First-Order Terms*

Christian Sternagel René Thiemann

April 15, 2024

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

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

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

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* $\Rightarrow$ $'v$ $\Rightarrow$ $'v$ **is** *fst* **.**
lift-definition *rename-2* :: $'v$ :: *infinite renaming2* $\Rightarrow$ $'v$ $\Rightarrow$ $'v$ **is** *snd* **.**

lemma *rename-12*: *inj* (*rename-1 r*) *inj* (*rename-2 r*) *range* (*rename-1 r*) $\cap$ *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 ∈ ?F and ∗: ∀ y. y < x ⟶ y ∉ ?F by auto
  obtain N where f N = x using ‹x ∈ ?F› by auto
  moreover have ∀ i > N. f i ∈ ?F by auto
  ultimately have ∀ i > N. ¬ f i < x using ∗ by auto
  moreover have ∀ i > N. f N ≥ f i
    using chainp-imp-rtranclp [of (≥) f, OF assms] by simp
  ultimately have ∀ 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 ∀ i. f i < f (Suc i)
  shows ∃ 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 ⊆ 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 ⇒ 'b)list ⇒ 'a set list ⇒ 'a ⇒ 'b
  where
    fun-merge fs as a = (fs ! (LEAST i. i < length as ∧ a ∈ 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 $fun\text{-}merge\ fs\ as\ a = (fs\ !\ i)\ a$
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 $fun\text{-}merge\ fs\ as\ a = (fs\ !\ i)\ a$
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 $guard :: bool \Rightarrow unit\ option$

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* $\gg=$ (λ -. *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* $\leftarrow$ *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 ∧ length zs = length xs ∧ 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 ⟷ length xs ≠ length ys
proof −
  {
    assume zip-option xs ys = None
    have length xs ≠ length ys
    proof (rule ccontr)
      assume ¬ length xs ≠ 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 ≠ 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⟧
    ⟹ zip-option xs ys = Some zs
  length xs ≠ length ys ⟹ 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* (λ*x*. *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: $\bigwedge y.\ mono\text{-}option\ (C\ y)$
shows $mono\text{-}option\ (\lambda f.\ mapM\ (\lambda y.\ C\ y\ f)\ B)$
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 ($funs\text{-}term : 'f,\ vars\text{-}term : 'v$) $term =$
$is\text{-}Var$: $Var\ (the\text{-}Var: 'v)$ |
$Fun\ 'f\ (args : ('f,\ 'v)\ term\ list)$
where
$args\ (Var\ \text{-}) = []$

lemmas $is\text{-}VarI = term.disc(1)$
lemmas $is\text{-}FunI = term.disc(2)$

abbreviation $is\text{-}Fun\ t \equiv \neg\ is\text{-}Var\ t$

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* ⇒ *'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* ⇒ ('*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*:
  $t \cdot \sigma = t \cdot \tau \longleftrightarrow (\forall x \in$ *vars-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* :: $('f,\ 'v)$ *subst* $\Rightarrow$ $'v$ *set*
  where
    subst-domain $\sigma = \{x.\ \sigma\ x \neq$ *Var* $x\}$

fun *subst-range* :: $('f,\ 'v)$ *subst* $\Rightarrow$ $('f,\ 'v)$ *term set*
  where
    subst-range $\sigma = \sigma$ ‘ *subst-domain* σ

lemma *vars-term-ms-subst* [*simp*]:
  vars-term-ms $(t \cdot \sigma) =$
    $(\sum x \in\#$*vars-term-ms* t. *vars-term-ms* $(\sigma\ x))$ (**is** - = *?V t*)
proof (*induct t*)
  case (*Fun f ts*)
  have *IH*: *map* $(\lambda\ t.$ *vars-term-ms* $(t \cdot \sigma))$ *ts* = *map* $(\lambda\ t.$ *?V t*) *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 *vars-term-ms s* $\subseteq\#$ *vars-term-ms t*
  shows *vars-term-ms* $(s \cdot \sigma) \subseteq\#$ *vars-term-ms* $(t \cdot \sigma)$
proof −
  from *assms*[*unfolded mset-subset-eq-exists-conv*] **obtain** *u* **where** *t*: *vars-term-ms t* = *vars-term-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* :: $('f,\ 'v)$ *subst* $\Rightarrow$ $'v$ *set*
where
  range-vars $\sigma = \bigcup$(*vars-term* ‘ *subst-range* σ)

lemma *mem-range-varsI*:
  assumes $\sigma\ x =$ *Var y* **and** $x \neq y$
  shows $y \in$ *range-vars* σ
  unfolding *range-vars-def UN-iff*
proof (*rule bexI*[*of - Var y*])
  show $y \in$ *vars-term* (*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 ∩ *subst-domain* σ = {} ⟹ *t* · σ = *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* · σ) = (⋃ *x* ∈ *vars-term t*. *vars-term* (σ *x*))
  by (*induction t*) (*auto simp add*: *insert-Diff-if subst-domain-def*)

corollary *vars-term-subst-apply-term-subset*:
  vars-term (*t* · σ) ⊆ *vars-term t* − *subst-domain* σ ∪ *range-vars* σ
  unfolding *vars-term-subst-apply-term*
proof (*induction t*)
  case (*Var x*)
  show *?case*
    by (*cases* σ *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* ⇒ *bool*
  where
    is-renaming σ ⟷ (∀ *x*. *is-Var* (σ *x*)) ∧ *inj-on* σ (*subst-domain* σ)

lemma *inv-renaming-sound*:
  assumes *is-var-*ϱ: ∀ *x*. *is-Var* (ϱ *x*) **and** *inj* ϱ
  shows ϱ $\circ_s$ (*Var* ∘ (*inv* (*the-Var* ∘ ϱ))) = *Var*
proof −
  define ϱ′ **where** ϱ′ = *the-Var* ∘ ϱ

```
  have ϱ-def: ϱ = Var ∘ 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 ϱ' ∘ ϱ' = id
    using inv-o-cancel[of ϱ'] by simp
  hence Var ∘ (inv ϱ' ∘ ϱ') = Var
    by simp
  hence ∀x. (Var ∘ (inv ϱ' ∘ ϱ')) x = Var x
    by metis
  hence ∀x. ((Var ∘ ϱ') ∘s (Var ∘ (inv ϱ'))) x = Var x
    unfolding subst-compose-def by auto
  thus ϱ ∘s (Var ∘ (inv ϱ')) = Var
    using ϱ-def by auto
qed

lemma ex-inverse-of-renaming:
  assumes ∀x. is-Var (ϱ x) and inj ϱ
  shows ∃τ. ϱ ∘s τ = Var
  using inv-renaming-sound[OF assms] by blast

lemma vars-term-subst:
  vars-term (t · σ) = ⋃(vars-term ‘ σ ‘ vars-term t)
  by (induct t) simp-all

lemma range-varsE [elim]:
  assumes x ∈ range-vars σ
    and ⋀t. x ∈ vars-term t ⟹ t ∈ subst-range σ ⟹ P
  shows P
  using assms by (auto simp: range-vars-def)

lemma range-vars-subst-compose-subset:
  range-vars (σ ∘s τ) ⊆ (range-vars σ − subst-domain τ) ∪ range-vars τ (is ?L ⊆
?R)
proof
  fix x
  assume x ∈ ?L
  then obtain y where y ∈ subst-domain (σ ∘s τ)
    and x ∈ vars-term ((σ ∘s τ) y) by (auto simp: range-vars-def)
  then show x ∈ ?R
  proof (cases)
    assume y ∈ subst-domain σ and x ∈ vars-term ((σ ∘s τ) y)
    moreover then obtain v where v ∈ vars-term (σ y)
      and x ∈ vars-term (τ v) by (auto simp: subst-compose-def vars-term-subst)
    ultimately show ?thesis
      by (cases v ∈ 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* ∈ {*y*. *subst x t y* ≠ *Var y*} ⟷ *y* ∈ (*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 σ *x* = *t* · σ
  shows *s* · *subst x t* · σ = *s* · σ
  using *assms* **by** (*induct s*) (*auto simp*: *subst-def*)

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

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

lemma *subst-self-idemp* [*simp*]:
  x ∉ *vars-term t* ⟹ *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* ⇒ (*'f*, *'v*, *'w*) *gsubst* ⇒ (*'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 $(\sigma \circ_s \tau) =$
$(subst\text{-}domain\ \sigma - \{x.\ \exists\, y.\ \sigma\ x = Var\ y \wedge \tau\ y = Var\ x\}) \cup$
$(subst\text{-}domain\ \tau - subst\text{-}domain\ \sigma)$
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*:
$\sigma \circ_s \sigma = \sigma \longleftrightarrow subst\text{-}domain\ \sigma \cap range\text{-}vars\ \sigma = \{\}$
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 $*$: $\bigwedge x.\ \forall\, y \in vars\text{-}term\ (\sigma\ x).\ \sigma\ y = Var\ y$
unfolding *term-subst-eq-conv* **by** *simp*
{ fix $x\ y$
assume $\sigma\ x \neq Var\ x$ **and** $x \in vars\text{-}term\ (\sigma\ y)$
with $*$ [*of y*] **have** *False* **by** *simp* **}**
then show $subst\text{-}domain\ \sigma \cap range\text{-}vars\ \sigma = \{\}$
by (*auto simp*: *subst-domain-def range-vars-def*)
next
assume $subst\text{-}domain\ \sigma \cap range\text{-}vars\ \sigma = \{\}$
then have $*$: $\bigwedge x\ y.\ \sigma\ x = Var\ x \vee \sigma\ y = Var\ y \vee x \notin vars\text{-}term\ (\sigma\ y)$
by (*auto simp*: *subst-domain-def range-vars-def*)
have $\bigwedge x.\ \forall\, y \in vars\text{-}term\ (\sigma\ x).\ \sigma\ y = Var\ y$
proof
fix $x\ y$
assume $y \in vars\text{-}term\ (\sigma\ x)$
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 σ ∩ subst-domain δ = {} and vars-term t ∩ 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* (σ 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* σ x *else* τ 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* σ 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) = $V \cap$ *subst-domain* σ
unfolding *restrict-subst-domain-def subst-domain-def* **by** *auto*

lemma *subst-apply-term-restrict-subst-domain*:
vars-term $t \subseteq V \Longrightarrow t \cdot$ *restrict-subst-domain* V $\sigma = t \cdot \sigma$
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* $x \in \varrho$ ‘ *subst-domain* σ *then*
σ (*the-inv* ϱ (*Var* x))
else
Var x)

lemma *rename-subst-domain-Var-lhs*[*simp*]:
rename-subst-domain Var $\sigma = \sigma$
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: $\forall x$. *is-Var* (ϱ x)
shows *subst-domain* (*rename-subst-domain* ϱ σ) $\subseteq$ *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) $\subseteq$ *subst-range* σ
proof (*intro Set.equalityI Set.subsetI*)
fix t **assume** $t \in$ *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 $x \neq$ *Var* x
by (*auto simp*: *image-iff subst-domain-def*)

show $t \in$ *subst-range* σ
proof (*cases* ‹*Var* $x \in \varrho$ ‘ *subst-domain* σ›)
case *True*
then obtain x' **where** ϱ 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 ϱ σ x ≠ 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 ϱ σ) ⊆ 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: ∀ x. is-Var (ϱ x) and inj ϱ and vars-t: vars-term t ⊆ subst-domain
σ
  shows t · ϱ · rename-subst-domain ϱ σ = t · σ
  unfolding subst-subst
proof (intro term-subst-eq ballI)
  fix x assume x ∈ vars-term t
  with vars-t have x-in: x ∈ subst-domain σ
    by blast

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

  have (ϱ ∘s rename-subst-domain ϱ σ) x = ϱ x · rename-subst-domain ϱ σ
    by (simp add: subst-compose-def)
  also have ... = rename-subst-domain ϱ σ x'
    using ϱ-x by simp
  also have ... = σ (the-inv ϱ (Var x'))
    by (simp add: rename-subst-domain-def if-P[OF x'-in])
  also have ... = σ (the-inv ϱ (ϱ x))
    by (simp add: ϱ-x)
  also have ... = σ x
    using ‹inj ϱ› by (simp add: the-inv-f-f)
  finally show (ϱ ∘s rename-subst-domain ϱ σ) x = σ 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 $\varrho \circ_s$ *rename-subst-domain-range* ϱ $\sigma = \sigma \circ_s \varrho$
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 ($\varrho \circ_s$ *rename-subst-domain-range* ϱ σ) x = ($\sigma \circ_s \varrho$) 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* $(\varrho\ x)$ **and** *inj* ϱ
shows $t \cdot \varrho \cdot$ *rename-subst-domain-range* $\varrho\ \sigma = t \cdot \sigma \cdot \varrho$
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 $(t \cdot \sigma) \longleftrightarrow (\forall x \in$ *vars-term t. ground* $(\sigma\ x))$
by (*induct t*) *simp-all*

lemma *ground-subst-apply*:
assumes *ground t*
shows $t \cdot \sigma = t$
proof −
have $t = t \cdot$ *Var* **by** *simp*
also have ... $= t \cdot \sigma$
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* $(\lambda x.\ x)$ *f*

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

lemma *map-vars-term-eq*:
map-vars-term f s $= s \cdot$ (*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 ((+) ∘ *pair-size*) *0 M* + *n* = *fold-mset* ((+) ∘ *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*)) ≤ *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* (λp. (*fst* $p \cdot \sigma$, *snd* $p \cdot \sigma$)) 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* · *subst x t*) $\cup$ *vars-term* (*v* · *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 *σ x* = *None* ∨ *σ x* = *Some t*
shows *matchers-map* (*λy*. *if y* = *x then Some t else σ y*) =
matchers-map σ ∩ {*τ*. *τ 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 $\sigma\ x = \mathit{None} \lor \sigma\ x = \mathit{Some}\ t$
shows $\mathit{matchers\text{-}map}\ (\sigma\ (x \mapsto t)) = \mathit{matchers\text{-}map}\ \sigma \cap \{\tau.\ \tau\ x = t\}$
using *matchers-map-upd* [*of* $\sigma\ x\ t$, *OF assms*]
by (*simp add*: *matchers-map-def matchers-def dom-def*)

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

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

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

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

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

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

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

lemma *MATCH1-size-mset*:
assumes *MATCH1 x y*
shows $\mathit{size\text{-}mset}\ (\mathit{fst}\ x) > \mathit{size\text{-}mset}\ (\mathit{fst}\ y)$

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 $(y, x) \in$ *matchless*
using *MATCH1-size-mset* [*OF assms*]
by (*simp add*: *matchless-def*)

lemma *converse-matchrel-subset-matchless*:
$matchrel^{-1} \subseteq$ *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*]:
$\neg\ MATCH1\ (\{\#(Var\ x,\ t)\#\},\ \sigma)\ (\{\#\},\ \sigma\ (x \mapsto t)) \Longrightarrow \sigma\ x \neq None \wedge \sigma\ x \neq Some\ t$
by *auto*

lemma *not-matchrelD*:
assumes $\neg\ (\exists\, y.\ ((\{\#e\#\},\ \sigma),\ y) \in matchrel)$
shows $(\exists\, f\ ss\ x.\ e = (Fun\ f\ ss,\ Var\ x)) \vee$
$(\exists\, x\ t.\ e = (Var\ x,\ t) \wedge \sigma\ x \neq None \wedge \sigma\ x \neq Some\ t) \vee$
$(\exists\, f\ g\ ss\ ts.\ e = (Fun\ f\ ss,\ Fun\ g\ ts) \wedge (f \neq g \vee length\ ss \neq length\ ts))$
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*: *matchrel-def*)+

```
  qed
qed

lemma ne-matchers-imp-matchrel:
  assumes matchers-map σ ∩ matchers {e} ≠ {}
  shows ∃ y. (({#e#}, σ), y) ∈ matchrel
proof (rule ccontr)
  assume ¬ ?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, σ) (P′ + M, σ′)
  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) ∈ matchrel
  shows ((fst x + M, snd x), (fst y + M, snd y)) ∈ matchrel
  using assms and MATCH1-mono [of fst x]
  by (simp add: MATCH1-matchrel-conv)

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

lemma ne-matchers-imp-empty-or-matchrel:
  assumes matchers-map σ ∩ matchers (set-mset P) ≠ {}
  shows P = {#} ∨ (∃ y. ((P, σ), y) ∈ matchrel)
proof (cases P)
  case (add e P′)
  then have [simp]: P = P′ + {#e#} by simp
  from assms have matchers-map σ ∩ matchers {e} ≠ {} by auto
  from ne-matchers-imp-matchrel [OF this]
    obtain P″ σ′ where MATCH1 ({#e#}, σ) (P″, σ′)
    by (auto simp: matchrel-def)
  from MATCH1-mono [OF this, of P′] have MATCH1 (P, σ) (P′ + P″, σ′) by
(simp add: ac-simps)
  then show ?thesis by (auto simp: matchrel-def)
qed simp

lemma matchrel-imp-converse-matchless [dest]:
  (x, y) ∈ matchrel ⟹ (y, x) ∈ matchless
```

using *MATCH1-matchless* **by** (*cases x*, *cases y*) (*auto simp*: *matchrel-def*)

lemma *ne-matchers-imp-empty*:
fixes P :: (('f, 'v) *term* × ('f, 'w) *term*) *multiset*
assumes *matchers-map* σ ∩ *matchers* (*set-mset* P) ≠ {}
shows ∃σ'. ((P, σ), ({#}, σ')) ∈ *matchrel**
using *assms*
proof (*induct P arbitrary*: σ *rule*: *wf-induct* [*OF wf-measure* [*of size-mset*]])
fix P :: (('f, 'v) *term* × ('f, 'w) *term*) *multiset*
and σ
presume *IH*: ⋀P' σ. ⟦(P', P) ∈ *measure size-mset*; *matchers-map* σ ∩ *matchers* (*set-mset* P') ≠ {}⟧ ⟹
∃σ'. ((P', σ), {#}, σ') ∈ *matchrel**
and ∗: *matchers-map* σ ∩ *matchers* (*set-mset* P) ≠ {}
show ∃σ'. ((P, σ), {#}, σ') ∈ *matchrel**
proof (*cases* P = {#})
assume P ≠ {#}
with *ne-matchers-imp-empty-or-matchrel* [*OF* ∗]
obtain P' σ' **where** ∗∗: ((P, σ), (P', σ')) ∈ *matchrel* **by** (*auto*)
with ∗ **have** (P', P) ∈ *measure size-mset*
and *matchers-map* σ' ∩ *matchers* (*set-mset* P') ≠ {}
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 ⋀σ'. ((P, σ), ({#}, σ')) ∉ *matchrel**
shows *matchers-map* σ ∩ *matchers* (*set-mset* P) = {}
using *ne-matchers-imp-empty* [*of* σ P] **and** *assms* **by** *blast*

lemma *irreducible-reachable-imp-matchers-empty*:
assumes ((P, σ), y) ∈ *matchrel*$^!$ **and** *fst* y ≠ {#}
shows *matchers-map* σ ∩ *matchers* (*set-mset* P) = {}
proof −
have ((P, σ), y) ∈ *matchrel**
and ⋀τ. (y, ({#}, τ)) ∉ *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) ∩ *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 σ ≠ {}
{} ≠ *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 $\neg$ *?thesis*
then have $*$: $\bigwedge y.$ ((*P*, *Map.empty*), *y*) $\in$ *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*) $\in$ *matchrel*$^!$
by (*metis SN-imp-WN UNIV-I WN-onE*)
with $*$ [*OF this*] **obtain** τ **where** ((*P*, *Map.empty*), ({#}, τ)) $\in$ *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*) $\in$ *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* $\times$ ('*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* $\Rightarrow$ ('*f*, '*v*) *subst set*
where
unifiers E = $\{\sigma.\ \forall\, p{\in}E.$ *fst p* $\cdot\ \sigma$ = *snd p* $\cdot\ \sigma\}$

Check whether a set of equations is unifiable.

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

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

Applying a substitution to a set of equations.

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

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

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

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

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

5.1.1 Properties of sets of unifiers

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

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

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

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

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

lemma *unifiers-insert-Var-swap* [*simp*]:
$\textit{unifiers}\ (\textit{insert}\ (t,\ \textit{Var}\ x)\ E) = \textit{unifiers}\ (\textit{insert}\ (\textit{Var}\ x,\ t)\ E)$
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** σ *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* $\sigma\ E$)
  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* $\sigma\ E$
  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 $\exists \mu.\ \tau =$ *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_ϱ = (*fst* $e \cdot \varrho$, *snd* $e \cdot \varrho$)
      by (*auto simp*: *subst-set-def*)

```
  then show fst e_ϱ · rename-subst-domain-range ϱ μ = snd e_ϱ · 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 υ ∈ unifiers (subst-set ϱ E)
  hence (ϱ ∘s υ) ∈ unifiers E
    by (simp add: subst-set-def unifiers-def)
  with imgu-μ have μ-ϱ-υ: μ ∘s ϱ ∘s υ = ϱ ∘s υ
    by (simp add: is-imgu-def subst-compose-assoc)

  show υ = rename-subst-domain-range ϱ μ ∘s υ
  proof (rule ext)
    fix x
    show υ x = (rename-subst-domain-range ϱ μ ∘s υ) x
    proof (cases Var x ∈ ϱ ` subst-domain μ)
      case True
       hence (rename-subst-domain-range ϱ μ ∘s υ) x = (μ ∘s ϱ ∘s υ) (the-inv ϱ
(Var x))
        by (simp add: rename-subst-domain-range-def subst-compose-def)
      also have ... = (ϱ ∘s υ) (the-inv ϱ (Var x))
        by (simp add: μ-ϱ-υ)
      also have ... = (ϱ (the-inv ϱ (Var x))) · υ
        by (simp add: subst-compose)
      also have ... = Var x · υ
        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: ∀x. is-Var (ϱ x) and inj ϱ
  shows is-imgu (rename-subst-domain-range ϱ μ) {(t · ϱ, u · ϱ)}
   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*:
$UNIF1\ \sigma\ E\ E' \Longrightarrow (E,\ E') \in unif$
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*) (λx. (*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* $\Rightarrow$ $('f, 'v)$ *equation multiset* $\Rightarrow$ $('f, 'v)$ *equation multiset* $\Rightarrow$ *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*:
$(E, E') \in$ *unif** $\longleftrightarrow$ ($\exists$ *ss*. *UNIF ss E E'*)
proof
assume $(E, E') \in$ *unif**
then show $\exists$ *ss*. *UNIF ss E E'*
by (*induct rule*: *converse-rtrancl-induct*) (*auto simp*: *unif-UNIF1-conv*)
next
assume $\exists$ *ss*. *UNIF ss E E'*
then obtain *ss* **where** *UNIF ss E E'* **..**
then show $(E, E') \in$ *unif** **by** (*induct*) (*auto dest*: *UNIF1-unifD*)
qed

Compose a list of substitutions.

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

lemma *compose-simps* [*simp*]:
compose [] = *Var*
compose (*Var* # *ss*) = *compose ss*
compose (σ # *ss*) = $\sigma \circ_s$ *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* $\sigma \cap$ *vars-mset E'* = {}
using *assms* **by** (*cases*) *simp*+

lemma *UNIF1-subst-domain-subset*:
assumes *UNIF1* σ *E E'*
shows *subst-domain* $\sigma \subseteq$ *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* ($\sigma \circ_s \tau$) *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″*) ∈ *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″* + *subst-mset σ 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** *τ* ∈ *unifiers* (*set-mset E*)
  shows (*σ* $\circ_s$ *τ*) ∈ *unifiers* (*set-mset E′*)
  using *assms* **by** (*cases*) (*auto simp*: *unifiers-def subst-mset-def*)

lemma *unif-preserves-unifiable*:
  assumes (*E*, *E′*) ∈ *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*) ∈ *unif* ⟹ (*y*, *x*) ∈ *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*, {#}) ∈ *unif**
  using *assms*
proof (*induct E rule*: *wf-induct* [*OF wf-unifless*])
  fix *E* :: (*′f*, *′v*) *equation multiset*
  presume *IH*: ⋀*E′*. ⟦(*E′*, *E*) ∈ *unifless*; *unifiable* (*set-mset E′*)⟧ ⟹
    (*E′*, {#}) ∈ *unif**
    and ∗: *unifiable* (*set-mset E*)
  show (*E*, {#}) ∈ *unif**
  proof (*cases E* = {#})
    assume *E* ≠ {#}
    with *unifiable-imp-empty-or-unif* [*OF* ∗]
    obtain *E′* **where** (*E*, *E′*) ∈ *unif* **by** *auto*
    with ∗ **have** (*E′*, *E*) ∈ *unifless* **and** *unifiable* (*set-mset E′*)
      by (*auto dest*: *unif-preserves-unifiable*)
    from *IH* [*OF this*] **and** ‹(*E*, *E′*) ∈ *unif*›
    show *?thesis* **by** *simp*
  qed *simp*
qed *simp*

lemma *unif-rtrancl-empty-imp-unifiable*:
  assumes $(E, \{\#\}) \in \mathit{unif}^*$
  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 $\exists E'.\ E' \neq \{\#\} \wedge (E, E') \in \mathit{unif}^!$
proof (*rule ccontr*)
  assume $\neg$ *?thesis*
  then have $*$: $\bigwedge E'.\ (E, E') \in \mathit{unif}^! \Longrightarrow E' = \{\#\}$ **by** *auto*
  have *SN unif* **using** *wf-converse-unif* **by** (*auto simp*: *SN-iff-wf*)
  then obtain E' **where** $(E, E') \in \mathit{unif}^!$
    by (*metis SN-imp-WN UNIV-I WN-onE*)
  with $*$ **have** $(E, \{\#\}) \in \mathit{unif}^*$ **by** *auto*
  from *unif-rtrancl-empty-imp-unifiable* [*OF this*] **and** *assms*
  show *False* **by** *contradiction*
qed

lemma *unif-rtrancl-preserves-unifiable*:
  assumes $(E, E') \in \mathit{unif}^*$ **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 $(E, \{\#\}) \notin \mathit{unif}^*$
  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 $(E, E') \in \mathit{unif}^!$ **and** $E' \neq \{\#\}$
  shows $\neg$ *unifiable* (*set-mset* E)
proof $-$
  have $(E, E') \in \mathit{unif}^*$ **and** $(E', \{\#\}) \notin \mathit{unif}^*$
    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*) ⇒ *if f* = *g then zip-option ss ts else None*
| - ⇒ *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* ⟹
f = *g* ∧ *length ss* = *length ts* ∧ *E* = *zip ss ts*
by (*cases f* = *g*) (*auto simp*: *decompose-def*)

lemma *decompose-None* [*dest*]:
decompose (*Fun f ss*) (*Fun g ts*) = *None* ⟹ *f* ≠ *g* ∨ *length ss* ≠ *length ts*
by (*cases f* = *g*) (*auto simp*: *decompose-def*)

Applying a substitution to a list of equations.

definition
subst-list :: ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *equation list* ⇒ ('*f*, '*v*) *equation list*
where
subst-list σ *ys* = *map* (λ*p*. (*fst p* · σ, *snd p* · σ)) *ys*

lemma *mset-subst-list* [*simp*]:
mset (*subst-list* (*subst x t*) *ys*) = *subst-mset* (*subst x t*) (*mset ys*)
by (*auto simp*: *subst-mset-def subst-list-def*)

