First-Order Terms*

Christian Sternagel   René Thiemann

September 13, 2023

Abstract

We formalize basic results on first-order terms, including a first-order unification algorithm, as well as well-foundedness of the subsumption order. This entry is part of the *Isabelle Formalization of Rewriting* IsaFoR [2], where first-order terms are omni-present: the unification algorithm is used to certify several confluence and termination techniques, like critical-pair computation and dependency graph approximations; and the subsumption order is a crucial ingredient for completion.

Contents

1 Introduction **2**

2 Auxiliary Results **3**
- 2.1 Reflexive Transitive Closures of Orders 3
- 2.2 Rename names in two ways. 3
- 2.3 Make lists instances of the infinite-class. 4
- 2.4 Renaming strings apart 4
- 2.5 Results on Infinite Sequences 4
- 2.6 Results on Bijections 5
- 2.7 Merging Functions 7
- 2.8 The Option Monad 8

3 First-Order Terms **11**
- 3.1 Restrict the Domain of a Substitution 22
- 3.2 Rename the Domain of a Substitution 23
- 3.3 Rename the Domain and Range of a Substitution 25
- 3.4 Multisets of Pairs of Terms 27
- 3.5 Size 27
  - 3.5.1 Substitutions 28
  - 3.5.2 Variables 29

*Supported by FWF (Austrian Science Fund) projects Y757 and P27502

4 Abstract Matching **30**

5 Unification **36**
5.1 Unifiers . . . 36
5.1.1 Properties of sets of unifiers . . . 37
5.1.2 Properties of unifiability . . . 39
5.1.3 Properties of *is-mgu* . . . 39
5.1.4 Properties of *is-imgu* . . . 40
5.2 Abstract Unification . . . 42
5.2.1 Inference Rules . . . 42
5.2.2 Termination of the Inference Rules . . . 43
5.2.3 Soundness of the Inference Rules . . . 46
5.2.4 Completeness of the Inference Rules . . . 47
5.3 A Concrete Unification Algorithm . . . 50
5.3.1 Unification of two terms where variables should be considered disjoint . . . 65
5.3.2 A variable disjoint unification algorithm without changing the type . . . 68

6 Matching **68**
6.1 A variable disjoint unification algorithm for terms with string variables . . . 72

7 Subsumption **73**
7.1 Equality of terms modulo variables . . . 76
7.2 Well-foundedness . . . 79

8 Subterms and Contexts **81**
8.1 Subterms . . . 81
8.1.1 Syntactic Sugar . . . 81
8.1.2 Transitivity Reasoning for Subterms . . . 83
8.2 Contexts . . . 88
8.3 The Connection between Contexts and the Superterm Relation 89

1 Introduction

We define first-order terms, substitutions, the subsumption order, and a unification algorithm. In all these definitions type-parameters are used to specify variables and function symbols, but there is no explicit signature.

The unification algorithm has been formalized following a textbook on term rewriting [1].

The complete IsaFoR library is available at:

http://cl-informatik.uibk.ac.at/isafor/

2 Auxiliary Results

2.1 Reflexive Transitive Closures of Orders

theory *Transitive-Closure-More*
  imports *Main*
begin

lemma (**in** *order*) *rtranclp-less-eq* [*simp*]:
  $(\leq)^{**} = (\leq)$
  by (*intro ext*) (*auto elim*: *rtranclp-induct*)

lemma (**in** *order*) *tranclp-less* [*simp*]:
  $(<)^{++} = (<)$
  by (*intro ext*) (*auto elim*: *tranclp-induct*)

lemma (**in** *order*) *rtranclp-greater-eq* [*simp*]:
  $(\geq)^{**} = (\geq)$
  by (*intro ext*) (*auto elim*: *rtranclp-induct*)

lemma (**in** *order*) *tranclp-greater* [*simp*]:
  $(>)^{++} = (>)$
  by (*intro ext*) (*auto elim*: *tranclp-induct*)

end

2.2 Rename names in two ways.

theory *Renaming2*
  imports
    Fresh-Identifiers.Fresh
begin

typedef ($'v$ :: *infinite*) *renaming2* = { ($v1$ :: $'v \Rightarrow 'v$, $v2$ :: $'v \Rightarrow 'v$) | $v1$ $v2$. *inj* $v1 \wedge$ *inj* $v2 \wedge$ *range* $v1 \cap$ *range* $v2 = \{\}$ }
proof −
  let *?U* = *UNIV* :: *'v set*
  have *inf*: *infinite ?U* **by** (*rule infinite-UNIV*)
  have *ordLeq3* (*card-of ?U*) (*card-of ?U*)
    using *card-of-refl ordIso-iff-ordLeq* **by** *blast*
  from *card-of-Plus-infinite1*[*OF inf this*, *folded card-of-ordIso*]
  obtain *f* **where** *bij*: *bij-betw f* (*?U* <+> *?U*) *?U* **by** *auto*
  hence *injf*: *inj f* **by** (*simp add*: *bij-is-inj*)
  define *v1* **where** *v1* = *f o Inl*
  define *v2* **where** *v2* = *f o Inr*
  have *inj*: *inj v1 inj v2* **unfolding** *v1-def v2-def* **by** (*intro inj-compose*[*OF injf*], *auto*)+
  have *range v1* $\cap$ *range v2* = {}
  proof (*rule ccontr*)
    assume ¬ *?thesis*

    then obtain *x* **where** *v1 x* = *v2 x*
      using *injD injf v1-def v2-def* **by** *fastforce*
    hence *f* (*Inl x*) = *f* (*Inr x*) **unfolding** *v1-def v2-def* **by** *auto*
    with *injf* **show** *False* **unfolding** *inj-on-def* **by** *blast*
  qed
  with *inj* **show** *?thesis* **by** *blast*
qed

setup-lifting *type-definition-renaming2*

lift-definition *rename-1* :: ′*v* :: *infinite renaming2* ⇒ ′*v* ⇒ ′*v* **is** *fst* **.**
lift-definition *rename-2* :: ′*v* :: *infinite renaming2* ⇒ ′*v* ⇒ ′*v* **is** *snd* **.**

lemma *rename-12*: *inj* (*rename-1 r*) *inj* (*rename-2 r*) *range* (*rename-1 r*) ∩ *range* (*rename-2 r*) = {}
  by (*transfer*, *auto*)+

end

2.3 Make lists instances of the infinite-class.

theory *Lists-are-Infinite*
  imports *Fresh-Identifiers.Fresh*
begin

instance *list* :: (*type*) *infinite*
  by (*intro-classes*, *rule infinite-UNIV-listI*)

end

2.4 Renaming strings apart

theory *Renaming2-String*
  imports
    Renaming2
    Lists-are-Infinite
begin

lift-definition *string-rename* :: *string renaming2* **is** (*Cons* (*CHR* ′′*x*′′), *Cons* (*CHR* ′′*y*′′))
  by *auto*

end

2.5 Results on Infinite Sequences

theory *Seq-More*
  imports
    Abstract−Rewriting.Seq
    Transitive-Closure-More

begin

lemma *down-chain-imp-eq*:
  fixes f :: *nat seq*
  assumes $\forall i.\ f\ i \geq f\ (Suc\ i)$
  shows $\exists N.\ \forall i > N.\ f\ i = f\ (Suc\ i)$
proof −
  let $?F = \{f\ i \mid i.\ True\}$
  from *wf-less* [*unfolded wf-eq-minimal*, *THEN spec*, *of ?F*]
    obtain x **where** $x \in\ ?F$ **and** $*$: $\forall y.\ y < x \longrightarrow y \notin\ ?F$ **by** *auto*
  obtain N **where** $f\ N = x$ **using** ‹$x \in\ ?F$› **by** *auto*
  moreover have $\forall i > N.\ f\ i \in\ ?F$ **by** *auto*
  ultimately have $\forall i > N.\ \neg\ f\ i < x$ **using** $*$ **by** *auto*
  moreover have $\forall i > N.\ f\ N \geq f\ i$
    using *chainp-imp-rtranclp* [*of* ($\geq$) f, *OF assms*] **by** *simp*
  ultimately have $\forall i > N.\ f\ i = f\ (Suc\ i)$
    using ‹$f\ N = x$› **by** (*auto*) (*metis less-SucI order.not-eq-order-implies-strict*)
  then show *?thesis* **..**
qed

lemma *inc-seq-greater*:
  fixes f :: *nat seq*
  assumes $\forall i.\ f\ i < f\ (Suc\ i)$
  shows $\exists i.\ f\ i > N$
  using *assms*
  apply (*induct N*)
   apply (*auto*)
   apply (*metis neq0-conv*)
  by (*metis Suc-lessI*)

end

2.6 Results on Bijections

theory *Fun-More* **imports** *Main* **begin**

lemma *finite-card-eq-imp-bij-betw*:
  assumes *finite A*
    and *card* (f ‘ A) = *card* A
  shows *bij-betw* f A (f ‘ A)
  using ‹*card* (f ‘ A) = *card* A›
  unfolding *inj-on-iff-eq-card* [*OF* ‹*finite A*›, *symmetric*]
  by (*rule inj-on-imp-bij-betw*)

Every bijective function between two subsets of a set can be turned into a compatible renaming (with finite domain) on the full set.

lemma *bij-betw-extend*:
  assumes $*$: *bij-betw* f A B
    and $A \subseteq V$

```
    and B ⊆ V
    and finite A
  shows ∃ g. finite {x. g x ≠ x} ∧
    (∀ x∈UNIV − (A ∪ B). g x = x) ∧
    (∀ x∈A. g x = f x) ∧
    bij-betw g V V
proof −
  have finite B using assms by (metis bij-betw-finite)
  have [simp]: card A = card B by (metis ∗ bij-betw-same-card)
  have card (A − B) = card (B − A)
  proof −
    have card (A − B) = card A − card (A ∩ B)
      by (metis ‹finite A› card-Diff-subset-Int finite-Int)
    moreover have card (B − A) = card B − card (A ∩ B)
      by (metis ‹finite A› card-Diff-subset-Int finite-Int inf-commute)
    ultimately show ?thesis by simp
  qed
  then obtain g where ∗∗: bij-betw g (B − A) (A − B)
    by (metis ‹finite A› ‹finite B› bij-betw-iff-card finite-Diff)
  define h where h = (λx. if x ∈ A then f x else if x ∈ B − A then g x else x)
  have bij-betw h A B
    by (metis (full-types) ∗ bij-betw-cong h-def)
  moreover have bij-betw h (V − (A ∪ B)) (V − (A ∪ B))
    by (auto simp: bij-betw-def h-def inj-on-def)
  moreover have B ∩ (V − (A ∪ B)) = {} by blast
  ultimately have bij-betw h (A ∪ (V − (A ∪ B))) (B ∪ (V − (A ∪ B)))
    by (rule bij-betw-combine)
  moreover have A ∪ (V − (A ∪ B)) = V − (B − A)
    and B ∪ (V − (A ∪ B)) = V − (A − B)
    using ‹A ⊆ V› and ‹B ⊆ V› by blast+
  ultimately have bij-betw h (V − (B − A)) (V − (A − B)) by simp
  moreover have bij-betw h (B − A) (A − B)
    using ∗∗ by (auto simp: bij-betw-def h-def inj-on-def)
  moreover have (V − (A − B)) ∩ (A − B) = {} by blast
  ultimately have bij-betw h ((V − (B − A)) ∪ (B − A)) ((V − (A − B)) ∪ (A
− B))
    by (rule bij-betw-combine)
  moreover have (V − (B − A)) ∪ (B − A) = V
    and (V − (A − B)) ∪ (A − B) = V
    using ‹A ⊆ V› and ‹B ⊆ V› by auto
  ultimately have bij-betw h V V by simp
  moreover have ∀ x∈A. h x = f x by (auto simp: h-def)
  moreover have finite {x. h x ≠ x}
  proof −
    have finite (A ∪ (B − A)) using ‹finite A› and ‹finite B› by auto
    moreover have {x. h x ≠ x} ⊆ (A ∪ (B − A)) by (auto simp: h-def)
    ultimately show ?thesis by (metis finite-subset)
  qed
  moreover have ∀ x∈UNIV − (A ∪ B). h x = x by (simp add: h-def)
```

ultimately show *?thesis* **by** *blast*
qed

2.7 Merging Functions

definition *fun-merge* :: $('a \Rightarrow 'b) list \Rightarrow 'a\ set\ list \Rightarrow 'a \Rightarrow 'b$
where
fun-merge fs as a = $(fs\ !\ (LEAST\ i.\ i < length\ as \wedge a \in as\ !\ i))\ a$

lemma *fun-merge-eq-nth*:
assumes *i*: $i < length\ as$
and *a*: $a \in as\ !\ i$
and *ident*: $\bigwedge i\ j\ a.\ i < length\ as \Longrightarrow j < length\ as \Longrightarrow a \in as\ !\ i \Longrightarrow a \in as\ !\ j \Longrightarrow (fs\ !\ i)\ a = (fs\ !\ j)\ a$
shows *fun-merge fs as a* = $(fs\ !\ i)\ a$
proof –
let $?p = \lambda\ i.\ i < length\ as \wedge a \in as\ !\ i$
let $?l = LEAST\ i.\ ?p\ i$
have *p*: $?p\ ?l$
by (*rule LeastI*, *insert i a*, *auto*)
show *?thesis* **unfolding** *fun-merge-def*
by (*rule ident*[*OF - i - a*], *insert p*, *auto*)
qed

lemma *fun-merge-part*:
assumes $\forall\, i<length\ as.\forall\, j<length\ as.\ i \neq j \longrightarrow as\ !\ i \cap as\ !\ j = \{\}$
and $i < length\ as$
and $a \in as\ !\ i$
shows *fun-merge fs as a* = $(fs\ !\ i)\ a$
proof(*rule fun-merge-eq-nth* [*OF assms*(*2*, *3*)])
fix *i j a*
assume $i < length\ as$ **and** $j < length\ as$ **and** $a \in as\ !\ i$ **and** $a \in as\ !\ j$
then have $i = j$ **using** *assms* **by** (*cases* $i = j$) *auto*
then show $(fs\ !\ i)\ a = (fs\ !\ j)\ a$ **by** *simp*
qed

lemma *fun-merge*:
assumes *part*: $\forall\, i<length\ Xs.\forall\, j<length\ Xs.\ i \neq j \longrightarrow Xs\ !\ i \cap Xs\ !\ j = \{\}$
shows $\exists\, \sigma.\ \forall\, i<length\ Xs.\ \forall\, x\in\ Xs\ !\ i.\ \sigma\ x = \tau\ i\ x$
proof –
let $?\tau = map\ \tau\ [0\ ..<\ length\ Xs]$
let $?\sigma = fun\text{-}merge\ ?\tau\ Xs$
show *?thesis*
by (*rule exI*[*of - ?σ*], *intro allI impI ballI*,
insert fun-merge-part[*OF part*, *of - - ?τ*], *auto*)
qed

end

2.8 The Option Monad

theory *Option-Monad*
  imports *HOL−Library.Monad-Syntax*
begin

declare *Option.bind-cong* [*fundef-cong*]

definition *guard* :: *bool* ⇒ *unit option*
  where
    guard b = (*if b then Some* () *else None*)

lemma *guard-cong* [*fundef-cong*]:
  $b = c \Longrightarrow (c \Longrightarrow m = n) \Longrightarrow guard\ b >> m = guard\ c >> n$
  by (*simp add*: *guard-def*)

lemma *guard-simps*:
  $guard\ b = Some\ x \longleftrightarrow b$
  $guard\ b = None \longleftrightarrow \neg\ b$
  by (*cases b*) (*simp-all add*: *guard-def*)

lemma *guard-elims*[*elim*]:
  $guard\ b = Some\ x \Longrightarrow (b \Longrightarrow P) \Longrightarrow P$
  $guard\ b = None \Longrightarrow (\neg\ b \Longrightarrow P) \Longrightarrow P$
  by (*simp-all add*: *guard-simps*)

lemma *guard-intros* [*intro*, *simp*]:
  $b \Longrightarrow guard\ b = Some\ ()$
  $\neg\ b \Longrightarrow guard\ b = None$
  by (*simp-all add*: *guard-simps*)

lemma *guard-True* [*simp*]: *guard True* = *Some* () **by** *simp*
lemma *guard-False* [*simp*]: *guard False* = *None* **by** *simp*

lemma *guard-and-to-bind*: $guard\ (a \wedge b) = guard\ a \gg\!= (\lambda\ \text{-}.\ guard\ b)$ **by** (*cases a*; *cases b*; *auto*)

fun *zip-option* :: $'a\ list \Rightarrow\ 'b\ list \Rightarrow ('a \times\ 'b)\ list\ option$
  where
    zip-option [] [] = *Some* []
  | *zip-option* (*x*#*xs*) (*y*#*ys*) = *do* { *zs* ← *zip-option xs ys*; *Some* ((*x*, *y*) # *zs*) }
  | *zip-option* (*x*#*xs*) [] = *None*
  | *zip-option* [] (*y*#*ys*) = *None*

induction scheme for zip

lemma *zip-induct* [*case-names Cons-Cons Nil1 Nil2*]:
  assumes $\bigwedge x\ xs\ y\ ys.\ P\ xs\ ys \Longrightarrow P\ (x\ \#\ xs)\ (y\ \#\ ys)$
    and $\bigwedge ys.\ P\ []\ ys$
    and $\bigwedge xs.\ P\ xs\ []$
  shows *P xs ys*

using *assms* **by** (*induction-schema*) (*pat-completeness*, *lexicographic-order*)

lemma *zip-option-same*[*simp*]:
zip-option xs xs = *Some* (*zip xs xs*)
by (*induction xs*) *simp-all*

lemma *zip-option-zip-conv*:
zip-option xs ys = *Some zs* $\longleftrightarrow$ *length ys* = *length xs* $\wedge$ *length zs* = *length xs* $\wedge$ *zs* = *zip xs ys*
proof −
{
assume *zip-option xs ys* = *Some zs*
hence *length ys* = *length xs* $\wedge$ *length zs* = *length xs* $\wedge$ *zs* = *zip xs ys*
proof (*induct xs ys arbitrary*: *zs rule*: *zip-option.induct*)
case (*2 x xs y ys*)
then obtain *zs'* **where** *zip-option xs ys* = *Some zs'*
and *zs* = (*x*, *y*) # *zs'* **by** (*cases zip-option xs ys*) *auto*
from *2*(*1*) [*OF this*(*1*)] **and** *this*(*2*) **show** *?case* **by** *simp*
qed *simp-all*
} **moreover** {
assume *length ys* = *length xs* **and** *zs* = *zip xs ys*
hence *zip-option xs ys* = *Some zs*
by (*induct xs ys arbitrary*: *zs rule*: *zip-induct*) *force*+
}
ultimately show *?thesis* **by** *blast*
qed

lemma *zip-option-None*:
zip-option xs ys = *None* $\longleftrightarrow$ *length xs* $\neq$ *length ys*
proof −
{
assume *zip-option xs ys* = *None*
have *length xs* $\neq$ *length ys*
proof (*rule ccontr*)
assume ¬ *length xs* $\neq$ *length ys*
hence *length xs* = *length ys* **by** *simp*
hence *zip-option xs ys* = *Some* (*zip xs ys*) **by** (*simp add*: *zip-option-zip-conv*)
with ‹*zip-option xs ys* = *None*› **show** *False* **by** *simp*
qed
} **moreover** {
assume *length xs* $\neq$ *length ys*
hence *zip-option xs ys* = *None*
by (*induct xs ys rule*: *zip-option.induct*) *simp-all*
}
ultimately show *?thesis* **by** *blast*
qed

declare *zip-option.simps* [*simp del*]

lemma *zip-option-intros* [*intro*]:
$[\![$*length ys* = *length xs*; *length zs* = *length xs*; *zs* = *zip xs ys*$]\!]$
$\Longrightarrow$ *zip-option xs ys* = *Some zs*
length xs $\neq$ *length ys* $\Longrightarrow$ *zip-option xs ys* = *None*
by (*simp-all add*: *zip-option-zip-conv zip-option-None*)

lemma *zip-option-elims* [*elim*]:
zip-option xs ys = *Some zs*
$\Longrightarrow$ ($[\![$*length ys* = *length xs*; *length zs* = *length xs*; *zs* = *zip xs ys*$]\!]$ $\Longrightarrow$ *P*)
$\Longrightarrow$ *P*
zip-option xs ys = *None* $\Longrightarrow$ (*length xs* $\neq$ *length ys* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
by (*simp-all add*: *zip-option-zip-conv zip-option-None*)

lemma *zip-option-simps* [*simp*]:
zip-option xs ys = *None* $\Longrightarrow$ *length xs* = *length ys* $\Longrightarrow$ *False*
zip-option xs ys = *None* $\Longrightarrow$ *length xs* $\neq$ *length ys*
zip-option xs ys = *Some zs* $\Longrightarrow$ *zs* = *zip xs ys*
by (*simp-all add*: *zip-option-None zip-option-zip-conv*)

fun *mapM* :: ($'a \Rightarrow 'b$ *option*) $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'b$ *list option*
where
mapM f [] = *Some* []
| *mapM f* (*x#xs*) = *do* {
y $\leftarrow$ *f x*;
ys $\leftarrow$ *mapM f xs*;
Some (*y* # *ys*)
}

lemma *mapM-None*:
mapM f xs = *None* $\longleftrightarrow$ ($\exists$ *x*$\in$*set xs*. *f x* = *None*)
proof (*induct xs*)
case (*Cons x xs*) **thus** *?case*
by (*cases f x*, *simp*, *cases mapM f xs*, *auto*)
qed *simp*

lemma *mapM-Some*:
mapM f xs = *Some ys* $\Longrightarrow$ *ys* = *map* (λx. *the* (*f x*)) *xs* $\wedge$ ($\forall$ *x*$\in$*set xs*. *f x* $\neq$ *None*)
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs ys*)
thus *?case*
by (*cases f x*, *simp*, *cases mapM f xs*, *auto*)
qed *simp*

lemma *mapM-Some-idx*:
assumes *some*: *mapM f xs* = *Some ys* **and** *i*: *i* < *length xs*
shows $\exists$ *y*. *f* (*xs* ! *i*) = *Some y* $\wedge$ *ys* ! *i* = *y*
proof −
note *m* = *mapM-Some* [*OF some*]

from m[*unfolded set-conv-nth*] i **have** f (xs ! i) $\neq$ *None* **by** *auto*
then obtain y **where** f (xs ! i) = *Some* y **by** *auto*
then have f (xs ! i) = *Some* $y \wedge ys$! $i = y$ **unfolding** m [*THEN conjunct1*] **using** i **by** *auto*
then show *?thesis* **..**
qed

lemma *mapM-cong* [*fundef-cong*]:
assumes $xs = ys$ **and** $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows *mapM f xs* = *mapM g ys*
unfolding *assms*(*1*) **using** *assms*(*2*) **by** (*induct ys*) *auto*

lemma *mapM-map*:
mapM f xs = (*if* ($\forall x \in set\ xs.\ f\ x \neq None$) *then Some* (*map* ($\lambda x.$ *the* ($f\ x$)) *xs*) *else None*)
proof (*cases mapM f xs*)
case *None*
thus *?thesis* **using** *mapM-None* **by** *auto*
next
case (*Some ys*)
with *mapM-Some* [*OF Some*] **show** *?thesis* **by** *auto*
qed

lemma *mapM-mono* [*partial-function-mono*]:
fixes $C :: {}'a \Rightarrow ({}'b \Rightarrow {}'c\ option) \Rightarrow {}'d\ option$
assumes C: $\bigwedge y.$ *mono-option* ($C\ y$)
shows *mono-option* ($\lambda f.$ *mapM* ($\lambda y.\ C\ y\ f$) B)
proof (*induct B*)
case *Nil*
show *?case* **unfolding** *mapM.simps*
by (*rule option.const-mono*)
next
case (*Cons b B*)
show *?case* **unfolding** *mapM.simps*
by (*rule bind-mono* [*OF C bind-mono* [*OF Cons option.const-mono*]])
qed

end

3 First-Order Terms

theory *Term*
imports
Main
HOL−Library.Multiset
begin

datatype (*funs-term* : $'f$, *vars-term* : $'v$) *term* =
is-Var: *Var* (*the-Var*: $'v$) |

Fun $'f$ (*args* : ($'f$, $'v$) *term list*)
where
args (*Var* -) = []

abbreviation *is-Fun t* $\equiv \neg$ *is-Var t*

lemma *is-VarE* [*elim*]:
is-Var t $\Longrightarrow$ ($\bigwedge x$. *t* = *Var x* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
by (*cases t*) *auto*

lemma *is-FunE* [*elim*]:
is-Fun t $\Longrightarrow$ ($\bigwedge f$ *ts*. *t* = *Fun f ts* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
by (*cases t*) *auto*

lemma *inj-on-Var*[*simp*]:
inj-on Var A
by (*rule inj-onI*) *simp*

lemma *member-image-the-Var-image-subst*:
assumes *is-var-*σ: $\forall x$. *is-Var* (σ *x*)
shows *x* $\in$ *the-Var* ‘ σ ‘ *V* $\longleftrightarrow$ *Var x* $\in$ σ ‘ *V*
using *is-var-*σ *image-iff*
by (*metis* (*no-types*, *opaque-lifting*) *term.collapse*(*1*) *term.sel*(*1*))

lemma *image-the-Var-image-subst-renaming-eq*:
assumes *is-var-*σ: $\forall x$. *is-Var* (ϱ *x*)
shows *the-Var* ‘ ϱ ‘ *V* = ($\bigcup x \in V$. *vars-term* (ϱ *x*))
proof (*rule Set.equalityI*; *rule Set.subsetI*)
from *is-var-*σ **show** $\bigwedge x$. *x* $\in$ *the-Var* ‘ ϱ ‘ *V* $\Longrightarrow$ *x* $\in$ ($\bigcup x \in V$. *vars-term* (ϱ *x*))
using *term.set-sel*(*3*) **by** *force*
next
from *is-var-*σ **show** $\bigwedge x$. *x* $\in$ ($\bigcup x \in V$. *vars-term* (ϱ *x*)) $\Longrightarrow$ *x* $\in$ *the-Var* ‘ ϱ ‘ *V*
by (*smt* (*verit*, *best*) *Term.term.simps*(*17*) *UN-iff image-eqI singletonD term.collapse*(*1*))
qed

The variables of a term as multiset.

fun *vars-term-ms* :: ($'f$, $'v$) *term* $\Rightarrow$ $'v$ *multiset*
where
vars-term-ms (*Var x*) = {#*x*#} |
vars-term-ms (*Fun f ts*) = $\sum_{\#}$ (*mset* (*map vars-term-ms ts*))

lemma *set-mset-vars-term-ms* [*simp*]:
set-mset (*vars-term-ms t*) = *vars-term t*
by (*induct t*) *auto*

Reorient equations of the form *Var x* = *t* and *Fun f ss* = *t* to facilitate simplification.

setup ‹
Reorient-Proc.add

  (*fn Const* (@{*const-name Var*}, -) $ - => *true* | - => *false*)
  #> *Reorient-Proc.add*
  (*fn Const* (@{*const-name Fun*}, -) $ - $ - => *true* | - => *false*)
›

simproc-setup *reorient-Var* (*Var x* = *t*) = ‹*K Reorient-Proc.proc*›
simproc-setup *reorient-Fun* (*Fun f ss* = *t*) = ‹*K Reorient-Proc.proc*›

The *root symbol* of a term is defined by:

fun *root* :: (′*f*, ′*v*) *term* ⇒ (′*f* × *nat*) *option*
where
  root (*Var x*) = *None* |
  root (*Fun f ts*) = *Some* (*f*, *length ts*)

lemma *finite-vars-term* [*simp*]:
  finite (*vars-term t*)
  by (*induct t*) *simp-all*

lemma *finite-Union-vars-term*:
  finite (⋃ *t* ∈ *set ts*. *vars-term t*)
  by *auto*

We define the evaluation of terms, under interpretation of function symbols and assignment of variables, as follows:

fun *eval-term* (-⟦(2-)⟧- [*999*,*1*,*100*]*100*) **where**
  I⟦*Var x*⟧α = α *x*
| *I*⟦*Fun f ss*⟧α = *I f* [*I*⟦*s*⟧α. *s* ← *ss*]

notation *eval-term* (-⟦(2-)⟧ [*999*,*1*]*100*)
notation *eval-term* (-⟦(2-)⟧- [*999*,*1*,*100*]*100*)

lemma *eval-same-vars*:
  assumes ∀ *x* ∈ *vars-term s*. α *x* = β *x*
  shows *I*⟦*s*⟧α = *I*⟦*s*⟧β
  by (*insert assms*, *induct s*, *auto intro*!:*map-cong*[*OF refl*] *cong*[*of I* -])

lemma *eval-same-vars-cong*:
  assumes *ref*: *s* = *t* **and** *v*: ⋀*x*. *x* ∈ *vars-term s* ⟹ α *x* = β *x*
  shows *I*⟦*s*⟧α = *I*⟦*t*⟧β
  by (*fold ref*, *rule eval-same-vars*, *auto dest*:*v*)

lemma *eval-with-fresh-var*: *x* ∉ *vars-term s* ⟹ *I*⟦*s*⟧α(*x*:=*a*) = *I*⟦*s*⟧α
  by (*auto intro*: *eval-same-vars*)

lemma *eval-map-term*: *I*⟦*map-term ff fv s*⟧α = (*I* ∘ *ff*)⟦*s*⟧(α ∘ *fv*)
  by (*induct s*, *auto intro*: *cong*[*of I* -])

A substitution is a mapping σ from variables to terms. We call a substitution that alters the type of variables a generalized substitution, since it does not have all properties that are expected of (standard) substitutions (e.g., there is no empty substitution).

