First-Order Rewriting

René Thiemann, Christian Sternagel, Christina Kirk (Kohl), Martin Avanzini, Bertram Felgenhauer, Julian Nagele, Thomas Sternagel, Sarah Winkler, and Akihisa Yamada

University of Innsbruck, Austria

April 13, 2025

Abstract

This entry, derived from the *Isabelle Formalization of Rewriting* (IsaFoR) [3], provides a formalized foundation for first-order term rewriting. This serves as the basis for the certifier CeTA, which is generated from IsaFoR and verifies termination, confluence, and complexity proofs for term rewrite systems (TRSs).

This formalization covers fundamental results for term rewriting, as presented in the foundational textbooks by Baader and Nipkow [1] and TeReSe [2]. These include:

- Various types of rewrite steps, such as root, ground, parallel, and multi-steps.
- Special cases of TRSs, such as linear and left-linear TRSs.
- A definition of critical pairs and key results, including the critical pair lemma.
- Orthogonality, notably that weak orthogonality implies confluence.
- Executable versions of relevant definitions, such as parallel and multi-step rewriting.

Contents

1 Introduction

A TRS, as formalized here, is defined as a binary relation over first-order terms. Given a TRS $\mathcal{R}$, a rule $(\ell, r) \in \mathcal{R}$ is typically written as $\ell \to r$. The rewrite relation induced by $\mathcal{R}$, denoted $\to_{\mathcal{R}}$, is defined as follows: a term s rewrites to a term t using the TRS $\mathcal{R}$ (i.e, $s \to_{\mathcal{R}} t$) if there are a context C, a substitution σ, and a rule $(\ell, r) \in \mathcal{R}$ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$.

The literature typically assumes two restrictions on the variables in a rule $\ell \to r$ of a TRS: ℓ must not be a variable, and all variables in r must appear in ℓ. However, many results in term rewriting do not depend on these conditions. In this entry, such constraints are enforced only where necessary. A TRS that meets these criteria is called a *well-formed* TRSs (*wf-trs*) in the formalization.

2 Preliminaries

This theory contains some auxiliary results previously located in Auxx.Util of IsaFoR.

theory *FOR-Preliminaries*
imports
Polynomial-Factorization.Missing-List
begin

lemma *in-set-idx*: $a \in set\ as \Longrightarrow \exists\, i.\ i < length\ as \wedge a = as\ !\ i$
⟨*proof*⟩

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 finite subsets of a set S can be turned into a compatible renaming (with finite domain) on S.

lemma *bij-betw-extend*:
assumes $*$: *bij-betw f A B*
and $A \subseteq S$
and $B \subseteq S$
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 S S
⟨*proof*⟩

lemma *concat-nth*:
assumes $m < length\ xs$ **and** $n < length\ (xs\ !\ m)$
and $i = sum\text{-}list\ (map\ length\ (take\ m\ xs)) + n$
shows $concat\ xs\ !\ i = xs\ !\ m\ !\ n$
⟨*proof*⟩

lemma *concat-nth-length*:
$i < length\ uss \Longrightarrow j < length\ (uss\ !\ i) \Longrightarrow$
$sum\text{-}list\ (map\ length\ (take\ i\ uss)) + j < length\ (concat\ uss)$
⟨*proof*⟩

lemma *less-length-concat*:
assumes $i < length\ (concat\ xs)$
shows $\exists\, m\ n.$
$i = sum\text{-}list\ (map\ length\ (take\ m\ xs)) + n \wedge$
$m < length\ xs \wedge n < length\ (xs\ !\ m) \wedge concat\ xs\ !\ i = xs\ !\ m\ !\ n$
⟨*proof*⟩

lemma *concat-remove-nth*:
assumes $i < length\ sss$
and $j < length\ (sss\ !\ i)$
defines $k \equiv sum\text{-}list\ (map\ length\ (take\ i\ sss)) + j$
shows $concat\ (take\ i\ sss\ @\ remove\text{-}nth\ j\ (sss\ !\ i)\ \#\ drop\ (Suc\ i)\ sss) = remove\text{-}nth\ k\ (concat\ sss)$
⟨*proof*⟩

lemma *nth-append-Cons*: $(xs \mathbin{@} y \# zs) ! i =$
$(\mathit{if}\ i < \mathit{length}\ xs\ \mathit{then}\ xs ! i\ \mathit{else}\ \mathit{if}\ i = \mathit{length}\ xs\ \mathit{then}\ y\ \mathit{else}\ zs ! (i - \mathit{Suc}\ (\mathit{length}\ xs)))$
⟨*proof*⟩

lemma *finite-imp-eq* [*simp*]:
$\mathit{finite}\ \{x.\ P\ x \longrightarrow Q\ x\} \longleftrightarrow \mathit{finite}\ \{x.\ \neg\ P\ x\} \wedge \mathit{finite}\ \{x.\ Q\ x\}$
⟨*proof*⟩

lemma *sum-list-take-eq*:
fixes $xs :: \mathit{nat\ list}$
shows $k < i \Longrightarrow i < \mathit{length}\ xs \Longrightarrow \mathit{sum\text{-}list}\ (\mathit{take}\ i\ xs) =$
$\mathit{sum\text{-}list}\ (\mathit{take}\ k\ xs) + xs ! k + \mathit{sum\text{-}list}\ (\mathit{take}\ (i - \mathit{Suc}\ k)\ (\mathit{drop}\ (\mathit{Suc}\ k)\ xs))$
⟨*proof*⟩

lemma *nth-equalityE*:
$xs = ys \Longrightarrow (\mathit{length}\ xs = \mathit{length}\ ys \Longrightarrow (\bigwedge i.\ i < \mathit{length}\ xs \Longrightarrow xs ! i = ys ! i) \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

fun *fold-map* :: $('a \Rightarrow 'b \Rightarrow 'c \times 'b) \Rightarrow 'a\ \mathit{list} \Rightarrow 'b \Rightarrow 'c\ \mathit{list} \times 'b$ **where**
$\mathit{fold\text{-}map}\ f\ []\ y = ([], y)$
| $\mathit{fold\text{-}map}\ f\ (x\#xs)\ y = (\mathit{case}\ f\ x\ y\ \mathit{of}$
$(x', y') \Rightarrow \mathit{case}\ \mathit{fold\text{-}map}\ f\ xs\ y'\ \mathit{of}$
$(xs', y'') \Rightarrow (x' \# xs', y''))$

lemma *fold-map-cong* [*fundef-cong*]:
assumes $a = b$ **and** $xs = ys$
and $\bigwedge x.\ x \in \mathit{set}\ xs \Longrightarrow f\ x = g\ x$
shows $\mathit{fold\text{-}map}\ f\ xs\ a = \mathit{fold\text{-}map}\ g\ ys\ b$
⟨*proof*⟩

lemma *fold-map-map-conv*:
assumes $\bigwedge x\ ys.\ x \in \mathit{set}\ xs \Longrightarrow f\ (g\ x)\ (g'\ x \mathbin{@} ys) = (h\ x, ys)$
shows $\mathit{fold\text{-}map}\ f\ (\mathit{map}\ g\ xs)\ (\mathit{concat}\ (\mathit{map}\ g'\ xs) \mathbin{@} ys) = (\mathit{map}\ h\ xs, ys)$
⟨*proof*⟩

lemma *map-fst-fold-map*:
$\mathit{map}\ f\ (\mathit{fst}\ (\mathit{fold\text{-}map}\ g\ xs\ y)) = \mathit{fst}\ (\mathit{fold\text{-}map}\ (\lambda a\ b.\ \mathit{apfst}\ f\ (g\ a\ b))\ xs\ y)$
⟨*proof*⟩

lemma *not-Nil-imp-last*: $xs \neq [] \Longrightarrow \exists\, ys\ y.\ xs = ys@[y]$
⟨*proof*⟩