lemma *subst-list-append*:
subst-list σ (*xs* @ *ys*) = *subst-list* σ *xs* @ *subst-list* σ *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. σ ∘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 ∘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 ∘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 ∃ E'. E' ≠ {#} ∧ (mset E, E') ∈ 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, {#}) ∈ 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 · σ, Fun g ts ·
σ)#}) ∈ unif*
        by (auto simp: subst-mset-def)
      moreover have {#(Fun f ss · σ, Fun g ts · σ)#} ∈ 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 ¬ unifiable {(Fun f ss, Fun g ts)}
        using * by (auto simp: unifiable-def)
      ultimately have ¬ 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' ≠ {#} and (mset (us @ E), E') ∈ unif! by blast
    moreover have (mset ((Fun f ss, Fun g ts) # E), mset (us @ E)) ∈ 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′ ≠ {#} and (mset E, E′) ∈ unif! using 3 by auto
    moreover have (mset ((Var x, t) # E), mset E) ∈ unif
      by (auto intro: UNIF1.trivial simp: unif-def)
    ultimately have ?case by auto }
  moreover
  { assume ∗: t ≠ Var x x ∉ vars-term t
    then obtain E′ where E′ ≠ {#}
      and (mset (subst-list (subst x t) E), E′) ∈ unif! using 3 by auto
    moreover have (mset ((Var x, t) # E), mset (subst-list (subst x t) E)) ∈ unif
      using ∗ by (auto intro: UNIF1.Var-left simp: unif-def)
    ultimately have ?case by auto }
  moreover
  { assume ∗: t ≠ Var x x ∈ vars-term t
    then have x ∈ vars-term t is-Fun t by auto
    then have ¬ unifiable {(Var x, t)} by (rule in-vars-is-Fun-not-unifiable)
    then have ∗∗: ¬ unifiable {(Var x · σ, t · 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, {#}) ∈ 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 · σ, t · σ)#}) ∈ unif∗
        by (auto simp: subst-mset-def)
      moreover obtain E′ where E′ ≠ {#}
        and ({#(Var x · σ, t · σ)#}, E′) ∈ unif!
        using not-unifiable-imp-not-empty-NF and ∗∗
          by (metis set-mset-single)
      ultimately show ?thesis by auto
    next
      case False
      moreover have ¬ unifiable {(Var x, t)}
        using ∗ by (force simp: unifiable-def)
     ultimately have ¬ 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 ∉ vars-term t
    then obtain E′ where E′ ≠ {#}
       and (mset (subst-list (subst x t) E), E′) ∈ unif! using 4 by (auto simp:
```

t-def)
moreover have (*mset* ((*t*, *Var x*) # *E*), *mset* (*subst-list* (*subst x t*) *E*)) ∈ *unif*
using ∗ **by** (*auto intro*: *UNIF1.Var-right simp*: *unif-def*)
ultimately have *?case* **by** (*auto simp*: *t-def*) **}**
moreover
{ assume *x* ∈ *vars-term t*
then have ∗: *x* ∈ *vars-term t t* ≠ *Var x* **by** (*auto simp*: *t-def*)
then have *x* ∈ *vars-term t is-Fun t* **by** *auto*
then have ¬ *unifiable* {(*Var x*, *t*)} **by** (*rule in-vars-is-Fun-not-unifiable*)
then have ∗∗: ¬ *unifiable* {(*Var x* · *σ*, *t* · *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*, {#}) ∈ *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* · *σ*, *Var x* · *σ*)#}) ∈ *unif*∗
by (*auto simp*: *subst-mset-def*)
moreover obtain *E′* **where** *E′* ≠ {#}
and ({#(*t* · *σ*, *Var x* · *σ*)#}, *E′*) ∈ *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 ¬ *unifiable* {(*t*, *Var x*)}
using ∗ **by** (*simp add*: *unifiable-def*)
ultimately have ¬ *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′* ≠ {#} **and** (*mset E*, *E′*) ∈ *unif*! **by** *blast*
then have ¬ *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 ≠ {} ⟹ *set xs* = *es* ⟹ ∃ *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*)} ≠ {} $\Longrightarrow$ ∃ *μ*. *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 ∃ *μ* :: (*'f*, *'v*) *subst*. *Unification.mgu t u* = *Some μ*
proof −
from *t-eq-u* **have** *unifiers* {(*t*, *u*)} ≠ {}
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*) ⊆ (⋃ *e* ∈ *set E*. *vars-term* (*fst e*) ∪ *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* $\mu = \{\}$
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* (μ *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-term* (*fst e*) $\cup$ *vars-term* (*snd e*)). *is-Var* (μ *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-term* (*fst e*) $\cup$ *vars-term* (*snd e*)))
proof (*rule inj-onI*)
fix *x y*
assume
x-in: $x \in -$ ($\bigcup e{\in}E$. *vars-term* (*fst e*) $\cup$ *vars-term* (*snd e*)) **and**
y-in: $y \in -$ ($\bigcup e{\in}E$. *vars-term* (*fst e*) $\cup$ *vars-term* (*snd e*)) **and**
μ *x* = μ *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**
τ = (λx. *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* $\cdot$ μ = *snd e* $\cdot$ μ
by *blast*
moreover from ‹$e \in E$› **have** *fst e* $\cdot$ τ = *fst e* $\cdot$ μ **and** *snd e* $\cdot$ τ = *snd e* $\cdot$ μ
unfolding *term-subst-eq-conv*
by (*auto simp*: τ*-def*)
ultimately show *fst e* $\cdot$ τ = *snd e* $\cdot$ τ
by *simp*
qed
with *minimal-*μ **obtain** γ **where** $\mu \circ_s \gamma = \tau$
by *auto*
hence ($\mu \circ_s \gamma$) *x* = *Var x* **and** ($\mu \circ_s \gamma$) *y* = *Var y*
using *ComplD*[*OF x-in*] *ComplD*[*OF y-in*]
by (*simp-all add*: τ*-def*)
with ‹μ *x* = μ *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 *Evars* $\equiv$ $(\bigcup e \in E.$ *vars-term* (*fst* e) $\cup$ *vars-term* (*snd* e))
  shows *subst-domain* $\mu \subseteq$ *Evars*
proof (*intro Set.subsetI*)
  from *imgu-μ* **have** *unif-μ*: $\mu \in$ *unifiers* E **and** *minimal-μ*: $\forall \tau \in$ *unifiers* $E.\ \mu \circ_s \tau = \tau$
    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* :: $('v \times ('f, 'v)$ *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**
    τ = *subst-of xs*

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

  have $\tau \in$ *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** $x \in$ *subst-domain* μ
  hence $\mu\ x \neq$ *Var* x
    by (*simp add*: *subst-domain-def*)

  show $x \in$ *Evars*
  proof (*cases* $x \in$ *subst-domain* τ)
    case *True*
    thus *?thesis*
      using *dom-τ* **by** *auto*
  next
    case *False*
    hence $\tau\ x =$ *Var* x
      by (*simp add*: *subst-domain-def*)
    hence $\mu\ x \cdot \tau =$ *Var* x
      using *μ-comp-τ*[*of* x] **by** (*simp add*: *subst-compose*)
    thus *?thesis*
    proof (*rule subst-apply-eq-Var*)

show $\bigwedge y.\ \mu\ x = Var\ y \Longrightarrow \tau\ y = Var\ x \Longrightarrow ?thesis$
using ‹$\mu\ x \neq Var\ x$› *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* $\equiv (\bigcup e \in E.$ *vars-term* (*fst* e) $\cup$ *vars-term* (*snd* e))
shows $(\bigcup x \in$ *Evars*. *vars-term* ($\mu\ x$)) $\subseteq$ *Evars*
proof (*rule Set.subsetI*)
from *imgu-μ* **have** *unif-μ*: $\mu \in$ *unifiers* E **and** *minimal-μ*: $\forall \tau \in$ *unifiers* $E.\ \mu \circ_s \tau = \tau$
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* :: $('v \times ('f, 'v)$ *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**
τ = *subst-of xs*

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

have $\tau \in$ *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 y :: $'v$ **assume** $y \in (\bigcup x \in$ *Evars*. *vars-term* ($\mu\ x$))
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 ∉ Evars
    hence y ∉ vars-term (τ x)
      using vars-τ-x by blast
    moreover have y ∈ vars-term ((μ ∘s τ) x)
    proof −
      have τ y = Var y
        using ‹y ∉ 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 μ E and fin-E: finite E
  shows range-vars μ ⊆ (⋃e ∈ E. vars-term (fst e) ∪ vars-term (snd e))
proof −
  have range-vars μ = (⋃x ∈ subst-domain μ. vars-term (μ x))
    by (simp add: range-vars-def)
   also have ... ⊆ (⋃x ∈ (⋃e ∈ E. vars-term (fst e) ∪ vars-term (snd e)).
vars-term (μ x))
    using imgu-subst-domain-subset[OF imgu-μ fin-E] by fast
  also have ... ⊆ (⋃e ∈ E. vars-term (fst e) ∪ 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 ⇒ ('f, 'v) term ⇒ ('f ,'v) subst where
  the-mgu s t = (case mgu s t of None ⇒ Var | Some δ ⇒ δ)

lemma the-mgu-is-imgu:
  fixes σ :: ('f, 'v) subst
  assumes s · σ = t · σ
  shows is-imgu (the-mgu s t) {(s, t)}
proof −
  from assms have unifiers {(s, t)} ≠ {} 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* $s\ t \wedge \sigma =$ *the-mgu* $s\ t \circ_s \sigma$
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 ::
    $('v \Rightarrow 'u) \Rightarrow ('w \Rightarrow 'u) \Rightarrow ('f, 'v)$ *term* $\Rightarrow ('f, 'w)$ *term* $\Rightarrow$
      $(('f, 'v, 'u)$ *gsubst* $\times ('f, 'w, 'u)$ *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 $\Rightarrow$ *None*
    | *Some* $\gamma \Rightarrow$ *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* $\cdot \gamma =$ *map-vars-term wu t* $\cdot \gamma$ **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*: $s \cdot \sigma = t \cdot \tau$
shows $\exists \mu 1\ \mu 2\ \delta$. *mgu-var-disjoint-generic vu wu s t* = *Some* ($\mu 1$, $\mu 2$) $\wedge$
$\sigma = \mu 1 \circ_s \delta \wedge$
$\tau = \mu 2 \circ_s \delta \wedge$
$s \cdot \mu 1 = t \cdot \mu 2$
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*: $s \cdot \sigma$ = *map-vars-term vu s* $\cdot\ \gamma$ **unfolding** *gamma*
by (*induct s*, *auto*)
have *idt*: $t \cdot \tau$ = *map-vars-term wu t* $\cdot\ \gamma$ **unfolding** *gamma*
by (*induct t*, *insert vwu*, *auto*)
from *unif-disj ids idt*
have *unif*: *map-vars-term vu s* $\cdot\ \gamma$ = *map-vars-term wu t* $\cdot\ \gamma$ (**is** *?s* $\cdot\ \gamma$ = *?t* $\cdot\ \gamma$) **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* $\circ_s\ \gamma$ **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*: $\gamma = \mu \circ_s \gamma$ **by** *auto*
let *?tau1* = $\mu \circ$ *vu*
let *?tau2* = $\mu \circ$ *wu*
show *?thesis* **unfolding** *mgu-var-disjoint-generic-def mu option.simps*
proof (*intro exI conjI*, *rule refl*)
show σ = *?tau1* $\circ_s\ \gamma$
proof (*rule ext*)
fix *x*
have (*?tau1* $\circ_s\ \gamma$) *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* $\circ_s\ \gamma$) *x* **by** *simp*
qed
next
show τ = *?tau2* $\circ_s\ \gamma$
proof (*rule ext*)
fix *x*
have (*?tau2* $\circ_s\ \gamma$) *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* $\circ_s\ \gamma$) *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 (γ1, γ2) ⟹ s · γ1 = t · γ2
  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 ∃μ1 μ2 δ. mgu-var-disjoint-sum s t = Some (μ1, μ2) ∧
    σ = μ1 ∘s δ ∧
    τ = μ2 ∘s δ ∧
    s · μ1 = t · μ2
 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 ∃μ1 μ2 τ. mgu-var-disjoint-sum s t = Some (μ1, μ2) ∧
    σ = μ1 ∘s τ ∧
    δ = μ2 ∘s τ ∧
    s · μ1 = t · μ2
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 μ1 μ2 τ where mgu: mgu-var-disjoint-sum s t = Some (μ1,μ2)
    and σ: ?m σ = μ1 ∘s τ
    and δ: ?m δ = μ2 ∘s τ
    and unif: s · μ1 = t · μ2 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* ⇒ - ⇒ - **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* ($\mu 1$, $\mu 2$) ∧
$\sigma = \mu 1 \circ_s \delta$ ∧
$\tau = \mu 2 \circ_s \delta$ ∧
$s \cdot \mu 1 = t \cdot \mu 2$
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* ∨ 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 ⇒ None
  | Some us ⇒ 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 ∘ 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)) ∈ 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 ∨ σ x = Some t
    and ∗∗: match-term-list P (σ (x ↦ 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 ↦ t))) ∈ 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)) ∈ 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 ∩ matchers (set P) = {}
using assms
proof (induction P σ rule: match-term-list.induct)
  case (2 x t P σ)
  have ¬ (σ x = None ∨ σ x = Some t) ∨
    (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 σ ∈ 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 ≡ ⋃ ((*vars-term* ∘ *fst*) ‘ *set P*)

lemma *match-list-complete'*:
  assumes ⋀*s t*. (*s*, *t*) ∈ *set P* ⟹ *s* · σ = *t*
  shows ∃τ. *match-list d P* = *Some* τ ∧ (∀ *x*∈*lvars P*. σ *x* = τ *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* ($\gamma 1$, $\gamma 2$) ⟹ *s* · $\gamma 1$ = *t* · $\gamma 2$
  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 :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** $\leq\cdot$ *50*)
SUBSUMES :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** $<\cdot$ *50*)
LITSIM :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** $\doteq$ *50*)

abbreviation (*input*) *INSTANCEQ* (**infix** $\cdot\geq$ *50*)
where
$x \cdot\geq y \equiv y \leq\cdot x$

abbreviation (*input*) *INSTANCE* (**infix** $\cdot>$ *50*)
where
$x \cdot> y \equiv y <\cdot x$

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

abbreviation *INSTANCE-SET* ($\{\cdot>\}$)
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 $s \leq\cdot\ t$
obtains σ **where** $t = s \cdot \sigma$
using *assms* **by** (*cases*)

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

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

fun *num-syms* :: ($'f$, $'v$) *term* $\Rightarrow$ *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* $\Rightarrow$ *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* $\Rightarrow$ *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* ! $i \cdot \tau$) = *num-syms* (*ts* ! *i*)

and $\forall i < length\ ts.\ num\text{-}funs\ (ts\ !\ i \cdot \tau) = num\text{-}funs\ (ts\ !\ i)$
by *auto*
with *Fun* **and** *IH* **show** *?thesis* **by** *auto*
qed
qed

lemma *subsumeseq-term-size-emv*:
assumes $s \cdot\geq t$ **and** $num\text{-}syms\ s = num\text{-}syms\ t$ **and** $num\text{-}funs\ s = num\text{-}funs\ t$
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 $s = t \cdot \sigma$
shows $vars\text{-}term\ s = (the\text{-}Var \circ \sigma)\ `\ vars\text{-}term\ t$
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 $s = t \cdot \sigma$
shows $num\text{-}unique\text{-}vars\ s \leq num\text{-}unique\text{-}vars\ t$
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 $s \cdot\geq t$
shows $num\text{-}unique\text{-}vars\ s \leq num\text{-}unique\text{-}vars\ t$
using *assms*(*2*) **and** *emv-subst-imp-num-unique-vars-le* [*OF assms*(*1*)] **by** (*cases*) *simp*

lemma *num-syms-geq-num-vars*:
$num\text{-}syms\ t \geq num\text{-}vars\ t$
proof (*induct t*)
case (*Fun f ts*)
with *sum-list-mono* [*of ts num-vars num-syms*]
have $sum\text{-}list\ (map\ num\text{-}vars\ ts) \leq sum\text{-}list\ (map\ num\text{-}syms\ ts)$ **by** *simp*
then show *?case* **by** *simp*
qed *simp*

lemma *num-unique-vars-Fun-Cons*:
$num\text{-}unique\text{-}vars\ (Fun\ f\ (t\ \#\ ts)) \leq num\text{-}unique\text{-}vars\ t + num\text{-}unique\text{-}vars\ (Fun\ f\ ts)$
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 ∃ i. num-unique-vars (f i) ≥ 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) ≥ 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* ·≥ *f* (*Suc i*) **by** (*metis term-subsumable.subsumption.less-imp-le*)
then have $\forall$ *i*. *num-syms* (*f i*) ≥ *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*) ≥ *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*: $\forall$ *i* > *K*. *num-funs* (*g i*) = *num-funs* (*g* (*Suc i*)) **..**
define *M* **where** *M* = *max K N*
have *strict-g*: $\forall$ *i* > *M*. *g i* ·> *g* (*Suc i*) **using** *strict* **by** (*simp add*: *g-def M-def*)
have *g*: $\forall$ *i* > *M*. *g i* ·≥ *g* (*Suc i*) **using** ∗ **by** (*simp add*: *g-def M-def*)
moreover have $\forall$ *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*: $\forall i > M.$ *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*: $\forall i > M.$ *emv* $(g\ i)\ (g\ (Suc\ i))$ **by** (*metis subsume-seq-term-size-emv*)
then have $\forall i > M.$ *num-unique-vars* $(g\ (Suc\ i)) \geq$ *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* $\lambda i.\ g\ (i + Suc\ M)$] **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 $s = t \cdot \sigma$ **by** (*cases*) (*auto simp*: *s-def t-def*)
then have *emv* $s\ t$ **and** *vars-term* $s = (the\text{-}Var \circ \sigma)$ ‘ *vars-term* t
using *emv-subst-vars-term* [*of* $s\ t\ \sigma$] **and** *emv* **and** ‹$i > M$› **by** (*auto simp*: *s-def t-def*)
with *card* **have** *card* $((the\text{-}Var \circ \sigma)$ ‘ *vars-term* $t)$ = *card* (*vars-term* t) **by** *simp*
from *finite-card-eq-imp-bij-betw* [*OF finite-vars-term this*]
have *bij-betw* $(the\text{-}Var \circ \sigma)$ (*vars-term* t) $((the\text{-}Var \circ \sigma)$ ‘ *vars-term* t) **.**

from *bij-betw-extend* [*OF this*, *of UNIV*]
obtain h **where** $*$: $\forall x \in$ *vars-term* $t.\ h\ x = (the\text{-}Var \circ \sigma)\ x$
and *finite* $\{x.\ h\ x \neq x\}$
and *bij* h **by** *auto*
have $\forall x \in$ *vars-term* $t.\ (Var \circ h)\ x = \sigma\ x$
proof
fix x
assume $x \in$ *vars-term* t
with $*$ **have** $h\ x = (the\text{-}Var \circ \sigma)\ x$ **by** *simp*
with *emv-subst-imp-is-Var* [*OF* ‹*emv* $s\ t$› ‹$s = t \cdot \sigma$›] ‹$x \in$ *vars-term* t›
show $(Var \circ h)\ x = \sigma\ x$ **by** *simp*
qed
then have $t \cdot (Var \circ h) = s$
using ‹$s = t \cdot \sigma$› **by** (*auto simp*: *term-subst-eq-conv*)
then have $t \cdot (Var \circ h) \circ_s (Var \circ inv\ h) = s \cdot (Var \circ inv\ h)$ **by** *auto*
then have $t = s \cdot (Var \circ inv\ h)$
unfolding *bij-Var-subst-compose-Var* [*OF* ‹*bij* h›] **by** *simp*
then have $t \cdot\geq s$ **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 $supt\text{-}pred\ s\ t \equiv (s, t) \in supt$
abbreviation $supteq\text{-}pred\ s\ t \equiv (s, t) \in supteq$
abbreviation (*input*) $subt\text{-}pred\ s\ t \equiv supt\text{-}pred\ t\ s$
abbreviation (*input*) $subteq\text{-}pred\ s\ t \equiv supteq\text{-}pred\ t\ s$

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** $\{\lhd\} \equiv \{\rhd\}^{-1}$
abbreviation *subteq* ({⊴}) **where** $\{\trianglelefteq\} \equiv \{\trianglerighteq\}^{-1}$

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
$\forall x{\trianglerighteq}y.\ P \rightharpoonup \forall x.\ x \trianglerighteq y \longrightarrow P$
$\exists x{\trianglerighteq}y.\ P \rightharpoonup \exists x.\ x \trianglerighteq y \wedge P$
$\forall x{\rhd}y.\ P \rightharpoonup \forall x.\ x \rhd y \longrightarrow P$
$\exists x{\rhd}y.\ P \rightharpoonup \exists x.\ x \rhd y \wedge P$

$\forall x{\trianglelefteq}y.\ P \rightharpoonup \forall x.\ x \trianglelefteq y \longrightarrow P$
$\exists x{\trianglelefteq}y.\ P \rightharpoonup \exists x.\ x \trianglelefteq y \wedge P$
$\forall x{\lhd}y.\ P \rightharpoonup \forall x.\ x \lhd y \longrightarrow P$
$\exists x{\lhd}y.\ P \rightharpoonup \exists x.\ x \lhd y \wedge P$

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*]:
$s \unrhd t \Longrightarrow t \unrhd u \Longrightarrow s \unrhd u$
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-list size ss*)
by (*induct ss*) *auto*

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

lemma *size-simp1*:
$t \in$ *set ts* $\Longrightarrow$ *size t* $<$ *Suc* (*size-list size ts*)
by (*induct ts*) *auto*

lemma *size-simp2*:
$t \in$ *set ts* $\Longrightarrow$ *size t* $<$ *Suc* (*Suc* (*size s* + *size-list size ts*))
by (*induct ts*) *auto*

lemma *size-simp3*:
  assumes $(x, y) \in$ *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 $(x, y) \in$ *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*:
  $s \rhd t \Longrightarrow$ *size s* $>$ *size t*
  by (*induct rule*: *supt.induct*) (*auto simp*: *size-simps*)

lemma *supteq-size*:
  $s \trianglerighteq t \Longrightarrow$ *size s* $\geq$ *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*: $s = t \Longrightarrow s \trianglerighteq t$ **by** *auto*

lemma *supt-neqD*: $s \rhd t \Longrightarrow s \neq t$ **using** *supt-size* **by** *auto*

lemma *supteq-Var* [*simp*]:
  $x \in$ *vars-term t* $\Longrightarrow t \trianglerighteq$ *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*: $s \rhd t \Longrightarrow s \unrhd t$
unfolding *supt-supteq-conv* **by** *auto*

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

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

lemma *supteq-antisym*:
assumes $s \unrhd t$ **and** $t \unrhd s$ **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*]:
$s \rhd t \Longrightarrow t \unrhd u \Longrightarrow s \rhd u$
by (*metis subterm.le-less-trans*)

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

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

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

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

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

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

lemma *supteq-imp-vars-term-subset*:
$s \unrhd t \Longrightarrow$ *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 $s \triangleright t$ **shows** $s \cdot \sigma \triangleright t \cdot \sigma$
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
□⟨*s*⟩ = *s* |
(*More f ss1 C ss2*)⟨*s*⟩ = *Fun f* (*ss1* @ *C*⟨*s*⟩ # *ss2*)

lemma *ctxt-eq* [*simp*]:
(*C*⟨*s*⟩ = *C*⟨*t*⟩) = (*s* = *t*) **by** (*induct C*) *auto*

lemma *size-ctxt*: *size t* ≤ *size* (*C*⟨*t*⟩)
by (*induct C*) *simp-all*

lemma *size-ne-ctxt*: *C* ≠ □ ⟹ *size t* < *size* (*C*⟨*t*⟩)
by (*induct C*) *force+*

fun *ctxt-compose* :: (*'f*, *'v*) *ctxt* ⇒ (*'f*, *'v*) *ctxt* ⇒ (*'f*, *'v*) *ctxt* (**infixl** $\circ_c$ *75*)
where
□ $\circ_c$ *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 $\square \circ_c C = C$ **by** *simp*
show $C \circ_c \square = C$ **by** (*induct C*) *simp-all*
qed

instantiation *ctxt* :: (*type*, *type*) *monoid-mult*
begin
definition [*simp*]: $1 = \square$
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
$\square \cdot_c \sigma = \square$ |
(*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 $s = \square\langle s\rangle$ **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** $C \neq \square$ **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* $\square$ *ss2*
have *?C* $\neq \square$ **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** $C \neq \square$ **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* $\neq \square$ **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 $C \neq \square$ **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** $t \trianglerighteq s$
  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*: $(\bigwedge x.\ x \in$ *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*)} ∪ ⋃(*set* (*map funas-term ts*))

lemma *finite-funas-term*:
  finite (*funas-term t*)
  by (*induct t*) *auto*

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 $s \trianglerighteq t$
  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* $\Rightarrow$ $'f$ *sig*
where
funas-ctxt Hole = $\{\}$ |
funas-ctxt (*More f ss1 D ss2*) = $\{(f,$ *Suc* (*length* (*ss1* @ *ss2*)))$\}$
$\cup$ *funas-ctxt D* $\cup \bigcup$(*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

9 Positions (of terms, contexts, etc.)

Positions are just list of natural numbers, and here we define standard notions such as the prefix-relation, parallel positions, left-of, etc. Note that we also instantiate lists in certain ways, such that we can write p^n for the n-fold concatenation of the position p.

theory *Position*
imports
HOL−Library.Infinite-Set
Show.Shows-Literal
begin

type-synonym *pos* = *nat list*

definition *less-eq-pos* :: *pos* $\Rightarrow$ *pos* $\Rightarrow$ *bool* (**infix** $\leq_p$ *50*) **where**
$p \leq_p q \longleftrightarrow (\exists\, r.\ p$ @ $r = q)$

definition *less-pos* :: *pos* $\Rightarrow$ *pos* $\Rightarrow$ *bool* (**infix** $<_p$ *50*) **where**
$p <_p q \longleftrightarrow p \leq_p q \wedge p \neq q$

interpretation *order-pos*: *order less-eq-pos less-pos*
by (*standard*) (*auto simp*: *less-eq-pos-def less-pos-def*)

lemma *Nil-least* [*intro*!, *simp*]:
$[] \leq_p p$
by (*auto simp*: *less-eq-pos-def*)

lemma *less-eq-pos-simps* [*simp*]:
$p \leq_p p @ q$
$p @ q1 \leq_p p @ q2 \longleftrightarrow q1 \leq_p q2$
$i \# q1 \leq_p [] \longleftrightarrow False$
$i \# q1 \leq_p j \# q2 \longleftrightarrow i = j \wedge q1 \leq_p q2$
$p @ q \leq_p p \longleftrightarrow q = []$
$p \leq_p [] \longleftrightarrow p = []$
by (*auto simp*: *less-eq-pos-def*)

lemma *less-eq-pos-code* [*code*]:
$([] :: pos) \leq_p p = True$
$(i \# q1 \leq_p []) = False$
$(i \# q1 \leq_p j \# q2) = (i = j \wedge q1 \leq_p q2)$
by *auto*

lemma *less-pos-simps*[*simp*]:
$(p <_p p @ q) = (q \neq [])$
$(p @ q1 <_p p @ q2) = (q1 <_p q2)$
$(p <_p []) = False$
$(i \# q1 <_p j \# q2) = (i = j \wedge q1 <_p q2)$
$(p @ q <_p p) = False$
by (*auto simp*: *less-pos-def*)

lemma *prefix-smaller* [*simp*]:
assumes $p <_p q$ **shows** $size\ p < size\ q$
using *assms* **by** (*auto simp*: *less-pos-def less-eq-pos-def*)

instantiation *list* :: (*type*) *one*
begin
definition *one-list-def* [*simp*]: $1 = []$
instance by (*intro-classes*)
end

instantiation *list* :: (*type*) *times*
begin
definition *times-list-def* [*simp*]: $times\ p\ q = p @ q$
instance by (*intro-classes*)
end

instantiation *list* :: (*type*) *semigroup-mult*
begin
instance by (*intro-classes*) *simp*
end

instantiation *list* :: (*type*) *power*

begin
  instance by (*intro-classes*)
end

lemma *power-append-distr*:
  $p \hat{}(m + n) = p \hat{} m$ @ $p \hat{} n$
  by (*induct m*) *auto*

lemma *power-pos-Suc*: $p \hat{}$ *Suc n* $= p \hat{} n$ @ p
proof −
  have $p \hat{}$ *Suc n* $= p \hat{}(n +$ *Suc 0*$)$ **by** *simp*
  also have ... $= p \hat{} n$ @ p **unfolding** *power-append-distr* **by** *auto*
  finally show *?thesis* **.**
qed

lemma *power-subtract-less-eq*:
  $p \hat{}(n - m) \leq_p p \hat{} n$
proof (*cases* $m \geq n$)
  case *False*
  then have $(n - m) + m = n$ **by** *auto*
  then show *?thesis* **unfolding** *less-eq-pos-def* **using** *power-append-distr* **by** *metis*
qed *simp*

lemma *power-size*: **fixes** p :: *pos* **shows** *size* $(p \hat{} n) =$ *size* $p * n$
 by (*induct n*, *simp*, *auto*)

fun *remove-prefix* :: *'a list* ⇒ *'a list* ⇒ *'a list option*
  where
    remove-prefix [] *ys* = *Some ys*
  | *remove-prefix* (*x*#*xs*) (*y*#*ys*) = (*if x* = *y then remove-prefix xs ys else None*)
  | *remove-prefix xs ys* = *None*

lemma *remove-prefix* [*simp*]:
  remove-prefix (*x* # *xs*) *ys* =
    (*case ys of*
      [] ⇒ *None*
    | *z* # *zs* ⇒ *if x* = *z then remove-prefix xs zs else None*)
  by (*cases ys*) *auto*

lemma *remove-prefix-Some* [*simp*]:
  remove-prefix xs ys = *Some zs* ⟷ *ys* = *xs* @ *zs*
  by (*induct xs ys rule*: *remove-prefix.induct*) (*auto*)

lemma *remove-prefix-append* [*simp*]:
  remove-prefix xs (*xs* @ *ys*) = *Some ys*
  by *simp*

lemma *less-eq-pos-remove-prefix*:
  assumes $p \leq_p q$

obtains r **where** $q = p$ @ r **and** *remove-prefix* p q = *Some* r
using *assms* **by** (*induct p arbitrary*: *q*) (*auto simp*: *less-eq-pos-def*)

lemma *suffix-exists*:
assumes $p \leq_p q$
shows $\exists r.$ p @ $r = q \wedge$ *remove-prefix* p q = *Some* r
using *assms* **by** (*elim less-eq-pos-remove-prefix*) *auto*

fun *remove-suffix* :: *'a list* ⇒ *'a list* ⇒ *'a list option*
where
remove-suffix p q =
(*case remove-prefix* (*rev p*) (*rev q*) *of*
None ⇒ *None*
| *Some r* ⇒ *Some* (*rev r*))

lemma *remove-suffix-Some* [*simp*]:
remove-suffix xs ys = *Some zs* ⟷ *ys* = *zs* @ *xs*
by (*auto split*: *option.splits*) (*metis rev-append rev-rev-ident*)

lemma *Nil-power* [*simp*]: [] ^ n = [] **by** (*induct n*) *auto*

fun *parallel-pos* :: *pos* ⇒ *pos* ⇒ *bool* (**infixr** ⊥ *64*)
where
[] ⊥ - ⟷ *False*
| - ⊥ [] ⟷ *False*
| i # p ⊥ j # q ⟷ $i \neq j \vee p \perp q$

lemma *parallel-pos*: $p \perp q = (\neg\, p \leq_p q \wedge \neg\, q \leq_p p)$
by (*induct p q rule*: *parallel-pos.induct*) *auto*

lemma *parallel-remove-prefix*: *p1* ⊥ *p2* ⟹
∃ *p i j q1 q2*. *p1* = *p* @ *i* # *q1* ∧ *p2* = *p* @ *j* # *q2* ∧ $i \neq j$
proof (*induct p1 p2 rule*: *parallel-pos.induct*)
case (*3 i p j q*)
then show *?case* **by** *simp* (*metis Cons-eq-append-conv*)
qed *auto*

lemma *pos-cases*: $p \leq_p q \vee q <_p p \vee p \perp q$
by (*induct p q rule*: *parallel-pos.induct*)
(*auto simp*: *less-pos-def*)

lemma *parallel-pos-sym*: *p1* ⊥ *p2* ⟹ *p2* ⊥ *p1*
unfolding *parallel-pos* **by** *auto*

lemma *less-pos-def'*: $(p <_p q) = (\exists\ r.\ q = p$ @ $r \wedge r \neq [])$ (**is** *?l* = *?r*)
by (*auto simp*: *less-pos-def less-eq-pos-def*)

lemma *pos-append-cases*:
p1 @ *p2* = *q1* @ *q2* ⟹

$(\exists\ q3.\ p1 = q1\ @\ q3 \wedge q2 = q3\ @\ p2) \vee$
$(\exists\ p3.\ q1 = p1\ @\ p3 \wedge p2 = p3\ @\ q2)$
proof (*induct p1 arbitrary*: *q1*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons i p1′ q1*) **note** *IH* = *this*
show *?case*
proof (*cases q1*)
case *Nil*
then show *?thesis* **using** *IH*(*2*) **by** *auto*
next
case (*Cons j q1′*)
with *IH*(*2*) **have** *id*: $p1' @ p2 = q1' @ q2$ **and** *ij*: $i = j$ **by** *auto*
from *IH*(*1*)[*OF id*]
show *?thesis* **unfolding** *Cons ij* **by** *auto*
qed
qed

lemma *pos-less-eq-append-not-parallel*:
assumes $q \leq_p p\ @\ q'$
shows $\neg\ (q \perp p)$
proof−
from *assms* **obtain** *r* **where** $q\ @\ r = p\ @\ q'$ **unfolding** *less-eq-pos-def* **..**
then have *dec*:$(\exists\ q3.\ q = p\ @\ q3 \wedge q' = q3\ @\ r) \vee$
$(\exists\ p3.\ p = q\ @\ p3 \wedge r = p3\ @\ q')$ (**is** *?a* ∨ *?b*) **by** (*rule pos-append-cases*)
then have $p \leq_p q \vee q \leq_p p$ **unfolding** *less-eq-pos-def* **by** *blast*
then show *?thesis* **unfolding** *parallel-pos* **by** *auto*
qed

lemma *less-pos-power-split*: $q <_p p\ \hat{}\ m \Longrightarrow \exists\ p'\ k.\ q = p\ \hat{}\ k\ @\ p' \wedge p' <_p p \wedge$
$k < m$
proof (*induct m arbitrary*: *q*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n q*)
show *?case*
proof (*cases* $q <_p p$)
case *True*
show *?thesis*
by (*rule exI*[*of - q*], *rule exI*[*of - 0*], *insert True*, *auto*)
next
case *False*
from *Suc*(*2*) **obtain** *r* **where** *pn*: $p\ @\ p\hat{}n = q\ @\ r$ **unfolding** *less-pos-def′*
by *auto*
from *pos-append-cases*[*OF this*]
have $\exists\ r.\ q = p\ @\ r$
proof