type-synonym $('f, 'v, 'w)$ *gsubst* $= 'v \Rightarrow ('f, 'w)$ *term*
type-synonym $('f, 'v)$ *subst* $= ('f, 'v, 'v)$ *gsubst*

abbreviation *subst-apply-term* :: $('f, 'v)$ *term* $\Rightarrow ('f, 'v, 'w)$ *gsubst* $\Rightarrow ('f, 'w)$ *term* (**infixl** $\cdot$ *67*)
 where *subst-apply-term* $\equiv$ *eval-term Fun*

definition
 subst-compose :: $('f, 'u, 'v)$ *gsubst* $\Rightarrow ('f, 'v, 'w)$ *gsubst* $\Rightarrow ('f, 'u, 'w)$ *gsubst*
 (**infixl** $\circ_s$ *75*)
 where
  $\sigma \circ_s \tau = (\lambda x.\ (\sigma\ x) \cdot \tau)$

lemma *subst-subst-compose* [*simp*]:
 $t \cdot (\sigma \circ_s \tau) = t \cdot \sigma \cdot \tau$
 by (*induct t*) (*simp-all add*: *subst-compose-def*)

lemma *subst-compose-assoc*:
 $\sigma \circ_s \tau \circ_s \mu = \sigma \circ_s (\tau \circ_s \mu)$
proof (*rule ext*)
 fix x **show** $(\sigma \circ_s \tau \circ_s \mu)\ x = (\sigma \circ_s (\tau \circ_s \mu))\ x$
 proof −
  have $(\sigma \circ_s \tau \circ_s \mu)\ x = \sigma(x) \cdot \tau \cdot \mu$ **by** (*simp add*: *subst-compose-def*)
  also have $\ldots = \sigma(x) \cdot (\tau \circ_s \mu)$ **by** *simp*
  finally show *?thesis* **by** (*simp add*: *subst-compose-def*)
 qed
qed

lemma *subst-apply-term-empty* [*simp*]:
 $t \cdot$ *Var* $= t$
proof (*induct t*)
 case (*Fun f ts*)
 from *map-ext* [*rule-format*, *of ts - id*, *OF Fun*] **show** *?case* **by** *simp*
qed *simp*

interpretation *subst-monoid-mult*: *monoid-mult Var* $(\circ_s)$
 by (*unfold-locales*) (*simp add*: *subst-compose-assoc*, *simp-all add*: *subst-compose-def*)

lemma *term-subst-eq*:
 assumes $\bigwedge x.\ x \in$ *vars-term* $t \Longrightarrow \sigma\ x = \tau\ x$
 shows $t \cdot \sigma = t \cdot \tau$
 using *assms* **by** (*induct t*) (*auto*)

lemma *term-subst-eq-rev*:
 $t \cdot \sigma = t \cdot \tau \Longrightarrow \forall x \in$ *vars-term* $t.\ \sigma\ x = \tau\ x$
 by (*induct t*) *simp-all*

lemma *term-subst-eq-conv*:
  $t \cdot \sigma = t \cdot \tau \longleftrightarrow (\forall x \in \mathit{vars\text{-}term}\ t.\ \sigma\ x = \tau\ x)$
  by (*auto intro*!: *term-subst-eq term-subst-eq-rev*)

lemma *subst-term-eqI*:
  assumes $(\bigwedge t.\ t \cdot \sigma = t \cdot \tau)$
  shows $\sigma = \tau$
  using *assms* [*of Var x* **for** *x*] **by** (*intro ext*) *simp*

definition *subst-domain* :: $('f, 'v)\ \mathit{subst} \Rightarrow 'v\ \mathit{set}$
  where
    $\mathit{subst\text{-}domain}\ \sigma = \{x.\ \sigma\ x \neq \mathit{Var}\ x\}$

fun *subst-range* :: $('f, 'v)\ \mathit{subst} \Rightarrow ('f, 'v)\ \mathit{term}\ \mathit{set}$
  where
    $\mathit{subst\text{-}range}\ \sigma = \sigma\ `\ \mathit{subst\text{-}domain}\ \sigma$

lemma *vars-term-ms-subst* [*simp*]:
  $\mathit{vars\text{-}term\text{-}ms}\ (t \cdot \sigma) =$
    $(\sum x{\in}\#\mathit{vars\text{-}term\text{-}ms}\ t.\ \mathit{vars\text{-}term\text{-}ms}\ (\sigma\ x))$ (**is** - = *?V t*)
proof (*induct t*)
  case (*Fun f ts*)
  have *IH*: $\mathit{map}\ (\lambda\ t.\ \mathit{vars\text{-}term\text{-}ms}\ (t \cdot \sigma))\ \mathit{ts} = \mathit{map}\ (\lambda\ t.\ ?V\ t)\ \mathit{ts}$
    by (*rule map-cong*[*OF refl Fun*])
  show *?case* **by** (*simp add*: *o-def IH*, *induct ts*, *auto*)
qed *simp*

lemma *vars-term-ms-subst-mono*:
  assumes $\mathit{vars\text{-}term\text{-}ms}\ s \subseteq\#\ \mathit{vars\text{-}term\text{-}ms}\ t$
  shows $\mathit{vars\text{-}term\text{-}ms}\ (s \cdot \sigma) \subseteq\#\ \mathit{vars\text{-}term\text{-}ms}\ (t \cdot \sigma)$
proof −
 from *assms*[*unfolded mset-subset-eq-exists-conv*] **obtain** *u* **where** *t*: $\mathit{vars\text{-}term\text{-}ms}\ t = \mathit{vars\text{-}term\text{-}ms}\ s + u$ **by** *auto*
  show *?thesis* **unfolding** *vars-term-ms-subst* **unfolding** *t* **by** *auto*
qed

The variables introduced by a substitution.

definition *range-vars* :: $('f, 'v)\ \mathit{subst} \Rightarrow 'v\ \mathit{set}$
where
  $\mathit{range\text{-}vars}\ \sigma = \bigcup(\mathit{vars\text{-}term}\ `\ \mathit{subst\text{-}range}\ \sigma)$

lemma *mem-range-varsI*:
  assumes $\sigma\ x = \mathit{Var}\ y$ **and** $x \neq y$
  shows $y \in \mathit{range\text{-}vars}\ \sigma$
  unfolding *range-vars-def UN-iff*
proof (*rule bexI*[*of - Var y*])
  show $y \in \mathit{vars\text{-}term}\ (\mathit{Var}\ y)$
    by *simp*
next

  from *assms* **show** *Var* $y \in$ *subst-range* σ
    by (*simp-all add*: *subst-domain-def*)
qed

lemma *subst-domain-Var* [*simp*]:
  subst-domain Var $= \{\}$
  by (*simp add*: *subst-domain-def*)

lemma *subst-range-Var*[*simp*]:
  subst-range Var $= \{\}$
  by *simp*

lemma *range-vars-Var*[*simp*]:
  range-vars Var $= \{\}$
  by (*simp add*: *range-vars-def*)

lemma *subst-apply-term-ident*:
  vars-term $t \cap$ *subst-domain* $\sigma = \{\} \Longrightarrow t \cdot \sigma = t$
proof (*induction t*)
  case (*Var x*)
  thus *?case*
    by (*simp add*: *subst-domain-def*)
next
  case (*Fun f ts*)
  thus *?case*
    by (*auto intro*: *list.map-ident-strong*)
qed

lemma *vars-term-subst-apply-term*:
  vars-term $(t \cdot \sigma) = (\bigcup x \in$ *vars-term* $t.$ *vars-term* $(\sigma\ x))$
  by (*induction t*) (*auto simp add*: *insert-Diff-if subst-domain-def*)

corollary *vars-term-subst-apply-term-subset*:
  vars-term $(t \cdot \sigma) \subseteq$ *vars-term* $t -$ *subst-domain* $\sigma \cup$ *range-vars* σ
  unfolding *vars-term-subst-apply-term*
proof (*induction t*)
  case (*Var x*)
  show *?case*
    by (*cases* $\sigma\ x =$ *Var x*) (*auto simp add*: *range-vars-def subst-domain-def*)
next
  case (*Fun f xs*)
  thus *?case* **by** *auto*
qed

definition *is-renaming* :: ($'f$, $'v$) *subst* $\Rightarrow$ *bool*
  where
    is-renaming $\sigma \longleftrightarrow (\forall x.$ *is-Var* $(\sigma\ x)) \wedge$ *inj-on* σ (*subst-domain* σ)

lemma *inv-renaming-sound*:

assumes *is-var-ϱ*: $\forall x.$ *is-Var* (ϱ x) **and** *inj* ϱ
shows $\varrho \circ_s$ (*Var* $\circ$ (*inv* (*the-Var* $\circ$ ϱ))) = *Var*
proof −
define ϱ' **where** ϱ' = *the-Var* $\circ$ ϱ
have *ϱ-def*: ϱ = *Var* $\circ$ ϱ'
unfolding *ϱ′-def* **using** *is-var-ϱ* **by** *auto*

from *is-var-ϱ* ‹*inj* ϱ› **have** *inj* ϱ'
unfolding *inj-def ϱ-def comp-def* **by** *fast*
hence *inv* $\varrho' \circ \varrho'$ = *id*
using *inv-o-cancel*[*of* ϱ'] **by** *simp*
hence *Var* $\circ$ (*inv* $\varrho' \circ \varrho'$) = *Var*
by *simp*
hence $\forall x.$ (*Var* $\circ$ (*inv* $\varrho' \circ \varrho'$)) x = *Var* x
by *metis*
hence $\forall x.$ ((*Var* $\circ$ ϱ') $\circ_s$ (*Var* $\circ$ (*inv* ϱ'))) x = *Var* x
unfolding *subst-compose-def* **by** *auto*
thus $\varrho \circ_s$ (*Var* $\circ$ (*inv* ϱ')) = *Var*
using *ϱ-def* **by** *auto*
qed

lemma *ex-inverse-of-renaming*:
assumes $\forall x.$ *is-Var* (ϱ x) **and** *inj* ϱ
shows $\exists \tau.$ $\varrho \circ_s \tau$ = *Var*
using *inv-renaming-sound*[*OF assms*] **by** *blast*

lemma *vars-term-subst*:
vars-term ($t \cdot \sigma$) = $\bigcup$(*vars-term* ‘ σ ‘ *vars-term* t)
by (*induct* t) *simp-all*

lemma *range-varsE* [*elim*]:
assumes $x \in$ *range-vars* σ
and $\bigwedge t.$ $x \in$ *vars-term* $t \Longrightarrow t \in$ *subst-range* $\sigma \Longrightarrow P$
shows P
using *assms* **by** (*auto simp*: *range-vars-def*)

lemma *range-vars-subst-compose-subset*:
range-vars ($\sigma \circ_s \tau$) $\subseteq$ (*range-vars* σ − *subst-domain* τ) $\cup$ *range-vars* τ (**is** *?L* $\subseteq$ *?R*)
proof
fix x
assume $x \in$ *?L*
then obtain y **where** $y \in$ *subst-domain* ($\sigma \circ_s \tau$)
and $x \in$ *vars-term* (($\sigma \circ_s \tau$) y) **by** (*auto simp*: *range-vars-def*)
then show $x \in$ *?R*
proof (*cases*)
assume $y \in$ *subst-domain* σ **and** $x \in$ *vars-term* (($\sigma \circ_s \tau$) y)
moreover then obtain v **where** $v \in$ *vars-term* (σ y)
and $x \in$ *vars-term* (τ v) **by** (*auto simp*: *subst-compose-def vars-term-subst*)

ultimately show *?thesis*
by (*cases* $v \in$ *subst-domain* τ) (*auto simp*: *range-vars-def subst-domain-def*)
qed (*auto simp*: *range-vars-def subst-compose-def subst-domain-def*)
qed

definition *subst* x t = *Var* (x := t)

lemma *subst-simps* [*simp*]:
subst x t x = t
subst x (*Var* x) = *Var*
by (*auto simp*: *subst-def*)

lemma *subst-subst-domain* [*simp*]:
subst-domain (*subst* x t) = (*if* t = *Var* x *then* {} *else* {x})
proof −
{ fix y
have $y \in \{y.$ *subst* x t $y \neq$ *Var* $y\} \longleftrightarrow y \in$ (*if* t = *Var* x *then* {} *else* {x})
by (*cases* $x = y$, *auto simp*: *subst-def*) **}**
then show *?thesis* **by** (*simp add*: *subst-domain-def*)
qed

lemma *subst-subst-range* [*simp*]:
subst-range (*subst* x t) = (*if* t = *Var* x *then* {} *else* {t})
by (*cases* t = *Var* x) (*auto simp*: *subst-domain-def subst-def*)

lemma *subst-apply-left-idemp* [*simp*]:
assumes $\sigma\ x = t \cdot \sigma$
shows $s \cdot$ *subst* x $t \cdot \sigma = s \cdot \sigma$
using *assms* **by** (*induct* s) (*auto simp*: *subst-def*)

lemma *subst-compose-left-idemp* [*simp*]:
assumes $\sigma\ x = t \cdot \sigma$
shows *subst* x $t \circ_s \sigma = \sigma$
by (*rule subst-term-eqI*) (*simp add*: *assms*)

lemma *subst-ident* [*simp*]:
assumes $x \notin$ *vars-term* t
shows $t \cdot$ *subst* x $u = t$
proof −
have $t \cdot$ *subst* x $u = t \cdot$ *Var*
by (*rule term-subst-eq*) (*auto simp*: *assms subst-def*)
then show *?thesis* **by** *simp*
qed

lemma *subst-self-idemp* [*simp*]:
$x \notin$ *vars-term* $t \Longrightarrow$ *subst* x $t \circ_s$ *subst* x t = *subst* x t
by (*metis subst-simps*(*1*) *subst-compose-left-idemp subst-ident*)

type-synonym ($'f$, $'v$) *terms* = ($'f$, $'v$) *term set*

Applying a substitution to every term of a given set.

abbreviation
subst-apply-set :: $('f, 'v)$ *terms* $\Rightarrow$ $('f, 'v, 'w)$ *gsubst* $\Rightarrow$ $('f, 'w)$ *terms* (**infixl** $\cdot_{set}$ *60*)
where
$T \cdot_{set} \sigma \equiv (\lambda t.\ t \cdot \sigma)$ ‘ T

Composition of substitutions

lemma *subst-compose*: $(\sigma \circ_s \tau)\ x = \sigma\ x \cdot \tau$ **by** (*auto simp*: *subst-compose-def*)

lemmas *subst-subst* = *subst-subst-compose* [*symmetric*]

lemma *subst-apply-eq-Var*:
assumes $s \cdot \sigma = Var\ x$
obtains y **where** $s = Var\ y$ **and** $\sigma\ y = Var\ x$
using *assms* **by** (*induct s*) *auto*

lemma *subst-domain-subst-compose*:
subst-domain $(\sigma \circ_s \tau)$ =
(*subst-domain* $\sigma - \{x.\ \exists y.\ \sigma\ x = Var\ y \wedge \tau\ y = Var\ x\}) \cup$
(*subst-domain* τ − *subst-domain* σ)
by (*auto simp*: *subst-domain-def subst-compose-def elim*: *subst-apply-eq-Var*)

A substitution is idempotent iff the variables in its range are disjoint from its domain. (See also "Term Rewriting and All That" [1, Lemma 4.5.7].)

lemma *subst-idemp-iff*:
$\sigma \circ_s \sigma = \sigma \longleftrightarrow$ *subst-domain* $\sigma \cap$ *range-vars* $\sigma = \{\}$
proof
assume $\sigma \circ_s \sigma = \sigma$
then have $\bigwedge x.\ \sigma\ x \cdot \sigma = \sigma\ x \cdot Var$ **by** *simp* (*metis subst-compose-def*)
then have $*$: $\bigwedge x.\ \forall y \in$*vars-term* $(\sigma\ x).\ \sigma\ y = Var\ y$
unfolding *term-subst-eq-conv* **by** *simp*
{ fix $x\ y$
assume $\sigma\ x \neq Var\ x$ **and** $x \in$ *vars-term* $(\sigma\ y)$
with $*$ [*of y*] **have** *False* **by** *simp* **}**
then show *subst-domain* $\sigma \cap$ *range-vars* $\sigma = \{\}$
by (*auto simp*: *subst-domain-def range-vars-def*)
next
assume *subst-domain* $\sigma \cap$ *range-vars* $\sigma = \{\}$
then have $*$: $\bigwedge x\ y.\ \sigma\ x = Var\ x \vee \sigma\ y = Var\ y \vee x \notin$ *vars-term* $(\sigma\ y)$
by (*auto simp*: *subst-domain-def range-vars-def*)
have $\bigwedge x.\ \forall y \in$*vars-term* $(\sigma\ x).\ \sigma\ y = Var\ y$
proof
fix $x\ y$
assume $y \in$ *vars-term* $(\sigma\ x)$
with $*$ [*of y x*] **show** $\sigma\ y = Var\ y$ **by** *auto*
qed
then show $\sigma \circ_s \sigma = \sigma$

```
    by (simp add: subst-compose-def term-subst-eq-conv [symmetric])
qed

lemma subst-compose-apply-eq-apply-lhs:
  assumes
    range-vars σ ∩ subst-domain δ = {}
    x ∉ subst-domain δ
  shows (σ ∘s δ) x = σ x
proof (cases σ x)
  case (Var y)
  show ?thesis
  proof (cases x = y)
    case True
    with Var have ‹σ x = Var x›
      by simp
    moreover from ‹x ∉ subst-domain δ› have δ x = Var x
      by (simp add: disjoint-iff subst-domain-def)
    ultimately show ?thesis
      by (simp add: subst-compose-def)
  next
    case False
    have y ∈ range-vars σ
      unfolding range-vars-def UN-iff
    proof (rule bexI)
      show y ∈ vars-term (Var y)
        by simp
    next
      from Var False show Var y ∈ subst-range σ
        by (simp-all add: subst-domain-def)
    qed
    hence y ∉ subst-domain δ
      using ‹range-vars σ ∩ subst-domain δ = {}›
      by (simp add: disjoint-iff)
    with Var show ?thesis
      unfolding subst-compose-def
      by (simp add: subst-domain-def)
  qed
next
  case (Fun f ys)
  hence Fun f ys ∈ subst-range σ ∨ (∀ y∈set ys. y ∈ subst-range σ)
    using subst-domain-def by fastforce
  hence ∀ x ∈ vars-term (Fun f ys). x ∈ range-vars σ
    by (metis UN-I range-vars-def term.distinct(1) term.sel(4) term.set-cases(2))
  hence Fun f ys · δ = Fun f ys · Var
    unfolding term-subst-eq-conv
    using ‹range-vars σ ∩ subst-domain δ = {}›
    by (simp add: disjoint-iff subst-domain-def)
  from this[unfolded subst-apply-term-empty] Fun show ?thesis
    by (simp add: subst-compose-def)
```

qed

lemma *subst-apply-term-subst-apply-term-eq-subst-apply-term-lhs*:
  assumes *range-vars* $\sigma \cap$ *subst-domain* $\delta = \{\}$ **and** *vars-term* $t \cap$ *subst-domain* $\delta = \{\}$
  shows $t \cdot \sigma \cdot \delta = t \cdot \sigma$
proof –
  from *assms* **have** $\bigwedge x.\ x \in$ *vars-term* $t \Longrightarrow (\sigma \circ_s \delta)\ x = \sigma\ x$
    using *subst-compose-apply-eq-apply-lhs* **by** *fastforce*
  hence $t \cdot \sigma \circ_s \delta = t \cdot \sigma$
    using *term-subst-eq-conv* **by** *metis*
  thus *?thesis*
    by *simp*
qed

fun *num-funs* :: $('f, 'v)$ *term* $\Rightarrow$ *nat*
  where
    num-funs (*Var x*) = *0* |
    num-funs (*Fun f ts*) = *Suc* (*sum-list* (*map num-funs ts*))

lemma *num-funs-0*:
  assumes *num-funs* t = *0*
  obtains x **where** t = *Var x*
  using *assms* **by** (*induct t*) *auto*

lemma *num-funs-subst*:
  num-funs $(t \cdot \sigma) \geq$ *num-funs* t
  by (*induct t*) (*simp-all*, *metis comp-apply sum-list-mono*)

lemma *sum-list-map-num-funs-subst*:
  assumes *sum-list* (*map* (*num-funs* $\circ$ $(\lambda t.\ t \cdot \sigma)$) *ts*) = *sum-list* (*map num-funs ts*)
  shows $\forall\, i <$ *length ts*. *num-funs* (*ts* ! $i \cdot \sigma$) = *num-funs* (*ts* ! i)
  using *assms*
proof (*induct ts*)
  case (*Cons t ts*)
  then have *num-funs* $(t \cdot \sigma)$ + *sum-list* (*map* (*num-funs* $\circ$ $(\lambda t.\ t \cdot \sigma)$) *ts*)
    = *num-funs* t + *sum-list* (*map num-funs ts*) **by** (*simp add*: *o-def*)
  moreover have *num-funs* $(t \cdot \sigma) \geq$ *num-funs* t **by** (*metis num-funs-subst*)
  moreover have *sum-list* (*map* (*num-funs* $\circ$ $(\lambda t.\ t \cdot \sigma)$) *ts*) $\geq$ *sum-list* (*map num-funs ts*)
    using *num-funs-subst* [*of* - σ] **by** (*induct ts*) (*auto intro*: *add-mono*)
  ultimately show *?case* **using** *Cons* **by** (*auto*) (*case-tac i*, *auto*)
qed *simp*

lemma *is-Fun-num-funs-less*:
  assumes $x \in$ *vars-term* t **and** *is-Fun* t
  shows *num-funs* $(\sigma\ x) <$ *num-funs* $(t \cdot \sigma)$
  using *assms*

proof (*induct t*)
  case (*Fun f ts*)
  then obtain *u* **where** *u*: $u \in$ *set ts* $x \in$ *vars-term u* **by** *auto*
  then have *num-funs* $(u \cdot \sigma) \leq$ *sum-list* (*map* (*num-funs* $\circ$ ($\lambda t.\ t \cdot \sigma$)) *ts*)
    by (*intro member-le-sum-list*) *simp*
  moreover have *num-funs* $(\sigma\ x) \leq$ *num-funs* $(u \cdot \sigma)$
    using *Fun.hyps* [*OF u*] **and** *u* **by** (*cases u*; *simp*)
  ultimately show *?case* **by** *simp*
qed *simp*

lemma *finite-subst-domain-subst*:
  finite (*subst-domain* (*subst x y*))
  by *simp*

lemma *subst-domain-compose*:
  subst-domain $(\sigma \circ_s \tau) \subseteq$ *subst-domain* $\sigma \cup$ *subst-domain* τ
  by (*auto simp*: *subst-domain-def subst-compose-def*)

lemma *vars-term-disjoint-imp-unifier*:
  fixes σ :: ($'f$, $'v$, $'w$) *gsubst*
  assumes *vars-term* $s \cap$ *vars-term* $t = \{\}$
    and $s \cdot \sigma = t \cdot \tau$
  shows $\exists\, \mu$ :: ($'f$, $'v$, $'w$) *gsubst*. $s \cdot \mu = t \cdot \mu$
proof −
  let $?\mu = \lambda x.$ *if* $x \in$ *vars-term s then* $\sigma\ x$ *else* $\tau\ x$
  have $s \cdot \sigma = s \cdot ?\mu$
    unfolding *term-subst-eq-conv*
    by (*induct s*) (*simp-all*)
  moreover have $t \cdot \tau = t \cdot ?\mu$
    using *assms*(*1*)
    unfolding *term-subst-eq-conv*
    by (*induct s arbitrary*: *t*) (*auto*)
  ultimately have $s \cdot ?\mu = t \cdot ?\mu$ **using** *assms*(*2*) **by** *simp*
  then show *?thesis* **by** *blast*
qed

lemma *vars-term-subset-subst-eq*:
  assumes *vars-term* $t \subseteq$ *vars-term s*
    and $s \cdot \sigma = s \cdot \tau$
  shows $t \cdot \sigma = t \cdot \tau$
  using *assms* **by** (*induct t*) (*induct s*, *auto*)

3.1 Restrict the Domain of a Substitution

definition *restrict-subst-domain* **where**
  restrict-subst-domain V σ $x \equiv$ (*if* $x \in V$ *then* $\sigma\ x$ *else Var x*)

lemma *restrict-subst-domain-empty*[*simp*]:
  restrict-subst-domain $\{\}$ σ = *Var*

unfolding *restrict-subst-domain-def* **by** *auto*

lemma *restrict-subst-domain-Var*[*simp*]:
restrict-subst-domain V Var = Var
unfolding *restrict-subst-domain-def* **by** *auto*

lemma *subst-domain-restrict-subst-domain*[*simp*]:
subst-domain (*restrict-subst-domain V σ*) = *V* ∩ *subst-domain σ*
unfolding *restrict-subst-domain-def subst-domain-def* **by** *auto*

lemma *subst-apply-term-restrict-subst-domain*:
vars-term t ⊆ *V* ⟹ *t* · *restrict-subst-domain V σ* = *t* · *σ*
by (*rule term-subst-eq*) (*simp add*: *restrict-subst-domain-def subsetD*)

3.2 Rename the Domain of a Substitution

definition *rename-subst-domain* **where**
rename-subst-domain ϱ σ x =
(*if Var x* ∈ *ϱ* ‘ *subst-domain σ then*
σ (*the-inv ϱ* (*Var x*))
else
Var x)

lemma *rename-subst-domain-Var-lhs*[*simp*]:
rename-subst-domain Var σ = *σ*
by (*rule ext*) (*simp add*: *rename-subst-domain-def inj-image-mem-iff the-inv-f-f subst-domain-def*)

lemma *rename-subst-domain-Var-rhs*[*simp*]:
rename-subst-domain ϱ Var = Var
by (*rule ext*) (*simp add*: *rename-subst-domain-def*)

lemma *subst-domain-rename-subst-domain-subset*:
assumes *is-var-ϱ*: ∀ *x*. *is-Var* (*ϱ x*)
shows *subst-domain* (*rename-subst-domain ϱ σ*) ⊆ *the-Var* ‘ *ϱ* ‘ *subst-domain σ*
by (*auto simp add*: *subst-domain-def rename-subst-domain-def*
member-image-the-Var-image-subst[*OF is-var-ϱ*])

lemma *subst-range-rename-subst-domain-subset*:
assumes *inj ϱ*
shows *subst-range* (*rename-subst-domain ϱ σ*) ⊆ *subst-range σ*
proof (*intro Set.equalityI Set.subsetI*)
fix *t* **assume** *t* ∈ *subst-range* (*rename-subst-domain ϱ σ*)
then obtain *x* **where**
t-def: *t* = *rename-subst-domain ϱ σ x* **and**
rename-subst-domain ϱ σ x ≠ *Var x*
by (*auto simp*: *image-iff subst-domain-def*)

show *t* ∈ *subst-range σ*

proof (*cases* ‹*Var* $x \in \varrho$ ‘ *subst-domain* σ›)
case *True*
then obtain x' **where** $\varrho\ x' =$ *Var* x **and** $x' \in$ *subst-domain* σ
by *auto*
then show *?thesis*
using *the-inv-f-f*[*OF* ‹*inj* ϱ›, *of* x']
by (*simp add*: *t-def rename-subst-domain-def*)
next
case *False*
hence *False*
using ‹*rename-subst-domain* $\varrho\ \sigma\ x \neq$ *Var* x›
by (*simp add*: *t-def rename-subst-domain-def*)
thus *?thesis* **..**
qed
qed

lemma *range-vars-rename-subst-domain-subset*:
assumes *inj* ϱ
shows *range-vars* (*rename-subst-domain* $\varrho\ \sigma$) $\subseteq$ *range-vars* σ
unfolding *range-vars-def*
using *subst-range-rename-subst-domain-subset*[*OF* ‹*inj* ϱ›]
by (*metis Union-mono image-mono*)

lemma *renaming-cancels-rename-subst-domain*:
assumes *is-var-ϱ*: $\forall x.$ *is-Var* ($\varrho\ x$) **and** *inj* ϱ **and** *vars-t*: *vars-term* $t \subseteq$ *subst-domain* σ
shows $t \cdot \varrho \cdot$ *rename-subst-domain* $\varrho\ \sigma = t \cdot \sigma$
unfolding *subst-subst*
proof (*intro term-subst-eq ballI*)
fix x **assume** $x \in$ *vars-term* t
with *vars-t* **have** *x-in*: $x \in$ *subst-domain* σ
by *blast*

obtain x' **where** *ϱ-x*: $\varrho\ x =$ *Var* x'
using *is-var-ϱ* **by** (*meson is-Var-def*)
with *x-in* **have** *x'-in*: *Var* $x' \in \varrho$ ‘ *subst-domain* σ
by (*metis image-eqI*)

have ($\varrho \circ_s$ *rename-subst-domain* $\varrho\ \sigma$) $x = \varrho\ x \cdot$ *rename-subst-domain* $\varrho\ \sigma$
by (*simp add*: *subst-compose-def*)
also have $\ldots =$ *rename-subst-domain* $\varrho\ \sigma\ x'$
using *ϱ-x* **by** *simp*
also have $\ldots = \sigma$ (*the-inv* ϱ (*Var* x'))
by (*simp add*: *rename-subst-domain-def if-P*[*OF* *x'-in*])
also have $\ldots = \sigma$ (*the-inv* ϱ ($\varrho\ x$))
by (*simp add*: *ϱ-x*)
also have $\ldots = \sigma\ x$
using ‹*inj* ϱ› **by** (*simp add*: *the-inv-f-f*)
finally show ($\varrho \circ_s$ *rename-subst-domain* $\varrho\ \sigma$) $x = \sigma\ x$

by *simp*
qed

3.3 Rename the Domain and Range of a Substitution

definition *rename-subst-domain-range* **where**
rename-subst-domain-range ϱ σ x =
(*if Var* $x \in \varrho$ ' *subst-domain* σ *then*
((*Var o the-inv* ϱ) $\circ_s$ σ $\circ_s$ ϱ) (*Var* x)
else
Var x)

lemma *rename-subst-domain-range-Var-lhs*[*simp*]:
rename-subst-domain-range Var $\sigma = \sigma$
by (*rule ext*) (*simp add*: *rename-subst-domain-range-def inj-image-mem-iff the-inv-f-f subst-domain-def subst-compose-def*)

lemma *rename-subst-domain-range-Var-rhs*[*simp*]:
rename-subst-domain-range ϱ *Var* = *Var*
by (*rule ext*) (*simp add*: *rename-subst-domain-range-def*)

lemma *subst-compose-renaming-rename-subst-domain-range*:
fixes σ ϱ :: ('*f*, '*v*) *subst*
assumes *is-var-*ϱ: $\forall x$. *is-Var* (ϱ x) **and** *inj* ϱ
shows ϱ $\circ_s$ *rename-subst-domain-range* ϱ σ = σ $\circ_s$ ϱ
proof (*rule ext*)
fix x
from *is-var-*ϱ **obtain** x' **where** ϱ x = *Var* x'
by (*meson is-Var-def is-renaming-def*)
with ‹*inj* ϱ› **have** *inv-*ϱ*-*x': *the-inv* ϱ (*Var* x') = x
by (*metis the-inv-f-f*)

show (ϱ $\circ_s$ *rename-subst-domain-range* ϱ σ) x = (σ $\circ_s$ ϱ) x
proof (*cases* $x \in$ *subst-domain* σ)
case *True*
hence *Var* $x' \in \varrho$ ' *subst-domain* σ
using ‹ϱ x = *Var* x'› **by** (*metis imageI*)
thus *?thesis*
by (*simp add*: ‹ϱ x = *Var* x'› *rename-subst-domain-range-def subst-compose-def inv-*ϱ*-*x')
next
case *False*
hence *Var* $x' \notin \varrho$ ' *subst-domain* σ
proof (*rule contrapos-nn*)
assume *Var* $x' \in \varrho$ ' *subst-domain* σ
hence ϱ $x \in \varrho$ ' *subst-domain* σ
unfolding ‹ϱ x = *Var* x'› **.**
thus $x \in$ *subst-domain* σ
unfolding *inj-image-mem-iff*[*OF* ‹*inj* ϱ›] **.**

qed
with *False* ‹ϱ *x* = *Var x'*› **show** *?thesis*
by (*simp add*: *subst-compose-def subst-domain-def rename-subst-domain-range-def*)
qed
qed

corollary *subst-apply-term-renaming-rename-subst-domain-range*:
— This might be easier to find with **find-theorems**.
fixes *t* :: ('*f*, '*v*) *term* **and** σ ϱ :: ('*f*, '*v*) *subst*
assumes *is-var-*ϱ: $\forall$ *x*. *is-Var* (ϱ *x*) **and** *inj* ϱ
shows *t* · ϱ · *rename-subst-domain-range* ϱ σ = *t* · σ · ϱ
unfolding *subst-subst*
unfolding *subst-compose-renaming-rename-subst-domain-range*[*OF assms*]
by (*rule refl*)

