First-Order Terms*

Christian Sternagel René Thiemann

April 13, 2025

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* ⇒ '*v*, *v2* :: '*v* ⇒ '*v*) | *v1 v2*. *inj v1* ∧ *inj v2* ∧ *range v1* ∩ *range v2* = {} }
⟨*proof*⟩

setup-lifting *type-definition-renaming2*

lift-definition *rename-1* :: '*v* :: *infinite renaming2* ⇒ '*v* ⇒ '*v* **is** *fst* ⟨*proof*⟩
lift-definition *rename-2* :: '*v* :: *infinite renaming2* ⇒ '*v* ⇒ '*v* **is** *snd* ⟨*proof*⟩

lemma *rename-12*: *inj* (*rename-1 r*) *inj* (*rename-2 r*) *range* (*rename-1 r*) ∩ *range* (*rename-2 r*) = {}
⟨*proof*⟩

end

2.3 Make lists instances of the infinite-class.

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

instance *list* :: (*type*) *infinite*
⟨*proof*⟩

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*''))
⟨*proof*⟩

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 ∀ *i*. *f i* ≥ *f* (*Suc i*)
shows ∃ *N*. ∀ *i* > *N*. *f i* = *f* (*Suc i*)
⟨*proof*⟩

lemma *inc-seq-greater*:
fixes *f* :: *nat seq*
assumes ∀ *i*. *f i* < *f* (*Suc i*)
shows ∃ *i*. *f i* > *N*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *bij-betw-extend*:
assumes ∗: *bij-betw f A B*

and $A \subseteq V$
and $B \subseteq V$
and *finite* A
shows $\exists g.$ *finite* $\{x.\ g\ x \neq x\} \wedge$
$(\forall x{\in}UNIV - (A \cup B).\ g\ x = x) \wedge$
$(\forall x{\in}A.\ g\ x = f\ x) \wedge$
bij-betw $g\ V\ V$
⟨*proof*⟩

2.7 Merging Functions

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

lemma *fun-merge-eq-nth*:
assumes *i*: *i* < *length as*
and *a*: *a* ∈ *as* ! *i*
and *ident*: ⋀ *i j a. i* < *length as* ⟹ *j* < *length as* ⟹ *a* ∈ *as* ! *i* ⟹ *a* ∈ *as* ! *j* ⟹ (*fs* ! *i*) *a* = (*fs* ! *j*) *a*
shows *fun-merge fs as a* = (*fs* ! *i*) *a*
⟨*proof*⟩

lemma *fun-merge-part*:
assumes ∀ *i*<*length as*.∀ *j*<*length as. i* ≠ *j* ⟶ *as* ! *i* ∩ *as* ! *j* = {}
and *i* < *length as*
and *a* ∈ *as* ! *i*
shows *fun-merge fs as a* = (*fs* ! *i*) *a*
⟨*proof*⟩

lemma *fun-merge*:
assumes *part*: ∀ *i*<*length Xs*.∀ *j*<*length Xs. i* ≠ *j* ⟶ *Xs* ! *i* ∩ *Xs* ! *j* = {}
shows ∃ *σ*. ∀ *i*<*length Xs*. ∀ *x*∈ *Xs* ! *i*. *σ x* = *τ i x*
⟨*proof*⟩

end

2.8 The Option Monad

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

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

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

lemma *guard-cong* [*fundef-cong*]:

$b = c \Longrightarrow (c \Longrightarrow m = n) \Longrightarrow guard\ b >> m = guard\ c >> n$
⟨*proof*⟩

lemma *guard-simps*:
$guard\ b = Some\ x \longleftrightarrow b$
$guard\ b = None \longleftrightarrow \neg\ b$
⟨*proof*⟩

lemma *guard-elims*[*elim*]:
$guard\ b = Some\ x \Longrightarrow (b \Longrightarrow P) \Longrightarrow P$
$guard\ b = None \Longrightarrow (\neg\ b \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *guard-intros* [*intro*, *simp*]:
$b \Longrightarrow guard\ b = Some\ ()$
$\neg\ b \Longrightarrow guard\ b = None$
⟨*proof*⟩

lemma *guard-True* [*simp*]: *guard True* = *Some* () ⟨*proof*⟩
lemma *guard-False* [*simp*]: *guard False* = *None* ⟨*proof*⟩

lemma *guard-and-to-bind*: $guard\ (a \wedge b) = guard\ a \gg\!= (\lambda\ \text{-}.\ guard\ b)$ ⟨*proof*⟩

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

induction scheme for zip

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

lemma *zip-option-same*[*simp*]:
zip-option xs xs = *Some* (*zip xs xs*)
⟨*proof*⟩

lemma *zip-option-zip-conv*:
zip-option xs ys = *Some zs* ⟷ *length ys* = *length xs* ∧ *length zs* = *length xs* ∧ *zs* = *zip xs ys*
⟨*proof*⟩

lemma *zip-option-None*:
zip-option xs ys = *None* ⟷ *length xs* ≠ *length ys*

⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *zip-option-elims* [*elim*]:
  zip-option xs ys = *Some zs*
    ⟹ (⟦*length ys* = *length xs*; *length zs* = *length xs*; *zs* = *zip xs ys*⟧ ⟹ *P*)
    ⟹ *P*
  zip-option xs ys = *None* ⟹ (*length xs* ≠ *length ys* ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

lemma *zip-option-simps* [*simp*]:
  zip-option xs ys = *None* ⟹ *length xs* = *length ys* ⟹ *False*
  zip-option xs ys = *None* ⟹ *length xs* ≠ *length ys*
  zip-option xs ys = *Some zs* ⟹ *zs* = *zip xs ys*
  ⟨*proof*⟩

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

lemma *mapM-None*:
  mapM f xs = *None* ⟷ (∃ *x*∈*set xs*. *f x* = *None*)
⟨*proof*⟩

lemma *mapM-Some*:
  mapM f xs = *Some ys* ⟹ *ys* = *map* (λ*x*. *the* (*f x*)) *xs* ∧ (∀ *x*∈*set xs*. *f x* ≠ *None*)
⟨*proof*⟩

lemma *mapM-Some-idx*:
  assumes *some*: *mapM f xs* = *Some ys* **and** *i*: *i* < *length xs*
  shows ∃ *y*. *f* (*xs* ! *i*) = *Some y* ∧ *ys* ! *i* = *y*
⟨*proof*⟩

lemma *mapM-cong* [*fundef-cong*]:
  assumes *xs* = *ys* **and** ⋀*x*. *x* ∈ *set ys* ⟹ *f x* = *g x*
  shows *mapM f xs* = *mapM g ys*

⟨*proof*⟩

lemma *mapM-map*:
mapM f xs = (*if* (∀ *x*∈*set xs*. *f x* ≠ *None*) *then Some* (*map* (λ*x*. *the* (*f x*)) *xs*) *else None*)
⟨*proof*⟩

lemma *mapM-mono* [*partial-function-mono*]:
fixes *C* :: *'a* ⇒ (*'b* ⇒ *'c option*) ⇒ *'d option*
assumes *C*: ⋀*y*. *mono-option* (*C y*)
shows *mono-option* (λ*f*. *mapM* (λ*y*. *C y f*) *B*)
⟨*proof*⟩

end

3 First-Order Terms

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

datatype (*funs-term* : *'f*, *vars-term* : *'v*) *term* =
is-Var: *Var* (*the-Var*: *'v*) |
Fun 'f (*args* : (*'f*, *'v*) *term list*)
where
args (*Var* -) = []

lemmas *is-VarI* = *term.disc*(*1*)
lemmas *is-FunI* = *term.disc*(*2*)

abbreviation *is-Fun t* ≡ ¬ *is-Var t*

lemma *is-VarE* [*elim*]:
is-Var t ⟹ (⋀*x*. *t* = *Var x* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *is-FunE* [*elim*]:
is-Fun t ⟹ (⋀*f ts*. *t* = *Fun f ts* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *inj-on-Var*[*simp*]:
inj-on Var A
⟨*proof*⟩

lemma
inj-on-Fun-fun[*simp*]: ⋀*A ts*. *inj-on* (λ*f*. *Fun f ts*) *A* **and**
inj-on-Fun-args[*simp*]: ⋀*A f*. *inj-on* (λ*ts*. *Fun f ts*) *A* **and**

inj-on-Fun[*simp*]: $\bigwedge A.$ *inj-on Fun A*
⟨*proof*⟩

lemma *member-image-the-Var-image-subst*:
assumes *is-var-σ*: $\forall x.$ *is-Var* $(\sigma\ x)$
shows $x \in$ *the-Var* ‘ σ ‘ $V \longleftrightarrow$ *Var* $x \in \sigma$ ‘ V
⟨*proof*⟩

lemma *image-the-Var-image-subst-renaming-eq*:
assumes *is-var-σ*: $\forall x.$ *is-Var* $(\varrho\ x)$
shows *the-Var* ‘ ϱ ‘ $V = (\bigcup x \in V.$ *vars-term* $(\varrho\ x))$
⟨*proof*⟩

The variables of a term as multiset.

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

lemma *set-mset-vars-term-ms* [*simp*]:
set-mset (*vars-term-ms t*) = *vars-term t*
⟨*proof*⟩

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

⟨*ML*⟩
The *root symbol* of a term is defined by:

fun *root* :: $('f, 'v)$ *term* $\Rightarrow$ $('f \times nat)$ *option*
where
root (*Var x*) = *None* |
root (*Fun f ts*) = *Some* (*f*, *length ts*)

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

lemma *finite-Union-vars-term*:
finite $(\bigcup t \in$ *set ts. vars-term t*$)$
⟨*proof*⟩

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**
$I\llbracket$*Var x*$\rrbracket\alpha = \alpha\ x$
| $I\llbracket$*Fun f ss*$\rrbracket\alpha = I\ f\ [I\llbracket s\rrbracket\alpha.\ s \leftarrow ss]$

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

lemma *eval-same-vars*:
assumes $\forall x \in$ *vars-term s*. $\alpha\ x = \beta\ x$
shows $I[\![s]\!]\alpha = I[\![s]\!]\beta$
⟨*proof*⟩

lemma *eval-same-vars-cong*:
assumes *ref*: $s = t$ **and** *v*: $\bigwedge x.\ x \in$ *vars-term* $s \Longrightarrow \alpha\ x = \beta\ x$
shows $I[\![s]\!]\alpha = I[\![t]\!]\beta$
⟨*proof*⟩

lemma *eval-with-fresh-var*: $x \notin$ *vars-term* $s \Longrightarrow I[\![s]\!]\alpha(x{:=}a) = I[\![s]\!]\alpha$
⟨*proof*⟩

lemma *eval-map-term*: $I[\![$*map-term ff fv s*$]\!]\alpha = (I \circ ff)[\![s]\!](\alpha \circ fv)$
⟨*proof*⟩

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

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

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

definition *eval-subst* (‹-[![-]!]$_s$ -› [*999*,*1*,*100*]*100*) **where**
$(I[\![\vartheta]\!]_s\ \alpha) \equiv \lambda x.\ I[\![\vartheta\ x]\!]\alpha$

lemma *eval-subst*: $I[\![s{\cdot}\vartheta]\!]\alpha = I[\![s]\!]\ I[\![\vartheta]\!]_s\ \alpha$
⟨*proof*⟩

abbreviation
subst-compose :: $('f, 'u, 'v)$ *gsubst* $\Rightarrow ('f, 'v, 'w)$ *gsubst* $\Rightarrow ('f, 'u, 'w)$ *gsubst*
(**infixl** ‹$\circ_s$› *75*)
where
$\sigma \circ_s \vartheta \equiv Fun[\![\sigma]\!]_s\ \vartheta$

lemmas *subst-compose-def* $=$ *eval-subst-def*[*of Fun*]

lemma *subst-subst-compose* [*simp*]:
$t \cdot (\sigma \circ_s \tau) = t \cdot \sigma \cdot \tau$
⟨*proof*⟩

lemma *subst-compose-assoc*:
$\sigma \circ_s \tau \circ_s \mu = \sigma \circ_s (\tau \circ_s \mu)$
⟨*proof*⟩

lemma *subst-apply-term-empty* [*simp*]:
$t \cdot Var = t$
⟨*proof*⟩

interpretation *subst-monoid-mult*: *monoid-mult Var* ($\circ_s$)
⟨*proof*⟩

lemma *term-subst-eq*:
assumes $\bigwedge x.\ x \in$ *vars-term* $t \Longrightarrow \sigma\ x = \tau\ x$
shows $t \cdot \sigma = t \cdot \tau$
⟨*proof*⟩

lemma *term-subst-eq-rev*:
$t \cdot \sigma = t \cdot \tau \Longrightarrow \forall x \in$ *vars-term* $t.\ \sigma\ x = \tau\ x$
⟨*proof*⟩

lemma *term-subst-eq-conv*:
$t \cdot \sigma = t \cdot \tau \longleftrightarrow (\forall x \in$ *vars-term* $t.\ \sigma\ x = \tau\ x)$
⟨*proof*⟩

lemma *subst-term-eqI*:
assumes $(\bigwedge t.\ t \cdot \sigma = t \cdot \tau)$
shows $\sigma = \tau$
⟨*proof*⟩

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*⟩

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*⟩

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 $y \in range\text{-}vars\ \sigma$
⟨*proof*⟩

lemma *subst-domain-Var* [*simp*]:
$subst\text{-}domain\ Var = \{\}$
⟨*proof*⟩

lemma *subst-range-Var*[*simp*]:
$subst\text{-}range\ Var = \{\}$
⟨*proof*⟩

lemma *range-vars-Var*[*simp*]:
$range\text{-}vars\ Var = \{\}$
⟨*proof*⟩

lemma *subst-apply-term-ident*:
$vars\text{-}term\ t \cap subst\text{-}domain\ \sigma = \{\} \Longrightarrow t \cdot \sigma = t$
⟨*proof*⟩

lemma *vars-term-subst-apply-term*:
$vars\text{-}term\ (t \cdot \sigma) = (\bigcup x \in vars\text{-}term\ t.\ vars\text{-}term\ (\sigma\ x))$
⟨*proof*⟩

corollary *vars-term-subst-apply-term-subset*:
$vars\text{-}term\ (t \cdot \sigma) \subseteq vars\text{-}term\ t - subst\text{-}domain\ \sigma \cup range\text{-}vars\ \sigma$
⟨*proof*⟩

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

lemma *inv-renaming-sound*:
assumes *is-var-ϱ*: $\forall x.\ is\text{-}Var\ (\varrho\ x)$ **and** $inj\ \varrho$
shows $\varrho \circ_s (Var \circ (inv\ (the\text{-}Var \circ \varrho))) = Var$
⟨*proof*⟩

lemma *ex-inverse-of-renaming*:
assumes $\forall x.\ is\text{-}Var\ (\varrho\ x)$ **and** $inj\ \varrho$
shows $\exists \tau.\ \varrho \circ_s \tau = Var$
⟨*proof*⟩

lemma *vars-term-subst*:
$vars\text{-}term\ (t \cdot \sigma) = \bigcup (vars\text{-}term\ `\ \sigma\ `\ vars\text{-}term\ t)$
⟨*proof*⟩

lemma *range-varsE* [*elim*]:
assumes $x \in range\text{-}vars\ \sigma$

and $\bigwedge t.\ x \in \mathit{vars\text{-}term}\ t \Longrightarrow t \in \mathit{subst\text{-}range}\ \sigma \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *range-vars-subst-compose-subset*:
range-vars $(\sigma \circ_s \tau) \subseteq (\mathit{range\text{-}vars}\ \sigma - \mathit{subst\text{-}domain}\ \tau) \cup \mathit{range\text{-}vars}\ \tau$ (**is** $?L \subseteq ?R$)
⟨*proof*⟩

definition $\mathit{subst}\ x\ t = \mathit{Var}\ (x := t)$

lemma *subst-simps* [*simp*]:
$\mathit{subst}\ x\ t\ x = t$
$\mathit{subst}\ x\ (\mathit{Var}\ x) = \mathit{Var}$
⟨*proof*⟩

lemma *subst-subst-domain* [*simp*]:
$\mathit{subst\text{-}domain}\ (\mathit{subst}\ x\ t) = (\mathit{if}\ t = \mathit{Var}\ x\ \mathit{then}\ \{\}\ \mathit{else}\ \{x\})$
⟨*proof*⟩

lemma *subst-subst-range* [*simp*]:
$\mathit{subst\text{-}range}\ (\mathit{subst}\ x\ t) = (\mathit{if}\ t = \mathit{Var}\ x\ \mathit{then}\ \{\}\ \mathit{else}\ \{t\})$
⟨*proof*⟩

lemma *subst-apply-left-idemp* [*simp*]:
assumes $\sigma\ x = t \cdot \sigma$
shows $s \cdot \mathit{subst}\ x\ t \cdot \sigma = s \cdot \sigma$
⟨*proof*⟩

lemma *subst-compose-left-idemp* [*simp*]:
assumes $\sigma\ x = t \cdot \sigma$
shows $\mathit{subst}\ x\ t \circ_s \sigma = \sigma$
⟨*proof*⟩

lemma *subst-ident* [*simp*]:
assumes $x \notin \mathit{vars\text{-}term}\ t$
shows $t \cdot \mathit{subst}\ x\ u = t$
⟨*proof*⟩

lemma *subst-self-idemp* [*simp*]:
$x \notin \mathit{vars\text{-}term}\ t \Longrightarrow \mathit{subst}\ x\ t \circ_s \mathit{subst}\ x\ t = \mathit{subst}\ x\ t$
⟨*proof*⟩

type-synonym $('f, 'v)\ \mathit{terms} = ('f, 'v)\ \mathit{term\ set}$

Applying a substitution to every term of a given set.

abbreviation
$\mathit{subst\text{-}apply\text{-}set} :: ('f, 'v)\ \mathit{terms} \Rightarrow ('f, 'v, 'w)\ \mathit{gsubst} \Rightarrow ('f, 'w)\ \mathit{terms}$ (**infixl** $\langle \cdot_{set} \rangle$ *60*)

where
$T \cdot_{set} \sigma \equiv (\lambda t.\ t \cdot \sigma)\ `\ T$

Composition of substitutions

lemma *subst-compose*: $(\sigma \circ_s \tau)\ x = \sigma\ x \cdot \tau$ ⟨*proof*⟩

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** $\sigma\ y = Var\ x$
⟨*proof*⟩

lemma *subst-domain-subst-compose*:
subst-domain $(\sigma \circ_s \tau) =$
$(subst\text{-}domain\ \sigma - \{x.\ \exists y.\ \sigma\ x = Var\ y \wedge \tau\ y = Var\ x\}) \cup$
$(subst\text{-}domain\ \tau - subst\text{-}domain\ \sigma)$
⟨*proof*⟩