```
      assume ∃ s. p = q @ s ∧ r = s @ p ^ n
      then obtain s where p: p = q @ s by auto
      with False show ?thesis by auto
    qed auto
    then obtain r where q: q = p @ r by auto
    with Suc(2) have r <p p ^ n by simp
    from Suc(1)[OF this] obtain p' k where r: r = p ^ k @ p' p' <p p k < n by
auto
    show ?thesis unfolding q
      by (rule exI[of - p'], rule exI[of - Suc k], insert r, auto)
  qed
qed

definition showsl-pos :: pos ⇒ showsl where
  showsl-pos = showsl-list-gen (λ i. showsl (Suc i)) (STR ''empty'') (STR '''') (STR
''.'') (STR '''')

fun proper-prefix-list :: pos ⇒ pos list
  where
    proper-prefix-list [] = [] |
    proper-prefix-list (i # p) = [] # map (Cons i) (proper-prefix-list p)

lemma proper-prefix-list [simp]: set (proper-prefix-list p) = {q. q <p p}
proof (induction p)
  case (Cons i p)
  note IH = this
  show ?case (is ?l = ?r)
  proof (rule set-eqI)
    fix q
    show q ∈ ?l = (q ∈ ?r)
    proof (cases q)
      case Nil
      have less: [] <p i # p unfolding less-pos-def by auto
      show ?thesis unfolding Nil using less by auto
    next
      case (Cons j q')
      show ?thesis unfolding Cons by (auto simp: IH)
    qed
  qed
qed simp

definition prefix-list :: pos ⇒ pos list
  where
    prefix-list p = p # proper-prefix-list p

lemma prefix-list [simp]: set (prefix-list p) = { q. q ≤p p}
  by (auto simp: prefix-list-def)

definition bounded-postfixes :: pos ⇒ pos list ⇒ pos list
```

where
bounded-postfixes p ps = *map the* [*opt*←*map* (*remove-prefix p*) *ps* . *opt* ≠ *None*]

lemma *bounded-postfixes* [*simp*]:
set (*bounded-postfixes p ps*) = { *r*. *p* @ *r* ∈ *set ps*} (**is** *?l* = *?r*)
by (*auto simp*: *bounded-postfixes-def*)
(*metis* (*mono-tags*, *lifting*) *image-eqI mem-Collect-eq option.sel remove-prefix-append*)

definition *left-of-pos* :: *pos* ⇒ *pos* ⇒ *bool*
where
left-of-pos p q = (∃ *r i j*. *r* @ [*i*] $\leq_p$ *p* ∧ *r* @ [*j*] $\leq_p$ *q* ∧ *i* < *j*)

lemma *left-of-pos-append*:
left-of-pos p q ⟹ *left-of-pos* (*p* @ *p*′) (*q* @ *q*′)
apply (*simp add*: *left-of-pos-def*)
using *less-eq-pos-simps*(*1*) *order-pos.order.trans* **by** *blast*

lemma *append-left-of-pos*:
left-of-pos p q = *left-of-pos* (*p*′ @ *p*) (*p*′ @ *q*)
proof (*rule iffI*)
assume *left-of-pos p q*
then show *left-of-pos* (*p*′ @ *p*) (*p*′ @ *q*)
unfolding *left-of-pos-def* **by** (*metis less-eq-pos-simps*(*2*) *append-assoc*)
next
assume *left-of-pos* (*p*′ @ *p*) (*p*′ @ *q*)
then show *left-of-pos p q*
proof (*induct p*′ *arbitrary*: *p q rule*:*rev-induct*)
case (*snoc a p*′)
then have *IH*:*left-of-pos* (*p*′ @ *p*) (*p*′ @ *q*) ⟹ *left-of-pos p q* **and**
left:*left-of-pos* ((*p*′ @ [*a*]) @ *p*) ((*p*′ @ [*a*]) @ *q*) **by** *auto*
from *left*[*unfolded left-of-pos-def*] **have** *left-of-pos* (*p*′ @ (*a* # *p*)) (*p*′ @ (*a* # *q*))
by (*metis append-assoc append-Cons append.left-neutral snoc.prems*)
with *IH* **have** *left-of-pos* (*a* # *p*) (*a* # *q*) **unfolding** *left-of-pos-def* **by** (*metis left-of-pos-def snoc.hyps*)
then obtain *r i j r*′ *r*′′ **where** *x*:*r* @ [*i*] @ *r*′ = *a* # *p* **and** *y*:(*r* @ [*j*]) @ *r*′′ = *a* # *q*
and *ij*:*i* < *j* **unfolding** *left-of-pos-def less-eq-pos-def* **by** *auto*
then have [] $<_p$ *r* **unfolding** *less-pos-def*′
by (*metis append-Nil append-Cons not-less-iff-gr-or-eq list.inject*)
with *x* **obtain** *rr* **where** *r* = *a* # *rr* **using** *list.exhaust*[*of r*]
by (*metis less-eq-pos-simps*(*1*) *less-eq-pos-simps*(*4*) *less-pos-simps*(*1*) *append.left-neutral*)
with *x y* **have** *rr* @ [*i*] @ *r*′ = *p* **and** *y*:(*rr* @ [*j*]) @ *r*′′ = *q* **by** *auto*
with *ij* **show** *?case* **unfolding** *left-of-pos-def less-eq-pos-def* **by** *auto*
qed *simp*
qed

lemma *left-pos-parallel*: *left-of-pos p q* ⟹ *q* ⊥ *p* **unfolding** *left-of-pos-def*

proof −
  assume $\exists\, r\ i\ j.\ r\ @\ [i] \leq_p p \wedge r\ @\ [j] \leq_p q \wedge i < j$
  then obtain *r i j* **where** *rp*:$r\ @\ [i] \leq_p p$ **and** *rq*:$r\ @\ [j] \leq_p q$ **and** *ij*:$i < j$ **by** *auto*
  from *rp* **obtain** p' **where** *rp*:$p = r\ @\ i\ \#\ p'$ **unfolding** *less-eq-pos-def* **by** *auto*
  from *rq* **obtain** q' **where** *rq*:$q = (r\ @\ (j\ \#\ q'))$ **unfolding** *less-eq-pos-def* **by** *auto*
  from *rp rq ij* **have** *pq*:$\neg\ p \leq_p q$ **by** *force*
  from *rp rq ij* **have** $\neg\ q \leq_p p$ **by** *force*
  with *pq* **show** *?thesis* **using** *parallel-pos* **by** *auto*
qed

lemma *left-of-append-cases*: *left-of-pos* $(p0\ @\ p1)\ q \Longrightarrow p0 <_p q \vee$ *left-of-pos* $p0\ q$
proof −
  assume *left-of-pos* $(p0\ @\ p1)\ q$
  then obtain *r i j* **where** *rp*:$r\ @\ [i] \leq_p (p0\ @\ p1)$ **and** *rq*:$r\ @\ [j] \leq_p q$ **and** *ij*:$i < j$
    unfolding *left-of-pos-def* **by** *auto*
  show *?thesis* **proof**(*cases* $p0 \leq_p r$)
    case *True*
    with *rq* **have** $p0 <_p q$
      by (*metis less-eq-pos-simps*(*1*) *less-eq-pos-simps*(*5*) *less-pos-def list.simps*(*3*) *order-pos.order.trans*)
    then show *?thesis* **by** *auto*
  next
    case *False*
    then have *aux*:$\neg\ (\exists\ r'.\ p0\ @\ r' = r)$ **unfolding** *less-eq-pos-def* **by** *auto*
    from *rp* **have** *par*:$\neg\ (r\ @\ [i] \perp p0)$ **using** *pos-less-eq-append-not-parallel* **by** *auto*
    from *aux* **have** *a*:$\neg\ (p0 \leq_p r)$ **unfolding** *less-eq-pos-def* **by** *auto*
    from *rp* **have** $\neg\ (p0 \perp r)$
    using *less-eq-pos-simps*(*1*) *order-pos.order.trans parallel-pos pos-less-eq-append-not-parallel* **by** *blast*
    with *a* **have** $r <_p p0$ **using** *pos-cases* **by** *auto*
    then obtain *oo* **where** *p0*:$p0 = r\ @\ oo$ **and** $[] <_p oo$ **unfolding** *less-pos-def less-eq-pos-def* **by** *auto*
    have $\neg\ (p0 <_p r\ @\ [i])$ **unfolding** *less-pos-def less-eq-pos-def*
      by (*metis aux butlast-append butlast-snoc self-append-conv*)
    with *par* **have** $r\ @\ [i] \leq_p p0$ **using** *pos-cases* **by** *auto*
   with *ij this*[*unfolded less-eq-pos-def*] **have** *left-of-pos p0 q* **unfolding** *left-of-pos-def* **using** *rq* **by** *auto*
    then show *?thesis* **by** *auto*
  qed
qed

lemma *append-left-of-cases*:
  assumes *left*: *left-of-pos q* $(p0\ @\ p1)$
  shows $p0 <_p q \vee$ *left-of-pos q p0*

proof −
from *left* **obtain** *r i j* **where** *rp*:*r* @ [*i*] $\leq_p$ *q* **and** *rq*:*r* @ [*j*] $\leq_p$ (*p0* @ *p1*) **and** *ij*:*i* < *j*
unfolding *left-of-pos-def* **by** *auto*
show *?thesis* **proof**(*cases p0* $\leq_p$ *r*)
case *True*
with *rp* **have** *p0* $<_p$ *q* **unfolding** *less-pos-def*
by (*meson less-eq-pos-simps*(*1*) *less-eq-pos-simps*(*5*) *list.simps*(*3*) *order-pos.order.trans*)
then show *?thesis* **by** *auto*
next
case *False*
then have *aux*:¬ (∃ *r′*. *p0* @ *r′* = *r*) **unfolding** *less-eq-pos-def* **by** *auto*
from *rp rq* **have** *par*:¬ (*r* @ [*j*] ⊥ *p0*) **using** *pos-less-eq-append-not-parallel* **by** *auto*
from *aux* **have** *a*:¬ (*p0* $\leq_p$ *r*) **unfolding** *less-eq-pos-def* **by** *auto*
from *rq* **have** ¬ (*p0* ⊥ *r*)
using *less-eq-pos-simps*(*1*) *order-pos.order.trans parallel-pos pos-less-eq-append-not-parallel* **by** *blast*
with *a* **have** *r* $<_p$ *p0* **using** *pos-cases* **by** *auto*
then obtain *oo* **where** *p0*:*p0* = *r* @ *oo* **and** [] $<_p$ *oo* **unfolding** *less-pos-def less-eq-pos-def* **by** *auto*
have ¬ (*p0* $<_p$ *r* @ [*j*]) **unfolding** *less-pos-def less-eq-pos-def* **using** *p0 a list.exhaust*[*of p0*]
by (*metis append-Nil2 aux butlast-append butlast-snoc*)
with *par* **have** *r* @ [*j*] $\leq_p$ *p0* **using** *pos-cases* **by** *auto*
with *ij this*[*unfolded less-eq-pos-def*] **have** *left-of-pos q p0* **unfolding** *left-of-pos-def* **using** *rp* **by** *auto*
then show *?thesis* **by** *auto*
qed
qed

lemma *parallel-imp-right-or-left-of*:
assumes *par*:*p* ⊥ *q* **shows** *left-of-pos p q* ∨ *left-of-pos q p*
proof −
from *parallel-remove-prefix*[*OF par*] **obtain** *r i j p′ q′* **where** *p* = *r* @ *i*# *p′* **and** *q* = *r* @ *j* # *q′*
and *ij*:*i* ≠ *j* **by** *blast*
then have *r* @ [*i*] $\leq_p$ *p* ∧ *r* @ [*j*] $\leq_p$ *q* **by** *simp*
then show *?thesis* **unfolding** *left-of-pos-def* **using** *ij less-linear* **by** *blast*
qed

lemma *left-of-imp-not-right-of*:
assumes *l*:*left-of-pos p q* **shows** ¬ *left-of-pos q p*
proof
assume *l′*:*left-of-pos q p*
from *l* **obtain** *r i j* **where** *r* @ [*i*] $\leq_p$ *p* **and** *ij*:*i* < *j* **and** *r* @ [*j*] $\leq_p$ *q* **unfolding** *left-of-pos-def* **by** *blast*
then obtain *p0 q0* **where** *p*:*p* = (*r* @ [*i*]) @ *p0* **and** *q*:*q* = (*r* @ [*j*]) @ *q0* **unfolding** *less-eq-pos-def* **by** *auto*

from l' **obtain** r' i' j' **where** r' @ $[j'] \leq_p p$ **and** ij':$i' < j'$ **and** r' @ $[i'] \leq_p q$ **unfolding** *left-of-pos-def* **by** *blast*
then obtain $p0'$ $q0'$ **where** p':$p = (r'$ @ $[j'])$ @ $p0'$ **and** q':$q = (r'$ @ $[i'])$ @ $q0'$ **unfolding** *less-eq-pos-def* **by** *auto*
from p p' **have** p:r @ $(i \# p0) = r'$ @ $(j' \# p0')$ **by** *auto*
from q q' **have** q:r @ $(j \# q0) = r'$ @ $(i' \# q0')$ **by** *auto*
with p ij ij' **have** ne:$r \neq r'$ **using** *same-append-eq*[*of r*] **by** (*metis less-imp-not-less list.inject*)
have nlt:$\neg$ $r <_p r'$ **proof**
assume $r <_p r'$
then obtain $r2$ **where** $r1$:$r' = r$ @ $r2$ **and** $r2$:$r2 \neq []$ **unfolding** *less-pos-def less-eq-pos-def* **by** *auto*
from p **have** p':$i \# p0 = r2$ @ $j' \# p0'$ **unfolding** *r1 append-assoc* **using** *less-eq-pos-simps*(*2*) **by** *auto*
from q **have** q':$j \# q0 = r2$ @ $i' \# q0'$ **unfolding** *r1 append-assoc* **using** *less-eq-pos-simps*(*2*) **by** *auto*
from $r2$ **obtain** k rr **where** $r2$:$r2 = k\# rr$ **by** (*cases r2*, *auto*)
from p' q' *ij list.inject* **show** *False* **unfolding** $r2$ **by** *simp*
qed
have $\neg$ $r' <_p r$ **proof**
assume $r' <_p r$
then obtain $r2$ **where** $r1$:$r = r'$ @ $r2$ **and** $r2$:$r2 \neq []$ **unfolding** *less-pos-def less-eq-pos-def* **by** *auto*
from p **have** p':$r2$ @ $i \# p0 = j' \# p0'$ **unfolding** *r1 append-assoc* **using** *less-eq-pos-simps*(*2*) **by** *auto*
from q **have** q':$r2$ @ $j \# q0 = i' \# q0'$ **unfolding** *r1 append-assoc* **using** *less-eq-pos-simps*(*2*) **by** *auto*
from $r2$ **obtain** k rr **where** $r2$:$r2 = k\# rr$ **by** (*cases r2*, *auto*)
from p' q' ij' *list.inject* **show** *False* **unfolding** $r2$ **by** *simp*
qed
with *nlt ne* **have** $r \perp r'$ **by** (*auto simp*: *parallel-pos less-pos-def*)
with p q **show** *False* **by** (*metis less-eq-pos-simps*(*1*) *pos-less-eq-append-not-parallel*)
qed

primrec *is-left-of* :: *pos* $\Rightarrow$ *pos* $\Rightarrow$ *bool*
where
left-Nil: *is-left-of* [] q = *False*
| *left-Cons*: *is-left-of* $(i \# p)$ q =
(*case* q *of*
[] $\Rightarrow$ *False*
| $j \# q' \Rightarrow$ *if* $i < j$ *then True else if* $i > j$ *then False else is-left-of* p q')

lemma *is-left-of*: *is-left-of* p q = *left-of-pos* p q
proof
assume l:*is-left-of* p q
then show *left-of-pos* p q
proof (*induct* p *arbitrary*:q)
case *Nil*
with *left-Nil* **show** *?case* **by** *auto*

```
  next
    case (Cons i p) note IH = this
    assume l:is-left-of (i # p) q
    show ?case
    proof (cases q)
      case Nil
      with l show ?thesis unfolding left-Cons by auto
    next
      case (Cons j q')
      show ?thesis
      proof (cases ¬ (i < j), cases j < i)
        case True
        with l Cons show ?thesis unfolding left-Cons by auto
      next
        assume ¬ j < i and ¬ i < j
        then have ij:i = j by auto
        with Cons l have is-left-of p q' unfolding left-Cons by auto
        with IH have left-of-pos p q' by blast
        with ij show left-of-pos (i # p) q unfolding Cons left-of-pos-def
          by (metis append-Cons less-eq-pos-simps(4))
      next
        assume ij:¬ ¬ (i < j)
       then have [] @ [i] ≤p (i # p) ∧ [] @ [j] ≤p (j # q') unfolding less-eq-pos-def
by auto
        with Cons ij show ?thesis unfolding left-of-pos-def by blast
      qed
    qed
  qed
next
  assume l:left-of-pos p q
  from this[unfolded left-of-pos-def] obtain r i j where r @[i] ≤p p and r @ [j]
≤p q
    and ij:i < j by blast
  then obtain p' q' where p = (r @[i]) @ p' and q = (r @[j]) @ q' unfolding
less-eq-pos-def
    by auto
  then show is-left-of p q
  proof (induct r arbitrary: p q p' q')
    case Nil
    assume p:p = ([] @ [i]) @ p' and q:q = ([] @ [j]) @ q'
    with l[unfolded p q append-Nil] show ?case using left-Cons ij by force
  next
    case (Cons k r') note IH = this
    assume p:p = ((k # r') @ [i]) @ p' and q:q = ((k # r') @ [j]) @ q'
      from ij have left-of-pos ((r' @ [i]) @ p') (( r' @ [j]) @ q') unfolding
left-of-pos-def
      by (metis less-eq-pos-def)
    with IH have is-left-of ((r' @ [i]) @ p') (( r' @ [j]) @ q') by auto
    then show is-left-of p q unfolding p q  using left-Cons by force
```

```
  qed
qed

abbreviation right-of-pos :: pos ⇒ pos ⇒ bool
where
  right-of-pos p q ≡ left-of-pos q p

lemma remove-prefix-same [simp]:
  remove-prefix p p = Some []
  by (induct p) simp-all

definition pos-diff p q = the (remove-prefix q p)

lemma prefix-pos-diff [simp]:
  assumes p ≤p q
  shows p @ pos-diff q p = q
  using suffix-exists [OF assms] by (auto simp: pos-diff-def)

lemma pos-diff-Nil2 [simp]:
  pos-diff p [] = p
  by (auto simp: pos-diff-def)

lemma inj-nat-to-pos: inj (rec-nat [] Cons) (is inj ?f)
  unfolding inj-on-def
proof (intro ballI impI)
  fix x y
  show ?f x = ?f y ⟹ x = y
  proof (induct x arbitrary: y)
    case 0
    then show ?case by (cases y, auto)
  next
    case (Suc x sy)
    then obtain y where sy: sy = Suc y by (cases sy, auto)
    from Suc(2)[unfolded sy] have id: ?f x = ?f y by auto
    from Suc(1)[OF this] sy show ?case by simp
  qed
qed

lemma infinite-UNIV-pos[simp]: infinite (UNIV :: pos set)
proof
  assume finite (UNIV :: pos set)
  from finite-subset[OF - this, of range (rec-nat [] Cons)]
    range-inj-infinite[OF inj-nat-to-pos]
  show False by blast
qed

lemma less-pos-right-mono:
  p @ q <p r @ q ⟹ p <p r
proof (induct q rule: rev-induct)
```

case (*snoc x xs*)
thus *?case*
by (*simp add*: *less-pos-def less-eq-pos-def*)
(*metis append-is-Nil-conv butlast-append butlast-snoc list.simps*(*3*))
qed *auto*

lemma *less-pos-left-mono*:
p @ $q <_p p$ @ $r \Longrightarrow q <_p r$
by *auto*

end

10 More Results on Terms

In this theory we introduce many more concepts of terms, we provide several results that link various notions, e.g., positions, subterms, contexts, substitutions, etc.

theory *Term-More*
imports
Position
Subterm-and-Context
Polynomial-Factorization.Missing-List
begin

showl-Instance for Terms

fun *showsl-term′* :: (*′f* ⇒ *showsl*) ⇒ (*′v* ⇒ *showsl*) ⇒ (*′f*, *′v*) *term* ⇒ *showsl*
where
showsl-term′ fun var (*Var x*) = *var x* |
showsl-term′ fun var (*Fun f ts*) =
fun f ∘ *showsl-list-gen id* (*STR ′′′′*) (*STR ′′(′′*) (*STR ′′, ′′*) (*STR ′′)′′*) (*map* (*showsl-term′ fun var*) *ts*)

abbreviation *showsl-nat-var* :: *nat* ⇒ *showsl*
where
showsl-nat-var i ≡ *showsl-lit* (*STR ′′x′′*) ∘ *showsl i*

abbreviation *showsl-nat-term* :: (*′f*::*showl*, *nat*) *term* ⇒ *showsl*
where
showsl-nat-term ≡ *showsl-term′ showsl showsl-nat-var*

instantiation *term* :: (*showl*,*showl*)*showl*
begin
definition *showsl* (*t* :: (*′a*,*′b*)*term*) = *showsl-term′ showsl showsl t*
definition *showsl-list* (*xs* :: (*′a*,*′b*)*term list*) = *default-showsl-list showsl xs*
instance ..
end

showl-Instance for Contexts

fun *showsl-ctxt'* :: ('f ⇒ *showsl*) ⇒ ('v ⇒ *showsl*) ⇒ ('f, 'v) *ctxt* ⇒ *showsl* **where**
  showsl-ctxt' fun var (*Hole*) = *showsl-lit* (*STR* ''[]'')
| *showsl-ctxt' fun var* (*More f ss1 D ss2*) = (
    fun f ∘ *showsl* (*STR* ''('') ∘
    showsl-list-gen (*showsl-term' fun var*) (*STR* '''') (*STR* '''') (*STR* '', '') (*STR* '', '') *ss1* ∘
    showsl-ctxt' fun var D ∘
    showsl-list-gen (*showsl-term' fun var*) (*STR* '')'') (*STR* '', '') (*STR* '', '') (*STR* '')'') *ss2*
)

instantiation *ctxt* :: (*showl*,*showl*)*showl*
begin
definition *showsl* (*t* :: ('a,'b)*ctxt*) = *showsl-ctxt' showsl showsl t*
definition *showsl-list* (*xs* :: ('a,'b)*ctxt list*) = *default-showsl-list showsl xs*
instance ..
end

General Folds on Terms

context
begin
qualified fun
  fold :: ('v ⇒ 'a) ⇒ ('f ⇒ 'a *list* ⇒ 'a) ⇒ ('f, 'v) *term* ⇒ 'a
  where
    fold var fun (*Var x*) = *var x* |
    fold var fun (*Fun f ts*) = *fun f* (*map* (*fold var fun*) *ts*)
end

declare *term.disc* [*intro*]

abbreviation *num-args t* ≡ *length* (*args t*)

definition *funas-args-term* :: ('f, 'v) *term* ⇒ 'f *sig*
  where
    funas-args-term t = ⋃(*set* (*map funas-term* (*args t*)))

fun *eroot* :: ('f, 'v) *term* ⇒ ('f × *nat*) + 'v
  where
    eroot (*Fun f ts*) = *Inl* (*f*, *length ts*) |
    eroot (*Var x*) = *Inr x*

abbreviation *root-set* ≡ *set-option* ∘ *root*

lemma *funas-term-conv*:
  funas-term t = *set-option* (*root t*) ∪ *funas-args-term t*
  by (*cases t*) (*simp-all add*: *funas-args-term-def*)

The *depth* of a term is defined as follows:

fun *depth* :: ('f, 'v) *term* ⇒ *nat*

where
depth (*Var* -) = *0* |
depth (*Fun* - []) = *0* |
depth (*Fun* - *ts*) = *1* + *Max* (*set* (*map depth ts*))

declare *conj-cong* [*fundef-cong*]

The positions of a term

fun *poss* :: ('*f*, '*v*) *term* ⇒ *pos set* **where**
poss (*Var x*) = {[]} |
poss (*Fun f ss*) = {[]} ∪ {*i* # *p* | *i p*. *i* < *length ss* ∧ *p* ∈ *poss* (*ss* ! *i*)}
declare *conj-cong* [*fundef-cong del*]

lemma *Cons-poss-Var* [*simp*]:
i # *p* ∈ *poss* (*Var x*) ⟷ *False*
by *simp*

lemma *elem-size-size-list-size* [*termination-simp*]:
x ∈ *set xs* ⟹ *size x* < *size-list size xs*
by (*induct xs*) *auto*

The set of function positions of a term

fun *fun-poss* :: ('*f*, '*v*) *term* ⇒ *pos set*
where
fun-poss (*Var x*) = {} |
fun-poss (*Fun f ts*) = {[]} ∪ (⋃*i*<*length ts*. {*i* # *p* | *p*. *p* ∈ *fun-poss* (*ts* ! *i*)})

lemma *fun-poss-imp-poss*:
assumes *p* ∈ *fun-poss t*
shows *p* ∈ *poss t*
using *assms* **by** (*induct t arbitrary*: *p*) *auto*

lemma *finite-fun-poss*:
finite (*fun-poss t*)
by (*induct t*) *auto*

The set of variable positions of a term

fun *var-poss* :: ('*f*, '*v*) *term* ⇒ *pos set*
where
var-poss (*Var x*) = {[]} |
var-poss (*Fun f ts*) = (⋃*i*<*length ts*. {*i* # *p* | *p*. *p* ∈ *var-poss* (*ts* ! *i*)})

lemma *var-poss-imp-poss*:
assumes *p* ∈ *var-poss t*
shows *p* ∈ *poss t*
using *assms* **by** (*induct t arbitrary*: *p*) *auto*

lemma *finite-var-poss*:
finite (*var-poss t*)

by (*induct t*) *auto*

lemma *poss-simps* [*symmetric*, *simp*]:
poss t = *fun-poss t* ∪ *var-poss t*
poss t = *var-poss t* ∪ *fun-poss t*
fun-poss t = *poss t* − *var-poss t*
var-poss t = *poss t* − *fun-poss t*
by (*induct-tac* [!] *t*) *auto*

lemma *finite-poss* [*simp*]:
finite (*poss t*)
by (*subst poss-simps* [*symmetric*]) (*metis finite-Un finite-fun-poss finite-var-poss*)

The subterm of a term s at position p is defined as follows:

fun *subt-at* :: ('*f*, '*v*) *term* ⇒ *pos* ⇒ ('*f*, '*v*) *term* (**infixl** |'- *67*)
where
s |- [] = *s* |
Fun f ss |- (*i* # *p*) = (*ss* ! *i*) |- *p*

lemma *var-poss-iff*:
p ∈ *var-poss t* ⟷ (∃ *x*. *p* ∈ *poss t* ∧ *t* |- *p* = *Var x*)
by (*induct t arbitrary*: *p*) *auto*

lemma *fun-poss-fun-conv*:
assumes *p* ∈ *fun-poss t*
shows ∃ *f ts*. *t* |- *p* = *Fun f ts*
proof (*cases t* |- *p*)
case (*Var x*)
have *p-in-t*: *p* ∈ *poss t* **using** *fun-poss-imp-poss*[*OF assms*]**.**
then have *p* ∈ *poss t* − *fun-poss t* **using** *Var*(*1*) *var-poss-iff* **by** *auto*
then show *?thesis* **using** *assms* **by** *blast*
next
case (*Fun f ts*) **then show** *?thesis* **by** *auto*
qed

lemma *pos-append-poss*:
p ∈ *poss t* ⟹ *q* ∈ *poss* (*t* |- *p*) ⟹ *p* @ *q* ∈ *poss t*
proof (*induct t arbitrary*: *p q*)
case (*Fun f ts p q*)
show *?case*
proof (*cases p*)
case *Nil* **then show** *?thesis* **using** *Fun* **by** *auto*
next
case (*Cons i p'*)
with *Fun* **have** *i*: *i* < *length ts* **and** *p'*: *p'* ∈ *poss* (*ts* ! *i*) **by** *auto*
then have *mem*: *ts* ! *i* ∈ *set ts* **by** *auto*
from *Fun*(*3*) **have** *q* ∈ *poss* (*ts* ! *i* |- *p'*) **by** (*auto simp*: *Cons*)
from *Fun*(*1*) [*OF mem p' this*]
show *?thesis* **by** (*auto simp*: *Cons i*)

```
  qed
qed simp
```

Creating a context from a term by adding a hole at a specific position.

```
fun
  ctxt-of-pos-term :: pos ⇒ ('f, 'v) term ⇒ ('f, 'v) ctxt
  where
    ctxt-of-pos-term [] t = □ |
    ctxt-of-pos-term (i # ps) (Fun f ts) =
    More f (take i ts) (ctxt-of-pos-term ps (ts!i)) (drop (Suc i) ts)

lemma ctxt-supt-id:
  assumes p ∈ poss t
  shows (ctxt-of-pos-term p t)⟨t |- p⟩ = t
  using assms by (induct t arbitrary: p) (auto simp: id-take-nth-drop [symmetric])
```

Let s and t be terms. The following three statements are equivalent:

1. $s \trianglerighteq t$
2. $\exists\, p{\in}poss\ s.\ s|\text{-}p = t$
3. $\exists\, C.\ s = C\langle t\rangle$

The position of the hole in a context is uniquely determined.

```
fun
  hole-pos :: ('f, 'v) ctxt ⇒ pos
  where
    hole-pos □ = [] |
    hole-pos (More f ss D ts) = length ss # hole-pos D

lemma hole-pos-ctxt-of-pos-term [simp]:
  assumes p ∈ poss t
  shows hole-pos (ctxt-of-pos-term p t) = p
  using assms
proof (induct t arbitrary: p)
  case (Fun f ts)
  show ?case
  proof (cases p)
    case Nil
    then show ?thesis using Fun by auto
  next
    case (Cons i q)
    with Fun(2) have i: i < length ts and q: q ∈ poss (ts ! i) by auto
    then have ts ! i ∈ set ts by auto
    from Fun(1)[OF this q] Cons i show ?thesis by simp
  qed
qed simp
```

```
lemma hole-pos-id-ctxt:
  assumes C⟨s⟩ = t
  shows ctxt-of-pos-term (hole-pos C) t = C
  using assms
proof (induct C arbitrary: t)
  case (More f bef C aft)
  then show ?case
  proof (cases t)
    case (Fun g ts)
    with More have [simp]: g = f by simp
    from Fun More have bef: take (length bef) ts = bef by auto
    from Fun More have aft: drop (Suc (length bef)) ts = aft by auto
    from Fun More have Cs: C⟨s⟩ = ts ! length bef by auto
    from Fun More show ?thesis by (simp add: bef aft More(1)[OF Cs])
  qed simp
qed simp

lemma supteq-imp-subt-at:
  assumes s ⊵ t
  shows ∃ p∈poss s. s|-p = t
  using assms proof (induct s t rule: supteq.induct)
  case (refl s)
  have [] ∈ poss s by (induct s rule: term.induct) auto
  have s|-[] = s by simp
  from ‹[] ∈ poss s› and ‹s|-[] = s› show ?case by best
next
  case (subt u ss t f)
  then obtain p where p ∈ poss u and u|-p = t by best
  from ‹u ∈ set ss› obtain i
    where i < length ss and u = ss!i by (auto simp: in-set-conv-nth)
  from ‹i < length ss› and ‹p ∈ poss u›
  have i#p ∈ poss (Fun f ss) unfolding ‹u = ss!i› by simp
  have Fun f ss|- (i#p) = t
    unfolding subt-at.simps unfolding ‹u = ss!i›[symmetric] by (rule ‹u|-p = t›)
  with ‹i#p ∈ poss (Fun f ss)› show ?case by best
qed

lemma subt-at-imp-ctxt:
  assumes p ∈ poss s
  shows ∃ C. C⟨s|-p⟩ = s
  using assms proof (induct p arbitrary: s)
  case (Nil s)
  have □⟨s|-[]⟩ = s by simp
  then show ?case by best
next
  case (Cons i p s)
  then obtain f ss where s = Fun f ss by (cases s) auto
  with ‹i#p ∈ poss s› obtain u :: ('a,'b) term
    where u = ss!i and p ∈ poss u and i < length ss by auto
```

```
  from Cons and ‹p∈poss u› obtain D where D⟨u|-p⟩ = u by auto
  let ?ss1 = take i ss and ?ss2 = drop (Suc i) ss
  let ?E = More f ?ss1 D ?ss2
 have ?ss1@D⟨u|-p⟩#?ss2 = ss (is ?ss = ss) unfolding ‹D⟨u|-p⟩ = u› unfolding
‹u = ss!i›
    unfolding id-take-nth-drop[OF ‹i < length ss›, symmetric] ..
  have s|- (i#p) = u|-p unfolding ‹s = Fun f ss› using ‹u = ss!i› by simp
  have ?E⟨s|-(i#p)⟩ = s
    unfolding ctxt-apply-term.simps ‹s|-(i#p) = u|-p› ‹?ss = ss› unfolding ‹s =
Fun f ss› ..
  then show ?case by best
qed

lemma subt-at-imp-supteq':
  assumes p ∈ poss s and s|-p = t
  shows s ⊵ t
proof −
  from ‹p ∈ poss s› obtain C where C⟨s|-p⟩ = s using subt-at-imp-ctxt by best
  then show ?thesis unfolding ‹s|-p = t› using ctxt-imp-supteq by auto
qed

lemma subt-at-imp-supteq: p ∈ poss s ⟹ s ⊵ s|-p
  by (simp add: subt-at-imp-supteq')

lemma fun-poss-ctxt-apply-term:
  assumes p ∈ fun-poss C⟨s⟩
  shows (∀ t. p ∈ fun-poss C⟨t⟩) ∨ (∃ q. p = (hole-pos C) @ q ∧ q ∈ fun-poss s)
  using assms
proof (induct p arbitrary: C)
  case Nil then show ?case by (cases C) auto
next
  case (Cons i p)
  then show ?case
  proof (cases C)
    case (More f bef C' aft)
    with Cons(2) have i < length (bef @ C'⟨s⟩ # aft) by auto
    consider i < length bef | (C') i = length bef | i > length bef
      by (cases i < length bef, auto, cases i = length bef, auto)
    then show ?thesis
    proof (cases)
      case C'
      then have p ∈ fun-poss C'⟨s⟩ using More Cons by auto
      from Cons(1)[OF this] More C' show ?thesis by auto
    qed (insert More Cons, auto simp: nth-append)
  qed auto
qed
```

Conversions between contexts and proper subterms.

By adding *non-empty* to statements 2 and 3 a similar characterisation for

proper subterms is obtained:

1. $s \rhd t$
2. $\exists\, i\ p.\ i\#p \in poss\ s \wedge s|\text{-}(i\#p) = t$
3. $\exists\, C.\ C \neq \Box \wedge s = C\langle t\rangle$

lemma *supt-imp-subt-at-nepos*:
  assumes $s \rhd t$ **shows** $\exists\, i\ p.\ i\#p \in poss\ s \wedge s|\text{-}\ (i\#p) = t$
proof −
  from *assms* **have** $s \unrhd t$ **and** $s \neq t$ **unfolding** *supt-supteq-conv* **by** *auto*
  then obtain p **where** *supteq*: $p \in poss\ s\ s|\text{-}p = t$ **using** *supteq-imp-subt-at* **by** *best*
  have $p \neq []$
  proof
    assume $p = []$ **then have** $s = t$ **using** ‹$s|\text{-}p = t$› **by** *simp*
    then show *False* **using** ‹$s \neq t$› **by** *simp*
  qed
  then obtain $i\ q$ **where** $p = i\#q$ **by** (*cases p*) *simp*
  with *supteq* **show** *?thesis* **by** *auto*
qed

lemma *arg-neq*:
  assumes $i < length\ ss$ **and** $ss!i = Fun\ f\ ss$ **shows** *False*
proof −
  from ‹$i < length\ ss$› **have** $(ss!i) \in set\ ss$ **by** *auto*
  with *assms* **show** *False* **by** *simp*
qed

lemma *subt-at-nepos-neq*:
  assumes $i\#p \in poss\ s$ **shows** $s|\text{-}(i\#p) \neq s$
proof (*cases s*)
  fix x **assume** $s = Var\ x$
  then have $i\#p \notin poss\ s$ **by** *simp*
  with *assms* **show** *?thesis* **by** *simp*
next
  fix $f\ ss$ **assume** $s = Fun\ f\ ss$ **show** *?thesis*
  proof
    assume $s|\text{-}\ (i\#p) = s$
    from *assms* **have** $i < length\ ss$ **unfolding** ‹$s = Fun\ f\ ss$› **by** *auto*
    then have $ss!i \in set\ ss$ **by** *simp*
    then have $Fun\ f\ ss \rhd ss!i$ **by** (*rule supt.arg*)
    then have $Fun\ f\ ss \neq ss!i$ **unfolding** *supt-supteq-conv* **by** *simp*
    from ‹$s|\text{-}(i\#p) = s$› **and** *assms*
    have $ss!i \unrhd Fun\ f\ ss$ **using** *subt-at-imp-supteq′* **unfolding** ‹$s = Fun\ f\ ss$› **by** *auto*
    with *supt-not-sym*[*OF* ‹$Fun\ f\ ss \rhd ss!i$›] **have** $ss!i = Fun\ f\ ss$ **by** *auto*
    with ‹$i < length\ ss$› **show** *False* **by** (*rule arg-neq*)