A term is called *ground* if it does not contain any variables.

fun *ground* :: ('*f*, '*v*) *term* $\Rightarrow$ *bool*
where
ground (*Var x*) $\longleftrightarrow$ *False* |
ground (*Fun f ts*) $\longleftrightarrow$ ($\forall$ *t* $\in$ *set ts*. *ground t*)

lemma *ground-vars-term-empty*:
ground t $\longleftrightarrow$ *vars-term t* = {}
by (*induct t*) *simp-all*

lemma *ground-subst* [*simp*]:
ground (*t* · σ) $\longleftrightarrow$ ($\forall$ *x* $\in$ *vars-term t*. *ground* (σ *x*))
by (*induct t*) *simp-all*

lemma *ground-subst-apply*:
assumes *ground t*
shows *t* · σ = *t*
proof −
have *t* = *t* · *Var* **by** *simp*
also have . . . = *t* · σ
by (*rule term-subst-eq*, *insert assms*[*unfolded ground-vars-term-empty*], *auto*)
finally show *?thesis* **by** *simp*
qed

Just changing the variables in a term

abbreviation *map-vars-term f* $\equiv$ *term.map-term* (λ*x*. *x*) *f*

lemma *map-vars-term-as-subst*:
map-vars-term f t = *t* · (λ *x*. *Var* (*f x*))
by (*induct t*) *simp-all*

lemma *map-vars-term-eq*:
map-vars-term f s = *s* · (*Var* $\circ$ *f*)
by (*induct s*) *auto*

lemma *ground-map-vars-term* [*simp*]:
  ground (*map-vars-term f t*) = *ground t*
  by (*induct t*) *simp-all*

lemma *map-vars-term-subst* [*simp*]:
  map-vars-term f $(t \cdot \sigma) = t \cdot (\lambda\ x.$ *map-vars-term f* $(\sigma\ x))$
  by (*induct t*) *simp-all*

lemma *map-vars-term-compose*:
  map-vars-term m1 (*map-vars-term m2 t*) = *map-vars-term* (*m1 o m2*) *t*
  by (*induct t*) *simp-all*

lemma *map-vars-term-id* [*simp*]:
  map-vars-term id t = *t*
  by (*induct t*) (*auto intro*: *map-idI*)

lemma *apply-subst-map-vars-term*:
  map-vars-term m t $\cdot\ \sigma = t \cdot (\sigma \circ m)$
  by (*induct t*) (*auto*)

end

3.4 Multisets of Pairs of Terms

theory *Term-Pair-Multiset*
  imports
    Term
    HOL−Library.Multiset
begin

Multisets of pairs of terms are used in abstract inference systems for matching and unification.

3.5 Size

Make sure that every pair has size at least *1*.

definition *pair-size p* = *size* (*fst p*) + *size* (*snd p*) + *1*

Compute the number of symbols in a multiset of term pairs.

definition *size-mset M* = *fold-mset* ((+) ∘ *pair-size*) *0 M*

interpretation *size-mset-fun*:
  comp-fun-commute (+) ∘ *pair-size*
  by *standard auto*

lemma *fold-pair-size-plus*:

fold-mset $((+) \circ$ *pair-size*) *0 M* + *n* = *fold-mset* $((+) \circ$ *pair-size*) *n M*
by (*induct M arbitrary*: *n*) (*simp add*: *size-mset-def*)+

lemma *size-mset-union* [*simp*]:
size-mset (*M* + *N*) = *size-mset N* + *size-mset M*
by (*auto simp*: *size-mset-def fold-pair-size-plus*)

lemma *size-mset-add-mset* [*simp*]:
size-mset (*add-mset x M*) = *pair-size x* + (*size-mset M*)
by (*auto simp*: *size-mset-def*)

lemma *nonempty-size-mset* [*simp*]:
assumes $M \neq \{\#\}$
shows *size-mset M* > *0*
using *assms* **by** (*induct M*) (*simp add*: *size-mset-def pair-size-def*)+

lemma *size-mset-singleton* [*simp*]:
size-mset {#(*l*, *r*)#} = *size l* + *size r* + *1*
by (*auto simp*: *size-mset-def pair-size-def*)

lemma *size-mset-empty* [*simp*]:
size-mset {#} = *0*
by (*auto simp*: *size-mset-def*)

lemma *size-mset-set-zip-leq*:
size-mset (*mset* (*zip ss ts*)) $\leq$ *size-list size ss* + *size-list size ts*
proof (*induct ss arbitrary*: *ts*)
case (*Cons s ss*)
then show *?case*
by (*cases ts*) (*auto intro*: *le-SucI simp*: *pair-size-def*)
qed *simp*

lemma *size-mset-Fun-less*:
size-mset {#(*Fun f ss*, *Fun g ts*)#} > *size-mset* (*mset* (*zip ss ts*))
by (*auto simp*: *pair-size-def intro*: *order-le-less-trans size-mset-set-zip-leq*)

lemma *decomp-size-mset-less*:
assumes *length ss* = *length ts*
shows *size-mset* (*M* + *mset* (*zip ss ts*)) < *size-mset* (*M* + {#(*Fun f ss*, *Fun f ts*)#})
using *assms* **and** *size-mset-Fun-less* [*of ss ts f f*] **by** *simp*

3.5.1 Substitutions

Applying a substitution to a multiset of term pairs.

definition *subst-mset* σ *M* = *image-mset* ($\lambda p.$ (*fst p* $\cdot$ σ, *snd p* $\cdot$ σ)) *M*
lemma *subst-mset-empty* [*simp*]:
subst-mset σ {#} = {#}
by (*auto simp*: *subst-mset-def*)

lemma *subst-mset-union*:
subst-mset σ $(M + N)$ = *subst-mset* σ M + *subst-mset* σ N
by (*auto simp*: *subst-mset-def*)

lemma *subst-mset-Var* [*simp*]:
subst-mset Var M = *M*
by (*auto simp*: *subst-mset-def*)

lemma *subst-mset-subst-compose* [*simp*]:
subst-mset $(\sigma \circ_s \tau)$ M = *subst-mset* τ (*subst-mset* σ M)
by (*simp add*: *subst-mset-def image-mset.compositionality o-def*)

3.5.2 Variables

Compute the set of variables occurring in a multiset of term pairs.

definition *vars-mset M* = $\bigcup$ (*set-mset* (*image-mset* (λr. *vars-term* (*fst r*) $\cup$ *vars-term* (*snd r*)) *M*))

lemma *vars-mset-singleton* [*simp*]:
vars-mset {#*p*#} = *vars-term* (*fst p*) $\cup$ *vars-term* (*snd p*)
by (*auto simp*: *vars-mset-def*)

lemma *vars-mset-union* [*simp*]:
vars-mset $(A + B)$ = *vars-mset A* $\cup$ *vars-mset B*
by (*auto simp*: *vars-mset-def*)

lemma *vars-mset-add-mset* [*simp*]:
vars-mset (*add-mset x M*) = *vars-term* (*fst x*) $\cup$ *vars-term* (*snd x*) $\cup$ *vars-mset M*
by (*auto simp*: *vars-mset-def*)

lemma *vars-mset-set-zip* [*simp*]:
assumes *length xs* = *length ys*
shows *vars-mset* (*mset* (*zip xs ys*)) = ($\bigcup x \in$ *set xs* $\cup$ *set ys*. *vars-term x*)
using *assms* **by** (*induct xs ys rule*: *list-induct2*) (*auto simp*: *vars-mset-def*)

lemma *not-in-vars-mset-subst-mset* [*simp*]:
assumes $x \notin$ *vars-term t*
shows $x \notin$ *vars-mset* (*subst-mset* (*subst x t*) *M*)
using *assms* **by** (*auto simp*: *vars-mset-def subst-mset-def vars-term-subst subst-def*)

lemma *vars-mset-subst-mset-subset*:
vars-mset (*subst-mset* (*subst x t*) *M*) $\subseteq$ *vars-mset M* $\cup$ *vars-term t* $\cup$ $\{x\}$ (**is** *?L* $\subseteq$ *?R*)
proof
fix *y*
assume $y \in$ *?L*
then obtain *u v* **where** $(u, v) \in\#$ *M*
and $y \in$ *vars-term* ($u \cdot$ *subst x t*) $\cup$ *vars-term* ($v \cdot$ *subst x t*)

by (*auto simp*: *vars-mset-def subst-mset-def*)
moreover then have $y \in$ *vars-term* $u \cup$ *vars-term* $v \cup$ *vars-term* t
unfolding *vars-term-subst subst-def fun-upd-def*
by (*auto*) (*metis empty-iff*)+
ultimately show $y \in$ *?R* **by** (*force simp*: *vars-mset-def*)
qed

lemma *Var-left-vars-mset-less*:
assumes $x \notin$ *vars-term* t
shows *vars-mset* (*subst-mset* (*subst* x t) M) $\subset$ *vars-mset* (*add-mset* (*Var* x, t) M) (**is** *?L* $\subset$ *?R*)
proof
show *?L* $\subseteq$ *?R* **using** *vars-mset-subst-mset-subset* [*of* x t M] **by** (*simp add*: *ac-simps*)
next
have $x \in$ *?R* **using** *assms* **by** (*force simp*: *vars-mset-def*)
moreover have $x \notin$ *?L* **using** *assms* **by** *simp*
ultimately show *?L* $\neq$ *?R* **by** *blast*
qed

lemma *Var-right-vars-mset-less*:
assumes $x \notin$ *vars-term* t
shows *vars-mset* (*subst-mset* (*subst* x t) M) $\subset$ *vars-mset* (*add-mset* (t, *Var* x) M)
using *Var-left-vars-mset-less* [*OF assms*] **by** *simp*

lemma *mem-vars-mset-subst-mset*:
assumes $y \in$ *vars-mset* (*subst-mset* (*subst* x t) M)
and $y \neq x$
and $y \notin$ *vars-term* t
shows $y \in$ *vars-mset* M
using *vars-mset-subst-mset-subset* [*of* x t M] **and** *assms* **by** *blast*

lemma *finite-vars-mset*:
finite (*vars-mset* A)
by (*auto simp*: *vars-mset-def*)

end

4 Abstract Matching

theory *Abstract-Matching*
imports
Term-Pair-Multiset
Abstract−Rewriting.Abstract-Rewriting
begin

lemma *singleton-eq-union-iff* [*iff*]:

$\{\#x\#\} = M + \{\#y\#\} \longleftrightarrow M = \{\#\} \wedge x = y$
by (*metis multi-self-add-other-not-self single-eq-single single-is-union*)

Turning functional maps into substitutions.

definition *subst-of-map d σ x* =
  (*case σ x of*
    None ⇒ *d x*
  | *Some t* ⇒ *t*)

lemma *size-mset-mset-less* [*simp*]:
  assumes *length ss* = *length ts*
  shows *size-mset* (*mset* (*zip ss ts*)) < *3* + (*size-list size ss* + *size-list size ts*)
  using *assms* **by** (*induct ss ts rule*: *list-induct2*) (*auto simp*: *pair-size-def*)

definition *matchers* :: (('*f*, '*v*) *term* × ('*f*, '*w*) *term*) *set* ⇒ ('*f*, '*v*, '*w*) *gsubst set*
  where
    matchers P = {*σ*. ∀ *e*∈*P*. *fst e* · *σ* = *snd e*}

lemma *matchers-vars-term-eq*:
  assumes *σ* ∈ *matchers P* **and** *τ* ∈ *matchers P*
    and (*s*, *t*) ∈ *P*
  shows ∀ *x*∈*vars-term s*. *σ x* = *τ x*
  using *assms* **unfolding** *term-subst-eq-conv* [*symmetric*] **by** (*force simp*: *matchers-def*)

lemma *matchers-empty* [*simp*]:
  matchers {} = *UNIV*
  by (*simp add*: *matchers-def*)

lemma *matchers-insert* [*simp*]:
  matchers (*insert e P*) = {*σ*. *fst e* · *σ* = *snd e*} ∩ *matchers P*
  by (*auto simp*: *matchers-def*)

lemma *matchers-Un* [*simp*]:
  matchers (*P* ∪ *P'*) = *matchers P* ∩ *matchers P'*
  by (*auto simp*: *matchers-def*)

lemma *matchers-set-zip* [*simp*]:
  assumes *length ss* = *length ts*
  shows *matchers* (*set* (*zip ss ts*)) = {*σ*. *map* (*λt*. *t* · *σ*) *ss* = *ts*}
  using *assms* **by** (*induct ss ts rule*: *list-induct2*) *auto*

definition *matchers-map m* = *matchers* ((*λx*. (*Var x*, *the* (*m x*))) ‘ *Map.dom m*)

lemma *matchers-map-empty* [*simp*]:
  matchers-map Map.empty = *UNIV*
  by (*simp add*: *matchers-map-def*)

lemma *matchers-map-upd* [*simp*]:

assumes $\sigma\ x = None \lor \sigma\ x = Some\ t$
shows *matchers-map* $(\lambda y.\ if\ y = x\ then\ Some\ t\ else\ \sigma\ y) =$
matchers-map $\sigma \cap \{\tau.\ \tau\ x = t\}$ (**is** $?L = ?R$)
proof
show $?L \supseteq ?R$ **by** (*auto simp*: *matchers-map-def matchers-def*)
next
show $?L \subseteq ?R$
by (*rule subsetI*)
(*insert assms*, *auto simp*: *matchers-map-def matchers-def dom-def*)
qed

lemma *matchers-map-upd'* [*simp*]:
assumes $\sigma\ x = None \lor \sigma\ x = Some\ t$
shows *matchers-map* $(\sigma\ (x \mapsto t))$ = *matchers-map* $\sigma \cap \{\tau.\ \tau\ x = t\}$
using *matchers-map-upd* [*of* σ *x t*, *OF assms*]
by (*simp add*: *matchers-map-def matchers-def dom-def*)

inductive *MATCH1* **where**
Var [*intro*!, *simp*]: $\sigma\ x = None \lor \sigma\ x = Some\ t \Longrightarrow$
MATCH1 $(P + \{\#(Var\ x,\ t)\#\},\ \sigma)\ (P,\ \sigma\ (x \mapsto t))$ |
Fun [*intro*]: *length ss* = *length ts* $\Longrightarrow$
MATCH1 $(P + \{\#(Fun\ f\ ss,\ Fun\ f\ ts)\#\},\ \sigma)\ (P + mset\ (zip\ ss\ ts),\ \sigma)$

lemma *MATCH1-matchers* [*simp*]:
assumes *MATCH1 x y*
shows *matchers-map* (*snd x*) $\cap$ *matchers* (*set-mset* (*fst x*)) =
matchers-map (*snd y*) $\cap$ *matchers* (*set-mset* (*fst y*))
using *assms* **by** (*induct*) (*simp-all add*: *ac-simps*)

definition *matchrel* = $\{(x,\ y).\ MATCH1\ x\ y\}$

lemma *MATCH1-matchrel-conv*:
MATCH1 x y $\longleftrightarrow (x,\ y) \in$ *matchrel*
by (*simp add*: *matchrel-def*)

lemma *matchrel-rtrancl-matchers* [*simp*]:
assumes $(x,\ y) \in matchrel^*$
shows *matchers-map* (*snd x*) $\cap$ *matchers* (*set-mset* (*fst x*)) =
matchers-map (*snd y*) $\cap$ *matchers* (*set-mset* (*fst y*))
using *assms* **by** (*induct*) (*simp-all add*: *matchrel-def*)

lemma *subst-of-map-in-matchers-map* [*simp*]:
subst-of-map d m $\in$ *matchers-map m*
by (*auto simp*: *subst-of-map-def* [*abs-def*] *matchers-map-def matchers-def*)

lemma *matchrel-sound*:
assumes $((P,\ Map.empty),\ (\{\#\},\ \sigma)) \in matchrel^*$
shows *subst-of-map d* $\sigma \in$ *matchers* (*set-mset P*)
using *matchrel-rtrancl-matchers* [*OF assms*] **by** *simp*

lemma *MATCH1-size-mset*:
  assumes *MATCH1 x y*
  shows *size-mset* (*fst x*) > *size-mset* (*fst y*)
  using *assms* **by** (*cases*) (*auto simp*: *pair-size-def*)+

definition *matchless* = *inv-image* (*measure size-mset*) *fst*

lemma *wf-matchless*:
  wf matchless
  by (*auto simp*: *matchless-def*)

lemma *MATCH1-matchless*:
  assumes *MATCH1 x y*
  shows (*y*, *x*) ∈ *matchless*
  using *MATCH1-size-mset* [*OF assms*]
  by (*simp add*: *matchless-def*)

lemma *converse-matchrel-subset-matchless*:
  $matchrel^{-1}$ ⊆ *matchless*
  using *MATCH1-matchless* **by** (*auto simp*: *matchrel-def*)

lemma *wf-converse-matchrel*:
  wf ($matchrel^{-1}$)
  by (*rule wf-subset* [*OF wf-matchless converse-matchrel-subset-matchless*])

lemma *MATCH1-singleton-Var* [*intro*]:
  $\sigma\ x = None \Longrightarrow MATCH1\ (\{\#(Var\ x,\ t)\#\},\ \sigma)\ (\{\#\},\ \sigma\ (x \mapsto t))$
  $\sigma\ x = Some\ t \Longrightarrow MATCH1\ (\{\#(Var\ x,\ t)\#\},\ \sigma)\ (\{\#\},\ \sigma\ (x \mapsto t))$
  using *MATCH1.Var* [*of* σ *x t* {#}] **by** *simp-all*

lemma *MATCH1-singleton-Fun* [*intro*]:
  $length\ ss = length\ ts \Longrightarrow MATCH1\ (\{\#(Fun\ f\ ss,\ Fun\ f\ ts)\#\},\ \sigma)\ (mset\ (zip\ ss\ ts),\ \sigma)$
  using *MATCH1.Fun* [*of ss ts* {#} *f* σ] **by** *simp*

lemma *not-MATCH1-singleton-Var* [*dest*]:
  $\neg\ MATCH1\ (\{\#(Var\ x,\ t)\#\},\ \sigma)\ (\{\#\},\ \sigma\ (x \mapsto t)) \Longrightarrow \sigma\ x \neq None \land \sigma\ x \neq Some\ t$
  by *auto*

lemma *not-matchrelD*:
  assumes $\neg\ (\exists\, y.\ ((\{\#e\#\},\ \sigma),\ y) \in matchrel)$
  shows $(\exists\, f\ ss\ x.\ e = (Fun\ f\ ss,\ Var\ x)) \lor$
    $(\exists\, x\ t.\ e = (Var\ x,\ t) \land \sigma\ x \neq None \land \sigma\ x \neq Some\ t) \lor$
    $(\exists\, f\ g\ ss\ ts.\ e = (Fun\ f\ ss,\ Fun\ g\ ts) \land (f \neq g \lor length\ ss \neq length\ ts))$
proof (*rule ccontr*)
  assume ∗: ¬ *?thesis*
  show *False*

proof (*cases e*)
case (*Pair s t*)
with *assms* **and** $*$ **show** *?thesis*
by (*cases s*) (*cases t*, *auto simp*: *matchrel-def*)+
qed
qed

lemma *ne-matchers-imp-matchrel*:
assumes *matchers-map* $\sigma \cap$ *matchers* $\{e\} \neq \{\}$
shows $\exists\, y.\ ((\{\#e\#\}, \sigma), y) \in$ *matchrel*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
from *not-matchrelD* [*OF this*] **and** *assms*
show *False* **by** (*auto simp*: *matchers-map-def matchers-def dom-def*)
qed

lemma *MATCH1-mono*:
assumes *MATCH1* (P, σ) (P', σ')
shows *MATCH1* $(P + M, \sigma)$ $(P' + M, \sigma')$
using *assms* **apply** (*cases*) **apply** (*auto simp*: *ac-simps*)
using *Var* **apply** *force*
using *Var* **apply** *force*
using *Fun*
by (*metis* (*no-types*, *lifting*) *add.assoc add-mset-add-single*)

lemma *matchrel-mono*:
assumes $(x, y) \in$ *matchrel*
shows ((*fst* $x + M$, *snd* x), (*fst* $y + M$, *snd* y)) $\in$ *matchrel*
using *assms* **and** *MATCH1-mono* [*of fst x*]
by (*simp add*: *MATCH1-matchrel-conv*)

lemma *matchrel-rtrancl-mono*:
assumes $(x, y) \in$ *matchrel**
shows ((*fst* $x + M$, *snd* x), (*fst* $y + M$, *snd* y)) $\in$ *matchrel**
using *assms* **by** (*induct*) (*auto dest*: *matchrel-mono* [*of - - M*])

lemma *ne-matchers-imp-empty-or-matchrel*:
assumes *matchers-map* $\sigma \cap$ *matchers* (*set-mset* P) $\neq \{\}$
shows $P = \{\#\} \vee (\exists\, y.\ ((P, \sigma), y) \in$ *matchrel*)
proof (*cases P*)
case (*add e P'*)
then have [*simp*]: $P = P' + \{\#e\#\}$ **by** *simp*
from *assms* **have** *matchers-map* $\sigma \cap$ *matchers* $\{e\} \neq \{\}$ **by** *auto*
from *ne-matchers-imp-matchrel* [*OF this*]
obtain P'' σ' **where** *MATCH1* $(\{\#e\#\}, \sigma)$ (P'', σ')
by (*auto simp*: *matchrel-def*)
from *MATCH1-mono* [*OF this*, *of P'*] **have** *MATCH1* (P, σ) $(P' + P'', \sigma')$ **by** (*simp add*: *ac-simps*)
then show *?thesis* **by** (*auto simp*: *matchrel-def*)

qed *simp*

lemma *matchrel-imp-converse-matchless* [*dest*]:
  $(x, y) \in$ *matchrel* $\Longrightarrow (y, x) \in$ *matchless*
  using *MATCH1-matchless* **by** (*cases* x, *cases* y) (*auto simp*: *matchrel-def*)

lemma *ne-matchers-imp-empty*:
  fixes P :: $(('f, 'v)$ *term* $\times$ $('f, 'w)$ *term*$)$ *multiset*
  assumes *matchers-map* $\sigma \cap$ *matchers* (*set-mset* P) $\neq \{\}$
  shows $\exists \sigma'. ((P, \sigma), (\{\#\}, \sigma')) \in$ *matchrel**
using *assms*
proof (*induct* P *arbitrary*: σ *rule*: *wf-induct* [*OF wf-measure* [*of size-mset*]])
  fix P :: $(('f, 'v)$ *term* $\times$ $('f, 'w)$ *term*$)$ *multiset*
    and σ
  presume *IH*: $\bigwedge P' \sigma. [\![(P', P) \in$ *measure size-mset*; *matchers-map* $\sigma \cap$ *matchers*
(*set-mset* P') $\neq \{\}]\!] \Longrightarrow$
    $\exists \sigma'. ((P', \sigma), \{\#\}, \sigma') \in$ *matchrel**
    and $*$: *matchers-map* $\sigma \cap$ *matchers* (*set-mset* P) $\neq \{\}$
  show $\exists \sigma'. ((P, \sigma), \{\#\}, \sigma') \in$ *matchrel**
  proof (*cases* $P = \{\#\}$)
    assume $P \neq \{\#\}$
    with *ne-matchers-imp-empty-or-matchrel* [*OF* $*$]
      obtain P' σ' **where** $**$: $((P, \sigma), (P', \sigma')) \in$ *matchrel* **by** (*auto*)
    with $*$ **have** $(P', P) \in$ *measure size-mset*
      and *matchers-map* $\sigma' \cap$ *matchers* (*set-mset* P') $\neq \{\}$
      using *MATCH1-matchers* [*of* (P, σ) (P', σ')]
      by (*auto simp*: *matchrel-def dest*: *MATCH1-size-mset*)
    from *IH* [*OF this*] **and** $**$
      show *?thesis* **by** (*auto intro*: *converse-rtrancl-into-rtrancl*)
  qed *force*
qed *simp*

lemma *empty-not-reachable-imp-matchers-empty*:
  assumes $\bigwedge\sigma'. ((P, \sigma), (\{\#\}, \sigma')) \notin$ *matchrel**
  shows *matchers-map* $\sigma \cap$ *matchers* (*set-mset* P) $= \{\}$
  using *ne-matchers-imp-empty* [*of* σ P] **and** *assms* **by** *blast*

lemma *irreducible-reachable-imp-matchers-empty*:
  assumes $((P, \sigma), y) \in$ *matchrel*$^!$ **and** *fst* $y \neq \{\#\}$
  shows *matchers-map* $\sigma \cap$ *matchers* (*set-mset* P) $= \{\}$
proof −
  have $((P, \sigma), y) \in$ *matchrel**
    and $\bigwedge\tau. (y, (\{\#\}, \tau)) \notin$ *matchrel**
    using *assms* **by** *auto* (*metis NF-not-suc fst-conv normalizability-E*)
  moreover with *empty-not-reachable-imp-matchers-empty*
    have *matchers-map* (*snd* y) $\cap$ *matchers* (*set-mset* (*fst* y)) $= \{\}$ **by** (*cases* y)
auto
  ultimately show *?thesis* **using** *matchrel-rtrancl-matchers* [*of* (P, σ)] **by** *simp*
qed

lemma *matchers-map-not-empty* [*simp*]:
  matchers-map $\sigma \neq \{\}$
  $\{\} \neq$ *matchers-map* σ
  by (*auto simp*: *matchers-map-def matchers-def*)

lemma *matchers-empty-imp-not-empty-NF*:
  assumes *matchers* (*set-mset* P) = {}
  shows $\exists\, y.$ *fst* $y \neq \{\#\} \wedge ((P,$ *Map.empty*$),\ y) \in$ *matchrel*$^!$
proof (*rule ccontr*)
  assume ¬ *?thesis*
  then have $*$: $\bigwedge y.$ ((*P*, *Map.empty*), *y*) ∈ *matchrel*$^!$ $\Longrightarrow$ *fst* $y = \{\#\}$ **by** *auto*
  have *SN matchrel* **using** *wf-converse-matchrel* **by** (*auto simp*: *SN-iff-wf*)
  then obtain *y* **where** ((*P*, *Map.empty*), *y*) ∈ *matchrel*$^!$
    by (*metis SN-imp-WN UNIV-I WN-onE*)
  with $*$ [*OF this*] **obtain** τ **where** ((*P*, *Map.empty*), ({#}, τ)) ∈ *matchrel** **by** (*cases y*) *auto*
  from *matchrel-rtrancl-matchers* [*OF this*] **and** *assms*
    show *False* **by** *simp*
qed

end

5 Unification

5.1 Unifiers

Definition and properties of (most general) unifiers

theory *Unifiers*
  imports *Term*
begin

lemma *map-eq-set-zipD* [*dest*]:
  assumes *map f xs* = *map f ys*
    and (*x*, *y*) ∈ *set* (*zip xs ys*)
  shows *f x* = *f y*
using *assms*
proof (*induct xs arbitrary*: *ys*)
  case (*Cons x xs*)
  then show *?case* **by** (*cases ys*) *auto*
qed *simp*

type-synonym ('*f*, '*v*) *equation* = ('*f*, '*v*) *term* × ('*f*, '*v*) *term*
type-synonym ('*f*, '*v*) *equations* = ('*f*, '*v*) *equation set*

The set of unifiers for a given set of equations.

definition *unifiers* :: ('*f*, '*v*) *equations* ⇒ ('*f*, '*v*) *subst set*

where
$\quad$ *unifiers* $E = \{\sigma.\ \forall\, p{\in}E.\ \mathit{fst}\ p \cdot \sigma = \mathit{snd}\ p \cdot \sigma\}$

Check whether a set of equations is unifiable.

definition *unifiable* $E \longleftrightarrow (\exists\, \sigma.\ \sigma \in \mathit{unifiers}\ E)$

lemma *in-unifiersE* [*elim*]:
$\quad [\![\sigma \in \mathit{unifiers}\ E;\ (\bigwedge e.\ e \in E \Longrightarrow \mathit{fst}\ e \cdot \sigma = \mathit{snd}\ e \cdot \sigma) \Longrightarrow P]\!] \Longrightarrow P$
by (*force simp*: *unifiers-def*)