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*⟩

lemma *subst-compose-apply-eq-apply-lhs*:
assumes
$range\text{-}vars\ \sigma \cap subst\text{-}domain\ \delta = \{\}$
$x \notin subst\text{-}domain\ \delta$
shows $(\sigma \circ_s \delta)\ x = \sigma\ x$
⟨*proof*⟩

lemma *subst-apply-term-subst-apply-term-eq-subst-apply-term-lhs*:
assumes $range\text{-}vars\ \sigma \cap subst\text{-}domain\ \delta = \{\}$ **and** $vars\text{-}term\ t \cap subst\text{-}domain\ \delta = \{\}$
shows $t \cdot \sigma \cdot \delta = t \cdot \sigma$
⟨*proof*⟩

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

lemma *num-funs-0*:
assumes $num\text{-}funs\ t = 0$
obtains x **where** $t = Var\ x$
⟨*proof*⟩

lemma *num-funs-subst*:

num-funs $(t \cdot \sigma) \geq$ *num-funs* t
⟨*proof*⟩

lemma *sum-list-map-num-funs-subst*:
assumes *sum-list* (*map* (*num-funs* ∘ (λ*t*. $t \cdot \sigma$)) *ts*) = *sum-list* (*map num-funs ts*)
shows ∀ $i <$ *length ts*. *num-funs* (*ts* ! $i \cdot \sigma$) = *num-funs* (*ts* ! i)
⟨*proof*⟩

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

lemma *finite-subst-domain-subst*:
finite (*subst-domain* (*subst* x y))
⟨*proof*⟩

lemma *subst-domain-compose*:
subst-domain ($\sigma \circ_s \tau$) ⊆ *subst-domain* σ ∪ *subst-domain* τ
⟨*proof*⟩

lemma *vars-term-disjoint-imp-unifier*:
fixes σ :: ('*f*, '*v*, '*w*) *gsubst*
assumes *vars-term* s ∩ *vars-term* t = {}
and $s \cdot \sigma = t \cdot \tau$
shows ∃ μ :: ('*f*, '*v*, '*w*) *gsubst*. $s \cdot \mu = t \cdot \mu$
⟨*proof*⟩

lemma *vars-term-subset-subst-eq*:
assumes *vars-term* $t \subseteq$ *vars-term* s
and $s \cdot \sigma = s \cdot \tau$
shows $t \cdot \sigma = t \cdot \tau$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *restrict-subst-domain-Var*[*simp*]:
restrict-subst-domain V *Var* = *Var*
⟨*proof*⟩

lemma *subst-domain-restrict-subst-domain*[*simp*]:

subst-domain (*restrict-subst-domain* V σ) = $V \cap$ *subst-domain* σ
⟨*proof*⟩

lemma *subst-apply-term-restrict-subst-domain*:
vars-term $t \subseteq V \Longrightarrow t \cdot$ *restrict-subst-domain* V $\sigma = t \cdot \sigma$
⟨*proof*⟩

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* $\sigma = \sigma$
⟨*proof*⟩

lemma *rename-subst-domain-Var-rhs*[*simp*]:
rename-subst-domain ϱ *Var* = *Var*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *subst-range-rename-subst-domain-subset*:
assumes *inj* ϱ
shows *subst-range* (*rename-subst-domain* ϱ σ) $\subseteq$ *subst-range* σ
⟨*proof*⟩

lemma *range-vars-rename-subst-domain-subset*:
assumes *inj* ϱ
shows *range-vars* (*rename-subst-domain* ϱ σ) $\subseteq$ *range-vars* σ
⟨*proof*⟩

lemma *renaming-cancels-rename-subst-domain*:
assumes *is-var-ϱ*: $\forall x$. *is-Var* (ϱ x) **and** *inj* ϱ **and** *vars-t*: *vars-term* $t \subseteq$ *subst-domain* σ
shows $t \cdot \varrho \cdot$ *rename-subst-domain* ϱ $\sigma = t \cdot \sigma$
⟨*proof*⟩

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 σ = σ
⟨*proof*⟩

lemma *rename-subst-domain-range-Var-rhs*[*simp*]:
rename-subst-domain-range ϱ *Var* = *Var*
⟨*proof*⟩

lemma *subst-compose-renaming-rename-subst-domain-range*:
fixes σ ϱ :: ('*f*, '*v*) *subst*
assumes *is-var-*ϱ: ∀ *x*. *is-Var* (ϱ *x*) **and** *inj* ϱ
shows ϱ $\circ_s$ *rename-subst-domain-range* ϱ σ = σ $\circ_s$ ϱ
⟨*proof*⟩

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: ∀ *x*. *is-Var* (ϱ *x*) **and** *inj* ϱ
shows *t* · ϱ · *rename-subst-domain-range* ϱ σ = *t* · σ · ϱ
⟨*proof*⟩

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

fun *ground* :: ('*f*, '*v*) *term* ⇒ *bool*
where
ground (*Var x*) ⟷ *False* |
ground (*Fun f ts*) ⟷ (∀ *t* ∈ *set ts*. *ground t*)

lemma *ground-vars-term-empty*:
ground t ⟷ *vars-term t* = {}
⟨*proof*⟩

lemma *ground-subst* [*simp*]:
ground (*t* · σ) ⟷ (∀ *x* ∈ *vars-term t*. *ground* (σ *x*))
⟨*proof*⟩

lemma *ground-subst-apply*:
assumes *ground t*
shows *t* · σ = *t*
⟨*proof*⟩

Just changing the variables in a term

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

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

⟨*proof*⟩

lemma *map-vars-term-eq*:
map-vars-term f s = *s* · (*Var* ∘ *f*)
⟨*proof*⟩

lemma *ground-map-vars-term* [*simp*]:
ground (*map-vars-term f t*) = *ground t*
⟨*proof*⟩

lemma *map-vars-term-subst* [*simp*]:
map-vars-term f (*t* · *σ*) = *t* · (*λ x. map-vars-term f* (*σ x*))
⟨*proof*⟩

lemma *map-vars-term-compose*:
map-vars-term m1 (*map-vars-term m2 t*) = *map-vars-term* (*m1 o m2*) *t*
⟨*proof*⟩

lemma *map-vars-term-id* [*simp*]:
map-vars-term id t = *t*
⟨*proof*⟩

lemma *eval-map-vars*: *I*⟦*map-vars-term f t*⟧*α* = *I*⟦*t*⟧(*α*∘*f*)
⟨*proof*⟩

lemma *apply-subst-map-vars-term*:
map-vars-term m t · *σ* = *t* · (*σ* ∘ *m*)
⟨*proof*⟩

lemmas *eval-o* = *eval-map-vars*[*symmetric*]

lemma *eval-eq-map-vars*:
assumes *1*: *t* = *map-vars-term f s* **and** *2*: ⋀*x*. *x* ∈ *vars-term s* ⟹ *α x* = *β* (*f x*)
shows *I*⟦*s*⟧*α* = *I*⟦*t*⟧*β*
⟨*proof*⟩

lemma *ground-term-subst*: *vars-term t* = {} ⟹ *t*·*ϑ* = *t*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fold-pair-size-plus*:
fold-mset ((+) ∘ *pair-size*) *0 M* + *n* = *fold-mset* ((+) ∘ *pair-size*) *n M*
⟨*proof*⟩

lemma *size-mset-union* [*simp*]:
size-mset (*M* + *N*) = *size-mset N* + *size-mset M*
⟨*proof*⟩

lemma *size-mset-add-mset* [*simp*]:
size-mset (*add-mset x M*) = *pair-size x* + (*size-mset M*)
⟨*proof*⟩

lemma *nonempty-size-mset* [*simp*]:
assumes *M* ≠ {#}
shows *size-mset M* > *0*
⟨*proof*⟩

lemma *size-mset-singleton* [*simp*]:
size-mset {#(*l*, *r*)#} = *size l* + *size r* + *1*
⟨*proof*⟩

lemma *size-mset-empty* [*simp*]:
size-mset {#} = *0*
⟨*proof*⟩

lemma *size-mset-set-zip-leq*:
size-mset (*mset* (*zip ss ts*)) ≤ *size-list size ss* + *size-list size ts*
⟨*proof*⟩

lemma *size-mset-Fun-less*:
size-mset {#(*Fun f ss*, *Fun g ts*)#} > *size-mset* (*mset* (*zip ss ts*))
⟨*proof*⟩

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)#})*
⟨*proof*⟩

3.5.1 Substitutions

Applying a substitution to a multiset of term pairs.

definition *subst-mset σ M = image-mset (λp. (fst p · σ, snd p · σ)) M*
lemma *subst-mset-empty* [*simp*]:
subst-mset σ {#} = {#}
⟨*proof*⟩

lemma *subst-mset-union*:
subst-mset σ (M + N) = subst-mset σ M + subst-mset σ N
⟨*proof*⟩

lemma *subst-mset-Var* [*simp*]:
subst-mset Var M = M
⟨*proof*⟩

lemma *subst-mset-subst-compose* [*simp*]:
subst-mset (σ ∘$_s$ τ) M = subst-mset τ (subst-mset σ M)
⟨*proof*⟩

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) ∪ vars-term (snd r)) M))*

lemma *vars-mset-singleton* [*simp*]:
vars-mset {#p#} = vars-term (fst p) ∪ vars-term (snd p)
⟨*proof*⟩

lemma *vars-mset-union* [*simp*]:
vars-mset (A + B) = vars-mset A ∪ vars-mset B
⟨*proof*⟩

lemma *vars-mset-add-mset* [*simp*]:
vars-mset (add-mset x M) = vars-term (fst x) ∪ vars-term (snd x) ∪ vars-mset M
⟨*proof*⟩

lemma *vars-mset-set-zip* [*simp*]:
assumes *length xs = length ys*
shows *vars-mset (mset (zip xs ys)) = (*$\bigcup$*x∈set xs ∪ set ys. vars-term x)*

⟨*proof*⟩

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)
⟨*proof*⟩
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*⟩

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*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *finite-vars-mset*:
finite (*vars-mset* A)
⟨*proof*⟩

end

4 Abstract Matching

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

lemma *singleton-eq-union-iff* [*iff*]:
$\{\#x\#\} = M + \{\#y\#\} \longleftrightarrow M = \{\#\} \wedge x = y$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *matchers-empty* [*simp*]:
matchers {} = *UNIV*
⟨*proof*⟩

lemma *matchers-insert* [*simp*]:
matchers (*insert e P*) = {σ. *fst e* · σ = *snd e*} ∩ *matchers P*
⟨*proof*⟩

lemma *matchers-Un* [*simp*]:
matchers (*P* ∪ *P'*) = *matchers P* ∩ *matchers P'*
⟨*proof*⟩

lemma *matchers-set-zip* [*simp*]:
assumes *length ss* = *length ts*
shows *matchers* (*set* (*zip ss ts*)) = {σ. *map* (λ*t*. *t* · σ) *ss* = *ts*}
⟨*proof*⟩

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

lemma *matchers-map-empty* [*simp*]:
matchers-map Map.empty = *UNIV*
⟨*proof*⟩

lemma *matchers-map-upd* [*simp*]:
assumes σ *x* = *None* ∨ σ *x* = *Some t*
shows *matchers-map* (λ*y*. *if y* = *x then Some t else* σ *y*) =
matchers-map σ ∩ {τ. τ *x* = *t*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *matchers-map-upd'* [*simp*]:
assumes $\sigma\ x = None \vee \sigma\ x = Some\ t$
shows $matchers\text{-}map\ (\sigma\ (x \mapsto t)) = matchers\text{-}map\ \sigma \cap \{\tau.\ \tau\ x = t\}$
⟨*proof*⟩

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

lemma *MATCH1-matchers* [*simp*]:
assumes *MATCH1 x y*
shows $matchers\text{-}map\ (snd\ x) \cap matchers\ (set\text{-}mset\ (fst\ x)) =$
$matchers\text{-}map\ (snd\ y) \cap matchers\ (set\text{-}mset\ (fst\ y))$
⟨*proof*⟩

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

lemma *MATCH1-matchrel-conv*:
$MATCH1\ x\ y \longleftrightarrow (x,\ y) \in matchrel$
⟨*proof*⟩

lemma *matchrel-rtrancl-matchers* [*simp*]:
assumes $(x,\ y) \in matchrel^*$
shows $matchers\text{-}map\ (snd\ x) \cap matchers\ (set\text{-}mset\ (fst\ x)) =$
$matchers\text{-}map\ (snd\ y) \cap matchers\ (set\text{-}mset\ (fst\ y))$
⟨*proof*⟩

lemma *subst-of-map-in-matchers-map* [*simp*]:
$subst\text{-}of\text{-}map\ d\ m \in matchers\text{-}map\ m$
⟨*proof*⟩

lemma *matchrel-sound*:
assumes $((P,\ Map.empty),\ (\{\#\},\ \sigma)) \in matchrel^*$
shows $subst\text{-}of\text{-}map\ d\ \sigma \in matchers\ (set\text{-}mset\ P)$
⟨*proof*⟩

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

definition $matchless = inv\text{-}image\ (measure\ size\text{-}mset)\ fst$

lemma *wf-matchless*:
wf matchless
⟨*proof*⟩

lemma *MATCH1-matchless*:
assumes *MATCH1 x y*
shows $(y, x) \in matchless$
⟨*proof*⟩

lemma *converse-matchrel-subset-matchless*:
$matchrel^{-1} \subseteq matchless$
⟨*proof*⟩

lemma *wf-converse-matchrel*:
$wf\ (matchrel^{-1})$
⟨*proof*⟩

lemma *MATCH1-singleton-Var* [*intro*]:
$\sigma\ x = None \Longrightarrow MATCH1\ (\{\#(Var\ x,\ t)\#\},\ \sigma)\ (\{\#\},\ \sigma\ (x \mapsto t))$
$\sigma\ x = Some\ t \Longrightarrow MATCH1\ (\{\#(Var\ x,\ t)\#\},\ \sigma)\ (\{\#\},\ \sigma\ (x \mapsto t))$
⟨*proof*⟩

lemma *MATCH1-singleton-Fun* [*intro*]:
$length\ ss = length\ ts \Longrightarrow MATCH1\ (\{\#(Fun\ f\ ss,\ Fun\ f\ ts)\#\},\ \sigma)\ (mset\ (zip\ ss\ ts),\ \sigma)$
⟨*proof*⟩

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

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*⟩

lemma *ne-matchers-imp-matchrel*:
assumes $matchers\text{-}map\ \sigma \cap matchers\ \{e\} \neq \{\}$
shows $\exists\, y.\ ((\{\#e\#\},\ \sigma),\ y) \in matchrel$
⟨*proof*⟩

lemma *MATCH1-mono*:
assumes $MATCH1\ (P,\ \sigma)\ (P',\ \sigma')$
shows $MATCH1\ (P + M,\ \sigma)\ (P' + M,\ \sigma')$
⟨*proof*⟩

lemma *matchrel-mono*:
assumes $(x,\ y) \in matchrel$
shows $((fst\ x + M,\ snd\ x),\ (fst\ y + M,\ snd\ y)) \in matchrel$

⟨*proof*⟩

lemma *matchrel-rtrancl-mono*:
assumes $(x, y) \in matchrel^*$
shows $((fst\ x + M, snd\ x), (fst\ y + M, snd\ y)) \in matchrel^*$
⟨*proof*⟩

lemma *ne-matchers-imp-empty-or-matchrel*:
assumes $matchers\text{-}map\ \sigma \cap matchers\ (set\text{-}mset\ P) \neq \{\}$
shows $P = \{\#\} \vee (\exists y.\ ((P, \sigma), y) \in matchrel)$
⟨*proof*⟩

lemma *matchrel-imp-converse-matchless* [*dest*]:
$(x, y) \in matchrel \Longrightarrow (y, x) \in matchless$
⟨*proof*⟩

lemma *ne-matchers-imp-empty*:
fixes $P :: (('f, 'v)\ term \times ('f, 'w)\ term)\ multiset$
assumes $matchers\text{-}map\ \sigma \cap matchers\ (set\text{-}mset\ P) \neq \{\}$
shows $\exists \sigma'.\ ((P, \sigma), (\{\#\}, \sigma')) \in matchrel^*$
⟨*proof*⟩

lemma *empty-not-reachable-imp-matchers-empty*:
assumes $\bigwedge \sigma'.\ ((P, \sigma), (\{\#\}, \sigma')) \notin matchrel^*$
shows $matchers\text{-}map\ \sigma \cap matchers\ (set\text{-}mset\ P) = \{\}$
⟨*proof*⟩

lemma *irreducible-reachable-imp-matchers-empty*:
assumes $((P, \sigma), y) \in matchrel^!$ **and** $fst\ y \neq \{\#\}$
shows $matchers\text{-}map\ \sigma \cap matchers\ (set\text{-}mset\ P) = \{\}$
⟨*proof*⟩

lemma *matchers-map-not-empty* [*simp*]:
$matchers\text{-}map\ \sigma \neq \{\}$
$\{\} \neq matchers\text{-}map\ \sigma$
⟨*proof*⟩

lemma *matchers-empty-imp-not-empty-NF*:
assumes $matchers\ (set\text{-}mset\ P) = \{\}$
shows $\exists y.\ fst\ y \neq \{\#\} \wedge ((P, Map.empty), y) \in matchrel^!$
⟨*proof*⟩

end

5 Unification

5.1 Unifiers

Definition and properties of (most general) unifiers

theory *Unifiers*
imports *Term*
begin

lemma *map-eq-set-zipD* [*dest*]:
assumes *map f xs* = *map f ys*
and (*x*, *y*) ∈ *set* (*zip xs ys*)
shows *f x* = *f y*
⟨*proof*⟩

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

The set of unifiers for a given set of equations.

definition *unifiers* :: ('*f*, '*v*) *equations* ⇒ ('*f*, '*v*) *subst set*
where
unifiers E = {σ. ∀ *p*∈*E*. *fst p* · σ = *snd p* · σ}

Check whether a set of equations is unifiable.

definition *unifiable E* ⟷ (∃ σ. σ ∈ *unifiers E*)

lemma *in-unifiersE* [*elim*]:
⟦σ ∈ *unifiers E*; (⋀*e*. *e* ∈ *E* ⟹ *fst e* · σ = *snd e* · σ) ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

Applying a substitution to a set of equations.

definition *subst-set* :: ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *equations* ⇒ ('*f*, '*v*) *equations*
where
subst-set σ *E* = (λ*e*. (*fst e* · σ, *snd e* · σ)) ‘ *E*