```
  qed
qed

lemma subt-at-nepos-imp-supt:
  assumes i#p ∈ poss s shows s ▷ s |- (i#p)
proof −
  from assms have s ⊵ s|-(i#p) by (rule subt-at-imp-supteq)
  from assms have s|-(i#p) ≠ s by (rule subt-at-nepos-neq)
  from ‹s ⊵ s|-(i#p)› and ‹s|-(i#p) ≠ s› show ?thesis by (auto simp: supt-supteq-conv)
qed

lemma subt-at-nepos-imp-nectxt:
  assumes i#p ∈ poss s and s|-(i#p) = t shows ∃ C. C ≠ □ ∧ C⟨t⟩ = s
proof −
  from assms obtain C where C⟨s|-(i#p)⟩ = s using subt-at-imp-ctxt by best
  from ‹i#p ∈ poss s›
   have t ≠ s unfolding ‹s|-(i#p) = t›[symmetric] using subt-at-nepos-neq by
best
  from assms and ‹C⟨s|-(i#p)⟩ = s› have C⟨t⟩ = s by simp
  have C ≠ □
  proof
    assume C = □
    with ‹C⟨t⟩ = s› have t = s by simp
    with ‹t ≠ s› show False by simp
  qed
  with ‹C⟨t⟩ = s› show ?thesis by auto
qed

lemma supteq-subst-cases':
  s · σ ⊵ t ⟹ (∃ u. s ⊵ u ∧ is-Fun u ∧ t = u · σ) ∨ (∃ x. x ∈ vars-term s ∧ σ
x ⊵ t)
proof (induct s)
  case (Fun f ss)
  from Fun(2)
  show ?case
  proof (cases rule: supteq.cases)
    case refl
    show ?thesis
      by (intro disjI1 exI[of - Fun f ss], auto simp: refl)
  next
    case (subt v ts g)
    then obtain si where si: si ∈ set ss si · σ ⊵ t by auto
    from Fun(1)[OF this] si(1) show ?thesis by auto
  qed
qed simp

lemma size-const-subst[simp]: size (t · (λ - . Fun f [])) = size t
proof (induct t)
  case (Fun f ts)
```

then show *?case* **by** (*induct ts*, *auto*)
qed *simp*

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

lemma *supteq-subst-cases* [*consumes 1*, *case-names in-term in-subst*]:
$s \cdot \sigma \trianglerighteq t \Longrightarrow$
$(\bigwedge u.\ s \trianglerighteq u \Longrightarrow$ *is-Fun* $u \Longrightarrow t = u \cdot \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ x \in$ *vars-term* $s \Longrightarrow \sigma\ x \trianglerighteq t \Longrightarrow P) \Longrightarrow$
P
using *supteq-subst-cases'* **by** *blast*

lemma *poss-subst-apply-term*:
assumes $p \in$ *poss* $(t \cdot \sigma)$ **and** $p \notin$ *fun-poss* t
obtains $q\ r\ x$ **where** $p = q$ @ r **and** $q \in$ *poss* t **and** t |- $q =$ *Var* x **and** $r \in$ *poss* $(\sigma\ x)$
using *assms*
proof (*induct t arbitrary*: *p*)
case (*Fun f ts*)
then show *?case* **by** (*auto*) (*metis append-Cons nth-mem subt-at.simps*(*2*))
qed *simp*

lemma *subt-at-subst* [*simp*]:
assumes $p \in$ *poss* t **shows** $(t \cdot \sigma)$ |- $p = (t$ |- $p) \cdot \sigma$
using *assms* **by** (*induct t arbitrary*: *p*) *auto*

lemma *vars-term-size*:
assumes $x \in$ *vars-term* t
shows *size* $(\sigma\ x) \leq$ *size* $(t \cdot \sigma)$
using *assms*
by (*induct t*)
(*auto*, *metis* (*no-types*) *comp-apply le-SucI size-list-estimation'*)

Restrict a substitution to a set of variables.

definition
subst-restrict :: ($'f$, $'v$) *subst* $\Rightarrow$ $'v$ *set* $\Rightarrow$ ($'f$, $'v$) *subst*
(**infix** |*s 67*)
where
σ |*s* $V = (\lambda x.$ *if* $x \in V$ *then* $\sigma(x)$ *else Var* $x)$

lemma *subst-restrict-Int* [*simp*]:
$(\sigma$ |*s* V) |*s* $W = \sigma$ |*s* $(V \cap W)$
by (*rule ext*) (*simp add*: *subst-restrict-def*)

lemma *subst-domain-Var-conv* [*iff*]:
subst-domain $\sigma = \{\} \longleftrightarrow \sigma =$ *Var*
proof
assume *subst-domain* $\sigma = \{\}$
show $\sigma =$ *Var*

proof (*rule ext*)
fix x **show** $\sigma(x) = Var\ x$
proof (*rule ccontr*)
assume $\sigma(x) \neq Var\ x$
then have $x \in$ *subst-domain* σ **by** (*simp add*: *subst-domain-def*)
with ‹*subst-domain* $\sigma = \{\}$› **show** *False* **by** *simp*
qed
qed
next
assume $\sigma = Var$ **then show** *subst-domain* $\sigma = \{\}$ **by** *simp*
qed

lemma *subst-compose-Var*[*simp*]: $\sigma \circ_s Var = \sigma$ **by** (*simp add*: *subst-compose-def*)

lemma *Var-subst-compose*[*simp*]: $Var \circ_s \sigma = \sigma$ **by** (*simp add*: *subst-compose-def*)

We use the same logical constant as for the power operations on functions and relations, in order to share their syntax.

overloading
substpow $\equiv$ *compow* :: *nat* $\Rightarrow$ ('*f*, '*v*) *subst* $\Rightarrow$ ('*f*, '*v*) *subst*
begin

primrec *substpow* :: *nat* $\Rightarrow$ ('*f*, '*v*) *subst* $\Rightarrow$ ('*f*, '*v*) *subst* **where**
substpow 0 $\sigma = Var$
| *substpow* (*Suc n*) $\sigma = \sigma \circ_s$ *substpow n* σ

end

lemma *subst-power-compose-distrib*:
μ ^^ $(m + n) = (\mu$ ^^ $m \circ_s \mu$ ^^ $n)$ **by** (*induct m*) (*simp-all add*: *ac-simps*)

lemma *subst-power-Suc*: μ ^^ $(Suc\ i) = \mu$ ^^ $i \circ_s \mu$
proof −
have μ ^^ $(Suc\ i) = \mu$ ^^ $(i + Suc\ 0)$ **by** *simp*
then show *?thesis* **unfolding** *subst-power-compose-distrib* **by** *simp*
qed

lemma *subst-pow-mult*: $((\sigma :: ('f,'v)subst)$ ^^ $n)$ ^^ $m = \sigma$ ^^ $(n * m)$
by (*induct m arbitrary*: *n*, *auto simp*: *subst-power-compose-distrib*)

lemma *subst-domain-pow*: *subst-domain* (σ ^^ n) $\subseteq$ *subst-domain* σ
unfolding *subst-domain-def*
by (*induct n*, *auto simp*: *subst-compose-def*)

lemma *subt-at-Cons-distr* [*simp*]:
assumes $i \# p \in$ *poss t* **and** $p \neq []$
shows t |- $(i \# p) = (t$ |- $[i])$ |- p
using *assms* **by** (*induct t*) *auto*

```
lemma subt-at-append [simp]:
  p ∈ poss t ⟹ t |- (p @ q) = (t |- p) |- q
proof (induct t arbitrary: p)
  case (Fun f ts)
  show ?case
  proof (cases p)
    case (Cons i p')
    with Fun(2) have i: i < length ts and p': p' ∈ poss (ts ! i) by auto
    from i have ti: ts ! i ∈ set ts by auto
    show ?thesis using Fun(1)[OF ti p'] unfolding Cons by auto
  qed auto
qed auto

lemma subt-at-pos-diff:
  assumes p <p q and p: p ∈ poss s
  shows s |- p |- pos-diff q p = s |- q
  using assms unfolding subt-at-append [OF p, symmetric] by simp

lemma empty-pos-in-poss[simp]: [] ∈ poss t by (induct t) auto

lemma poss-append-poss[simp]: (p @ q ∈ poss t) = (p ∈ poss t ∧ q ∈ poss (t |-
p)) (is ?l = ?r)
proof
  assume ?r
  with pos-append-poss[of p t q] show ?l by auto
next
  assume ?l
  then show ?r
  proof (induct p arbitrary: t)
    case (Cons i p t)
    then obtain f ts where t: t = Fun f ts by (cases t, auto)
    note IH = Cons[unfolded t]
    from IH(2) have i: i < length ts and rec: p @ q ∈ poss (ts ! i) by auto
    from IH(1)[OF rec] i show ?case unfolding t by auto
  qed auto
qed

lemma subterm-poss-conv:
  assumes p ∈ poss t and [simp]: p = q @ r and t |- q = s
  shows t |- p = s |- r ∧ r ∈ poss s (is ?A ∧ ?B)
proof −
  have qr: q @ r ∈ poss t using assms(1) by simp
  then have q-in-t: q ∈ poss t using poss-append-poss by auto
  show ?thesis
  proof
    have t |- p = t |- (q @ r) by simp
    also have ... = s |- r using subt-at-append[OF q-in-t] assms(3) by simp
    finally show ?A .
  next
```

```
    show ?B using poss-append-poss qr assms(3) by auto
  qed
qed

lemma poss-imp-subst-poss [simp]:
  assumes p ∈ poss t
  shows p ∈ poss (t · σ)
  using assms by (induct t arbitrary: p) auto

lemma iterate-term:
  assumes id: t · σ |- p = t and pos: p ∈ poss (t · σ)
  shows t · σ ^^ n |- (p^n) = t ∧ p ^ n ∈ poss (t · σ ^^ n)
proof (induct n)
  case (Suc n)
  then have p: p ^ n ∈ poss (t · σ ^^ n) by simp
  note p' = poss-imp-subst-poss[OF p]
  note p'' = subt-at-append[OF p']
  have idt: t · σ ^^ (Suc n) = t · σ^^ n · σ unfolding subst-power-Suc by simp
  have t · σ ^^ (Suc n) |- (p ^ Suc n)
    = t · σ ^^ n · σ |- (p ^ n @ p) unfolding idt power-pos-Suc ..
  also have ... = ((t · σ ^^ n |- p ^ n) · σ) |- p unfolding p'' subt-at-subst[OF p]
..
  also have ... = t · σ |- p unfolding Suc[THEN conjunct1] ..
  also have ... = t unfolding id ..
  finally have one: t · σ ^^ Suc n |- (p ^ Suc n) = t .
  show ?case
  proof (rule conjI[OF one])
    show p ^ Suc n ∈ poss (t · σ ^^ Suc n)
      unfolding power-pos-Suc poss-append-poss idt
    proof (rule conjI[OF poss-imp-subst-poss[OF p]])
      have t · σ ^^ n · σ |- (p ^ n) = t · σ ^^ n |- (p ^ n) · σ
        by (rule subt-at-subst[OF p])
      also have ... = t · σ using Suc by simp
      finally show p ∈ poss (t · σ ^^ n · σ |- p ^ n) using pos by auto
    qed
  qed
qed simp

lemma hole-pos-poss [simp]: hole-pos C ∈ poss (C⟨t⟩)
  by (induct C) auto

lemma hole-pos-poss-conv: (hole-pos C @ q) ∈ poss (C⟨t⟩) ⟷ q ∈ poss t
  by (induct C) auto

lemma subt-at-hole-pos [simp]: C⟨t⟩ |- hole-pos C = t
  by (induct C) auto

lemma hole-pos-power-poss [simp]: (hole-pos C) ^ (n::nat) ∈ poss ((C ^ n)⟨t⟩)
  by (induct n) (auto simp: hole-pos-poss-conv)
```

lemma *poss-imp-ctxt-subst-poss* [*simp*]:
  assumes $p \in poss\ (C\langle t\rangle)$
  shows $p \in poss\ ((C \cdot_c \sigma)\langle t \cdot \sigma\rangle)$
proof −
  have $p \in poss\ (C\langle t\rangle \cdot \sigma)$ **by** (*rule poss-imp-subst-poss* [*OF assms*])
  then show *?thesis* **by** *simp*
qed

lemma *poss-Cons-poss*[*simp*]: $(i \# p \in poss\ t) = (i < length\ (args\ t) \wedge p \in poss\ (args\ t\ !\ i))$
  by (*cases t*, *auto*)

lemma *less-pos-imp-supt*:
  assumes *less*: $p' <_p p$ **and** *p*: $p \in poss\ t$
  shows t |- $p \lhd t$ |- p'
proof −
  from *less* **obtain** p'' **where** p'': $p = p'$ @ p'' **unfolding** *less-pos-def less-eq-pos-def* **by** *auto*
  with *less* **have** *ne*: $p'' \neq []$ **by** *auto*
  then obtain *i q* **where** *ne*: $p'' = i \# q$ **by** (*cases* p'', *auto*)
  from *p* **have** p': $p' \in poss\ t$ **unfolding** p'' **by** *simp*
  from *p* **have** $p'' \in poss\ (t$ |- $p')$ **unfolding** p'' **by** *simp*
  from *subt-at-nepos-imp-supt*[*OF this*[*unfolded ne*]] **have** t |- $p' \rhd t$ |- p' |- p'' **unfolding** *ne* **by** *simp*
  then show t |- $p' \rhd t$ |- p **unfolding** p'' *subt-at-append*[*OF* p'] **.**
qed

lemma *less-eq-pos-imp-supt-eq*:
  assumes *less-eq*: $p' \leq_p p$ **and** *p*: $p \in poss\ t$
  shows t |- $p \unlhd t$ |- p'
proof −
  from *less-eq* **obtain** p'' **where** p'': $p = p'$ @ p'' **unfolding** *less-eq-pos-def* **by** *auto*
  from *p* **have** p': $p' \in poss\ t$ **unfolding** p'' **by** *simp*
  from *p* **have** $p'' \in poss\ (t$ |- $p')$ **unfolding** p'' **by** *simp*
  from *subt-at-imp-supteq*[*OF this*] **have** t |- $p' \unrhd t$ |- p' |- p'' **by** *simp*
  then show t |- $p' \unrhd t$ |- p **unfolding** p'' *subt-at-append*[*OF* p'] **.**
qed

lemma *funas-term-poss-conv*:
  $funas\text{-}term\ t = \{(f,\ length\ ts) \mid p\ f\ ts.\ p \in poss\ t \wedge t|\text{-}p = Fun\ f\ ts\}$
proof (*induct t*)
  case (*Fun f ts*)
  let $?f = \lambda\ f\ ts.\ (f, length\ ts)$
  let $?fs = \lambda\ t.\ \{?f\ f\ ts \mid p\ f\ ts.\ p \in poss\ t \wedge t|\text{-}p = Fun\ f\ ts\}$
  let $?l = funas\text{-}term\ (Fun\ f\ ts)$
  let $?r = ?fs\ (Fun\ f\ ts)$
  {

fix *gn*
have $(gn \in ?l) = (gn \in ?r)$
proof (*cases gn* = (*f*,*length ts*))
case *False*
obtain *g n* **where** *gn*: *gn* = (*g*,*n*) **by** *force*
have $(gn \in ?l) = (\exists\ t \in set\ ts.\ gn \in funas\text{-}term\ t)$ **using** *False* **by** *auto*
also have ... = $(\exists\ i < length\ ts.\ gn \in funas\text{-}term\ (ts\ !\ i))$ **unfolding** *set-conv-nth* **by** *auto*
also have ... = $(\exists\ i < length\ ts.\ (g,n) \in ?fs\ (ts\ !\ i))$ **using** *Fun*[*unfolded set-conv-nth*] *gn* **by** *blast*
also have ... = $((g,n) \in ?fs\ (Fun\ f\ ts))$ (**is** $?l' = ?r'$)
proof
assume $?l'$
then obtain *i p ss* **where** *p*: $p \in poss\ (ts\ !\ i)\ ts\ !\ i$ |- $p = Fun\ g\ ss\ n = length\ ss\ i < length\ ts$ **by** *auto*
show $?r'$
by (*rule*, *rule exI*[*of - i # p*], *intro exI conjI*, *unfold p*(*3*), *rule refl*, *insert p*(*1*) *p*(*2*) *p*(*4*), *auto*)
next
assume $?r'$
then obtain *p ss* **where** *p*: $p \in poss\ (Fun\ f\ ts)\ Fun\ f\ ts$ |- $p = Fun\ g\ ss\ n = length\ ss$ **by** *auto*
from *p False gn* **obtain** $i\ p'$ **where** *pp*: $p = i\ \#\ p'$ **by** (*cases p*, *auto*)
show $?l'$
by (*rule exI*[*of - i*], *insert p pp*, *auto*)
qed
finally show *?thesis* **unfolding** *gn* **.**
qed *force*
}
then show *?case* **by** *blast*
qed *simp*

inductive
subst-instance :: $('f,\ 'v)\ term \Rightarrow ('f,\ 'v)\ term \Rightarrow bool$ ((-/ $\preceq$ -) [*56*, *56*] *55*)
where
subst-instanceI [*intro*]:
$s \cdot \sigma = t \Longrightarrow s \preceq t$

lemma *subst-instance-trans*[*trans*]:
assumes $s \preceq t$ **and** $t \preceq u$ **shows** $s \preceq u$
proof −
from ‹$s \preceq t$› **obtain** σ **where** $s{\cdot}\sigma = t$ **by** (*cases rule*: *subst-instance.cases*) *best*
from ‹$t \preceq u$› **obtain** τ **where** $t{\cdot}\tau = u$ **by** (*cases rule*: *subst-instance.cases*) *best*
then have $(s{\cdot}\sigma){\cdot}\tau = u$ **unfolding** ‹$s{\cdot}\sigma = t$› **.**
then have $s{\cdot}(\sigma \circ_s \tau) = u$ **by** *simp*
then show *?thesis* **by** (*rule subst-instanceI*)
qed

lemma *subst-instance-refl*: $s \preceq s$

using *subst-instanceI*[**where** $\sigma = Var$ **and** $s = s$ **and** $t = s$] **by** *simp*

lemma *subst-neutral*: *subst-domain* $\sigma \subseteq V \Longrightarrow (Var\ x)\cdot(\sigma\ |s\ (V - \{x\})) = (Var\ x)$
by (*auto simp*: *subst-domain-def subst-restrict-def*)

lemma *subst-restrict-domain*[*simp*]: $\sigma\ |s$ *subst-domain* $\sigma = \sigma$
proof −
have $\sigma\ |s$ *subst-domain* $\sigma = (\lambda x.\ if\ x \in$ *subst-domain* $\sigma\ then\ \sigma(x)\ else\ Var\ x)$
by (*simp add*: *subst-restrict-def*)
also have ... $= \sigma$ **by** (*rule ext*) (*simp add*: *subst-domain-def*)
finally show *?thesis* **.**
qed

lemma *notin-subst-domain-imp-Var*:
assumes $x \notin$ *subst-domain* σ
shows $\sigma\ x = Var\ x$
using *assms* **by** (*auto simp*: *subst-domain-def*)

lemma *subst-domain-neutral*[*simp*]:
assumes *subst-domain* $\sigma \subseteq V$
shows $(\sigma\ |s\ V) = \sigma$
proof −
{
fix x
have $(if\ x \in V\ then\ \sigma(x)\ else\ Var\ x) = (if\ x \in$ *subst-domain* $\sigma\ then\ \sigma(x)\ else\ Var\ x)$
proof (*cases* $x \in$ *subst-domain* σ)
case *True*
then have x: $x \in V = True$ **using** *assms* **by** *auto*
show *?thesis* **unfolding** x **using** *True* **by** *simp*
next
case *False*
then have x: $x \notin$ *subst-domain* (σ) **.**
show *?thesis* **unfolding** *notin-subst-domain-imp-Var*[*OF x*] *if-cancel* **..**
qed
}
then have $\bigwedge x.(if\ x \in V\ then\ \sigma\ x\ else\ Var\ x) = (if\ x \in$ *subst-domain* $\sigma\ then\ \sigma\ x\ else\ Var\ x)$ **.**
then have $\bigwedge x.(\lambda x.\ if\ x \in V\ then\ \sigma\ x\ else\ Var\ x)\ x = (\lambda x.\ if\ x \in$ *subst-domain* $\sigma\ then\ \sigma\ x\ else\ Var\ x)\ x$ **.**
then have $\bigwedge x.\ (\lambda x.\ if\ x \in V\ then\ \sigma\ x\ else\ Var\ x)\ x = \sigma\ x$ **by** (*auto simp*: *subst-domain-def*)
then have $(\lambda x.\ if\ x \in V\ then\ \sigma\ x\ else\ Var\ x) = \sigma$ **by** (*rule ext*)
then have $\sigma\ |s\ V = \sigma$ **by** (*simp add*: *subst-restrict-def*)
then show *?thesis* **.**
qed

lemma *subst-restrict-UNIV*[*simp*]: $\sigma\ |s\ UNIV = \sigma$ **by** (*auto simp*: *subst-restrict-def*)

lemma *subst-restrict-empty*[*simp*]: σ |*s* {} = *Var* **by** (*simp add*: *subst-restrict-def*)

lemma *vars-term-subst-pow*:
vars-term ($t \cdot \sigma$^^*n*) $\subseteq$ *vars-term t* $\cup \bigcup$(*vars-term* ‘ *subst-range* σ) (**is** - $\subseteq$ *?R t*)
unfolding *vars-term-subst*
proof (*induct n arbitrary*: *t*)
case (*Suc n t*)
show *?case*
proof
fix *x*
assume $x \in \bigcup$(*vars-term* ‘ (σ ^^ *Suc n*) ‘ *vars-term t*)
then obtain *y u* **where** *1*: $y \in$ *vars-term t u* = (σ ^^ *Suc n*) *y x* $\in$ *vars-term u*
by *auto*
from *1*(*2*) **have** *u* = σ *y* $\cdot$ σ ^^ *n* **by** (*auto simp*: *subst-compose-def*)
from *1*(*3*)[*unfolded this*, *unfolded vars-term-subst*]
have $x \in \bigcup$(*vars-term* ‘ (σ ^^ *n*) ‘ *vars-term* (σ *y*)) **.**
with *Suc*[*of* σ *y*] **have** *x*: $x \in$ *?R* (σ *y*) **by** *auto*
then show $x \in$ *?R t*
proof
assume $x \in$ *vars-term* (σ *y*)
then show *?thesis* **using** *1*(*1*) **by** (*cases* σ *y* = *Var y*, *auto simp*: *subst-domain-def*)
qed *auto*
qed
qed *auto*

lemma *coincidence-lemma*:
$t \cdot \sigma = t \cdot$ (σ |*s vars-term t*)
unfolding *term-subst-eq-conv subst-restrict-def* **by** *auto*

lemma *subst-domain-vars-term-subset*:
subst-domain (σ |*s vars-term t*) $\subseteq$ *vars-term t*
by (*auto simp*: *subst-domain-def subst-restrict-def*)

lemma *subst-restrict-single-Var* [*simp*]:
assumes $x \notin$ *subst-domain* σ **shows** σ |*s* {*x*} = *Var*
proof −
have *A*: $\bigwedge x.\ x \notin$ *subst-domain* $\sigma \Longrightarrow \sigma\ x$ = *Var x* **by** (*simp add*: *subst-domain-def*)
have σ |*s* {*x*} = (λy. *if* $y \in$ {*x*} *then* σ *y else Var y*) **by** (*simp add*: *subst-restrict-def*)
also have ... = (λy. *if* $y = x$ *then* σ *y else Var y*) **by** *simp*
also have ... = (λy. *if* $y = x$ *then* σ *x else Var y*) **by** (*simp cong*: *if-cong*)
also have ... = (λy. *if* $y = x$ *then Var x else Var y*) **unfolding** *A*[*OF assms*] **by** *simp*
also have ... = (λy. *if* $y = x$ *then Var y else Var y*) **by** (*simp cong*: *if-cong*)
also have ... = (λy. *Var y*) **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *subst-restrict-single-Var'*:
  assumes $x \notin$ *subst-domain* σ **and** σ |*s* $V = Var$ **shows** σ |*s* $(\{x\} \cup V) = Var$
proof −
  have $(\lambda y.$ *if* $y \in V$ *then* $\sigma\ y$ *else* $Var\ y) = (\lambda y.\ Var\ y)$
    using ‹σ |*s* $V = Var$› **by** (*simp add*: *subst-restrict-def*)
  then have $(\lambda y.$ *if* $y \in V$ *then* $\sigma\ y$ *else* $Var\ y) = (\lambda y.\ Var\ y)$ **by** *simp*
  then have A: $\bigwedge y.$ (*if* $y \in V$ *then* $\sigma\ y$ *else* $Var\ y) = Var\ y$ **by** (*rule fun-cong*)
  {
    fix y
    have (*if* $y \in \{x\} \cup V$ *then* $\sigma\ y$ *else* $Var\ y) = Var\ y$
    proof (*cases* $y = x$)
      assume $y = x$ **then show** *?thesis* **using** ‹$x \notin$ *subst-domain* σ› **by** (*auto simp*: *subst-domain-def*)
    next
      assume $y \neq x$ **then show** *?thesis* **using** A **by** *simp*
    qed
  }
  then have $\bigwedge y.$ (*if* $y \in \{x\} \cup V$ *then* $\sigma\ y$ *else* $Var\ y) = Var\ y$ **by** *simp*
  then show *?thesis* **by** (*auto simp*: *subst-restrict-def*)
qed

lemma *subst-restrict-empty-set*:
  finite $V \Longrightarrow V \cap$ *subst-domain* $\sigma = \{\} \Longrightarrow \sigma$ |*s* $V = Var$
proof (*induct rule*: *finite.induct*)
  case (*insertI* $V\ x$)
  then have $V \cap$ *subst-domain* $\sigma = \{\}$ **by** *simp*
  with *insertI* **have** σ |*s* $V = Var$ **by** *simp*
  then show *?case* **using** *insertI subst-restrict-single-Var'*[**where** $\sigma = \sigma$ **and** $x = x$ **and** $V = V$] **by** *simp*
qed *auto*

lemma *subst-restrict-Var*: $x \neq y \Longrightarrow Var\ y \cdot (\sigma$ |*s* $(UNIV - \{x\})) = Var\ y \cdot \sigma$
  by (*auto simp*: *subst-restrict-def*)

lemma *var-cond-stable*:
  assumes *vars-term* $r \subseteq$ *vars-term* l
  shows *vars-term* $(r \cdot \mu) \subseteq$ *vars-term* $(l \cdot \mu)$
  unfolding *vars-term-subst* **using** *assms* **by** *blast*

lemma *instance-no-supt-imp-no-supt*:
  assumes $\neg\ s \cdot \sigma \rhd t \cdot \sigma$
  shows $\neg\ s \rhd t$
proof
  assume $s \rhd t$
  hence $s \cdot \sigma \rhd t \cdot \sigma$ **by** (*rule supt-subst*)
  with *assms* **show** *False* **by** *simp*
qed

lemma *subst-image-subterm*:

```
  assumes x ∈ vars-term (Fun f ss)
  shows Fun f ss · σ ▷ σ x
proof −
  have Fun f ss ⊵ Var x using supteq-Var[OF assms(1)] .
  then have Fun f ss ▷ Var x by cases auto
  from supt-subst [OF this]
  show ?thesis by simp
qed

lemma funas-term-subst-pow:
  funas-term (t · σ^^n) ⊆ funas-term t ∪ ⋃(funas-term ‘ subst-range σ)
proof −
  {
    fix Xs
    have ⋃(funas-term ‘ (σ ^^ n) ‘ Xs) ⊆ ⋃(funas-term ‘ subst-range σ)
    proof (induct n arbitrary: Xs)
      case (Suc n Xs)
      show ?case (is ⋃ ?L ⊆ ?R)
      proof (rule subsetI)
        fix f
        assume f ∈ ⋃ ?L
        then obtain x where f ∈ funas-term ((σ ^^ Suc n) x) by auto
       then have f ∈ funas-term (σ x · σ ^^ n) by (auto simp: subst-compose-def)
        from this[unfolded funas-term-subst]
        show f ∈ ?R using Suc[of vars-term (σ x)]
           unfolding subst-range.simps subst-domain-def by (cases σ x = Var x,
auto)
      qed
    qed auto
  }
  then show ?thesis unfolding funas-term-subst by auto
qed

lemma funas-term-subterm-args:
  assumes sF: funas-term s ⊆ F
    and q: q ∈ poss s
  shows ⋃(funas-term ‘ set (args (s |- q))) ⊆ F
proof −
  from subt-at-imp-ctxt[OF q] obtain C where s: s = C ⟨ s |- q ⟩ by metis
  from sF arg-cong[OF this, of funas-term] have funas-term (s |- q) ⊆ F by auto
  then show ?thesis by (cases s |- q, auto)
qed

lemma get-var-or-const: ∃ C t. s = C⟨t⟩ ∧ args t = []
proof (induct s)
  case (Var y)
  show ?case by (rule exI[of - Hole], auto)
next
  case (Fun f ts)
```

show *?case*
proof (*cases ts*)
case *Nil*
show *?thesis* **unfolding** *Nil*
by (*rule exI*[*of - Hole*], *auto*)
next
case (*Cons s ss*)
then have $s \in$ *set ts* **by** *auto*
from *Fun*[*OF this*] **obtain** *C* **where** *C*: $\exists$ *t*. $s = C\langle t\rangle \wedge$ *args t* = [] **by** *auto*
show *?thesis* **unfolding** *Cons*
by (*rule exI*[*of - More f* [] *C ss*], *insert C*, *auto*)
qed
qed

lemma *supteq-Var-id* [*simp*]:
assumes *Var x* $\trianglerighteq$ *s* **shows** *s* = *Var x*
using *assms* **by** (*cases*)

lemma *arg-not-term* [*simp*]:
assumes $t \in$ *set ts* **shows** *Fun f ts* $\neq$ *t*
proof (*rule ccontr*)
assume $\neg$ *Fun f ts* $\neq$ *t*
then have *size* (*Fun f ts*) = *size t* **by** *simp*
moreover have *size t* < *size-list size ts* **using** *assms* **by** (*induct ts*) *auto*
ultimately show *False* **by** *simp*
qed

lemma *arg-subteq* [*simp*]: $t \in$ *set ts* $\Longrightarrow$ *Fun f ts* $\trianglerighteq$ *t*
by *auto*

lemma *supt-imp-args*:
assumes $\forall$ *t*. $s \triangleright t \longrightarrow P\ t$
shows $\forall$ *t*$\in$*set* (*args s*). *P t*
using *assms* **by** (*cases s*) *simp-all*

lemma *ctxt-apply-eq-False*[*simp*]: (*More f ss1 D ss2*)$\langle t\rangle \neq t$ (**is** *?C*$\langle$-$\rangle \neq$ -)
proof
assume *eq*: *?C*$\langle t\rangle = t$
have *?C* $\neq \square$ **by** *auto*
from *ctxt-supt*[*OF this eq*[*symmetric*]]
have $t \triangleright t$ **.**
then show *False* **by** *auto*
qed

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

lemma *funs-term-subst*: *funs-term* $(t \cdot \sigma)$ = *funs-term t* $\cup \bigcup$ ((λ *x*. *funs-term* (σ *x*)) ‘ (*vars-term t*))

by (*induct t*) *auto*

lemma *set-set-cons*:
assumes $P\ x$ **and** $\bigwedge y.\ y \in set\ xs \Longrightarrow P\ y$
shows $y \in set\ (x\ \#\ xs) \Longrightarrow P\ y$
using *assms* **by** *auto*

lemma *ctxt-power-compose-distr*: $C\ \hat{}\ (m + n) = C\ \hat{}\ m \circ_c C\ \hat{}\ n$
by (*induct m*) (*simp-all add*: *ac-simps*)

lemma *subst-apply-id'*:
assumes $vars\text{-}term\ t \cap V = \{\}$
shows $t \cdot (\sigma\ |s\ V) = t$
using *assms*
proof (*induct t*)
case (*Var x*) **then show** *?case* **by** (*simp add*: *subst-restrict-def*)
next
case (*Fun f ts*)
then have $\forall s \in set\ ts.\ s \cdot (\sigma\ |s\ V) = s$ **by** *auto*
with *map-idI* [*of ts* $\lambda t.\ t \cdot (\sigma\ |s\ V)$] **show** *?case* **by** *simp*
qed

lemma *subst-apply-ctxt-id*:
assumes $vars\text{-}ctxt\ C \cap V = \{\}$
shows $C \cdot_c (\sigma\ |s\ V) = C$
using *assms*
proof (*induct C*)
case (*More f ss1 D ss2*)
then have *IH*: $D \cdot_c (\sigma\ |s\ V) = D$ **by** *auto*
from *More* **have** $\forall s \in set(ss1@ss2).\ vars\text{-}term\ s \cap V = \{\}$ **by** *auto*
with *subst-apply-id'* **have** *args*: $\forall s \in set(ss1@ss2).\ s \cdot (\sigma\ |s\ V) = s$ **by** *best*
from *args* **have** $\forall s \in set\ ss1.\ s \cdot (\sigma\ |s\ V) = s$ **by** *simp*
with *map-idI*[*of ss1* $\lambda t.\ t \cdot (\sigma\ |s\ V)$] **have** *ss1*: $map\ (\lambda s.\ s \cdot (\sigma\ |s\ V))\ ss1 = ss1$
by *best*
from *args* **have** $\forall s \in set\ ss2.\ s \cdot (\sigma\ |s\ V) = s$ **by** *simp*
with *map-idI*[*of ss2* $\lambda t.\ t \cdot (\sigma\ |s\ V)$] **have** *ss2*: $map\ (\lambda s.\ s \cdot (\sigma\ |s\ V))\ ss2 = ss2$
by *best*
show *?case* **by** (*simp add*: *ss1 ss2 IH*)
qed *simp*

lemma *vars-term-Var-id*: $vars\text{-}term\ o\ Var = (\lambda x.\ \{x\})$
by (*rule ext*) *simp*

lemma *ctxt-exhaust-rev*[*case-names Hole More*]:
assumes $C = \Box \Longrightarrow P$ **and**
$\bigwedge D\ f\ ss1\ ss2.\ C = D \circ_c (More\ f\ ss1\ \Box\ ss2) \Longrightarrow P$
shows P
proof (*cases C*)
case *Hole* **with** *assms* **show** *?thesis* **by** *simp*

next
  case (*More g ts1 E ts2*)
  then have $\exists$ *D f ss1 ss2*. *C* = *D* $\circ_c$ (*More f ss1* $\square$ *ss2*)
  proof (*induct E arbitrary*: *C g ts1 ts2*)
    case *Hole* **then have** *C* = $\square$ $\circ_c$ (*More g ts1* $\square$ *ts2*) **by** *simp*
    then show *?case* **by** *best*
  next
    case (*More h us1 F us2*)
    from *More*(*1*)[*of More h us1 F us2*]
    obtain *G i vs1 vs2* **where** *IH*: *More h us1 F us2* = *G* $\circ_c$ *More i vs1* $\square$ *vs2* **by** *force*
    from *More* **have** *C* = (*More g ts1* $\square$ *ts2* $\circ_c$ *G*) $\circ_c$ *More i vs1* $\square$ *vs2* **unfolding** *IH* **by** *simp*
    then show *?case* **by** *best*
  qed
  then show *?thesis* **using** *assms* **by** *auto*
qed

fun
  subst-extend :: ('*f*, '*v*, '*w*) *gsubst* $\Rightarrow$ ('*v* $\times$ ('*f*, '*w*) *term*) *list* $\Rightarrow$ ('*f*, '*v*, '*w*) *gsubst*
  where
    subst-extend σ *vts* = (λ*x*.
    (*case map-of vts x of*
      Some t $\Rightarrow$ *t*
    | *None* $\Rightarrow$ σ(*x*)))

lemma *subst-extend-id*:
  assumes *V* $\cap$ *set vs* = {} **and** *vars-term t* $\subseteq$ *V*
  shows *t* $\cdot$ *subst-extend* σ (*zip vs ts*) = *t* $\cdot$ σ
  using *assms*
proof (*induct t*)
  case (*Var x*) **then show** *?case*
    using *map-of-SomeD*[*of zip vs ts x*]
    using *set-zip-leftD* [*of x - vs ts*]
    using *IntI* [*of x V set vs*]
    using *emptyE*
    by (*case-tac map-of* (*zip vs ts*) *x*) *auto*
qed *auto*

lemma *funas-term-args*:
  $\bigcup$(*funas-term* ‘ *set* (*args t*)) $\subseteq$ *funas-term t*
  by (*cases t*) *auto*

lemma *subst-extend-absorb*:
  assumes *distinct vs* **and** *length vs* = *length ss*
  shows *map* (λ*t*. *t* $\cdot$ *subst-extend* σ (*zip vs ss*)) (*map Var vs*) = *ss* (**is** *?ss* = -)
proof −
  let *?*σ = *subst-extend* σ (*zip vs ss*)
  from *assms* **have** *length vs* $\leq$ *length ss* **by** *simp*

from *assms* **have** *length ?ss* = *length ss* **by** *simp*
moreover have $\forall$ *i*<*length ?ss*. *?ss* ! *i* = *ss* ! *i*
proof (*intro allI impI*)
fix *i* **assume** *i* < *length ?ss*
then have *i*: *i* < *length* (*map Var vs*) **by** *simp*
then have *len*: *i* < *length vs* **by** *simp*
have *?ss*!*i* = (*map Var vs* ! *i*) · *?σ* **unfolding** *nth-map*[*OF i*, *of* λ*t*. *t*·*?σ*] **by** *simp*
also have ... = *Var*(*vs*!*i*)·*?σ* **unfolding** *nth-map*[*OF len*] **by** *simp*
also have ... = (*case map-of* (*zip vs ss*) (*vs* ! *i*) *of None* ⇒ *σ* (*vs* ! *i*) | *Some t* ⇒ *t*) **by** *simp*
also have ... = *ss* ! *i* **using** ‹*distinct vs*› ‹*length vs* ≤ *length ss*› *len*
by (*simp add*: *assms*(*2*) *map-of-zip-nth*)
finally show *?ss*!*i* = *ss*!*i* **by** *simp*
qed
ultimately show *?thesis* **by** (*metis nth-equalityI*)
qed

abbreviation *map-funs-term f* ≡ *term.map-term f* (λ*x*. *x*)
abbreviation *map-funs-ctxt f* ≡ *ctxt.map-ctxt f* (λ*x*. *x*)

lemma *funs-term-map-funs-term-id*: (⋀ *f*. *f* ∈ *funs-term t* ⟹ *h f* = *f*) ⟹ *map-funs-term h t* = *t*
proof (*induct t*)
case (*Fun f ts*)
then have ⋀ *t*. *t* ∈ *set ts* ⟹ *map-funs-term h t* = *t* **by** *auto*
with *Fun*(*2*)[*of f*] **show** *?case*
by (*auto intro*: *nth-equalityI*)
qed *simp*

lemma *funs-term-map-funs-term*:
funs-term (*map-funs-term h t*) ⊆ *range h*
by (*induct t*) *auto*

fun *map-funs-subst* :: (*'f* ⇒ *'g*) ⇒ (*'f*, *'v*) *subst* ⇒ (*'g*, *'v*) *subst* **where**
map-funs-subst fg σ = (λ*x*. *map-funs-term fg* (*σ x*))

lemma *map-funs-term-comp*:
map-funs-term fg (*map-funs-term gh t*) = *map-funs-term* (*fg* ∘ *gh*) *t*
by (*induct t*) *simp-all*

lemma *map-funs-subst-distrib* [*simp*]:
map-funs-term fg (*t* · *σ*) = *map-funs-term fg t* · *map-funs-subst fg σ*
by (*induct t*) *simp-all*

lemma *size-map-funs-term* [*simp*]:
size (*map-funs-term fg t*) = *size t*
proof (*induct t*)
case (*Fun f ts*)

then show *?case* **by** (*induct ts*) *auto*
qed *simp*

lemma *fold-ident* [*simp*]: *Term-More.fold Var Fun t* = *t*
by (*induct t*) (*auto simp*: *map-ext* [*of - Term-More.fold Var Fun id*])

lemma *map-funs-term-ident* [*simp*]:
map-funs-term id t = *t*
by (*induct t*) (*simp-all add*: *map-idI*)

lemma *ground-map-funs-term* [*simp*]:
ground (*map-funs-term fg t*) = *ground t*
by (*induct t*) *auto*

lemma *map-funs-term-power*:
fixes *f* :: *'f* ⇒ *'f*
shows ((*map-funs-term f*) ^^ *n*) = *map-funs-term* (*f* ^^ *n*)
proof (*rule sym*, *intro ext*)
fix *t* :: (*'f*,*'v*)*term*
show *map-funs-term* (*f* ^^ *n*) *t* = (*map-funs-term f* ^^ *n*) *t*
proof (*induct n*)
case *0*
show *?case* **by** (*simp add*: *term.map-ident*)
next
case (*Suc n*)
show *?case* **by** (*simp add*: *Suc*[*symmetric*] *map-funs-term-comp o-def*)
qed
qed

lemma *map-funs-term-ctxt-distrib* [*simp*]:
map-funs-term fg (*C*⟨*t*⟩) = (*map-funs-ctxt fg C*)⟨*map-funs-term fg t*⟩
by (*induct C*) (*auto*)

mapping function symbols (w)ith (a)rities taken into account (wa)

fun *map-funs-term-wa* :: (*'f* × *nat* ⇒ *'g*) ⇒ (*'f*, *'v*) *term* ⇒ (*'g*, *'v*) *term*
where
map-funs-term-wa fg (*Var x*) = *Var x* |
map-funs-term-wa fg (*Fun f ts*) = *Fun* (*fg* (*f*, *length ts*)) (*map* (*map-funs-term-wa fg*) *ts*)

lemma *map-funs-term-map-funs-term-wa*:
map-funs-term (*fg* :: (*'f* ⇒ *'g*)) = *map-funs-term-wa* (λ (*f*,*n*). (*fg f*))
proof (*intro ext*)
fix *t* :: (*'f*,*'v*)*term*
show *map-funs-term fg t* = *map-funs-term-wa* (λ (*f*,*n*). *fg f*) *t*
by (*induct t*, *auto*)
qed

fun *map-funs-ctxt-wa* :: (*'f* × *nat* ⇒ *'g*) ⇒ (*'f*, *'v*) *ctxt* ⇒ (*'g*, *'v*) *ctxt*

where
map-funs-ctxt-wa fg $\square$ = $\square$ |
map-funs-ctxt-wa fg (*More f bef C aft*) =
More (*fg* (*f*, *Suc* (*length bef* + *length aft*))) (*map* (*map-funs-term-wa fg*) *bef*) (*map-funs-ctxt-wa fg C*) (*map* (*map-funs-term-wa fg*) *aft*)

abbreviation *map-funs-subst-wa* :: ('*f* × *nat* ⇒ '*g*) ⇒ ('*f*, '*v*) *subst* ⇒ ('*g*, '*v*) *subst* **where**
map-funs-subst-wa fg σ ≡ (λx. *map-funs-term-wa fg* (σ *x*))

lemma *map-funs-term-wa-subst* [*simp*]:
map-funs-term-wa fg (*t* · σ) = *map-funs-term-wa fg t* · *map-funs-subst-wa fg* σ
by (*induct t*, *auto*)

lemma *map-funs-term-wa-ctxt* [*simp*]:
map-funs-term-wa fg (*C*⟨*t*⟩) = (*map-funs-ctxt-wa fg C*) ⟨*map-funs-term-wa fg t*⟩
by (*induct C*, *auto*)

lemma *map-funs-term-wa-funas-term-id*:
assumes *t*: *funas-term t* ⊆ *F*
and *id*: ⋀ *f n*. (*f*,*n*) ∈ *F* ⟹ *fg* (*f*,*n*) = *f*
shows *map-funs-term-wa fg t* = *t*
using *t*
proof (*induct t*)
case (*Fun f ss*)
then have *IH*: ⋀ *s*. *s* ∈ *set ss* ⟹ *map-funs-term-wa fg s* = *s* **by** *auto*
from *Fun*(*2*) *id* **have** [*simp*]: *fg* (*f*, *length ss*) = *f* **by** *simp*
show *?case* **by** (*simp*, *insert IH*, *induct ss*, *auto*)
qed *simp*

lemma *funas-term-map-funs-term-wa*:
funas-term (*map-funs-term-wa fg t*) = (λ (*f*,*n*). (*fg* (*f*,*n*),*n*)) ‘ (*funas-term t*)
by (*induct t*) *auto*+

lemma *notin-subst-restrict* [*simp*]:
assumes *x* ∉ *V* **shows** (σ |*s* *V*) *x* = *Var x*
using *assms* **by** (*auto simp*: *subst-restrict-def*)

lemma *in-subst-restrict* [*simp*]:
assumes *x* ∈ *V* **shows** (σ |*s* *V*) *x* = σ *x*
using *assms* **by** (*auto simp*: *subst-restrict-def*)

lemma *coincidence-lemma'*:
assumes *vars-term t* ⊆ *V*
shows *t* · (σ |*s* *V*) = *t* · σ
using *assms* **by** (*metis in-mono in-subst-restrict term-subst-eq*)

lemma *vars-term-map-funs-term* [*simp*]:
vars-term ∘ *map-funs-term* (*f* :: ('*f* ⇒ '*g*)) = *vars-term*

proof
  fix t :: $('f,'v)term$
  show $(vars\text{-}term \circ map\text{-}funs\text{-}term\ f)\ t = vars\text{-}term\ t$
    by $(induct\ t)\ (auto)$
qed

lemma *vars-term-map-funs-term2* [*simp*]:
  $vars\text{-}term\ (map\text{-}funs\text{-}term\ f\ t) = vars\text{-}term\ t$
  using *fun-cong* [*OF vars-term-map-funs-term*, *of f t*]
  by (*simp del*: *vars-term-map-funs-term*)

lemma *map-funs-term-wa-ctxt-split*:
  assumes $map\text{-}funs\text{-}term\text{-}wa\ fg\ s = lC\langle lt\rangle$
  shows $\exists\ C\ t.\ s = C\langle t\rangle \wedge map\text{-}funs\text{-}term\text{-}wa\ fg\ t = lt \wedge map\text{-}funs\text{-}ctxt\text{-}wa\ fg\ C = lC$
  using *assms*
proof (*induct lC arbitrary*: *s*)
  case *Hole*
  show *?case*
    by (*rule exI*[*of - Hole*], *insert Hole*, *auto*)
next
  case (*More lf lbef lC laft s*)
  from *More*(*2*) **obtain** *fs ss* **where** *s*: $s = Fun\ fs\ ss$ **by** (*cases s*, *auto*)
  note *More* = *More*[*unfolded s*, *simplified*]
  let $?lb = length\ lbef$
  let $?la = length\ laft$
  let $?n = Suc\ (?lb + ?la)$
  let $?m = map\text{-}funs\text{-}term\text{-}wa\ fg$
  from *More*(*2*) **have** *rec*: $map\ ?m\ ss = lbef\ @\ lC\langle lt\rangle\ \#\ laft$
    and *lf*: $lf = fg\ (fs,length\ ss)$ **by** *blast+*
  from *arg-cong*[*OF rec*, *of length*] **have** *len*: $length\ ss = ?n$ **by** *auto*
  then have *lb*: $?lb < length\ ss$ **by** *auto*
  note $ss = id\text{-}take\text{-}nth\text{-}drop[OF\ this]$
  from *rec ss* **have** $map\ ?m\ (take\ ?lb\ ss\ @\ ss\ !\ ?lb\ \#\ drop\ (Suc\ ?lb)\ ss) = lbef\ @\ lC\langle lt\rangle\ \#\ laft$ **by** *auto*
  then have *id*: $take\ ?lb\ (map\ ?m\ ss)\ @\ ?m\ (ss\ !\ ?lb)\ \#\ drop\ (Suc\ ?lb)\ (map\ ?m\ ss) = lbef\ @\ lC\langle lt\rangle\ \#\ laft$
    (**is** $?l1\ @\ ?l2\ \#\ ?l3 = ?r1\ @\ ?r2\ \#\ ?r3$)
    unfolding *take-map drop-map*
    by *auto*
  from *len* **have** *len2*: $\bigwedge P.\ length\ ?l1 = length\ ?r1 \vee P$
    unfolding *length-take* **by** *auto*
  from *id*[*unfolded List.append-eq-append-conv*[*OF len2*]]
  have *id*: $?l1 = ?r1\ ?l2 = ?r2\ ?l3 = ?r3$ **by** *auto*
  from *More*(*1*)[*OF id*(*2*)] **obtain** *C t* **where** *sb*: $ss\ !\ ?lb = C\langle t\rangle$ **and** *map*: $map\text{-}funs\text{-}term\text{-}wa\ fg\ t = lt$ **and** *ma*: $map\text{-}funs\text{-}ctxt\text{-}wa\ fg\ C = lC$ **by** *auto*
  let $?C = More\ fs\ (take\ ?lb\ ss)\ C\ (drop\ (Suc\ ?lb)\ ss)$
  have *s*: $s = ?C\langle t\rangle$
    unfolding *s* **using** *ss*[*unfolded sb*] **by** *simp*

have *len3*: *Suc* (*length* (*take ?lb ss*) + *length* (*drop* (*Suc ?lb*) *ss*)) = *length ss*
unfolding *length-take length-drop len* **by** *auto*
show *?case*
proof (*intro exI conjI*, *rule s*, *rule map*)
show *map-funs-ctxt-wa fg ?C* = *More lf lbef lC laft*
unfolding *map-funs-ctxt-wa.simps*
unfolding *len3*
using *id ma lf*
unfolding *take-map drop-map*
by *auto*
qed
qed

lemma *subst-extend-flat-ctxt*:
assumes *dist*: *distinct vs*
and *len1*: *length*(*take i* (*map Var vs*)) = *length ss1*
and *len2*: *length*(*drop* (*Suc i*) (*map Var vs*)) = *length ss2*
and *i*: *i* < *length vs*
shows *More f* (*take i* (*map Var vs*)) □ (*drop* (*Suc i*) (*map Var vs*)) $\cdot_c$ *subst-extend σ* (*zip* (*take i vs@drop* (*Suc i*) *vs*) (*ss1@ss2*)) = *More f ss1* □ *ss2*
proof −
let *?V* = *map Var vs*
let *?vs1* = *take i vs* **and** *?vs2* = *drop* (*Suc i*) *vs*
let *?ss1* = *take i ?V* **and** *?ss2* = *drop* (*Suc i*) *?V*
let *?σ* = *subst-extend σ* (*zip* (*?vs1@?vs2*) (*ss1@ss2*))
from *len1* **and** *len2* **have** *len*: *length*(*?vs1@?vs2*) = *length*(*ss1@ss2*) **using** *i* **by** *simp*
from *dist i* **have** *distinct*(*?vs1@?vs2*)
by (*simp add*: *set-take-disj-set-drop-if-distinct*)
from *subst-extend-absorb*[*OF this len*,*of σ*]
have *map*: *map* (λ*t*. *t*·*?σ*) (*?ss1@?ss2*) = *ss1@ss2* **unfolding** *take-map drop-map map-append* **.**
from *len1* **and** *map* **have** *map* (λ*t*. *t*·*?σ*) *?ss1* = *ss1* **by** *auto*
moreover from *len2* **and** *map* **have** *map* (λ*t*. *t*·*?σ*) *?ss2* = *ss2* **by** *auto*
ultimately show *?thesis* **by** *simp*
qed

lemma *subst-extend-flat-ctxt''*:
assumes *dist*: *distinct vs*
and *len1*: *length*(*take i* (*map Var vs*)) = *length ss1*
and *len2*: *length*(*drop i* (*map Var vs*)) = *length ss2*
and *i*: *i* < *length vs*
shows *More f* (*take i* (*map Var vs*)) □ (*drop i* (*map Var vs*)) $\cdot_c$ *subst-extend σ* (*zip* (*take i vs@drop i vs*) (*ss1@ss2*)) = *More f ss1* □ *ss2*
proof −
let *?V* = *map Var vs*
let *?vs1* = *take i vs* **and** *?vs2* = *drop i vs*
let *?ss1* = *take i ?V* **and** *?ss2* = *drop i ?V*
let *?σ* = *subst-extend σ* (*zip* (*?vs1@?vs2*) (*ss1@ss2*))

from *len1* **and** *len2* **have** *len*: *length*(*?vs1*@*?vs2*) = *length*(*ss1*@*ss2*) **using** *i* **by** *simp*
have *distinct*(*?vs1*@*?vs2*) **using** *dist* **unfolding** *append-take-drop-id* **by** *simp*
from *subst-extend-absorb*[*OF this len*,*of* σ]
have *map*: *map* ($\lambda t.\ t\cdot ?\sigma$) (*?ss1*@*?ss2*) = *ss1*@*ss2* **unfolding** *take-map drop-map map-append* **.**
from *len1* **and** *map* **have** *map* ($\lambda t.\ t\cdot ?\sigma$) *?ss1* = *ss1* **unfolding** *map-append* **by** *auto*
moreover from *len2* **and** *map* **have** *map* ($\lambda t.\ t\cdot ?\sigma$) *?ss2* = *ss2* **unfolding** *map-append* **by** *auto*
ultimately show *?thesis* **by** *simp*
qed

lemma *distinct-map-Var*:
assumes *distinct xs* **shows** *distinct* (*map Var xs*)
using *assms* **by** (*induct xs*) *auto*

lemma *variants-imp-is-Var*:
assumes $s \cdot \sigma = t$ **and** $t \cdot \tau = s$
shows $\forall x \in$*vars-term s*. *is-Var* (σ *x*)
using *assms*
proof (*induct s arbitrary*: *t*)
case (*Var x*)
then show *?case* **by** (*cases* σ *x*) *auto*
next
case (*Fun f ts*)
then show *?case*
by (*auto simp*: *o-def*) (*metis map-eq-conv map-ident*)
qed

The range (in a functional sense) of a substitution.

definition *subst-fun-range* :: ('*f*, '*v*, '*w*) *gsubst* $\Rightarrow$ '*w set*
where
subst-fun-range σ = $\bigcup$(*vars-term* ‘ *range* σ)

lemma *subst-variants-imp-is-Var*:
assumes $\sigma \circ_s \sigma' = \tau$ **and** $\tau \circ_s \tau' = \sigma$
shows $\forall x \in$*subst-fun-range* σ. *is-Var* (σ' *x*)
using *assms* **by** (*auto simp*: *subst-compose-def subst-fun-range-def*) (*metis variants-imp-is-Var*)

lemma *variants-imp-image-vars-term-eq*:
assumes $s \cdot \sigma = t$ **and** $t \cdot \tau = s$
shows (*the-Var* $\circ$ σ) ‘ *vars-term s* = *vars-term t*
using *assms*
proof (*induct s arbitrary*: *t*)
case (*Var x*)
then show *?case* **by** (*cases t*) *auto*
next

case (*Fun f ss*)
then have *IH*: $\bigwedge t.\ \forall i{<}length\ ss.\ (ss\ !\ i) \cdot \sigma = t \wedge t \cdot \tau = ss\ !\ i \longrightarrow$
(*the-Var* $\circ\ \sigma$) ‘ *vars-term* (*ss* ! *i*) = *vars-term t*
by (*auto simp*: *o-def*)
from *Fun.prems* **have** *t*: *t* = *Fun f* (*map* ($\lambda t.\ t \cdot \sigma$) *ss*)
and *ss*: *ss* = *map* ($\lambda t.\ t \cdot \sigma \cdot \tau$) *ss* **by** (*auto simp*: *o-def*)
have $\forall i{<}length\ ss.$ (*the-Var* $\circ\ \sigma$) ‘ *vars-term* (*ss* ! *i*) = *vars-term* (*ss* ! *i* $\cdot\ \sigma$)
proof (*intro allI impI*)
fix *i*
assume $*$: *i* < *length ss*
have (*ss* ! *i*) $\cdot\ \sigma$ = (*ss* ! *i*) $\cdot\ \sigma$ **by** *simp*
moreover have (*ss* ! *i*) $\cdot\ \sigma \cdot \tau$ = *ss* ! *i*
using $*$ **by** (*subst* (*2*) *ss*) *simp*
ultimately show (*the-Var* $\circ\ \sigma$) ‘ *vars-term* (*ss* ! *i*) = *vars-term* ((*ss* ! *i*) $\cdot\ \sigma$)
using *IH* **and** $*$ **by** *blast*
qed
then have $\forall s{\in}set\ ss.$ (*the-Var* $\circ\ \sigma$) ‘ *vars-term s* = *vars-term* ($s \cdot \sigma$) **by** (*metis in-set-conv-nth*)
then show *?case* **by** (*simp add*: *o-def t image-UN*)
qed

lemma *terms-to-vars*:
assumes $\forall t{\in}set\ ts.$ *is-Var t*
shows $\bigcup$(*set* (*map vars-term ts*)) = *set* (*map the-Var ts*)
using *assms* **by** (*induct ts*) *auto*

lemma *Var-the-Var-id*:
assumes $\forall t{\in}set\ ts.$ *is-Var t*
shows *map Var* (*map the-Var ts*) = *ts*
using *assms* **by** (*induct ts*) *auto*

lemma *distinct-the-vars*:
assumes $\forall t{\in}set\ ts.$ *is-Var t*
and *distinct ts*
shows *distinct* (*map the-Var ts*)
using *assms* **by** (*induct ts*) *auto*

lemma *map-funs-term-eq-imp-map-funs-term-map-vars-term-eq*:
map-funs-term fg s = *map-funs-term fg t* $\Longrightarrow$ *map-funs-term fg* (*map-vars-term vw s*) = *map-funs-term fg* (*map-vars-term vw t*)
proof (*induct s arbitrary*: *t*)
case (*Var x t*)
then show *?case* **by** (*cases t*, *auto*)
next
case (*Fun f ss t*)
then obtain *g ts* **where** *t*: *t* = *Fun g ts* **by** (*cases t*, *auto*)
from *Fun*(*2*)[*unfolded t*, *simplified*]
have *f*: *fg f* = *fg g* **and** *ss*: *map* (*map-funs-term fg*) *ss* = *map* (*map-funs-term fg*) *ts* **by** *auto*

from *arg-cong*[*OF ss*, *of length*] **have** *length ss* = *length ts* **by** *simp*
from *this ss Fun*(*1*) **have** *map* (*map-funs-term fg* ∘ *map-vars-term vw*) *ss* = *map* (*map-funs-term fg* ∘ *map-vars-term vw*) *ts*
by (*induct ss ts rule*: *list-induct2*, *auto*)
then show *?case* **unfolding** *t* **by** (*simp add*: *f*)
qed

lemma *var-type-conversion*:
assumes *inf*: *infinite* (*UNIV* :: *'v set*)
and *fin*: *finite* (*T* :: (*'f*, *'w*) *terms*)
shows ∃ (*σ* :: (*'f*, *'w*, *'v*) *gsubst*) *τ*. ∀ *t*∈*T*. *t* = *t* · *σ* · *τ*
proof −
obtain *V* **where** *V*: *V* = ⋃(*vars-term* ‘ *T*) **by** *auto*
have *fin*: *finite V* **unfolding** *V*
by (*rule*, *rule*, *rule fin*,
insert finite-vars-term, *auto*)
from *finite-imp-inj-to-nat-seg*[*OF fin*] **obtain** *to-nat* :: *'w* ⇒ *nat* **and** *n* :: *nat*
where *to-nat*: *to-nat* ‘ *V* = {*i*. *i* < *n*} *inj-on to-nat V* **by** *blast*+
from *infinite-countable-subset*[*OF inf*] **obtain** *of-nat* :: *nat* ⇒ *'v* **where**
of-nat: *range of-nat* ⊆ *UNIV inj of-nat* **by** *auto*
let *?conv* = λ *v*. *of-nat* (*to-nat v*)
have *inj*: *inj-on ?conv V* **using** *of-nat*(*2*) *to-nat*(*2*) **unfolding** *inj-on-def* **by** *auto*
let *?rev* = *the-inv-into V ?conv*
note *rev* = *the-inv-into-f-eq*[*OF inj*]
obtain *σ* **where** *σ*: *σ* = (λ *v*. *Var* (*?conv v*) :: (*'f*,*'v*)*term*) **by** *simp*
obtain *τ* **where** *τ*: *τ* = (λ *v*. *Var* (*?rev v*) :: (*'f*,*'w*)*term*) **by** *simp*
show *?thesis*
proof (*rule exI*[*of - σ*], *rule exI*[*of - τ*], *intro ballI*)
fix *t*
assume *t*: *t* ∈ *T*
have *t* · *σ* · *τ* = *t* · (*σ* $\circ_s$ *τ*) **by** *simp*
also have ... = *t* · *Var*
proof (*rule term-subst-eq*)
fix *x*
assume *x* ∈ *vars-term t*
with *t* **have** *x*: *x* ∈ *V* **unfolding** *V* **by** *auto*
show (*σ* $\circ_s$ *τ*) *x* = *Var x* **unfolding** *σ τ subst-compose-def*
eval-term.simps term.simps
by (*rule rev*[*OF refl x*])
qed
finally show *t* = *t* · *σ* · *τ* **by** *simp*
qed
qed

combine two substitutions via sum-type

fun
merge-substs :: (*'f*, *'u*, *'v*) *gsubst* ⇒ (*'f*, *'w*, *'v*) *gsubst* ⇒ (*'f*, *'u* + *'w*, *'v*) *gsubst*
where

merge-substs σ τ = (λx.
(*case* x *of*
Inl $y \Rightarrow \sigma$ y
| *Inr* $y \Rightarrow \tau$ y))

lemma *merge-substs-left*:
map-vars-term Inl $s \cdot$ (*merge-substs* σ δ) = $s \cdot \sigma$
by (*induct* s) *auto*

lemma *merge-substs-right*:
map-vars-term Inr $s \cdot$ (*merge-substs* σ δ) = $s \cdot \delta$
by (*induct* s) *auto*

fun *map-vars-subst-ran* :: ($'w \Rightarrow 'u$) $\Rightarrow$ ($'f$, $'v$, $'w$) *gsubst* $\Rightarrow$ ($'f$, $'v$, $'u$) *gsubst*
where
map-vars-subst-ran m σ = (λv. *map-vars-term* m (σ v))

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

lemma *size-subst*: *size* $t \leq$ *size* ($t \cdot \sigma$)
proof (*induct* t)
case (*Var* x)
then show *?case* **by** (*cases* σ x) *auto*
next
case (*Fun* f ss)
then show *?case*
by (*simp add*: *o-def*, *induct ss*, *force*+)
qed

lemma *eq-ctxt-subst-iff* [*simp*]:
(t = $C\langle t \cdot \sigma\rangle$) $\longleftrightarrow$ $C = \square \wedge (\forall x \in$*vars-term* t. σ x = *Var* x) (**is** *?L* = *?R*)
proof
assume t: *?L*
then have *size* t = *size* ($C\langle t \cdot \sigma\rangle$) **by** *simp*
with *size-ne-ctxt* [*of* C $t \cdot \sigma$] **and** *size-subst* [*of* t σ]
have [*simp*]: $C = \square$ **by** *auto*
have $\forall x \in$*vars-term* t. σ x = *Var* x **using** t **and** *term-subst-eq-conv* [*of* t *Var*]
by *simp*
then show *?R* **by** *auto*
next
assume *?R*
then show *?L* **using** *term-subst-eq-conv* [*of* t *Var*] **by** *simp*
qed

lemma *Fun-Nil-supt*[*elim*!]: *Fun* f [] $\rhd$ $t \Longrightarrow P$ **by** *auto*

lemma *map-vars-term-vars-term*:

assumes $\bigwedge x.\ x \in$ *vars-term t* $\Longrightarrow f\ x = g\ x$
shows *map-vars-term f t* = *map-vars-term g t*
using *assms*
proof (*induct t*)
case (*Fun h ts*)
{
fix *t*
assume *t*: $t \in$ *set ts*
with *Fun*(*2*) **have** $\bigwedge x.\ x \in$ *vars-term t* $\Longrightarrow f\ x = g\ x$
by *auto*
from *Fun*(*1*)[*OF t this*] **have** *map-vars-term f t* = *map-vars-term g t* **by** *simp*
}
then show *?case* **by** *auto*
qed *simp*

lemma *map-funs-term-ctxt-decomp*:
assumes *map-funs-term fg t* = $C\langle s\rangle$
shows $\exists$ *D u*. *C* = *map-funs-ctxt fg D* $\wedge$ *s* = *map-funs-term fg u* $\wedge$ *t* = $D\langle u\rangle$
using *assms*
proof (*induct C arbitrary*: *t*)
case *Hole*
show *?case*
by (*rule exI*[*of - Hole*], *rule exI*[*of - t*], *insert Hole*, *auto*)
next
case (*More g bef C aft*)
from *More*(*2*) **obtain** *f ts* **where** *t*: *t* = *Fun f ts* **by** (*cases t*, *auto*)
from *More*(*2*)[*unfolded t*] **have** *f*: *fg f* = *g* **and** *ts*: *map* (*map-funs-term fg*) *ts* = *bef* @ $C\langle s\rangle$ # *aft* (**is** *?ts* = *?bca*) **by** *auto*
from *ts* **have** *length ?ts* = *length ?bca* **by** *auto*
then have *len*: *length ts* = *length ?bca* **by** *auto*
let *?i* = *length bef*
from *len* **have** *i*: *?i* < *length ts* **by** *auto*
from *arg-cong*[*OF ts*, *of* λ *xs*. *xs* ! *?i*] *len*
have *map-funs-term fg* (*ts* ! *?i*) = $C\langle s\rangle$ **by** *auto*
from *More*(*1*)[*OF this*] **obtain** *D u* **where** *D*: *C* = *map-funs-ctxt fg D* **and**
u: *s* = *map-funs-term fg u* **and** *id*: *ts* ! *?i* = $D\langle u\rangle$ **by** *auto*
from *ts* **have** *take ?i ?ts* = *take ?i ?bca* **by** *simp*
also have ... = *bef* **by** *simp*
finally have *bef*: *map* (*map-funs-term fg*) (*take ?i ts*) = *bef* **by** (*simp add*: *take-map*)
from *ts* **have** *drop* (*Suc ?i*) *?ts* = *drop* (*Suc ?i*) *?bca* **by** *simp*
also have ... = *aft* **by** *simp*
finally have *aft*: *map* (*map-funs-term fg*) (*drop* (*Suc ?i*) *ts*) = *aft* **by** (*simp add*:*drop-map*)
let *?bda* = *take ?i ts* @ $D\langle u\rangle$ # *drop* (*Suc ?i*) *ts*
show *?case*
proof (*rule exI*[*of - More f* (*take ?i ts*) *D* (*drop* (*Suc ?i*) *ts*)],
rule exI[*of - u*], *simp add*: *u f D bef aft t*)
have *ts* = *take ?i ts* @ *ts* ! *?i* # *drop* (*Suc ?i*) *ts*

