First-Order Terms*

Christian Sternagel René Thiemann

September 13, 2023

Abstract

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

Contents

*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

lemma (**in** *order*) *rtranclp-less-eq* [*simp*]:
$(\leq)^{**} = (\leq)$
⟨*proof*⟩

lemma (**in** *order*) *tranclp-less* [*simp*]:
$(<)^{++} = (<)$
⟨*proof*⟩

lemma (**in** *order*) *rtranclp-greater-eq* [*simp*]:
$(\geq)^{**} = (\geq)$
⟨*proof*⟩

lemma (**in** *order*) *tranclp-greater* [*simp*]:
$(>)^{++} = (>)$
⟨*proof*⟩

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* $(f \text{ ` } A) = card\ A$
shows *bij-betw f A* $(f \text{ ` } A)$
⟨*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
$fun\text{-}merge\ fs\ as\ a = (fs\ !\ (LEAST\ i.\ i < length\ as \wedge a \in as\ !\ i))\ a$

lemma *fun-merge-eq-nth*:
assumes *i*: $i < length\ as$
and *a*: $a \in as\ !\ i$
and *ident*: $\bigwedge i\ j\ a.\ i < length\ as \Longrightarrow j < length\ as \Longrightarrow a \in as\ !\ i \Longrightarrow a \in as\ !\ j \Longrightarrow (fs\ !\ i)\ a = (fs\ !\ j)\ a$
shows *fun-merge fs as a* $= (fs\ !\ i)\ a$
⟨*proof*⟩

lemma *fun-merge-part*:
assumes $\forall i{<}length\ as.\forall j{<}length\ as.\ i \neq j \longrightarrow as\ !\ i \cap as\ !\ j = \{\}$
and $i < length\ as$
and $a \in as\ !\ i$
shows *fun-merge fs as a* $= (fs\ !\ i)\ a$
⟨*proof*⟩

lemma *fun-merge*:
assumes *part*: $\forall i{<}length\ Xs.\forall j{<}length\ Xs.\ i \neq j \longrightarrow Xs\ !\ i \cap Xs\ !\ j = \{\}$
shows $\exists \sigma.\ \forall i{<}length\ Xs.\ \forall x{\in}\ Xs\ !\ i.\ \sigma\ x = \tau\ i\ x$
⟨*proof*⟩

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* ⟹ (*c* ⟹ *m* = *n*) ⟹ *guard b* >> *m* = *guard c* >> *n*
⟨*proof*⟩

lemma *guard-simps*:
guard b = *Some x* ⟷ *b*
guard b = *None* ⟷ ¬ *b*
⟨*proof*⟩

lemma *guard-elims*[*elim*]:
guard b = *Some x* ⟹ (*b* ⟹ *P*) ⟹ *P*
guard b = *None* ⟹ (¬ *b* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *guard-intros* [*intro*, *simp*]:
b ⟹ *guard b* = *Some* ()
¬ *b* ⟹ *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* ∧ *b*) = *guard a* ⟜ (λ -. *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 ⋀*x xs y ys*. *P xs ys* ⟹ *P* (*x* # *xs*) (*y* # *ys*)
and ⋀*ys*. *P* [] *ys*
and ⋀*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 (λx. 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 (λx. 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 (λf. mapM (λy. 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* -) = []

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 *member-image-the-Var-image-subst*:
assumes *is-var-σ*: ∀ *x*. *is-Var* (*σ x*)
shows *x* ∈ *the-Var* ‘ *σ* ‘ *V* ⟷ *Var x* ∈ *σ* ‘ *V*
⟨*proof*⟩

lemma *image-the-Var-image-subst-renaming-eq*:
assumes *is-var-σ*: ∀ *x*. *is-Var* (*ϱ x*)
shows *the-Var* ‘ *ϱ* ‘ *V* = (⋃ *x* ∈ *V*. *vars-term* (*ϱ x*))
⟨*proof*⟩

The variables of a term as multiset.

fun *vars-term-ms* :: (′*f*, ′*v*) *term* ⇒ ′*v multiset*
where
vars-term-ms (*Var x*) = {#*x*#} |
vars-term-ms (*Fun f ts*) = ∑# (*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* ⇒ (′*f* × *nat*) *option*
where
root (*Var x*) = *None* |
root (*Fun f ts*) = *Some* (*f*, *length ts*)

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

lemma *finite-Union-vars-term*:
finite (⋃ *t* ∈ *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⟦*Var x*⟧*α* = *α x*
| *I*⟦*Fun f ss*⟧*α* = *I f* [*I*⟦*s*⟧*α*. *s* ← *ss*]

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

lemma *eval-same-vars*:
assumes $\forall x \in$ *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** $\cdot$ 67)
where *subst-apply-term* $\equiv$ *eval-term Fun*