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

definition *is-mgu* :: ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *equations* ⇒ *bool*
where
is-mgu σ *E* ⟷ σ ∈ *unifiers E* ∧ (∀ τ ∈ *unifiers E*. (∃ γ. τ = σ ∘$_s$ γ))

The following property characterizes idempotent mgus, that is, mgus σ for which σ ∘$_s$ σ = σ holds.

definition *is-imgu* :: ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *equations* ⇒ *bool*
where
is-imgu σ *E* ⟷ σ ∈ *unifiers E* ∧ (∀ τ ∈ *unifiers E*. τ = σ ∘$_s$ τ)

5.1.1 Properties of sets of unifiers

lemma *unifiers-Un* [*simp*]:
unifiers (*s* ∪ *t*) = *unifiers s* ∩ *unifiers t*
⟨*proof*⟩

lemma *unifiers-empty* [*simp*]:
unifiers {} = *UNIV*
⟨*proof*⟩

lemma *unifiers-insert*:
unifiers (*insert p t*) = {σ. *fst p* · σ = *snd p* · σ} ∩ *unifiers t*
⟨*proof*⟩

lemma *unifiers-insert-ident* [*simp*]:
unifiers (*insert* (*t*, *t*) *E*) = *unifiers E*
⟨*proof*⟩

lemma *unifiers-insert-swap*:
unifiers (*insert* (*s*, *t*) *E*) = *unifiers* (*insert* (*t*, *s*) *E*)
⟨*proof*⟩

lemma *unifiers-insert-Var-swap* [*simp*]:
unifiers (*insert* (*t*, *Var x*) *E*) = *unifiers* (*insert* (*Var x*, *t*) *E*)
⟨*proof*⟩

lemma *unifiers-subst-set* [*simp*]:
τ ∈ *unifiers* (*subst-set* σ *E*) ⟷ σ ∘$_s$ τ ∈ *unifiers E*
⟨*proof*⟩

lemma *unifiers-insert-VarD*:
shows σ ∈ *unifiers* (*insert* (*Var x*, *t*) *E*) ⟹ *subst x t* ∘$_s$ σ = σ
and σ ∈ *unifiers* (*insert* (*t*, *Var x*) *E*) ⟹ *subst x t* ∘$_s$ σ = σ
⟨*proof*⟩

lemma *unifiers-insert-Var-left*:
σ ∈ *unifiers* (*insert* (*Var x*, *t*) *E*) ⟹ σ ∈ *unifiers* (*subst-set* (*subst x t*) *E*)
⟨*proof*⟩

lemma *unifiers-set-zip* [*simp*]:
assumes *length ss* = *length ts*
shows *unifiers* (*set* (*zip ss ts*)) = {σ. *map* (λ*t*. *t* · σ) *ss* = *map* (λ*t*. *t* · σ) *ts*}
⟨*proof*⟩

lemma *unifiers-Fun* [*simp*]:
σ ∈ *unifiers* {(*Fun f ss*, *Fun g ts*)} ⟷
length ss = *length ts* ∧ *f* = *g* ∧ σ ∈ *unifiers* (*set* (*zip ss ts*))
⟨*proof*⟩

lemma *unifiers-occur-left-is-Fun*:
fixes *t* :: ('*f*, '*v*) *term*
assumes *x* ∈ *vars-term t* **and** *is-Fun t*
shows *unifiers* (*insert* (*Var x*, *t*) *E*) = {}
⟨*proof*⟩

lemma *unifiers-occur-left-not-Var*:
$x \in$ *vars-term* $t \Longrightarrow t \neq$ *Var* $x \Longrightarrow$ *unifiers* (*insert* (*Var* x, t) E) = {}
⟨*proof*⟩

lemma *unifiers-occur-left-Fun*:
$x \in (\bigcup t{\in}$*set ts*. *vars-term* t) $\Longrightarrow$ *unifiers* (*insert* (*Var* x, *Fun* f *ts*) E) = {}
⟨*proof*⟩

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 ¬ *unifiable* {(*Var* x, t)}
⟨*proof*⟩

lemma *unifiable-insert-swap*:
unifiable (*insert* (s, t) E) = *unifiable* (*insert* (t, s) E)
⟨*proof*⟩

lemma *subst-set-reflects-unifiable*:
fixes σ :: ('f, 'v) *subst*
assumes *unifiable* (*subst-set* σ E)
shows *unifiable* E
⟨*proof*⟩

5.1.3 Properties of *is-mgu*

lemma *is-mgu-empty* [*simp*]:
is-mgu Var {}
⟨*proof*⟩

lemma *is-mgu-insert-trivial* [*simp*]:
is-mgu σ (*insert* (t, t) E) = *is-mgu* σ E
⟨*proof*⟩

lemma *is-mgu-insert-decomp* [*simp*]:
assumes *length ss* = *length ts*
shows *is-mgu* σ (*insert* (*Fun* f *ss*, *Fun* f *ts*) E) ⟷
is-mgu σ ($E \cup$ *set* (*zip ss ts*))
⟨*proof*⟩

lemma *is-mgu-insert-swap*:
is-mgu σ (*insert* (s, t) E) = *is-mgu* σ (*insert* (t, s) E)
⟨*proof*⟩

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

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*⟩

lemma *is-imgu-imp-is-mgu*:
assumes *is-imgu* σ E
shows *is-mgu* σ E
⟨*proof*⟩

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*⟩

corollary *rename-subst-domain-range-preserves-is-imgu-singleton*:
fixes t u :: ('f, 'v) *term* **and** μ ϱ :: ('f, 'v) *subst*
assumes *imgu-μ*: *is-imgu* μ {(t, u)} **and** *is-var-ϱ*: $\forall x$. *is-Var* (ϱ x) **and** *inj* ϱ
shows *is-imgu* (*rename-subst-domain-range* ϱ μ) {($t \cdot \varrho$, $u \cdot \varrho$)}
⟨*proof*⟩

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
⟨*proof*⟩

lemma *union-commutes*:
$M + \{\#x\#\} + N = M + N + \{\#x\#\}$
$M + mset\ xs + N = M + N + mset\ xs$
⟨*proof*⟩

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*:
$(E, E') \in unif \longleftrightarrow (\exists\sigma.\ UNIF1\ \sigma\ E\ E')$
⟨*proof*⟩

lemma *UNIF1-unifD*:
$UNIF1\ \sigma\ E\ E' \Longrightarrow (E, E') \in unif$
⟨*proof*⟩

A termination order for *UNIF1*.

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

lemma *wf-unifless*:
wf unifless
⟨*proof*⟩

lemma *UNIF1-vars-mset-leq*:
assumes $UNIF1\ \sigma\ E\ E'$
shows $vars\text{-}mset\ E' \subseteq vars\text{-}mset\ E$
⟨*proof*⟩

lemma *vars-mset-subset-size-mset-unifflessI* [*intro*]:
$vars\text{-}mset\ M \subseteq vars\text{-}mset\ N \Longrightarrow size\text{-}mset\ M < size\text{-}mset\ N \Longrightarrow (M, N) \in unifless$

⟨*proof*⟩

lemma *vars-mset-psubset-uniflessI* [*intro*]:
vars-mset M ⊂ *vars-mset N* ⟹ (*M*, *N*) ∈ *unifless*
⟨*proof*⟩

lemma *UNIF1-unifless*:
assumes *UNIF1 σ E E′*
shows (*E′*, *E*) ∈ *unifless*
⟨*proof*⟩

lemma *converse-unif-subset-unifless*:
unif^{-1} ⊆ *unifless*
⟨*proof*⟩

5.2.2 Termination of the Inference Rules

lemma *wf-converse-unif*:
wf (*unif*$^{-1}$)
⟨*proof*⟩

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

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

lemma *unif-rtrancl-UNIF-conv*:
(*E*, *E′*) ∈ *unif** ⟷ (∃ *ss*. *UNIF ss E E′*)
⟨*proof*⟩

Compose a list of substitutions.

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

lemma *compose-simps* [*simp*]:
compose [] = *Var*
compose (*Var* # *ss*) = *compose ss*
compose (*σ* # *ss*) = *σ* ∘$_s$ *compose ss*
⟨*proof*⟩

lemma *compose-append* [*simp*]:
compose (*ss* @ *ts*) = *compose ss* ∘$_s$ *compose ts*
⟨*proof*⟩

lemma *set-mset-subst-mset* [*simp*]:
set-mset (*subst-mset* σ *E*) = *subst-set* σ (*set-mset E*)
⟨*proof*⟩

lemma *UNIF1-subst-domain-Int*:
assumes *UNIF1* σ *E E'*
shows *subst-domain* $\sigma \cap$ *vars-mset E'* = {}
⟨*proof*⟩

lemma *UNIF1-subst-domain-subset*:
assumes *UNIF1* σ *E E'*
shows *subst-domain* $\sigma \subseteq$ *vars-mset E*
⟨*proof*⟩

lemma *UNIF-subst-domain-subset*:
assumes *UNIF ss E E'*
shows *subst-domain* (*compose ss*) $\subseteq$ *vars-mset E*
⟨*proof*⟩

lemma *UNIF1-range-vars-subset*:
assumes *UNIF1* σ *E E'*
shows *range-vars* $\sigma \subseteq$ *vars-mset E*
⟨*proof*⟩

lemma *UNIF1-subst-domain-range-vars-Int*:
assumes *UNIF1* σ *E E'*
shows *subst-domain* $\sigma \cap$ *range-vars* σ = {}
⟨*proof*⟩

lemma *UNIF-range-vars-subset*:
assumes *UNIF ss E E'*
shows *range-vars* (*compose ss*) $\subseteq$ *vars-mset E*
⟨*proof*⟩

lemma *UNIF-subst-domain-range-vars-Int*:
assumes *UNIF ss E E'*
shows *subst-domain* (*compose ss*) $\cap$ *range-vars* (*compose ss*) = {}
⟨*proof*⟩

The inference rules generate idempotent substitutions.

lemma *UNIF-idemp*:
assumes *UNIF ss E E'*
shows *compose ss* $\circ_s$ *compose ss* = *compose ss*
⟨*proof*⟩

lemma *UNIF1-mono*:
assumes *UNIF1* σ *E E'*
shows *UNIF1* σ (*E* + *M*) (*E'* + *subst-mset* σ *M*)
⟨*proof*⟩

lemma *unif-mono*:
assumes $(E,\ E') \in unif$
shows $\exists \sigma.\ (E + M,\ E' + subst\text{-}mset\ \sigma\ M) \in unif$
⟨*proof*⟩

lemma *unif-rtrancl-mono*:
assumes $(E,\ E') \in unif^*$
shows $\exists \sigma.\ (E + M,\ E' + subst\text{-}mset\ \sigma\ M) \in unif^*$
⟨*proof*⟩

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 $is\text{-}mgu\ (compose\ ss)\ (set\text{-}mset\ E)$
⟨*proof*⟩

5.2.4 Completeness of the Inference Rules

lemma *UNIF1-singleton-decomp* [*intro*]:
assumes $length\ ss = length\ ts$
shows $UNIF1\ Var\ \{\#(Fun\ f\ ss,\ Fun\ f\ ts)\#\}\ (mset\ (zip\ ss\ ts))$
⟨*proof*⟩

lemma *UNIF1-singleton-Var-left* [*intro*]:
$x \notin vars\text{-}term\ t \Longrightarrow UNIF1\ (subst\ x\ t)\ \{\#(Var\ x,\ t)\#\}\ \{\#\}$
⟨*proof*⟩

lemma *UNIF1-singleton-Var-right* [*intro*]:
$x \notin vars\text{-}term\ t \Longrightarrow UNIF1\ (subst\ x\ t)\ \{\#(t,\ Var\ x)\#\}\ \{\#\}$
⟨*proof*⟩

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

lemma *not-unifD*:
assumes $\neg\ (\exists E'.\ (\{\#e\#\},\ E') \in unif)$
shows $(\exists x\ t.\ (e = (Var\ x,\ t) \vee e = (t,\ Var\ x)) \wedge x \in vars\text{-}term\ t \wedge is\text{-}Fun\ t) \vee$
$(\exists f\ g\ ss\ ts.\ e = (Fun\ f\ ss,\ Fun\ g\ ts) \wedge (f \neq g \vee length\ ss \neq length\ ts))$
⟨*proof*⟩

lemma *unifiable-imp-unif*:
assumes $unifiable\ \{e\}$

shows $\exists\, E'.\ (\{\#e\#\},\ E') \in \mathit{unif}$
⟨*proof*⟩

lemma *unifiable-imp-empty-or-unif*:
assumes *unifiable* (*set-mset E*)
shows $E = \{\#\} \vee (\exists\, E'.\ (E,\ E') \in \mathit{unif})$
⟨*proof*⟩

lemma *UNIF1-preserves-unifiers*:
assumes *UNIF1* σ *E E'* **and** $\tau \in$ *unifiers* (*set-mset E*)
shows $(\sigma \circ_s \tau) \in$ *unifiers* (*set-mset E'*)
⟨*proof*⟩

lemma *unif-preserves-unifiable*:
assumes $(E,\ E') \in \mathit{unif}$ **and** *unifiable* (*set-mset E*)
shows *unifiable* (*set-mset E'*)
⟨*proof*⟩

lemma *unif-imp-converse-unifless* [*dest*]:
$(x,\ y) \in \mathit{unif} \Longrightarrow (y,\ x) \in \mathit{unifless}$
⟨*proof*⟩

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

lemma *unifiable-imp-empty*:
assumes *unifiable* (*set-mset E*)
shows $(E,\ \{\#\}) \in \mathit{unif}^*$
⟨*proof*⟩

lemma *unif-rtrancl-empty-imp-unifiable*:
assumes $(E,\ \{\#\}) \in \mathit{unif}^*$
shows *unifiable* (*set-mset E*)
⟨*proof*⟩

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*⟩

lemma *unif-rtrancl-preserves-unifiable*:
assumes $(E,\ E') \in \mathit{unif}^*$ **and** *unifiable* (*set-mset E*)
shows *unifiable* (*set-mset E'*)
⟨*proof*⟩

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 ¬ *unifiable* (*set-mset E*)
⟨*proof*⟩

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

lemma *irreducible-reachable-imp-not-unifiable*:
assumes (*E*, *E'*) ∈ *unif*$^!$ **and** *E'* ≠ {#}
shows ¬ *unifiable* (*set-mset E*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *decompose-Some* [*dest*]:
decompose (*Fun f ss*) (*Fun g ts*) = *Some E* ⟹
f = *g* ∧ *length ss* = *length ts* ∧ *E* = *zip ss ts*
⟨*proof*⟩

lemma *decompose-None* [*dest*]:
decompose (*Fun f ss*) (*Fun g ts*) = *None* ⟹ *f* ≠ *g* ∨ *length ss* ≠ *length ts*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *subst-list-append*:
subst-list σ (*xs* @ *ys*) = *subst-list* σ *xs* @ *subst-list* σ *ys*
⟨*proof*⟩

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*))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *unify-append-prefix-same*:
(∀ *e* ∈ *set es1*. *fst e* = *snd e*) ⟹ *unify* (*es1* @ *es2*) *bs* = *unify es2 bs*
⟨*proof*⟩

corollary *unify-Cons-same*:
fst e = *snd e* ⟹ *unify* (*e* # *es*) *bs* = *unify es bs*
⟨*proof*⟩

corollary *unify-same*:
(∀ *e* ∈ *set es*. *fst e* = *snd e*) ⟹ *unify es bs* = *Some bs*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *subst-of-append* [*simp*]:
subst-of (*ss* @ *ts*) = *subst-of ts* ∘$_s$ *subst-of ss*
⟨*proof*⟩

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*) {#}
⟨*proof*⟩

lemma *unify-sound*:
assumes *unify E* [] = *Some cs*
shows *is-imgu* (*subst-of cs*) (*set E*)
⟨*proof*⟩

lemma *mgu-sound*:
assumes *mgu s t* = *Some* σ
shows *is-imgu* σ {(*s*, *t*)}
⟨*proof*⟩

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*$^!$
⟨*proof*⟩

lemma *unify-complete*:
assumes *unify E bs* = *None*
shows *unifiers* (*set E*) = {}
⟨*proof*⟩

corollary *ex-unify-if-unifiers-not-empty*:
unifiers es ≠ {} ⟹ *set xs* = *es* ⟹ ∃ *ys*. *unify xs* [] = *Some ys*
⟨*proof*⟩

lemma *mgu-complete*:
mgu s t = *None* ⟹ *unifiers* {(*s*, *t*)} = {}
⟨*proof*⟩

corollary *ex-mgu-if-unifiers-not-empty*:
unifiers {(*t*,*u*)} ≠ {} ⟹ ∃ μ. *mgu t u* = *Some* μ
⟨*proof*⟩

corollary *ex-mgu-if-subst-apply-term-eq-subst-apply-term*:
fixes $t\ u :: ('f, 'v)\ Term.term$ **and** $\sigma :: ('f, 'v)\ subst$
assumes *t-eq-u*: $t \cdot \sigma = u \cdot \sigma$
shows $\exists \mu :: ('f, 'v)\ subst.\ Unification.mgu\ t\ u = Some\ \mu$
⟨*proof*⟩

lemma *finite-subst-domain-subst-of*:
finite (*subst-domain* (*subst-of xs*))
⟨*proof*⟩

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*⟩

lemma *mgu-subst-domain*:
assumes *mgu s t* = *Some* σ
shows *subst-domain* σ ⊆ *vars-term s* ∪ *vars-term t*
⟨*proof*⟩

lemma *mgu-finite-subst-domain*:
mgu s t = *Some* σ ⟹ *finite* (*subst-domain* σ)
⟨*proof*⟩

lemma *unify-range-vars*:
assumes *unif*: *unify E* [] = *Some xs*
shows *range-vars* (*subst-of xs*) ⊆ (⋃ *e* ∈ *set E*. *vars-term* (*fst e*) ∪ *vars-term* (*snd e*))
⟨*proof*⟩

lemma *mgu-range-vars*:
assumes *mgu s t* = *Some* μ
shows *range-vars* μ ⊆ *vars-term s* ∪ *vars-term t*
⟨*proof*⟩

lemma *unify-subst-domain-range-vars-disjoint*:
assumes *unif*: *unify E* [] = *Some xs*
shows *subst-domain* (*subst-of xs*) ∩ *range-vars* (*subst-of xs*) = {}
⟨*proof*⟩

lemma *mgu-subst-domain-range-vars-disjoint*:
assumes *mgu s t* = *Some* μ
shows *subst-domain* μ ∩ *range-vars* μ = {}
⟨*proof*⟩

corollary *subst-apply-term-eq-subst-apply-term-if-mgu*:
assumes *mgu-t-u*: *mgu t u* = *Some* μ

shows $t \cdot \mu = u \cdot \mu$
⟨*proof*⟩

lemma *mgu-same*: *mgu t t* = *Some Var*
⟨*proof*⟩

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: $x \notin (\bigcup e{\in}E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$
shows *is-Var* (*μ x*)
⟨*proof*⟩

corollary *mgu-ball-is-Var*:
is-mgu μ E $\Longrightarrow \forall x \in -\ (\bigcup e{\in}E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e)).$ *is-Var* (*μ x*)
⟨*proof*⟩

lemma *mgu-inj-on*:
fixes μ :: ('*f*, '*v*) *subst* **and** *E* :: ('*f*, '*v*) *equations*
assumes *mgu-μ*: *is-mgu μ E*
shows *inj-on μ* $(-\ (\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e)))$
⟨*proof*⟩