```
      by (rule id-take-nth-drop[OF i])
    also have ... = ?bda by (simp add: id)
    finally show ts = ?bda .
  qed
qed

lemma funas-term-map-vars-term [simp]:
  funas-term (map-vars-term τ t) = funas-term t
  by (induct t) auto

lemma funs-term-funas-term:
  funs-term t = fst ‘ (funas-term t)
  by (induct t) auto

lemma funas-term-map-funs-term:
  funas-term (map-funs-term fg t) = (λ (f,n). (fg f,n)) ‘ (funas-term t)
  by (induct t) auto+

lemma supt-imp-arg-or-supt-of-arg:
  assumes Fun f ss ⊳ t
  shows t ∈ set ss ∨ (∃ s ∈ set ss. s ⊳ t)
  using assms by (rule supt.cases) auto

lemma supt-Fun-imp-arg-supteq:
  assumes Fun f ss ⊳ t shows ∃ s ∈ set ss. s ⊵ t
  using assms by (cases rule: supt.cases) auto

lemma subt-iff-eq-or-subt-of-arg:
  assumes s = Fun f ss
  shows {t. s ⊵ t} = ((⋃ u ∈ set ss. {t. u ⊵ t})∪{s})
  using assms proof (induct s)
  case (Var x) then show ?case by auto
next
  case (Fun g ts)
  then have g = f and ts = ss by auto
  show ?case
  proof
    show {a. Fun g ts ⊵ a} ⊆ (⋃ u∈set ss. {a. u ⊵ a}) ∪ {Fun g ts}
    proof
      fix x
      assume x ∈ {a. Fun g ts ⊵ a}
      then have Fun g ts ⊵ x by simp
      then have Fun g ts ⊳ x ∨ Fun g ts = x by auto
      then show x ∈ (⋃ u∈set ss. {a. u ⊵a}) ∪ {Fun g ts}
      proof
        assume Fun g ts ⊳ x
       then obtain u where u ∈ set ts and u ⊵ x using supt-Fun-imp-arg-supteq
by best
        then have x ∈ {a. u ⊵ a} by simp
```

```
      with ‹u ∈ set ts› have x ∈ (⋃u∈set ts. {a. u ⊵ a}) by auto
      then show ?thesis unfolding ‹ts = ss› by simp
    next
      assume Fun g ts = x then show ?thesis by simp
    qed
  qed
next
  show (⋃u∈set ss. {a. u ⊵ a}) ∪ {Fun g ts} ⊆ {a. Fun g ts ⊵ a}
  proof
    fix x
    assume x ∈ (⋃u∈set ss. {a. u ⊵ a}) ∪ {Fun g ts}
    then have x ∈ (⋃u∈set ss. {a. u ⊵ a}) ∨ x = Fun g ts by auto
    then show x ∈ {a. Fun g ts ⊵ a}
    proof
      assume x ∈ (⋃u∈set ss. {a. u ⊵ a})
      then obtain u where u ∈ set ss and u ⊵ x by auto
      then show ?thesis unfolding ‹ts = ss› by auto
    next
      assume x = Fun g ts then show ?thesis by auto
    qed
  qed
qed
qed
```

The set of subterms of a term is finite.

```
lemma finite-subterms: finite {s. t ⊵ s}
proof (induct t)
  case (Var x)
  then have ⋀s. (Var x ⊵ s) = (Var x = s) using supteq.cases by best
  then show ?case unfolding ‹⋀s. (Var x ⊵ s) = (Var x = s)› by simp
next
  case (Fun f ss)
  have Fun f ss = Fun f ss by simp
  from Fun show ?case
    unfolding subt-iff-eq-or-subt-of-arg[OF ‹Fun f ss = Fun f ss›] by auto
qed

lemma Fun-supteq: Fun f ts ⊵ u ⟷ Fun f ts = u ∨ (∃ t∈set ts. t ⊵ u)
  using subt-iff-eq-or-subt-of-arg[of Fun f ts f ts] by auto

lemma subst-ctxt-distr: s = C⟨t⟩·σ ⟹ ∃ D. s = D⟨t·σ⟩
  using subst-apply-term-ctxt-apply-distrib by auto

lemma ctxt-of-pos-term-subst:
  assumes p ∈ poss t
  shows ctxt-of-pos-term p (t · σ) = ctxt-of-pos-term p t ·c σ
  using assms
proof (induct p arbitrary: t)
  case (Cons i p t)
```

then obtain *f ts* **where** *t*: $t = Fun\ f\ ts$ **and** *i*: $i < length\ ts$ **and** *p*: $p \in poss\ (ts\ !\ i)$ **by** (*cases t*, *auto*)
note *id* = *id-take-nth-drop*[*OF i*, *symmetric*]
with *t* **have** *t*: $t = Fun\ f\ (take\ i\ ts\ @\ ts\ !\ i\ \#\ drop\ (Suc\ i)\ ts)$ **by** *auto*
from *i* **have** *i'*: $min\ (length\ ts)\ i = i$ **by** *simp*
show *?case* **unfolding** *t* **using** *Cons*(*1*)[*OF p*, *symmetric*] *i'*
by (*simp add*: *id*, *insert i*, *auto simp*: *take-map drop-map*)
qed *simp*

lemma *subt-at-ctxt-of-pos-term*:
assumes *t*: $(ctxt\text{-}of\text{-}pos\text{-}term\ p\ t)\langle u\rangle = t$ **and** *p*: $p \in poss\ t$
shows $t\ |\text{-}\ p = u$
proof −
let $?C = ctxt\text{-}of\text{-}pos\text{-}term\ p\ t$
from *t* **and** *ctxt-supt-id* [*OF p*] **have** $?C\langle u\rangle = ?C\langle t\ |\text{-}\ p\rangle$ **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *subst-ext*:
assumes $\forall x \in V.\ \sigma\ x = \tau\ x$ **shows** $\sigma\ |s\ V = \tau\ |s\ V$
proof
fix *x* **show** $(\sigma\ |s\ V)\ x = (\tau\ |s\ V)\ x$ **using** *assms*
unfolding *subst-restrict-def* **by** (*cases* $x \in V$) *auto*
qed

abbreviation $map\text{-}vars\text{-}ctxt\ f \equiv ctxt.map\text{-}ctxt\ (\lambda x.\ x)\ f$

lemma *map-vars-term-ctxt-commute*:
$map\text{-}vars\text{-}term\ m\ (c\langle t\rangle) = (map\text{-}vars\text{-}ctxt\ m\ c)\langle map\text{-}vars\text{-}term\ m\ t\rangle$
by (*induct c*) *auto*

lemma *map-vars-term-inj-compose*:
assumes *inj*: $\bigwedge x.\ n\ (m\ x) = x$
shows $map\text{-}vars\text{-}term\ n\ (map\text{-}vars\text{-}term\ m\ t) = t$
unfolding *map-vars-term-compose o-def inj* **by** (*auto simp*: *term.map-ident*)

lemma *inj-map-vars-term-the-inv*:
assumes *inj f*
shows $map\text{-}vars\text{-}term\ (the\text{-}inv\ f)\ (map\text{-}vars\text{-}term\ f\ t) = t$
unfolding *map-vars-term-compose o-def the-inv-f-f*[*OF assms*]
by (*simp add*: *term.map-ident*)

lemma *map-vars-ctxt-subst*:
$map\text{-}vars\text{-}ctxt\ m\ (C \cdot_c \sigma) = C \cdot_c map\text{-}vars\text{-}subst\text{-}ran\ m\ \sigma$
by (*induct C*) (*auto simp*: *map-vars-subst-ran*)

lemma *poss-map-vars-term* [*simp*]:
$poss\ (map\text{-}vars\text{-}term\ f\ t) = poss\ t$
by (*induct t*) *auto*

lemma *map-vars-term-subt-at* [*simp*]:
  $p \in$ *poss t* $\Longrightarrow$ *map-vars-term f* (*t* |- *p*) = (*map-vars-term f t*) |- *p*
proof (*induct p arbitrary*: *t*)
  case *Nil* **show** *?case* **by** *auto*
next
  case (*Cons i p t*)
  from *Cons*(*2*) **obtain** *g ts* **where** *t*: *t* = *Fun g ts* **by** (*cases t*, *auto*)
  from *Cons* **show** *?case* **unfolding** *t* **by** *auto*
qed

lemma *hole-pos-subst*[*simp*]: *hole-pos* ($C \cdot_c \sigma$) = *hole-pos C*
  by (*induct C*, *auto*)

lemma *hole-pos-ctxt-compose*[*simp*]: *hole-pos* ($C \circ_c D$) = *hole-pos C* @ *hole-pos D*
  by (*induct C*, *auto*)

lemma *subst-left-right*: $t \cdot \mu \hat{\hat{}} n \cdot \mu = t \cdot \mu \cdot \mu \hat{\hat{}} n$
proof −
  have $t \cdot \mu \hat{\hat{}} n \cdot \mu = t \cdot (\mu \hat{\hat{}} n \circ_s \mu)$ **by** *simp*
  also have ... $= t \cdot (\mu \circ_s \mu \hat{\hat{}} n)$
    using *subst-power-compose-distrib*[*of n Suc 0* μ] **by** *auto*
  finally show *?thesis* **by** *simp*
qed

lemma *subst-right-left*: $t \cdot \mu \cdot \mu \hat{\hat{}} n = t \cdot \mu \hat{\hat{}} n \cdot \mu$ **unfolding** *subst-left-right* **..**

lemma *subt-at-id-imp-eps*:
  assumes *p*: $p \in$ *poss t* **and** *id*: *t* |- *p* = *t*
  shows *p* = []
proof (*cases p*)
  case (*Cons i q*)
  from *subt-at-nepos-imp-supt*[*OF p*[*unfolded Cons*], *unfolded Cons*[*symmetric*]
      , *unfolded id*] **have** *False* **by** *simp*
  then show *?thesis* **by** *auto*
qed *simp*

lemma *pos-into-subst*:
  assumes *t*: $t \cdot \sigma = s$ **and** *p*: $p \in$ *poss s* **and** *nt*: $\neg$ ($p \in$ *poss t* $\wedge$ *is-Fun* (*t* |- *p*))
  shows $\exists$ *q q′ x*. *p* = *q* @ *q′* $\wedge$ $q \in$ *poss t* $\wedge$ *t* |- *q* = *Var x*
  using *p nt* **unfolding** *t*[*symmetric*]
proof (*induct t arbitrary*: *p*)
  case (*Var x*)
  show *?case*
    by (*rule exI*[*of* - []], *rule exI*[*of* - *p*], *rule exI*[*of* - *x*], *insert Var*, *auto*)
next
  case (*Fun f ts*)
  from *Fun*(*3*) **obtain** *i q* **where** *p*: *p* = *i* # *q* **by** (*cases p*, *auto*)

note *Fun* = *Fun*[*unfolded p*]
from *Fun*(*2*) **have** *i*: *i* < *length ts* **and** *q*: *q* ∈ *poss* (*ts* ! *i* · σ) **by** *auto*
then have *mem*: *ts* ! *i* ∈ *set ts* **by** *auto*
from *Fun*(*3*) *i* **have** ¬ (*q* ∈ *poss* (*ts* ! *i*) ∧ *is-Fun* (*ts* ! *i* |- *q*)) **by** *auto*
from *Fun*(*1*)[*OF mem q this*]
obtain *r r′ x* **where** *q*: *q* = *r* @ *r′* ∧ *r* ∈ *poss* (*ts* ! *i*) ∧ *ts* ! *i* |- *r* = *Var x* **by** *blast*
show *?case*
by (*rule exI*[*of - i # r*], *rule exI*[*of - r′*], *rule exI*[*of - x*],
unfold p, *insert i q*, *auto*)
qed

abbreviation (*input*) *replace-at t p s* ≡ (*ctxt-of-pos-term p t*)⟨*s*⟩

lemma *replace-at-ident*:
assumes *p* ∈ *poss t* **and** *t* |- *p* = *s*
shows *replace-at t p s* = *t*
using *assms* **by** (*metis ctxt-supt-id*)

lemma *ctxt-of-pos-term-append*:
assumes *p* ∈ *poss t*
shows *ctxt-of-pos-term* (*p* @ *q*) *t* = *ctxt-of-pos-term p t* $\circ_c$ *ctxt-of-pos-term q* (*t* |- *p*)
using *assms*
proof (*induct p arbitrary*: *t*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons i p t*)
from *Cons*(*2*) **obtain** *f ts* **where** *t*: *t* = *Fun f ts* **and** *i*: *i* < *length ts* **and** *p*: *p* ∈ *poss* (*ts* ! *i*) **by** (*cases t*, *auto*)
from *Cons*(*1*)[*OF p*]
show *?case* **unfolding** *t* **using** *i* **by** *auto*
qed

lemma *parallel-replace-at*:
fixes *p1* :: *pos*
assumes *parallel*: *p1* ⊥ *p2*
and *p1*: *p1* ∈ *poss t*
and *p2*: *p2* ∈ *poss t*
shows *replace-at* (*replace-at t p1 s1*) *p2 s2* = *replace-at* (*replace-at t p2 s2*) *p1 s1*
proof −
from *parallel-remove-prefix*[*OF parallel*]
obtain *p i j q1 q2* **where** *p1-id*: *p1* = *p* @ *i* # *q1* **and** *p2-id*: *p2* = *p* @ *j* # *q2*
and *ij*: *i* ≠ *j* **by** *blast*
from *p1 p2* **show** *?thesis* **unfolding** *p1-id p2-id*
proof (*induct p arbitrary*: *t*)
case (*Cons k p*)

from *Cons*(*3*) **obtain** *f ts* **where** *t*: *t* = *Fun f ts* **and** *k*: *k* < *length ts* **by** (*cases t*, *auto*)
note *Cons* = *Cons*[*unfolded t*]
let *?p1* = *p* @ *i* # *q1* **let** *?p2* = *p* @ *j* # *q2*
from *Cons*(*2*) *Cons*(*3*) **have** *?p1* ∈ *poss* (*ts* ! *k*) *?p2* ∈ *poss* (*ts* ! *k*) **by** *auto*
from *Cons*(*1*)[*OF this*] **have** *rec*: *replace-at* (*replace-at* (*ts* ! *k*) *?p1 s1*) *?p2 s2* = *replace-at* (*replace-at* (*ts* ! *k*) *?p2 s2*) *?p1 s1* **.**
from *k* **have** *min*: *min* (*length ts*) *k* = *k* **by** *simp*
show *?case* **unfolding** *t* **using** *rec min k*
by (*simp add*: *nth-append*)
next
case *Nil*
from *Nil*(*2*) **obtain** *f ts* **where** *t*: *t* = *Fun f ts* **and** *j*: *j* < *length ts* **by** (*cases t*, *auto*)
note *Nil* = *Nil*[*unfolded t*]
from *Nil* **have** *i*: *i* < *length ts* **by** *auto*
let *?p1* = *i* # *q1*
let *?p2* = *j* # *q2*
let *?s1* = *replace-at* (*ts* ! *i*) *q1 s1*
let *?s2* = *replace-at* (*ts* ! *j*) *q2 s2*
let *?ts1* = *ts*[*i* := *?s1*]
let *?ts2* = *ts*[*j* := *?s2*]
from *j* **have** *j'*: *j* < *length ?ts1* **by** *simp*
from *i* **have** *i'*: *i* < *length ?ts2* **by** *simp*
have *replace-at* (*replace-at t ?p1 s1*) *?p2 s2* = *replace-at* (*Fun f ?ts1*) *?p2 s2*
unfolding *t upd-conv-take-nth-drop*[*OF i*] **by** *simp*
also have ... = *Fun f* (*?ts1*[*j* := *?s2*])
unfolding *upd-conv-take-nth-drop*[*OF j'*] **using** *ij* **by** *simp*
also have ... = *Fun f* (*?ts2*[*i* := *?s1*]) **using** *list-update-swap*[*OF ij*]
by *simp*
also have ... = *replace-at* (*Fun f ?ts2*) *?p1 s1*
unfolding *upd-conv-take-nth-drop*[*OF i'*] **using** *ij* **by** *simp*
also have ... = *replace-at* (*replace-at t ?p2 s2*) *?p1 s1* **unfolding** *t*
upd-conv-take-nth-drop[*OF j*] **by** *simp*
finally show *?case* **by** *simp*
qed
qed

lemma *parallel-replace-at-subt-at*:
fixes *p1* :: *pos*
assumes *parallel*: *p1* ⊥ *p2*
and *p1*: *p1* ∈ *poss t*
and *p2*: *p2* ∈ *poss t*
shows (*replace-at t p1 s1*) |- *p2* = *t* |- *p2*
proof −
from *parallel-remove-prefix*[*OF parallel*]
obtain *p i j q1 q2* **where** *p1-id*: *p1* = *p* @ *i* # *q1* **and** *p2-id*: *p2* = *p* @ *j* # *q2*
and *ij*: *i* ≠ *j* **by** *blast*
from *p1 p2* **show** *?thesis* **unfolding** *p1-id p2-id*

```
  proof (induct p arbitrary: t)
    case (Cons k p)
     from Cons(3) obtain f ts where t: t = Fun f ts and k: k < length ts by
(cases t, auto)
    note Cons = Cons[unfolded t]
    let ?p1 = p @ i # q1 let ?p2 = p @ j # q2
    from Cons(2) Cons(3) have ?p1 ∈ poss (ts ! k) ?p2 ∈ poss (ts ! k) by auto
    from Cons(1)[OF this] have rec: (replace-at (ts ! k) ?p1 s1) |- ?p2 = (ts ! k)
|- ?p2 .
    from k have min: min (length ts) k = k by simp
    show ?case unfolding t using rec min k
      by (simp add: nth-append)
  next
    case Nil
    from Nil(2) obtain f ts where t: t = Fun f ts and j: j < length ts by (cases
t, auto)
    note Nil = Nil[unfolded t]
    from Nil have i: i < length ts by auto
    let ?p1 = i # q1
    let ?p2 = j # q2
    let ?s1 = replace-at (ts ! i) q1 s1
    let ?ts1 = ts[i := ?s1]
    from j have j': j < length ?ts1 by simp
   have (replace-at t ?p1 s1) |- ?p2 = (Fun f ?ts1) |- ?p2 unfolding t upd-conv-take-nth-drop[OF
i] by simp
    also have ... = ts ! j |- q2 using ij by simp
    finally show ?case unfolding t by simp
  qed
qed

lemma parallel-poss-replace-at:
  fixes p1 :: pos
  assumes parallel: p1 ⊥ p2
    and p1: p1 ∈ poss t
  shows (p2 ∈ poss (replace-at t p1 s1)) = (p2 ∈ poss t)
proof −
  from parallel-remove-prefix[OF parallel]
  obtain p i j q1 q2 where p1-id: p1 = p @ i # q1 and p2-id: p2 = p @ j # q2
    and ij: i ≠ j by blast
  from p1 show ?thesis unfolding p1-id p2-id
  proof (induct p arbitrary: t)
    case (Cons k p)
     from Cons(2) obtain f ts where t: t = Fun f ts and k: k < length ts by
(cases t, auto)
    note Cons = Cons[unfolded t]
    let ?p1 = p @ i # q1 let ?p2 = p @ j # q2
    from Cons(2) have ?p1 ∈ poss (ts ! k) by auto
    from Cons(1)[OF this] have rec: (?p2 ∈ poss (replace-at (ts ! k) ?p1 s1)) =
(?p2 ∈ poss (ts ! k)) .
```

from k **have** *min*: *min* (*length ts*) $k = k$ **by** *simp*
show *?case* **unfolding** t **using** *rec min k*
by (*auto simp*: *nth-append*)
next
case *Nil*
then obtain *f ts* **where** t: $t = Fun\ f\ ts$ **and** i: $i < length\ ts$ **by** (*cases t*, *auto*)
let *?p1* = i # *q1*
let *?s1* = *replace-at* (*ts* ! i) *q1 s1*
have *replace-at t ?p1 s1* = *Fun f* (*ts*[i := *?s1*]) **unfolding** *t upd-conv-take-nth-drop*[*OF i*] **by** *simp*
then show *?case* **unfolding** t **using** *ij* **by** *auto*
qed
qed

lemma *replace-at-subt-at*: $p \in poss\ t \Longrightarrow$ (*replace-at t p s*) |- $p = s$
by (*metis hole-pos-ctxt-of-pos-term subt-at-hole-pos*)

lemma *replace-at-below-poss*:
assumes p: $p' \in poss\ t$ **and** *le*: $p \leq_p p'$
shows $p \in poss$ (*replace-at t* p' s)
proof −
from *le* **obtain** p'' **where** p'': $p' = p$ @ p'' **unfolding** *less-eq-pos-def* **by** *auto*
from p **show** *?thesis* **unfolding** p''
by (*metis hole-pos-ctxt-of-pos-term hole-pos-poss poss-append-poss*)
qed

lemma *ctxt-of-pos-term-replace-at-below*:
assumes p: $p \in poss\ t$ **and** *le*: $p \leq_p p'$
shows *ctxt-of-pos-term p* (*replace-at t* p' u) = *ctxt-of-pos-term p t*
proof −
from *le* **obtain** p'' **where** p': $p' = p$ @ p'' **unfolding** *less-eq-pos-def* **by** *auto*
from p **show** *?thesis* **unfolding** p'
proof (*induct p arbitrary*: t)
case (*Cons i p*)
from *Cons*(*2*) **obtain** *f ts* **where** t: $t = Fun\ f\ ts$ **and** i: $i < length\ ts$ **and** p: $p \in poss$ (*ts* ! i)
by (*cases t*, *auto*)
from i **have** *min*: *min* (*length ts*) $i = i$ **by** *simp*
show *?case* **unfolding** t **using** *Cons*(*1*)[*OF p*] i **by** (*auto simp*: *nth-append min*)
next
case *Nil* **show** *?case* **by** *simp*
qed
qed

lemma *ctxt-of-pos-term-hole-pos*[*simp*]:
ctxt-of-pos-term (*hole-pos C*) ($C\langle t\rangle$) = C
by (*induct C*) *simp-all*

lemma *ctxt-poss-imp-ctxt-subst-poss*:
assumes p:$p' \in$ *poss* $C\langle t\rangle$ **shows** $p' \in$ *poss* $C\langle t \cdot \mu\rangle$
proof(*rule disjE*[*OF pos-cases*[*of* p' *hole-pos* C]])
assume $p' \leq_p$ *hole-pos* C
then show *?thesis* **using** *hole-pos-poss* **by** (*metis less-eq-pos-def poss-append-poss*)
next
assume *or*:*hole-pos* $C <_p p' \vee p' \perp$ *hole-pos* C
show *?thesis* **proof**(*rule disjE*[*OF or*])
assume *hole-pos* $C <_p p'$
then obtain q **where** *dec*:p' = *hole-pos* C @ q **unfolding** *less-pos-def less-eq-pos-def* **by** *auto*
with p **have** $q \in$ *poss* $(t \cdot \mu)$ **using** *hole-pos-poss-conv poss-imp-subst-poss* **by** *auto*
then show *?thesis* **using** *dec hole-pos-poss-conv* **by** *auto*
next
assume $p' \perp$ *hole-pos* C
then have *par*:*hole-pos* $C \perp p'$ **by** (*rule parallel-pos-sym*)
have *aux*:*hole-pos* $C \in$ *poss* $C\langle t \cdot \mu\rangle$ **using** *hole-pos-poss* **by** *auto*
from p **show** *?thesis* **using** *parallel-poss-replace-at*[*OF par aux*,*unfolded ctxt-of-pos-term-hole-pos*] **by** *fast*
qed
qed

lemma *var-pos-maximal*:
assumes *pt*:$p \in$ *poss* t **and** x:t |- p = *Var* x **and** $q \neq []$
shows p @ $q \notin$ *poss* t
proof−
from *assms* **have** $q \notin$ *poss* (*Var* x) **by** *force*
with *poss-append-poss*[*of* p q] *pt* x **show** *?thesis* **by** *simp*
qed

Positions in a context

definition *possc* :: $('f, 'v)$ *ctxt* $\Rightarrow$ *pos set* **where** *possc* $C \equiv \{p \mid p.\ \forall t.\ p \in$ *poss* $C\langle t\rangle\}$

lemma *poss-imp-possc*: $p \in$ *possc* $C \Longrightarrow p \in$ *poss* $C\langle t\rangle$ **unfolding** *possc-def* **by** *auto*

lemma *less-eq-hole-pos-in-possc*:
assumes *pleq*:$p \leq_p$ *hole-pos* C **shows** $p \in$ *possc* C
unfolding *possc-def*
using *replace-at-below-poss*[*OF hole-pos-poss pleq*, *unfolded hole-pos-id-ctxt*[*OF refl*]] **by** *simp*

lemma *hole-pos-in-possc*:*hole-pos* $C \in$ *possc* C
using *less-eq-hole-pos-in-possc order-refl* **by** *blast*

lemma *par-hole-pos-in-possc*:
assumes *par*:*hole-pos* $C \perp p$ **and** *ex*:$p \in$ *poss* $C\langle t\rangle$ **shows** $p \in$ *possc* C

using *parallel-poss-replace-at*[*OF par hole-pos-poss*, *unfolded hole-pos-id-ctxt*[*OF refl*], *of t*] *ex*
unfolding *possc-def* **by** *simp*

lemma *possc-not-below-hole-pos*:
assumes $p \in$ *possc* (C::($'a$,$'b$) *ctxt*) **shows** $\neg$ (*hole-pos* $C <_p p$)
proof(*rule notI*)
assume *hole-pos* $C <_p p$
then obtain r **where** p':p = *hole-pos* C @ r **and** r:$r \neq []$
unfolding *less-pos-def less-eq-pos-def* **by** *auto*
fix x::$'b$ **from** r **have** n:$r \notin$ *poss* (*Var* x) **using** *poss.simps*(*1*) **by** *auto*
from *assms* **have** $p \in$ (*poss* $C\langle$*Var* $x\rangle$) **unfolding** *possc-def* **by** *auto*
with *this*[*unfolded* p'] *hole-pos-poss-conv*[*of* C r] **have** $r \in$ *poss* (*Var* x) **by** *auto*
with n **show** *False* **by** *simp*
qed

lemma *possc-subst-not-possc-not-poss*:
assumes y:$p \in$ *possc* ($C \cdot_c \sigma$) **and** n:$p \notin$ *possc* C **shows** $p \notin$ *poss* $C\langle t\rangle$
proof−
from n **obtain** u **where** a:$p \notin$ *poss* $C\langle u\rangle$ **unfolding** *possc-def* **by** *auto*
from *possc-not-below-hole-pos*[*OF y*] **have** b:$\neg$ (*hole-pos* $C <_p p$)
unfolding *hole-pos-subst*[*of* C σ] **by** *auto*
from n a **have** c:$\neg$ ($p \leq_p$ *hole-pos* C) **unfolding** *less-pos-def* **using** *less-eq-hole-pos-in-possc* **by** *blast*
with *pos-cases b* **have** $p \perp$ *hole-pos* C **by** *blast*
with *par-hole-pos-in-possc*[*OF parallel-pos-sym*[*OF this*]] n **show** $p \notin$ *poss* ($C\langle t\rangle$) **by** *fast*
qed

All proper terms in a context

fun *ctxt-to-term-list* :: ($'f$, $'v$) *ctxt* $\Rightarrow$ ($'f$, $'v$) *term list*
where
ctxt-to-term-list Hole = [] |
ctxt-to-term-list (*More f bef C aft*) = *ctxt-to-term-list C* @ *bef* @ *aft*

lemma *ctxt-to-term-list-supt*: $t \in$ *set* (*ctxt-to-term-list* C) $\Longrightarrow$ $C\langle s\rangle \rhd t$
proof (*induct C*)
case (*More f bef C aft*)
from *More*(*2*) **have** *choice*: $t \in$ *set* (*ctxt-to-term-list* C) $\vee$ $t \in$ *set bef* $\vee$ $t \in$ *set aft* **by** *simp*
{
assume $t \in$ *set bef* $\vee$ $t \in$ *set aft*
then have $t \in$ *set* (*bef* @ $C\langle s\rangle$ # *aft*) **by** *auto*
then have *?case* **by** *auto*
}
moreover
{
assume t: $t \in$ *set* (*ctxt-to-term-list* C)
have (*More f bef C aft*)$\langle s\rangle \rhd C\langle s\rangle$ **by** *auto*