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

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

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

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

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

lemma *subst-apply-term-ident*:
vars-term t ∩ *subst-domain σ* = {} ⟹ *t* · *σ* = *t*
⟨*proof*⟩

lemma *vars-term-subst-apply-term*:
vars-term (*t* · *σ*) = (⋃ *x* ∈ *vars-term t*. *vars-term* (*σ x*))
⟨*proof*⟩

corollary *vars-term-subst-apply-term-subset*:
vars-term (*t* · *σ*) ⊆ *vars-term t* − *subst-domain σ* ∪ *range-vars σ*
⟨*proof*⟩

definition *is-renaming* :: ('*f*, '*v*) *subst* ⇒ *bool*
where
is-renaming σ ⟷ (∀ *x*. *is-Var* (*σ x*)) ∧ *inj-on σ* (*subst-domain σ*)

lemma *inv-renaming-sound*:
assumes *is-var-ϱ*: ∀ *x*. *is-Var* (*ϱ x*) **and** *inj ϱ*
shows *ϱ* ∘$_s$ (*Var* ∘ (*inv* (*the-Var* ∘ *ϱ*))) = *Var*
⟨*proof*⟩

lemma *ex-inverse-of-renaming*:
assumes ∀ *x*. *is-Var* (*ϱ x*) **and** *inj ϱ*
shows ∃ *τ*. *ϱ* ∘$_s$ *τ* = *Var*
⟨*proof*⟩

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

lemma *range-varsE* [*elim*]:
assumes *x* ∈ *range-vars σ*
and ⋀*t*. *x* ∈ *vars-term t* ⟹ *t* ∈ *subst-range σ* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *range-vars-subst-compose-subset*:

range-vars $(\sigma \circ_s \tau) \subseteq$ (*range-vars* σ − *subst-domain* τ) $\cup$ *range-vars* τ (**is** *?L* $\subseteq$ *?R*)
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

Applying a substitution to every term of a given set.

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

Composition of substitutions

lemma *subst-compose*: $(\sigma \circ_s \tau)\ x = \sigma\ x \cdot \tau$ ⟨*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*:
$num\text{-}funs\ (t \cdot \sigma) \geq num\text{-}funs\ t$
⟨*proof*⟩

lemma *sum-list-map-num-funs-subst*:
assumes $sum\text{-}list\ (map\ (num\text{-}funs \circ (\lambda t.\ t \cdot \sigma))\ ts) = sum\text{-}list\ (map\ num\text{-}funs$

ts)
shows $\forall\, i < \mathit{length}\ \mathit{ts}.\ \mathit{num\text{-}funs}\ (\mathit{ts}\ !\ i \cdot \sigma) = \mathit{num\text{-}funs}\ (\mathit{ts}\ !\ i)$
⟨*proof*⟩

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

lemma *finite-subst-domain-subst*:
$\mathit{finite}\ (\mathit{subst\text{-}domain}\ (\mathit{subst}\ x\ y))$
⟨*proof*⟩

lemma *subst-domain-compose*:
$\mathit{subst\text{-}domain}\ (\sigma \circ_s \tau) \subseteq \mathit{subst\text{-}domain}\ \sigma \cup \mathit{subst\text{-}domain}\ \tau$
⟨*proof*⟩

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

lemma *vars-term-subset-subst-eq*:
assumes $\mathit{vars\text{-}term}\ t \subseteq \mathit{vars\text{-}term}\ s$
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**
$\mathit{restrict\text{-}subst\text{-}domain}\ V\ \sigma\ x \equiv (\mathit{if}\ x \in V\ \mathit{then}\ \sigma\ x\ \mathit{else}\ \mathit{Var}\ x)$

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

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

lemma *subst-domain-restrict-subst-domain*[*simp*]:
$\mathit{subst\text{-}domain}\ (\mathit{restrict\text{-}subst\text{-}domain}\ V\ \sigma) = V \cap \mathit{subst\text{-}domain}\ \sigma$
⟨*proof*⟩

lemma *subst-apply-term-restrict-subst-domain*:
$\mathit{vars\text{-}term}\ t \subseteq V \Longrightarrow t \cdot \mathit{restrict\text{-}subst\text{-}domain}\ V\ \sigma = t \cdot \sigma$

⟨*proof*⟩

3.2 Rename the Domain of a Substitution

definition *rename-subst-domain* **where**
rename-subst-domain $\varrho\ \sigma\ x =$
(*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* $\varrho\ \sigma = t \cdot \sigma$
⟨*proof*⟩

3.3 Rename the Domain and Range of a Substitution

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

lemma *rename-subst-domain-range-Var-lhs*[*simp*]:

rename-subst-domain-range Var $\sigma = \sigma$
⟨*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: $\forall x.$ *is-Var* (ϱ *x*) **and** *inj* ϱ
shows $\varrho \circ_s$ *rename-subst-domain-range* ϱ $\sigma = \sigma \circ_s \varrho$
⟨*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: $\forall x.$ *is-Var* (ϱ *x*) **and** *inj* ϱ
shows $t \cdot \varrho \cdot$ *rename-subst-domain-range* ϱ $\sigma = t \cdot \sigma \cdot \varrho$
⟨*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*) ⟷ ($\forall$ *t* ∈ *set ts. ground t*)