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* ≡ $(\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$
shows *subst-domain μ* ⊆ *Evars*
⟨*proof*⟩

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* ≡ $(\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$
shows $(\bigcup x \in Evars.\ vars\text{-}term\ (\mu\ x)) \subseteq Evars$
⟨*proof*⟩

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* ⊆ $(\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$
⟨*proof*⟩

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

lemma *the-mgu*:
fixes $\sigma :: ('f, 'v)\ subst$
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*⟩

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 ⇒ *None*
| *Some* γ ⇒ *Some* $(\gamma \circ vu, \gamma \circ wu)$)

lemma *mgu-var-disjoint-generic-sound*:
assumes *unif*: *mgu-var-disjoint-generic vu wu s t* = *Some* $(\gamma 1, \gamma 2)$
shows $s \cdot \gamma 1 = t \cdot \gamma 2$
⟨*proof*⟩

lemma *mgu-var-disjoint-generic-complete*:
fixes $\sigma :: ('f, 'v, 'u)\ gsubst$ **and** $\tau :: ('f, 'w, 'u)\ gsubst$
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*⟩

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

lemma *mgu-var-disjoint-sum-sound*:
mgu-var-disjoint-sum s t = *Some* $(\gamma 1, \gamma 2) \Longrightarrow s \cdot \gamma 1 = t \cdot \gamma 2$
⟨*proof*⟩

lemma *mgu-var-disjoint-sum-complete*:
fixes σ :: ('*f*, '*v*, '*v* + '*w*) *gsubst* **and** τ :: ('*f*, '*w*, '*v* + '*w*) *gsubst*
assumes *unif-disj*: $s \cdot \sigma = t \cdot \tau$
shows $\exists \mu 1\ \mu 2\ \delta.$ *mgu-var-disjoint-sum 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*⟩

lemma *mgu-var-disjoint-sum-instance*:
fixes σ :: ('*f*, '*v*) *subst* **and** δ :: ('*f*, '*v*) *subst*
assumes *unif-disj*: $s \cdot \sigma = t \cdot \delta$
shows $\exists \mu 1\ \mu 2\ \tau.$ *mgu-var-disjoint-sum s t* = *Some* ($\mu 1$, $\mu 2$) $\wedge$
$\sigma = \mu 1 \circ_s \tau \wedge$
$\delta = \mu 2 \circ_s \tau \wedge$
$s \cdot \mu 1 = t \cdot \mu 2$
⟨*proof*⟩

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* ($\gamma 1$, $\gamma 2$) ⟹ $s \cdot \gamma 1 = t \cdot \gamma 2$
⟨*proof*⟩

lemma *mgu-vd-complete*:
fixes σ :: ('*f*, '*v* :: *infinite*) *subst* **and** τ :: ('*f*, '*v*) *subst*
assumes *unif-disj*: $s \cdot \sigma = t \cdot \tau$
shows $\exists \mu 1\ \mu 2\ \delta.$ *mgu-vd r s t* = *Some* ($\mu 1$, $\mu 2$) $\wedge$
$\sigma = \mu 1 \circ_s \delta \wedge$
$\tau = \mu 2 \circ_s \delta \wedge$
$s \cdot \mu 1 = t \cdot \mu 2$
⟨*proof*⟩

end

6 Matching

theory *Matching*
imports
Abstract-Matching
Unification
begin

function *match-term-list*

where
match-term-list [] σ = *Some* σ |
match-term-list ((*Var* x, t) # P) σ =
(*if* σ x = *None* $\vee$ σ x = *Some* t *then match-term-list* P (σ ($x \mapsto t$))
else None) |
match-term-list ((*Fun* f *ss*, *Fun* g *ts*) # P) σ =
(*case decompose* (*Fun* f *ss*) (*Fun* g *ts*) *of*
None ⇒ *None*
| *Some us* ⇒ *match-term-list* (*us* @ P) σ) |
match-term-list ((*Fun* f *ss*, *Var* x) # P) σ = *None*
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *match-term-list-Some-matchrel*:
assumes *match-term-list* P σ = *Some* τ
shows ((*mset* P, σ), ({#}, τ)) ∈ *matchrel**
⟨*proof*⟩

lemma *match-term-list-None*:
assumes *match-term-list* P σ = *None*
shows *matchers-map* σ ∩ *matchers* (*set* P) = {}
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *match-list-matches*:
assumes *match-list* d P = *Some* σ
shows $\bigwedge p$ t. (p, t) ∈ *set* P ⟹ $p \cdot \sigma = t$
⟨*proof*⟩

lemma *match-list-complete*:
assumes *match-list* d P = *None*
shows *matchers* (*set* P) = {}
⟨*proof*⟩

lemma *match-list-None-conv*:
match-list d P = *None* ⟷ *matchers* (*set* P) = {}

⟨*proof*⟩

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

lemma *match-sound*:
assumes *match t p* = *Some σ*
shows *σ* ∈ *matchers* {(*p*, *t*)}
⟨*proof*⟩

lemma *match-matches*:
assumes *match t p* = *Some σ*
shows *p* · *σ* = *t*
⟨*proof*⟩

lemma *match-complete*:
assumes *match t p* = *None*
shows *matchers* {(*p*, *t*)} = {}
⟨*proof*⟩

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

lemma *matches-iff*:
matches t p ⟷ (∃ *σ*. *p* · *σ* = *t*)
⟨*proof*⟩

lemma *match-complete'*:
assumes *p* · *σ* = *t*
shows ∃ *τ*. *match t p* = *Some τ* ∧ (∀ *x*∈*vars-term p*. *σ x* = *τ x*)
⟨*proof*⟩

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*⟩

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*''))
⟨*proof*⟩

lemma *mgu-vd-string-sound*:
mgu-vd-string s t = *Some* (*γ1*, *γ2*) ⟹ *s* · *γ1* = *t* · *γ2*
⟨*proof*⟩

lemma *mgu-vd-string-complete*:
fixes *σ* :: ('*f*, *string*) *subst* **and** *τ* :: ('*f*, *string*) *subst*
assumes *s* · *σ* = *t* · *τ*
shows ∃ *μ1 μ2 δ*. *mgu-vd-string s t* = *Some* (*μ1*, *μ2*) ∧
σ = *μ1* ∘$_s$ *δ* ∧
τ = *μ2* ∘$_s$ *δ* ∧
s · *μ1* = *t* · *μ2*
⟨*proof*⟩
end

7 Subsumption

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

theory *Subsumption*
imports
Term
Abstract−Rewriting.Seq
Fun-More
Seq-More
begin

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

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

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

abbreviation *INSTANCEEQ-SET* (‹{·≥}›)
where

$\{\cdot\geq\} \equiv \{(x,\ y).\ y \leq\cdot\ x\}$

abbreviation *INSTANCE-SET* $(\langle\{\cdot>\}\rangle)$
where
$\{\cdot>\} \equiv \{(x,\ y).\ y <\cdot\ x\}$

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

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

abbreviation *LITSIM-SET* $(\langle\{\doteq\}\rangle)$
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 $\rightleftharpoons$ *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 $\rightleftharpoons$ *subsumes* **and**
LITSIM $\rightleftharpoons$ *litsim*

lemma *litsim-refl* [*simp*]:
$s \doteq s$
⟨*proof*⟩

lemma *litsim-sym*:
$s \doteq t \Longrightarrow t \doteq s$
⟨*proof*⟩

lemma *litsim-trans*:
$s \doteq t \Longrightarrow t \doteq u \Longrightarrow s \doteq u$
⟨*proof*⟩

end

sublocale *subsumable* $\subseteq$ *subsumption*: *preorder* $(\leq\cdot)$ $(<\cdot)$
⟨*proof*⟩

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 $\rightleftharpoons$ *subsumeseq-term*

lemma *subsumeseq-termE* [*elim*]:
assumes $s \leq\cdot t$
obtains σ **where** $t = s \cdot \sigma$
⟨*proof*⟩

lemma *subsumeseq-term-refl*:
fixes t :: $('a, 'b)$ *term*
shows $t \leq\cdot t$
⟨*proof*⟩

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*⟩

interpretation *term-subsumable*: *subsumable subsumeseq-term*
⟨*proof*⟩

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

lemma *subsumeseq-term-iff*:
$s \cdot\geq t \longleftrightarrow (\exists \sigma.\ s = t \cdot \sigma)$
⟨*proof*⟩

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 ≥ 1*
⟨*proof*⟩

lemma *num-syms-subst*:
num-syms (t · σ) ≥ num-syms t
⟨*proof*⟩

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*; ∀ *i < length ts. emv (ss ! i) (ts ! i)*⟧
⟹
emv (Fun f ss) (Fun g ts)

lemma *sum-list-map-num-syms-subst*:
assumes *sum-list (map (num-syms ∘ (λt. t · σ)) ts) = sum-list (map num-syms ts)*
shows ∀ *i < length ts. num-syms (ts ! i · σ) = num-syms (ts ! i)*
⟨*proof*⟩

lemma *subst-size-emv*:
assumes *s = t · τ* **and** *num-syms s = num-syms t* **and** *num-funs s = num-funs t*
shows *emv s t*
⟨*proof*⟩

lemma *subsumeseq-term-size-emv*:
assumes *s ·≥ t* **and** *num-syms s = num-syms t* **and** *num-funs s = num-funs t*
shows *emv s t*
⟨*proof*⟩

lemma *emv-subst-vars-term*:
assumes *emv s t*
and *s = t · σ*
shows *vars-term s = (the-Var ∘ σ) ‘ vars-term t*
⟨*proof*⟩

lemma *emv-subst-imp-num-unique-vars-le*:
assumes *emv s t*
and *s = t · σ*
shows *num-unique-vars s ≤ num-unique-vars t*
⟨*proof*⟩

lemma *emv-subsumeseq-term-imp-num-unique-vars-le*:
assumes *emv s t*
and *s ·≥ t*

shows *num-unique-vars s* ≤ *num-unique-vars t*
⟨*proof*⟩

lemma *num-syms-geq-num-vars*:
num-syms t ≥ *num-vars t*
⟨*proof*⟩

lemma *num-unique-vars-Fun-Cons*:
num-unique-vars (*Fun f* (*t* # *ts*)) ≤ *num-unique-vars t* + *num-unique-vars* (*Fun f ts*)
⟨*proof*⟩

lemma *sum-list-map-unique-vars*:
sum-list (*map num-unique-vars ts*) ≥ *num-unique-vars* (*Fun f ts*)
⟨*proof*⟩

lemma *num-unique-vars-Var-1* [*simp*]:
num-unique-vars (*Var x*) = *1*
⟨*proof*⟩

lemma *num-vars-geq-num-unique-vars*:
num-vars t ≥ *num-unique-vars t*
⟨*proof*⟩

lemma *num-syms-ge-num-unique-vars*:
num-syms t ≥ *num-unique-vars t*
⟨*proof*⟩

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*⟩

lemma *emv-subst-imp-is-Var*:
assumes *emv s t*
and *s* = *t* · *σ*
shows ∀ *x* ∈ *vars-term t*. *is-Var* (*σ x*)
⟨*proof*⟩

lemma *bij-Var-subst-compose-Var*:
assumes *bij g*
shows (*Var* ∘ *g*) ∘$_s$ (*Var* ∘ *inv g*) = *Var*
⟨*proof*⟩

7.2 Well-foundedness

lemma *wf-subsumes*:
wf ({<·} :: (*'f*, *'v*) *term rel*)

⟨*proof*⟩

end

8 Subterms and Contexts

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

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

8.1 Subterms

The *superterm* relation.

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

The *proper superterm* relation.

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

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

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

8.1.1 Syntactic Sugar

Infix syntax.

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

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

notation
supt (‹{▷}›) **and**
supt-pred (‹(-/ ▷ -)› [*56*, *56*] *55*) **and**
subt-pred (**infix** ‹◁› *55*) **and**
supteq (‹{▷=}›) **and**
supteq-pred (‹(-/ ▷= -)› [*56*, *56*] *55*) **and**
subteq-pred (**infix** ‹◁=› *55*)

abbreviation *subt* (‹{◁}›) **where** {◁} ≡ {▷}$^{-1}$
abbreviation *subteq* (‹{◁=}›) **where** {◁=} ≡ {▷=}$^{-1}$

Quantifier syntax.

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

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

syntax-consts
-All-supteq -All-supt -All-subteq -All-subt ⇌ *All* **and**
-Ex-supteq -Ex-supt -Ex-subteq -Ex-subt ⇌ *Ex*

translations
∀ *x*▷=*y*. *P* ⇀ ∀ *x*. *x* ▷= *y* ⟶ *P*
∃ *x*▷=*y*. *P* ⇀ ∃ *x*. *x* ▷= *y* ∧ *P*
∀ *x*▷*y*. *P* ⇀ ∀ *x*. *x* ▷ *y* ⟶ *P*
∃ *x*▷*y*. *P* ⇀ ∃ *x*. *x* ▷ *y* ∧ *P*

∀ *x*◁=*y*. *P* ⇀ ∀ *x*. *x* ◁= *y* ⟶ *P*
∃ *x*◁=*y*. *P* ⇀ ∃ *x*. *x* ◁= *y* ∧ *P*
∀ *x*◁*y*. *P* ⇀ ∀ *x*. *x* ◁ *y* ⟶ *P*
∃ *x*◁*y*. *P* ⇀ ∃ *x*. *x* ◁ *y* ∧ *P*

⟨*ML*⟩

8.1.2 Transitivity Reasoning for Subterms

lemma *supt-trans* [*trans*]:
s ▷ *t* ⟹ *t* ▷ *u* ⟹ *s* ▷ *u*

⟨*proof*⟩

lemma *trans-supt*: *trans* {⊳} ⟨*proof*⟩

lemma *supteq-trans* [*trans*]:
$s \trianglerighteq t \Longrightarrow t \trianglerighteq u \Longrightarrow s \trianglerighteq u$
⟨*proof*⟩

Auxiliary lemmas about term size.

lemma *size-simp5*:
$s \in set\ ss \Longrightarrow s \rhd t \Longrightarrow size\ t < size\ s \Longrightarrow size\ t < Suc\ (size\text{-}list\ size\ ss)$
⟨*proof*⟩

lemma *size-simp6*:
$s \in set\ ss \Longrightarrow s \trianglerighteq t \Longrightarrow size\ t \leq size\ s \Longrightarrow size\ t \leq Suc\ (size\text{-}list\ size\ ss)$
⟨*proof*⟩

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

lemma *size-simp2*:
$t \in set\ ts \Longrightarrow size\ t < Suc\ (Suc\ (size\ s + size\text{-}list\ size\ ts))$
⟨*proof*⟩

lemma *size-simp3*:
assumes $(x, y) \in set\ (zip\ xs\ ys)$
shows $size\ x < Suc\ (size\text{-}list\ size\ xs)$
⟨*proof*⟩

lemma *size-simp4*:
assumes $(x, y) \in set\ (zip\ xs\ ys)$
shows $size\ y < Suc\ (size\text{-}list\ size\ ys)$
⟨*proof*⟩

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

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

lemma *supt-size*:
$s \rhd t \Longrightarrow size\ s > size\ t$
⟨*proof*⟩

lemma *supteq-size*:
$s \trianglerighteq t \Longrightarrow size\ s \geq size\ t$
⟨*proof*⟩

Reflexivity and Asymmetry.

lemma *reflcl-supteq* [*simp*]:
  $supteq^{=} = supteq$ ⟨*proof*⟩

lemma *trancl-supteq* [*simp*]:
  $supteq^{+} = supteq$
  ⟨*proof*⟩

lemma *rtrancl-supteq* [*simp*]:
  $supteq^{*} = supteq$
  ⟨*proof*⟩

lemma *eq-supteq*: $s = t \Longrightarrow s \trianglerighteq t$ ⟨*proof*⟩

lemma *supt-neqD*: $s \rhd t \Longrightarrow s \neq t$ ⟨*proof*⟩

lemma *supteq-Var* [*simp*]:
  $x \in vars\text{-}term\ t \Longrightarrow t \trianglerighteq Var\ x$
⟨*proof*⟩

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

lemma *term-not-arg* [*iff*]:
  $Fun\ f\ ss \notin set\ ss$ (**is** $?t \notin set\ ss$)
⟨*proof*⟩

lemma *supt-Fun* [*simp*]:
  assumes $s \rhd Fun\ f\ ss$ (**is** $s \rhd ?t$) **and** $s \in set\ ss$
  shows *False*
⟨*proof*⟩

lemma *supt-supteq-conv*: $s \rhd t = (s \trianglerighteq t \wedge s \neq t)$
⟨*proof*⟩

lemma *supt-not-sym*: $s \rhd t \Longrightarrow \neg\ (t \rhd s)$
⟨*proof*⟩

lemma *supt-irrefl*[*iff*]: $\neg\ t \rhd t$
  ⟨*proof*⟩

lemma *irrefl-subt*: *irrefl* $\{\lhd\}$ ⟨*proof*⟩

lemma *supt-imp-supteq*: $s \rhd t \Longrightarrow s \trianglerighteq t$
  ⟨*proof*⟩

lemma *supt-supteq-not-supteq*: $s \rhd t = (s \trianglerighteq t \wedge \neg\ (t \trianglerighteq s))$
  ⟨*proof*⟩

lemma *supteq-supt-conv*: $(s \trianglerighteq t) = (s \rhd t \vee s = t)$ ⟨*proof*⟩

lemma *supteq-antisym*:
assumes $s \trianglerighteq t$ **and** $t \trianglerighteq s$ **shows** $s = t$
⟨*proof*⟩

The subterm relation is an order on terms.