Applying a substitution to a set of equations.

definition *subst-set* :: $('f,\ 'v)\ \mathit{subst} \Rightarrow ('f,\ 'v)\ \mathit{equations} \Rightarrow ('f,\ 'v)\ \mathit{equations}$
where
$\quad \mathit{subst\text{-}set}\ \sigma\ E = (\lambda e.\ (\mathit{fst}\ e \cdot \sigma,\ \mathit{snd}\ e \cdot \sigma))\ `\ E$

Check whether a substitution is a most-general unifier (mgu) of a set of equations.

definition *is-mgu* :: $('f,\ 'v)\ \mathit{subst} \Rightarrow ('f,\ 'v)\ \mathit{equations} \Rightarrow \mathit{bool}$
where
$\quad \mathit{is\text{-}mgu}\ \sigma\ E \longleftrightarrow \sigma \in \mathit{unifiers}\ E \wedge (\forall\, \tau \in \mathit{unifiers}\ E.\ (\exists\, \gamma.\ \tau = \sigma \circ_s \gamma))$

The following property characterizes idempotent mgus, that is, mgus σ for which $\sigma \circ_s \sigma = \sigma$ holds.

definition *is-imgu* :: $('f,\ 'v)\ \mathit{subst} \Rightarrow ('f,\ 'v)\ \mathit{equations} \Rightarrow \mathit{bool}$
where
$\quad \mathit{is\text{-}imgu}\ \sigma\ E \longleftrightarrow \sigma \in \mathit{unifiers}\ E \wedge (\forall\, \tau \in \mathit{unifiers}\ E.\ \tau = \sigma \circ_s \tau)$

5.1.1 Properties of sets of unifiers

lemma *unifiers-Un* [*simp*]:
$\quad \mathit{unifiers}\ (s \cup t) = \mathit{unifiers}\ s \cap \mathit{unifiers}\ t$
by (*auto simp*: *unifiers-def*)

lemma *unifiers-empty* [*simp*]:
$\quad \mathit{unifiers}\ \{\} = \mathit{UNIV}$
by (*auto simp*: *unifiers-def*)

lemma *unifiers-insert*:
$\quad \mathit{unifiers}\ (\mathit{insert}\ p\ t) = \{\sigma.\ \mathit{fst}\ p \cdot \sigma = \mathit{snd}\ p \cdot \sigma\} \cap \mathit{unifiers}\ t$
by (*auto simp*: *unifiers-def*)

lemma *unifiers-insert-ident* [*simp*]:
$\quad \mathit{unifiers}\ (\mathit{insert}\ (t,\ t)\ E) = \mathit{unifiers}\ E$
by (*auto simp*: *unifiers-insert*)

lemma *unifiers-insert-swap*:
$\quad \mathit{unifiers}\ (\mathit{insert}\ (s,\ t)\ E) = \mathit{unifiers}\ (\mathit{insert}\ (t,\ s)\ E)$
by (*auto simp*: *unifiers-insert*)

lemma *unifiers-insert-Var-swap* [*simp*]:
unifiers (*insert* (*t*, *Var x*) *E*) = *unifiers* (*insert* (*Var x*, *t*) *E*)
by (*rule unifiers-insert-swap*)

lemma *unifiers-subst-set* [*simp*]:
$\tau \in$ *unifiers* (*subst-set* σ *E*) $\longleftrightarrow \sigma \circ_s \tau \in$ *unifiers E*
by (*auto simp*: *unifiers-def subst-set-def*)

lemma *unifiers-insert-VarD*:
shows $\sigma \in$ *unifiers* (*insert* (*Var x*, *t*) *E*) $\Longrightarrow$ *subst x t* $\circ_s \sigma = \sigma$
and $\sigma \in$ *unifiers* (*insert* (*t*, *Var x*) *E*) $\Longrightarrow$ *subst x t* $\circ_s \sigma = \sigma$
by (*auto simp*: *unifiers-def*)

lemma *unifiers-insert-Var-left*:
$\sigma \in$ *unifiers* (*insert* (*Var x*, *t*) *E*) $\Longrightarrow \sigma \in$ *unifiers* (*subst-set* (*subst x t*) *E*)
by (*auto simp*: *unifiers-def subst-set-def*)

lemma *unifiers-set-zip* [*simp*]:
assumes *length ss* = *length ts*
shows *unifiers* (*set* (*zip ss ts*)) = $\{\sigma.$ *map* ($\lambda t.\ t \cdot \sigma$) *ss* = *map* ($\lambda t.\ t \cdot \sigma$) *ts*$\}$
using *assms* **by** (*induct ss ts rule*: *list-induct2*) (*auto simp*: *unifiers-def*)

lemma *unifiers-Fun* [*simp*]:
$\sigma \in$ *unifiers* $\{$(*Fun f ss*, *Fun g ts*)$\}$ $\longleftrightarrow$
length ss = *length ts* $\wedge$ *f* = *g* $\wedge$ $\sigma \in$ *unifiers* (*set* (*zip ss ts*))
by (*auto simp*: *unifiers-def dest*: *map-eq-imp-length-eq*)
(*induct ss ts rule*: *list-induct2*, *simp-all*)

lemma *unifiers-occur-left-is-Fun*:
fixes *t* :: ('*f*, '*v*) *term*
assumes $x \in$ *vars-term t* **and** *is-Fun t*
shows *unifiers* (*insert* (*Var x*, *t*) *E*) = $\{\}$
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then obtain σ :: ('*f*, '*v*) *subst* **where** $\sigma\ x = t \cdot \sigma$ **by** (*auto simp*: *unifiers-def*)
with *is-Fun-num-funs-less* [*OF assms*, *of* σ] **show** *False* **by** *auto*
qed

lemma *unifiers-occur-left-not-Var*:
$x \in$ *vars-term t* $\Longrightarrow t \neq$ *Var x* $\Longrightarrow$ *unifiers* (*insert* (*Var x*, *t*) *E*) = $\{\}$
using *unifiers-occur-left-is-Fun* [*of x t*] **by** (*cases t*) *simp-all*

lemma *unifiers-occur-left-Fun*:
$x \in (\bigcup t{\in}$*set ts*. *vars-term t*) $\Longrightarrow$ *unifiers* (*insert* (*Var x*, *Fun f ts*) *E*) = $\{\}$
using *unifiers-occur-left-is-Fun* [*of x Fun f ts*] **by** *simp*

lemmas *unifiers-occur-left-simps* [*simp*] =
unifiers-occur-left-is-Fun

unifiers-occur-left-not-Var
unifiers-occur-left-Fun

5.1.2 Properties of unifiability

lemma *in-vars-is-Fun-not-unifiable*:
assumes $x \in$ *vars-term* t **and** *is-Fun* t
shows $\neg$ *unifiable* $\{(Var\ x,\ t)\}$
proof
assume *unifiable* $\{(Var\ x,\ t)\}$
then obtain σ **where** $\sigma \in$ *unifiers* $\{(Var\ x,\ t)\}$
by (*auto simp*: *unifiable-def*)
then have $\sigma\ x = t \cdot \sigma$ **by** (*auto*)
moreover have *num-funs* $(\sigma\ x) <$ *num-funs* $(t \cdot \sigma)$
using *is-Fun-num-funs-less* [*OF assms*] **by** *auto*
ultimately show *False* **by** *auto*
qed

lemma *unifiable-insert-swap*:
unifiable (*insert* $(s,\ t)$ E) = *unifiable* (*insert* $(t,\ s)$ E)
by (*auto simp*: *unifiable-def unifiers-insert-swap*)

lemma *subst-set-reflects-unifiable*:
fixes σ :: $('f,\ 'v)$ *subst*
assumes *unifiable* (*subst-set* σ E)
shows *unifiable* E
proof −
{ fix τ :: $('f,\ 'v)$ *subst* **assume** $\forall p \in E.$ *fst* $p \cdot \sigma \cdot \tau =$ *snd* $p \cdot \sigma \cdot \tau$
then have $\exists \sigma$ :: $('f,\ 'v)$ *subst*. $\forall p \in E.$ *fst* $p \cdot \sigma =$ *snd* $p \cdot \sigma$
by (*intro exI* [*of* - $\sigma \circ_s \tau$]) *auto* **}**
then show *?thesis* **using** *assms* **by** (*auto simp*: *unifiable-def unifiers-def subst-set-def*)
qed

5.1.3 Properties of *is-mgu*

lemma *is-mgu-empty* [*simp*]:
is-mgu Var {}
by (*auto simp*: *is-mgu-def*)

lemma *is-mgu-insert-trivial* [*simp*]:
is-mgu σ (*insert* $(t,\ t)$ E) = *is-mgu* σ E
by (*auto simp*: *is-mgu-def*)

lemma *is-mgu-insert-decomp* [*simp*]:
assumes *length ss* = *length ts*
shows *is-mgu* σ (*insert* (*Fun f ss*, *Fun f ts*) E) $\longleftrightarrow$
is-mgu σ ($E \cup$ *set* (*zip ss ts*))
using *assms* **by** (*auto simp*: *is-mgu-def unifiers-insert*)

lemma *is-mgu-insert-swap*:

is-mgu σ (*insert* (*s*, *t*) *E*) = *is-mgu* σ (*insert* (*t*, *s*) *E*)
by (*auto simp*: *is-mgu-def unifiers-def*)

lemma *is-mgu-insert-Var-swap* [*simp*]:
is-mgu σ (*insert* (*t*, *Var x*) *E*) = *is-mgu* σ (*insert* (*Var x*, *t*) *E*)
by (*rule is-mgu-insert-swap*)

lemma *is-mgu-subst-set-subst*:
assumes $x \notin$ *vars-term t*
and *is-mgu* σ (*subst-set* (*subst x t*) *E*) (**is** *is-mgu* σ *?E*)
shows *is-mgu* (*subst x t* $\circ_s$ σ) (*insert* (*Var x*, *t*) *E*) (**is** *is-mgu* *?*σ *?E'*)
proof −
from ‹*is-mgu* σ *?E*›
have *?*$\sigma \in$ *unifiers E*
and ∗: $\forall \tau$. (*subst x t* $\circ_s$ τ) $\in$ *unifiers E* $\longrightarrow$ ($\exists \mu$. $\tau = \sigma \circ_s \mu$)
by (*auto simp*: *is-mgu-def*)
then have *?*$\sigma \in$ *unifiers ?E'* **using** *assms* **by** (*simp add*: *unifiers-insert subst-compose*)
moreover have $\forall \tau$. $\tau \in$ *unifiers ?E'* $\longrightarrow$ ($\exists \mu$. $\tau =$ *?*$\sigma \circ_s \mu$)
proof (*intro allI impI*)
fix τ
assume ∗∗: $\tau \in$ *unifiers ?E'*
then have [*simp*]: *subst x t* $\circ_s$ $\tau = \tau$ **by** (*blast dest*: *unifiers-insert-VarD*)
from *unifiers-insert-Var-left* [*OF* ∗∗]
have *subst x t* $\circ_s$ $\tau \in$ *unifiers E* **by** (*simp*)
with ∗ **obtain** μ **where** $\tau = \sigma \circ_s \mu$ **by** *blast*
then have *subst x t* $\circ_s$ τ = *subst x t* $\circ_s$ $\sigma \circ_s \mu$ **by** (*auto simp*: *ac-simps*)
then show $\exists \mu$. τ = *subst x t* $\circ_s$ $\sigma \circ_s \mu$ **by** *auto*
qed
ultimately show *is-mgu ?*σ *?E'* **by** (*simp add*: *is-mgu-def*)
qed

lemma *is-imgu-imp-is-mgu*:
assumes *is-imgu* σ *E*
shows *is-mgu* σ *E*
using *assms* **by** (*auto simp*: *is-imgu-def is-mgu-def*)

5.1.4 Properties of *is-imgu*

lemma *rename-subst-domain-range-preserves-is-imgu*:
fixes *E* :: ('*f*, '*v*) *equations* **and** μ ϱ :: ('*f*, '*v*) *subst*
assumes *imgu-*μ: *is-imgu* μ *E* **and** *is-var-*ϱ: $\forall x$. *is-Var* (ϱ *x*) **and** *inj* ϱ
shows *is-imgu* (*rename-subst-domain-range* ϱ μ) (*subst-set* ϱ *E*)
proof (*unfold is-imgu-def*, *intro conjI ballI*)
from *imgu-*μ **have** *unif-*μ: $\mu \in$ *unifiers E*
by (*simp add*: *is-imgu-def*)

show *rename-subst-domain-range* ϱ $\mu \in$ *unifiers* (*subst-set* ϱ *E*)
unfolding *unifiers-subst-set unifiers-def mem-Collect-eq*
proof (*rule ballI*)

fix e_ϱ **assume** $e_\varrho \in$ *subst-set* ϱ E
then obtain e **where** $e \in E$ **and** $e_\varrho = (fst\ e \cdot \varrho,\ snd\ e \cdot \varrho)$
by (*auto simp*: *subst-set-def*)
then show *fst* $e_\varrho \cdot$ *rename-subst-domain-range* ϱ μ = *snd* $e_\varrho \cdot$ *rename-subst-domain-range* ϱ μ
using *unif-μ subst-apply-term-renaming-rename-subst-domain-range*[*OF is-var-ϱ* ‹*inj* ϱ›, *of* - μ]
by (*simp add*: *unifiers-def*)
qed
next
fix υ :: ($'f$, $'v$) *subst*
assume $\upsilon \in$ *unifiers* (*subst-set* ϱ E)
hence $(\varrho \circ_s \upsilon) \in$ *unifiers* E
by (*simp add*: *subst-set-def unifiers-def*)
with *imgu-μ* **have** *μ-ϱ-υ*: $\mu \circ_s \varrho \circ_s \upsilon = \varrho \circ_s \upsilon$
by (*simp add*: *is-imgu-def subst-compose-assoc*)

show υ = *rename-subst-domain-range* ϱ $\mu \circ_s \upsilon$
proof (*rule ext*)
fix x
show $\upsilon\ x$ = (*rename-subst-domain-range* ϱ $\mu \circ_s \upsilon$) x
proof (*cases Var* $x \in \varrho$ ‘ *subst-domain* μ)
case *True*
hence (*rename-subst-domain-range* ϱ $\mu \circ_s \upsilon$) $x = (\mu \circ_s \varrho \circ_s \upsilon)$ (*the-inv* ϱ (*Var* x))
by (*simp add*: *rename-subst-domain-range-def subst-compose-def*)
also have ... = $(\varrho \circ_s \upsilon)$ (*the-inv* ϱ (*Var* x))
by (*simp add*: *μ-ϱ-υ*)
also have ... = (ϱ (*the-inv* ϱ (*Var* x))) $\cdot$ υ
by (*simp add*: *subst-compose*)
also have ... = *Var* $x \cdot \upsilon$
using *True f-the-inv-into-f*[*OF* ‹*inj* ϱ›, *of Var* x] **by** *force*
finally show *?thesis*
by *simp*
next
case *False*
thus *?thesis*
by (*simp add*: *rename-subst-domain-range-def subst-compose*)
qed
qed
qed

corollary *rename-subst-domain-range-preserves-is-imgu-singleton*:
fixes t u :: ($'f$, $'v$) *term* **and** μ ϱ :: ($'f$, $'v$) *subst*
assumes *imgu-μ*: *is-imgu* μ $\{(t, u)\}$ **and** *is-var-ϱ*: $\forall x$. *is-Var* (ϱ x) **and** *inj* ϱ
shows *is-imgu* (*rename-subst-domain-range* ϱ μ) $\{(t \cdot \varrho,\ u \cdot \varrho)\}$
by (*rule rename-subst-domain-range-preserves-is-imgu*[*OF imgu-μ is-var-ϱ* ‹*inj* ϱ›,
unfolded subst-set-def, *simplified*])

end

5.2 Abstract Unification

We formalize an inference system for unification.

theory *Abstract-Unification*
  imports
    Unifiers
    Term-Pair-Multiset
    Abstract−Rewriting.Abstract-Rewriting
begin

lemma *foldr-assoc*:
  assumes $\bigwedge f\ g\ h.\ b\ (b\ f\ g)\ h = b\ f\ (b\ g\ h)$
  shows $foldr\ b\ xs\ (b\ y\ z) = b\ (foldr\ b\ xs\ y)\ z$
  using *assms* **by** (*induct xs*) *simp-all*

lemma *union-commutes*:
  $M + \{\#x\#\} + N = M + N + \{\#x\#\}$
  $M + mset\ xs + N = M + N + mset\ xs$
  by (*auto simp*: *ac-simps*)

5.2.1 Inference Rules

Inference rules with explicit substitutions.

inductive
  $UNIF1 :: ('f, 'v)\ subst \Rightarrow ('f, 'v)\ equation\ multiset \Rightarrow ('f, 'v)\ equation\ multiset \Rightarrow bool$
where
  trivial [*simp*]: $UNIF1\ Var\ (add\text{-}mset\ (t, t)\ E)\ E$ |
  decomp: $[\![length\ ss = length\ ts]\!] \Longrightarrow$
    $UNIF1\ Var\ (add\text{-}mset\ (Fun\ f\ ss, Fun\ f\ ts)\ E)\ (E + mset\ (zip\ ss\ ts))$ |
  Var-left: $[\![x \notin vars\text{-}term\ t]\!] \Longrightarrow$
    $UNIF1\ (subst\ x\ t)\ (add\text{-}mset\ (Var\ x,\ t)\ E)\ (subst\text{-}mset\ (subst\ x\ t)\ E)$ |
  Var-right: $[\![x \notin vars\text{-}term\ t]\!] \Longrightarrow$
    $UNIF1\ (subst\ x\ t)\ (add\text{-}mset\ (t,\ Var\ x)\ E)\ (subst\text{-}mset\ (subst\ x\ t)\ E)$

Relation version of *UNIF1* with implicit substitutions.

definition $unif = \{(x, y).\ \exists\sigma.\ UNIF1\ \sigma\ x\ y\}$

lemma *unif-UNIF1-conv*:
  $(E, E') \in unif \longleftrightarrow (\exists\sigma.\ UNIF1\ \sigma\ E\ E')$
  by (*auto simp*: *unif-def*)

lemma *UNIF1-unifD*:

UNIF1 σ *E E'* $\Longrightarrow$ (*E*, *E'*) $\in$ *unif*
by (*auto simp*: *unif-def*)

A termination order for *UNIF1*.

definition *unifless* :: (('*f*, '*v*) *equation multiset* $\times$ ('*f*, '*v*) *equation multiset*) *set*
where
unifless = *inv-image* (*finite-psubset* <∗*lex*∗> *measure size-mset*) (λ*x*. (*vars-mset* *x*, *x*))

lemma *wf-unifless*:
wf unifless
by (*auto simp*: *unifless-def*)

lemma *UNIF1-vars-mset-leq*:
assumes *UNIF1* σ *E E'*
shows *vars-mset E'* $\subseteq$ *vars-mset E*
using *assms* **by** (*cases*) (*auto dest*: *mem-vars-mset-subst-mset*)

lemma *vars-mset-subset-size-mset-uniflessI* [*intro*]:
vars-mset M $\subseteq$ *vars-mset N* $\Longrightarrow$ *size-mset M* < *size-mset N* $\Longrightarrow$ (*M*, *N*) $\in$ *unifless*
by (*auto simp*: *unifless-def finite-vars-mset*)

lemma *vars-mset-psubset-uniflessI* [*intro*]:
vars-mset M $\subset$ *vars-mset N* $\Longrightarrow$ (*M*, *N*) $\in$ *unifless*
by (*auto simp*: *unifless-def finite-vars-mset*)

lemma *UNIF1-unifless*:
assumes *UNIF1* σ *E E'*
shows (*E'*, *E*) $\in$ *unifless*
proof −
have *vars-mset E'* $\subseteq$ *vars-mset E*
using *UNIF1-vars-mset-leq* [*OF assms*] **.**
with *assms*
show *?thesis*
apply *cases*
apply (*auto simp*: *pair-size-def intro*!: *Var-left-vars-mset-less Var-right-vars-mset-less*)
apply (*rule vars-mset-subset-size-mset-uniflessI*)
apply *auto*
using *size-mset-Fun-less* **by** *fastforce*
qed

lemma *converse-unif-subset-unifless*:
unif^{-1} $\subseteq$ *unifless*
using *UNIF1-unifless* **by** (*auto simp*: *unif-def*)

5.2.2 Termination of the Inference Rules

lemma *wf-converse-unif*:

wf (*unif*$^{-1}$)
by (*rule wf-subset* [*OF wf-unifless converse-unif-subset-unifless*])

Reflexive and transitive closure of *UNIF1* collecting substitutions produced by single steps.

inductive
UNIF :: (*'f*, *'v*) *subst list* ⇒ (*'f*, *'v*) *equation multiset* ⇒ (*'f*, *'v*) *equation multiset* ⇒ *bool*
where
empty [*simp*, *intro*!]: *UNIF* [] *E E* |
step [*intro*]: *UNIF1* σ *E E'* $\Longrightarrow$ *UNIF ss E' E''* $\Longrightarrow$ *UNIF* (σ # *ss*) *E E''*

lemma *unif-rtrancl-UNIF-conv*:
(*E*, *E'*) ∈ *unif** ⟷ (∃ *ss*. *UNIF ss E E'*)
proof
assume (*E*, *E'*) ∈ *unif**
then show ∃ *ss*. *UNIF ss E E'*
by (*induct rule*: *converse-rtrancl-induct*) (*auto simp*: *unif-UNIF1-conv*)
next
assume ∃ *ss*. *UNIF ss E E'*
then obtain *ss* **where** *UNIF ss E E'* **..**
then show (*E*, *E'*) ∈ *unif** **by** (*induct*) (*auto dest*: *UNIF1-unifD*)
qed

Compose a list of substitutions.

definition *compose* :: (*'f*, *'v*) *subst list* ⇒ (*'f*, *'v*) *subst*
where
compose ss = *List.foldr* ($\circ_s$) *ss Var*

lemma *compose-simps* [*simp*]:
compose [] = *Var*
compose (*Var* # *ss*) = *compose ss*
compose (σ # *ss*) = σ $\circ_s$ *compose ss*
by (*simp-all add*: *compose-def*)

lemma *compose-append* [*simp*]:
compose (*ss* @ *ts*) = *compose ss* $\circ_s$ *compose ts*
using *foldr-assoc* [*of* ($\circ_s$) *ss Var foldr* ($\circ_s$) *ts Var*]
by (*simp add*: *compose-def ac-simps*)

lemma *set-mset-subst-mset* [*simp*]:
set-mset (*subst-mset* σ *E*) = *subst-set* σ (*set-mset E*)
by (*auto simp*: *subst-set-def subst-mset-def*)

lemma *UNIF1-subst-domain-Int*:
assumes *UNIF1* σ *E E'*
shows *subst-domain* σ ∩ *vars-mset E'* = {}
using *assms* **by** (*cases*) *simp*+

```
lemma UNIF1-subst-domain-subset:
  assumes UNIF1 σ E E'
  shows subst-domain σ ⊆ vars-mset E
  using assms by (cases) simp+

lemma UNIF-subst-domain-subset:
  assumes UNIF ss E E'
  shows subst-domain (compose ss) ⊆ vars-mset E
  using assms
  by (induct)
    (auto dest: UNIF1-subst-domain-subset UNIF1-vars-mset-leq simp: subst-domain-subst-compose)

lemma UNIF1-range-vars-subset:
  assumes UNIF1 σ E E'
  shows range-vars σ ⊆ vars-mset E
  using assms by (cases) (auto simp: range-vars-def)

lemma UNIF1-subst-domain-range-vars-Int:
  assumes UNIF1 σ E E'
  shows subst-domain σ ∩ range-vars σ = {}
  using assms by (cases) auto

lemma UNIF-range-vars-subset:
  assumes UNIF ss E E'
  shows range-vars (compose ss) ⊆ vars-mset E
  using assms
  by (induct)
     (auto dest: UNIF1-range-vars-subset UNIF1-vars-mset-leq
          dest!: range-vars-subst-compose-subset [THEN subsetD])

lemma UNIF-subst-domain-range-vars-Int:
  assumes UNIF ss E E'
  shows subst-domain (compose ss) ∩ range-vars (compose ss) = {}
using assms
proof (induct)
  case (step σ E E' ss E'')
  from UNIF1-subst-domain-Int [OF step(1)]
    and UNIF-subst-domain-subset [OF step(2)]
    and UNIF1-subst-domain-range-vars-Int [OF step(1)]
    and UNIF-range-vars-subset [OF step(2)]
    have subst-domain σ ∩ range-vars σ = {}
      and subst-domain (compose ss) ∩ subst-domain σ = {}
      and subst-domain σ ∩ range-vars (compose ss) = {} by blast+
  then have (subst-domain σ ∪ subst-domain (compose ss)) ∩
    ((range-vars σ − subst-domain (compose ss)) ∪ range-vars (compose ss)) = {}
    using step(3) by auto
  then show ?case
    using subst-domain-subst-compose [of σ compose ss]
    and range-vars-subst-compose-subset [of σ compose ss]
```

by (*auto*)
qed *simp*

The inference rules generate idempotent substitutions.

lemma *UNIF-idemp*:
assumes *UNIF ss E E'*
shows *compose ss* $\circ_s$ *compose ss* = *compose ss*
using *UNIF-subst-domain-range-vars-Int* [*OF assms*]
by (*simp only*: *subst-idemp-iff*)

lemma *UNIF1-mono*:
assumes *UNIF1* σ *E E'*
shows *UNIF1* σ (*E* + *M*) (*E'* + *subst-mset* σ *M*)
using *assms*
by (*cases*) (*auto intro*: *UNIF1.intros simp*: *union-commutes subst-mset-union* [*symmetric*])

lemma *unif-mono*:
assumes (*E*, *E'*) ∈ *unif*
shows ∃σ. (*E* + *M*, *E'* + *subst-mset* σ *M*) ∈ *unif*
using *assms* **by** (*auto simp*: *unif-UNIF1-conv intro*: *UNIF1-mono*)

lemma *unif-rtrancl-mono*:
assumes (*E*, *E'*) ∈ *unif**
shows ∃σ. (*E* + *M*, *E'* + *subst-mset* σ *M*) ∈ *unif**
using *assms*
proof (*induction arbitrary*: *M rule*: *converse-rtrancl-induct*)
case *base*
have (*E'* + *M*, *E'* + *subst-mset Var M*) ∈ *unif** **by** *auto*
then show *?case* **by** *blast*
next
case (*step E F*)
obtain σ **where** (*E* + *M*, *F* + *subst-mset* σ *M*) ∈ *unif*
using *unif-mono* [*OF* ‹(*E*, *F*) ∈ *unif*›] **..**
moreover obtain τ
where (*F* + *subst-mset* σ *M*, *E'* + *subst-mset* τ (*subst-mset* σ *M*)) ∈ *unif**
using *step.IH* **by** *blast*
ultimately have (*E* + *M*, *E'* + *subst-mset* (σ $\circ_s$ τ) *M*) ∈ *unif** **by** *simp*
then show *?case* **by** *blast*
qed

5.2.3 Soundness of the Inference Rules

The inference rules of unification are sound in the sense that when the empty set of equations is reached, a most general unifier is obtained.

lemma *UNIF-empty-imp-is-mgu-compose*:
fixes *E* :: (*'f*, *'v*) *equation multiset*
assumes *UNIF ss E* {#}

shows *is-mgu* (*compose ss*) (*set-mset E*)
using *assms*
proof (*induct ss E* {#}::('f, 'v) *equation multiset*)
case (*step* σ *E E'* *ss*)
then show *?case*
by (*cases*) (*auto simp*: *is-mgu-subst-set-subst*)
qed *simp*

5.2.4 Completeness of the Inference Rules

lemma *UNIF1-singleton-decomp* [*intro*]:
assumes *length ss* = *length ts*
shows *UNIF1 Var* {#(*Fun f ss*, *Fun f ts*)#} (*mset* (*zip ss ts*))
using *UNIF1.decomp* [*OF assms*, *of f* {#}] **by** *simp*

lemma *UNIF1-singleton-Var-left* [*intro*]:
$x \notin$ *vars-term t* $\Longrightarrow$ *UNIF1* (*subst x t*) {#(*Var x*, *t*)#} {#}
using *UNIF1.Var-left* [*of x t* {#}] **by** *simp*

lemma *UNIF1-singleton-Var-right* [*intro*]:
$x \notin$ *vars-term t* $\Longrightarrow$ *UNIF1* (*subst x t*) {#(*t*, *Var x*)#} {#}
using *UNIF1.Var-right* [*of x t* {#}] **by** *simp*

lemma *not-UNIF1-singleton-Var-right* [*dest*]:
$\neg$ *UNIF1 Var* {#(*Var x*, *Var y*)#} {#} $\Longrightarrow x \neq y$
$\neg$ *UNIF1* (*subst x* (*Var y*)) {#(*Var x*, *Var y*)#} {#} $\Longrightarrow x = y$
by *auto*

lemma *not-unifD*:
assumes $\neg$ ($\exists$ *E'*. ({#*e*#}, *E'*) $\in$ *unif*)
shows ($\exists$ *x t*. (*e* = (*Var x*, *t*) $\vee$ *e* = (*t*, *Var x*)) $\wedge$ *x* $\in$ *vars-term t* $\wedge$ *is-Fun t*) $\vee$
($\exists$ *f g ss ts*. *e* = (*Fun f ss*, *Fun g ts*) $\wedge$ (*f* $\neq$ *g* $\vee$ *length ss* $\neq$ *length ts*))
proof (*rule ccontr*)
assume $*$: $\neg$ *?thesis*
show *False*
proof (*cases e*)
case (*Pair s t*)
with *assms* **and** $*$ **show** *?thesis*
by (*cases s*) (*cases t*, *auto simp*: *unif-def simp del*: *term.simps*, (*blast* | *succeed*))+
qed
qed

lemma *unifiable-imp-unif*:
assumes *unifiable* {*e*}
shows $\exists$ *E'*. ({#*e*#}, *E'*) $\in$ *unif*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
from *not-unifD* [*OF this*] **and** *assms*

show *False* **by** (*auto simp*: *unifiable-def*)
qed

lemma *unifiable-imp-empty-or-unif*:
assumes *unifiable* (*set-mset E*)
shows $E = \{\#\} \vee (\exists E'. (E, E') \in unif)$
proof (*cases E*)
case [*simp*]: (*add e E'*)
from *assms* **have** *unifiable* $\{e\}$ **by** (*auto simp*: *unifiable-def unifiers-insert*)
from *unifiable-imp-unif* [*OF this*]
obtain E'' **where** $(\{\#e\#\}, E'') \in unif$ **..**
then obtain σ **where** *UNIF1* σ $\{\#e\#\}$ E'' **by** (*auto simp*: *unif-UNIF1-conv*)
from *UNIF1-mono* [*OF this*] **have** *UNIF1* σ E $(E'' + \textit{subst-mset}\ \sigma\ E')$ **by** (*auto simp*: *ac-simps*)
then show *?thesis* **by** (*auto simp*: *unif-UNIF1-conv*)
qed *simp*

lemma *UNIF1-preserves-unifiers*:
assumes *UNIF1* σ E E' **and** $\tau \in$ *unifiers* (*set-mset E*)
shows $(\sigma \circ_s \tau) \in$ *unifiers* (*set-mset E'*)
using *assms* **by** (*cases*) (*auto simp*: *unifiers-def subst-mset-def*)

lemma *unif-preserves-unifiable*:
assumes $(E, E') \in unif$ **and** *unifiable* (*set-mset E*)
shows *unifiable* (*set-mset E'*)
using *UNIF1-preserves-unifiers* [*of - E E'*] **and** *assms*
by (*auto simp*: *unif-UNIF1-conv unifiable-def*)

lemma *unif-imp-converse-unifless* [*dest*]:
$(x, y) \in unif \Longrightarrow (y, x) \in unifless$
by (*metis UNIF1-unifless unif-UNIF1-conv*)