lemma *Nil-or-last*: $xs = [] \vee (\exists\, ys\ y.\ xs = ys@[y])$
⟨*proof*⟩

2.1 Combinators

definition *const* :: $'a \Rightarrow 'b \Rightarrow 'a$ **where**
const $\equiv (\lambda x\ y.\ x)$

definition *flip* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'a \Rightarrow 'c)$ **where**
flip $f \equiv (\lambda x\ y.\ f\ y\ x)$
declare *flip-def*[*simp*]

lemma *const-apply*[*simp*]: *const* $x\ y = x$
⟨*proof*⟩

lemma *foldr-Cons-append-conv* [*simp*]:
foldr (#) *xs ys* = *xs* @ *ys*
⟨*proof*⟩

lemma *foldl-flip-Cons*[*simp*]:
foldl (*flip* (#)) *xs ys* = *rev ys* @ *xs*
⟨*proof*⟩

already present as *foldr-conv-foldl*, but direction seems odd

lemma *foldr-flip-rev*[*simp*]:
foldr (*flip f*) (*rev xs*) *a* = *foldl f a xs*
⟨*proof*⟩

already present as *foldl-conv-foldr*, but direction seems odd

lemma *foldl-flip-rev*[*simp*]:
foldl (*flip f*) *a* (*rev xs*) = *foldr f xs a*
⟨*proof*⟩

fun *debug* :: (*String.literal* ⇒ *String.literal*) ⇒ *String.literal* ⇒ $'a \Rightarrow 'a$ **where**
debug i t x = *x*

2.2 Distinct Lists and Partitions

This theory provides some auxiliary lemmas related to lists with distinct elements and partitions. This is mainly used for dealing with *linear* terms.

lemma *distinct-alt*:
assumes ∀ *x*. *length* (*filter* ((=) *x*) *xs*) ≤ *1*
shows *distinct xs*
⟨*proof*⟩

lemma *distinct-filter2*:
assumes ∀ *i* < *size xs*. ∀ *j* < *size xs*. *i* ≠ *j* ∧ *f* (*xs*!*i*) ∧ *f* (*xs*!*j*) ⟶ *xs*!*i* ≠ *xs*!*j*
shows *distinct* (*filter f xs*)
⟨*proof*⟩

lemma *distinct-is-partition*:
assumes *distinct xs*

shows *is-partition* (*map* (λ*x*. {*x*}) *xs*)
⟨*proof*⟩

lemma *is-partition-append*:
assumes *is-partition xs* **and** *is-partition zs*
and ∀ *i* < *length xs*. *xs*!*i* ∩ ⋃ (*set zs*) = {}
shows *is-partition* (*xs*@*zs*)
⟨*proof*⟩

lemma *distinct-is-partitition-sets*:
assumes *distinct xs*
and *xs* = *concat ys*
shows *is-partition* (*map set ys*)
⟨*proof*⟩

end

2.3 Option Type

theory *Option-Util*
imports *Main*
begin

primrec *option-to-list* :: *'a option* ⇒ *'a list*
where
option-to-list (*Some a*) = [*a*] |
option-to-list None = []

lemma *set-option-to-list-sound* [*simp*]:
set (*option-to-list t*) = *set-option t*
⟨*proof*⟩

fun *fun-of-map* :: (*'a* ⇒ *'b option*) ⇒ *'b* ⇒ (*'a* ⇒ *'b*) **where**
fun-of-map m d a = (*case m a of Some b* ⇒ *b* | *None* ⇒ *d*)

end

2.4 Sublists

theory *SubList*
imports
HOL−Library.Sublist
HOL−Library.Multiset
begin

lemmas *subseq-trans* = *subseq-order.order-trans*

lemma *subseq-Cons-Cons*:
assumes *subseq* (*a* # *as*) (*b* # *bs*)
shows *subseq as bs*

⟨*proof*⟩

lemma *subseq-induct2*:
⟦ *subseq xs ys*;
⋀ *bs*. *P* [] *bs*;
⋀ *a as bs*. ⟦ *subseq as bs*; *P as bs* ⟧ ⟹ *P* (*a* # *as*) (*a* # *bs*);
⋀ *a as b bs*. ⟦ *a* ≠ *b*; *subseq as bs*; *subseq* (*a* # *as*) *bs*; *P as bs*; *P* (*a* # *as*) *bs* ⟧
⟹ *P* (*a* # *as*) (*b* # *bs*) ⟧
⟹ *P xs ys*
⟨*proof*⟩

lemma *subseq-submultiset*:
subseq xs ys ⟹ *mset xs* ⊆# *mset ys*
⟨*proof*⟩

lemma *subseq-subset*:
subseq xs ys ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *remove1-subseq*:
subseq (*remove1 x xs*) *xs*
⟨*proof*⟩

lemma *subseq-concat*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *subseq* (*f x*) (*g x*)
shows *subseq* (*concat* (*map f xs*)) (*concat* (*map g xs*))
⟨*proof*⟩

end

3 Extensions for Existing AFP Entries

3.1 First Order Terms

theory *Term-Impl*
imports
First-Order-Terms.Term-More
Certification-Monads.Check-Monad
Deriving.Compare-Order-Instances
Show.Shows-Literal
FOR-Preliminaries
begin

derive *compare-order term*

3.1.1 Positions

fun *poss-list* :: ('*f*, '*v*) *term* ⇒ *pos list*
where

```
    poss-list (Var x) = [[]] |
    poss-list (Fun f ss) = ([] # concat (map (λ (i, ps).
    map ((#) i) ps) (zip [0 ..< length ss] (map poss-list ss))))

lemma poss-list-sound [simp]:
  set (poss-list s) = poss s
⟨proof⟩

declare poss-list.simps [simp del]

fun var-poss-list :: ('f, 'v) term ⇒ pos list
  where
    var-poss-list (Var x) = [[]] |
    var-poss-list (Fun f ss) = (concat (map (λ (i, ps).
    map ((#) i) ps) (zip [0 ..< length ss] (map var-poss-list ss))))


lemma var-poss-list-sound [simp]:
  set (var-poss-list s) = var-poss s
⟨proof⟩

lemma length-var-poss-list: length (var-poss-list t) = length (vars-term-list t)
⟨proof⟩

lemma vars-term-list-var-poss-list:
  assumes i < length (vars-term-list t)
  shows Var ((vars-term-list t)!i) = t|-((var-poss-list t)!i)
  ⟨proof⟩

lemma var-poss-list-map-vars-term:
  shows var-poss-list (map-vars-term f t) = var-poss-list t
⟨proof⟩

lemma distinct-var-poss-list:
  shows distinct (var-poss-list t)
⟨proof⟩

fun fun-poss-list :: ('f, 'v) term ⇒ pos list
  where
    fun-poss-list (Var x) = [] |
    fun-poss-list (Fun f ss) = ([] # concat (map (λ (i, ps).
    map ((#) i) ps) (zip [0 ..< length ss] (map fun-poss-list ss))))

lemma set-fun-poss-list [simp]:
  set (fun-poss-list t) = fun-poss t
  ⟨proof⟩

context
begin
```

private fun *in-poss* :: *pos* ⇒ ('*f*, '*v*) *term* ⇒ *bool*
where
in-poss [] - ⟷ *True* |
in-poss (*Cons i p*) (*Fun f ts*) ⟷ *i* < *length ts* ∧ *in-poss p* (*ts* ! *i*) |
in-poss (*Cons i p*) (*Var* -) ⟷ *False*

lemma *poss-code*[*code-unfold*]:
p ∈ *poss t* = *in-poss p t* ⟨*proof*⟩
end

3.1.2 List of Distinct Variables

We introduce a duplicate free version of *vars-term-list* that preserves order of appearance of variables. This is used for the theory on proof terms.