interpretation *subterm*: *order* $(\trianglelefteq)$ $(\triangleleft)$
⟨*proof*⟩

More transitivity rules.

lemma *supt-supteq-trans*[*trans*]:
$s \triangleright t \Longrightarrow t \trianglerighteq u \Longrightarrow s \triangleright u$
⟨*proof*⟩

lemma *supteq-supt-trans*[*trans*]:
$s \trianglerighteq t \Longrightarrow t \triangleright u \Longrightarrow s \triangleright u$
⟨*proof*⟩

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

lemma *suptE* [*elim*]: $s \triangleright t \Longrightarrow (s \trianglerighteq t \Longrightarrow P) \Longrightarrow (s \neq t \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

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

lemma *supt-supteq-set-conv*:
$\{\triangleright\} = \{\trianglerighteq\} - Id$
⟨*proof*⟩

lemma *supteq-supt-set-conv*:
$\{\trianglerighteq\} = \{\triangleright\}^=$
⟨*proof*⟩

lemma *supteq-imp-vars-term-subset*:
$s \trianglerighteq t \Longrightarrow$ *vars-term* $t \subseteq$ *vars-term* s
⟨*proof*⟩

lemma *set-supteq-into-supt* [*simp*]:
assumes $t \in$ *set ts* **and** $t \trianglerighteq s$
shows *Fun f ts* $\triangleright s$
⟨*proof*⟩

The superterm relation is strongly normalizing.

lemma *SN-supt*:
SN $\{\triangleright\}$
⟨*proof*⟩

lemma *supt-not-refl*[*elim*!]:

assumes $t \rhd t$ **shows** *False*
⟨*proof*⟩

lemma *supteq-not-supt* [*elim*]:
assumes $s \trianglerighteq t$ **and** $\neg\ (s \rhd t)$
shows $s = t$
⟨*proof*⟩

lemma *supteq-not-supt-conv* [*simp*]:
$\{\trianglerighteq\} - \{\rhd\} = Id$ ⟨*proof*⟩

lemma *supteq-subst* [*simp*, *intro*]:
assumes $s \trianglerighteq t$ **shows** $s \cdot \sigma \trianglerighteq t \cdot \sigma$
⟨*proof*⟩

lemma *supt-subst* [*simp*, *intro*]:
assumes $s \rhd t$ **shows** $s \cdot \sigma \rhd t \cdot \sigma$
⟨*proof*⟩

lemma *subterm-induct*:
assumes $\bigwedge t.\ \forall s \lhd t.\ P\ s \Longrightarrow P\ t$
shows [*case-names subterm*]: $P\ t$
⟨*proof*⟩

8.2 Contexts

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

We generalize contexts to *abstract contexts* so that arguments can be arbitrary elements.

datatype $('f,'a)$ *actxt* =
Hole (‹□›) | *More* $'f$ $'a$ *list* $('f,'a)$ *actxt* $'a$ *list*

declare *actxt.map-ident*[*simp*]

type-synonym $('f,'v)$ *ctxt* = $('f,('f,'v)$ *term*$)$ *actxt*

fun *map-ctxt* **where**
map-ctxt f v □ = □
| *map-ctxt f v* (*More g ls C rs*) =
More (*f g*) (*map* (*map-term f v*) *ls*) (*map-ctxt f v C*) (*map* (*map-term f v*) *rs*)

fun *vars-ctxt* **where**
vars-ctxt □ = {}
| *vars-ctxt* (*More f ls C rs*) =
$\bigcup$(*vars-term* ‘ *set ls*) ∪ *vars-ctxt C* ∪ $\bigcup$(*vars-term* ‘ *set rs*)

fun *funs-ctxt* **where**

funs-ctxt $\square = \{\}$
| *funs-ctxt* (*More f ls C rs*) =
insert f ($\bigcup$(*funs-term* ‘ *set ls*) $\cup$ *funs-ctxt C* $\cup$ $\bigcup$(*funs-term* ‘ *set rs*))

Interpretation of abstract context.

primrec *intp-actxt* (‹(*1*-⟨-;/-⟩)›[*999*,*0*,*0*]*100*) **where**
$I\langle Hole;a\rangle = a$
| $I\langle More\ f\ ls\ C\ rs;a\rangle = I\ f\ (ls\ @\ I\langle C;a\rangle\ \#\ rs)$

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

abbreviation *ctxt-apply-term* (‹-⟨-⟩› [*1000*, *0*] *1000*) **where**
$C\langle s\rangle \equiv Fun\langle C;s\rangle$

primrec *actxt-compose* (**infixl** ‹∘$_c$› *75*) **where**
Hole $\circ_c$ *D* = *D*
| *More f ls C rs* $\circ_c$ *D* = *More f ls* (*C* $\circ_c$ *D*) *rs*

lemma *intp-actxt-compose*: $I\langle C \circ_c D;a\rangle = I\langle C;I\langle D;a\rangle\rangle$
⟨*proof*⟩

thm *intp-actxt-compose*[*of Fun*]
abbreviation *map-args-actxt* ≡ *map-actxt* (λ*x*. *x*)

abbreviation *eval-ctxt* (‹(*1*-⟦-⟧$_c$/ -)›[*999*,*1*,*100*]*100*) **where**
$I[\![C]\!]_c\ \alpha \equiv$ *map-args-actxt* (λ*t*. $I[\![t]\!]\alpha$) *C*

lemma *eval-ctxt-simps*:
$I[\![\square]\!]_c\ \alpha = \square$
$I[\![More\ f\ ls\ C\ rs]\!]_c\ \alpha = More\ f\ [I[\![l]\!]\alpha.\ l \leftarrow ls]\ (I[\![C]\!]_c\ \alpha)\ [I[\![r]\!]\alpha.\ r \leftarrow rs]$
⟨*proof*⟩

lemma *eval-ctxt*: $I[\![C\langle s\rangle]\!]\alpha = I\langle I[\![C]\!]_c\ \alpha;\ I[\![s]\!]\alpha\rangle$
⟨*proof*⟩

Applying substitutions to contexts.

abbreviation *subst-apply-actxt* (**infixl** ‹·$_c$› *67*) **where**
C ·$_c$ ϑ ≡ *map-args-actxt* (λ*t*. *t* · ϑ) *C*

lemma *apply-ctxt-Var*[*simp*]: *C* ·$_c$ *Var* = *C*
⟨*proof*⟩

lemma *eval-subst-ctxt*: $I[\![C \cdot_c \vartheta]\!]_c\ \varrho = I[\![C]\!]_c\ I[\![\vartheta]\!]_s\ \varrho$
⟨*proof*⟩

lemmas *ctxt-subst-subst*[*simp*] = *eval-subst-ctxt*[*of Fun*]

lemma *ctxt-eq* [*simp*]:
($C\langle s\rangle$ = $C\langle t\rangle$) = (*s* = *t*) ⟨*proof*⟩

lemma *size-ctxt*: $size\ t \leq size\ (C\langle t\rangle)$
⟨*proof*⟩

lemma *size-ne-ctxt*: $C \neq \Box \Longrightarrow size\ t < size\ (C\langle t\rangle)$
⟨*proof*⟩

interpretation *ctxt-monoid-mult*: *monoid-mult* $\Box\ (\circ_c)$
⟨*proof*⟩

instantiation *actxt* :: (*type*, *type*) *monoid-mult*
begin
definition [*simp*]: $1 = \Box$
definition [*simp*]: $(*) = (\circ_c)$
instance ⟨*proof*⟩
end

lemmas *ctxt-ctxt-compose*[*simp*] = *intp-actxt-compose*[*of Fun*]

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

lemmas *subst-apply-term-ctxt-apply-distrib* [*simp*] = *eval-ctxt*[*of Fun*]

lemma *eval-ctxt-compose-distrib*:
$I[\![C \circ_c D]\!]_c\ \sigma = (I[\![C]\!]_c\ \sigma) \circ_c (I[\![D]\!]_c\ \sigma)$
⟨*proof*⟩

lemmas *subst-compose-ctxt-compose-distrib* [*simp*] =
eval-ctxt-compose-distrib[*of Fun*]

lemma *eval-ctxt-eval-subst*:
$I[\![C]\!]_c\ (I[\![\sigma]\!]_s\ \tau) = I[\![C \cdot_c \sigma]\!]_c\ \tau$
⟨*proof*⟩

lemmas *ctxt-compose-subst-compose-distrib* [*simp*] =
eval-ctxt-eval-subst[*of Fun*]

8.3 The Connection between Contexts and the Superterm Relation

lemma *ctxt-imp-supteq* [*simp*]:
shows $C\langle t\rangle \trianglerighteq t$ ⟨*proof*⟩

lemma *supteq-ctxtE*[*elim*]:
assumes $s \trianglerighteq t$ **obtains** C **where** $s = C\langle t\rangle$
⟨*proof*⟩

lemma *ctxt-supteq*[*intro*]:
assumes $s = C\langle t\rangle$ **shows** $s \trianglerighteq t$

⟨*proof*⟩

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

lemma *supt-ctxtE*[*elim*]:
assumes $s \triangleright t$ **obtains** C **where** $C \neq \Box$ **and** $s = C\langle t\rangle$
⟨*proof*⟩

lemma *ctxt-supt*[*intro 2*]:
assumes $C \neq \Box$ **and** $s = C\langle t\rangle$ **shows** $s \triangleright t$
⟨*proof*⟩

lemma *supt-ctxt-conv*: $(s \triangleright t) = (\exists\, C.\ C \neq \Box \wedge s = C\langle t\rangle)$ (**is** - = *?rhs*)
⟨*proof*⟩

lemma *nectxt-imp-supt-ctxt*: $C \neq \Box \Longrightarrow C\langle t\rangle \triangleright t$ ⟨*proof*⟩

lemma *supt-var*: $\neg\ (\mathit{Var}\ x \triangleright u)$
⟨*proof*⟩

lemma *supt-const*: $\neg\ (\mathit{Fun}\ f\ [] \triangleright u)$
⟨*proof*⟩

lemma *supteq-var-imp-eq*:
$(\mathit{Var}\ x \trianglerighteq t) = (t = \mathit{Var}\ x)$ (**is** - = (- = *?x*))
⟨*proof*⟩

lemma *Var-supt* [*elim*!]:
assumes $\mathit{Var}\ x \triangleright t$ **shows** P
⟨*proof*⟩

lemma *Fun-supt* [*elim*]:
assumes $\mathit{Fun}\ f\ ts \triangleright s$ **obtains** t **where** $t \in \mathit{set}\ ts$ **and** $t \trianglerighteq s$
⟨*proof*⟩

lemma *inj-ctxt-apply-term*: *inj* (*ctxt-apply-term* C)
⟨*proof*⟩

lemma *ctxt-subst-eq*: $(\bigwedge x.\ x \in \mathit{vars\text{-}ctxt}\ C \Longrightarrow \sigma\ x = \tau\ x) \Longrightarrow C \cdot_c \sigma = C \cdot_c \tau$
⟨*proof*⟩

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 $'f\ \mathit{sig} = ('f \times \mathit{nat})\ \mathit{set}$

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

fun *funas-term* :: $('f, 'v)\ \mathit{term} \Rightarrow 'f\ \mathit{sig}$
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*)
⟨*proof*⟩

lemma *supt-imp-funas-term-subset*:
assumes *s* ⊳ *t*
shows *funas-term t* ⊆ *funas-term s*
⟨*proof*⟩

lemma *supteq-imp-funas-term-subset*[*simp*]:
assumes *s* ⊵ *t*
shows *funas-term t* ⊆ *funas-term s*
⟨*proof*⟩

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

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

lemma *funas-term-ctxt-apply* [*simp*]:
funas-term (*C*⟨*t*⟩) = *funas-ctxt C* ∪ *funas-term t*
⟨*proof*⟩

lemma *funas-term-subst*:
funas-term (*t* · *σ*) = *funas-term t* ∪ ⋃(*funas-term* ‘ *σ* ‘ *vars-term t*)
⟨*proof*⟩

lemma *funas-ctxt-compose* [*simp*]:
funas-ctxt (*C* ∘*c D*) = *funas-ctxt C* ∪ *funas-ctxt D*
⟨*proof*⟩

lemma *funas-ctxt-subst* [*simp*]:
funas-ctxt (*C* ·*c σ*) = *funas-ctxt C* ∪ ⋃(*funas-term* ‘ *σ* ‘ *vars-ctxt C*)
⟨*proof*⟩

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
HOL−Library.Sublist
Show.Shows-Literal
begin

type-synonym *pos* = *nat list*

definition *less-eq-pos* :: *pos* ⇒ *pos* ⇒ *bool* (**infix** ‹≤$_p$› *50*) **where**
$p \leq_p q \longleftrightarrow (\exists r.\ p \mathbin{@} r = q)$

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

lemma *less-eq-pos-eq-prefix*:
less-eq-pos = *Sublist.prefix*
⟨*proof*⟩

lemma *less-pos-eq-strict-prefix*:
less-pos = *Sublist.strict-prefix*
⟨*proof*⟩

interpretation *order-pos*: *order less-eq-pos less-pos*
⟨*proof*⟩

lemma *Nil-least* [*intro*!, *simp*]:
$[] \leq_p p$
⟨*proof*⟩

lemma *less-eq-pos-simps* [*simp*]:
$p \leq_p p \mathbin{@} q$
$p \mathbin{@} q1 \leq_p p \mathbin{@} q2 \longleftrightarrow q1 \leq_p q2$
$i \# q1 \leq_p [] \longleftrightarrow False$
$i \# q1 \leq_p j \# q2 \longleftrightarrow i = j \wedge q1 \leq_p q2$
$p \mathbin{@} q \leq_p p \longleftrightarrow q = []$
$p \leq_p [] \longleftrightarrow p = []$
⟨*proof*⟩

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

lemma *less-pos-simps*[*simp*]:
$(p <_p p \mathbin{@} q) = (q \neq [])$
$(p \mathbin{@} q1 <_p p \mathbin{@} q2) = (q1 <_p q2)$

$(p <_p []) = False$
$(i \# q1 <_p j \# q2) = (i = j \wedge q1 <_p q2)$
$(p @ q <_p p) = False$
⟨*proof*⟩

lemma *prefix-smaller* [*simp*]:
assumes $p <_p q$ **shows** $size\ p < size\ q$
⟨*proof*⟩

instantiation *list* :: (*type*) *one*
begin
definition *one-list-def* [*simp*]: $1 = []$
instance ⟨*proof*⟩
end

instantiation *list* :: (*type*) *times*
begin
definition *times-list-def* [*simp*]: $times\ p\ q = p @ q$
instance ⟨*proof*⟩
end

instantiation *list* :: (*type*) *semigroup-mult*
begin
instance ⟨*proof*⟩
end

instantiation *list* :: (*type*) *power*
begin
instance ⟨*proof*⟩
end

lemma *power-append-distr*:
$p \hat{} (m + n) = p \hat{} m @ p \hat{} n$
⟨*proof*⟩

lemma *power-pos-Suc*: $p \hat{} Suc\ n = p \hat{} n @ p$
⟨*proof*⟩

lemma *power-subtract-less-eq*:
$p \hat{} (n - m) \leq_p p \hat{} n$
⟨*proof*⟩

lemma *power-size*: **fixes** p :: *pos* **shows** $size\ (p \hat{} n) = size\ p * n$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *remove-prefix-Some* [*simp*]:
remove-prefix xs ys = *Some zs* ⟷ *ys* = *xs* @ *zs*
⟨*proof*⟩

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

lemma *less-eq-pos-remove-prefix*:
assumes *p* ≤$_p$ *q*
obtains *r* **where** *q* = *p* @ *r* **and** *remove-prefix p q* = *Some r*
⟨*proof*⟩

lemma *suffix-exists*:
assumes *p* ≤$_p$ *q*
shows ∃ *r*. *p* @ *r* = *q* ∧ *remove-prefix p q* = *Some r*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Nil-power* [*simp*]: [] ^ *n* = [] ⟨*proof*⟩

fun *parallel-pos* :: *pos* ⇒ *pos* ⇒ *bool* (**infixr** ‹⊥› *64*)
where
[] ⊥ - ⟷ *False*
| - ⊥ [] ⟷ *False*
| *i* # *p* ⊥ *j* # *q* ⟷ *i* ≠ *j* ∨ *p* ⊥ *q*

lemma *parallel-pos-eq-parallel*:
parallel-pos = *Sublist.parallel*
⟨*proof*⟩

lemma *parallel-pos*: $p \perp q = (\neg\ p \leq_p q \wedge \neg\ q \leq_p p)$
⟨*proof*⟩

lemma *parallel-remove-prefix*: $p1 \perp p2 \Longrightarrow$
$\exists\ p\ i\ j\ q1\ q2.\ p1 = p\ @\ i\ \#\ q1 \wedge p2 = p\ @\ j\ \#\ q2 \wedge i \neq j$
⟨*proof*⟩

lemma *pos-cases*: $p \leq_p q \vee q <_p p \vee p \perp q$
⟨*proof*⟩

lemma *parallel-pos-sym*: $p1 \perp p2 \Longrightarrow p2 \perp p1$
⟨*proof*⟩

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

lemma *pos-append-cases*:
$p1\ @\ p2 = q1\ @\ q2 \Longrightarrow$
$(\exists\ q3.\ p1 = q1\ @\ q3 \wedge q2 = q3\ @\ p2) \vee$
$(\exists\ p3.\ q1 = p1\ @\ p3 \wedge p2 = p3\ @\ q2)$
⟨*proof*⟩

lemma *pos-less-eq-append-not-parallel*:
assumes $q \leq_p p\ @\ q'$
shows $\neg\ (q \perp p)$
⟨*proof*⟩