Every unifiable set of equations can be reduced to the empty set by applying the inference rules of unification.

lemma *unifiable-imp-empty*:
assumes *unifiable* (*set-mset E*)
shows $(E, \{\#\}) \in unif^*$
using *assms*
proof (*induct E rule*: *wf-induct* [*OF wf-unifless*])
fix E :: $('f, 'v)$ *equation multiset*
presume *IH*: $\bigwedge E'. [\![(E', E) \in unifless;\ unifiable\ (set\text{-}mset\ E')]\!] \Longrightarrow$
$(E', \{\#\}) \in unif^*$
and $*$: *unifiable* (*set-mset E*)
show $(E, \{\#\}) \in unif^*$
proof (*cases* $E = \{\#\}$)
assume $E \neq \{\#\}$
with *unifiable-imp-empty-or-unif* [*OF* $*$]
obtain E' **where** $(E, E') \in unif$ **by** *auto*
with $*$ **have** $(E', E) \in unifless$ **and** *unifiable* (*set-mset E'*)

    by (*auto dest*: *unif-preserves-unifiable*)
  from *IH* [*OF this*] **and** ‹$(E, E') \in unif$›
  show *?thesis* **by** *simp*
 qed *simp*
qed *simp*

lemma *unif-rtrancl-empty-imp-unifiable*:
 assumes $(E, \{\#\}) \in unif^*$
 shows *unifiable* (*set-mset* E)
 using *assms*
 by (*auto simp*: *unif-rtrancl-UNIF-conv unifiable-def is-mgu-def*
   dest!: *UNIF-empty-imp-is-mgu-compose*)

lemma *not-unifiable-imp-not-empty-NF*:
 assumes $\neg$ *unifiable* (*set-mset* E)
 shows $\exists E'. E' \neq \{\#\} \wedge (E, E') \in unif^!$
proof (*rule ccontr*)
 assume $\neg$ *?thesis*
 then have $*$: $\bigwedge E'. (E, E') \in unif^! \Longrightarrow E' = \{\#\}$ **by** *auto*
 have *SN unif* **using** *wf-converse-unif* **by** (*auto simp*: *SN-iff-wf*)
 then obtain E' **where** $(E, E') \in unif^!$
  by (*metis SN-imp-WN UNIV-I WN-onE*)
 with $*$ **have** $(E, \{\#\}) \in unif^*$ **by** *auto*
 from *unif-rtrancl-empty-imp-unifiable* [*OF this*] **and** *assms*
 show *False* **by** *contradiction*
qed

lemma *unif-rtrancl-preserves-unifiable*:
 assumes $(E, E') \in unif^*$ **and** *unifiable* (*set-mset* E)
 shows *unifiable* (*set-mset* E')
 using *assms* **by** (*induct*) (*auto simp*: *unif-preserves-unifiable*)

The inference rules for unification are complete in the sense that whenever it is not possible to reduce a set of equations E to the empty set, then E is not unifiable.

lemma *empty-not-reachable-imp-not-unifiable*:
 assumes $(E, \{\#\}) \notin unif^*$
 shows $\neg$ *unifiable* (*set-mset* E)
 using *unifiable-imp-empty* [*of* E] **and** *assms* **by** *blast*

It is enough to reach an irreducible set of equations to conclude non-unifiability.

lemma *irreducible-reachable-imp-not-unifiable*:
 assumes $(E, E') \in unif^!$ **and** $E' \neq \{\#\}$
 shows $\neg$ *unifiable* (*set-mset* E)
proof −
 have $(E, E') \in unif^*$ **and** $(E', \{\#\}) \notin unif^*$
  using *assms* **by** (*auto simp*: *NF-not-suc*)
 moreover with *empty-not-reachable-imp-not-unifiable*
  have $\neg$ *unifiable* (*set-mset* E') **by** *fast*

ultimately show *?thesis*
  using *unif-rtrancl-preserves-unifiable* **by** *fast*
qed

end

5.3 A Concrete Unification Algorithm

theory *Unification*
  imports
    Abstract-Unification
    Option-Monad
    Renaming2
begin

definition
  $decompose\ s\ t =$
    $(case\ (s,\ t)\ of$
      $(Fun\ f\ ss,\ Fun\ g\ ts) \Rightarrow if\ f = g\ then\ zip\text{-}option\ ss\ ts\ else\ None$
    $|\ \text{-} \Rightarrow None)$

lemma *decompose-same-Fun*[*simp*]:
  $decompose\ (Fun\ f\ ss)\ (Fun\ f\ ss) = Some\ (zip\ ss\ ss)$
  by (*simp add*: *decompose-def*)

lemma *decompose-Some* [*dest*]:
  $decompose\ (Fun\ f\ ss)\ (Fun\ g\ ts) = Some\ E \Longrightarrow$
    $f = g \wedge length\ ss = length\ ts \wedge E = zip\ ss\ ts$
  by (*cases* $f = g$) (*auto simp*: *decompose-def*)

lemma *decompose-None* [*dest*]:
  $decompose\ (Fun\ f\ ss)\ (Fun\ g\ ts) = None \Longrightarrow f \neq g \vee length\ ss \neq length\ ts$
  by (*cases* $f = g$) (*auto simp*: *decompose-def*)

Applying a substitution to a list of equations.

definition
  $subst\text{-}list :: ('f,\ 'v)\ subst \Rightarrow ('f,\ 'v)\ equation\ list \Rightarrow ('f,\ 'v)\ equation\ list$
  where
    $subst\text{-}list\ \sigma\ ys = map\ (\lambda p.\ (fst\ p \cdot \sigma,\ snd\ p \cdot \sigma))\ ys$

lemma *mset-subst-list* [*simp*]:
  $mset\ (subst\text{-}list\ (subst\ x\ t)\ ys) = subst\text{-}mset\ (subst\ x\ t)\ (mset\ ys)$
  by (*auto simp*: *subst-mset-def subst-list-def*)

lemma *subst-list-append*:
  $subst\text{-}list\ \sigma\ (xs\ @\ ys) = subst\text{-}list\ \sigma\ xs\ @\ subst\text{-}list\ \sigma\ ys$
by (*auto simp*: *subst-list-def*)

function (*sequential*)

```
  unify ::
    ('f, 'v) equation list ⇒ ('v × ('f, 'v) term) list ⇒ ('v × ('f, 'v) term) list option
where
  unify [] bs = Some bs
| unify ((Fun f ss, Fun g ts) # E) bs =
    (case decompose (Fun f ss) (Fun g ts) of
      None ⇒ None
    | Some us ⇒ unify (us @ E) bs)
| unify ((Var x, t) # E) bs =
    (if t = Var x then unify E bs
    else if x ∈ vars-term t then None
    else unify (subst-list (subst x t) E) ((x, t) # bs))
| unify ((t, Var x) # E) bs =
    (if x ∈ vars-term t then None
    else unify (subst-list (subst x t) E) ((x, t) # bs))
  by pat-completeness auto
termination
  by (standard, rule wf-inv-image [of unif⁻¹ mset ∘ fst, OF wf-converse-unif])
     (force intro: UNIF1.intros simp: unif-def union-commute)+

lemma unify-append-prefix-same:
  (∀ e ∈ set es1. fst e = snd e) ⟹ unify (es1 @ es2) bs = unify es2 bs
proof (induction es1 @ es2 bs arbitrary: es1 es2 bs rule: unify.induct)
  case (1 bs)
  thus ?case by simp
next
  case (2 f ss g ts E bs)
  show ?case
  proof (cases es1)
    case Nil
    thus ?thesis by simp
  next
    case (Cons e es1')
    hence e-def: e = (Fun f ss, Fun g ts) and E-def: E = es1' @ es2
      using 2 by simp-all
    hence f = g and ss = ts
      using 2.prems local.Cons by auto
    hence unify (es1 @ es2) bs = unify ((zip ts ts @ es1') @ es2) bs
      by (simp add: Cons e-def)
    also have ... = unify es2 bs
    proof (rule 2.hyps(1))
      show decompose (Fun f ss) (Fun g ts) = Some (zip ts ts)
        by (simp add: ‹f = g› ‹ss = ts›)
    next
      show zip ts ts @ E = (zip ts ts @ es1') @ es2
        by (simp add: E-def)
    next
      show ∀ e∈set (zip ts ts @ es1'). fst e = snd e
        using 2.prems by (auto simp: Cons zip-same)
```

    qed
    finally show *?thesis* **.**
  qed
next
  case (*3 x t E bs*)
  show *?case*
  proof (*cases es1*)
    case *Nil*
    thus *?thesis* **by** *simp*
  next
    case (*Cons e es1′*)
    hence *e-def*: *e* = (*Var x*, *t*) **and** *E-def*: *E* = *es1′* @ *es2*
      using *3* **by** *simp-all*
    show *?thesis*
    proof (*cases t* = *Var x*)
      case *True*
      show *?thesis*
        using *3*(*1*)[*OF True E-def*]
        using *3.hyps*(*3*) *3.prems local.Cons* **by** *fastforce*
    next
      case *False*
      thus *?thesis*
        using *3.prems e-def local.Cons* **by** *force*
    qed
  qed
next
  case (*4 v va x E bs*)
  then show *?case*
  proof (*cases es1*)
    case *Nil*
    thus *?thesis* **by** *simp*
  next
    case (*Cons e es1′*)
    hence *e-def*: *e* = (*Fun v va*, *Var x*) **and** *E-def*: *E* = *es1′* @ *es2*
      using *4* **by** *simp-all*
    thus *?thesis*
      using *4.prems local.Cons* **by** *fastforce*
  qed
qed

corollary *unify-Cons-same*:
  fst e = *snd e* ⟹ *unify* (*e* # *es*) *bs* = *unify es bs*
  by (*rule unify-append-prefix-same*[*of* [-], *simplified*])

corollary *unify-same*:
  (∀ *e* ∈ *set es*. *fst e* = *snd e*) ⟹ *unify es bs* = *Some bs*
  by (*rule unify-append-prefix-same*[*of* - [], *simplified*])

definition *subst-of* :: (′*v* × (′*f*, ′*v*) *term*) *list* ⇒ (′*f*, ′*v*) *subst*

where
  subst-of ss = *List.foldr* (λ(*x*, *t*) σ. $\sigma \circ_s$ *subst x t*) *ss Var*

Computing the mgu of two terms.

definition *mgu* :: ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *subst option* **where**
  mgu s t =
    (*case unify* [(*s*, *t*)] [] *of*
      None ⇒ *None*
    | *Some res* ⇒ *Some* (*subst-of res*))

lemma *subst-of-simps* [*simp*]:
  subst-of [] = *Var*
  subst-of ((*x*, *Var x*) # *ss*) = *subst-of ss*
  subst-of (*b* # *ss*) = *subst-of ss* $\circ_s$ *subst* (*fst b*) (*snd b*)
  by (*simp-all add*: *subst-of-def split*: *prod.splits*)

lemma *subst-of-append* [*simp*]:
  subst-of (*ss* @ *ts*) = *subst-of ts* $\circ_s$ *subst-of ss*
  by (*induct ss*) (*auto simp*: *ac-simps*)

The concrete algorithm *unify* can be simulated by the inference rules of *UNIF*.

lemma *unify-Some-UNIF*:
  assumes *unify E bs* = *Some cs*
  shows ∃ *ds ss*. *cs* = *ds* @ *bs* ∧ *subst-of ds* = *compose ss* ∧ *UNIF ss* (*mset E*) {#}
using *assms*
proof (*induction E bs arbitrary*: *cs rule*: *unify.induct*)
  case (*2 f ss g ts E bs*)
  then obtain *us* **where** *decompose* (*Fun f ss*) (*Fun g ts*) = *Some us*
    and [*simp*]: *f* = *g length ss* = *length ts us* = *zip ss ts*
    and *unify* (*us* @ *E*) *bs* = *Some cs* **by** (*auto split*: *option.splits*)
  from *2.IH* [*OF this*(*1*, *5*)] **obtain** *xs ys*
    where *cs* = *xs* @ *bs*
    and [*simp*]: *subst-of xs* = *compose ys*
    and ∗: *UNIF ys* (*mset* (*us* @ *E*)) {#} **by** *auto*
  then have *UNIF* (*Var* # *ys*) (*mset* ((*Fun f ss*, *Fun g ts*) # *E*)) {#}
    by (*force intro*: *UNIF1.decomp simp*: *ac-simps*)
  moreover have *cs* = *xs* @ *bs* **by** *fact*
  moreover have *subst-of xs* = *compose* (*Var* # *ys*) **by** *simp*
  ultimately show *?case* **by** *blast*
next
  case (*3 x t E bs*)
  show *?case*
  proof (*cases t* = *Var x*)
    assume *t* = *Var x*
    then show *?case*
      using *3* **by** *auto* (*metis UNIF.step compose-simps*(*2*) *UNIF1.trivial*)
  next

assume *t* ≠ *Var x*
with *3* **obtain** *xs ys*
where [*simp*]: *cs* = (*ys* @ [(*x*, *t*)]) @ *bs*
and [*simp*]: *subst-of ys* = *compose xs*
and *x* ∉ *vars-term t*
and *UNIF xs* (*mset* (*subst-list* (*subst x t*) *E*)) {#}
by (*cases x* ∈ *vars-term t*) *force+*
then have *UNIF* (*subst x t* # *xs*) (*mset* ((*Var x*, *t*) # *E*)) {#}
by (*force intro*: *UNIF1.Var-left simp*: *ac-simps*)
moreover have *cs* = (*ys* @ [(*x*, *t*)]) @ *bs* **by** *simp*
moreover have *subst-of* (*ys* @ [(*x*, *t*)]) = *compose* (*subst x t* # *xs*) **by** *simp*
ultimately show *?case* **by** *blast*
qed
next
case (*4 f ss x E bs*)
with *4* **obtain** *xs ys*
where [*simp*]: *cs* = (*ys* @ [(*x*, *Fun f ss*)]) @ *bs*
and [*simp*]: *subst-of ys* = *compose xs*
and *x* ∉ *vars-term* (*Fun f ss*)
and *UNIF xs* (*mset* (*subst-list* (*subst x* (*Fun f ss*)) *E*)) {#}
by (*cases x* ∈ *vars-term* (*Fun f ss*)) *force+*
then have *UNIF* (*subst x* (*Fun f ss*) # *xs*) (*mset* ((*Fun f ss*, *Var x*) # *E*)) {#}
by (*force intro*: *UNIF1.Var-right simp*: *ac-simps*)
moreover have *cs* = (*ys* @ [(*x*, *Fun f ss*)]) @ *bs* **by** *simp*
moreover have *subst-of* (*ys* @ [(*x*, *Fun f ss*)]) = *compose* (*subst x* (*Fun f ss*) # *xs*) **by** *simp*
ultimately show *?case* **by** *blast*
qed *force*

lemma *unify-sound*:
assumes *unify E* [] = *Some cs*
shows *is-imgu* (*subst-of cs*) (*set E*)
proof −
from *unify-Some-UNIF* [*OF assms*] **obtain** *ss*
where *subst-of cs* = *compose ss*
and *UNIF ss* (*mset E*) {#} **by** *auto*
with *UNIF-empty-imp-is-mgu-compose* [*OF this*(*2*)]
and *UNIF-idemp* [*OF this*(*2*)]
show *?thesis*
by (*auto simp add*: *is-imgu-def is-mgu-def*)
(*metis subst-compose-assoc*)
qed

lemma *mgu-sound*:
assumes *mgu s t* = *Some σ*
shows *is-imgu σ* {(*s*, *t*)}
proof −
obtain *ss* **where** *unify* [(*s*, *t*)] [] = *Some ss*
and *σ* = *subst-of ss*

    using *assms* **by** (*auto simp*: *mgu-def split*: *option.splits*)
  then have *is-imgu* σ (*set* [(*s*, *t*)]) **by** (*metis unify-sound*)
  then show *?thesis* **by** *simp*
qed

If *unify* gives up, then the given set of equations cannot be reduced to the empty set by *UNIF*.

lemma *unify-None*:
  assumes *unify E ss* = *None*
  shows $\exists E'.\ E' \neq \{\#\} \wedge (mset\ E,\ E') \in unif^!$
using *assms*
proof (*induction E ss rule*: *unify.induct*)
  case (*1 bs*)
  then show *?case* **by** *simp*
next
  case (*2 f ss g ts E bs*)
  moreover
  { assume $*$: *decompose* (*Fun f ss*) (*Fun g ts*) = *None*
    have *?case*
    proof (*cases unifiable* (*set E*))
      case *True*
      then have $(mset\ E,\ \{\#\}) \in unif^*$
        by (*simp add*: *unifiable-imp-empty*)
      from *unif-rtrancl-mono* [*OF this*, *of* {#(*Fun f ss*, *Fun g ts*)#}] **obtain** σ
         where (*mset E* + {#(*Fun f ss*, *Fun g ts*)#}, {#(*Fun f ss* $\cdot$ σ, *Fun g ts* $\cdot$ σ)#}) $\in unif^*$
        by (*auto simp*: *subst-mset-def*)
      moreover have {#(*Fun f ss* $\cdot$ σ, *Fun g ts* $\cdot$ σ)#} $\in$ *NF unif*
        using *decompose-None* [*OF* $*$]
        by (*auto simp*: *single-is-union NF-def unif-def elim*!: *UNIF1.cases*)
          (*metis length-map*)
      ultimately show *?thesis*
        by *auto* (*metis normalizability-I add-mset-not-empty*)
    next
      case *False*
      moreover have $\neg$ *unifiable* {(*Fun f ss*, *Fun g ts*)}
        using $*$ **by** (*auto simp*: *unifiable-def*)
      ultimately have $\neg$ *unifiable* (*set* ((*Fun f ss*, *Fun g ts*) # *E*)) **by** (*auto simp*: *unifiable-def unifiers-def*)
      then show *?thesis* **by** (*simp add*: *not-unifiable-imp-not-empty-NF*)
    qed }
  moreover
  { fix *us*
    assume $*$: *decompose* (*Fun f ss*) (*Fun g ts*) = *Some us*
      and *unify* (*us* @ *E*) *bs* = *None*
    from *2.IH* [*OF this*] **obtain** E'
      where $E' \neq \{\#\}$ **and** $(mset\ (us\ @\ E),\ E') \in unif^!$ **by** *blast*
    moreover have (*mset* ((*Fun f ss*, *Fun g ts*) # *E*), *mset* (*us* @ *E*)) $\in$ *unif*
    proof –

    have $g = f$ **and** *length ss* $=$ *length ts* **and** *us* $=$ *zip ss ts*
      using $*$ **by** *auto*
    then show *?thesis*
      by (*auto intro*: *UNIF1.decomp simp*: *unif-def ac-simps*)
  qed
  ultimately have *?case* **by** *auto* **}**
 ultimately show *?case* **by** (*auto split*: *option.splits*)
next
 case (*3 x t E bs*)
 { assume [*simp*]: $t = $ *Var x*
  obtain E' **where** $E' \neq \{\#\}$ **and** (*mset E*, E') $\in$ $unif^!$ **using** *3* **by** *auto*
  moreover have (*mset* ((*Var x*, *t*) # *E*), *mset E*) $\in$ *unif*
   by (*auto intro*: *UNIF1.trivial simp*: *unif-def*)
  ultimately have *?case* **by** *auto* **}**
 moreover
 { assume $*$: $t \neq$ *Var x* $x \notin$ *vars-term t*
  then obtain E' **where** $E' \neq \{\#\}$
   and (*mset* (*subst-list* (*subst x t*) *E*), E') $\in$ $unif^!$ **using** *3* **by** *auto*
  moreover have (*mset* ((*Var x*, *t*) # *E*), *mset* (*subst-list* (*subst x t*) *E*)) $\in$ *unif*
   using $*$ **by** (*auto intro*: *UNIF1.Var-left simp*: *unif-def*)
  ultimately have *?case* **by** *auto* **}**
 moreover
 { assume $*$: $t \neq$ *Var x* $x \in$ *vars-term t*
  then have $x \in$ *vars-term t is-Fun t* **by** *auto*
  then have $\neg$ *unifiable* {(*Var x*, *t*)} **by** (*rule in-vars-is-Fun-not-unifiable*)
  then have $**$: $\neg$ *unifiable* {(*Var x* $\cdot$ σ, *t* $\cdot$ σ)} **for** σ :: ($'b$, $'a$) *subst*
  using *subst-set-reflects-unifiable* [*of* σ {(*Var x*, *t*)}] **by** (*auto simp*: *subst-set-def*)
  have *?case*
  proof (*cases unifiable* (*set E*))
   case *True*
   then have (*mset E*, {#}) $\in$ $unif^*$
    by (*simp add*: *unifiable-imp-empty*)
   from *unif-rtrancl-mono* [*OF this*, *of* {#(*Var x*, *t*)#}] **obtain** σ
    where (*mset E* + {#(*Var x*, *t*)#}, {#(*Var x* $\cdot$ σ, *t* $\cdot$ σ)#}) $\in$ $unif^*$
    by (*auto simp*: *subst-mset-def*)
   moreover obtain E' **where** $E' \neq \{\#\}$
    and ({#(*Var x* $\cdot$ σ, *t* $\cdot$ σ)#}, E') $\in$ $unif^!$
    using *not-unifiable-imp-not-empty-NF* **and** $**$
     by (*metis set-mset-single*)
   ultimately show *?thesis* **by** *auto*
  next
   case *False*
   moreover have $\neg$ *unifiable* {(*Var x*, *t*)}
    using $*$ **by** (*force simp*: *unifiable-def*)
  ultimately have $\neg$ *unifiable* (*set* ((*Var x*, *t*) # *E*)) **by** (*auto simp*: *unifiable-def unifiers-def*)
   then show *?thesis*
    by (*simp add*: *not-unifiable-imp-not-empty-NF*)
  qed }

ultimately show *?case* **by** *blast*
next
case (*4 f ss x E bs*)
define *t* **where** $t = Fun\ f\ ss$
{ assume $*$: $x \notin$ *vars-term t*
then obtain E' **where** $E' \neq \{\#\}$
and (*mset* (*subst-list* (*subst x t*) *E*), E') $\in$ $unif^!$ **using** *4* **by** (*auto simp*: *t-def*)
moreover have (*mset* ((*t*, *Var x*) # *E*), *mset* (*subst-list* (*subst x t*) *E*)) $\in$ *unif*
using $*$ **by** (*auto intro*: *UNIF1.Var-right simp*: *unif-def*)
ultimately have *?case* **by** (*auto simp*: *t-def*) **}**
moreover
{ assume $x \in$ *vars-term t*
then have $*$: $x \in$ *vars-term t* $t \neq$ *Var x* **by** (*auto simp*: *t-def*)
then have $x \in$ *vars-term t is-Fun t* **by** *auto*
then have $\neg$ *unifiable* {(*Var x*, *t*)} **by** (*rule in-vars-is-Fun-not-unifiable*)
then have $**$: $\neg$ *unifiable* {(*Var x* $\cdot$ σ, *t* $\cdot$ σ)} **for** σ :: ('*b*, '*a*) *subst*
using *subst-set-reflects-unifiable* [*of* σ {(*Var x*, *t*)}] **by** (*auto simp*: *subst-set-def*)
have *?case*
proof (*cases unifiable* (*set E*))
case *True*
then have (*mset E*, {#}) $\in$ $unif^*$
by (*simp add*: *unifiable-imp-empty*)
from *unif-rtrancl-mono* [*OF this*, *of* {#(*t*, *Var x*)#}] **obtain** σ
where (*mset E* + {#(*t*, *Var x*)#}, {#(*t* $\cdot$ σ, *Var x* $\cdot$ σ)#}) $\in$ $unif^*$
by (*auto simp*: *subst-mset-def*)
moreover obtain E' **where** $E' \neq \{\#\}$
and ({#(*t* $\cdot$ σ, *Var x* $\cdot$ σ)#}, E') $\in$ $unif^!$
using *not-unifiable-imp-not-empty-NF* **and** $**$
by (*metis unifiable-insert-swap set-mset-single*)
ultimately show *?thesis* **by** (*auto simp*: *t-def*)
next
case *False*
moreover have $\neg$ *unifiable* {(*t*, *Var x*)}
using $*$ **by** (*simp add*: *unifiable-def*)
ultimately have $\neg$ *unifiable* (*set* ((*t*, *Var x*) # *E*)) **by** (*auto simp*: *unifiable-def unifiers-def*)
then show *?thesis* **by** (*simp add*: *not-unifiable-imp-not-empty-NF t-def*)
qed }
ultimately show *?case* **by** *blast*
qed

lemma *unify-complete*:
assumes *unify E bs* = *None*
shows *unifiers* (*set E*) = {}
proof –
from *unify-None* [*OF assms*] **obtain** E'
where $E' \neq \{\#\}$ **and** (*mset E*, E') $\in$ $unif^!$ **by** *blast*
then have $\neg$ *unifiable* (*set E*)

using *irreducible-reachable-imp-not-unifiable* **by** *force*
then show *?thesis*
by (*auto simp*: *unifiable-def*)
qed

corollary *ex-unify-if-unifiers-not-empty*:
unifiers es $\neq$ {} $\Longrightarrow$ *set xs* = *es* $\Longrightarrow$ $\exists$ *ys. unify xs* [] = *Some ys*
using *unify-complete* **by** *auto*

lemma *mgu-complete*:
mgu s t = *None* $\Longrightarrow$ *unifiers* {(*s*, *t*)} = {}
proof −
assume *mgu s t* = *None*
then have *unify* [(*s*, *t*)] [] = *None* **by** (*cases unify* [(*s*, *t*)] [], *auto simp*: *mgu-def*)
then have *unifiers* (*set* [(*s*, *t*)]) = {} **by** (*rule unify-complete*)
then show *?thesis* **by** *simp*
qed

corollary *ex-mgu-if-unifiers-not-empty*:
unifiers {(*t*,*u*)} $\neq$ {} $\Longrightarrow$ $\exists\mu$. *mgu t u* = *Some* μ
using *mgu-complete* **by** *auto*

corollary *ex-mgu-if-subst-apply-term-eq-subst-apply-term*:
fixes *t u* :: (*'f*, *'v*) *Term.term* **and** σ :: (*'f*, *'v*) *subst*
assumes *t-eq-u*: *t* · σ = *u* · σ
shows $\exists\mu$ :: (*'f*, *'v*) *subst*. *Unification.mgu t u* = *Some* μ
proof −
from *t-eq-u* **have** *unifiers* {(*t*, *u*)} $\neq$ {}
unfolding *unifiers-def* **by** *auto*
thus *?thesis*
by (*rule ex-mgu-if-unifiers-not-empty*)
qed

lemma *finite-subst-domain-subst-of*:
finite (*subst-domain* (*subst-of xs*))
proof (*induct xs*)
case (*Cons x xs*)
moreover have *finite* (*subst-domain* (*subst* (*fst x*) (*snd x*))) **by** (*metis finite-subst-domain-subst*)
ultimately show *?case*
using *subst-domain-compose* [*of subst-of xs subst* (*fst x*) (*snd x*)]
by (*simp del*: *subst-subst-domain*) (*metis finite-subset infinite-Un*)
qed *simp*

lemma *unify-subst-domain*:
assumes *unif*: *unify E* [] = *Some xs*
shows *subst-domain* (*subst-of xs*) $\subseteq$ ($\bigcup$ *e* $\in$ *set E*. *vars-term* (*fst e*) $\cup$ *vars-term*
(*snd e*))
proof −
from *unify-Some-UNIF*[*OF unif*] **obtain** *xs'* **where**

subst-of xs = *compose xs'* **and** *UNIF xs'* (*mset E*) {#}
by *auto*
thus *?thesis*
using *UNIF-subst-domain-subset*
by (*metis* (*mono-tags*, *lifting*) *multiset.set-map set-mset-mset vars-mset-def*)
qed

lemma *mgu-subst-domain*:
assumes *mgu s t* = *Some* σ
shows *subst-domain* $\sigma \subseteq$ *vars-term s* $\cup$ *vars-term t*
proof −
obtain *xs* **where** *unify* [(*s*, *t*)] [] = *Some xs* **and** σ = *subst-of xs*
using *assms* **by** (*simp add*: *mgu-def split*: *option.splits*)
thus *?thesis*
using *unify-subst-domain* **by** *fastforce*
qed

lemma *mgu-finite-subst-domain*:
mgu s t = *Some* $\sigma \Longrightarrow$ *finite* (*subst-domain* σ)
by (*drule mgu-subst-domain*) (*simp add*: *finite-subset*)