abbreviation *vars-distinct* :: ('*f*, '*v*) *term* ⇒ '*v list* **where** *vars-distinct t* ≡ (*rev* ∘ *remdups* ∘ *rev*) (*vars-term-list t*)

lemma *single-var*[*simp*]: *vars-distinct* (*Var x*) = [*x*]
⟨*proof*⟩

lemma *vars-term-list-vars-distinct*:
assumes *i* < *length* (*vars-term-list t*)
shows ∃*j* < *length* (*vars-distinct t*). (*vars-term-list t*)!*i* = (*vars-distinct t*)!*j*
⟨*proof*⟩

3.1.3 Useful abstractions

Given that we perform the same operations on terms in order to get a list of the variables and a list of the functions, we define functions that run through the term and perform these actions.

context
begin
private fun *contains-var-term* :: '*v* ⇒ ('*f*, '*v*) *term* ⇒ *bool* **where**
contains-var-term x (*Var y*) = (*x* = *y*)
| *contains-var-term x* (*Fun* - *ts*) = *Bex* (*set ts*) (*contains-var-term x*)

lemma *contains-var-term-sound*[*simp*]:
contains-var-term x t ⟷ *x* ∈ *vars-term t*
⟨*proof*⟩

lemma *in-vars-term-code*[*code-unfold*]: *x* ∈ *vars-term t* = *contains-var-term x t*
⟨*proof*⟩
end

3.1.4 Linear Terms

lemma *distinct-vars-linear-term*:

assumes *distinct* (*vars-term-list t*)
shows *linear-term t*
⟨*proof*⟩

fun
linear-term-impl :: *'v set* ⇒ (*'f*, *'v*) *term* ⇒ (*'v set*) *option*
where
linear-term-impl xs (*Var x*) = (*if x* ∈ *xs then None else Some* (*insert x xs*)) |
linear-term-impl xs (*Fun - []*) = *Some xs* |
linear-term-impl xs (*Fun f* (*t # ts*)) = (*case linear-term-impl xs t of*
None ⇒ *None*
| *Some ys* ⇒ *linear-term-impl ys* (*Fun f ts*))

lemma *linear-term-code*[*code*]: *linear-term t* = (*linear-term-impl* {} *t* ≠ *None*)
⟨*proof*⟩

definition *check-linear-term* :: (*'f* :: *showl*, *'v* :: *showl*) *term* ⇒ *showsl check*
where
check-linear-term s = *check* (*linear-term s*)
(*showsl* (*STR ''the term ''*) ∘ *showsl s* ∘ *showsl* (*STR '' is not linear*↩*''*))

lemma *check-linear-term* [*simp*]:
isOK (*check-linear-term s*) = *linear-term s*
⟨*proof*⟩

3.1.5 Subterms

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

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

lemma *supteq-list* [*simp*]:
set (*supteq-list t*) = {*s*. *t* ⊵ *s*}
⟨*proof*⟩

lemma *supt-list-sound* [*simp*]:
set (*supt-list t*) = {*s*. *t* ⊳ *s*}
⟨*proof*⟩

fun *supt-impl* :: (*'f*, *'v*) *term* ⇒ (*'f*, *'v*) *term* ⇒ *bool*
where
supt-impl (*Var x*) *t* ⟷ *False* |
supt-impl (*Fun f ss*) *t* ⟷ *t* ∈ *set ss* ∨ *Bex* (*set ss*) (λ*s*. *supt-impl s t*)

lemma *supt-impl* [*code-unfold*]:
  $s \rhd t \longleftrightarrow$ *supt-impl s t*
⟨*proof*⟩

lemma *supteq-impl*[*code-unfold*]: $s \trianglerighteq t \longleftrightarrow s = t \vee$ *supt-impl s t*
  ⟨*proof*⟩

definition *check-no-var* :: ('*f*::*showl*, '*v*::*showl*) *term* ⇒ *showsl check* **where**
  check-no-var t ≡ *check* (*is-Fun t*) (*showsl* (*STR* ''*variable found*↩''))

lemma *check-no-var-sound*[*simp*]:
  isOK (*check-no-var t*) ⟷ *is-Fun t*
  ⟨*proof*⟩

definition
  check-supt :: ('*f*::*showl*, '*v*::*showl*) *term* ⇒ ('*f*, '*v*) *term* ⇒ *showsl check*
  where
   check-supt s t ≡ *check* (*s* ⊳ *t*) (*showsl t* ∘ *showsl* (*STR* '' *is not a proper subterm of* '') ∘ *showsl s*)

definition
  check-supteq :: ('*f*::*showl*, '*v*::*showl*) *term* ⇒ ('*f*, '*v*) *term* ⇒ *showsl check*
  where
   check-supteq s t ≡ *check* (*s* ⊵ *t*) (*showsl t* ∘ *showsl* (*STR* '' *is not a subterm of* '') ∘ *showsl s*)

lemma *isOK-check-supt* [*simp*]:
  isOK (*check-supt s t*) ⟷ *s* ⊳ *t*
  ⟨*proof*⟩

lemma *isOK-check-supteq* [*simp*]:
  isOK (*check-supteq s t*) ⟷ *s* ⊵ *t*
  ⟨*proof*⟩

3.1.6 Additional Functions on Terms

fun *with-arity* :: ('*f*, '*v*) *term* ⇒ ('*f* × *nat*, '*v*) *term* **where**
  with-arity (*Var x*) = *Var x*
| *with-arity* (*Fun f ts*) = *Fun* (*f*, *length ts*) (*map with-arity ts*)

fun *add-vars-term* :: ('*f*, '*v*) *term* ⇒ '*v list* ⇒ '*v list*
  where
   add-vars-term (*Var x*) *xs* = *x* # *xs* |
   add-vars-term (*Fun - ts*) *xs* = *foldr add-vars-term ts xs*

fun *add-funs-term* :: ('*f*, '*v*) *term* ⇒ '*f list* ⇒ '*f list*
  where

add-funs-term (*Var* -) *fs* = *fs* |
add-funs-term (*Fun f ts*) *fs* = *f* # *foldr add-funs-term ts fs*

fun *add-funas-term* :: ('*f*, '*v*) *term* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
add-funas-term (*Var* -) *fs* = *fs* |
add-funas-term (*Fun f ts*) *fs* = (*f*, *length ts*) # *foldr add-funas-term ts fs*

definition *add-funas-args-term* :: ('*f*, '*v*) *term* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
add-funas-args-term t fs = *foldr add-funas-term* (*args t*) *fs*

lemma *add-vars-term-vars-term-list-conv* [*simp*]:
add-vars-term t xs = *vars-term-list t* @ *xs*
⟨*proof*⟩

lemma *add-funs-term-funs-term-list-conv* [*simp*]:
add-funs-term t fs = *funs-term-list t* @ *fs*
⟨*proof*⟩

lemma *add-funas-term-funas-term-list-conv* [*simp*]:
add-funas-term t fs = *funas-term-list t* @ *fs*
⟨*proof*⟩

lemma *add-vars-term-vars-term-list-abs-conv* [*simp*]:
add-vars-term = (@) ∘ *vars-term-list*
⟨*proof*⟩

lemma *add-funs-term-funs-term-list-abs-conv* [*simp*]:
add-funs-term = (@) ∘ *funs-term-list*
⟨*proof*⟩

lemma *add-funas-term-funas-term-list-abs-conv* [*simp*]:
add-funas-term = (@) ∘ *funas-term-list*
⟨*proof*⟩

lemma *add-funas-args-term-funas-args-term-list-conv* [*simp*]:
add-funas-args-term t fs = *funas-args-term-list t* @ *fs*
⟨*proof*⟩

fun *insert-vars-term* :: ('*f*, '*v*) *term* ⇒ '*v list* ⇒ '*v list*
where
insert-vars-term (*Var x*) *xs* = *List.insert x xs* |
insert-vars-term (*Fun f ts*) *xs* = *foldr insert-vars-term ts xs*

fun *insert-funs-term* :: ('*f*, '*v*) *term* ⇒ '*f list* ⇒ '*f list*
where
insert-funs-term (*Var x*) *fs* = *fs* |
insert-funs-term (*Fun f ts*) *fs* = *List.insert f* (*foldr insert-funs-term ts fs*)