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

lemma *ground-subst* [*simp*]:
ground ($t \cdot \sigma$) ⟷ ($\forall$ *x* ∈ *vars-term t. ground* (σ *x*))
⟨*proof*⟩

lemma *ground-subst-apply*:
assumes *ground t*
shows $t \cdot \sigma = t$
⟨*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 = $t \cdot$ (λ *x. Var* (*f x*))
⟨*proof*⟩

lemma *map-vars-term-eq*:
map-vars-term f s = $s \cdot$ (*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 *apply-subst-map-vars-term*:
map-vars-term m t · *σ* = *t* · (*σ* ∘ *m*)
⟨*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* (λ*p*. (*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* = ⋃ (*set-mset* (*image-mset* (λ*r*. *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*)) = (⋃ *x*∈*set xs* ∪ *set ys*. *vars-term x*)
⟨*proof*⟩

lemma *not-in-vars-mset-subst-mset* [*simp*]:
assumes *x* ∉ *vars-term t*
shows *x* ∉ *vars-mset* (*subst-mset* (*subst x t*) *M*)
⟨*proof*⟩
lemma *vars-mset-subst-mset-subset*:
vars-mset (*subst-mset* (*subst x t*) *M*) ⊆ *vars-mset M* ∪ *vars-term t* ∪ {*x*} (**is** *?L* ⊆ *?R*)
⟨*proof*⟩

lemma *Var-left-vars-mset-less*:
assumes *x* ∉ *vars-term t*
shows *vars-mset* (*subst-mset* (*subst x t*) *M*) ⊂ *vars-mset* (*add-mset* (*Var x*, *t*) *M*) (**is** *?L* ⊂ *?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 $P = \{\sigma.\ \forall\, e{\in}P.\ fst\ e \cdot \sigma = snd\ e\}$

lemma *matchers-vars-term-eq*:
assumes $\sigma \in$ *matchers* P **and** $\tau \in$ *matchers* P
and (s, t) $\in P$

shows $\forall x{\in}vars\text{-}term\ s.\ \sigma\ x = \tau\ x$
⟨*proof*⟩

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

lemma *matchers-insert* [*simp*]:
$matchers\ (insert\ e\ P) = \{\sigma.\ fst\ e \cdot \sigma = snd\ e\} \cap matchers\ P$
⟨*proof*⟩

lemma *matchers-Un* [*simp*]:
$matchers\ (P \cup P') = matchers\ P \cap matchers\ P'$
⟨*proof*⟩

lemma *matchers-set-zip* [*simp*]:
assumes $length\ ss = length\ ts$
shows $matchers\ (set\ (zip\ ss\ ts)) = \{\sigma.\ map\ (\lambda t.\ t \cdot \sigma)\ ss = ts\}$
⟨*proof*⟩

definition $matchers\text{-}map\ m = matchers\ ((\lambda x.\ (Var\ x,\ the\ (m\ x)))\ `\ Map.dom\ m)$

lemma *matchers-map-empty* [*simp*]:
$matchers\text{-}map\ Map.empty = UNIV$
⟨*proof*⟩

lemma *matchers-map-upd* [*simp*]:
assumes $\sigma\ x = None \vee \sigma\ x = Some\ t$
shows $matchers\text{-}map\ (\lambda y.\ if\ y = x\ then\ Some\ t\ else\ \sigma\ y) =$
$matchers\text{-}map\ \sigma \cap \{\tau.\ \tau\ x = t\}$ (**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 ⟷ (*x*, *y*) ∈ *matchrel*
⟨*proof*⟩

lemma *matchrel-rtrancl-matchers* [*simp*]:
assumes (*x*, *y*) ∈ *matchrel**
shows *matchers-map* (*snd x*) ∩ *matchers* (*set-mset* (*fst x*)) =
matchers-map (*snd y*) ∩ *matchers* (*set-mset* (*fst y*))
⟨*proof*⟩

lemma *subst-of-map-in-matchers-map* [*simp*]:
subst-of-map d m ∈ *matchers-map m*
⟨*proof*⟩

lemma *matchrel-sound*:
assumes ((*P*, *Map.empty*), ({#}, *σ*)) ∈ *matchrel**
shows *subst-of-map d σ* ∈ *matchers* (*set-mset P*)
⟨*proof*⟩

lemma *MATCH1-size-mset*:
assumes *MATCH1 x y*
shows *size-mset* (*fst x*) > *size-mset* (*fst y*)
⟨*proof*⟩

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

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

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

lemma *converse-matchrel-subset-matchless*:
matchrel^{-1} ⊆ *matchless*
⟨*proof*⟩

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

lemma *MATCH1-singleton-Var* [*intro*]:
σ x = *None* ⟹ *MATCH1* ({#(*Var x*, *t*)#}, *σ*) ({#}, *σ* (*x* ↦ *t*))
σ x = *Some t* ⟹ *MATCH1* ({#(*Var x*, *t*)#}, *σ*) ({#}, *σ* (*x* ↦ *t*))
⟨*proof*⟩