```
    moreover have C⟨s⟩ ▷ t
      by (rule More(1)[OF t])
    ultimately have ?case
      by (rule supt-trans)
  }
  ultimately show ?case using choice by auto
qed auto

lemma subteq-Var-imp-in-vars-term:
  r ⊵ Var x ⟹ x ∈ vars-term r
proof (induct r rule: term.induct)
  case (Var y)
  then have x = y by (cases rule: supteq.cases) auto
  then show ?case by simp
next
  case (Fun f ss)
  from ‹Fun f ss ⊵ Var x› have (Fun f ss = Var x) ∨ (Fun f ss ▷ Var x) by auto
  then show ?case
  proof
    assume Fun f ss = Var x then show ?thesis by auto
  next
    assume Fun f ss ▷ Var x
   then obtain s where s ∈ set ss and s ⊵ Var x using supt-Fun-imp-arg-supteq
by best
    with Fun have s ⊵ Var x ⟹ x ∈ vars-term s by best
    with ‹s ⊵ Var x› have x ∈ vars-term s by simp
    with ‹s ∈ set ss› show ?thesis by auto
  qed
qed

fun instance-term :: ('f, 'v) term ⇒ ('f set, 'v) term ⇒ bool
  where
    instance-term (Var x) (Var y) ⟷ x = y |
    instance-term (Fun f ts) (Fun fs ss) ⟷
    f ∈ fs ∧ length ts = length ss ∧ (∀ i<length ts. instance-term (ts ! i) (ss ! i)) |
    instance-term - - = False

fun subt-at-ctxt :: ('f, 'v) ctxt ⇒ pos ⇒ ('f, 'v) ctxt (infixl |'-c 67)
  where
    C |-c [] = C |
    More f bef C aft |-c (i#p) = C |-c p

lemma subt-at-subt-at-ctxt:
  assumes hole-pos C = p @ q
  shows C⟨t⟩ |- p = (C |-c p)⟨t⟩
  using assms
proof (induct p arbitrary: C)
  case (Cons i p)
  then obtain f bef D aft where C: C = More f bef D aft by (cases C, auto)
```

from *Cons*(*2*) **have** *hole-pos D* = *p* @ *q* **unfolding** *C* **by** *simp*
from *Cons*(*1*)[*OF this*] **have** *id*: (*D* |-*c p*) ⟨*t*⟩ = *D*⟨*t*⟩ |- *p* **by** *simp*
show *?case* **unfolding** *C subt-at-ctxt.simps id* **using** *Cons*(*2*) *C* **by** *auto*
qed *simp*

lemma *hole-pos-subt-at-ctxt*:
assumes *hole-pos C* = *p* @ *q*
shows *hole-pos* (*C* |-*c p*) = *q*
using *assms*
proof (*induct p arbitrary*: *C*)
case (*Cons i p*)
then obtain *f bef D aft* **where** *C*: *C* = *More f bef D aft* **by** (*cases C*, *auto*)
show *?case* **unfolding** *C subt-at-ctxt.simps*
by (*rule Cons*(*1*), *insert Cons*(*2*) *C*, *auto*)
qed *simp*

lemma *subt-at-ctxt-compose*[*simp*]: (*C* $\circ_c$ *D*) |-*c hole-pos C* = *D*
by (*induct C*, *auto*)

lemma *split-ctxt*:
assumes *hole-pos C* = *p* @ *q*
shows ∃ *D E*. *C* = *D* $\circ_c$ *E* ∧ *hole-pos D* = *p* ∧ *hole-pos E* = *q* ∧ *E* = *C* |-*c p*
using *assms*
proof (*induct p arbitrary*: *C*)
case *Nil*
show *?case*
by (*rule exI*[*of* - □], *rule exI*[*of* - *C*], *insert Nil*, *auto*)
next
case (*Cons i p*)
then obtain *f bef C′ aft* **where** *C*: *C* = *More f bef C′ aft* **by** (*cases C*, *auto*)
from *Cons*(*2*) **have** *hole-pos C′* = *p* @ *q* **unfolding** *C* **by** *simp*
from *Cons*(*1*)[*OF this*] **obtain** *D E* **where** *C′*: *C′* = *D* $\circ_c$ *E*
and *p*: *hole-pos D* = *p* **and** *q*: *hole-pos E* = *q* **and** *E*: *E* = *C′* |-*c p*
by *auto*
show *?case*
by (*rule exI*[*of* - *More f bef D aft*], *rule exI*[*of* - *E*], *unfold C C′*,
insert p[*symmetric*] *q E Cons*(*2*) *C*, *simp*)
qed

lemma *ctxt-subst-id*[*simp*]: *C* $\cdot_c$ *Var* = *C* **by** (*induct C*, *auto*)

the strict subterm relation between contexts and terms

inductive-set
suptc :: (('*f*, '*v*) *ctxt* × ('*f*, '*v*) *term*) *set*
where
arg: *s* ∈ *set bef* ∪ *set aft* ⟹ *s* ⊵ *t* ⟹ (*More f bef C aft*, *t*) ∈ *suptc*
| *ctxt*: (*C*,*s*) ∈ *suptc* ⟹ (*D* $\circ_c$ *C*, *s*) ∈ *suptc*

hide-const *suptcp*

abbreviation *suptc-pred* **where** *suptc-pred* $C\ t \equiv (C,\ t) \in$ *suptc*

notation (*xsymbols*)
  suptc-pred ((-/ $\triangleright c$ -) [*56*, *56*] *55*)

lemma *suptc-subst*: $C \triangleright c\ s \Longrightarrow C \cdot_c \sigma \triangleright c\ s \cdot \sigma$
proof (*induct rule*: *suptc.induct*)
  case (*arg s bef aft t f C*)
  let *?s* = λ *t*. $t \cdot \sigma$
  let *?m* = *map ?s*
  have *id*: *More f bef C aft* $\cdot_c \sigma$ = *More f* (*?m bef*) ($C \cdot_c \sigma$) (*?m aft*) **by** *simp*
  show *?case* **unfolding** *id*
    by (*rule suptc.arg*[*OF - supteq-subst*[*OF arg*(*2*)]],
      insert arg(*1*), *auto*)
next
  case (*ctxt C s D*)
  have *id*: $D \circ_c C \cdot_c \sigma = (D \cdot_c \sigma) \circ_c (C \cdot_c \sigma)$ **by** *simp*
  show *?case* **unfolding** *id*
    by (*rule suptc.ctxt*[*OF ctxt*(*2*)])
qed

lemma *suptc-imp-supt*: $C \triangleright c\ s \Longrightarrow C\langle t\rangle \triangleright s$
proof (*induct rule*: *suptc.induct*)
  case (*arg s bef aft u f C*)
  let *?C* = (*More f bef C aft*)
  from *arg*(*1*) **have** $s \in$ *set* (*args* (*?C*$\langle t\rangle$)) **by** *auto*
  then have *?C*$\langle t\rangle \triangleright s$ **by** *auto*
  from *supt-supteq-trans*[*OF this arg*(*2*)]
  show *?case* **.**
next
  case (*ctxt C s D*)
  have *supteq*: $(D \circ_c C)\langle t\rangle \trianglerighteq C\langle t\rangle$ **by** *auto*
  from *supteq-supt-trans*[*OF this ctxt*(*2*)]
  show *?case* **.**
qed

lemma *suptc-supteq-trans*: $C \triangleright c\ s \Longrightarrow s \trianglerighteq t \Longrightarrow C \triangleright c\ t$
proof (*induct rule*: *suptc.induct*)
  case (*arg s bef aft u f C*)
  show *?case*
    by (*rule suptc.arg*[*OF arg*(*1*) *supteq-trans*[*OF arg*(*2*) *arg*(*3*)]])
next
  case (*ctxt C s D*)
  then have *supt*: $C \triangleright c\ t$ **by** *auto*
  then show *?case* **by** (*rule suptc.ctxt*)
qed

lemma *supteq-suptc-trans*: $C = D \circ_c E \Longrightarrow E \triangleright c\ s \Longrightarrow C \triangleright c\ s$

```
  by (auto intro: suptc.ctxt)

hide-fact (open)
  suptcp.arg suptcp.cases suptcp.induct suptcp.intros suptc.arg suptc.ctxt

lemma supteq-ctxt-cases': C ⟨ t ⟩ ⊵ u ⟹
  C ⊳c u ∨ t ⊵ u ∨ (∃ D C'. C = D ∘c C' ∧ u = C' ⟨ t ⟩ ∧ C' ≠ □)
proof (induct C)
  case (More f bef C aft)
  let ?C = More f bef C aft
  let ?ba = bef @ C ⟨ t ⟩ # aft
  from More(2) have Fun f ?ba ⊵ u by simp
  then show ?case
  proof (cases rule: supteq.cases)
    case refl
    show ?thesis unfolding refl
      by (intro disjI2, rule exI[of - Hole], rule exI[of - ?C], auto)
  next
    case (subt v)
    show ?thesis
    proof (cases v ∈ set bef ∪ set aft)
      case True
      from suptc.arg[OF this subt(2)]
      show ?thesis by simp
    next
      case False
      with subt have C⟨ t ⟩ ⊵ u by simp
      from More(1)[OF this]
      show ?thesis
      proof (elim disjE exE conjE)
        assume C ⊳c u
        from suptc.ctxt[OF this, of More f bef □ aft] show ?thesis by simp
      next
        fix D C'
        assume ∗: C = D ∘c C' u = C'⟨t⟩ C' ≠ □
        show ?thesis
          by (intro disjI2 conjI, rule exI[of - More f bef D aft], rule exI[of - C'],
              insert ∗, auto)
      qed simp
    qed
  qed
qed simp

lemma supteq-ctxt-cases[consumes 1, case-names in-ctxt in-term sub-ctxt]: C ⟨ t
⟩ ⊵ u ⟹
  (C ⊳c u ⟹ P) ⟹
  (t ⊵ u ⟹ P) ⟹
  (⋀ D C'. C = D ∘c C' ⟹ u = C' ⟨ t ⟩ ⟹ C' ≠ □ ⟹ P) ⟹ P
  using supteq-ctxt-cases' by blast
```

definition *vars-subst* :: $('f, 'v)$ *subst* $\Rightarrow$ $'v$ *set*
where
vars-subst σ = *subst-domain* $\sigma \cup \bigcup$(*vars-term* ‘ *subst-range* σ)

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* $(\sigma\ y)$
and $x \in$ *vars-term* $(\tau\ v)$ **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

A substitution is idempotent iff the variables in its range are disjoint from its domain. See also "Term Rewriting and All That" 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

definition subst-compose' :: ('f, 'v) subst ⇒ ('f, 'v) subst ⇒ ('f, 'v) subst where
  subst-compose' σ τ = (λ x. if (x ∈ subst-domain σ) then σ x · τ else Var x)

lemma vars-subst-compose':
  assumes vars-subst τ ∩ subst-domain σ = {}
  shows σ ∘s τ = τ ∘s (subst-compose' σ τ) (is ?l = ?r)
proof
  fix x
  show ?l x = ?r x
  proof (cases x ∈ subst-domain σ)
    case True
    with assms have nmem: x ∉ vars-subst τ by auto
    then have nmem: x ∉ subst-domain τ unfolding vars-subst-def by auto
    then have id: τ x = Var x unfolding subst-domain-def by auto
    have ?l x = σ x · τ unfolding subst-compose-def by simp
    also have ... = ?r x unfolding subst-compose'-def subst-compose-def using
True unfolding id by simp
    finally show ?thesis .
  next
    case False
    then have l: ?l x = τ x · Var unfolding subst-domain-def subst-compose-def
by auto
    let ?στ = (λ x. if x ∈ subst-domain σ then σ x · τ else Var x)
    have r: ?r x = τ x · ?στ
      unfolding subst-compose'-def subst-compose-def ..
    show ?l x = ?r x unfolding l r
    proof (rule term-subst-eq)
      fix y
      assume y: y ∈ vars-term (τ x)
      have y ∈ vars-subst τ ∨ τ x = Var x
      proof (cases x ∈ subst-domain τ)
        case True then show ?thesis using y unfolding vars-subst-def by auto
      next
        case False
        then show ?thesis unfolding subst-domain-def by auto
      qed
      then show Var y = ?στ y
      proof
        assume y ∈ vars-subst τ
        with assms have y ∉ subst-domain σ by auto
        then show ?thesis by simp
      next
        assume τ x = Var x
        with y have y: y = x by simp
        show ?thesis unfolding y using False by auto
      qed
    qed
```

qed
qed

lemma *vars-subst-compose′-pow*:
assumes *vars-subst* $\tau \cap$ *subst-domain* $\sigma = \{\}$
shows $\sigma \hat{\hat{}} n \circ_s \tau = \tau \circ_s$ (*subst-compose′* $\sigma\ \tau$) $\hat{\hat{}}\ n$
proof (*induct n*)
case *0* **then show** *?case* **by** *auto*
next
case (*Suc n*)
let *?στ* = *subst-compose′* $\sigma\ \tau$
have $\sigma \hat{\hat{}}$ *Suc n* $\circ_s \tau = \sigma \circ_s (\sigma \hat{\hat{}} n \circ_s \tau)$ **by** (*simp add*: *ac-simps*)
also have ... $= \sigma \circ_s (\tau \circ_s$ *?στ*$\hat{\hat{}}n)$ **unfolding** *Suc* **..**
also have ... $= (\sigma \circ_s \tau) \circ_s$ *?στ* $\hat{\hat{}}\ n$ **by** (*auto simp*: *ac-simps*)
also have ... $= (\tau \circ_s$ *?στ*$) \circ_s$ *?στ* $\hat{\hat{}}\ n$ **unfolding** *vars-subst-compose′*[*OF assms*]
..
finally show *?case* **by** (*simp add*: *ac-simps*)
qed

lemma *subst-pow-commute*:
assumes $\sigma \circ_s \tau = \tau \circ_s \sigma$
shows $\sigma \circ_s (\tau \hat{\hat{}}\ n) = \tau \hat{\hat{}}\ n \circ_s \sigma$
proof (*induct n*)
case (*Suc n*)
have $\sigma \circ_s \tau \hat{\hat{}}$ *Suc n* $= (\sigma \circ_s \tau) \circ_s \tau \hat{\hat{}}\ n$ **by** (*simp add*: *ac-simps*)
also have ... $= \tau \circ_s (\sigma \circ_s \tau \hat{\hat{}}\ n)$ **unfolding** *assms* **by** (*simp add*: *ac-simps*)
also have ... $= \tau \hat{\hat{}}$ *Suc n* $\circ_s \sigma$ **unfolding** *Suc* **by** (*simp add*: *ac-simps*)
finally show *?case* **.**
qed *simp*

lemma *subst-power-commute*:
assumes $\sigma \circ_s \tau = \tau \circ_s \sigma$
shows $(\sigma \hat{\hat{}}\ n) \circ_s (\tau \hat{\hat{}}\ n) = (\sigma \circ_s \tau)\hat{\hat{}}n$
proof (*induct n*)
case (*Suc n*)
have $(\sigma \hat{\hat{}}$ *Suc n*$) \circ_s (\tau \hat{\hat{}}$ *Suc n*$) = (\sigma\hat{\hat{}}n \circ_s (\sigma \circ_s \tau \hat{\hat{}}\ n) \circ_s \tau)$
unfolding *subst-power-Suc* **by** (*simp add*: *ac-simps*)
also have ... $= (\sigma\hat{\hat{}}n \circ_s \tau \hat{\hat{}}\ n) \circ_s (\sigma \circ_s \tau)$
unfolding *subst-pow-commute*[*OF assms*] **by** (*simp add*: *ac-simps*)
also have ... $= (\sigma \circ_s \tau)\hat{\hat{}}$*Suc n* **unfolding** *Suc*
unfolding *subst-power-Suc* **..**
finally show *?case* **.**
qed *simp*

lemma *vars-term-ctxt-apply*:
vars-term $(C\langle t\rangle)$ = *vars-ctxt* $C \cup$ *vars-term* t
by (*induct C*) (*auto*)

lemma *vars-ctxt-pos-term*:

assumes $p \in$ *poss t*
shows *vars-term t* = *vars-ctxt* (*ctxt-of-pos-term p t*) ∪ *vars-term* (*t* |- *p*)
proof −
let *?C* = *ctxt-of-pos-term p t*
have *t* = *?C*⟨*t* |- *p*⟩ **using** *ctxt-supt-id* [*OF assms*] **by** *simp*
then have *vars-term t* = *vars-term* (*?C*⟨*t* |- *p*⟩) **by** *simp*
then show *?thesis* **unfolding** *vars-term-ctxt-apply* **.**
qed

lemma *vars-term-subt-at*:
assumes $p \in$ *poss t*
shows *vars-term* (*t* |- *p*) ⊆ *vars-term t*
using *vars-ctxt-pos-term* [*OF assms*] **by** *simp*

lemma *Var-pow-Var*[*simp*]: *Var* ^^ *n* = *Var*
by (*rule*, *induct n*, *auto*)

definition *is-inverse-renaming* :: ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *subst* **where**
is-inverse-renaming σ *y* = (
if Var y ∈ *subst-range* σ *then Var* (*the-inv-into* (*subst-domain* σ) σ (*Var y*))
else Var y)

lemma *is-renaming-inverse-domain*:
assumes *ren*: *is-renaming* σ
and *x*: *x* ∈ *subst-domain* σ
shows *Var x* · σ · *is-inverse-renaming* σ = *Var x* (**is** - · *?*σ = -)
proof −
note *ren* = *ren*[*unfolded is-renaming-def*]
from *ren* **obtain** *y* **where** σ*x*: σ *x* = *Var y* **by** *force*
from *ren* **have** *inj*: *inj-on* σ (*subst-domain* σ) **by** *auto*
note *inj* = *the-inv-into-f-eq*[*OF inj*, *OF* σ*x*]
note *d* = *is-inverse-renaming-def*
from *x* **have** *Var y* ∈ *subst-range* σ **using** σ*x* **by** *force*
then have *?*σ *y* = *Var* (*the-inv-into* (*subst-domain* σ) σ (*Var y*)) **unfolding** *d*
by *simp*
also have ... = *Var x* **using** *inj*[*OF x*] **by** *simp*
finally show *?thesis* **using** σ*x* **by** *simp*
qed

lemma *is-renaming-inverse-range*:
assumes *varren*: *is-renaming* σ
and *x*: *Var x* ∉ *subst-range* σ
shows *Var x* · σ · *is-inverse-renaming* σ = *Var x* (**is** - · *?*σ = -)
proof (*cases x* ∈ *subst-domain* σ)
case *True*
from *is-renaming-inverse-domain*[*OF varren True*]
show *?thesis* **.**
next
case *False*

then have σx: $\sigma\ x = Var\ x$ **unfolding** *subst-domain-def* **by** *auto*
note *ren* = *varren*[*unfolded is-renaming-def*]
note *d* = *is-inverse-renaming-def*
have $Var\ x \cdot \sigma \cdot ?\sigma = ?\sigma\ x$ **using** σx **by** *auto*
also have ... = $Var\ x$
unfolding *d* **using** *x* **by** *simp*
finally show *?thesis* **.**
qed

lemma *vars-subst-compose*:
vars-subst $(\sigma \circ_s \tau) \subseteq$ *vars-subst* $\sigma \cup$ *vars-subst* τ
proof
fix *x*
assume $x \in$ *vars-subst* $(\sigma \circ_s \tau)$
from *this*[*unfolded vars-subst-def subst-range.simps*]
obtain *y* **where** $y \in$ *subst-domain* $(\sigma \circ_s \tau) \wedge (x = y \vee x \in$ *vars-term* $((\sigma \circ_s \tau)\ y))$ **by** *blast*
with *subst-domain-compose*[*of* $\sigma\ \tau$] **have** *y*: $y \in$ *subst-domain* $\sigma \cup$ *subst-domain* τ **and** *disj*:
$x = y \vee (x \neq y \wedge x \in$ *vars-term* $(\sigma\ y \cdot \tau))$ **unfolding** *subst-compose-def* **by** *auto*
from *disj*
show $x \in$ *vars-subst* $\sigma \cup$ *vars-subst* τ
proof
assume $x = y$
with *y* **show** *?thesis* **unfolding** *vars-subst-def* **by** *auto*
next
assume $x \neq y \wedge x \in$ *vars-term* $(\sigma\ y \cdot \tau)$
then obtain *z* **where** *neq*: $x \neq y$ **and** *x*: $x \in$ *vars-term* $(\tau\ z)$ **and** *z*: $z \in$ *vars-term* $(\sigma\ y)$ **unfolding** *vars-term-subst* **by** *auto*
show *?thesis*
proof (*cases* $\tau\ z = Var\ z$)
case *False*
with *x* **have** $x \in$ *vars-subst* τ **unfolding** *vars-subst-def subst-domain-def subst-range.simps* **by** *blast*
then show *?thesis* **by** *auto*
next
case *True*
with *x* **have** *x*: $z = x$ **by** *auto*
with *z* **have** *y*: $x \in$ *vars-term* $(\sigma\ y)$ **by** *auto*
show *?thesis*
proof (*cases* $\sigma\ y = Var\ y$)
case *False*
with *y* **have** $x \in$ *vars-subst* σ **unfolding** *vars-subst-def subst-domain-def subst-range.simps* **by** *blast*
then show *?thesis* **by** *auto*
next
case *True*
with *y* **have** $x = y$ **by** *auto*

```
      with neq show ?thesis by auto
    qed
  qed
 qed
qed

lemma vars-subst-compose-update:
 assumes x: x ∉ vars-subst σ
 shows σ ∘s τ(x := t) = (σ ∘s τ)(x := t) (is ?l = ?r)
proof
 fix z
 note x = x[unfolded vars-subst-def subst-domain-def]
 from x have xx: σ x = Var x by auto
 show ?l z = ?r z
 proof (cases z = x)
  case True
  with xx show ?thesis by (simp add: subst-compose-def)
 next
  case False
  then have ?r z = σ z · τ unfolding subst-compose-def by auto
  also have ... = ?l z unfolding subst-compose-def
  proof (rule term-subst-eq)
   fix y
   assume y ∈ vars-term (σ z)
    with False x have v: y ≠ x unfolding subst-range.simps subst-domain-def
by force
   then show τ y = (τ(x := t)) y by simp
  qed
  finally show ?thesis by simp
 qed
qed

lemma subst-variants-imp-eq:
 assumes σ ∘s σ' = τ and τ ∘s τ' = σ
 shows ⋀x. σ x · σ' = τ x ⋀x. τ x · τ' = σ x
 using assms by (metis subst-compose-def)+

lemma poss-subst-choice: assumes p ∈ poss (t · σ) shows
 p ∈ poss t ∧ is-Fun (t |- p) ∨
 (∃ x q1 q2. q1 ∈ poss t ∧ q2 ∈ poss (σ x) ∧ t |- q1 = Var x ∧ x ∈ vars-term t
  ∧ p = q1 @ q2 ∧ t · σ |- p = σ x |- q2) (is - ∨ (∃ x q1 q2. ?p x q1 q2 t p))
 using assms
proof (induct p arbitrary: t)
 case (Cons i p t)
 show ?case
 proof (cases t)
  case (Var x)
  have ?p x [] (i # p) t (i # p) using Cons(2) unfolding Var by simp
  then show ?thesis by blast
```

```
  next
    case (Fun f ts)
    with Cons(2) have i: i < length ts and p: p ∈ poss (ts ! i · σ) by auto
    from Cons(1)[OF p]
    show ?thesis
    proof
      assume ∃ x q1 q2. ?p x q1 q2 (ts ! i) p
      then obtain x q1 q2 where ?p x q1 q2 (ts ! i) p by auto
      with Fun i have ?p x (i # q1) q2 (Fun f ts) (i # p) by auto
      then show ?thesis unfolding Fun by blast
    next
      assume p ∈ poss (ts ! i) ∧ is-Fun (ts ! i |- p)
      then show ?thesis using Fun i by auto
    qed
  qed
next
  case Nil
  show ?case
  proof (cases t)
    case (Var x)
    have ?p x [] [] t [] unfolding Var by auto
    then show ?thesis by auto
  qed simp
qed

fun vars-term-list :: ('f, 'v) term ⇒ 'v list
  where
    vars-term-list (Var x) = [x] |
    vars-term-list (Fun - ts) = concat (map vars-term-list ts)

lemma set-vars-term-list [simp]:
  set (vars-term-list t) = vars-term t
  by (induct t) simp-all

lemma unary-vars-term-list:
  assumes t: funas-term t ⊆ F
    and unary: ⋀ f n. (f, n) ∈ F ⟹ n ≤ 1
  shows vars-term-list t = [] ∨ (∃ x ∈ vars-term t. vars-term-list t = [x])
proof −
  from t show ?thesis
  proof (induct t)
    case Var then show ?case by auto
  next
    case (Fun f ss)
    show ?case
    proof (cases ss)
      case Nil
      then show ?thesis by auto
    next
```

```
      case (Cons t ts)
      let ?n = length ss
      from Fun(2) have (f,?n) ∈ F by auto
      from unary[OF this] have n: ?n ≤ Suc 0 by auto
      with Cons have ?n = Suc 0 by simp
      with Cons have ss: ss = [t] by (cases ts, auto)
      note IH = Fun(1)[unfolded ss, simplified]
      from ss have id1: vars-term-list (Fun f ss) = vars-term-list t by simp
      from ss have id2: vars-term (Fun f ss) = vars-term t by simp
      from Fun(2) ss have mem: funas-term t ⊆ F by auto
      show ?thesis unfolding id1 id2 using IH[OF refl mem] by simp
    qed
  qed
qed
```

declare *vars-term-list.simps* [*simp del*]

The list of function symbols in a term (without removing duplicates).

```
fun funs-term-list :: ('f, 'v) term ⇒ 'f list
  where
    funs-term-list (Var -) = [] |
    funs-term-list (Fun f ts) = f # concat (map funs-term-list ts)
```

```
lemma set-funs-term-list [simp]:
  set (funs-term-list t) = funs-term t
  by (induct t) simp-all
```

declare *funs-term-list.simps* [*simp del*]

The list of function symbol and arity pairs in a term (without removing duplicates).

```
fun funas-term-list :: ('f, 'v) term ⇒ ('f × nat) list
  where
    funas-term-list (Var -) = [] |
    funas-term-list (Fun f ts) = (f, length ts) # concat (map funas-term-list ts)
```

```
lemma set-funas-term-list [simp]:
  set (funas-term-list t) = funas-term t
  by (induct t) simp-all
```

declare *funas-term-list.simps* [*simp del*]

```
definition funas-args-term-list :: ('f, 'v) term ⇒ ('f × nat) list
  where
    funas-args-term-list t = concat (map funas-term-list (args t))
```

```
lemma set-funas-args-term-list [simp]:
  set (funas-args-term-list t) = funas-args-term t
  by (simp add: funas-args-term-def funas-args-term-list-def)
```

lemma *vars-term-list-map-funs-term*:
vars-term-list (*map-funs-term fg t*) = *vars-term-list t*
proof (*induct t*)
case (*Var x*) **then show** *?case* **by** (*simp add*: *vars-term-list.simps*)
next
case (*Fun f ss*)
show *?case* **by** (*simp add*: *vars-term-list.simps o-def*, *insert Fun*, *induct ss*, *auto*)
qed

lemma *funs-term-list-map-funs-term*:
funs-term-list (*map-funs-term fg t*) = *map fg* (*funs-term-list t*)
proof (*induct t*)
case (*Var x*) **show** *?case* **by** (*simp add*: *funs-term-list.simps*)
next
case (*Fun f ts*)
show *?case*
by (*simp add*: *funs-term-list.simps*, *insert Fun*, *induct ts*, *auto*)
qed

Next we provide some functions to compute multisets instead of sets of function symbols, variables, etc. they may be helpful for non-duplicating TRSs.

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

fun *funs-ctxt-ms* :: ('*f*,'*v*)*ctxt* ⇒ '*f multiset*
where
funs-ctxt-ms Hole = {#} |
funs-ctxt-ms (*More f bef C aft*) =
{#*f*#} + $\sum_{\#}$ (*mset* (*map funs-term-ms bef*)) +
funs-ctxt-ms C + $\sum_{\#}$ (*mset* (*map funs-term-ms aft*))

lemma *funs-term-ms-ctxt-apply*:
funs-term-ms $C\langle t\rangle$ = *funs-ctxt-ms C* + *funs-term-ms t*
by (*induct C*) (*auto simp*: *multiset-eq-iff*)

lemma *funs-term-ms-subst-apply*:
funs-term-ms ($t \cdot \sigma$) =
funs-term-ms t + $\sum_{\#}$ (*image-mset* (λ *x*. *funs-term-ms* (σ *x*)) (*vars-term-ms t*))
proof (*induct t*)
case (*Fun f ts*)
let *?m* = *mset*
let *?f* = *funs-term-ms*
let *?ts* = $\sum_{\#}$ (*?m* (*map ?f ts*))
let *?σ* = $\sum_{\#}$ (*image-mset* (λx. *?f* (σ *x*)) ($\sum_{\#}$ (*?m* (*map vars-term-ms ts*))))

let *?tsσ* = $\sum_{\#}$ (*?m* (*map* (λ *x*. *?f* (*x* · *σ*)) *ts*))
have *ind*: *?tsσ* = *?ts* + *?σ* **using** *Fun*
by (*induct ts*, *auto simp*: *multiset-eq-iff*)
then show *?case* **unfolding** *multiset-eq-iff* **by** (*simp add*: *o-def*)
qed *auto*

lemma *ground-vars-term-ms-empty*:
ground t = (*vars-term-ms t* = {#})
unfolding *ground-vars-term-empty*
unfolding *set-mset-vars-term-ms* [*symmetric*]
by (*simp del*: *set-mset-vars-term-ms*)

lemma *vars-term-ms-map-funs-term* [*simp*]:
vars-term-ms (*map-funs-term fg t*) = *vars-term-ms t*
proof (*induct t*)
case (*Fun f ts*)
then show *?case* **by** (*induct ts*) *auto*
qed *simp*

lemma *funs-term-ms-map-funs-term*:
funs-term-ms (*map-funs-term fg t*) = *image-mset fg* (*funs-term-ms t*)
proof (*induct t*)
case (*Fun f ss*)
then show *?case* **by** (*induct ss*, *auto*)
qed *auto*

lemma *supteq-imp-vars-term-ms-subset*:
s ⊵ *t* ⟹ *vars-term-ms t* ⊆# *vars-term-ms s*
proof (*induct rule*: *supteq.induct*)
case (*subt u ss t f*)
from *subt*(*1*) **obtain** *bef aft* **where** *ss*: *ss* = *bef* @ *u* # *aft*
by (*metis in-set-conv-decomp*)
have *vars-term-ms t* ⊆# *vars-term-ms u* **by** *fact*
also have ... ⊆# $\sum_{\#}$ (*mset* (*map vars-term-ms ss*))
unfolding *ss* **by** (*simp add*: *ac-simps*)
also have ... = *vars-term-ms* (*Fun f ss*) **by** *auto*
finally show *?case* **by** *auto*
qed *auto*

lemma *mset-funs-term-list*:
mset (*funs-term-list t*) = *funs-term-ms t*
proof (*induct t*)
case (*Var x*) **show** *?case* **by** (*simp add*: *funs-term-list.simps*)
next
case (*Fun f ts*)
show *?case*
by (*simp add*: *funs-term-list.simps*, *insert Fun*, *induct ts*, *auto simp*: *funs-term-list.simps multiset-eq-iff*)
qed