fun *insert-funas-term* :: ('*f*, '*v*) *term* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
insert-funas-term (*Var x*) *fs* = *fs* |
insert-funas-term (*Fun f ts*) *fs* = *List.insert* (*f*, *length ts*) (*foldr insert-funas-term ts fs*)

definition *insert-funas-args-term* :: ('*f*, '*v*) *term* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
insert-funas-args-term t fs = *foldr insert-funas-term* (*args t*) *fs*

lemma *set-insert-vars-term-vars-term* [*simp*]:
set (*insert-vars-term t xs*) = *vars-term t* ∪ *set xs*
⟨*proof*⟩

lemma *set-insert-funs-term-funs-term* [*simp*]:
set (*insert-funs-term t fs*) = *funs-term t* ∪ *set fs*
⟨*proof*⟩

lemma *set-insert-funas-term-funas-term* [*simp*]:
set (*insert-funas-term t fs*) = *funas-term t* ∪ *set fs*
⟨*proof*⟩

lemma *set-insert-funas-args-term* [*simp*]:
set (*insert-funas-args-term t fs*) = ⋃ (*funas-term* ‘ *set* (*args t*)) ∪ *set fs*
⟨*proof*⟩

Implementations of corresponding set-based functions.

abbreviation *vars-term-impl t* ≡ *insert-vars-term t* []
abbreviation *funs-term-impl t* ≡ *insert-funs-term t* []
abbreviation *funas-term-impl t* ≡ *insert-funas-term t* []

lemma *vars-funs-term-list-code*[*code*]:
vars-term-list t = *add-vars-term t* []
funs-term-list t = *add-funs-term t* []
⟨*proof*⟩

lemma *with-arity-term-fold* [*code*]:
with-arity = *Term-More.fold Var* (λ*f ts*. *Fun* (*f*, *length ts*) *ts*)
⟨*proof*⟩

fun *flatten-term-enum* :: ('*f list*, '*v*) *term* ⇒ ('*f*, '*v*) *term list*
where
flatten-term-enum (*Var x*) = [*Var x*] |
flatten-term-enum (*Fun fs ts*) =
(*let*

lts = *map flatten-term-enum ts*;
ss = *concat-lists lts*
in concat (*map* (λ *f*. *map* (*Fun f*) *ss*) *fs*))

lemma *flatten-term-enum*:
set (*flatten-term-enum t*) = {*u*. *instance-term u* (*map-funs-term set t*)}
⟨*proof*⟩

definition *check-ground-term* :: ('*f* :: *showl*, '*v* :: *showl*) *term* ⇒ *showsl check*
where
check-ground-term s = *check* (*ground s*)
(*showsl* (*STR* ''*the term* '') ∘ *showsl s* ∘ *showsl* (*STR* '' *is not a ground term*↩'')

lemma *check-ground-term* [*simp*]:
isOK (*check-ground-term s*) ⟷ *ground s*
⟨*proof*⟩

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

fun *check-funas-term* :: '*f* :: *showl sig* ⇒ ('*f*,'*v* :: *showl*)*term* ⇒ *showsl check* **where**
check-funas-term F (*Fun f ts*) = *do* {
check ((*f*, *length ts*) ∈ *F*) (*showsl* (*Fun f ts*)
o showsl-lit (*STR* ''*problem*: *root of subterm* '') *o showsl f o showsl-lit* (*STR* '' *not in signature*↩''));
check-allm (*check-funas-term F*) *ts*
}
| *check-funas-term F* (*Var* -) = *return* ()

lemma *check-funas-term*[*simp*]: *isOK*(*check-funas-term F t*) = (*funas-term t* ⊆ *F*)
⟨*proof*⟩

3.1.7 Substitutions

definition *mk-subst-domain* :: ('*f*, '*v*) *substL* ⇒ ('*v* × ('*f*, '*v*) *term*) *list* **where**
mk-subst-domain σ ≡
let τ = *mk-subst Var* σ *in*
(*filter* (λ(*x*, *t*). *Var x* ≠ *t*) (*map* (λ *x*. (*x*, τ *x*)) (*remdups* (*map fst* σ))))

lemma *mk-subst-domain*:
set (*mk-subst-domain* σ) = (λ *x*. (*x*, *mk-subst Var* σ *x*)) ' *subst-domain* (*mk-subst Var* σ)
(**is** *?I* = *?R*)
⟨*proof*⟩

lemma *finite-mk-subst*: *finite* (*subst-domain* (*mk-subst Var* σ))
⟨*proof*⟩

definition *subst-eq* :: ('*f*, '*v*) *substL* ⇒ ('*f*, '*v*) *substL* ⇒ *bool* **where**
subst-eq σ τ = (*let* σ′ = *mk-subst-domain* σ; τ′ = *mk-subst-domain* τ *in set* σ′ = *set* τ′)

lemma *subst-eq* [*simp*]:
subst-eq σ τ = (*mk-subst Var* σ = *mk-subst Var* τ)
⟨*proof*⟩

definition *range-vars-impl* :: ('*f*, '*v*) *substL* ⇒ '*v list*
where
range-vars-impl σ =
(*let* σ′ = *mk-subst-domain* σ *in*
concat (*map* (*vars-term-list o snd*) σ′))

definition *vars-subst-impl* :: ('*f*, '*v*) *substL* ⇒ '*v list*
where
vars-subst-impl σ =
(*let* σ′ = *mk-subst-domain* σ *in*
map fst σ′ @ *concat* (*map* (*vars-term-list o snd*) σ′))

lemma *vars-subst-impl* [*simp*]:
set (*vars-subst-impl* σ) = *vars-subst* (*mk-subst Var* σ)
⟨*proof*⟩

lemma *range-vars-impl* [*simp*]:
set (*range-vars-impl* σ) = *range-vars* (*mk-subst Var* σ)
⟨*proof*⟩

lemma *mk-subst-one* [*simp*]: *mk-subst Var* [(*x*, *t*)] = *subst x t*
⟨*proof*⟩

lemma *fst-image* [*simp*]: *fst* ‘ (λ *x*. (*x*,*g x*)) ‘ *a* = *a* ⟨*proof*⟩

definition
subst-compose-impl :: ('*f*, '*v*) *substL* ⇒ ('*f*, '*v*) *substL* ⇒ ('*f*, '*v*) *substL*
where
subst-compose-impl σ τ ≡
let
σ′ = *mk-subst-domain* σ;
τ′ = *mk-subst-domain* τ;
dσ = *map fst* σ′
in map (λ (*x*, *t*). (*x*, *t* · *mk-subst Var* τ′)) σ′ @ *filter* (λ (*x*, *t*). *x* ∉ *set dσ*) τ′

lemma *mk-subst-mk-subst-domain* [*simp*]:
mk-subst Var (*mk-subst-domain* σ) = *mk-subst Var* σ
⟨*proof*⟩

lemma *subst-compose-impl* [*simp*]:
mk-subst Var (*subst-compose-impl* σ τ) = *mk-subst Var* σ ∘$_s$ *mk-subst Var* τ (**is**