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

lemma *not-MATCH1-singleton-Var* [*dest*]:
¬ *MATCH1* ({#(*Var x*, *t*)#}, *σ*) ({#}, *σ* (*x* ↦ *t*)) ⟹ *σ x* ≠ *None* ∧ *σ x* ≠ *Some t*
⟨*proof*⟩

lemma *not-matchrelD*:
assumes ¬ (∃ *y*. (({#*e*#}, *σ*), *y*) ∈ *matchrel*)
shows (∃ *f ss x*. *e* = (*Fun f ss*, *Var x*)) ∨
(∃ *x t*. *e* = (*Var x*, *t*) ∧ *σ x* ≠ *None* ∧ *σ x* ≠ *Some t*) ∨
(∃ *f g ss ts*. *e* = (*Fun f ss*, *Fun g ts*) ∧ (*f* ≠ *g* ∨ *length ss* ≠ *length ts*))
⟨*proof*⟩

lemma *ne-matchers-imp-matchrel*:
assumes *matchers-map σ* ∩ *matchers* {*e*} ≠ {}
shows ∃ *y*. (({#*e*#}, *σ*), *y*) ∈ *matchrel*
⟨*proof*⟩

lemma *MATCH1-mono*:
assumes *MATCH1* (*P*, *σ*) (*P′*, *σ′*)
shows *MATCH1* (*P* + *M*, *σ*) (*P′* + *M*, *σ′*)
⟨*proof*⟩

lemma *matchrel-mono*:
assumes (*x*, *y*) ∈ *matchrel*
shows ((*fst x* + *M*, *snd x*), (*fst y* + *M*, *snd y*)) ∈ *matchrel*
⟨*proof*⟩

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

lemma *ne-matchers-imp-empty-or-matchrel*:
assumes *matchers-map σ* ∩ *matchers* (*set-mset P*) ≠ {}
shows *P* = {#} ∨ (∃ *y*. ((*P*, *σ*), *y*) ∈ *matchrel*)
⟨*proof*⟩

lemma *matchrel-imp-converse-matchless* [*dest*]:
(*x*, *y*) ∈ *matchrel* ⟹ (*y*, *x*) ∈ *matchless*
⟨*proof*⟩

lemma *ne-matchers-imp-empty*:
fixes *P* :: (('*f*, '*v*) *term* × ('*f*, '*w*) *term*) *multiset*

assumes *matchers-map* $\sigma \cap$ *matchers* (*set-mset* P) $\neq \{\}$
shows $\exists \sigma'.$ ((P, σ), ($\{\#\}$, σ')) $\in$ *matchrel**
⟨*proof*⟩

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

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

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

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

end

5 Unification

5.1 Unifiers

Definition and properties of (most general) unifiers

theory *Unifiers*
imports *Term*
begin

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

type-synonym ('f, 'v) *equation* $=$ ('f, 'v) *term* $\times$ ('f, 'v) *term*
type-synonym ('f, 'v) *equations* $=$ ('f, 'v) *equation set*

The set of unifiers for a given set of equations.

definition *unifiers* :: ('f, 'v) *equations* $\Rightarrow$ ('f, 'v) *subst set*
where

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

Check whether a set of equations is unifiable.

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

lemma *in-unifiersE* [*elim*]:
$[\![\sigma \in \mathit{unifiers}\ E;\ (\bigwedge e.\ e \in E \Longrightarrow \mathit{fst}\ e \cdot \sigma = \mathit{snd}\ e \cdot \sigma) \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

Applying a substitution to a set of equations.

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

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

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

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

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

5.1.1 Properties of sets of unifiers

lemma *unifiers-Un* [*simp*]:
unifiers $(s \cup t) = \mathit{unifiers}\ s \cap \mathit{unifiers}\ t$
⟨*proof*⟩

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

lemma *unifiers-insert*:
unifiers $(\mathit{insert}\ p\ t) = \{\sigma.\ \mathit{fst}\ p \cdot \sigma = \mathit{snd}\ p \cdot \sigma\} \cap \mathit{unifiers}\ t$
⟨*proof*⟩

lemma *unifiers-insert-ident* [*simp*]:
unifiers $(\mathit{insert}\ (t,\ t)\ E) = \mathit{unifiers}\ E$
⟨*proof*⟩

lemma *unifiers-insert-swap*:
unifiers $(\mathit{insert}\ (s,\ t)\ E) = \mathit{unifiers}\ (\mathit{insert}\ (t,\ s)\ E)$
⟨*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*]:
$\tau \in$ *unifiers* (*subst-set* σ *E*) $\longleftrightarrow$ $\sigma \circ_s \tau \in$ *unifiers E*
⟨*proof*⟩