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*⟩

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*⟩

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

lemma *proper-prefix-list-append-self-eq-prefixes*:

proper-prefix-list xs @ [xs] = Sublist.prefixes xs
⟨*proof*⟩

lemma *rotate1-prefix-list-eq-prefixes*:
rotate1 (prefix-list xs) = Sublist.prefixes xs
⟨*proof*⟩

lemma *prefix-list* [*simp*]: *set (prefix-list p) = { q. q ≤ₚ p}*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *left-of-pos-append*:
left-of-pos p q ⟹ left-of-pos (p @ p′) (q @ q′)
⟨*proof*⟩

lemma *append-left-of-pos*:
left-of-pos p q = left-of-pos (p′ @ p) (p′ @ q)
⟨*proof*⟩

lemma *left-pos-parallel*: *left-of-pos p q ⟹ q ⊥ p* ⟨*proof*⟩

lemma *left-of-append-cases*: *left-of-pos (p0 @ p1) q ⟹ p0 <ₚ q ∨ left-of-pos p0 q*
⟨*proof*⟩

lemma *append-left-of-cases*:
assumes *left*: *left-of-pos q (p0 @ p1)*
shows *p0 <ₚ q ∨ left-of-pos q p0*
⟨*proof*⟩

lemma *parallel-imp-right-or-left-of*:
assumes *par*:*p ⊥ q* **shows** *left-of-pos p q ∨ left-of-pos q p*
⟨*proof*⟩

lemma *left-of-imp-not-right-of*:
assumes *l*:*left-of-pos p q* **shows** ¬ *left-of-pos q p*
⟨*proof*⟩

primrec *is-left-of* :: *pos* ⇒ *pos* ⇒ *bool*
  where
    left-Nil: *is-left-of* [] *q* = *False*
  | *left-Cons*: *is-left-of* (*i* # *p*) *q* =
      (*case q of*
        [] ⇒ *False*
      | *j* # *q'* ⇒ *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*⟩

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* []
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *pos-diff-Nil2* [*simp*]:
  pos-diff p [] = *p*
  ⟨*proof*⟩

lemma *inj-nat-to-pos*: *inj* (*rec-nat* [] *Cons*) (**is** *inj ?f*)
  ⟨*proof*⟩

lemma *infinite-UNIV-pos*[*simp*]: *infinite* (*UNIV* :: *pos set*)
⟨*proof*⟩

lemma *less-pos-right-mono*:
  p @ *q* <$_p$ *r* @ *q* ⟹ *p* <$_p$ *r*
⟨*proof*⟩

lemma *less-pos-left-mono*:
  p @ *q* <$_p$ *p* @ *r* ⟹ *q* <$_p$ *r*
  ⟨*proof*⟩

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 ⟨*proof*⟩
end

showl-Instance for Contexts

fun *showsl-actxt′* :: (*′f* ⇒ *showsl*) ⇒ (*′a* ⇒ *showsl*) ⇒ (*′f*, *′a*) *actxt* ⇒ *showsl*
where
  showsl-actxt′ fun arg (*Hole*) = *showsl-lit* (*STR ′′□′′*)
| *showsl-actxt′ fun arg* (*More f ss1 D ss2*) = (
    fun f ∘ *showsl* (*STR ′′(′′*) ∘
    showsl-list-gen arg (*STR ′′′′*) (*STR ′′′′*) (*STR ′′, ′′*) (*STR ′′, ′′*) *ss1* ∘
    showsl-actxt′ fun arg D ∘
    showsl-list-gen arg (*STR ′′)′′*) (*STR ′′, ′′*) (*STR ′′, ′′*) (*STR ′′)′′*) *ss2*
)

instantiation *actxt* :: (*showl*,*showl*)*showl*
begin

definition *showsl* (*t* :: ('*a*,'*b*)*actxt*) = *showsl-actxt'* *showsl showsl t*
definition *showsl-list* (*xs* :: ('*a*,'*b*)*actxt list*) = *default-showsl-list showsl xs*
instance ⟨*proof*⟩
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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *elem-size-size-list-size* [*termination-simp*]:
x ∈ set xs ⟹ size x < size-list size xs
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *finite-fun-poss*:
finite (fun-poss t)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *finite-var-poss*:
finite (var-poss t)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *finite-poss* [*simp*]:
finite (poss t)
⟨*proof*⟩

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

fun *subt-at* :: $('f, 'v)$ *term* $\Rightarrow$ *pos* $\Rightarrow$ $('f, 'v)$ *term* (**infixl** $\langle|'\rangle$ *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*)
⟨*proof*⟩

lemma *fun-poss-fun-conv*:
assumes *p* ∈ *fun-poss t*
shows ∃ *f ts*. *t* |- *p* = *Fun f ts*
⟨*proof*⟩

lemma *pos-append-poss*:
p ∈ *poss t* ⟹ *q* ∈ *poss* (*t* |- *p*) ⟹ *p* @ *q* ∈ *poss t*
⟨*proof*⟩

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

fun
ctxt-of-pos-term :: *pos* $\Rightarrow$ $('f, 'v)$ *term* $\Rightarrow$ $('f, 'v)$ *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*
⟨*proof*⟩

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* $\Rightarrow$ *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*

⟨*proof*⟩

lemma *hole-pos-id-ctxt*:
assumes $C\langle s\rangle = t$
shows *ctxt-of-pos-term* (*hole-pos* C) $t = C$
⟨*proof*⟩

lemma *supteq-imp-subt-at*:
assumes $s \trianglerighteq t$
shows $\exists\, p{\in}\mathit{poss}\ s.\ s|\text{-}p = t$
⟨*proof*⟩

lemma *subt-at-imp-ctxt*:
assumes $p \in \mathit{poss}\ s$
shows $\exists\, C.\ C\langle s|\text{-}p\rangle = s$
⟨*proof*⟩

lemma *subt-at-imp-supteq'*:
assumes $p \in \mathit{poss}\ s$ **and** $s|\text{-}p = t$
shows $s \trianglerighteq t$
⟨*proof*⟩

lemma *subt-at-imp-supteq*: $p \in \mathit{poss}\ s \Longrightarrow s \trianglerighteq s|\text{-}p$
⟨*proof*⟩

lemma *fun-poss-ctxt-apply-term*:
assumes $p \in \mathit{fun\text{-}poss}\ C\langle s\rangle$
shows $(\forall\, t.\ p \in \mathit{fun\text{-}poss}\ C\langle t\rangle) \vee (\exists\, q.\ p = (\mathit{hole\text{-}pos}\ C) \mathbin{@} q \wedge q \in \mathit{fun\text{-}poss}\ s)$
⟨*proof*⟩

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

lemma *supt-imp-subt-at-nepos*:
assumes $s \rhd t$ **shows** $\exists\, i\ p.\ i\#p \in \mathit{poss}\ s \wedge s|\text{-}\ (i\#p) = t$
⟨*proof*⟩

lemma *arg-neq*:
assumes $i < \mathit{length}\ ss$ **and** $ss!i = \mathit{Fun}\ f\ ss$ **shows** *False*
⟨*proof*⟩

lemma *subt-at-nepos-neq*:
assumes $i\#p \in poss\ s$ **shows** $s|\text{-}(i\#p) \neq s$
⟨*proof*⟩

lemma *subt-at-nepos-imp-supt*:
assumes $i\#p \in poss\ s$ **shows** $s \rhd s\ |\text{-}\ (i\#p)$
⟨*proof*⟩

lemma *subt-at-nepos-imp-nectxt*:
assumes $i\#p \in poss\ s$ **and** $s|\text{-}(i\#p) = t$ **shows** $\exists\, C.\ C \neq \Box \wedge C\langle t\rangle = s$
⟨*proof*⟩

lemma *supteq-subst-cases'*:
$s \cdot \sigma \trianglerighteq t \Longrightarrow (\exists\ u.\ s \trianglerighteq u \wedge is\text{-}Fun\ u \wedge t = u \cdot \sigma) \vee (\exists\ x.\ x \in vars\text{-}term\ s \wedge \sigma\ x \trianglerighteq t)$
⟨*proof*⟩

lemma *size-const-subst*[*simp*]: $size\ (t \cdot (\lambda\ \text{-}\ .\ Fun\ f\ []))\ =\ size\ t$
⟨*proof*⟩

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

lemma *supteq-subst-cases* [*consumes 1*, *case-names in-term in-subst*]:
$s \cdot \sigma \trianglerighteq t \Longrightarrow$
$(\bigwedge u.\ s \trianglerighteq u \Longrightarrow is\text{-}Fun\ u \Longrightarrow t = u \cdot \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ x \in vars\text{-}term\ s \Longrightarrow \sigma\ x \trianglerighteq t \Longrightarrow P) \Longrightarrow$
P
⟨*proof*⟩

lemma *poss-subst-apply-term*:
assumes $p \in poss\ (t \cdot \sigma)$ **and** $p \notin fun\text{-}poss\ t$
obtains $q\ r\ x$ **where** $p = q\ @\ r$ **and** $q \in poss\ t$ **and** $t\ |\text{-}\ q = Var\ x$ **and** $r \in poss\ (\sigma\ x)$
⟨*proof*⟩

lemma *subt-at-subst* [*simp*]:
assumes $p \in poss\ t$ **shows** $(t \cdot \sigma)\ |\text{-}\ p = (t\ |\text{-}\ p) \cdot \sigma$
⟨*proof*⟩

lemma *vars-term-size*:
assumes $x \in vars\text{-}term\ t$
shows $size\ (\sigma\ x) \leq size\ (t \cdot \sigma)$
⟨*proof*⟩

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

$\sigma \mid s\ V = (\lambda x.\ \mathit{if}\ x \in V\ \mathit{then}\ \sigma(x)\ \mathit{else}\ \mathit{Var}\ x)$

lemma *subst-restrict-Int* [*simp*]:
$(\sigma \mid s\ V\ ) \mid s\ W = \sigma \mid s\ (V \cap W)$
⟨*proof*⟩

lemma *subst-domain-Var-conv* [*iff*]:
$\mathit{subst\text{-}domain}\ \sigma = \{\} \longleftrightarrow \sigma = \mathit{Var}$
⟨*proof*⟩

lemma *subst-compose-Var*[*simp*]: $\sigma \circ_s \mathit{Var} = \sigma$ ⟨*proof*⟩

lemma *Var-subst-compose*[*simp*]: $\mathit{Var} \circ_s \sigma = \sigma$ ⟨*proof*⟩

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

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

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

end

lemma *subst-power-compose-distrib*:
$\mu\ \widehat{\ }\widehat{\ }\ (m + n) = (\mu\ \widehat{\ }\widehat{\ }\ m \circ_s \mu\ \widehat{\ }\widehat{\ }\ n)$ ⟨*proof*⟩

lemma *subst-power-Suc*: $\mu\ \widehat{\ }\widehat{\ }\ (\mathit{Suc}\ i) = \mu\ \widehat{\ }\widehat{\ }\ i \circ_s \mu$
⟨*proof*⟩

lemma *subst-pow-mult*: $((\sigma :: ('f,'v)\mathit{subst})\ \widehat{\ }\widehat{\ }\ n)\ \widehat{\ }\widehat{\ }\ m = \sigma\ \widehat{\ }\widehat{\ }\ (n * m)$
⟨*proof*⟩

lemma *subst-domain-pow*: $\mathit{subst\text{-}domain}\ (\sigma\ \widehat{\ }\widehat{\ }\ n) \subseteq \mathit{subst\text{-}domain}\ \sigma$
⟨*proof*⟩

lemma *subt-at-Cons-distr* [*simp*]:
assumes $i \#\ p \in \mathit{poss}\ t$ **and** $p \neq []$
shows $t \mid\text{-}\ (i \#\ p) = (t \mid\text{-}\ [i]) \mid\text{-}\ p$
⟨*proof*⟩

lemma *subt-at-append* [*simp*]:
$p \in \mathit{poss}\ t \Longrightarrow t \mid\text{-}\ (p\ @@\ q) = (t \mid\text{-}\ p) \mid\text{-}\ q$
⟨*proof*⟩

lemma *subt-at-pos-diff*:
assumes $p <_p q$ **and** p: $p \in \mathit{poss}\ s$

shows s |- p |- *pos-diff* q p = s |- q
⟨*proof*⟩

lemma *empty-pos-in-poss*[*simp*]: [] ∈ *poss* t ⟨*proof*⟩

lemma *poss-append-poss*[*simp*]: (p @ q ∈ *poss* t) = (p ∈ *poss* t ∧ q ∈ *poss* (t |- p)) (**is** *?l* = *?r*)
⟨*proof*⟩

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*⟩

lemma *poss-imp-subst-poss* [*simp*]:
assumes p ∈ *poss* t
shows p ∈ *poss* (t · σ)
⟨*proof*⟩

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*⟩

lemma *hole-pos-poss* [*simp*]: *hole-pos* C ∈ *poss* (C⟨t⟩)
⟨*proof*⟩

lemma *hole-pos-poss-conv*: (*hole-pos* C @ q) ∈ *poss* (C⟨t⟩) ⟷ q ∈ *poss* t
⟨*proof*⟩

lemma *subt-at-hole-pos* [*simp*]: C⟨t⟩ |- *hole-pos* C = t
⟨*proof*⟩

lemma *hole-pos-power-poss* [*simp*]: (*hole-pos* C) ^ (*n*::*nat*) ∈ *poss* ((C ^ n)⟨t⟩)
⟨*proof*⟩

lemma *poss-imp-ctxt-subst-poss* [*simp*]:
assumes p ∈ *poss* (C⟨t⟩)
shows p ∈ *poss* ((C ·$_c$ σ)⟨t · σ⟩)
⟨*proof*⟩

lemma *poss-Cons-poss*[*simp*]: (i # p ∈ *poss* t) = (i < *length* (*args* t) ∧ p ∈ *poss* (*args* t ! i))
⟨*proof*⟩

lemma *less-pos-imp-supt*:
assumes *less*: p' <$_p$ p **and** p: p ∈ *poss* t
shows t |- p ◁ t |- p'
⟨*proof*⟩

lemma *less-eq-pos-imp-supt-eq*:
assumes *less-eq*: $p' \leq_p p$ **and** *p*: $p \in poss\ t$
shows $t\ |\text{-}\ p \trianglelefteq t\ |\text{-}\ p'$
⟨*proof*⟩

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*⟩

inductive
$subst\text{-}instance :: ('f, 'v)\ term \Rightarrow ('f, 'v)\ term \Rightarrow bool\ (\langle(\text{-}/ \preceq \text{-})\rangle\ [56, 56]\ 55)$
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*⟩

lemma *subst-instance-refl*: $s \preceq s$
⟨*proof*⟩

lemma *subst-neutral*: $subst\text{-}domain\ \sigma \subseteq V \Longrightarrow (Var\ x)\cdot(\sigma\ |s\ (V - \{x\})) = (Var\ x)$
⟨*proof*⟩

lemma *subst-restrict-domain*[*simp*]: $\sigma\ |s\ subst\text{-}domain\ \sigma = \sigma$
⟨*proof*⟩

lemma *notin-subst-domain-imp-Var*:
assumes $x \notin subst\text{-}domain\ \sigma$
shows $\sigma\ x = Var\ x$
⟨*proof*⟩

lemma *subst-domain-neutral*[*simp*]:
assumes $subst\text{-}domain\ \sigma \subseteq V$
shows $(\sigma\ |s\ V) = \sigma$
⟨*proof*⟩

lemma *subst-restrict-UNIV*[*simp*]: $\sigma\ |s\ UNIV = \sigma$ ⟨*proof*⟩

lemma *subst-restrict-empty*[*simp*]: $\sigma\ |s\ \{\} = Var$ ⟨*proof*⟩

lemma *vars-term-subst-pow*:
$vars\text{-}term\ (t \cdot \sigma \widehat{\ } n) \subseteq vars\text{-}term\ t \cup \bigcup (vars\text{-}term\ `\ subst\text{-}range\ \sigma)$ (**is** - ⊆ *?R t*)
⟨*proof*⟩

lemma *coincidence-lemma*:

$t \cdot \sigma = t \cdot (\sigma \mid s \; vars\text{-}term \; t)$
⟨*proof*⟩

lemma *subst-domain-vars-term-subset*:
$subst\text{-}domain \; (\sigma \mid s \; vars\text{-}term \; t) \subseteq vars\text{-}term \; t$
⟨*proof*⟩

lemma *subst-restrict-single-Var* [*simp*]:
assumes $x \notin subst\text{-}domain \; \sigma$ **shows** $\sigma \mid s \; \{x\} = Var$
⟨*proof*⟩

lemma *subst-restrict-single-Var'*:
assumes $x \notin subst\text{-}domain \; \sigma$ **and** $\sigma \mid s \; V = Var$ **shows** $\sigma \mid s \; (\{x\} \cup V) = Var$
⟨*proof*⟩

lemma *subst-restrict-empty-set*:
$finite \; V \Longrightarrow V \cap subst\text{-}domain \; \sigma = \{\} \Longrightarrow \sigma \mid s \; V = Var$
⟨*proof*⟩

lemma *subst-restrict-Var*: $x \neq y \Longrightarrow Var \; y \cdot (\sigma \mid s \; (UNIV - \{x\})) = Var \; y \cdot \sigma$
⟨*proof*⟩

lemma *var-cond-stable*:
assumes $vars\text{-}term \; r \subseteq vars\text{-}term \; l$
shows $vars\text{-}term \; (r \cdot \mu) \subseteq vars\text{-}term \; (l \cdot \mu)$
⟨*proof*⟩

lemma *instance-no-supt-imp-no-supt*:
assumes $\neg \; s \cdot \sigma \rhd t \cdot \sigma$
shows $\neg \; s \rhd t$
⟨*proof*⟩

lemma *subst-image-subterm*:
assumes $x \in vars\text{-}term \; (Fun \; f \; ss)$
shows $Fun \; f \; ss \cdot \sigma \rhd \sigma \; x$
⟨*proof*⟩

lemma *funas-term-subst-pow*:
$funas\text{-}term \; (t \cdot \sigma \widehat{\;} n) \subseteq funas\text{-}term \; t \cup \bigcup (funas\text{-}term \; `\; subst\text{-}range \; \sigma)$
⟨*proof*⟩

lemma *funas-term-subterm-args*:
assumes *sF*: $funas\text{-}term \; s \subseteq F$
and *q*: $q \in poss \; s$
shows $\bigcup (funas\text{-}term \; `\; set \; (args \; (s \mid\text{-}\; q))) \subseteq F$
⟨*proof*⟩

lemma *get-var-or-const*: $\exists \; C \; t. \; s = C\langle t \rangle \wedge args \; t = []$
⟨*proof*⟩

lemma *supteq-Var-id* [*simp*]:
assumes *Var x* $\trianglerighteq$ *s* **shows** *s* = *Var x*
⟨*proof*⟩

lemma *arg-not-term* [*simp*]:
assumes *t* ∈ *set ts* **shows** *Fun f ts* ≠ *t*
⟨*proof*⟩

lemma *arg-subteq* [*simp*]: *t* ∈ *set ts* ⟹ *Fun f ts* $\trianglerighteq$ *t*
⟨*proof*⟩

lemma *supt-imp-args*:
assumes ∀ *t*. *s* ▷ *t* ⟶ *P t*
shows ∀ *t*∈*set* (*args s*). *P t*
⟨*proof*⟩

lemma *ctxt-apply-eq-False*[*simp*]: (*More f ss1 D ss2*)⟨*t*⟩ ≠ *t* (**is** *?C*⟨-⟩ ≠ -)
⟨*proof*⟩

lemma *supteq-imp-funs-term-subset*: *t* $\trianglerighteq$ *s* ⟹ *funs-term s* ⊆ *funs-term t*
⟨*proof*⟩

lemma *funs-term-subst*: *funs-term* (*t* · *σ*) = *funs-term t* ∪ ⋃ ((λ *x*. *funs-term* (*σ x*)) ‘ (*vars-term t*))
⟨*proof*⟩

lemma *set-set-cons*:
assumes *P x* **and** ⋀*y*. *y* ∈ *set xs* ⟹ *P y*
shows *y* ∈ *set* (*x* # *xs*) ⟹ *P y*
⟨*proof*⟩

lemma *ctxt-power-compose-distr*: *C* ^ (*m* + *n*) = *C* ^ *m* ∘$_c$ *C* ^ *n*
⟨*proof*⟩

lemma *subst-apply-id′*:
assumes *vars-term t* ∩ *V* = {}
shows *t* · (*σ* |*s V*) = *t*
⟨*proof*⟩

lemma *subst-apply-ctxt-id*:
assumes *vars-ctxt C* ∩ *V* = {}
shows *C* ·$_c$ (*σ* |*s V*) = *C*
⟨*proof*⟩

lemma *vars-term-Var-id*: *vars-term o Var* = (λ*x*. {*x*})
⟨*proof*⟩

lemma *ctxt-exhaust-rev*[*case-names Hole More*]:

assumes $C = \square \Longrightarrow P$ **and**
$\bigwedge D\ f\ ss1\ ss2.\ C = D \circ_c (More\ f\ ss1\ \square\ ss2) \Longrightarrow P$
shows P
⟨*proof*⟩