?l = ?r)
⟨*proof*⟩

fun *subst-power-impl* :: ('*f*, '*v*) *substL* ⇒ *nat* ⇒ ('*f*, '*v*) *substL* **where**
  subst-power-impl σ 0 = []
| *subst-power-impl σ* (*Suc n*) = *subst-compose-impl σ* (*subst-power-impl σ n*)

lemma *subst-power-impl* [*simp*]:
  mk-subst Var (*subst-power-impl σ n*) = (*mk-subst Var σ*)^^*n*
  ⟨*proof*⟩

definition *commutes-impl* :: ('*f*, '*v*) *substL* ⇒ ('*f*, '*v*) *substL* ⇒ *bool* **where**
  commutes-impl σ μ ≡ *subst-eq* (*subst-compose-impl σ μ*) (*subst-compose-impl μ σ*)

lemma *commutes-impl* [*simp*]:
  commutes-impl σ μ = ((*mk-subst Var σ* ∘$_s$ *mk-subst Var μ*) = (*mk-subst Var μ* ∘$_s$ *mk-subst Var σ*))
  ⟨*proof*⟩

definition
  subst-compose'-impl :: ('*f*, '*v*) *substL* ⇒ ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *substL*
  where
    subst-compose'-impl σ ϱ ≡ *map* (λ (*x*, *s*). (*x*, *s* · *ϱ*)) (*mk-subst-domain σ*)

lemma *subst-compose'-impl* [*simp*]:
  mk-subst Var (*subst-compose'-impl σ ϱ*) = *subst-compose'* (*mk-subst Var σ*) *ϱ* (**is** *?l = ?r*)
⟨*proof*⟩

definition
  subst-replace-impl :: ('*f*, '*v*) *substL* ⇒ '*v* ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *substL*
  where
    subst-replace-impl σ x t ≡ (*x*, *t*) # *filter* (λ (*y*, *t*). *y* ≠ *x*) *σ*

lemma *subst-replace-impl* [*simp*]:
  mk-subst Var (*subst-replace-impl σ x t*) = (λ *y*. *if x = y then t else mk-subst Var σ y*) (**is** *?l = ?r*)
⟨*proof*⟩

lemma *mk-subst-domain-distinct*: *distinct* (*map fst* (*mk-subst-domain σ*))
  ⟨*proof*⟩

definition *is-renaming-impl* :: ('*f*,'*v*) *substL* ⇒ *bool* **where**
  is-renaming-impl σ ≡
    let σ' = *map snd* (*mk-subst-domain σ*) *in*
    (∀ *t* ∈ *set σ'*. *is-Var t*) ∧ *distinct σ'*

lemma *is-renaming-impl* [*simp*]:
is-renaming-impl σ = *is-renaming* (*mk-subst Var* σ) (**is** *?l* = *?r*)
⟨*proof*⟩

definition *is-inverse-renaming-impl* :: ('*f*, '*v*) *substL* ⇒ ('*f*, '*v*) *substL* **where**
is-inverse-renaming-impl σ ≡
let σ' = *mk-subst-domain* σ *in*
map (λ (*x*, *y*). (*the-Var y*, *Var x*)) σ'

lemma *is-inverse-renaming-impl* [*simp*]:
fixes σ :: ('*f*, '*v*) *substL*
assumes *var*: *is-renaming* (*mk-subst Var* σ)
shows *mk-subst Var* (*is-inverse-renaming-impl* σ) = *is-inverse-renaming* (*mk-subst Var* σ) (**is** *?l* = *?r*)
⟨*proof*⟩

definition
mk-subst-case :: '*v list* ⇒ ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *substL* ⇒ ('*f*, '*v*) *substL*
where
mk-subst-case xs σ τ = *subst-compose-impl* (*map* (λ *x*. (*x*, σ *x*)) *xs*) τ

lemma *mk-subst-case* [*simp*]:
mk-subst Var (*mk-subst-case xs* σ τ) =
(λ *x*. *if x* ∈ *set xs then* σ *x* · *mk-subst Var* τ *else mk-subst Var* τ *x*)
⟨*proof*⟩

end

3.1.8 A Concrete Unification Algorithm

theory *Unification-More*
imports
First-Order-Terms.Unification
First-Order-Rewriting.Term-Impl
begin

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

lemma *mgu-var-disjoint-right*:
fixes *s t* :: ('*f*, '*v*) *term* **and** σ τ :: ('*f*, '*v*) *subst* **and** *T*
assumes *s*: *vars-term s* ⊆ *S*
and *inj*: *inj T*
and *ST*: *S* ∩ *range T* = {}
and *id*: *s* · σ = *t* · τ
shows ∃ μ δ. *mgu s* (*map-vars-term T t*) = *Some* μ ∧
s · σ = *s* · μ · δ ∧
(∀ *t*::('*f*, '*v*) *term*. *t* · τ = *map-vars-term T t* · μ · δ) ∧

$(\forall\, x{\in}S.\ \sigma\ x = \mu\ x \cdot \delta)$
⟨*proof*⟩

abbreviation (*input*) *x-var* :: *string* ⇒ *string* **where** *x-var* ≡ *Cons* (*CHR* ''*x*'')
abbreviation (*input*) *y-var* :: *string* ⇒ *string* **where** *y-var* ≡ *Cons* (*CHR* ''*y*'')
abbreviation (*input*) *z-var* :: *string* ⇒ *string* **where** *z-var* ≡ *Cons* (*CHR* ''*z*'')

lemma *mgu-var-disjoint-right-string*:
  fixes *s t* :: ('*f*, *string*) *term* **and** *σ τ* :: ('*f*, *string*) *subst*
  assumes *s*: *vars-term s* ⊆ *range x-var* ∪ *range z-var*
    and *id*: *s* · *σ* = *t* · *τ*
  shows ∃ *μ δ*. *mgu s* (*map-vars-term y-var t*) = *Some μ* ∧
    s · *σ* = *s* · *μ* · *δ* ∧ (∀ *t*::('*f*, *string*) *term*. *t* · *τ* = *map-vars-term y-var t* · *μ* · *δ*)
∧
    $(\forall\, x \in \mathit{range\ x\text{-}var} \cup \mathit{range\ z\text{-}var}.\ \sigma\ x = \mu\ x \cdot \delta)$
⟨*proof*⟩

lemma *not-elem-subst-of*:
  assumes *x* ∉ *set* (*map fst xs*)
  shows (*subst-of xs*) *x* = *Var x*
  ⟨*proof*⟩

lemma *subst-of-id*:
  assumes ⋀*s*. *s* ∈ (*set ss*) ⟶ (∃ *x t*. *s* = (*x*, *t*) ∧ *t* = *Var x*)
  shows *subst-of ss* = *Var*
  ⟨*proof*⟩

lemma *subst-of-apply*:
  assumes (*x*, *t*) ∈ *set ss*
    and ∀ (*y*,*s*) ∈ *set ss*. (*y* = *x* ⟶ *s* = *t*)
    and *set* (*map fst ss*) ∩ *vars-term t* = {}
  shows *subst-of ss x* = *t*
  ⟨*proof*⟩

lemma *unify-equation-same*:
  assumes *fst e* = *snd e*
  shows *unify* (*E1*@*e*#*E2*) *ys* = *unify* (*E1*@*E2*) *ys*
  ⟨*proof*⟩

lemma *unify-filter-same*:
  shows *unify* (*filter* (λ*e*. *fst e* ≠ *snd e*) *E*) *ys* = *unify E ys*
⟨*proof*⟩

lemma *unify-ctxt-same*:
  shows *unify* ((*C*⟨*s*⟩, *C*⟨*t*⟩)#*xs*) *ys* = *unify* ((*s*, *t*)#*xs*) *ys*
⟨*proof*⟩