lemma *unifiers-insert-VarD*:
shows $\sigma \in$ *unifiers* (*insert* (*Var x*, *t*) *E*) $\Longrightarrow$ *subst x t* $\circ_s$ $\sigma = \sigma$
and $\sigma \in$ *unifiers* (*insert* (*t*, *Var x*) *E*) $\Longrightarrow$ *subst x t* $\circ_s$ $\sigma = \sigma$
⟨*proof*⟩

lemma *unifiers-insert-Var-left*:
$\sigma \in$ *unifiers* (*insert* (*Var x*, *t*) *E*) $\Longrightarrow$ $\sigma \in$ *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* (λt. $t \cdot \sigma$) *ss* = *map* (λt. $t \cdot \sigma$) *ts*}
⟨*proof*⟩

lemma *unifiers-Fun* [*simp*]:
$\sigma \in$ *unifiers* {(*Fun f ss*, *Fun g ts*)} $\longleftrightarrow$
length ss = *length ts* $\wedge$ *f* = *g* $\wedge$ $\sigma \in$ *unifiers* (*set* (*zip ss ts*))
⟨*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 ∈ *vars-term t* $\Longrightarrow$ *t* ≠ *Var x* $\Longrightarrow$ *unifiers* (*insert* (*Var x*, *t*) *E*) = {}
⟨*proof*⟩

lemma *unifiers-occur-left-Fun*:
x ∈ ($\bigcup$ *t*∈*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 $\neg$ *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 $\sigma :: ('f,\ 'v)$ *subst*
assumes *unifiable* (*subst-set* $\sigma\ E$)
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* $\sigma\ E$
⟨*proof*⟩

lemma *is-mgu-insert-decomp* [*simp*]:
assumes *length ss* = *length ts*
shows *is-mgu* σ (*insert* (*Fun f ss*, *Fun f ts*) E) $\longleftrightarrow$
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* $x\ t \circ_s \sigma$) (*insert* $(Var\ x,\ t)\ E$) (**is** *is-mgu ?*σ *?E*$'$)
⟨*proof*⟩

lemma *is-imgu-imp-is-mgu*:
assumes *is-imgu* $\sigma\ E$
shows *is-mgu* $\sigma\ E$
⟨*proof*⟩

5.1.4 Properties of *is-imgu*

lemma *rename-subst-domain-range-preserves-is-imgu*:
fixes $E :: ('f, 'v)$ *equations* **and** $\mu\ \varrho :: ('f, 'v)$ *subst*
assumes *imgu-*μ: *is-imgu* $\mu\ E$ **and** *is-var-*ϱ: $\forall x.$ *is-Var* $(\varrho\ x)$ **and** *inj* ϱ
shows *is-imgu* (*rename-subst-domain-range* $\varrho\ \mu$) (*subst-set* $\varrho\ E$)
⟨*proof*⟩

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

lemma *union-commutes*:
$M + \{\#x\#\} + N = M + N + \{\#x\#\}$
$M + \mathit{mset}\ xs + N = M + N + \mathit{mset}\ xs$
⟨*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-mset* $(t, t)\ E)\ E$ |
decomp: $[\![\mathit{length}\ ss = \mathit{length}\ ts]\!] \Longrightarrow$
UNIF1 Var (*add-mset* (*Fun f ss*, *Fun f ts*) E) $(E + \mathit{mset}\ (\mathit{zip}\ ss\ ts))$ |
Var-left: $[\![x \notin \mathit{vars-term}\ t]\!] \Longrightarrow$

UNIF1 (*subst* x t) (*add-mset* (*Var* x, t) E) (*subst-mset* (*subst* x t) E) |
Var-right: $[\![x \notin$ *vars-term* $t]\!] \Longrightarrow$
UNIF1 (*subst* x t) (*add-mset* (t, *Var* x) E) (*subst-mset* (*subst* x t) E)

Relation version of *UNIF1* with implicit substitutions.

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

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

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

A termination order for *UNIF1*.

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

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

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

lemma *vars-mset-subset-size-mset-uniflessI* [*intro*]:
vars-mset $M \subseteq$ *vars-mset* $N \Longrightarrow$ *size-mset* $M <$ *size-mset* $N \Longrightarrow$ (M, N) $\in$ *unifless*
⟨*proof*⟩

lemma *vars-mset-psubset-uniflessI* [*intro*]:
vars-mset $M \subset$ *vars-mset* $N \Longrightarrow$ (M, N) $\in$ *unifless*
⟨*proof*⟩

lemma *UNIF1-unifless*:
assumes *UNIF1* σ E E'
shows (E', E) $\in$ *unifless*
⟨*proof*⟩

lemma *converse-unif-subset-unifless*:
$unif^{-1} \subseteq$ *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* σ ∩ *vars-mset E*′ = {}
⟨*proof*⟩

lemma *UNIF1-subst-domain-subset*:
assumes *UNIF1* σ *E E*′
shows *subst-domain* σ ⊆ *vars-mset E*
⟨*proof*⟩

lemma *UNIF-subst-domain-subset*:

assumes *UNIF ss E E′*
shows *subst-domain* (*compose ss*) ⊆ *vars-mset E*
⟨*proof*⟩

lemma *UNIF1-range-vars-subset*:
assumes *UNIF1 σ E E′*
shows *range-vars σ* ⊆ *vars-mset E*
⟨*proof*⟩

lemma *UNIF1-subst-domain-range-vars-Int*:
assumes *UNIF1 σ E E′*
shows *subst-domain σ* ∩ *range-vars σ* = {}
⟨*proof*⟩

lemma *UNIF-range-vars-subset*:
assumes *UNIF ss E E′*
shows *range-vars* (*compose ss*) ⊆ *vars-mset E*
⟨*proof*⟩

lemma *UNIF-subst-domain-range-vars-Int*:
assumes *UNIF ss E E′*
shows *subst-domain* (*compose ss*) ∩ *range-vars* (*compose ss*) = {}
⟨*proof*⟩

The inference rules generate idempotent substitutions.

lemma *UNIF-idemp*:
assumes *UNIF ss E E′*
shows *compose ss* ∘$_s$ *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′*) ∈ *unif*
shows ∃*σ*. (*E* + *M*, *E′* + *subst-mset σ M*) ∈ *unif*
⟨*proof*⟩

lemma *unif-rtrancl-mono*:
assumes (*E*, *E′*) ∈ *unif**
shows ∃*σ*. (*E* + *M*, *E′* + *subst-mset σ M*) ∈ *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-mgu* (*compose ss*) (*set-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-term t* $\Longrightarrow$ *UNIF1* (*subst x t*) $\{\#$(*Var x*, *t*)$\#\}$ $\{\#\}$
⟨*proof*⟩

lemma *UNIF1-singleton-Var-right* [*intro*]:
$x \notin$ *vars-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-term t* $\wedge$ *is-Fun t*) $\vee$
$(\exists f\ g\ ss\ ts.$ $e =$ (*Fun f ss*, *Fun g ts*) $\wedge$ ($f \neq g \vee$ *length ss* $\neq$ *length ts*))
⟨*proof*⟩

lemma *unifiable-imp-unif*:
assumes *unifiable* $\{e\}$
shows $\exists E'.$ $(\{\#e\#\}, E') \in$ *unif*
⟨*proof*⟩

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

shows *unifiable* (*set-mset E′*)
⟨*proof*⟩

lemma *unif-imp-converse-unifless* [*dest*]:
(*x*, *y*) ∈ *unif* ⟹ (*y*, *x*) ∈ *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*, {#}) ∈ *unif**
⟨*proof*⟩

lemma *unif-rtrancl-empty-imp-unifiable*:
assumes (*E*, {#}) ∈ *unif**
shows *unifiable* (*set-mset E*)
⟨*proof*⟩

lemma *not-unifiable-imp-not-empty-NF*:
assumes ¬ *unifiable* (*set-mset E*)
shows ∃ *E′*. *E′* ≠ {#} ∧ (*E*, *E′*) ∈ *unif*$^!$
⟨*proof*⟩

lemma *unif-rtrancl-preserves-unifiable*:
assumes (*E*, *E′*) ∈ *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*, {#}) ∉ *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 $x \in$ *vars-term* t *then None*
else unify (*subst-list* (*subst* x t) E) ((x, t) # bs))
| *unify* ((t, *Var* x) # E) bs =
(*if* $x \in$ *vars-term* t *then None*
else unify (*subst-list* (*subst* x t) E) ((x, t) # bs))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *unify-append-prefix-same*:
(∀ $e \in$ *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*:
(∀ $e \in$ *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** σ :: ('*f*, '*v*) *subst*
  assumes *t-eq-u*: *t* · σ = *u* · σ
  shows ∃ μ :: ('*f*, '*v*) *subst*. *Unification.mgu t u* = *Some* μ
⟨*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* · μ = *u* · μ
⟨*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* ∉ (⋃ *e*∈*E*. *vars-term* (*fst e*) ∪ *vars-term* (*snd e*))
shows *is-Var* (μ *x*)
⟨*proof*⟩

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

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

lemma *imgu-subst-domain-subset*:
fixes $\mu :: ('f, 'v)\ subst$ **and** $E :: ('f, 'v)\ equations$ **and** $Evars :: 'v\ set$
assumes *imgu-*μ: *is-imgu* μ E **and** *fin-E*: *finite* E
defines $Evars \equiv (\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$
shows *subst-domain* $\mu \subseteq Evars$
⟨*proof*⟩

lemma *imgu-range-vars-of-equations-vars-subset*:
fixes $\mu :: ('f, 'v)\ subst$ **and** $E :: ('f, 'v)\ equations$ **and** $Evars :: 'v\ set$
assumes *imgu-*μ: *is-imgu* μ E **and** *fin-E*: *finite* E
defines $Evars \equiv (\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$
shows $(\bigcup x \in Evars.\ vars\text{-}term\ (\mu\ x)) \subseteq Evars$
⟨*proof*⟩

lemma *imgu-range-vars-subset*:
fixes $\mu :: ('f, 'v)\ subst$ **and** $E :: ('f, 'v)\ equations$
assumes *imgu-*μ: *is-imgu* μ E **and** *fin-E*: *finite* E
shows *range-vars* $\mu \subseteq (\bigcup e \in E.\ vars\text{-}term\ (fst\ e) \cup vars\text{-}term\ (snd\ e))$
⟨*proof*⟩

