
    unfolding stuck-def by simp
next
  case (fail-func-paar F t t1 t2 xs ys)
   hence ¬ (∃ P2 nabla s. ((Func F t, Paar t1 t2) # xs, ys) ⊨ (nabla,s) ⇒ P2)
    by (auto elim: red-plus.cases s-red.cases c-red.cases)
  then show ((Func F t, Paar t1 t2) # xs, ys) ∈ stuck
    unfolding stuck-def by simp
next
  case (fail-diff-func F1 F2 t1 t2 xs ys)
   hence ¬ (∃ P2 nabla s. ((Func F1 t1, Func F2 t2) # xs, ys) ⊨ (nabla,s) ⇒ P2)
    by (auto elim: red-plus.cases s-red.cases c-red.cases)
  then show ((Func F1 t1, Func F2 t2) # xs, ys) ∈ stuck
    unfolding stuck-def by simp
next
  case (fail-sym s t xs ys)
  hence not-reduce: ∄ P2 nabla s1. ((s, t) # xs, ys) ⊨ (nabla, s1) ⇒ P2
    unfolding stuck-def by simp
  have∄ P2 nabla s1. ((t, s) # xs, ys) ⊨ (nabla, s1) ⇒ P2
  proof
    assume ∃ P2 nabla s1. ((t, s) # xs, ys) ⊨ (nabla, s1) ⇒ P2
    then obtain P2 nabla s1 where
      reduces: ((t, s) # xs, ys) ⊨ (nabla, s1) ⇒ P2
      by auto
    hence ∃ P3 nabla1 s2. ((s, t) # xs, ys) ⊨ (nabla1, s2) ⇒ P3
      using red-plus-symm[of ‹((t, s) # xs, ys)› t s xs ys nabla s1 P2] by auto
    with not-reduce show False by simp
  qed
  then show ((t, s) # xs, ys) ∈ stuck unfolding stuck-def by simp
qed
```

lemma *stuck-equiv*:
shows $stuck = \{([],[])\} \cup \{P1.\ fail\ P1\}$
proof (*rule set-eqI*, *rule iffI*)
fix P
{assume *P-is-stuck*: $P \in stuck$
then obtain *eqs freshs* **where**
P-def: $P = (eqs, freshs)$ **by** (*cases P*)
show $P \in \{([], [])\} \cup \{P1.\ fail\ P1\}$
proof(*cases eqs*)
case *Nil*
then show $P \in \{([], [])\} \cup \{P1.\ fail\ P1\}$
proof(*cases freshs*)
case *Nil*
with ‹$eqs = []$›
show $P \in \{([], [])\} \cup \{P1.\ fail\ P1\}$ **using** *P-def* **by** *simp*
next
case (*Cons c freshs'*)
then obtain *a t* **where** *c-def*: $c = a\ \sharp?\ t$ **by** *force*
have $t = Atom\ a$
using *fresh-reduces-if-not-atom P-is-stuck*
unfolding *stuck-def P-def Nil Cons c-def* **by** *blast*
hence *fail P*
unfolding *P-def Nil Cons c-def* **using** *fail-fresh-atom* **by** *simp*
thus $P \in \{([], [])\} \cup \{P1.\ fail\ P1\}$ **by** *auto*
qed
next
case(*Cons e eqs'*)
then obtain *s t* **where** *e-def*: $e = s \approx? t$ **by** *force*
have *fail P*
using *P-is-stuck* **unfolding** *P-def Cons e-def*
stuck-def **using** *not-reduce-then-fail* **by** *simp*
thus $P \in \{([], [])\} \cup \{P1.\ fail\ P1\}$ **by** *auto*
qed }

{assume $P \in \{([], [])\} \cup \{P1.\ fail\ P1\}$
then show $P \in stuck$
using *empty-stuck fail-is-stuck* **by** *blast*
}
qed

lemma *u-empty-sred*:
assumes *P1*$\vdash$*s*$\rightsquigarrow$*P2* **and** *U P2* $=\{\}$
shows *U P1* $= \{\}$
using *assms P1-from-P2-sred all-solutions-def P1-to-P2-sred* **by** *blast*

lemma *u-empty-cred*:
assumes *P1*⊢*nabla*→*P2* **and** *U P2* ={}
shows *U P1*={}
using *assms P1-from-P2-cred all-solutions-def P1-to-P2-cred* **by** *blast*

lemma *u-empty-red-plus*:
assumes *P1*⊨(*nabla*,*s*)⇒*P2* **and** *U P2* ={}
shows *U P1*={}
using *assms P1-from-P2-red-plus all-solutions-def P1-to-P2-red-plus1* **by** *fast*

lemma *empty-then-fail*:
assumes *U P1*={}
shows (∀ *P* ∈ *normal-form P1*. *fail P*)
proof
fix *P*
assume *P-is-nf*: *P* ∈ *normal-form P1*
hence *P-is-stuck*: *P* ∈ *stuck*
using *normal-form-def* **by** (*cases P1* ∈ *stuck*) *auto*
hence *P-is-empty-or-fails*: *P* = ([],[]) ∨ *fail P*
using *stuck-equiv* **by** *auto*
have *P* ≠ ([],[])
proof
assume *P-empty*: *P* = ([],[])
hence *solution*: ({},[]) ∈ *U P*
using *all-solutions-def* **by** *simp*
hence *P-neq*: *P* ≠ *P1*
using *assms* **by** *auto*
show *False*
proof(*cases P1*∈ *stuck*)
case *True*
then have *normal-form P1* = {*P1*}
unfolding *normal-form-def* **by** *simp*
with *P-is-nf* **have** *P* = *P1* **by** *simp*
with *P-neq* **show** *False* **by** *simp*
next
case *False*
with *P-is-nf*
obtain *nabla s* **where** *P1-goes-to-P*: *P1* ⊨ (*nabla*, *s*) ⇒ *P*
unfolding *normal-form-def* **by** *auto*
hence ({} ∪ *nabla*, [] · *s*) ∈ *U P1*
using *P1-from-P2-red-plus*[*OF P1-goes-to-P solution ext-subst-id*] **by** *simp*
with *assms* **show** *False* **by** *simp*
qed
qed
thus *fail P*

```
    using P-is-empty-or-fails by simp
qed



lemma not-empty-then-not-fail:
  assumes U P1≠{}
  shows ¬(∃ P∈ normal-form P1. fail P)
proof(simp, rule ballI)
  fix P
  assume P-is-nf: P ∈ normal-form P1
  show ¬ fail P
  proof
    assume P-fails: fail P
    show False
    proof(cases P1∈ stuck)
      case True
      have normal-form P1 = {P1}
        unfolding normal-form-def using True by simp
      hence P = P1 using P-is-nf by simp
      with P-fails have U P1 = {}
        using fail-then-empty by simp
      thus False using assms by simp
      next
        case False
        with P-is-nf
        obtain nabla s where P1-goes-to-P: P1 ⊨ (nabla, s) ⇒ P
          unfolding normal-form-def by auto
        moreover have U P = {}
          using P-fails fail-then-empty by simp
        ultimately have U P1 = {}
          using u-empty-red-plus by simp
        then show False
          using assms by simp
      qed
    qed
  qed
```

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.