3.1.9 Unification of Linear and variable disjoint terms

definition *left-substs* :: ('*f*, '*v*) *term* ⇒ ('*f*, '*w*) *term* ⇒ ('*v* × ('*f*, '*w*) *term*) *list*
  where *left-substs s t* = (*let filtered-vars* = *filter* (λ(-, *p*). *p* ∈ *poss t*) (*zip* (*vars-term-list s*)(*var-poss-list s*))
    in map (λ(*x*, *p*). (*x*, *t*|-*p*)) *filtered-vars*)

definition *right-substs* :: ('*f*, '*v*) *term* ⇒ ('*f*, '*w*) *term* ⇒ ('*w* × ('*f*, '*v*) *term*) *list*
  where *right-substs s t* = (*let filtered-vars* = *filter* (λ(-, *q*). *q* ∈ *fun-poss s*) (*zip* (*vars-term-list t*)(*var-poss-list t*))
    in map (λ(*y*, *q*). (*y*, *s*|-*q*)) *filtered-vars*)

abbreviation *linear-unifier s t* ≡ *subst-of* ((*left-substs s t*) @ (*right-substs s t*))

lemma *left-substs-imp-props*:
  assumes (*x*, *u*) ∈ *set* (*left-substs s t*)
  shows ∃ *p*. *p* ∈ *poss s* ∧ *s*|-*p* = *Var x* ∧ *p* ∈ *poss t* ∧ *t*|-*p* = *u*
⟨*proof*⟩

lemma *props-imp-left-substs*:
  assumes *p* ∈ *poss s* **and** *s*|-*p* = *Var x* **and** *p* ∈ *poss t* **and** *t*|-*p* = *u*
  shows (*x*, *u*) ∈ *set* (*left-substs s t*)
⟨*proof*⟩

lemma *right-substs-imp-props*:
  assumes (*x*, *u*) ∈ *set* (*right-substs s t*)
  shows ∃ *q*. *q* ∈ *fun-poss s* ∧ *s*|-*q* = *u* ∧ *q* ∈ *poss t* ∧ *t*|-*q* = *Var x*
⟨*proof*⟩

lemma *props-imp-right-substs*:
  assumes *q* ∈ *fun-poss s* **and** *s*|-*q* = *u* **and** *q* ∈ *poss t* **and** *t*|-*q* = *Var x*
  shows (*x*, *u*) ∈ *set* (*right-substs s t*)
⟨*proof*⟩

lemma *map-fst-left-substs*:
  set (*map fst* (*left-substs s t*)) ⊆ *vars-term s*
  ⟨*proof*⟩

lemma *map-snd-left-substs*:
  assumes *t'* ∈ *set* (*map snd* (*left-substs s t*))
  shows *vars-term t'* ⊆ *vars-term t*
⟨*proof*⟩

lemma *map-fst-right-substs*:
  set (*map fst* (*right-substs s t*)) ⊆ *vars-term t*
  ⟨*proof*⟩

lemma *map-snd-right-substs*:
  assumes *t'* ∈ *set* (*map snd* (*right-substs s t*))

shows *vars-term* $t' \subseteq$ *vars-term* s
⟨*proof*⟩

lemma *distinct-map-fst-left-substs*:
assumes *linear-term* t
shows *distinct* (*map fst* (*left-substs* t s))
⟨*proof*⟩

lemma *distinct-map-fst-right-substs*:
assumes *linear-term* t
shows *distinct* (*map fst* (*right-substs* s t))
⟨*proof*⟩

lemma *is-partition-map-snd-left-substs*:
assumes *linear-term* s *linear-term* t
shows *is-partition* (*map* (*vars-term* ∘ *snd*) (*left-substs* t s))
⟨*proof*⟩

lemma *is-partition-map-snd-right-substs*:
assumes *linear-term* s *linear-term* t
shows *is-partition* (*map* (*vars-term* ∘ *snd*) (*right-substs* t s))
⟨*proof*⟩

lemma *distinct-fst-lsubsts-snd-rsubsts*:
assumes *linear-term* s
shows (*set* (*map fst* (*left-substs* s t))) ∩ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) (*right-substs* s t))) = {}
⟨*proof*⟩

lemma *distinct-fst-rsubsts-snd-lsubsts*:
assumes *linear-term* t
shows (*set* (*map fst* (*right-substs* s t))) ∩ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) (*left-substs* s t))) = {}
⟨*proof*⟩

lemma *linear-unifier-same*:
shows (*linear-unifier* t t) = *Var*
⟨*proof*⟩

lemma *linear-unifier-var1*:
shows *linear-unifier* (*Var* x) t = *subst* x t
⟨*proof*⟩

lemma *linear-unifier-var2*:
shows *linear-unifier* (*Fun* f *ts*) (*Var* x) = *subst* x (*Fun* f *ts*)
⟨*proof*⟩

lemma *linear-unifier-id*:
assumes $x \notin$ *vars-term* s **and** $x \notin$ *vars-term* t

shows (*linear-unifier s t*) *x* = *Var x*
⟨*proof*⟩

lemma *vars-subst-of*:
vars-subst (*subst-of ts*) ⊆ *set* (*map fst ts*) ∪ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) *ts*))
⟨*proof*⟩

lemma *vars-subst-linear-unifier*: *vars-subst* (*linear-unifier s t*) ⊆ *vars-term s* ∪ *vars-term t*
⟨*proof*⟩

lemma *decompose-is-partition-vars-subst*:
assumes *lin*:*linear-term* (*Fun f ss*) *linear-term* (*Fun g ts*)
and *disj*:*vars-term* (*Fun f ss*) ∩ *vars-term* (*Fun g ts*) = {}
and *ds*:*decompose* (*Fun f ss*) (*Fun g ts*) = *Some ds*
shows *is-partition* (*map vars-subst* (*map* (λ(*s*,*t*). *linear-unifier s t*) *ds*))
⟨*proof*⟩

lemma *compose-exists-subst*:
assumes *compose* σ*s x* ≠ *Var x*
shows ∃ *i* < *length* σ*s*. (∀ *j* < *i*. (σ*s*!*j*) *x* = *Var x*) ∧ (σ*s*!*i*) *x* ≠ *Var x*
⟨*proof*⟩

lemma *subst-of-exists-binding*:
assumes *subst-of xs y* ≠ *Var y*
shows ∃ *i* < *length xs*. *fst* (*xs*!*i*) = *y* ∧ (∀ *x* ∈ *set* (*drop* (*i*+*1*) *xs*). *fst x* ≠ *y*)
⟨*proof*⟩

lemma *linear-unifier-obtain-binding*:
assumes *disj*:*vars-term s* ∩ *vars-term t* = {} **and** *lin-s*:*linear-term s* **and** *lin-t*:*linear-term t*
and *u*:(*linear-unifier s t*) *x* = *u u* ≠ *Var x*
shows (*x* ∈ *vars-term s* ∧ (*x*,*u*) ∈ *set* (*left-substs s t*)) ∨ (*x* ∈ *vars-term t* ∧ (*x*,*u*) ∈ *set* (*right-substs s t*))
⟨*proof*⟩

connection between *left-substs* and *right-substs* and decomposition of functions

lemma *decompose-left-substs*:
assumes *decompose* (*Fun f ss*) (*Fun g ts*) = *Some ds*
shows *set* (*left-substs* (*Fun f ss*) (*Fun g ts*)) = (⋃ *e*∈*set ds*. *set* (*left-substs* (*fst e*) (*snd e*))) (**is** *?left* = *?right*)
⟨*proof*⟩

lemma *decompose-right-substs*:
assumes *decompose* (*Fun f ss*) (*Fun g ts*) = *Some ds*
shows *set* (*right-substs* (*Fun f ss*) (*Fun g ts*)) = (⋃ *e*∈*set ds*. *set* (*right-substs* (*fst e*) (*snd e*))) (**is** *?left* = *?right*)
⟨*proof*⟩

lemma *subst-compose-id*:
assumes $\bigwedge\tau.\ \tau \in set\ \tau s \Longrightarrow t \cdot \tau = t$
shows $t \cdot (compose\ \tau s) = t$
⟨*proof*⟩

lemma *subst-compose-distinct-vars*:
assumes $\sigma = compose\ \tau s$ **and** *part*:*is-partition* $(map\ vars\text{-}subst\ \tau s)$
and $\tau i{:}\tau i \in set\ \tau s$ **and** $s{:}\tau i\ x = s\ s \neq Var\ x$
shows $\sigma\ x = s$
⟨*proof*⟩

lemma *subst-id-compose*:
assumes $\sigma = compose\ \tau s$ **and** *part*:*is-partition* $(map\ vars\text{-}subst\ \tau s)$
and $t \cdot \sigma = t$
and $\tau \in set\ \tau s$
shows $t \cdot \tau = t$
⟨*proof*⟩