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

lemma *subst-extend-id*:
assumes $V \cap$ *set vs* = {} **and** *vars-term t* ⊆ *V*
shows *t* · *subst-extend* σ (*zip vs ts*) = *t* · σ
⟨*proof*⟩

lemma *funas-term-args*:
$\bigcup$(*funas-term* ' *set* (*args t*)) ⊆ *funas-term t*
⟨*proof*⟩

lemma *subst-extend-absorb*:
assumes *distinct vs* **and** *length vs* = *length ss*
shows *map* ($\lambda t.$ *t* · *subst-extend* σ (*zip vs ss*)) (*map Var vs*) = *ss* (**is** *?ss* = -)
⟨*proof*⟩

abbreviation *map-funs-term f* ≡ *term.map-term f* ($\lambda x.$ *x*)

abbreviation *map-funs-ctxt f* ≡ *map-ctxt f* ($\lambda x.$ *x*)

lemma *funs-term-map-funs-term-id*: ($\bigwedge$ *f*. *f* ∈ *funs-term t* ⟹ *h f* = *f*) ⟹
map-funs-term h t = *t*
⟨*proof*⟩

lemma *funs-term-map-funs-term*:
funs-term (*map-funs-term h t*) ⊆ *range h*
⟨*proof*⟩

fun *map-funs-subst* :: ('*f* ⇒ '*g*) ⇒ ('*f*, '*v*) *subst* ⇒ ('*g*, '*v*) *subst* **where**
map-funs-subst fg σ = ($\lambda x.$ *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*
⟨*proof*⟩

lemma *map-funs-subst-distrib* [*simp*]:
map-funs-term fg (*t* · σ) = *map-funs-term fg t* · *map-funs-subst fg* σ
⟨*proof*⟩

lemma *size-map-funs-term* [*simp*]:
size (*map-funs-term fg t*) = *size t*
⟨*proof*⟩

lemma *fold-ident* [*simp*]: *Term-More.fold Var Fun t* = *t*
⟨*proof*⟩

lemma *map-funs-term-ident* [*simp*]:
map-funs-term id t = *t*
⟨*proof*⟩

lemma *ground-map-funs-term* [*simp*]:
ground (*map-funs-term fg t*) = *ground t*
⟨*proof*⟩

lemma *map-funs-term-power*:
fixes *f* :: *'f* ⇒ *'f*
shows ((*map-funs-term f*) ^^ *n*) = *map-funs-term* (*f* ^^ *n*)
⟨*proof*⟩

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

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*⟩

fun *map-funs-ctxt-wa* :: (*'f* × *nat* ⇒ *'g*) ⇒ (*'f*, *'v*) *ctxt* ⇒ (*'g*, *'v*) *ctxt*
where
map-funs-ctxt-wa fg □ = □ |
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 ≡ (λ*x*. *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*

⟨*proof*⟩

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*⟩
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *notin-subst-restrict* [*simp*]:
assumes *x* ∉ *V* **shows** (*σ* |*s* *V*) *x* = *Var x*
⟨*proof*⟩

lemma *in-subst-restrict* [*simp*]:
assumes *x* ∈ *V* **shows** (*σ* |*s* *V*) *x* = *σ x*
⟨*proof*⟩

lemma *coincidence-lemma′*:
assumes *vars-term t* ⊆ *V*
shows *t* · (*σ* |*s* *V*) = *t* · *σ*
⟨*proof*⟩

lemma *vars-term-map-funs-term* [*simp*]:
vars-term ∘ *map-funs-term* (*f* :: (′*f* ⇒ ′*g*)) = *vars-term*
⟨*proof*⟩

lemma *vars-term-map-funs-term2* [*simp*]:
vars-term (*map-funs-term f t*) = *vars-term t*
⟨*proof*⟩

lemma *map-funs-term-wa-ctxt-split*:
assumes *map-funs-term-wa fg s* = *lC*⟨*lt*⟩
shows ∃ *C t*. *s* = *C*⟨*t*⟩ ∧ *map-funs-term-wa fg t* = *lt* ∧ *map-funs-ctxt-wa fg C* = *lC*
⟨*proof*⟩

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*)) ·$_c$ *subst-extend*

σ (*zip* (*take i vs*@*drop* (*Suc i*) *vs*) (*ss1*@*ss2*)) = *More f ss1* □ *ss2*
⟨*proof*⟩

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*)) ·$_c$ *subst-extend* σ
(*zip* (*take i vs*@*drop i vs*) (*ss1*@*ss2*)) = *More f ss1* □ *ss2*
⟨*proof*⟩

lemma *distinct-map-Var*:
assumes *distinct xs* **shows** *distinct* (*map Var xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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

definition *subst-fun-range* :: ('*f*, '*v*, '*w*) *gsubst* ⇒ '*w set*
where
subst-fun-range $\sigma = \bigcup$(*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*)
⟨*proof*⟩

lemma *variants-imp-image-vars-term-eq*:
assumes $s \cdot \sigma = t$ **and** $t \cdot \tau = s$
shows (*the-Var* ∘ σ) ‘ *vars-term s* = *vars-term t*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Var-the-Var-id*:
assumes $\forall t \in$ *set ts*. *is-Var t*
shows *map Var* (*map the-Var ts*) = *ts*
⟨*proof*⟩

lemma *distinct-the-vars*:
assumes $\forall t \in$ *set ts*. *is-Var t*
and *distinct ts*

shows *distinct* (*map the-Var ts*)
⟨*proof*⟩

lemma *map-funs-term-eq-imp-map-funs-term-map-vars-term-eq*:
map-funs-term fg s = *map-funs-term fg t* ⟹ *map-funs-term fg* (*map-vars-term vw s*) = *map-funs-term fg* (*map-vars-term vw t*)
⟨*proof*⟩

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*⟩

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 ⇒ *σ y*
| *Inr y* ⇒ *τ y*))

lemma *merge-substs-left*:
map-vars-term Inl s · (*merge-substs σ δ*) = *s* · *σ*
⟨*proof*⟩

lemma *merge-substs-right*:
map-vars-term Inr s · (*merge-substs σ δ*) = *s* · *δ*
⟨*proof*⟩

fun *map-vars-subst-ran* :: (*'w* ⇒ *'u*) ⇒ (*'f*, *'v*, *'w*) *gsubst* ⇒ (*'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* · *σ*) = *t* · *map-vars-subst-ran m σ*
⟨*proof*⟩

lemma *size-subst*: *size t* ≤ *size* (*t* · *σ*)
⟨*proof*⟩

lemma *eq-ctxt-subst-iff* [*simp*]:
(*t* = *C*⟨*t* · *σ*⟩) ⟷ *C* = □ ∧ (∀ *x*∈*vars-term t*. *σ x* = *Var x*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *Fun-Nil-supt*[*elim*!]: *Fun f* [] ⊳ *t* ⟹ *P* ⟨*proof*⟩

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

assumes $\bigwedge x.\ x \in \mathit{vars\text{-}term}\ t \Longrightarrow f\ x = g\ x$
shows $\mathit{map\text{-}vars\text{-}term}\ f\ t = \mathit{map\text{-}vars\text{-}term}\ g\ t$
⟨*proof*⟩

lemma *map-funs-term-ctxt-decomp*:
assumes $\mathit{map\text{-}funs\text{-}term}\ \mathit{fg}\ t = C\langle s\rangle$
shows $\exists\ D\ u.\ C = \mathit{map\text{-}funs\text{-}ctxt}\ \mathit{fg}\ D \wedge s = \mathit{map\text{-}funs\text{-}term}\ \mathit{fg}\ u \wedge t = D\langle u\rangle$
⟨*proof*⟩

lemma *funas-term-map-vars-term* [*simp*]:
$\mathit{funas\text{-}term}\ (\mathit{map\text{-}vars\text{-}term}\ \tau\ t) = \mathit{funas\text{-}term}\ t$
⟨*proof*⟩

lemma *funs-term-funas-term*:
$\mathit{funs\text{-}term}\ t = \mathit{fst}\ `\ (\mathit{funas\text{-}term}\ t)$
⟨*proof*⟩

lemma *funas-term-map-funs-term*:
$\mathit{funas\text{-}term}\ (\mathit{map\text{-}funs\text{-}term}\ \mathit{fg}\ t) = (\lambda\ (f,n).\ (\mathit{fg}\ f,n))\ `\ (\mathit{funas\text{-}term}\ t)$
⟨*proof*⟩

lemma *supt-imp-arg-or-supt-of-arg*:
assumes $\mathit{Fun}\ f\ \mathit{ss} \rhd t$
shows $t \in \mathit{set}\ \mathit{ss} \vee (\exists\, s \in \mathit{set}\ \mathit{ss}.\ s \rhd t)$
⟨*proof*⟩

lemma *supt-Fun-imp-arg-supteq*:
assumes $\mathit{Fun}\ f\ \mathit{ss} \rhd t$ **shows** $\exists\, s \in \mathit{set}\ \mathit{ss}.\ s \trianglerighteq t$
⟨*proof*⟩

lemma *subt-iff-eq-or-subt-of-arg*:
assumes $s = \mathit{Fun}\ f\ \mathit{ss}$
shows $\{t.\ s \trianglerighteq t\} = ((\bigcup u \in \mathit{set}\ \mathit{ss}.\ \{t.\ u \trianglerighteq t\}) \cup \{s\})$
⟨*proof*⟩

The set of subterms of a term is finite.

lemma *finite-subterms*: $\mathit{finite}\ \{s.\ t \trianglerighteq s\}$
⟨*proof*⟩

lemma *Fun-supteq*: $\mathit{Fun}\ f\ \mathit{ts} \trianglerighteq u \longleftrightarrow \mathit{Fun}\ f\ \mathit{ts} = u \vee (\exists\, t{\in}\mathit{set}\ \mathit{ts}.\ t \trianglerighteq u)$
⟨*proof*⟩

lemma *subst-ctxt-distr*: $s = C\langle t\rangle{\cdot}\sigma \Longrightarrow \exists\, D.\ s = D\langle t{\cdot}\sigma\rangle$
⟨*proof*⟩

lemma *ctxt-of-pos-term-subst*:
assumes $p \in \mathit{poss}\ t$
shows $\mathit{ctxt\text{-}of\text{-}pos\text{-}term}\ p\ (t \cdot \sigma) = \mathit{ctxt\text{-}of\text{-}pos\text{-}term}\ p\ t \cdot_c \sigma$
⟨*proof*⟩

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*⟩

lemma *subst-ext*:
assumes $\forall x{\in}V.\ \sigma\ x = \tau\ x$ **shows** $\sigma\ |s\ V = \tau\ |s\ V$
⟨*proof*⟩

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

lemma *map-vars-term-ctxt-commute*:
$map\text{-}vars\text{-}term\ m\ (c\langle t\rangle) = (map\text{-}vars\text{-}ctxt\ m\ c)\langle map\text{-}vars\text{-}term\ m\ t\rangle$
⟨*proof*⟩

lemma *map-vars-term-inj-compose*:
assumes *inj*: $\bigwedge x.\ n\ (m\ x) = x$
shows $map\text{-}vars\text{-}term\ n\ (map\text{-}vars\text{-}term\ m\ t) = t$
⟨*proof*⟩

lemma *inj-map-vars-term-the-inv*:
assumes *inj f*
shows $map\text{-}vars\text{-}term\ (the\text{-}inv\ f)\ (map\text{-}vars\text{-}term\ f\ t) = t$
⟨*proof*⟩

lemma *map-vars-ctxt-subst*:
$map\text{-}vars\text{-}ctxt\ m\ (C\ \cdot_c\ \sigma) = C\ \cdot_c\ map\text{-}vars\text{-}subst\text{-}ran\ m\ \sigma$
⟨*proof*⟩

lemma *poss-map-vars-term* [*simp*]:
$poss\ (map\text{-}vars\text{-}term\ f\ t) = poss\ t$
⟨*proof*⟩

lemma *map-vars-term-subt-at* [*simp*]:
$p \in poss\ t \Longrightarrow map\text{-}vars\text{-}term\ f\ (t\ |\text{-}\ p) = (map\text{-}vars\text{-}term\ f\ t)\ |\text{-}\ p$
⟨*proof*⟩

lemma *hole-pos-subst*[*simp*]: $hole\text{-}pos\ (C\ \cdot_c\ \sigma) = hole\text{-}pos\ C$
⟨*proof*⟩

lemma *hole-pos-ctxt-compose*[*simp*]: $hole\text{-}pos\ (C\ \circ_c\ D) = hole\text{-}pos\ C\ @\ hole\text{-}pos\ D$
⟨*proof*⟩

lemma *subst-left-right*: $t \cdot \mu^{\frown\frown} n \cdot \mu = t \cdot \mu \cdot \mu^{\frown\frown} n$
⟨*proof*⟩

lemma *subst-right-left*: $t \cdot \mu \cdot \mu^{\frown\frown} n = t \cdot \mu^{\frown\frown} n \cdot \mu$ ⟨*proof*⟩

lemma *subt-at-id-imp-eps*:
assumes *p*: $p \in poss\ t$ **and** *id*: $t\ |\text{-}\ p = t$
shows $p = []$
⟨*proof*⟩

lemma *pos-into-subst*:
assumes *t*: $t \cdot \sigma = s$ **and** *p*: $p \in poss\ s$ **and** *nt*: $\neg\ (p \in poss\ t \wedge is\text{-}Fun\ (t\ |\text{-}\ p))$
shows $\exists\, q\ q'\ x.\ p = q\ @\ q' \wedge q \in poss\ t \wedge t\ |\text{-}\ q = Var\ x$
⟨*proof*⟩

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

lemma *replace-at-ident*:
assumes $p \in poss\ t$ **and** $t\ |\text{-}\ p = s$
shows *replace-at t p s* = *t*
⟨*proof*⟩

lemma *ctxt-of-pos-term-append*:
assumes $p \in poss\ t$
shows *ctxt-of-pos-term* $(p\ @\ q)\ t$ = *ctxt-of-pos-term p t* $\circ_c$ *ctxt-of-pos-term q* $(t\ |\text{-}\ p)$
⟨*proof*⟩

lemma *parallel-replace-at*:
fixes *p1* :: *pos*
assumes *parallel*: $p1 \perp p2$
and *p1*: $p1 \in poss\ t$
and *p2*: $p2 \in poss\ t$
shows *replace-at* (*replace-at t p1 s1*) *p2 s2* = *replace-at* (*replace-at t p2 s2*) *p1 s1*
⟨*proof*⟩

lemma *parallel-replace-at-subt-at*:
fixes *p1* :: *pos*
assumes *parallel*: $p1 \perp p2$
and *p1*: $p1 \in poss\ t$
and *p2*: $p2 \in poss\ t$
shows (*replace-at t p1 s1*) $|\text{-}\ p2 = t\ |\text{-}\ p2$
⟨*proof*⟩

lemma *parallel-poss-replace-at*:
fixes *p1* :: *pos*
assumes *parallel*: $p1 \perp p2$
and *p1*: $p1 \in poss\ t$
shows $(p2 \in poss\ (replace\text{-}at\ t\ p1\ s1)) = (p2 \in poss\ t)$
⟨*proof*⟩

lemma *replace-at-subt-at*: $p \in poss\ t \Longrightarrow (replace\text{-}at\ t\ p\ s)\ |\text{-}\ p = s$
⟨*proof*⟩

lemma *replace-at-below-poss*:
assumes *p*: $p' \in poss\ t$ **and** *le*: $p \leq_p p'$
shows $p \in poss\ (replace\text{-}at\ t\ p'\ s)$
⟨*proof*⟩

lemma *ctxt-of-pos-term-replace-at-below*:
assumes *p*: $p \in poss\ t$ **and** *le*: $p \leq_p p'$
shows $ctxt\text{-}of\text{-}pos\text{-}term\ p\ (replace\text{-}at\ t\ p'\ u) = ctxt\text{-}of\text{-}pos\text{-}term\ p\ t$
⟨*proof*⟩

lemma *ctxt-of-pos-term-hole-pos*[*simp*]:
$ctxt\text{-}of\text{-}pos\text{-}term\ (hole\text{-}pos\ C)\ (C\langle t\rangle) = C$
⟨*proof*⟩

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*⟩

lemma *var-pos-maximal*:
assumes *pt*:$p \in poss\ t$ **and** *x*:$t\ |\text{-}\ p = Var\ x$ **and** $q \neq []$
shows $p\ @\ q \notin poss\ t$
⟨*proof*⟩

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*: $p \in possc\ C \Longrightarrow p \in poss\ C\langle t\rangle$ ⟨*proof*⟩

lemma *less-eq-hole-pos-in-possc*:
assumes *pleq*:$p \leq_p hole\text{-}pos\ C$ **shows** $p \in possc\ C$
⟨*proof*⟩

lemma *hole-pos-in-possc*:*hole-pos* $C \in possc\ C$
⟨*proof*⟩

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

lemma *possc-not-below-hole-pos*:
assumes $p \in possc\ (C::('a,'b)\ ctxt)$ **shows** $\neg\ (hole\text{-}pos\ C <_p p)$
⟨*proof*⟩