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

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 $s \cdot the\text{-}mgu\ s\ t = t \cdot the\text{-}mgu\ s\ t \wedge \sigma = the\text{-}mgu\ s\ t \circ_s \sigma$
⟨*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 $\Rightarrow$ *None*
| *Some* γ $\Rightarrow$ *Some* ($\gamma \circ vu$, $\gamma \circ wu$))

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

lemma *mgu-var-disjoint-generic-complete*:
fixes σ :: ($'f$, $'v$, $'u$) *gsubst* **and** τ :: ($'f$, $'w$, $'u$) *gsubst*
and *vu* :: $'v \Rightarrow 'u$ **and** *wu*:: $'w \Rightarrow 'u$
assumes *inj*: *inj vu inj wu*
and *vwu*: *range vu* $\cap$ *range wu* = {}
and *unif-disj*: $s \cdot \sigma = t \cdot \tau$
shows $\exists \mu 1\ \mu 2\ \delta$. *mgu-var-disjoint-generic vu wu s t* = *Some* ($\mu 1$, $\mu 2$) $\wedge$
$\sigma = \mu 1 \circ_s \delta \wedge$
$\tau = \mu 2 \circ_s \delta \wedge$
$s \cdot \mu 1 = t \cdot \mu 2$
⟨*proof*⟩

abbreviation *mgu-var-disjoint-sum* $\equiv$ *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* (*γ1*, *γ2*) ⟹ $s \cdot \gamma 1 = t \cdot \gamma 2$
⟨*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* (*μ1*, *μ2*) ∧
$\sigma = \mu 1 \circ_s \delta \wedge$
$\tau = \mu 2 \circ_s \delta \wedge$
$s \cdot \mu 1 = t \cdot \mu 2$
⟨*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* ∨ *σ x* = *Some t then match-term-list P* (*σ* (*x* ↦ *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 *⋀p t. (p, t) ∈ set P ⟹ p · σ = 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* (γ*1*, γ*2*) ⟹ *s* · γ*1* = *t* · γ*2*
⟨*proof*⟩

lemma *mgu-vd-string-complete*:

fixes $\sigma :: ('f, string)\ subst$ **and** $\tau :: ('f, string)\ subst$
assumes $s \cdot \sigma = t \cdot \tau$
shows $\exists \mu 1\ \mu 2\ \delta.\ mgu\text{-}vd\text{-}string\ s\ t = Some\ (\mu 1, \mu 2) \wedge$
$\sigma = \mu 1 \circ_s \delta \wedge$
$\tau = \mu 2 \circ_s \delta \wedge$
$s \cdot \mu 1 = t \cdot \mu 2$
⟨*proof*⟩
end

7 Subsumption

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

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

consts
$SUBSUMESEQ :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\leq\cdot$ 50)
$SUBSUMES :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $<\cdot$ 50)
$LITSIM :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\doteq$ 50)

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

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

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

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

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

abbreviation *SUBSUMES-SET* ($\{<\cdot\}$)
where

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

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

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

adhoc-overloading
SUBSUMESEQ subsumeseq

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

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

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

lemma *litsim-refl* [*simp*]:
$s \doteq s$
⟨*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 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 term-subsumable.subsumes **and**
LITSIM 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 ($t \cdot \sigma$) ≥ *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* ∘ (λ*t*. *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 ($\{\rhd\}$) **and**
supt-pred ((-/ $\rhd$ -) [*56*, *56*] *55*) **and**
subt-pred (**infix** $\lhd$ *55*) **and**
supteq ($\{\trianglerighteq\}$) **and**
supteq-pred ((-/ $\trianglerighteq$ -) [*56*, *56*] *55*) **and**
subteq-pred (**infix** $\trianglelefteq$ *55*)

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

Quantifier syntax.