Creating substitutions from lists

type-synonym $('f, 'v, 'w)$ *gsubstL* $= ('v \times ('f, 'w)$ *term*$)$ *list*
type-synonym $('f, 'v)$ *substL* $= ('f, 'v, 'v)$ *gsubstL*

definition *mk-subst* :: $('v \Rightarrow ('f, 'w)$ *term*$) \Rightarrow ('f, 'v, 'w)$ *gsubstL* $\Rightarrow ('f, 'v, 'w)$ *gsubst* **where**
  mk-subst d xts $\equiv$
    $(\lambda x.$ *case map-of xts x of*
      None $\Rightarrow$ *d x*
    | *Some t* $\Rightarrow$ *t*$)$

lemma *mk-subst-not-mem*:
  assumes *x*: $x \notin$ *set xs*
  shows *mk-subst f* (*zip xs ts*) *x* = *f x*
proof −
  have *map-of* (*zip xs ts*) *x* = *None*
    unfolding *map-of-eq-None-iff set-zip* **using** *x*[*unfolded set-conv-nth*] **by** *auto*
  then show *?thesis* **unfolding** *mk-subst-def* **by** *auto*
qed

lemma *mk-subst-not-mem′*:
  assumes *x*: $x \notin$ *set* (*map fst ss*)
  shows *mk-subst f ss x* = *f x*
proof −
  have *map-of ss x* = *None*
    unfolding *map-of-eq-None-iff* **using** *x* **by** *auto*
  then show *?thesis* **unfolding** *mk-subst-def* **by** *auto*
qed

lemma *mk-subst-distinct*:
  assumes *dist*: *distinct xs*
    and *i*: *i* < *length xs i* < *length ls*
  shows *mk-subst f* (*zip xs ls*) (*xs* ! *i*) = *ls* ! *i*
proof −
  from *i* **have** *in-zip*:(*xs*!*i*, *ls*!*i*) $\in$ *set* (*zip xs ls*)
    using *nth-zip*[*OF i*] *set-zip* **by** *auto*
  from *dist* **have** *dist′*:*distinct* (*map fst* (*zip xs ls*))
    by (*simp add*: *map-fst-zip-take*)
  then show *?thesis*
    unfolding *mk-subst-def* **using** *map-of-is-SomeI*[*OF dist′ in-zip*] **by** *simp*
qed

lemma *mk-subst-Nil* [*simp*]:
  mk-subst d [] = *d*
  by (*simp add*: *mk-subst-def*)

lemma *mk-subst-concat*:
  assumes $x \notin$ *set* (*map fst xs*)
  shows (*mk-subst f* (*xs*@*ys*)) *x* = (*mk-subst f ys*) *x*

using *assms* **unfolding** *mk-subst-def map-of-append*
by (*simp add*: *dom-map-of-conv-image-fst map-add-dom-app-simps*(*3*))

lemma *mk-subst-concat-Cons*:
assumes $x \in$ *set* (*map fst ss*)
shows *mk-subst f* (*concat* (*ss*#*ss'*)) *x* = *mk-subst f ss x*
proof−
from *assms* **obtain** *y* **where** *map-of ss x* = *Some y*
by (*metis list.set-map map-of-eq-None-iff not-None-eq*)
then show *?thesis* **unfolding** *mk-subst-def concat.simps map-of-append*
by *simp*
qed

lemma *vars-term-var-poss-iff*:
$x \in$ *vars-term t* $\longleftrightarrow$ ($\exists$ *p*. *p* $\in$ *var-poss t* $\wedge$ *Var x* = *t* |- *p*) (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *x*: *?L*
obtain *p* **where** *p* $\in$ *poss t* **and** *Var x* = *t* |- *p*
using *supteq-imp-subt-at* [*OF supteq-Var* [*OF x*]] **by** *force*
then show *?R* **using** *var-poss-iff* **by** *auto*
next
assume *p*: *?R*
then obtain *p* **where** *1*: *p* $\in$ *var-poss t* **and** *2*: *Var x* = *t* |- *p* **by** *auto*
from *var-poss-imp-poss* [*OF 1*] **have** *p* $\in$ *poss t* **.**
then show *?L* **by** (*simp add*: *2 subt-at-imp-supteq subteq-Var-imp-in-vars-term*)
qed

lemma *vars-term-poss-subt-at*:
assumes *x* $\in$ *vars-term t*
obtains *q* **where** *q* $\in$ *poss t* **and** *t* |- *q* = *Var x*
using *assms*
proof (*induct t*)
case (*Fun f ts*)
then obtain *t*:*t* $\in$ *set ts* **and** *x*:*x* $\in$ *vars-term t* **by** *auto*
moreover then obtain *i* **where** *t* = *ts* ! *i* **and** *i* < *length ts* **using** *in-set-conv-nth*
by *metis*
ultimately show *?case* **using** *Fun*(*1*)[*OF t - x*] *Fun*(*2*)[*of Cons i q* **for** *q*] **by**
auto
qed *auto*

lemma *vars-ctxt-subt-at'*:
assumes *x* $\in$ *vars-ctxt C*
and *p* $\in$ *poss C*$\langle t \rangle$
and *hole-pos C* = *p*
shows $\exists$ *q*. *q* $\in$ *poss C*$\langle t \rangle$ $\wedge$ *parallel-pos p q* $\wedge$ *C*$\langle t \rangle$ |- *q* = *Var x*
using *assms*
proof (*induct C arbitrary*: *p*)
case (*More f bef C aft*)
then have [*simp*]: *p* = *length bef* # *hole-pos C* **by** *auto*

```
  consider
    (C) x ∈ vars-ctxt C |
    (bef) t where t ∈ set bef and x ∈ vars-term t |
    (aft) t where t ∈ set aft and x ∈ vars-term t
    using More by auto
  then show ?case
  proof (cases)
    case C
     from More(1)[OF this] obtain q where q ∈ poss C⟨t⟩ ∧ hole-pos C ⊥ q ∧
C⟨t⟩ |- q = Var x
      by fastforce
    then show ?thesis by (force intro!: exI[of - length bef # q])
  next
    case bef
    then obtain q where q ∈ poss t and t |- q = Var x
      using vars-term-poss-subt-at by force
    moreover from bef obtain i where i < length bef and bef ! i = t
      using in-set-conv-nth by metis
    ultimately show ?thesis
      by (force simp: nth-append intro!: exI[of - i # q])
  next
    case aft
    then obtain q where q ∈ poss t and t |- q = Var x
      using vars-term-poss-subt-at by force
    moreover from aft obtain i where i < length aft and aft ! i = t
      using in-set-conv-nth by metis
    ultimately show ?thesis
      by (force simp: nth-append intro!: exI[of - (Suc (length bef) + i) # q])
  qed
qed auto

lemma vars-ctxt-subt-at:
  assumes x ∈ vars-ctxt C
    and p ∈ poss C⟨t⟩
    and hole-pos C = p
  obtains q where q ∈ poss C⟨t⟩ and parallel-pos p q and C⟨t⟩ |- q = Var x
  using vars-ctxt-subt-at' assms by force

lemma poss-is-Fun-fun-poss:
  assumes p ∈ poss t
    and is-Fun (t |- p)
  shows p ∈ fun-poss t
  using assms by (metis DiffI is-Var-def poss-simps(3) var-poss-iff)

lemma fun-poss-map-vars-term:
  fun-poss (map-vars-term f t) = fun-poss t
  unfolding map-vars-term-eq proof(induct t)
  case (Fun g ts)
  {fix i assume i < length ts
```

with *Fun* **have** *fun-poss* (*map* ($\lambda t.\ t \cdot (Var \circ f)$) *ts* ! *i*) = *fun-poss* (*ts*!*i*)
by *fastforce*
then have {*i* # *p* |*p*. *p* ∈ *fun-poss* (*map* ($\lambda t.\ t \cdot (Var \circ f)$) *ts* ! *i*)} = {*i* # *p* |*p*. *p* ∈ *fun-poss* (*ts* ! *i*)}
by *presburger*
}
then show *?case* **unfolding** *fun-poss.simps eval-term.simps length-map*
by *auto*
qed *simp*

lemma *fun-poss-append-poss*:
assumes *p*@*q* ∈ *poss t q* ≠ []
shows *p* ∈ *fun-poss t*
by (*meson assms is-Var-def poss-append-poss poss-is-Fun-fun-poss var-pos-maximal*)

lemma *fun-poss-append-poss′*:
assumes *p*@*q* ∈ *fun-poss t*
shows *p* ∈ *fun-poss t*
by (*metis append.right-neutral assms fun-poss-append-poss fun-poss-imp-poss*)

lemma *fun-poss-in-ctxt*:
assumes *q*@*p* ∈ *fun-poss* (*C*⟨*t*⟩)
and *hole-pos C* = *q*
shows *p* ∈ *fun-poss t*
by (*metis Term.term.simps*(*4*) *assms fun-poss-fun-conv fun-poss-imp-poss hole-pos-poss hole-pos-poss-conv is-VarE poss-is-Fun-fun-poss subt-at-append subt-at-hole-pos*)

lemma *args-poss*:
assumes *i* # *p* ∈ *poss t*
obtains *f ts* **where** *t* = *Fun f ts p* ∈ *poss* (*ts*!*i*) *i* < *length ts*
by (*metis Cons-poss-Var assms poss.elims poss-Cons-poss term.sel*(*4*))

lemma *var-poss-parallel*:
assumes *p* ∈ *var-poss t* **and** *q* ∈ *var-poss t* **and** *p* ≠ *q*
shows *p* ⊥ *q*
using *assms* **proof**(*induct t arbitrary*:*p q*)
case (*Fun f ts*)
from *Fun*(*2*) **obtain** *i p′* **where** *i*:*i* < *length ts p′* ∈ *var-poss* (*ts*!*i*) **and** *p*:*p* = *i*#*p′*
using *var-poss-iff* **by** *fastforce*
with *Fun*(*3*) **obtain** *j q′* **where** *j*:*j* < *length ts q′* ∈ *var-poss* (*ts*!*j*) **and** *q*:*q* = *j*#*q′*
using *var-poss-iff* **by** *fastforce*
then show *?case* **proof**(*cases i* = *j*)
case *True*
from *Fun*(*4*) **have** *p′* ≠ *q′*
unfolding *p q True* **by** *simp*
with *Fun*(*1*) *i j* **show** *?thesis*
unfolding *True p q parallel-pos.simps* **using** *nth-mem* **by** *blast*

next
case *False*
then show *?thesis* **unfolding** *p q*
by *simp*
qed
qed *simp*

lemma *ctxt-comp-equals*:
assumes *poss*:$p \in$ *poss s p* $\in$ *poss t*
and *ctxt-of-pos-term p s* $\circ_c$ *C* = *ctxt-of-pos-term p t* $\circ_c$ *D*
shows *C* = *D*
using *assms* **proof**(*induct p arbitrary*:*s t*)
case (*Cons i p*)
from *Cons*(*2*) **obtain** *f ss* **where** *s*:*s* = *Fun f ss* **and** *p*:$p \in$ *poss* (*ss*!*i*)
using *args-poss* **by** *blast*
from *Cons*(*3*) **obtain** *g ts* **where** *t*:*t* = *Fun g ts* **and** *p'*:$p \in$ *poss* (*ts*!*i*)
using *args-poss* **by** *blast*
from *Cons*(*1*)[*OF p p'*] *Cons*(*4*) **show** *?case*
unfolding *s t ctxt-of-pos-term.simps ctxt-compose.simps* **by** *simp*
qed *simp*

lemma *ctxt-subst-comp-pos*:
assumes $q \in$ *poss t* **and** $p \in$ *poss* ($t \cdot \tau$)
and (*ctxt-of-pos-term q t* $\cdot_c$ σ) $\circ_c$ *C* = *ctxt-of-pos-term p* ($t \cdot \tau$)
shows $q \leq_p p$
using *assms* **by** (*metis hole-pos-ctxt-compose hole-pos-ctxt-of-pos-term hole-pos-subst less-eq-pos-simps*(*1*))

Predicate whether a context is ground, i.e., whether the context contains no variables.

fun *ground-ctxt* :: ('*f*,'*v*)*ctxt* $\Rightarrow$ *bool* **where**
ground-ctxt Hole = *True*
| *ground-ctxt* (*More f ss1 C ss2*) =
(($\forall$ *s*$\in$*set ss1*. *ground s*) $\wedge$ ($\forall$ *s*$\in$*set ss2*. *ground s*) $\wedge$ *ground-ctxt C*)

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

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

Linearity of a term

fun *linear-term* :: ('*f*, '*v*) *term* $\Rightarrow$ *bool*
where
linear-term (*Var* -) = *True* |
linear-term (*Fun* - *ts*) = (*is-partition* (*map vars-term ts*) $\wedge$ ($\forall$ *t*$\in$*set ts*. *linear-term t*))

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

Matching for linear terms

fun *weak-match* :: $('f,\ 'v)\ term \Rightarrow ('f,\ 'v)\ term \Rightarrow bool$
  where
    weak-match - (Var -) $\longleftrightarrow$ *True* |
    weak-match (Var -) (Fun - -) $\longleftrightarrow$ *False* |
    weak-match (Fun f ts) (Fun g ss) $\longleftrightarrow$
    $f = g \wedge length\ ts = length\ ss \wedge (\forall i < length\ ts.\ weak\text{-}match\ (ts\ !\ i)\ (ss\ !\ i))$

lemma *weak-match-refl*: *weak-match t t*
  by (*induct t*) *auto*

lemma *weak-match-match*: *weak-match* $(t \cdot \sigma)\ t$
  by (*induct t*) *auto*

lemma *weak-match-map-funs-term*:
  weak-match t s $\Longrightarrow$ *weak-match* (*map-funs-term g t*) (*map-funs-term g s*)
proof (*induct s arbitrary*: *t*)
  case (*Fun f ss t*)
  from *Fun*(*2*) **obtain** *ts* **where** *t*: *t* = *Fun f ts* **by** (*cases t*, *auto*)
  from *Fun*(*1*)[*unfolded set-conv-nth*] *Fun*(*2*)[*unfolded t*]
  show *?case* **unfolding** *t* **by** *force*
qed *simp*

lemma *linear-weak-match*:
  assumes *linear-term l* **and** *weak-match t s* **and** $s = l \cdot \sigma$
  shows $\exists \tau.\ t = l{\cdot}\tau \wedge (\forall x{\in}vars\text{-}term\ l.\ weak\text{-}match\ (Var\ x \cdot \tau)\ (Var\ x \cdot \sigma))$
  using *assms* **proof** (*induct l arbitrary*: *s t*)
  case (*Var x*)
  show *?case*
  proof (*rule exI*[*of -* (λ *y. t*)], *intro conjI*, *simp*)
    from *Var* **show** $\forall x{\in}vars\text{-}term\ (Var\ x).\ weak\text{-}match\ (Var\ x \cdot (\lambda y.\ t))\ (Var\ x \cdot \sigma)$
      by *force*
  qed
next
  case (*Fun f ls*)
  let *?n* = *length ls*
  from *Fun*(*4*) **obtain** *ss* **where** *s*: *s* = *Fun f ss* **and** *lss*: *length ss* = *?n* **by** (*cases*

s, *auto*)
with *Fun*(*4*) **have** *match*: $\bigwedge$ *i*. *i* < *?n* $\Longrightarrow$ *ss* ! *i* = (*ls* ! *i*) · σ **by** *auto*
from *Fun*(*3*) *s lss* **obtain** *ts* **where** *t*: *t* = *Fun f ts*
and *lts*: *length ts* = *?n* **by** (*cases t*, *auto*)
with *Fun*(*3*) *s* **have** *weak-match*: $\bigwedge$ *i*. *i* < *?n* $\Longrightarrow$ *weak-match* (*ts* ! *i*) (*ss* ! *i*) **by** *auto*
from *Fun*(*2*) **have** *linear*: $\bigwedge$ *i*. *i* < *?n* $\Longrightarrow$ *linear-term* (*ls* ! *i*) **by** *simp*
let *?cond* = λ τ *i*. *ts* ! *i* = *ls* ! *i* · τ $\wedge$ ($\forall$ *x*$\in$*vars-term* (*ls* ! *i*). *weak-match* (*Var x* · τ) (*Var x* · σ))
{
fix *i*
assume *i*: *i* < *?n*
then have *ls* ! *i* $\in$ *set ls* **by** *simp*
from *Fun*(*1*)[*OF this linear*[*OF i*] *weak-match*[*OF i*] *match*[*OF i*]]
have $\exists$ τ. *?cond* τ *i* **.**
}
then have $\forall$ *i*. $\exists$ τ. (*i* < *?n* $\longrightarrow$ *?cond* τ *i*) **by** *auto*
from *choice*[*OF this*] **obtain** *subs* **where** *subs*: $\bigwedge$ *i*. *i* < *?n* $\Longrightarrow$ *?cond* (*subs i*) *i* **by** *auto*
from *Fun*(*2*) **have** *distinct*: *is-partition*(*map vars-term ls*) **by** *simp*
from *subst-merge*[*OF this*, *of subs*]
obtain τ **where** τ: $\bigwedge$ *i x* . *i* < *length ls* $\Longrightarrow$ *x* $\in$ *vars-term* (*ls* ! *i*) $\Longrightarrow$ τ *x* = *subs i x* **by** *auto*
show *?case*
proof (*rule exI*[*of* - τ], *simp add*: *t*, *intro ballI conjI*)
fix *li x*
assume *li*: *li* $\in$ *set ls* **and** *x*: *x* $\in$ *vars-term li*
from *li* **obtain** *i* **where** *i*: *i* < *?n* **and** *li*: *li* = *ls* ! *i* **unfolding** *set-conv-nth* **by** *auto*
with *x* **have** *x*: *x* $\in$ *vars-term* (*ls* ! *i*) **by** *simp*
from *subs*[*OF i*, *THEN conjunct2*, *THEN bspec*, *OF x*] **show** *weak-match* (τ *x*) (σ *x*) **unfolding** τ[*OF i x*[*unfolded li*]]
by *simp*
next
show *ts* = *map* (λ *t*. *t* · τ) *ls*
proof (*rule nth-equalityI*, *simp add*: *lts*)
fix *i*
assume *i* < *length ts*
with *lts* **have** *i*: *i* < *?n* **by** *simp*
have *ts* ! *i* = *ls* ! *i* · *subs i*
by (*rule subs*[*THEN conjunct1*, *OF i*])
also have ... = *ls* ! *i* · τ **unfolding** *term-subst-eq-conv* **using** τ[*OF i*] **by** *auto*
finally show *ts* ! *i* = *map* (λ *t*. *t* · τ) *ls* ! *i*
by (*simp add*: *nth-map*[*OF i*])
qed
qed
qed

lemma *map-funs-subst-split*:

assumes *map-funs-term fg t* = *s* · *σ*
and *linear-term s*
shows ∃ *u τ*. *t* = *u* · *τ* ∧ *map-funs-term fg u* = *s* ∧ (∀ *x*∈*vars-term s*. *map-funs-term fg* (*τ x*) = (*σ x*))
using *assms*
proof (*induct s arbitrary*: *t*)
case (*Var x t*)
show *?case*
proof (*intro exI conjI*)
show *t* = *Var x* · (λ -. *t*) **by** *simp*
qed (*insert Var*, *auto*)
next
case (*Fun g ss t*)
from *Fun*(*2*) **obtain** *f ts* **where** *t*: *t* = *Fun f ts* **by** (*cases t*, *auto*)
note *Fun* = *Fun*[*unfolded t*, *simplified*]
let *?n* = *length ss*
from *Fun* **have** *rec*: *map* (*map-funs-term fg*) *ts* = *map* (λ *t*. *t* · *σ*) *ss*
and *g*: *fg f* = *g*
and *lin*: ⋀ *s*. *s* ∈ *set ss* ⟹ *linear-term s*
and *part*: *is-partition* (*map vars-term ss*) **by** *auto*
from *arg-cong*[*OF rec*, *of length*] **have** *len*: *length ts* = *?n* **by** *simp*
from *map-nth-conv*[*OF rec*] **have** *rec*: ⋀ *i*. *i* < *?n* ⟹ *map-funs-term fg* (*ts* ! *i*) = *ss* ! *i* · *σ* **unfolding** *len* **by** *auto*
let *?p* = λ *i u τ*. *ts* ! *i* = *u* · *τ* ∧ *map-funs-term fg u* = *ss* ! *i* ∧ (∀ *x*∈*vars-term* (*ss* ! *i*). *map-funs-term fg* (*τ x*) = *σ x*)
{
fix *i*
assume *i*: *i* < *?n*
then have *ss* ! *i* ∈ *set ss* **by** *auto*
from *Fun*(*1*)[*OF this rec*[*OF i*] *lin*[*OF this*]]
have ∃ *u τ*. *?p i u τ* **.**
}
then have ∀ *i*. ∃ *u τ*. *i* < *?n* ⟶ *?p i u τ* **by** *blast*
from *choice*[*OF this*] **obtain** *us* **where** ∀ *i*. ∃ *τ*. *i* < *?n* ⟶ *?p i* (*us i*) *τ* **..**
from *choice*[*OF this*] **obtain** *τs* **where** *p*: ⋀ *i*. *i* < *?n* ⟹ *?p i* (*us i*) (*τs i*) **by** *blast*
from *subst-merge*[*OF part*, *of τs*] **obtain** *τ* **where** *τ*: ⋀ *i x*. *i* < *?n* ⟹ *x* ∈ *vars-term* (*ss* ! *i*) ⟹ *τ x* = *τs i x* **by** *blast*
{
fix *i*
assume *i*: *i* < *?n*
from *p*[*OF i*] **have** *map-funs-term fg* (*us i*) = *ss* ! *i* **by** *auto*
from *arg-cong*[*OF this*, *of vars-term*] *vars-term-map-funs-term*[*of fg*]
have *vars-term* (*us i*) = *vars-term* (*ss* ! *i*) **by** *auto*
} note *vars* = *this*
let *?us* = *map us* [*0* ..< *?n*]
show *?case*
proof (*intro exI conjI ballI*)
have *ss*: *ts* = *map* (λ *t*. *t* · *τ*) *?us*

```
      unfolding list-eq-iff-nth-eq
      unfolding len
    proof (intro conjI allI impI)
      fix i
      assume i: i < ?n
      have us: ?us ! i = us i using nth-map-upt[of i ?n 0] i by auto
      have (map (λ t. t · τ) ?us) ! i = us i · τ
        unfolding us[symmetric]
        using nth-map[of i ?us λ t. t · τ]  i by force
      also have ... = us i · τs i
        by (rule term-subst-eq, rule τ[OF i], insert vars[OF i], auto )
      also have ... = ts ! i using p[OF i] by simp
      finally
      show ts ! i = map (λ t. t · τ) ?us ! i ..
    qed auto
    show t = Fun f ?us  · τ
      unfolding t
      unfolding ss by auto
  next
    show map-funs-term fg (Fun f ?us) = Fun g ss
      using p g by (auto simp: list-eq-iff-nth-eq[of - ss])
  next
    fix x
    assume x: x ∈ vars-term (Fun g ss)
    then obtain s where s: s ∈ set ss and x: x ∈ vars-term s by auto
    from s[unfolded set-conv-nth] obtain i where s: s = ss ! i and i: i < ?n by
auto
    note x = x[unfolded s]
    from p[OF i] vars[OF i] x τ[OF i x]
    show map-funs-term fg (τ x) = σ x by auto
  qed
qed

lemma linear-map-funs-term [simp]:
  linear-term (map-funs-term f t) = linear-term t
  by (induct t) simp-all

lemma linear-term-map-inj-on-linear-term:
  assumes linear-term l
    and inj-on f (vars-term l)
  shows linear-term (map-vars-term f l)
  using assms
proof (induct l)
  case (Fun g ls)
  then have part:is-partition (map vars-term ls) by auto
  { fix l
    assume l:l ∈ set ls
    then have vars-term l ⊆ vars-term (Fun g ls) by auto
    then have inj-on f (vars-term l) using Fun(3) subset-inj-on by blast
```

```
    with Fun(1,2) l have linear-term (map-vars-term f l) by auto
  }
  moreover have is-partition (map (vars-term ∘ map-vars-term f) ls)
   using is-partition-inj-map[OF part, of f] Fun(3) by (simp add: o-def term.set-map)
  ultimately show ?case by auto
qed auto

lemma linear-term-replace-in-subst:
  assumes linear-term t
    and p ∈ poss t
    and t |- p = Var x
    and ⋀ y. y ∈ vars-term t ⟹ y ≠ x ⟹ σ y = τ y
    and τ x = s
  shows replace-at (t · σ) p s = t · τ
  using assms
proof (induct p arbitrary: t)
  case (Cons i p t)
  then obtain f ts where t [simp]: t = Fun f ts and i: i < length ts and p: p ∈
poss (ts ! i)
    by (cases t) auto
  from Cons have linear-term (ts ! i) and ts ! i |- p = Var x by auto
  have id: replace-at (ts ! i · σ) p (τ x) = ts ! i · τ using Cons by force
  let ?l = (take i (map (λt. t · σ) ts) @ (ts ! i · τ) # drop (Suc i) (map (λt. t ·
σ) ts))
  from i have len: length ts = length ?l by auto
  { fix j
    assume j: j < length ts
    have ts ! j · τ = ?l ! j
    proof (cases j = i)
      case True
      with i show ?thesis by (auto simp: nth-append)
    next
      case False
      let ?ts = map (λt. t · σ) ts
      from i j have le:j ≤ length ?ts i ≤ length ?ts by auto
      from nth-append-take-drop-is-nth-conv[OF le] False have ?l ! j = ?ts ! j by
simp
      also have ... = ts ! j · σ using j by simp
      also have ... = ts ! j · τ
      proof (rule term-subst-eq)
        fix y
        assume y: y ∈ vars-term (ts ! j)
        from p have ts ! i ⊵ ts ! i |- p by (rule subt-at-imp-supteq)
        then have x: x ∈ vars-term (ts ! i) using ‹ts ! i |- p = Var x›
          by (auto intro: subteq-Var-imp-in-vars-term)
        from Cons(2) have is-partition (map vars-term ts) by simp
        from this[unfolded is-partition-alt is-partition-alt-def, rule-format] j i False
        have vars-term (ts ! i) ∩ vars-term (ts ! j) = {} by auto
        with y x have y ≠ x by auto
```

```
      with Cons(5) y j show σ y = τ y by force
    qed
    finally show ?thesis by simp
  qed
 }
 then show ?case
  by (auto simp: ‹τ x = s›[symmetric] id nth-map[OF i, of λt. t · σ])
    (metis len map-nth-eq-conv[OF len])
qed auto


lemma var-in-linear-args:
 assumes linear-term (Fun f ts)
  and i < length ts and x ∈ vars-term (ts!i) and j < length ts ∧ j ≠ i
 shows x ∉ vars-term (ts!j)
proof−
 from assms(1) have is-partition (map vars-term ts)
  by simp
 with assms show ?thesis unfolding is-partition-alt is-partition-alt-def
  by auto
qed

lemma subt-at-linear:
 assumes linear-term t and p ∈ poss t
 shows linear-term (t|-p)
 using assms proof(induct p arbitrary:t)
 case (Cons i p)
 then obtain f ts where f:t = Fun f ts and i:i < length ts and p:p ∈ poss (ts!i)
  by (meson args-poss)
 with Cons(2) have linear-term (ts!i)
  by force
 then show ?case
  unfolding f subt-at.simps using Cons.hyps p by blast
qed simp

lemma linear-subterms-disjoint-vars:
 assumes linear-term t
  and p ∈ poss t and q ∈ poss t and p ⊥ q
 shows vars-term (t|-p) ∩ vars-term (t|-q) = {}
 using assms proof(induct t arbitrary: p q)
 case (Fun f ts)
 from Fun(3,5) obtain i p′ where i:i < length ts p′ ∈ poss (ts!i) and p:p =
i#p′
  by auto
 with Fun(4,5) obtain j q′ where j:j < length ts q′ ∈ poss (ts!j) and q:q =
j#q′
  by auto
 then show ?case proof(cases i=j)
  case True
```

```
    from Fun(5) have p′ ⊥ q′
      unfolding p q True by simp
    with Fun(1,2) i j have vars-term ((ts!j) |- p′) ∩ vars-term ((ts!j) |- q′) = {}
      unfolding True by auto
    then show ?thesis unfolding p q subt-at.simps True by simp
  next
    case False
    from i have vars-term ((Fun f ts)|-p) ⊆ vars-term (ts!i)
      unfolding p subt-at.simps by (simp add: vars-term-subt-at)
    moreover from j have vars-term ((Fun f ts)|-q) ⊆ vars-term (ts!j)
      unfolding q subt-at.simps by (simp add: vars-term-subt-at)
    ultimately show ?thesis using False Fun(2) i j
      by (meson disjoint-iff subsetD var-in-linear-args)
  qed
qed simp

lemma ground-imp-linear-term [simp]: ground t ⟹ linear-term t
  by (induct t) (auto simp add: is-partition-def ground-vars-term-empty)
```

exhaustively apply several maps on function symbols

```
fun map-funs-term-enum :: (′f ⇒ ′g list) ⇒ (′f, ′v) term ⇒ (′g, ′v) term list
where
  map-funs-term-enum fgs (Var x) = [Var x] |
  map-funs-term-enum fgs (Fun f ts) = (
    let
      lts = map (map-funs-term-enum fgs) ts;
      ss = concat-lists lts;
      gs = fgs f
    in concat (map (λg. map (Fun g) ss) gs))


lemma map-funs-term-enum:
  assumes gf: ⋀ f g. g ∈ set (fgs f) ⟹ gf g = f
  shows set (map-funs-term-enum fgs t) = {u. map-funs-term gf u = t ∧ (∀ g n.
(g,n) ∈ funas-term u ⟶ g ∈ set (fgs (gf g)))}
proof (induct t)
  case (Var x)
  show ?case (is - = ?R)
  proof −
    {
      fix t
      assume t ∈ ?R
      then have t = Var x by (cases t, auto)
    }
    then show ?thesis by auto
  qed
next
  case (Fun f ts)
  show ?case (is ?L = ?R)
```

```
proof −
  {
    fix i
    assume i < length ts
    then have ts ! i ∈ set ts by auto
    note Fun[OF this]
  } note ind = this
  let ?cf = λ u. (∀ g n. (g,n) ∈ funas-term u ⟶ g ∈ set (fgs (gf g)))
   have id: ?L = {Fun g ss | g ss. g ∈ set (fgs f) ∧ length ss = length ts ∧
(∀ i<length ts. ss ! i ∈ set (map-funs-term-enum fgs (ts ! i)))} (is - = ?M1) by
auto
   have ?R = {Fun g ss | g ss. map-funs-term gf (Fun g ss) = Fun f ts ∧ ?cf
(Fun g ss)} (is - = ?M2)
  proof −
    {
      fix u
      assume u: u ∈ ?R
      then obtain g ss where u = Fun g ss by (cases u, auto)
      with u have u ∈ ?M2 by auto
    }
    then have ?R ⊆ ?M2 by auto
    moreover have ?M2 ⊆ ?R by blast
    finally show ?thesis by auto
  qed
  also have ... = ?M1
  proof −
    {
      fix u
      assume u ∈ ?M1
      then obtain g ss where u: u = Fun g ss and g: g ∈ set (fgs f) and
         len: length ss = length ts and rec: ⋀ i. i < length ts ⟹ ss ! i ∈ set
(map-funs-term-enum fgs (ts ! i)) by auto
      from gf[OF g] have gf: gf g = f by simp
      {
        fix i
        assume i: i < length ts
        from ind[OF i] rec[OF i]
        have map-funs-term gf (ss ! i) = ts ! i by auto
      } note ssts = this
      have map (map-funs-term gf) ss = ts
        by (unfold map-nth-eq-conv[OF len], insert ssts, auto)
      with gf
      have mt: map-funs-term gf (Fun g ss) = Fun f ts by auto
      have u ∈ ?M2
      proof (rule, rule, rule, rule, rule u, rule, rule mt, intro allI impI)
        fix h n
        assume h: (h,n) ∈ funas-term (Fun g ss)
        show h ∈ set (fgs (gf h))
        proof (cases (h,n) = (g,length ss))
```

```
          case True
          then have h = g by auto
          with g gf show ?thesis by auto
        next
          case False
           with h obtain s where s: s ∈ set ss and h: (h,n) ∈ funas-term s by
auto
          from s[unfolded set-conv-nth] obtain i where i: i < length ss and si: s
= ss ! i by force
          from i len have i': i < length ts by auto
          from ind[OF i'] rec[OF i'] h[unfolded si] show ?thesis by auto
        qed
      qed
    }
    then have m1m2: ?M1 ⊆ ?M2 by blast
    {
      fix u
      assume u: u ∈ ?M2
      then obtain g ss where u: u = Fun g ss
        and map: map-funs-term gf (Fun g ss) = Fun f ts
        and c: ?cf (Fun g ss)
        by blast
      from map have len: length ss = length ts by auto
      from map have g: gf g = f by auto
      from map have map: map (map-funs-term gf) ss = ts by auto
      from c have g2: g ∈ set (fgs f) using g by auto
      have u ∈ ?M1
      proof (intro CollectI exI conjI allI impI, rule u, rule g2, rule len)
        fix i
        assume i: i < length ts
        with map[unfolded map-nth-eq-conv[OF len]]
        have id: map-funs-term gf (ss ! i) = ts ! i by auto
        from i len have si: ss ! i ∈ set ss by auto
        show ss ! i ∈ set (map-funs-term-enum fgs (ts ! i))
          unfolding ind[OF i]
        proof (intro CollectI conjI impI allI, rule id)
          fix g n
          assume (g,n) ∈ funas-term (ss ! i)
          with c si
          show g ∈ set (fgs (gf g)) by auto
        qed
      qed
    }
    then have m2m1: ?M2 ⊆ ?M1 by blast
    show ?M2 = ?M1
      by (rule, rule m2m1, rule m1m2)
  qed
  finally show ?case unfolding id by simp
qed
```

qed

declare *map-funs-term-enum.simps*[*simp del*]

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.