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*⟩

All proper terms in a context

fun *ctxt-to-term-list* :: ('*f*, '*v*) *ctxt* ⇒ ('*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* ∈ *set* (*ctxt-to-term-list C*) ⟹ *C*⟨*s*⟩ ▷ *t*
⟨*proof*⟩

lemma *subteq-Var-imp-in-vars-term*:
r ⊵ *Var x* ⟹ *x* ∈ *vars-term r*
⟨*proof*⟩

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*⟩
⟨*proof*⟩

lemma *hole-pos-subt-at-ctxt*:
assumes *hole-pos C* = *p* @ *q*
shows *hole-pos* (*C* |-*c p*) = *q*
⟨*proof*⟩

lemma *subt-at-ctxt-compose*[*simp*]: (*C* ∘*c D*) |-*c hole-pos C* = *D*
⟨*proof*⟩

lemma *split-ctxt*:
assumes *hole-pos C* = *p* @ *q*
shows ∃ *D E*. *C* = *D* ∘*c E* ∧ *hole-pos D* = *p* ∧ *hole-pos E* = *q* ∧ *E* = *C* |-*c p*
⟨*proof*⟩

lemma *ctxt-subst-id*[*simp*]: *C* ·*c Var* = *C* ⟨*proof*⟩

the strict subterm relation between contexts and terms

inductive-set

suptc :: $(('f, 'v)\ ctxt \times ('f, 'v)\ term)\ set$
where
arg: $s \in set\ bef \cup set\ aft \Longrightarrow s \trianglerighteq t \Longrightarrow (More\ f\ bef\ C\ aft, t) \in suptc$
| *ctxt*: $(C,s) \in suptc \Longrightarrow (D \circ_c C, s) \in suptc$

hide-const *suptcp*

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

notation
$suptc\text{-}pred\ (\langle(-/ \rhd c\ -)\rangle\ [56, 56]\ 55)$

lemma *suptc-subst*: $C \rhd c\ s \Longrightarrow C \cdot_c \sigma \rhd c\ s \cdot \sigma$
⟨*proof*⟩

lemma *suptc-imp-supt*: $C \rhd c\ s \Longrightarrow C\langle t\rangle \rhd s$
⟨*proof*⟩

lemma *suptc-supteq-trans*: $C \rhd c\ s \Longrightarrow s \trianglerighteq t \Longrightarrow C \rhd c\ t$
⟨*proof*⟩

lemma *supteq-suptc-trans*: $C = D \circ_c E \Longrightarrow E \rhd c\ s \Longrightarrow C \rhd c\ s$
⟨*proof*⟩

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

lemma *supteq-ctxt-cases'*: $C\langle t \rangle \trianglerighteq u \Longrightarrow$
$C \rhd c\ u \vee t \trianglerighteq u \vee (\exists\ D\ C'.\ C = D \circ_c C' \wedge u = C'\langle t \rangle \wedge C' \neq \Box)$
⟨*proof*⟩

lemma *supteq-ctxt-cases*[*consumes 1*, *case-names in-ctxt in-term sub-ctxt*]: $C\langle t \rangle \trianglerighteq u \Longrightarrow$
$(C \rhd c\ u \Longrightarrow P) \Longrightarrow$
$(t \trianglerighteq u \Longrightarrow P) \Longrightarrow$
$(\bigwedge D\ C'.\ C = D \circ_c C' \Longrightarrow u = C'\langle t \rangle \Longrightarrow C' \neq \Box \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

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

lemma *range-vars-subst-compose-subset*:
$range\text{-}vars\ (\sigma \circ_s \tau) \subseteq (range\text{-}vars\ \sigma - subst\text{-}domain\ \tau) \cup range\text{-}vars\ \tau$ (**is** $?L \subseteq ?R$)
⟨*proof*⟩

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*⟩

definition *subst-compose'* :: $('f, 'v)$ *subst* $\Rightarrow$ $('f, 'v)$ *subst* $\Rightarrow$ $('f, 'v)$ *subst* **where**
subst-compose' $\sigma\ \tau = (\lambda\ x.$ *if* $(x \in$ *subst-domain* $\sigma)$ *then* $\sigma\ x \cdot \tau$ *else Var* $x)$

lemma *vars-subst-compose'*:
assumes *vars-subst* $\tau \cap$ *subst-domain* $\sigma = \{\}$
shows $\sigma \circ_s \tau = \tau \circ_s$ (*subst-compose'* $\sigma\ \tau$) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *vars-subst-compose'-pow*:
assumes *vars-subst* $\tau \cap$ *subst-domain* $\sigma = \{\}$
shows $\sigma\ \widehat{\ }\widehat{\ }\ n \circ_s \tau = \tau \circ_s$ (*subst-compose'* $\sigma\ \tau$) $\widehat{\ }\widehat{\ }\ n$
⟨*proof*⟩

lemma *subst-pow-commute*:
assumes $\sigma \circ_s \tau = \tau \circ_s \sigma$
shows $\sigma \circ_s (\tau\ \widehat{\ }\widehat{\ }\ n) = \tau\ \widehat{\ }\widehat{\ }\ n \circ_s \sigma$
⟨*proof*⟩

lemma *subst-power-commute*:
assumes $\sigma \circ_s \tau = \tau \circ_s \sigma$
shows $(\sigma\ \widehat{\ }\widehat{\ }\ n) \circ_s (\tau\ \widehat{\ }\widehat{\ }\ n) = (\sigma \circ_s \tau)\widehat{\ }\widehat{\ }n$
⟨*proof*⟩

lemma *vars-term-ctxt-apply*:
vars-term $(C\langle t\rangle) =$ *vars-ctxt* $C \cup$ *vars-term* t
⟨*proof*⟩

lemma *vars-ctxt-pos-term*:
assumes $p \in$ *poss* t
shows *vars-term* $t =$ *vars-ctxt* (*ctxt-of-pos-term* $p\ t) \cup$ *vars-term* $(t$ |- $p)$
⟨*proof*⟩

lemma *vars-term-subt-at*:
assumes $p \in$ *poss* t
shows *vars-term* $(t$ |- $p) \subseteq$ *vars-term* t
⟨*proof*⟩

lemma *Var-pow-Var*[*simp*]: *Var* $\widehat{\ }\widehat{\ }\ n =$ *Var*
⟨*proof*⟩

definition *is-inverse-renaming* :: $('f, 'v)$ *subst* $\Rightarrow$ $('f, 'v)$ *subst* **where**
is-inverse-renaming $\sigma\ y = ($
if Var $y \in$ *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*: $x \in$ *subst-domain* σ
shows *Var* $x \cdot \sigma \cdot$ *is-inverse-renaming* $\sigma =$ *Var* x (**is** - $\cdot$ *?σ* = -)
⟨*proof*⟩

lemma *is-renaming-inverse-range*:
assumes *varren*: *is-renaming* σ
and *x*: *Var* $x \notin$ *subst-range* σ
shows *Var* $x \cdot \sigma \cdot$ *is-inverse-renaming* $\sigma =$ *Var* x (**is** - $\cdot$ *?σ* = -)
⟨*proof*⟩

lemma *vars-subst-compose*:
vars-subst $(\sigma \circ_s \tau) \subseteq$ *vars-subst* $\sigma \cup$ *vars-subst* τ
⟨*proof*⟩

lemma *vars-subst-compose-update*:
assumes *x*: $x \notin$ *vars-subst* σ
shows $\sigma \circ_s \tau(x := t) = (\sigma \circ_s \tau)(x := t)$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *subst-variants-imp-eq*:
assumes $\sigma \circ_s \sigma' = \tau$ **and** $\tau \circ_s \tau' = \sigma$
shows $\bigwedge x.\ \sigma\ x \cdot \sigma' = \tau\ x\ \bigwedge x.\ \tau\ x \cdot \tau' = \sigma\ x$
⟨*proof*⟩

lemma *poss-subst-choice*: **assumes** $p \in$ *poss* $(t \cdot \sigma)$ **shows**
$p \in$ *poss* $t \wedge$ *is-Fun* $(t$ |- $p) \vee$
$(\exists\ x\ q1\ q2.\ q1 \in$ *poss* $t \wedge q2 \in$ *poss* $(\sigma\ x) \wedge t$ |- $q1 =$ *Var* $x \wedge x \in$ *vars-term* t
$\wedge\ p = q1$ @ $q2 \wedge t \cdot \sigma$ |- $p = \sigma\ x$ |- $q2)$ (**is** - $\vee$ $(\exists\ x\ q1\ q2.$ *?p x q1 q2 t p*))
⟨*proof*⟩

fun *vars-term-list* :: $('f, 'v)$ *term* $\Rightarrow$ $'v$ *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*
⟨*proof*⟩

lemma *unary-vars-term-list*:
assumes *t*: *funas-term t* $\subseteq F$
and *unary*: $\bigwedge f\ n.\ (f, n) \in F \Longrightarrow n \leq 1$
shows *vars-term-list t* = [] $\vee$ $(\exists\ x \in$ *vars-term t*. *vars-term-list t* = [*x*])
⟨*proof*⟩

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* $\Rightarrow$ $'f$ *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*
  ⟨*proof*⟩

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* $\Rightarrow$ $('f \times nat)$ *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*
  ⟨*proof*⟩

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

definition *funas-args-term-list* :: $('f, 'v)$ *term* $\Rightarrow$ $('f \times nat)$ *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*
  ⟨*proof*⟩

lemma *vars-term-list-map-funs-term*:
  vars-term-list (*map-funs-term fg t*) = *vars-term-list t*
⟨*proof*⟩

lemma *funs-term-list-map-funs-term*:
  funs-term-list (*map-funs-term fg t*) = *map fg* (*funs-term-list t*)
⟨*proof*⟩

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* $\Rightarrow$ $'f$ *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* $\Rightarrow$ $'f$ *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*
⟨*proof*⟩

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*⟩

lemma *ground-vars-term-ms-empty*:
ground t = (*vars-term-ms t* = {#})
⟨*proof*⟩

lemma *vars-term-ms-map-funs-term* [*simp*]:
vars-term-ms (*map-funs-term fg t*) = *vars-term-ms t*
⟨*proof*⟩

lemma *funs-term-ms-map-funs-term*:
funs-term-ms (*map-funs-term fg t*) = *image-mset fg* (*funs-term-ms t*)
⟨*proof*⟩

lemma *supteq-imp-vars-term-ms-subset*:
s $\trianglerighteq$ *t* $\Longrightarrow$ *vars-term-ms t* $\subseteq\#$ *vars-term-ms s*
⟨*proof*⟩

lemma *mset-funs-term-list*:
mset (*funs-term-list t*) = *funs-term-ms t*
⟨*proof*⟩

Creating substitutions from lists

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

definition *mk-subst* :: ('*v* ⇒ ('*f*, '*w*) *term*) ⇒ ('*f*, '*v*, '*w*) *gsubstL* ⇒ ('*f*, '*v*, '*w*) *gsubst* **where**
mk-subst d xts ≡
(λ*x*. *case map-of xts x of*
None ⇒ *d x*
| *Some t* ⇒ *t*)

lemma *mk-subst-not-mem*:
assumes *x*: *x* ∉ *set xs*

shows *mk-subst f* (*zip xs ts*) *x* = *f x*
⟨*proof*⟩

lemma *mk-subst-not-mem′*:
assumes *x*: *x* ∉ *set* (*map fst ss*)
shows *mk-subst f ss x* = *f x*
⟨*proof*⟩

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*⟩

lemma *mk-subst-Nil* [*simp*]:
mk-subst d [] = *d*
⟨*proof*⟩

lemma *mk-subst-concat*:
assumes *x* ∉ *set* (*map fst xs*)
shows (*mk-subst f* (*xs*@*ys*)) *x* = (*mk-subst f ys*) *x*
⟨*proof*⟩

lemma *mk-subst-concat-Cons*:
assumes *x* ∈ *set* (*map fst ss*)
shows *mk-subst f* (*concat* (*ss*#*ss′*)) *x* = *mk-subst f ss x*
⟨*proof*⟩

lemma *vars-term-var-poss-iff*:
x ∈ *vars-term t* ⟷ (∃ *p*. *p* ∈ *var-poss t* ∧ *Var x* = *t* |- *p*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *vars-term-poss-subt-at*:
assumes *x* ∈ *vars-term t*
obtains *q* **where** *q* ∈ *poss t* **and** *t* |- *q* = *Var x*
⟨*proof*⟩

lemma *vars-ctxt-subt-at′*:
assumes *x* ∈ *vars-ctxt C*
and *p* ∈ *poss C*⟨*t*⟩
and *hole-pos C* = *p*
shows ∃ *q*. *q* ∈ *poss C*⟨*t*⟩ ∧ *parallel-pos p q* ∧ *C*⟨*t*⟩ |- *q* = *Var x*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *poss-is-Fun-fun-poss*:
assumes $p \in poss\ t$
and *is-Fun* $(t\ |\text{-}\ p)$
shows $p \in fun\text{-}poss\ t$
⟨*proof*⟩

lemma *fun-poss-map-vars-term*:
$fun\text{-}poss\ (map\text{-}vars\text{-}term\ f\ t) = fun\text{-}poss\ t$
⟨*proof*⟩

lemma *fun-poss-append-poss*:
assumes $p@q \in poss\ t\ q \neq []$
shows $p \in fun\text{-}poss\ t$
⟨*proof*⟩

lemma *fun-poss-append-poss'*:
assumes $p@q \in fun\text{-}poss\ t$
shows $p \in fun\text{-}poss\ t$
⟨*proof*⟩

lemma *fun-poss-in-ctxt*:
assumes $q@p \in fun\text{-}poss\ (C\langle t\rangle)$
and $hole\text{-}pos\ C = q$
shows $p \in fun\text{-}poss\ t$
⟨*proof*⟩

lemma *args-poss*:
assumes $i\ \#\ p \in poss\ t$
obtains $f\ ts$ **where** $t = Fun\ f\ ts\ p \in poss\ (ts!i)\ i < length\ ts$
⟨*proof*⟩

lemma *var-poss-parallel*:
assumes $p \in var\text{-}poss\ t$ **and** $q \in var\text{-}poss\ t$ **and** $p \neq q$
shows $p \perp q$
⟨*proof*⟩

lemma *ctxt-comp-equals*:
assumes *poss*:$p \in poss\ s\ p \in poss\ t$
and $ctxt\text{-}of\text{-}pos\text{-}term\ p\ s \circ_c C = ctxt\text{-}of\text{-}pos\text{-}term\ p\ t \circ_c D$
shows $C = D$
⟨*proof*⟩

lemma *ctxt-subst-comp-pos*:
assumes $q \in poss\ t$ **and** $p \in poss\ (t \cdot \tau)$
and $(ctxt\text{-}of\text{-}pos\text{-}term\ q\ t \cdot_c \sigma) \circ_c C = ctxt\text{-}of\text{-}pos\text{-}term\ p\ (t \cdot \tau)$
shows $q \leq_p p$
⟨*proof*⟩

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

fun *ground-ctxt* :: ('*f*,'*v*)*ctxt* ⇒ *bool* **where**
  ground-ctxt Hole = *True*
| *ground-ctxt* (*More f ss1 C ss2*) =
    ((∀ *s*∈*set ss1*. *ground s*) ∧ (∀ *s*∈*set ss2*. *ground s*) ∧ *ground-ctxt C*)

lemma *ground-ctxt-apply*[*simp*]: *ground* (*C*⟨*t*⟩) = (*ground-ctxt C* ∧ *ground t*)
  ⟨*proof*⟩

lemma *ground-ctxt-compose*[*simp*]: *ground-ctxt* (*C* ∘$_c$ *D*) = (*ground-ctxt C* ∧ *ground-ctxt D*)
  ⟨*proof*⟩

Linearity of a term

fun *linear-term* :: ('*f*, '*v*) *term* ⇒ *bool*
  where
    linear-term (*Var* -) = *True* |
     linear-term (*Fun* - *ts*) = (*is-partition* (*map vars-term ts*) ∧ (∀ *t*∈*set ts*. *linear-term t*))

lemma *subst-merge*:
  assumes *part*: *is-partition* (*map vars-term ts*)
  shows ∃ σ. ∀ *i*<*length ts*. ∀ *x*∈*vars-term* (*ts* ! *i*). σ *x* = τ *i x*
⟨*proof*⟩

Matching for linear terms

fun *weak-match* :: ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term* ⇒ *bool*
  where
    weak-match - (*Var* -) ⟷ *True* |
    weak-match (*Var* -) (*Fun* - -) ⟷ *False* |
    weak-match (*Fun f ts*) (*Fun g ss*) ⟷
    f = *g* ∧ *length ts* = *length ss* ∧ (∀ *i* < *length ts*. *weak-match* (*ts* ! *i*) (*ss* ! *i*))

lemma *weak-match-refl*: *weak-match t t*
  ⟨*proof*⟩

lemma *weak-match-match*: *weak-match* (*t* · σ) *t*
  ⟨*proof*⟩

lemma *weak-match-map-funs-term*:
  weak-match t s ⟹ *weak-match* (*map-funs-term g t*) (*map-funs-term g s*)
⟨*proof*⟩

lemma *linear-weak-match*:
  assumes *linear-term l* **and** *weak-match t s* **and** *s* = *l* · σ
  shows ∃ τ. *t* = *l*·τ ∧ (∀ *x*∈*vars-term l*. *weak-match* (*Var x* · τ) (*Var x* · σ))
  ⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *linear-map-funs-term* [*simp*]:
linear-term (*map-funs-term f t*) = *linear-term t*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *subt-at-linear*:
assumes *linear-term t* **and** *p* ∈ *poss t*
shows *linear-term* (*t*|-*p*)
⟨*proof*⟩

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*) = {}
⟨*proof*⟩

lemma *ground-imp-linear-term* [*simp*]: *ground t* ⟹ *linear-term t*
⟨*proof*⟩

lemma *linear-vars-term-list*:

assumes *linear-term t*
shows *length* (*filter* ((=) *x*) (*vars-term-list t*)) ≤ *1*
⟨*proof*⟩

lemma *distinct-alt*:
assumes ∀ *x*. *length* (*filter* ((=) *x*) *xs*) ≤ *1*
shows *distinct xs*
⟨*proof*⟩

lemma *linear-term-distinct-vars*:
assumes *linear-term t*
shows *distinct* (*vars-term-list t*)
⟨*proof*⟩

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* (λ*g*. *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*⟩

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.