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

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

translations
∀ *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 ⊵ *t* ⟹ *t* ⊵ *u* ⟹ *s* ⊵ *u*
⟨*proof*⟩

Auxiliary lemmas about term size.

lemma *size-simp5*:
s ∈ *set ss* ⟹ *s* ⊳ *t* ⟹ *size t* < *size s* ⟹ *size t* < *Suc* (*size-list size ss*)
⟨*proof*⟩

lemma *size-simp6*:
s ∈ *set ss* ⟹ *s* ⊵ *t* ⟹ *size t* ≤ *size s* ⟹ *size t* ≤ *Suc* (*size-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 ∉ *set ss* (**is** *?t* ∉ *set ss*)
⟨*proof*⟩

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

lemma *supt-supteq-conv*: *s* ⊳ *t* = (*s* ⊵ *t* ∧ *s* ≠ *t*)
⟨*proof*⟩

lemma *supt-not-sym*: *s* ⊳ *t* ⟹ ¬ (*t* ⊳ *s*)
⟨*proof*⟩

lemma *supt-irrefl*[*iff*]: ¬ *t* ⊳ *t*
  ⟨*proof*⟩

lemma *irrefl-subt*: *irrefl* {⊲} ⟨*proof*⟩

lemma *supt-imp-supteq*: *s* ⊳ *t* ⟹ *s* ⊵ *t*
  ⟨*proof*⟩

lemma *supt-supteq-not-supteq*: *s* ⊳ *t* = (*s* ⊵ *t* ∧ ¬ (*t* ⊵ *s*))
  ⟨*proof*⟩

lemma *supteq-supt-conv*: (*s* ⊵ *t*) = (*s* ⊳ *t* ∨ *s* = *t*) ⟨*proof*⟩

lemma *supteq-antisym*:
  assumes *s* ⊵ *t* **and** *t* ⊵ *s* **shows** *s* = *t*
  ⟨*proof*⟩

The subterm relation is an order on terms.

interpretation *subterm*: *order* (⊴) (⊲)
  ⟨*proof*⟩

More transitivity rules.

lemma *supt-supteq-trans*[*trans*]:
  s ⊳ *t* ⟹ *t* ⊵ *u* ⟹ *s* ⊳ *u*
  ⟨*proof*⟩

lemma *supteq-supt-trans*[*trans*]:
  s ⊵ *t* ⟹ *t* ⊳ *u* ⟹ *s* ⊳ *u*
  ⟨*proof*⟩

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

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

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

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

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

The superterm relation is strongly normalizing.

lemma *SN-supt*:
SN $\{\rhd\}$
⟨*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*.

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

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

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

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

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

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

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

lemma *ctxt-ctxt-compose* [*simp*]: (*C* $\circ_c$ *D*)⟨*t*⟩ = *C*⟨*D*⟨*t*⟩⟩ ⟨*proof*⟩

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

Applying substitutions to contexts.

fun *subst-apply-ctxt* :: ($'f$, $'v$) *ctxt* ⇒ ($'f$, $'v$, $'w$) *gsubst* ⇒ ($'f$, $'w$) *ctxt* (**infixl** $\cdot_c$ *67*)

where
$\Box \cdot_c \sigma = \Box$ |
$(More\ f\ ss1\ D\ ss2) \cdot_c \sigma = More\ f\ (map\ (\lambda t.\ t \cdot \sigma)\ ss1)\ (D \cdot_c \sigma)\ (map\ (\lambda t.\ t \cdot \sigma)\ ss2)$

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

lemma *subst-compose-ctxt-compose-distrib* [*simp*]:
$(C \circ_c D) \cdot_c \sigma = (C \cdot_c \sigma) \circ_c (D \cdot_c \sigma)$
⟨*proof*⟩

lemma *ctxt-compose-subst-compose-distrib* [*simp*]:
$C \cdot_c (\sigma \circ_s \tau) = (C \cdot_c \sigma) \cdot_c \tau$
⟨*proof*⟩

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 $s \rhd t$ **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** $s \rhd t$
⟨*proof*⟩

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

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

lemma *supt-var*: $\neg\ (Var\ x \rhd u)$
⟨*proof*⟩

lemma *supt-const*: $\neg$ (*Fun f* [] $\rhd$ *u*)
⟨*proof*⟩

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

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

lemma *Fun-supt* [*elim*]:
assumes *Fun f ts* $\rhd$ *s* **obtains** *t* **where** *t* $\in$ *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* $\in$ *vars-ctxt C* $\Longrightarrow$ σ *x* = τ *x*) $\Longrightarrow$ *C* $\cdot_c$ σ = *C* $\cdot_c$ τ
⟨*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 sig* = (*'f* × *nat*) *set*

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

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

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

lemma *supteq-imp-funas-term-subset*[*simp*]:
assumes *s* $\trianglerighteq$ *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* ∪ $\bigcup$(*set* (*map funas-term* (*ss1* @ *ss2*)))

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

lemma *funas-term-subst*:
funas-term $(t \cdot \sigma)$ = *funas-term* $t \cup \bigcup$ (*funas-term* ‘ σ ‘ *vars-term* t)
⟨*proof*⟩

lemma *funas-ctxt-compose* [*simp*]:
funas-ctxt $(C \circ_c D)$ = *funas-ctxt* $C \cup$ *funas-ctxt* D
⟨*proof*⟩

lemma *funas-ctxt-subst* [*simp*]:
funas-ctxt $(C \cdot_c \sigma)$ = *funas-ctxt* $C \cup \bigcup$ (*funas-term* ‘ σ ‘ *vars-ctxt* C)
⟨*proof*⟩

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.