lemma *compose-subst-of*:
assumes $set\ ss = \bigcup\ (set\ `\ set\ ss')$
and *is-partition* $(map\ (vars\text{-}term \circ snd)\ ss)$ **and** *distinct* $(map\ fst\ ss)$
and $set\ (map\ fst\ ss) \cap \bigcup\ (set\ (map\ (vars\text{-}term \circ snd)\ ss)) = \{\}$
and *is-partition* $(map\ vars\text{-}subst\ (map\ subst\text{-}of\ ss'))$
shows $subst\text{-}of\ ss = compose\ (map\ subst\text{-}of\ ss')$ (**is** $?\sigma = ?\tau$)
⟨*proof*⟩

lemma *linear-term-decompose-subst-id*:
assumes *lin*:*linear-term* $(Fun\ f\ ss)$ *linear-term* $(Fun\ g\ ts)$
and *disj*:*vars-term* $(Fun\ f\ ss) \cap vars\text{-}term\ (Fun\ g\ ts) = \{\}$
and *decompose* $(Fun\ f\ ss)\ (Fun\ g\ ts) = Some\ ds$
and $i{:}i < length\ ds$ **and** $\sigma{:}\sigma = linear\text{-}unifier\ (fst\ (ds!i))\ (snd\ (ds!i))$
and $j{:}j < length\ ds\ j \neq i$
shows $fst\ (ds!j) \cdot \sigma = fst\ (ds!j) \wedge snd\ (ds!j) \cdot \sigma = snd\ (ds!j)$
⟨*proof*⟩

lemma *linear-unifier-decompose*:
assumes *linear-term* $(Fun\ f\ ss)$ *linear-term* $(Fun\ g\ ts)$
and *disj*:*vars-term* $(Fun\ f\ ss) \cap vars\text{-}term\ (Fun\ g\ ts) = \{\}$
and *ds*:*decompose* $(Fun\ f\ ss)\ (Fun\ g\ ts) = Some\ ds$
shows *linear-unifier* $(Fun\ f\ ss)\ (Fun\ g\ ts) = compose\ (map\ (\lambda(s,t).\ linear\text{-}unifier\ s\ t)\ ds)$
⟨*proof*⟩

Main lemma: for a list of unifiable terms that are linear and have distinct variables, the unification algorithm yields the same result as composing the list of substitutions obtained by *linear-unifier*.

lemma *unify-linear-terms*:
assumes *unify es substs* = *Some res*

and *compose* (*subst-of substs* # (*map* (λ(*s*,*t*). *linear-unifier s t*) *es*)) = τ
and ∀ *t* ∈ *set* (*map fst es*) ∪ *set* (*map snd es*). *linear-term t*
and ⋀*i j* σ. *i* < *j* ⟹ *j* < *length es* ⟹ σ = *linear-unifier* (*fst* (*es*!*i*)) (*snd* (*es*!*i*)) ⟹
(*fst* (*es*!*j*)) · σ = *fst* (*es*!*j*) ∧ (*snd* (*es*!*j*)) · σ = *snd* (*es*!*j*)
and ⋀*i*. *i* < *length es* ⟹ *vars-term* (*fst* (*es*!*i*)) ∩ *vars-term* (*snd* (*es*!*i*)) = {}
shows *subst-of res* = τ
⟨*proof*⟩

lemma *mgu-distinct-vars-term-list*:
assumes *unif*:*unifiers* {(*s*, *t*)} ≠ {}
and *distinct*:*distinct* ((*vars-term-list s*) @ (*vars-term-list t*))
shows *mgu s t* = *Some* (*linear-unifier s t*)
⟨*proof*⟩

end

3.1.10 Sets of Unifiers

theory *Unifiers-More*
imports
First-Order-Terms.Term-More
First-Order-Terms.Unifiers
begin

lemma *is-mguI*:
fixes σ :: ('*f*, '*v*) *subst*
assumes ∀ (*s*, *t*) ∈ *E*. *s* · σ = *t* · σ
and ⋀τ :: ('*f*, '*v*) *subst*. ∀ (*s*, *t*) ∈ *E*. *s* · τ = *t* · τ ⟹ ∃ γ :: ('*f*, '*v*) *subst*. τ = σ ∘$_s$ γ
shows *is-mgu* σ *E*
⟨*proof*⟩

lemma *subst-set-insert* [*simp*]:
subst-set σ (*insert e E*) = *insert* (*fst e* · σ, *snd e* · σ) (*subst-set* σ *E*)
⟨*proof*⟩

lemma *unifiable-UnD* [*dest*]:
unifiable (*M* ∪ *N*) ⟹ *unifiable M* ∧ *unifiable N*
⟨*proof*⟩

lemma *supt-imp-not-unifiable*:
assumes *s* ⊳ *t*
shows ¬ *unifiable* {(*t*, *s*)}
⟨*proof*⟩

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

lemma *unifiers-Int1* [*simp*]:
(*s*, *t*) ∈ *E* ⟹ *unifiers* {(*s*, *t*)} ∩ *unifiers E* = *unifiers E*
⟨*proof*⟩

lemma *imgu-linear-var-disjoint*:
assumes *is-imgu* σ {(*l2* |- *p*, *l1*)}
and *p* ∈ *poss l2*
and *linear-term l2*
and *vars-term l1* ∩ *vars-term l2* = {}
and *q* ∈ *poss l2*
and *parallel-pos p q*
shows *l2* |- *q* = *l2* |- *q* · σ
⟨*proof*⟩

end

3.2 Abstract Rewriting

theory *Abstract-Rewriting-Impl*
imports
Abstract−Rewriting.Abstract-Rewriting
begin

partial-function (*option*) *compute-NF* :: ('*a* ⇒ '*a option*) ⇒ '*a* ⇒ '*a option*
where [*simp*,*code*]: *compute-NF f a* = (*case f a of None* ⇒ *Some a* | *Some b* ⇒ *compute-NF f b*)

lemma *compute-NF-sound*: **assumes** *res*: *compute-NF f a* = *Some b*
and *f-sound*: ⋀ *a b*. *f a* = *Some b* ⟹ (*a*,*b*) ∈ *r*
shows (*a*,*b*) ∈ *r*^*
⟨*proof*⟩

lemma *compute-NF-complete*: **assumes** *res*: *compute-NF f a* = *Some b*
and *f-complete*: ⋀ *a*. *f a* = *None* ⟹ *a* ∈ *NF r*
shows *b* ∈ *NF r*
⟨*proof*⟩

lemma *compute-NF-SN*: **assumes** *SN*: *SN r*
and *f-sound*: ⋀ *a b*. *f a* = *Some b* ⟹ (*a*,*b*) ∈ *r*
shows ∃ *b*. *compute-NF f a* = *Some b* (**is** *?P a*)
⟨*proof*⟩

definition *compute-trancl A R* = *R*^+ `` *A*
lemma *compute-trancl-rtrancl*[*code-unfold*]: {*b*. (*a*,*b*) ∈ *R*^*} = *insert a* (*compute-trancl* {*a*} *R*)
⟨*proof*⟩

lemma *compute-trancl-code*[*code*]: *compute-trancl A R* = (*let B* = *R* `` *A in*

if $B \subseteq \{\}$ *then* $\{\}$ *else* $B \cup$ *compute-trancl* B $\{$ $ab \in R$. *fst* $ab \notin A \wedge$ *snd* $ab \notin B\}$)
⟨*proof*⟩

lemma *trancl-image-code*[*code-unfold*]: $R\hat{}+$ `` A = *compute-trancl* A R ⟨*proof*⟩
lemma *compute-rtrancl*[*code-unfold*]: $R\hat{}*$ `` A = $A \cup$ *compute-trancl* A R ⟨*proof*⟩
lemma *trancl-image-code'*[*code-unfold*]: $(a,b) \in R\hat{}+ \longleftrightarrow b \in$ *compute-trancl* $\{a\}$ R ⟨*proof*⟩
lemma *rtrancl-image-code*[*code-unfold*]: $(a,b) \in R\hat{}* \longleftrightarrow b = a \vee b \in$ *compute-trancl* $\{a\}$ R
⟨*proof*⟩