lemma *unify-range-vars*:
assumes *unif*: *unify E* [] = *Some xs*
shows *range-vars* (*subst-of xs*) $\subseteq$ ($\bigcup e \in$ *set E*. *vars-term* (*fst e*) $\cup$ *vars-term* (*snd e*))
proof −
from *unify-Some-UNIF*[*OF unif*] **obtain** *xs'* **where**
subst-of xs = *compose xs'* **and** *UNIF xs'* (*mset E*) {#}
by *auto*
thus *?thesis*
using *UNIF-range-vars-subset*
by (*metis* (*mono-tags*, *lifting*) *multiset.set-map set-mset-mset vars-mset-def*)
qed

lemma *mgu-range-vars*:
assumes *mgu s t* = *Some* μ
shows *range-vars* $\mu \subseteq$ *vars-term s* $\cup$ *vars-term t*
proof −
obtain *xs* **where** *unify* [(*s*, *t*)] [] = *Some xs* **and** μ = *subst-of xs*
using *assms* **by** (*simp add*: *mgu-def split*: *option.splits*)
thus *?thesis*
using *unify-range-vars* **by** *fastforce*
qed

lemma *unify-subst-domain-range-vars-disjoint*:
assumes *unif*: *unify E* [] = *Some xs*
shows *subst-domain* (*subst-of xs*) $\cap$ *range-vars* (*subst-of xs*) = {}
proof −
from *unify-Some-UNIF*[*OF unif*] **obtain** *xs'* **where**

subst-of xs = *compose xs'* **and** *UNIF xs'* (*mset E*) {#}
    by *auto*
  thus *?thesis*
    using *UNIF-subst-domain-range-vars-Int* **by** *metis*
qed

lemma *mgu-subst-domain-range-vars-disjoint*:
  assumes *mgu s t* = *Some* μ
  shows *subst-domain* $\mu \cap$ *range-vars* μ = {}
proof −
  obtain *xs* **where** *unify* [(*s*, *t*)] [] = *Some xs* **and** μ = *subst-of xs*
    using *assms* **by** (*simp add*: *mgu-def split*: *option.splits*)
  thus *?thesis*
    using *unify-subst-domain-range-vars-disjoint* **by** *metis*
qed

corollary *subst-apply-term-eq-subst-apply-term-if-mgu*:
  assumes *mgu-t-u*: *mgu t u* = *Some* μ
  shows $t \cdot \mu = u \cdot \mu$
  using *mgu-sound*[*OF mgu-t-u*]
  by (*simp add*: *is-imgu-def unifiers-def*)

lemma *mgu-same*: *mgu t t* = *Some Var*
  by (*simp add*: *mgu-def unify-same*)

lemma *mgu-is-Var-if-not-in-equations*:
  fixes μ :: ('*f*, '*v*) *subst* **and** *E* :: ('*f*, '*v*) *equations* **and** *x* :: '*v*
  assumes
    mgu-μ: *is-mgu* μ *E* **and**
    x-not-in: $x \notin (\bigcup e{\in}E.$ *vars-term* (*fst e*) $\cup$ *vars-term* (*snd e*))
  shows *is-Var* (μ *x*)
proof −
  from *mgu-*μ **have** *unif-*μ: $\mu \in$ *unifiers E* **and** *minimal-*μ: $\forall \tau \in$ *unifiers E*. $\exists \gamma$.
$\tau = \mu \circ_s \gamma$
    by (*simp-all add*: *is-mgu-def*)

  define τ :: ('*f*, '*v*) *subst* **where**
    $\tau = (\lambda x.$ *if* $x \in (\bigcup e \in E.$ *vars-term* (*fst e*) $\cup$ *vars-term* (*snd e*)) *then* μ *x else*
Var x)

  have ‹$\tau \in$ *unifiers E*›
    unfolding *unifiers-def mem-Collect-eq*
  proof (*rule ballI*)
    fix *e* **assume** $e \in E$
    with *unif-*μ **have** *fst e* $\cdot \mu$ = *snd e* $\cdot \mu$
      by *blast*
    moreover from ‹$e \in E$› **have** *fst e* $\cdot \tau$ = *fst e* $\cdot \mu$ **and** *snd e* $\cdot \tau$ = *snd e* $\cdot \mu$
      unfolding *term-subst-eq-conv*
      by (*auto simp*: τ*-def*)

ultimately show $fst\ e \cdot \tau = snd\ e \cdot \tau$
by *simp*
qed
with *minimal-μ* **obtain** γ **where** $\mu \circ_s \gamma = \tau$
by *auto*
with *x-not-in* **have** $(\mu \circ_s \gamma)\ x = Var\ x$
by (*simp add*: *τ-def*)
thus *is-Var* ($\mu\ x$)
by (*metis subst-apply-eq-Var subst-compose term.disc*(*1*))
qed

corollary *mgu-ball-is-Var*:
is-mgu μ $E \Longrightarrow \forall x \in - (\bigcup e{\in}E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e)).\ is\text{-}Var\ (\mu\ x)$
by (*rule ballI*) (*rule mgu-is-Var-if-not-in-equations*[*folded Compl-iff*])

lemma *mgu-inj-on*:
fixes μ :: ($'f$, $'v$) *subst* **and** E :: ($'f$, $'v$) *equations*
assumes *mgu-μ*: *is-mgu* μ E
shows *inj-on* μ $(- (\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e)))$
proof (*rule inj-onI*)
fix x y
assume
x-in: $x \in - (\bigcup e{\in}E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$ **and**
y-in: $y \in - (\bigcup e{\in}E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$ **and**
$\mu\ x = \mu\ y$

from *mgu-μ* **have** *unif-μ*: $\mu \in unifiers\ E$ **and** *minimal-μ*: $\forall \tau \in unifiers\ E.\ \exists \gamma.\ \tau = \mu \circ_s \gamma$
by (*simp-all add*: *is-mgu-def*)

define τ :: ($'f$, $'v$) *subst* **where**
$\tau = (\lambda x.\ if\ x \in (\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))\ then\ \mu\ x\ else\ Var\ x)$

have ‹$\tau \in unifiers\ E$›
unfolding *unifiers-def mem-Collect-eq*
proof (*rule ballI*)
fix e **assume** $e \in E$
with *unif-μ* **have** $fst\ e \cdot \mu = snd\ e \cdot \mu$
by *blast*
moreover from ‹$e \in E$› **have** $fst\ e \cdot \tau = fst\ e \cdot \mu$ **and** $snd\ e \cdot \tau = snd\ e \cdot \mu$
unfolding *term-subst-eq-conv*
by (*auto simp*: *τ-def*)
ultimately show $fst\ e \cdot \tau = snd\ e \cdot \tau$
by *simp*
qed
with *minimal-μ* **obtain** γ **where** $\mu \circ_s \gamma = \tau$
by *auto*

hence $(\mu \circ_s \gamma)\ x = \mathit{Var}\ x$ **and** $(\mu \circ_s \gamma)\ y = \mathit{Var}\ y$
  using *ComplD*[*OF x-in*] *ComplD*[*OF y-in*]
  by (*simp-all add*: τ-*def*)
with ‹$\mu\ x = \mu\ y$› **show** $x = y$
  by (*simp add*: *subst-compose-def*)
qed

lemma *imgu-subst-domain-subset*:
  fixes μ :: ($'f$, $'v$) *subst* **and** E :: ($'f$, $'v$) *equations* **and** *Evars* :: $'v$ *set*
  assumes *imgu-*μ: *is-imgu* μ E **and** *fin-E*: *finite* E
  defines $\mathit{Evars} \equiv (\bigcup e \in E.\ \mathit{vars\text{-}term}\ (\mathit{fst}\ e) \cup \mathit{vars\text{-}term}\ (\mathit{snd}\ e))$
  shows *subst-domain* $\mu \subseteq \mathit{Evars}$
proof (*intro Set.subsetI*)
  from *imgu-*μ **have** *unif-*μ: $\mu \in \mathit{unifiers}\ E$ **and** *minimal-*μ: $\forall \tau \in \mathit{unifiers}\ E.\ \mu \circ_s \tau = \tau$
    by (*simp-all add*: *is-imgu-def*)

  from *fin-E* **obtain** *es* :: ($'f$, $'v$) *equation list* **where**
    $\mathit{set}\ \mathit{es} = E$
    using *finite-list* **by** *auto*
  then obtain *xs* :: ($'v \times$ ($'f$, $'v$) *Term.term*) *list* **where**
    unify-es: *unify es* [] = *Some xs*
    using *unif-*μ *ex-unify-if-unifiers-not-empty* **by** *blast*

  define τ :: ($'f$, $'v$) *subst* **where**
    $\tau = \mathit{subst\text{-}of}\ \mathit{xs}$

  have *dom-*τ: *subst-domain* $\tau \subseteq \mathit{Evars}$
    using *unify-subst-domain*[*OF unify-es*, *unfolded* ‹*set es* = *E*›, *folded Evars-def* τ-*def*] **.**
  have *range-vars-*τ: *range-vars* $\tau \subseteq \mathit{Evars}$
    using *unify-range-vars*[*OF unify-es*, *unfolded* ‹*set es* = *E*›, *folded Evars-def* τ-*def*] **.**

  have $\tau \in \mathit{unifiers}\ E$
    using ‹*set es* = *E*› *unify-es* τ-*def is-imgu-def unify-sound* **by** *blast*
  with *minimal-*μ **have** μ-*comp-*τ: $\bigwedge x.\ (\mu \circ_s \tau)\ x = \tau\ x$
    by *auto*

  fix x :: $'v$ **assume** $x \in \mathit{subst\text{-}domain}\ \mu$
  hence $\mu\ x \neq \mathit{Var}\ x$
    by (*simp add*: *subst-domain-def*)

  show $x \in \mathit{Evars}$
  proof (*cases* $x \in \mathit{subst\text{-}domain}\ \tau$)
    case *True*
    thus *?thesis*
      using *dom-*τ **by** *auto*
  next

```
    case False
    hence τ x = Var x
      by (simp add: subst-domain-def)
    hence μ x · τ = Var x
      using μ-comp-τ[of x] by (simp add: subst-compose)
    thus ?thesis
    proof (rule subst-apply-eq-Var)
      show ⋀y. μ x = Var y ⟹ τ y = Var x ⟹ ?thesis
        using ‹μ x ≠ Var x› range-vars-τ mem-range-varsI[of τ - x] by auto
    qed
  qed
qed

lemma imgu-range-vars-of-equations-vars-subset:
  fixes μ :: ('f, 'v) subst and E :: ('f, 'v) equations and Evars :: 'v set
  assumes imgu-μ: is-imgu μ E and fin-E: finite E
  defines Evars ≡ (⋃e ∈ E. vars-term (fst e) ∪ vars-term (snd e))
  shows (⋃x ∈ Evars. vars-term (μ x)) ⊆ Evars
proof (rule Set.subsetI)
  from imgu-μ have unif-μ: μ ∈ unifiers E and minimal-μ: ∀τ ∈ unifiers E. μ ∘s
τ = τ
    by (simp-all add: is-imgu-def)

  from fin-E obtain es :: ('f, 'v) equation list where
    set es = E
    using finite-list by auto
  then obtain xs :: ('v × ('f, 'v) Term.term) list where
    unify-es: unify es [] = Some xs
    using unif-μ ex-unify-if-unifiers-not-empty by blast

  define τ :: ('f, 'v) subst where
    τ = subst-of xs

  have dom-τ: subst-domain τ ⊆ Evars
    using unify-subst-domain[OF unify-es, unfolded ‹set es = E›, folded Evars-def
τ-def] .
  have range-vars-τ: range-vars τ ⊆ Evars
     using unify-range-vars[OF unify-es, unfolded ‹set es = E›, folded Evars-def
τ-def] .
  hence ball-vars-apply-τ-subset: ∀x ∈ subst-domain τ. vars-term (τ x) ⊆ Evars
    unfolding range-vars-def
    by (simp add: SUP-le-iff)

  have τ ∈ unifiers E
    using ‹set es = E› unify-es τ-def is-imgu-def unify-sound by blast
  with minimal-μ have μ-comp-τ: ⋀x. (μ ∘s τ) x = τ x
    by auto

  fix y :: 'v assume y ∈ (⋃x ∈ Evars. vars-term (μ x))
```

then obtain x :: $'v$ **where**
x-in: $x \in$ *Evars* **and** *y-in*: $y \in$ *vars-term* $(\mu\ x)$
by (*auto simp*: *subst-domain-def*)
have *vars-τ-x*: *vars-term* $(\tau\ x) \subseteq$ *Evars*
using *ball-vars-apply-τ-subset subst-domain-def x-in* **by** *fastforce*

show $y \in$ *Evars*
proof (*rule ccontr*)
assume $y \notin$ *Evars*
hence $y \notin$ *vars-term* $(\tau\ x)$
using *vars-τ-x* **by** *blast*
moreover have $y \in$ *vars-term* $((\mu \circ_s \tau)\ x)$
proof –
have $\tau\ y = Var\ y$
using ‹$y \notin$ *Evars*› *dom-τ*
by (*auto simp add*: *subst-domain-def*)
thus *?thesis*
unfolding *subst-compose-def vars-term-subst-apply-term UN-iff*
using *y-in* **by** *force*
qed
ultimately show *False*
using *μ-comp-τ*[*of x*] **by** *simp*
qed
qed

lemma *imgu-range-vars-subset*:
fixes μ :: $('f,\ 'v)$ *subst* **and** E :: $('f,\ 'v)$ *equations*
assumes *imgu-μ*: *is-imgu* $\mu\ E$ **and** *fin-E*: *finite* E
shows *range-vars* $\mu \subseteq (\bigcup e \in E.$ *vars-term* $(fst\ e) \cup$ *vars-term* $(snd\ e))$
proof –
have *range-vars* $\mu = (\bigcup x \in$ *subst-domain* $\mu.$ *vars-term* $(\mu\ x))$
by (*simp add*: *range-vars-def*)
also have $\ldots \subseteq (\bigcup x \in (\bigcup e \in E.$ *vars-term* $(fst\ e) \cup$ *vars-term* $(snd\ e)).$ *vars-term* $(\mu\ x))$
using *imgu-subst-domain-subset*[*OF imgu-μ fin-E*] **by** *fast*
also have $\ldots \subseteq (\bigcup e \in E.$ *vars-term* $(fst\ e) \cup$ *vars-term* $(snd\ e))$
using *imgu-range-vars-of-equations-vars-subset*[*OF imgu-μ fin-E*] **by** *metis*
finally show *?thesis* **.**
qed

definition *the-mgu* :: $('f,\ 'v)$ *term* $\Rightarrow$ $('f,\ 'v)$ *term* $\Rightarrow$ $('f\ ,'v)$ *subst* **where**
the-mgu $s\ t = (case\ mgu\ s\ t\ of\ None \Rightarrow Var \mid Some\ \delta \Rightarrow \delta)$

lemma *the-mgu-is-imgu*:
fixes σ :: $('f,\ 'v)$ *subst*
assumes $s \cdot \sigma = t \cdot \sigma$
shows *is-imgu* (*the-mgu* $s\ t$) $\{(s,\ t)\}$
proof –

from *assms* **have** *unifiers* $\{(s,\ t)\} \neq \{\}$ **by** (*force simp*: *unifiers-def*)
then obtain τ **where** *mgu s t* = *Some* τ
and *the-mgu s t* = τ
using *mgu-complete* **by** (*auto simp*: *the-mgu-def*)
with *mgu-sound* **show** *?thesis* **by** *blast*
qed

lemma *the-mgu*:
fixes σ :: ('f, 'v) *subst*
assumes $s \cdot \sigma = t \cdot \sigma$
shows $s \cdot$ *the-mgu s t* = $t \cdot$ *the-mgu s t* $\wedge$ σ = *the-mgu s t* $\circ_s$ σ
proof −
have $*$: $\sigma \in$ *unifiers* $\{(s,\ t)\}$ **by** (*force simp*: *assms unifiers-def*)
show *?thesis*
proof (*cases mgu s t*)
assume *mgu s t* = *None*
then have *unifiers* $\{(s,\ t)\} = \{\}$ **by** (*rule mgu-complete*)
with $*$ **show** *?thesis* **by** *simp*
next
fix τ
assume *mgu s t* = *Some* τ
moreover then have *is-imgu* τ $\{(s,\ t)\}$ **by** (*rule mgu-sound*)
ultimately have *is-imgu* (*the-mgu s t*) $\{(s,\ t)\}$ **by** (*unfold the-mgu-def*, *simp*)
with $*$ **show** *?thesis* **by** (*auto simp*: *is-imgu-def unifiers-def*)
qed
qed

5.3.1 Unification of two terms where variables should be considered disjoint

definition
mgu-var-disjoint-generic ::
('v ⇒ 'u) ⇒ ('w ⇒ 'u) ⇒ ('f, 'v) *term* ⇒ ('f, 'w) *term* ⇒
(('f, 'v, 'u) *gsubst* × ('f, 'w, 'u) *gsubst*) *option*
where
mgu-var-disjoint-generic vu wu s t =
(*case mgu* (*map-vars-term vu s*) (*map-vars-term wu t*) *of*
None ⇒ *None*
| *Some* γ ⇒ *Some* ($\gamma \circ vu$, $\gamma \circ wu$))

lemma *mgu-var-disjoint-generic-sound*:
assumes *unif*: *mgu-var-disjoint-generic vu wu s t* = *Some* ($\gamma 1$, $\gamma 2$)
shows $s \cdot \gamma 1 = t \cdot \gamma 2$
proof −
from *unif*[*unfolded mgu-var-disjoint-generic-def*] **obtain** γ **where**
unif2: *mgu* (*map-vars-term vu s*) (*map-vars-term wu t*) = *Some* γ
by (*cases mgu* (*map-vars-term vu s*) (*map-vars-term wu t*), *auto*)
from *mgu-sound*[*OF unif2*[*unfolded mgu-var-disjoint-generic-def*], *unfolded is-imgu-def unifiers-def*]

have *map-vars-term vu s* $\cdot$ γ = *map-vars-term wu t* $\cdot$ γ **by** *auto*
from *this*[*unfolded apply-subst-map-vars-term*] *unif*[*unfolded mgu-var-disjoint-generic-def unif2*]
show *?thesis* **by** *simp*
qed

lemma *mgu-var-disjoint-generic-complete*:
fixes σ :: ('*f*, '*v*, '*u*) *gsubst* **and** τ :: ('*f*, '*w*, '*u*) *gsubst*
and *vu* :: '*v* $\Rightarrow$ '*u* **and** *wu*:: '*w* $\Rightarrow$ '*u*
assumes *inj*: *inj vu inj wu*
and *vwu*: *range vu* $\cap$ *range wu* = {}
and *unif-disj*: *s* $\cdot$ σ = *t* $\cdot$ τ
shows $\exists$ μ*1* μ*2* δ. *mgu-var-disjoint-generic vu wu s t* = *Some* (μ*1*, μ*2*) $\wedge$
σ = μ*1* $\circ_s$ δ $\wedge$
τ = μ*2* $\circ_s$ δ $\wedge$
s $\cdot$ μ*1* = *t* $\cdot$ μ*2*
proof −
note *inv1*[*simp*] = *the-inv-f-f*[*OF inj*(*1*)]
note *inv2*[*simp*] = *the-inv-f-f*[*OF inj*(*2*)]
obtain γ :: ('*f*,'*u*)*subst* **where** *gamma*: γ = (λ *x*. *if x* $\in$ *range vu then* σ (*the-inv vu x*) *else* τ (*the-inv wu x*)) **by** *auto*
have *ids*: *s* $\cdot$ σ = *map-vars-term vu s* $\cdot$ γ **unfolding** *gamma*
by (*induct s*, *auto*)
have *idt*: *t* $\cdot$ τ = *map-vars-term wu t* $\cdot$ γ **unfolding** *gamma*
by (*induct t*, *insert vwu*, *auto*)
from *unif-disj ids idt*
have *unif*: *map-vars-term vu s* $\cdot$ γ = *map-vars-term wu t* $\cdot$ γ (**is** *?s* $\cdot$ γ = *?t* $\cdot$ γ) **by** *auto*
from *the-mgu*[*OF unif*] **have** *unif2*: *?s* $\cdot$ *the-mgu ?s ?t* = *?t* $\cdot$ *the-mgu ?s ?t* **and** *inst*: γ = *the-mgu ?s ?t* $\circ_s$ γ **by** *auto*
have *mgu ?s ?t* = *Some* (*the-mgu ?s ?t*) **unfolding** *the-mgu-def*
using *mgu-complete*[*unfolded unifiers-def*] *unif*
by (*cases mgu ?s ?t*, *auto*)
with *inst* **obtain** μ **where** *mu*: *mgu ?s ?t* = *Some* μ **and** *gamma-mu*: γ = μ $\circ_s$ γ **by** *auto*
let *?tau1* = μ $\circ$ *vu*
let *?tau2* = μ $\circ$ *wu*
show *?thesis* **unfolding** *mgu-var-disjoint-generic-def mu option.simps*
proof (*intro exI conjI*, *rule refl*)
show σ = *?tau1* $\circ_s$ γ
proof (*rule ext*)
fix *x*
have (*?tau1* $\circ_s$ γ) *x* = γ (*vu x*) **using** *fun-cong*[*OF gamma-mu*, *of vu x*] **by** (*simp add*: *subst-compose-def*)
also have ... = σ *x* **unfolding** *gamma* **by** *simp*
finally show σ *x* = (*?tau1* $\circ_s$ γ) *x* **by** *simp*
qed
next

show τ = *?tau2* $\circ_s$ γ
proof (*rule ext*)
fix *x*
have (*?tau2* $\circ_s$ γ) *x* = γ (*wu x*) **using** *fun-cong*[*OF gamma-mu*, *of wu x*] **by** (*simp add*: *subst-compose-def*)
also have ... = τ *x* **unfolding** *gamma* **using** *vwu* **by** *auto*
finally show τ *x* = (*?tau2* $\circ_s$ γ) *x* **by** *simp*
qed
next
have *s* · *?tau1* = *map-vars-term vu s* · μ **unfolding** *apply-subst-map-vars-term* **..**
also have ... = *map-vars-term wu t* · μ
unfolding *unif2*[*unfolded the-mgu-def mu option.simps*] **..**
also have ... = *t* · *?tau2* **unfolding** *apply-subst-map-vars-term* **..**
finally show *s* · *?tau1* = *t* · *?tau2* **.**
qed
qed

abbreviation *mgu-var-disjoint-sum* ≡ *mgu-var-disjoint-generic Inl Inr*

lemma *mgu-var-disjoint-sum-sound*:
mgu-var-disjoint-sum s t = *Some* (γ*1*, γ*2*) ⟹ *s* · γ*1* = *t* · γ*2*
by (*rule mgu-var-disjoint-generic-sound*)

lemma *mgu-var-disjoint-sum-complete*:
fixes σ :: (*'f*, *'v*, *'v* + *'w*) *gsubst* **and** τ :: (*'f*, *'w*, *'v* + *'w*) *gsubst*
assumes *unif-disj*: *s* · σ = *t* · τ
shows ∃μ*1* μ*2* δ. *mgu-var-disjoint-sum s t* = *Some* (μ*1*, μ*2*) ∧
σ = μ*1* $\circ_s$ δ ∧
τ = μ*2* $\circ_s$ δ ∧
s · μ*1* = *t* · μ*2*
by (*rule mgu-var-disjoint-generic-complete*[*OF - - - unif-disj*], *auto simp*: *inj-on-def*)

lemma *mgu-var-disjoint-sum-instance*:
fixes σ :: (*'f*, *'v*) *subst* **and** δ :: (*'f*, *'v*) *subst*
assumes *unif-disj*: *s* · σ = *t* · δ
shows ∃μ*1* μ*2* τ. *mgu-var-disjoint-sum s t* = *Some* (μ*1*, μ*2*) ∧
σ = μ*1* $\circ_s$ τ ∧
δ = μ*2* $\circ_s$ τ ∧
s · μ*1* = *t* · μ*2*
proof −
let *?map* = λ *m* σ *v*. *map-vars-term m* (σ *v*)
let *?m* = *?map* (*Inl* :: (*'v* ⇒ *'v* + *'v*))
let *?m'* = *?map* (*case-sum* (λ *x*. *x*) (λ *x*. *x*))
from *unif-disj* **have** *id*: *map-vars-term Inl* (*s* · σ) = *map-vars-term Inl* (*t* · δ) **by** *simp*
from *mgu-var-disjoint-sum-complete*[*OF id*[*unfolded map-vars-term-subst*]]
obtain μ*1* μ*2* τ **where** *mgu*: *mgu-var-disjoint-sum s t* = *Some* (μ*1*,μ*2*)
and σ: *?m* σ = μ*1* $\circ_s$ τ

and δ: *?m* $\delta = \mu 2 \circ_s \tau$
and *unif*: $s \cdot \mu 1 = t \cdot \mu 2$ **by** *blast*
{
fix σ :: $('f, 'v)$ *subst*
have *?m'* (*?m* σ) = σ **by** (*simp add*: *map-vars-term-compose o-def term.map-ident*)
} **note** *id* = *this*
{
fix μ :: $('f,'v,'v+'v)$*gsubst* **and** τ :: $('f,'v + 'v)$*subst*
have *?m'* ($\mu \circ_s \tau$) = $\mu \circ_s$ *?m'* τ
by (*rule ext*, *unfold subst-compose-def*, *simp add*: *map-vars-term-subst*)
} **note** *id'* = *this*
from *arg-cong*[*OF* σ, *of ?m'*, *unfolded id id'*] **have** σ: $\sigma = \mu 1 \circ_s$ *?m'* τ **.**
from *arg-cong*[*OF* δ, *of ?m'*, *unfolded id id'*] **have** δ: $\delta = \mu 2 \circ_s$ *?m'* τ **.**
show *?thesis*
by (*intro exI conjI*, *rule mgu*, *rule* σ, *rule* δ, *rule unif*)
qed

5.3.2 A variable disjoint unification algorithm without changing the type

We pass the renaming function as additional argument

definition *mgu-vd* :: *'v* :: *infinite renaming2* $\Rightarrow$ - $\Rightarrow$ - **where**
mgu-vd r = *mgu-var-disjoint-generic* (*rename-1 r*) (*rename-2 r*)

lemma *mgu-vd-sound*: *mgu-vd r s t* = *Some* ($\gamma 1$, $\gamma 2$) $\Longrightarrow s \cdot \gamma 1 = t \cdot \gamma 2$
unfolding *mgu-vd-def* **by** (*rule mgu-var-disjoint-generic-sound*)

lemma *mgu-vd-complete*:
fixes σ :: $('f, 'v$ :: *infinite*) *subst* **and** τ :: $('f, 'v)$ *subst*
assumes *unif-disj*: $s \cdot \sigma = t \cdot \tau$
shows $\exists \mu 1\ \mu 2\ \delta.$ *mgu-vd r s t* = *Some* ($\mu 1$, $\mu 2$) $\wedge$
$\sigma = \mu 1 \circ_s \delta \wedge$
$\tau = \mu 2 \circ_s \delta \wedge$
$s \cdot \mu 1 = t \cdot \mu 2$
unfolding *mgu-vd-def*
by (*rule mgu-var-disjoint-generic-complete*[*OF rename-12 unif-disj*])

end

6 Matching

theory *Matching*
imports
Abstract-Matching
Unification
begin

function *match-term-list*

where
match-term-list [] σ = *Some* σ |
match-term-list ((*Var x*, *t*) # *P*) σ =
(*if* σ *x* = *None* $\vee$ σ *x* = *Some t then match-term-list P* (σ (*x* $\mapsto$ *t*))
else None) |
match-term-list ((*Fun f ss*, *Fun g ts*) # *P*) σ =
(*case decompose* (*Fun f ss*) (*Fun g ts*) *of*
None $\Rightarrow$ *None*
| *Some us* $\Rightarrow$ *match-term-list* (*us* @ *P*) σ) |
match-term-list ((*Fun f ss*, *Var x*) # *P*) σ = *None*
by (*pat-completeness*) *auto*
termination
by (*standard*, *rule wf-inv-image* [*OF wf-measure* [*of size-mset*], *of mset* $\circ$ *fst*])
(*auto simp*: *pair-size-def*)

lemma *match-term-list-Some-matchrel*:
assumes *match-term-list P* σ = *Some* τ
shows ((*mset P*, σ), ({#}, τ)) $\in$ *matchrel**
using *assms*
proof (*induction P* σ *rule*: *match-term-list.induct*)
case (*2 x t P* σ)
from *2.prems*
have $*$: σ *x* = *None* $\vee$ σ *x* = *Some t*
and $**$: *match-term-list P* (σ (*x* $\mapsto$ *t*)) = *Some* τ **by** (*auto split*: *if-splits*)
from *MATCH1.Var* [*of* σ *x t mset P*, *OF* $*$]
have ((*mset* ((*Var x*, *t*) # *P*), σ), (*mset P*, σ (*x* $\mapsto$ *t*))) $\in$ *matchrel**
by (*simp add*: *MATCH1-matchrel-conv*)
with *2.IH* [*OF* $*$ $**$] **show** *?case* **by** (*blast dest*: *rtrancl-trans*)
next
case (*3 f ss g ts P* σ)
let *?s* = *Fun f ss* **and** *?t* = *Fun g ts*
from *3.prems* **have** [*simp*]: *f* = *g*
and $*$: *length ss* = *length ts*
and $**$: *decompose ?s ?t* = *Some* (*zip ss ts*)
match-term-list (*zip ss ts* @ *P*) σ = *Some* τ
by (*auto split*: *option.splits*)
from *MATCH1.Fun* [*OF* $*$, *of mset P g* σ]
have ((*mset* ((*?s*, *?t*) # *P*), σ), (*mset* (*zip ss ts* @ *P*), σ)) $\in$ *matchrel**
by (*simp add*: *MATCH1-matchrel-conv ac-simps*)
with *3.IH* [*OF* $**$] **show** *?case* **by** (*blast dest*: *rtrancl-trans*)
qed *simp-all*

lemma *match-term-list-None*:
assumes *match-term-list P* σ = *None*
shows *matchers-map* σ $\cap$ *matchers* (*set P*) = {}
using *assms*
proof (*induction P* σ *rule*: *match-term-list.induct*)
case (*2 x t P* σ)
have $\neg$ (σ *x* = *None* $\vee$ σ *x* = *Some t*) $\vee$

(σ x = None ∨ σ x = Some t) ∧ match-term-list P (σ (x ↦ t)) = None
using 2.prems **by** (auto split: if-splits)
then show ?case
proof
assume ∗: ¬ (σ x = None ∨ σ x = Some t)
have ¬ (∃ y. (({#(Var x, t)#}, σ), y) ∈ matchrel)
proof
presume ¬ ?thesis
then obtain y **where** MATCH1 ({#(Var x, t)#}, σ) y
by (auto simp: MATCH1-matchrel-conv)
then show False **using** ∗ **by** (cases) simp-all
qed simp
moreover have (({#(Var x, t)#}, σ), ({#(Var x, t)#}, σ)) ∈ matchrel* **by** simp
ultimately have (({#(Var x, t)#}, σ), ({#(Var x, t)#}, σ)) ∈ matchrel$^!$
by (metis NF-I normalizability-I)
from irreducible-reachable-imp-matchers-empty [OF this]
have matchers-map σ ∩ matchers {(Var x, t)} = {} **by** simp
then show ?case **by** auto
next
presume ∗: σ x = None ∨ σ x = Some t
and match-term-list P (σ (x ↦ t)) = None
from 2.IH [OF this] **have** matchers-map (σ (x ↦ t)) ∩ matchers (set P) = {} .
with MATCH1-matchers [OF MATCH1.Var [of σ x, OF ∗], of mset P]
show ?case **by** simp
qed auto
next
case (3 f ss g ts P σ)
let ?s = Fun f ss **and** ?t = Fun g ts
have decompose ?s ?t = None ∨
decompose ?s ?t = Some (zip ss ts) ∧ match-term-list (zip ss ts @ P) σ = None
using 3.prems **by** (auto split: option.splits)
then show ?case
proof
assume decompose ?s ?t = None
then show ?case **by** auto
next
presume decompose ?s ?t = Some (zip ss ts)
and match-term-list (zip ss ts @ P) σ = None
from 3.IH [OF this] **show** ?case **by** auto
qed auto
qed simp-all

Compute a matching substitution for a list of term pairs P, where left-hand sides are "patterns" against which the right-hand sides are matched.

definition match-list ::
('v ⇒ ('f, 'w) term) ⇒ (('f, 'v) term × ('f, 'w) term) list ⇒ ('f, 'v, 'w) gsubst option

where
match-list d P = *map-option* (*subst-of-map d*) (*match-term-list P Map.empty*)

lemma *match-list-sound*:
assumes *match-list d P* = *Some* σ
shows $\sigma \in$ *matchers* (*set P*)
using *matchrel-sound* [*of mset P*]
and *match-term-list-Some-matchrel* [*of P Map.empty*]
and *assms* **by** (*auto simp*: *match-list-def*)

lemma *match-list-matches*:
assumes *match-list d P* = *Some* σ
shows $\bigwedge p\ t.\ (p,\ t) \in set\ P \Longrightarrow p \cdot \sigma = t$
using *match-list-sound* [*OF assms*] **by** (*force simp*: *matchers-def*)

lemma *match-list-complete*:
assumes *match-list d P* = *None*
shows *matchers* (*set P*) = {}
using *match-term-list-None* [*of P Map.empty*] **and** *assms* **by** (*simp add*: *match-list-def*)

lemma *match-list-None-conv*:
match-list d P = *None* $\longleftrightarrow$ *matchers* (*set P*) = {}
using *match-list-sound* [*of d P*] **and** *match-list-complete* [*of d P*]
by (*metis empty-iff not-None-eq*)

definition *match t l* = *match-list Var* [(*l*, *t*)]

lemma *match-sound*:
assumes *match t p* = *Some* σ
shows $\sigma \in$ *matchers* {(*p*, *t*)}
using *match-list-sound* [*of Var* [(*p*, *t*)]] **and** *assms* **by** (*simp add*: *match-def*)

lemma *match-matches*:
assumes *match t p* = *Some* σ
shows $p \cdot \sigma = t$
using *match-sound* [*OF assms*] **by** (*force simp*: *matchers-def*)

lemma *match-complete*:
assumes *match t p* = *None*
shows *matchers* {(*p*, *t*)} = {}
using *match-list-complete* [*of Var* [(*p*, *t*)]] **and** *assms* **by** (*simp add*: *match-def*)

definition *matches* :: ($'f$, $'w$) *term* $\Rightarrow$ ($'f$, $'v$) *term* $\Rightarrow$ *bool*
where
matches t p = (*case match-list* (λ -. *t*) [(*p*,*t*)] *of None* $\Rightarrow$ *False* | *Some* - $\Rightarrow$ *True*)

lemma *matches-iff*:
matches t p $\longleftrightarrow$ ($\exists\,\sigma.\ p \cdot \sigma = t$)
using *match-list-sound* [*of* - [(*p*,*t*)]]

and *match-list-complete* [*of* - [(*p*,*t*)]]
unfolding *matches-def matchers-def*
by (*force simp*: *split*: *option.splits*)

lemma *match-complete'*:
assumes $p \cdot \sigma = t$
shows $\exists \tau.$ *match t p* $=$ *Some* $\tau \wedge (\forall x{\in}$*vars-term p*. $\sigma\ x = \tau\ x)$
proof −
from *assms* **have** σ: $\sigma \in$ *matchers* $\{(p,t)\}$ **by** (*simp add*: *matchers-def*)
with *match-complete*[*of t p*]
obtain τ **where** *match*: *match t p* $=$ *Some* τ **by** (*auto split*: *option.splits*)
from *match-sound*[*OF this*]
have $\tau \in$ *matchers* $\{(p, t)\}$ **.**
from *matchers-vars-term-eq*[*OF* σ *this*] *match* **show** *?thesis* **by** *auto*
qed

abbreviation *lvars* :: (('f, 'v) *term* × ('f, 'w) *term*) *list* ⇒ 'v *set*
where
lvars P $\equiv \bigcup$ ((*vars-term* ∘ *fst*) ‘ *set P*)

lemma *match-list-complete'*:
assumes $\bigwedge s\ t.\ (s,\ t) \in$ *set P* $\Longrightarrow s \cdot \sigma = t$
shows $\exists \tau.$ *match-list d P* $=$ *Some* $\tau \wedge (\forall x{\in}$*lvars P*. $\sigma\ x = \tau\ x)$
proof −
from *assms* **have** $\sigma \in$ *matchers* (*set P*) **by** (*force simp*: *matchers-def*)
moreover with *match-list-complete* [*of d P*] **obtain** τ
where *match-list d P* $=$ *Some* τ **by** *auto*
moreover with *match-list-sound* [*of d P*]
have $\tau \in$ *matchers* (*set P*)
by (*auto simp*: *match-def split*: *option.splits*)
ultimately show *?thesis*
using *matchers-vars-term-eq* [*of* σ *set P* τ] **by** *auto*
qed

end

6.1 A variable disjoint unification algorithm for terms with string variables

theory *Unification-String*
imports
Unification
Renaming2-String
begin
definition *mgu-vd-string* = *mgu-vd string-rename*

lemma *mgu-vd-string-code*[*code*]: *mgu-vd-string* = *mgu-var-disjoint-generic* (*Cons* (*CHR ''x''*)) (*Cons* (*CHR ''y''*))
unfolding *mgu-vd-string-def mgu-vd-def*

by (*transfer*, *auto*)

lemma *mgu-vd-string-sound*:
mgu-vd-string s t = *Some* ($\gamma 1$, $\gamma 2$) $\Longrightarrow$ $s \cdot \gamma 1 = t \cdot \gamma 2$
unfolding *mgu-vd-string-def* **by** (*rule mgu-vd-sound*)

lemma *mgu-vd-string-complete*:
fixes σ :: ('*f*, *string*) *subst* **and** τ :: ('*f*, *string*) *subst*
assumes $s \cdot \sigma = t \cdot \tau$
shows $\exists \mu 1\ \mu 2\ \delta$. *mgu-vd-string s t* = *Some* ($\mu 1$, $\mu 2$) $\wedge$
$\sigma = \mu 1 \circ_s \delta \wedge$
$\tau = \mu 2 \circ_s \delta \wedge$
$s \cdot \mu 1 = t \cdot \mu 2$
unfolding *mgu-vd-string-def*
by (*rule mgu-vd-complete*[*OF assms*])
end

7 Subsumption

We define the subsumption relation on terms and prove its well-foundedness.

theory *Subsumption*
imports
Term
Abstract−Rewriting.Seq
HOL−Library.Adhoc-Overloading
Fun-More
Seq-More
begin

consts
SUBSUMESEQ :: '*a* ⇒ '*a* ⇒ *bool* (**infix** ≤· *50*)
SUBSUMES :: '*a* ⇒ '*a* ⇒ *bool* (**infix** <· *50*)
LITSIM :: '*a* ⇒ '*a* ⇒ *bool* (**infix** ≐ *50*)

abbreviation (*input*) *INSTANCEQ* (**infix** ·≥ *50*)
where
x ·≥ *y* ≡ *y* ≤· *x*

abbreviation (*input*) *INSTANCE* (**infix** ·> *50*)
where
x ·> *y* ≡ *y* <· *x*

abbreviation *INSTANCEEQ-SET* ({·≥})
where
{·≥} ≡ {(*x*, *y*). *y* ≤· *x*}

abbreviation *INSTANCE-SET* ({·>})
where

$\{\cdot\!>\} \equiv \{(x,\ y).\ y <\cdot\ x\}$

abbreviation *SUBSUMESEQ-SET* ($\{\leq\cdot\}$)
where
$\{\leq\cdot\} \equiv \{(x,\ y).\ x \leq\cdot\ y\}$

abbreviation *SUBSUMES-SET* ($\{<\cdot\}$)
where
$\{<\cdot\} \equiv \{(x,\ y).\ x <\cdot\ y\}$

abbreviation *LITSIM-SET* ($\{\doteq\}$)
where
$\{\doteq\} \equiv \{(x,\ y).\ x \doteq y\}$

locale *subsumable* =
fixes *subsumeseq* :: $'a \Rightarrow\ 'a \Rightarrow$ *bool*
assumes *refl*: *subsumeseq x x*
and *trans*: *subsumeseq x y* $\Longrightarrow$ *subsumeseq y z* $\Longrightarrow$ *subsumeseq x z*
begin

adhoc-overloading
SUBSUMESEQ subsumeseq

definition *subsumes t s* $\longleftrightarrow t \leq\cdot\ s \wedge \neg\ s \leq\cdot\ t$

definition *litsim s t* $\longleftrightarrow s \leq\cdot\ t \wedge t \leq\cdot\ s$

adhoc-overloading
SUBSUMES subsumes **and**
LITSIM litsim

lemma *litsim-refl* [*simp*]:
$s \doteq s$
by (*auto simp*: *litsim-def refl*)

lemma *litsim-sym*:
$s \doteq t \Longrightarrow t \doteq s$
by (*auto simp*: *litsim-def*)

lemma *litsim-trans*:
$s \doteq t \Longrightarrow t \doteq u \Longrightarrow s \doteq u$
by (*auto simp*: *litsim-def dest*: *trans*)

end

sublocale *subsumable* $\subseteq$ *subsumption*: *preorder* ($\leq\cdot$) ($<\cdot$)
by (*unfold-locales*) (*auto simp*: *subsumes-def refl elim*: *trans*)

inductive *subsumeseq-term* :: $('a,\ 'b)$ *term* $\Rightarrow$ $('a,\ 'b)$ *term* $\Rightarrow$ *bool*

where
  $[intro]$: $t = s \cdot \sigma \Longrightarrow$ *subsumeseq-term s t*

adhoc-overloading
  SUBSUMESEQ subsumeseq-term

lemma *subsumeseq-termE* [*elim*]:
  assumes $s \le\cdot t$
  obtains σ **where** $t = s \cdot \sigma$
  using *assms* **by** (*cases*)

lemma *subsumeseq-term-refl*:
  fixes t :: ('a, 'b) *term*
  shows $t \le\cdot t$
  by (*rule subsumeseq-term.intros* [*of t t Var*]) *simp*

lemma *subsumeseq-term-trans*:
  fixes *s t u* :: ('a, 'b) *term*
  assumes $s \le\cdot t$ **and** $t \le\cdot u$
  shows $s \le\cdot u$
proof −
  obtain σ τ
    where [*simp*]: $t = s \cdot \sigma$ $u = t \cdot \tau$ **using** *assms* **by** *fastforce*
  show *?thesis*
    by (*rule subsumeseq-term.intros* [*of - -* $\sigma \circ_s \tau$]) *simp*
qed

interpretation *term-subsumable*: *subsumable subsumeseq-term*
  by *standard* (*force simp*: *subsumeseq-term-refl dest*: *subsumeseq-term-trans*)+

adhoc-overloading
  SUBSUMES term-subsumable.subsumes **and**
  LITSIM term-subsumable.litsim

lemma *subsumeseq-term-iff*:
  $s \cdot\ge t \longleftrightarrow (\exists \sigma.\ s = t \cdot \sigma)$
  by *auto*

fun *num-syms* :: ('f, 'v) *term* ⇒ *nat*
  where
    num-syms (*Var x*) = *1* |
    num-syms (*Fun f ts*) = *Suc* (*sum-list* (*map num-syms ts*))

fun *num-vars* :: ('f, 'v) *term* ⇒ *nat*
  where
    num-vars (*Var x*) = *1* |
    num-vars (*Fun f ts*) = *sum-list* (*map num-vars ts*)

definition *num-unique-vars* :: ('f, 'v) *term* ⇒ *nat*

where
num-unique-vars t = *card* (*vars-term* t)

lemma *num-syms-1*: *num-syms* $t \geq 1$
by (*induct* t) *auto*

lemma *num-syms-subst*:
num-syms $(t \cdot \sigma) \geq$ *num-syms* t
using *num-syms-1*
by (*induct* t) (*auto*, *metis comp-apply sum-list-mono*)

7.1 Equality of terms modulo variables

inductive *emv* **where**
Var [*simp*, *intro*!]: *emv* (*Var* x) (*Var* y) |
Fun [*intro*]: $[\![f = g;$ *length* ss = *length* ts; $\forall i <$ *length* ts. *emv* $(ss\ !\ i)$ $(ts\ !\ i)]\!]$ $\Longrightarrow$
emv (*Fun* f ss) (*Fun* g ts)

lemma *sum-list-map-num-syms-subst*:
assumes *sum-list* (*map* (*num-syms* $\circ$ $(\lambda t.\ t \cdot \sigma)$) ts) = *sum-list* (*map num-syms* ts)
shows $\forall i <$ *length* ts. *num-syms* $(ts\ !\ i \cdot \sigma)$ = *num-syms* $(ts\ !\ i)$
using *assms*
proof (*induct* ts)
case (*Cons* t ts)
then have *num-syms* $(t \cdot \sigma)$ + *sum-list* (*map* (*num-syms* $\circ$ $(\lambda t.\ t \cdot \sigma)$) ts)
= *num-syms* t + *sum-list* (*map num-syms* ts) **by** (*simp add*: *o-def*)
moreover have *num-syms* $(t \cdot \sigma) \geq$ *num-syms* t **by** (*metis num-syms-subst*)
moreover have *sum-list* (*map* (*num-syms* $\circ$ $(\lambda t.\ t \cdot \sigma)$) ts) $\geq$ *sum-list* (*map num-syms* ts)
using *num-syms-subst* [*of* - σ] **by** (*induct* ts) (*auto intro*: *add-mono*)
ultimately show *?case* **using** *Cons* **by** (*auto*) (*case-tac* i, *auto*)
qed *simp*

lemma *subst-size-emv*:
assumes $s = t \cdot \tau$ **and** *num-syms* s = *num-syms* t **and** *num-funs* s = *num-funs* t
shows *emv* s t
using *assms*
proof (*induct* t *arbitrary*: s)
case (*Var* x)
then show *?case* **by** (*force elim*: *num-funs-0*)
next
case (*Fun* g ts)
note *IH* = *this*
show *?case*
proof (*cases* s)
case (*Var* x)

then show *?thesis* **using** *Fun* **by** *simp*
next
case (*Fun f ss*)
from *IH*(*2*−) [*unfolded Fun*]
and *sum-list-map-num-syms-subst* [*of* τ *ts*]
and *sum-list-map-num-funs-subst* [*of* τ *ts*]
have $\forall$ *i* < *length ts*. *num-syms* (*ts* ! *i* · τ) = *num-syms* (*ts* ! *i*)
and $\forall$ *i* < *length ts*. *num-funs* (*ts* ! *i* · τ) = *num-funs* (*ts* ! *i*)
by *auto*
with *Fun* **and** *IH* **show** *?thesis* **by** *auto*
qed
qed

lemma *subsumeseq-term-size-emv*:
assumes *s* ·≥ *t* **and** *num-syms s* = *num-syms t* **and** *num-funs s* = *num-funs t*
shows *emv s t*
using *assms*(*1*) **and** *subst-size-emv* [*OF* - *assms*(*2*−)] **by** (*cases*) *simp*

lemma *emv-subst-vars-term*:
assumes *emv s t*
and *s* = *t* · σ
shows *vars-term s* = (*the-Var* ∘ σ) ‘ *vars-term t*
using *assms* [*unfolded subsumeseq-term-iff*]
apply (*induct*)
apply (*auto simp*: *in-set-conv-nth iff*: *image-iff*)
apply (*metis nth-mem*)
by (*metis comp-apply imageI nth-mem*)

lemma *emv-subst-imp-num-unique-vars-le*:
assumes *emv s t*
and *s* = *t* · σ
shows *num-unique-vars s* ≤ *num-unique-vars t*
using *emv-subst-vars-term* [*OF assms*]
apply (*simp add*: *num-unique-vars-def*)
by (*metis card-image-le finite-vars-term*)

lemma *emv-subsumeseq-term-imp-num-unique-vars-le*:
assumes *emv s t*
and *s* ·≥ *t*
shows *num-unique-vars s* ≤ *num-unique-vars t*
using *assms*(*2*) **and** *emv-subst-imp-num-unique-vars-le* [*OF assms*(*1*)] **by** (*cases*) *simp*

lemma *num-syms-geq-num-vars*:
num-syms t ≥ *num-vars t*
proof (*induct t*)
case (*Fun f ts*)
with *sum-list-mono* [*of ts num-vars num-syms*]
have *sum-list* (*map num-vars ts*) ≤ *sum-list* (*map num-syms ts*) **by** *simp*

then show *?case* **by** *simp*
qed *simp*

lemma *num-unique-vars-Fun-Cons*:
num-unique-vars (*Fun f* (*t* # *ts*)) ≤ *num-unique-vars t* + *num-unique-vars* (*Fun f ts*)
apply (*simp-all add*: *num-unique-vars-def*)
unfolding *card-Un-Int* [*OF finite-vars-term finite-Union-vars-term*]
apply *simp*
done

lemma *sum-list-map-unique-vars*:
sum-list (*map num-unique-vars ts*) ≥ *num-unique-vars* (*Fun f ts*)
proof (*induct ts*)
case (*Cons t ts*)
with *num-unique-vars-Fun-Cons* [*of f t ts*]
show *?case* **by** *simp*
qed (*simp add*: *num-unique-vars-def*)

lemma *num-unique-vars-Var-1* [*simp*]:
num-unique-vars (*Var x*) = *1*
by (*simp-all add*: *num-unique-vars-def*)

lemma *num-vars-geq-num-unique-vars*:
num-vars t ≥ *num-unique-vars t*
proof −
note ∗ =
sum-list-mono [*of - num-unique-vars num-vars*, *THEN sum-list-map-unique-vars* [*THEN le-trans*]]
show *?thesis* **by** (*induct t*) (*auto intro*: ∗)
qed

lemma *num-syms-ge-num-unique-vars*:
num-syms t ≥ *num-unique-vars t*
by (*metis le-trans num-syms-geq-num-vars num-vars-geq-num-unique-vars*)

lemma *num-syms-num-unique-vars-clash*:
assumes ∀ *i*. *num-syms* (*f i*) = *num-syms* (*f* (*Suc i*))
and ∀ *i*. *num-unique-vars* (*f i*) < *num-unique-vars* (*f* (*Suc i*))
shows *False*
proof −
have ∗: ∀ *i j*. *i* ≤ *j* ⟶ *num-syms* (*f i*) = *num-syms* (*f j*)
proof (*intro allI impI*)
fix *i j* :: *nat*
assume *i* ≤ *j*
then show *num-syms* (*f i*) = *num-syms* (*f j*)
using *assms*(*1*)
apply (*induct j* − *i arbitrary*: *i*)
apply *auto*

by (*metis Suc-diff-diff diff-zero less-eq-Suc-le order.not-eq-order-implies-strict*)
qed
have $\exists\, i.$ *num-unique-vars* (*f i*) $\geq$ *num-syms* (*f 0*)
using *inc-seq-greater* [*OF assms*(*2*), *of num-syms* (*f 0*)] **by** (*metis nat-less-le*)
then obtain *i* **where** *num-unique-vars* (*f i*) $\geq$ *num-syms* (*f 0*) **by** *auto*
with $*$ **and** *assms*(*2*) **have** *num-unique-vars* (*f* (*Suc i*)) $>$ *num-syms* (*f* (*Suc i*))
by (*metis le0 le-antisym num-syms-ge-num-unique-vars*)
then show *False*
by (*metis less-Suc-eq-le not-less-eq num-syms-ge-num-unique-vars*)
qed

lemma *emv-subst-imp-is-Var*:
assumes *emv s t*
and $s = t \cdot \sigma$
shows $\forall\, x \in$ *vars-term t. is-Var* (σ *x*)
using *assms*
apply (*induct*)
apply *auto*
by (*metis in-set-conv-nth*)

lemma *bij-Var-subst-compose-Var*:
assumes *bij g*
shows (*Var* $\circ$ *g*) $\circ_s$ (*Var* $\circ$ *inv g*) = *Var*
proof
fix *x*
show ((*Var* $\circ$ *g*) $\circ_s$ (*Var* $\circ$ *inv g*)) *x* = *Var x*
using *assms*
apply (*auto simp*: *subst-compose-def*)
by (*metis UNIV-I bij-is-inj inv-into-f-f*)
qed

7.2 Well-foundedness

lemma *wf-subsumes*:
wf ({<·} :: ('*f*, '*v*) *term rel*)
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *f* :: ('*f*, '*v*) *term seq*
where *strict*: $\forall\, i.$ *f i* ·> *f* (*Suc i*)
by (*metis mem-Collect-eq case-prodD wf-iff-no-infinite-down-chain*)
then have $*$: $\forall\, i.$ *f i* ·$\geq$ *f* (*Suc i*) **by** (*metis term-subsumable.subsumption.less-imp-le*)
then have $\forall\, i.$ *num-syms* (*f i*) $\geq$ *num-syms* (*f* (*Suc i*))
by (*auto simp*: *subsumeseq-term-iff*) (*metis num-syms-subst*)
from *down-chain-imp-eq* [*OF this*] **obtain** *N*
where *N-syms*: $\forall\, i > N.$ *num-syms* (*f i*) = *num-syms* (*f* (*Suc i*)) **..**
define *g* **where** *g i* = *f* (*i* + *N*) **for** *i*
from $*$ **have** $\forall\, i.$ *num-funs* (*g i*) $\geq$ *num-funs* (*g* (*Suc i*))
by (*auto simp*: *subsumeseq-term-iff g-def*) (*metis num-funs-subst*)

from *down-chain-imp-eq* [*OF this*] **obtain** *K*
  where *K-funs*: ∀ *i* > *K*. *num-funs* (*g i*) = *num-funs* (*g* (*Suc i*)) **..**
define *M* **where** *M* = *max K N*
have *strict-g*: ∀ *i* > *M*. *g i* ·> *g* (*Suc i*) **using** *strict* **by** (*simp add*: *g-def M-def*)
have *g*: ∀ *i* > *M*. *g i* ·≥ *g* (*Suc i*) **using** ∗ **by** (*simp add*: *g-def M-def*)
moreover have ∀ *i* > *M*. *num-funs* (*g i*) = *num-funs* (*g* (*Suc i*))
  using *K-funs* **unfolding** *M-def* **by** (*metis max-less-iff-conj*)
moreover have *syms*: ∀ *i* > *M*. *num-syms* (*g i*) = *num-syms* (*g* (*Suc i*))
  using *N-syms* **unfolding** *M-def g-def*
  by (*metis add-Suc-right add-lessD1 add-strict-left-mono add.commute*)
ultimately have *emv*: ∀ *i* > *M*. *emv* (*g i*) (*g* (*Suc i*)) **by** (*metis subsume-seq-term-size-emv*)
then have ∀ *i* > *M*. *num-unique-vars* (*g* (*Suc i*)) ≥ *num-unique-vars* (*g i*)
  using *emv-subsumeseq-term-imp-num-unique-vars-le* **and** *g* **by** *fast*
then obtain *i* **where** *i* > *M*
  and *nuv*: *num-unique-vars* (*g* (*Suc i*)) = *num-unique-vars* (*g i*)
  using *num-syms-num-unique-vars-clash* [*of* λ*i*. *g* (*i* + *Suc M*)] **and** *syms*
  by (*metis add-Suc-right add-Suc-shift le-eq-less-or-eq less-add-Suc2*)
define *s* **and** *t* **where** *s* = *g i* **and** *t* = *g* (*Suc i*)
from *nuv* **have** *card*: *card* (*vars-term s*) = *card* (*vars-term t*)
  by (*simp add*: *num-unique-vars-def s-def t-def*)
from *g* [*THEN spec*, *THEN mp*, *OF* ‹*i* > *M*›] **obtain** σ
  where *s* = *t* · σ **by** (*cases*) (*auto simp*: *s-def t-def*)
then have *emv s t* **and** *vars-term s* = (*the-Var* ∘ σ) ‘ *vars-term t*
  using *emv-subst-vars-term* [*of s t* σ] **and** *emv* **and** ‹*i* > *M*› **by** (*auto simp*: *s-def t-def*)
with *card* **have** *card* ((*the-Var* ∘ σ) ‘ *vars-term t*) = *card* (*vars-term t*) **by** *simp*
from *finite-card-eq-imp-bij-betw* [*OF finite-vars-term this*]
have *bij-betw* (*the-Var* ∘ σ) (*vars-term t*) ((*the-Var* ∘ σ) ‘ *vars-term t*) **.**

from *bij-betw-extend* [*OF this*, *of UNIV*]
obtain *h* **where** ∗: ∀ *x*∈*vars-term t*. *h x* = (*the-Var* ∘ σ) *x*
  and *finite* {*x*. *h x* ≠ *x*}
  and *bij h* **by** *auto*
have ∀ *x*∈*vars-term t*. (*Var* ∘ *h*) *x* = σ *x*
proof
  fix *x*
  assume *x* ∈ *vars-term t*
  with ∗ **have** *h x* = (*the-Var* ∘ σ) *x* **by** *simp*
  with *emv-subst-imp-is-Var* [*OF* ‹*emv s t*› ‹*s* = *t* · σ›] ‹*x* ∈ *vars-term t*›
  show (*Var* ∘ *h*) *x* = σ *x* **by** *simp*
qed
then have *t* · (*Var* ∘ *h*) = *s*
  using ‹*s* = *t* · σ› **by** (*auto simp*: *term-subst-eq-conv*)
then have *t* · (*Var* ∘ *h*) $\circ_s$ (*Var* ∘ *inv h*) = *s* · (*Var* ∘ *inv h*) **by** *auto*
then have *t* = *s* · (*Var* ∘ *inv h*)
  unfolding *bij-Var-subst-compose-Var* [*OF* ‹*bij h*›] **by** *simp*
then have *t* ·≥ *s* **by** *auto*
with *strict-g* **and** ‹*i* > *M*› **show** *False* **by** (*auto simp*: *s-def t-def term-subsumable.subsumes-def*)

qed

end

8 Subterms and Contexts

We define the (proper) sub- and superterm relations on first order terms, as well as contexts (you can think of contexts as terms with exactly one hole, where we can plug-in another term). Moreover, we establish several connections between these concepts as well as to other concepts such as substitutions.

theory *Subterm-and-Context*
  imports
    Term
    Abstract−Rewriting.Abstract-Rewriting
begin

8.1 Subterms

The *superterm* relation.

inductive-set
  supteq :: $(('f, 'v)\ term \times ('f, 'v)\ term)\ set$
  where
    refl [*simp*, *intro*]: $(t, t) \in supteq$ |
    subt [*intro*]: $u \in set\ ss \Longrightarrow (u, t) \in supteq \Longrightarrow (Fun\ f\ ss, t) \in supteq$

The *proper superterm* relation.

inductive-set
  supt :: $(('f, 'v)\ term \times ('f, 'v)\ term)\ set$
  where
    arg [*simp*, *intro*]: $s \in set\ ss \Longrightarrow (Fun\ f\ ss, s) \in supt$ |
    subt [*intro*]: $s \in set\ ss \Longrightarrow (s, t) \in supt \Longrightarrow (Fun\ f\ ss, t) \in supt$

hide-const *suptp supteqp*
hide-fact
  suptp.arg suptp.cases suptp.induct suptp.intros suptp.subt suptp-supt-eq
hide-fact
  supteqp.cases supteqp.induct supteqp.intros supteqp.refl supteqp.subt supteqp-supteq-eq

hide-fact (**open**) *supt.arg supt.subt supteq.refl supteq.subt*

8.1.1 Syntactic Sugar

Infix syntax.

abbreviation *supt-pred* $s\ t \equiv (s, t) \in supt$
abbreviation *supteq-pred* $s\ t \equiv (s, t) \in supteq$

abbreviation (*input*) *subt-pred s t* ≡ *supt-pred t s*
abbreviation (*input*) *subteq-pred s t* ≡ *supteq-pred t s*

notation
supt ({⊳}) **and**
supt-pred ((-/ ⊳ -) [*56*, *56*] *55*) **and**
subt-pred (**infix** ⊲ *55*) **and**
supteq ({⊵}) **and**
supteq-pred ((-/ ⊵ -) [*56*, *56*] *55*) **and**
subteq-pred (**infix** ⊴ *55*)

abbreviation *subt* ({⊲}) **where** {⊲} ≡ $\{⊳\}^{-1}$
abbreviation *subteq* ({⊴}) **where** {⊴} ≡ $\{⊵\}^{-1}$

Quantifier syntax.

syntax
-All-supteq :: [*idt*, *'a*, *bool*] ⇒ *bool* ((*3*∀ -⊵-./ -) [*0*, *0*, *10*] *10*)
-Ex-supteq :: [*idt*, *'a*, *bool*] ⇒ *bool* ((*3*∃ -⊵-./ -) [*0*, *0*, *10*] *10*)
-All-supt :: [*idt*, *'a*, *bool*] ⇒ *bool* ((*3*∀ -⊳-./ -) [*0*, *0*, *10*] *10*)
-Ex-supt :: [*idt*, *'a*, *bool*] ⇒ *bool* ((*3*∃ -⊳-./ -) [*0*, *0*, *10*] *10*)

-All-subteq :: [*idt*, *'a*, *bool*] ⇒ *bool* ((*3*∀ -⊴-./ -) [*0*, *0*, *10*] *10*)
-Ex-subteq :: [*idt*, *'a*, *bool*] ⇒ *bool* ((*3*∃ -⊴-./ -) [*0*, *0*, *10*] *10*)
-All-subt :: [*idt*, *'a*, *bool*] ⇒ *bool* ((*3*∀ -⊲-./ -) [*0*, *0*, *10*] *10*)
-Ex-subt :: [*idt*, *'a*, *bool*] ⇒ *bool* ((*3*∃ -⊲-./ -) [*0*, *0*, *10*] *10*)

translations
$\forall x \trianglerighteq y.\ P \rightharpoonup \forall x.\ x \trianglerighteq y \longrightarrow P$
$\exists x \trianglerighteq y.\ P \rightharpoonup \exists x.\ x \trianglerighteq y \wedge P$
$\forall x \triangleright y.\ P \rightharpoonup \forall x.\ x \triangleright y \longrightarrow P$
$\exists x \triangleright y.\ P \rightharpoonup \exists x.\ x \triangleright y \wedge P$

$\forall x \trianglelefteq y.\ P \rightharpoonup \forall x.\ x \trianglelefteq y \longrightarrow P$
$\exists x \trianglelefteq y.\ P \rightharpoonup \exists x.\ x \trianglelefteq y \wedge P$
$\forall x \triangleleft y.\ P \rightharpoonup \forall x.\ x \triangleleft y \longrightarrow P$
$\exists x \triangleleft y.\ P \rightharpoonup \exists x.\ x \triangleleft y \wedge P$

print-translation ‹
let
val All-binder = *Mixfix.binder-name* @{*const-syntax All*};
val Ex-binder = *Mixfix.binder-name* @{*const-syntax Ex*};
val impl = @{*const-syntax implies*};
val conj = @{*const-syntax conj*};
val supt = @{*const-syntax supt-pred*};
val supteq = @{*const-syntax supteq-pred*};

val trans =

[((*All-binder*, *impl*, *supt*), (*-All-supt*, *-All-subt*)),
((*All-binder*, *impl*, *supteq*), (*-All-supteq*, *-All-subteq*)),
((*Ex-binder*, *conj*, *supt*), (*-Ex-supt*, *-Ex-subt*)),
((*Ex-binder*, *conj*, *supteq*), (*-Ex-supteq*, *-Ex-subteq*))];

fun matches-bound v t =
case t of (*Const* (*-bound*, -) $ *Free* (*v'*, -)) => (*v* = *v'*)
| - => *false*
fun contains-var v = *Term.exists-subterm* (*fn Free* (*x*, -) => *x* = *v* | - => *false*)
fun mk x c n P = *Syntax.const c* $ *Syntax-Trans.mark-bound-body x* $ *n* $ *P*

fun tr' q = (*q*,
K (*fn* [*Const* (*-bound*, -) $ *Free* (*v*, *T*), *Const* (*c*, -) $ (*Const* (*d*, -) $ *t* $ *u*) $ *P*] =>
(*case AList.lookup* (=) *trans* (*q*, *c*, *d*) *of*
NONE => *raise Match*
| *SOME* (*l*, *g*) =>
if matches-bound v t andalso not (*contains-var v u*) *then mk* (*v*, *T*) *l u P*
else if matches-bound v u andalso not (*contains-var v t*) *then mk* (*v*, *T*) *g t P*
else raise Match)
| - => *raise Match*));
in [*tr' All-binder*, *tr' Ex-binder*] *end*
›

8.1.2 Transitivity Reasoning for Subterms

lemma *supt-trans* [*trans*]:
$s \rhd t \Longrightarrow t \rhd u \Longrightarrow s \rhd u$
by (*induct s t rule*: *supt.induct*) *auto*

lemma *trans-supt*: *trans* $\{\rhd\}$ **by** (*auto simp*: *trans-def dest*: *supt-trans*)

lemma *supteq-trans* [*trans*]:
$s \trianglerighteq t \Longrightarrow t \trianglerighteq u \Longrightarrow s \trianglerighteq u$
by (*induct s t rule*: *supteq.induct*) (*auto*)

Auxiliary lemmas about term size.

lemma *size-simp5*:
$s \in set\ ss \Longrightarrow s \rhd t \Longrightarrow size\ t < size\ s \Longrightarrow size\ t < Suc\ (size\text{-}list\ size\ ss)$
by (*induct ss*) *auto*

lemma *size-simp6*:
$s \in set\ ss \Longrightarrow s \trianglerighteq t \Longrightarrow size\ t \leq size\ s \Longrightarrow size\ t \leq Suc\ (size\text{-}list\ size\ ss)$
by (*induct ss*) *auto*

lemma *size-simp1*:
$t \in set\ ts \Longrightarrow size\ t < Suc\ (size\text{-}list\ size\ ts)$

by (*induct ts*) *auto*

lemma *size-simp2*:
t ∈ *set ts* ⟹ *size t* < *Suc* (*Suc* (*size s* + *size-list size ts*))
by (*induct ts*) *auto*

lemma *size-simp3*:
assumes (*x*, *y*) ∈ *set* (*zip xs ys*)
shows *size x* < *Suc* (*size-list size xs*)
using *set-zip-leftD* [*OF assms*] *size-simp1* **by** *auto*

lemma *size-simp4*:
assumes (*x*, *y*) ∈ *set* (*zip xs ys*)
shows *size y* < *Suc* (*size-list size ys*)
using *set-zip-rightD* [*OF assms*] **using** *size-simp1* **by** *auto*

lemmas *size-simps* =
size-simp1 size-simp2 size-simp3 size-simp4 size-simp5 size-simp6

declare *size-simps* [*termination-simp*]

lemma *supt-size*:
s ⊳ *t* ⟹ *size s* > *size t*
by (*induct rule*: *supt.induct*) (*auto simp*: *size-simps*)

lemma *supteq-size*:
s ⊵ *t* ⟹ *size s* ≥ *size t*
by (*induct rule*: *supteq.induct*) (*auto simp*: *size-simps*)

Reflexivity and Asymmetry.

lemma *reflcl-supteq* [*simp*]:
supteq$^=$ = *supteq* **by** *auto*

lemma *trancl-supteq* [*simp*]:
supteq$^+$ = *supteq*
by (*rule trancl-id*) (*auto simp*: *trans-def intro*: *supteq-trans*)

lemma *rtrancl-supteq* [*simp*]:
supteq* = *supteq*
unfolding *trancl-reflcl*[*symmetric*] **by** *auto*

lemma *eq-supteq*: *s* = *t* ⟹ *s* ⊵ *t* **by** *auto*

lemma *supt-neqD*: *s* ⊳ *t* ⟹ *s* ≠ *t* **using** *supt-size* **by** *auto*

lemma *supteq-Var* [*simp*]:
x ∈ *vars-term t* ⟹ *t* ⊵ *Var x*
proof (*induct t*)
case (*Var y*) **then show** *?case* **by** (*cases x* = *y*) *auto*

next
  case (*Fun f ss*) **then show** *?case* **by** (*auto*)
qed

lemmas *vars-term-supteq* = *supteq-Var*

lemma *term-not-arg* [*iff*]:
  Fun f ss $\notin$ *set ss* (**is** *?t* $\notin$ *set ss*)
proof
  assume *?t* $\in$ *set ss*
  then have *?t* $\rhd$ *?t* **by** (*auto*)
  then have *?t* $\neq$ *?t* **by** (*auto dest*: *supt-neqD*)
  then show *False* **by** *simp*
qed

lemma *supt-Fun* [*simp*]:
  assumes *s* $\rhd$ *Fun f ss* (**is** *s* $\rhd$ *?t*) **and** *s* $\in$ *set ss*
  shows *False*
proof –
  from ‹*s* $\in$ *set ss*› **have** *?t* $\rhd$ *s* **by** (*auto*)
  then have *size ?t* > *size s* **by** (*rule supt-size*)
  from ‹*s* $\rhd$ *?t*› **have** *size s* > *size ?t* **by** (*rule supt-size*)
  with ‹*size ?t* > *size s*› **show** *False* **by** *simp*
qed

lemma *supt-supteq-conv*: *s* $\rhd$ *t* = (*s* $\trianglerighteq$ *t* $\wedge$ *s* $\neq$ *t*)
proof
  assume *s* $\rhd$ *t* **then show** *s* $\trianglerighteq$ *t* $\wedge$ *s* $\neq$ *t*
  proof (*induct rule*: *supt.induct*)
    case (*subt s ss t f*)
    have *s* $\trianglerighteq$ *s* **..**
    from ‹*s* $\in$ *set ss*› **have** *Fun f ss* $\trianglerighteq$ *s* **by** (*auto*)
    from ‹*s* $\trianglerighteq$ *t* $\wedge$ *s* $\neq$ *t*› **have** *s* $\trianglerighteq$ *t* **..**
    with ‹*Fun f ss* $\trianglerighteq$ *s*› **have** *first*: *Fun f ss* $\trianglerighteq$ *t* **by** (*rule supteq-trans*)
    from ‹*s* $\in$ *set ss*› **and** ‹*s* $\rhd$ *t*› **have** *Fun f ss* $\rhd$ *t* **..**
    then have *second*: *Fun f ss* $\neq$ *t* **by** (*auto dest*: *supt-neqD*)
    from *first* **and** *second* **show** *Fun f ss* $\trianglerighteq$ *t* $\wedge$ *Fun f ss* $\neq$ *t* **by** *auto*
  qed (*auto simp*: *size-simps*)
next
  assume *s* $\trianglerighteq$ *t* $\wedge$ *s* $\neq$ *t*
  then have *s* $\trianglerighteq$ *t* **and** *s* $\neq$ *t* **by** *auto*
  then show *s* $\rhd$ *t* **by** (*induct*) (*auto*)
qed

lemma *supt-not-sym*: *s* $\rhd$ *t* $\Longrightarrow$ $\neg$ (*t* $\rhd$ *s*)
proof
  assume *s* $\rhd$ *t* **and** *t* $\rhd$ *s* **then have** *s* $\rhd$ *s* **by** (*rule supt-trans*)
  then show *False* **unfolding** *supt-supteq-conv* **by** *simp*
qed

lemma *supt-irrefl*[*iff*]: $\neg\ t \rhd t$
using *supt-not-sym*[*of t t*] **by** *auto*

lemma *irrefl-subt*: *irrefl* $\{\lhd\}$ **by** (*auto simp*: *irrefl-def*)

lemma *supt-imp-supteq*: $s \rhd t \Longrightarrow s \trianglerighteq t$
unfolding *supt-supteq-conv* **by** *auto*

lemma *supt-supteq-not-supteq*: $s \rhd t = (s \trianglerighteq t \wedge \neg\ (t \trianglerighteq s))$
using *supt-not-sym* **unfolding** *supt-supteq-conv* **by** *auto*

lemma *supteq-supt-conv*: $(s \trianglerighteq t) = (s \rhd t \vee s = t)$ **by** (*auto simp*: *supt-supteq-conv*)

lemma *supteq-antisym*:
assumes $s \trianglerighteq t$ **and** $t \trianglerighteq s$ **shows** $s = t$
using *assms* **unfolding** *supteq-supt-conv* **by** (*auto simp*: *supt-not-sym*)

The subterm relation is an order on terms.

interpretation *subterm*: *order* $(\trianglelefteq)$ $(\lhd)$
by (*unfold-locales*)
(*rule supt-supteq-not-supteq*, *auto intro*: *supteq-trans supteq-antisym supt-supteq-not-supteq*)

More transitivity rules.

lemma *supt-supteq-trans*[*trans*]:
$s \rhd t \Longrightarrow t \trianglerighteq u \Longrightarrow s \rhd u$
by (*metis subterm.le-less-trans*)

lemma *supteq-supt-trans*[*trans*]:
$s \trianglerighteq t \Longrightarrow t \rhd u \Longrightarrow s \rhd u$
by (*metis subterm.less-le-trans*)

declare *subterm.le-less-trans*[*trans*]
declare *subterm.less-le-trans*[*trans*]

lemma *suptE* [*elim*]: $s \rhd t \Longrightarrow (s \trianglerighteq t \Longrightarrow P) \Longrightarrow (s \neq t \Longrightarrow P) \Longrightarrow P$
by (*auto simp*: *supt-supteq-conv*)

lemmas *suptI* [*intro*] =
subterm.dual-order.not-eq-order-implies-strict

lemma *supt-supteq-set-conv*:
$\{\rhd\} = \{\trianglerighteq\} - Id$
using *supt-supteq-conv* [*to-set*] **by** *auto*

lemma *supteq-supt-set-conv*:
$\{\trianglerighteq\} = \{\rhd\}^=$
by (*auto simp*: *supt-supteq-conv*)

lemma *supteq-imp-vars-term-subset*:
  $s \trianglerighteq t \Longrightarrow$ *vars-term* $t \subseteq$ *vars-term* s
  by (*induct rule*: *supteq.induct*) *auto*

lemma *set-supteq-into-supt* [*simp*]:
  assumes $t \in$ *set ts* **and** $t \trianglerighteq s$
  shows *Fun f ts* $\triangleright s$
proof −
  from ‹$t \trianglerighteq s$› **have** $t = s \vee t \triangleright s$ **by** *auto*
  then show *?thesis*
  proof
    assume $t = s$
    with ‹$t \in$ *set ts*› **show** *?thesis* **by** *auto*
  next
    assume $t \triangleright s$
    from *supt.subt*[*OF* ‹$t \in$ *set ts*› *this*] **show** *?thesis* **.**
  qed
qed

The superterm relation is strongly normalizing.

lemma *SN-supt*:
  SN $\{\triangleright\}$
  unfolding *SN-iff-wf* **by** (*rule wf-subset*) (*auto intro*: *supt-size*)

lemma *supt-not-refl*[*elim!*]:
  assumes $t \triangleright t$ **shows** *False*
proof −
  from *assms* **have** $t \neq t$ **by** *auto*
  then show *False* **by** *simp*
qed

lemma *supteq-not-supt* [*elim*]:
  assumes $s \trianglerighteq t$ **and** $\neg (s \triangleright t)$
  shows $s = t$
  using *assms* **by** (*induct*) *auto*

lemma *supteq-not-supt-conv* [*simp*]:
  $\{\trianglerighteq\} - \{\triangleright\} = $ *Id* **by** *auto*

lemma *supteq-subst* [*simp*, *intro*]:
  assumes $s \trianglerighteq t$ **shows** $s \cdot \sigma \trianglerighteq t \cdot \sigma$
  using *assms*
proof (*induct rule*: *supteq.induct*)
  case (*subt u ss t f*)
  from ‹$u \in$ *set ss*› **have** $u \cdot \sigma \in$ *set* (*map* ($\lambda t.\ t \cdot \sigma$) *ss*) $u \cdot \sigma \trianglerighteq u \cdot \sigma$ **by** *auto*
  then have *Fun f ss* $\cdot\, \sigma \trianglerighteq u \cdot \sigma$ **unfolding** *eval-term.simps* **by** *blast*
  from *this* **and** ‹$u \cdot \sigma \trianglerighteq t \cdot \sigma$› **show** *?case* **by** (*rule supteq-trans*)
qed *auto*

lemma *supt-subst* [*simp*, *intro*]:
  assumes $s \rhd t$ **shows** $s \cdot \sigma \rhd t \cdot \sigma$
  using *assms*
proof (*induct rule*: *supt.induct*)
  case (*arg s ss f*)
  then have $s \cdot \sigma \in set\ (map\ (\lambda t.\ t \cdot \sigma)\ ss)$ **by** *simp*
  then show *?case* **unfolding** *eval-term.simps* **by** (*rule supt.arg*)
next
  case (*subt u ss t f*)
  from ‹$u \in set\ ss$› **have** $u \cdot \sigma \in set\ (map\ (\lambda t.\ t \cdot \sigma)\ ss)$ **by** *simp*
  then have $Fun\ f\ ss \cdot \sigma \rhd u \cdot \sigma$ **unfolding** *eval-term.simps* **by** (*rule supt.arg*)
  with ‹$u \cdot \sigma \rhd t \cdot \sigma$› **show** *?case* **by** (*metis supt-trans*)
qed

lemma *subterm-induct*:
  assumes $\bigwedge t.\ \forall s \lhd t.\ P\ s \Longrightarrow P\ t$
  shows [*case-names subterm*]: $P\ t$
  by (*rule wf-induct*[*OF wf-measure*[*of size*], *of P t*], *rule assms*, *insert supt-size*, *auto*)

8.2 Contexts

A *context* is a term containing exactly one *hole*.

datatype (*funs-ctxt*: $'f$, *vars-ctxt*: $'v$) *ctxt* =
  Hole (□) |
  More $'f$ ($'f$, $'v$) *term list* ($'f$, $'v$) *ctxt* ($'f$, $'v$) *term list*

We also say that we apply a context C to a term t when we replace the hole in a C by t.

fun *ctxt-apply-term* :: ($'f$, $'v$) *ctxt* ⇒ ($'f$, $'v$) *term* ⇒ ($'f$, $'v$) *term* (-⟨-⟩ [*1000*, *0*] *1000*)
  where
    $\Box\langle s\rangle = s$ |
    $(More\ f\ ss1\ C\ ss2)\langle s\rangle = Fun\ f\ (ss1\ @\ C\langle s\rangle\ \#\ ss2)$

lemma *ctxt-eq* [*simp*]:
  $(C\langle s\rangle = C\langle t\rangle) = (s = t)$ **by** (*induct C*) *auto*

fun *ctxt-compose* :: ($'f$, $'v$) *ctxt* ⇒ ($'f$, $'v$) *ctxt* ⇒ ($'f$, $'v$) *ctxt* (**infixl** $\circ_c$ *75*)
  where
    $\Box \circ_c D = D$ |
    $(More\ f\ ss1\ C\ ss2) \circ_c D = More\ f\ ss1\ (C \circ_c D)\ ss2$

interpretation *ctxt-monoid-mult*: *monoid-mult* □ ($\circ_c$)
proof
  fix *C D E* :: ($'f$, $'v$) *ctxt*
  show $C \circ_c D \circ_c E = C \circ_c (D \circ_c E)$ **by** (*induct C*) *simp-all*

show $\Box \circ_c C = C$ **by** *simp*
show $C \circ_c \Box = C$ **by** (*induct* C) *simp-all*
qed

instantiation *ctxt* :: (*type*, *type*) *monoid-mult*
begin
definition [*simp*]: $1 = \Box$
definition [*simp*]: $(*) = (\circ_c)$
instance by (*intro-classes*) (*simp-all add*: *ac-simps*)
end

lemma *ctxt-ctxt-compose* [*simp*]: $(C \circ_c D)\langle t\rangle = C\langle D\langle t\rangle\rangle$ **by** (*induct* C) *simp-all*

lemmas *ctxt-ctxt* = *ctxt-ctxt-compose* [*symmetric*]

Applying substitutions to contexts.

fun *subst-apply-ctxt* :: $('f, 'v)$ *ctxt* $\Rightarrow$ $('f, 'v, 'w)$ *gsubst* $\Rightarrow$ $('f, 'w)$ *ctxt* (**infixl** $\cdot_c$ *67*)
where
$\Box \cdot_c \sigma = \Box$ |
(*More f ss1 D ss2*) $\cdot_c \sigma$ = *More f* (*map* ($\lambda t.\ t \cdot \sigma$) *ss1*) $(D \cdot_c \sigma)$ (*map* ($\lambda t.\ t \cdot \sigma$) *ss2*)

lemma *subst-apply-term-ctxt-apply-distrib* [*simp*]:
$C\langle t\rangle \cdot \mu = (C \cdot_c \mu)\langle t \cdot \mu\rangle$
by (*induct* C) *auto*

lemma *subst-compose-ctxt-compose-distrib* [*simp*]:
$(C \circ_c D) \cdot_c \sigma = (C \cdot_c \sigma) \circ_c (D \cdot_c \sigma)$
by (*induct* C) *auto*

lemma *ctxt-compose-subst-compose-distrib* [*simp*]:
$C \cdot_c (\sigma \circ_s \tau) = (C \cdot_c \sigma) \cdot_c \tau$
by (*induct* C) (*auto*)

8.3 The Connection between Contexts and the Superterm Relation

lemma *ctxt-imp-supteq* [*simp*]:
shows $C\langle t\rangle \trianglerighteq t$ **by** (*induct* C) *auto*

lemma *supteq-ctxtE*[*elim*]:
assumes $s \trianglerighteq t$ **obtains** C **where** $s = C\langle t\rangle$
using *assms* **proof** (*induct arbitrary*: *thesis*)
case (*refl s*)
have $s = \Box\langle s\rangle$ **by** *simp*
from *refl*[*OF this*] **show** *?case* **.**
next
case (*subt u ss t f*)

then obtain C where $u = C\langle t\rangle$ by *auto*
from *split-list*[*OF* ‹$u \in$ *set ss*›] obtain *ss1* and *ss2* where $ss = ss1 @ u \# ss2$ by *auto*
then have *Fun f ss* = (*More f ss1 C ss2*)$\langle t\rangle$ using ‹$u = C\langle t\rangle$› by *simp*
with *subt* show *?case* by *best*
qed

lemma *ctxt-supteq*[*intro*]:
assumes $s = C\langle t\rangle$ shows $s \trianglerighteq t$
proof (*cases C*)
case *Hole* then show *?thesis* using *assms* by *auto*
next
case (*More f ss1 D ss2*)
with *assms* have *s*: $s =$ *Fun f* (*ss1* @ $D\langle t\rangle$ # *ss2*) (is - = *Fun* - *?ss*) by *simp*
have $D\langle t\rangle \in$ *set ?ss* by *simp*
moreover have $D\langle t\rangle \trianglerighteq t$ by (*induct D*) *auto*
ultimately show *?thesis* unfolding *s* ..
qed

lemma *supteq-ctxt-conv*: $(s \trianglerighteq t) = (\exists\, C.\ s = C\langle t\rangle)$ by *auto*

lemma *supt-ctxtE*[*elim*]:
assumes $s \triangleright t$ obtains C where $C \neq \Box$ and $s = C\langle t\rangle$
using *assms*
proof (*induct arbitrary*: *thesis*)
case (*arg s ss f*)
from *split-list*[*OF* ‹$s \in$ *set ss*›] obtain *ss1* and *ss2* where *ss*: $ss = ss1 @ s \# ss2$ by *auto*
let *?C* = *More f ss1* $\Box$ *ss2*
have $?C \neq \Box$ by *simp*
moreover have *Fun f ss* = $?C\langle s\rangle$ by (*simp add*: *ss*)
ultimately show *?case* using *arg* by *best*
next
case (*subt s ss t f*)
then obtain C where $C \neq \Box$ and $s = C\langle t\rangle$ by *auto*
from *split-list*[*OF* ‹$s \in$ *set ss*›] obtain *ss1* and *ss2* where *ss*: $ss = ss1 @ s \# ss2$ by *auto*
have *More f ss1 C ss2* $\neq \Box$ by *simp*
moreover have *Fun f ss* = (*More f ss1 C ss2*)$\langle t\rangle$ using ‹$s = C\langle t\rangle$› by (*simp add*: *ss*)
ultimately show *?case* using *subt*(*4*) by *best*
qed

lemma *ctxt-supt*[*intro 2*]:
assumes $C \neq \Box$ and $s = C\langle t\rangle$ shows $s \triangleright t$
proof (*cases C*)
case *Hole* with *assms* show *?thesis* by *simp*
next
case (*More f ss1 D ss2*)

with *assms* **have** *s*: $s = Fun\ f\ (ss1\ @\ D\langle t\rangle\ \#\ ss2)$ **by** *simp*
have $D\langle t\rangle \in set\ (ss1\ @\ D\langle t\rangle\ \#\ ss2)$ **by** *simp*
then have $s \rhd D\langle t\rangle$ **unfolding** *s* **..**
also have $D\langle t\rangle \trianglerighteq t$ **by** (*induct D*) *auto*
finally show $s \rhd t$ **.**
qed

lemma *supt-ctxt-conv*: $(s \rhd t) = (\exists C.\ C \neq \Box \wedge s = C\langle t\rangle)$ (**is** - = *?rhs*)
proof
assume $s \rhd t$
then have $s \trianglerighteq t$ **and** $s \neq t$ **by** *auto*
from ‹$s \trianglerighteq t$› **obtain** *C* **where** $s = C\langle t\rangle$ **by** *auto*
with ‹$s \neq t$› **have** $C \neq \Box$ **by** *auto*
with ‹$s = C\langle t\rangle$› **show** *?rhs* **by** *auto*
next
assume *?rhs* **then show** $s \rhd t$ **by** *auto*
qed

lemma *nectxt-imp-supt-ctxt*: $C \neq \Box \Longrightarrow C\langle t\rangle \rhd t$ **by** *auto*

lemma *supt-var*: $\neg\ (Var\ x \rhd u)$
proof
assume $Var\ x \rhd u$
then obtain *C* **where** $C \neq \Box$ **and** $Var\ x = C\langle u\rangle$ **..**
then show *False* **by** (*cases C*) *auto*
qed

lemma *supt-const*: $\neg\ (Fun\ f\ []\ \rhd u)$
proof
assume $Fun\ f\ []\ \rhd u$
then obtain *C* **where** $C \neq \Box$ **and** $Fun\ f\ [] = C\langle u\rangle$ **..**
then show *False* **by** (*cases C*) *auto*
qed

lemma *supteq-var-imp-eq*:
$(Var\ x \trianglerighteq t) = (t = Var\ x)$ (**is** - = (- = *?x*))
proof
assume $t = Var\ x$ **then show** $Var\ x \trianglerighteq t$ **by** *auto*
next
assume *st*: $?x \trianglerighteq t$
from *st* **obtain** *C* **where** $?x = C\langle t\rangle$ **by** *best*
then show $t = ?x$ **by** (*cases C*) *auto*
qed

lemma *Var-supt* [*elim*!]:
assumes $Var\ x \rhd t$ **shows** *P*
using *assms supt-var*[*of x t*] **by** *simp*

lemma *Fun-supt* [*elim*]:

assumes *Fun f ts* $\rhd$ *s* **obtains** *t* **where** $t \in$ *set ts* **and** $t \unrhd s$
using *assms* **by** (*cases*) (*auto simp*: *supt-supteq-conv*)

lemma *inj-ctxt-apply-term*: *inj* (*ctxt-apply-term C*)
by (*auto simp*: *inj-on-def*)

lemma *ctxt-subst-eq*: $(\bigwedge x.\ x \in \textit{vars-ctxt}\ C \Longrightarrow \sigma\ x = \tau\ x) \Longrightarrow C \cdot_c \sigma = C \cdot_c \tau$
proof (*induct C*)
case (*More f bef C aft*)
{ fix *t*
assume *t*:$t \in$ *set bef*
have $t \cdot \sigma = t \cdot \tau$ **using** *t More*(*2*) **by** (*auto intro*: *term-subst-eq*)
}
moreover
{ fix *t*
assume *t*:$t \in$ *set aft*
have $t \cdot \sigma = t \cdot \tau$ **using** *t More*(*2*) **by** (*auto intro*: *term-subst-eq*)
}
moreover have $C \cdot_c \sigma = C \cdot_c \tau$ **using** *More* **by** *auto*
ultimately show *?case* **by** *auto*
qed *auto*

A *signature* is a set of function symbol/arity pairs, where the arity of a function symbol, denotes the number of arguments it expects.

type-synonym *'f sig* = (*'f* × *nat*) *set*

The set of all function symbol/ arity pairs occurring in a term.

fun *funas-term* :: (*'f*, *'v*) *term* ⇒ *'f sig*
where
funas-term (*Var* -) = {} |
funas-term (*Fun f ts*) = {(*f*, *length ts*)} ∪ $\bigcup$(*set* (*map funas-term ts*))

lemma *supt-imp-funas-term-subset*:
assumes $s \rhd t$
shows *funas-term t* ⊆ *funas-term s*
using *assms* **by** *induct auto*

lemma *supteq-imp-funas-term-subset*[*simp*]:
assumes $s \unrhd t$
shows *funas-term t* ⊆ *funas-term s*
using *assms* **by** *induct auto*

The set of all function symbol/arity pairs occurring in a context.

fun *funas-ctxt* :: (*'f*, *'v*) *ctxt* ⇒ *'f sig*
where
funas-ctxt Hole = {} |
funas-ctxt (*More f ss1 D ss2*) = {(*f*, *Suc* (*length* (*ss1* @ *ss2*)))}
∪ *funas-ctxt D* ∪ $\bigcup$(*set* (*map funas-term* (*ss1* @ *ss2*)))

lemma *funas-term-ctxt-apply* [*simp*]:
  funas-term $(C\langle t\rangle)$ = *funas-ctxt* $C \cup$ *funas-term* t
  by (*induct* C, *auto*)

lemma *funas-term-subst*:
  funas-term $(t \cdot \sigma)$ = *funas-term* $t \cup \bigcup$(*funas-term* ' σ ' *vars-term* t)
  by (*induct* t) *auto*

lemma *funas-ctxt-compose* [*simp*]:
  funas-ctxt $(C \circ_c D)$ = *funas-ctxt* $C \cup$ *funas-ctxt* D
  by (*induct* C) *auto*

lemma *funas-ctxt-subst* [*simp*]:
  funas-ctxt $(C \cdot_c \sigma)$ = *funas-ctxt* $C \cup \bigcup$(*funas-term* ' σ ' *vars-ctxt* C)
  by (*induct* C, *auto simp*: *funas-term-subst*)

end

References

[1] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[2] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.