end

3.2.1 Closure-Operations on Relations

theory *Relation-Closure*
imports *Abstract−Rewriting.Relative-Rewriting*
begin

locale *rel-closure* =
fixes *cop* :: $'b \Rightarrow 'a \Rightarrow 'a$ — closure operator
and *nil* :: $'b$
and *add* :: $'b \Rightarrow 'b \Rightarrow 'b$
assumes *cop-nil*: *cop nil* $x = x$
assumes *cop-add*: *cop* (*add* a b) x = *cop* a (*cop* b x)
begin

inductive-set *closure* **for** r :: $'a$ *rel*
where
[*intro*]: $(x, y) \in r \Longrightarrow$ (*cop* a x, *cop* a y) $\in$ *closure* r

lemma *closureI2*: $(x, y) \in r \Longrightarrow u =$ *cop* a $x \Longrightarrow v =$ *cop* a $y \Longrightarrow (u, v) \in$ *closure* r ⟨*proof*⟩

lemma *closure-mono*: $r \subseteq s \Longrightarrow$ *closure* $r \subseteq$ *closure* s ⟨*proof*⟩

lemma *subset-closure*: $r \subseteq$ *closure* r
⟨*proof*⟩

definition *closed* $r \longleftrightarrow$ *closure* $r \subseteq r$

lemma *closure-subset*: *closed* $r \Longrightarrow$ *closure* $r \subseteq r$
⟨*proof*⟩

lemma *closedI* [*Pure.intro*, *intro*]: $(\bigwedge x\ y\ a.\ (x, y) \in r \Longrightarrow$ (*cop* a x, *cop* a y) $\in r)$ $\Longrightarrow$ *closed* r
⟨*proof*⟩

lemma *closedD* [*dest*]: *closed r* $\Longrightarrow$ $(x, y) \in r$ $\Longrightarrow$ $(cop\ a\ x, cop\ a\ y) \in r$
⟨*proof*⟩

lemma *closed-closure* [*intro*]: *closed* (*closure r*)
⟨*proof*⟩

lemma *subset-closure-Un*:
closure r $\subseteq$ *closure* $(r \cup s)$
closure s $\subseteq$ *closure* $(r \cup s)$
⟨*proof*⟩

lemma *closure-Un*: *closure* $(r \cup s)$ = *closure r* $\cup$ *closure s*
⟨*proof*⟩

lemma *closure-id* [*simp*]: *closed r* $\Longrightarrow$ *closure r* = *r*
⟨*proof*⟩

lemma *closed-Un* [*intro*]: *closed r* $\Longrightarrow$ *closed s* $\Longrightarrow$ *closed* $(r \cup s)$ ⟨*proof*⟩

lemma *closed-Inr* [*intro*]: *closed r* $\Longrightarrow$ *closed s* $\Longrightarrow$ *closed* $(r \cap s)$ ⟨*proof*⟩

lemma *closed-rtrancl* [*intro*]: *closed r* $\Longrightarrow$ *closed* (r^*)
⟨*proof*⟩

lemma *closed-trancl* [*intro*]: *closed r* $\Longrightarrow$ *closed* (r^+)
⟨*proof*⟩

lemma *closed-converse* [*intro*]: *closed r* $\Longrightarrow$ *closed* (r^{-1}) ⟨*proof*⟩

lemma *closed-comp* [*intro*]: *closed r* $\Longrightarrow$ *closed s* $\Longrightarrow$ *closed* $(r\ O\ s)$ ⟨*proof*⟩

lemma *closed-relpow* [*intro*]: *closed r* $\Longrightarrow$ *closed* $(r \hat{\ } n)$
⟨*proof*⟩

lemma *closed-conversion* [*intro*]: *closed r* $\Longrightarrow$ *closed* $(r^{\leftrightarrow *})$
⟨*proof*⟩

lemma *closed-relto* [*intro*]: *closed r* $\Longrightarrow$ *closed s* $\Longrightarrow$ *closed* $(relto\ r\ s)$ ⟨*proof*⟩

lemma *closure-diff-subset*: *closure r* − *closure s* $\subseteq$ *closure* $(r - s)$ ⟨*proof*⟩

end

end

4 Term Rewrite Systems

theory *Trs*

imports
Relation-Closure
First-Order-Terms.Term-More
Abstract−Rewriting.Relative-Rewriting
begin

A rewrite rule is a pair of terms. A term rewrite system (TRS) is a set of rewrite rules.

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

inductive-set *rstep* :: - $\Rightarrow ('f, 'v)$ *term rel* **for** $R :: ('f, 'v)$ *trs*
where
rstep: $\bigwedge C\ \sigma\ l\ r.\ (l, r) \in R \Longrightarrow s = C\langle l \cdot \sigma\rangle \Longrightarrow t = C\langle r \cdot \sigma\rangle \Longrightarrow (s, t) \in$ *rstep* R

lemma *rstep-induct-rule* [*case-names IH*, *induct set*: *rstep*]:
assumes $(s, t) \in$ *rstep* R
and $\bigwedge C\ \sigma\ l\ r.\ (l, r) \in R \Longrightarrow P\ (C\langle l \cdot \sigma\rangle)\ (C\langle r \cdot \sigma\rangle)$
shows $P\ s\ t$
⟨*proof*⟩

An alternative induction scheme that treats the rule-case, the substition-case, and the context-case separately.

lemma *rstep-induct* [*consumes 1*, *case-names rule subst ctxt*]:
assumes $(s, t) \in$ *rstep* R
and *rule*: $\bigwedge l\ r.\ (l, r) \in R \Longrightarrow P\ l\ r$
and *subst*: $\bigwedge s\ t\ \sigma.\ P\ s\ t \Longrightarrow P\ (s \cdot \sigma)\ (t \cdot \sigma)$
and *ctxt*: $\bigwedge s\ t\ C.\ P\ s\ t \Longrightarrow P\ (C\langle s\rangle)\ (C\langle t\rangle)$
shows $P\ s\ t$
⟨*proof*⟩

lemmas *rstepI* = *rstep.intros* [*intro*]

lemmas *rstepE* = *rstep.cases* [*elim*]

lemma *rstep-ctxt* [*intro*]: $(s, t) \in$ *rstep* $R \Longrightarrow (C\langle s\rangle, C\langle t\rangle) \in$ *rstep* R
⟨*proof*⟩

lemma *rstep-rule* [*intro*]: $(l, r) \in R \Longrightarrow (l, r) \in$ *rstep* R
⟨*proof*⟩

lemma *rstep-subst* [*intro*]: $(s, t) \in$ *rstep* $R \Longrightarrow (s \cdot \sigma, t \cdot \sigma) \in$ *rstep* R
⟨*proof*⟩

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

lemma *rstep-mono*: $R \subseteq S \Longrightarrow$ *rstep* $R \subseteq$ *rstep* S

⟨*proof*⟩

lemma *rstep-union*: *rstep* $(R \cup S)$ = *rstep* $R \cup$ *rstep* S
⟨*proof*⟩

lemma *rstep-converse* [*simp*]: *rstep* (R^{-1}) = $(rstep\ R)^{-1}$
⟨*proof*⟩

interpretation *subst*: *rel-closure* $\lambda\sigma\ t.\ t \cdot \sigma$ *Var* $\lambda x\ y.\ y \circ_s x$ ⟨*proof*⟩
declare *subst.closure.induct* [*consumes 1*, *case-names subst*, *induct pred*: *subst.closure*]
declare *subst.closure.cases* [*consumes 1*, *case-names subst*, *cases pred*: *subst.closure*]

interpretation *ctxt*: *rel-closure ctxt-apply-term* $\Box$ $(\circ_c)$ ⟨*proof*⟩
declare *ctxt.closure.induct* [*consumes 1*, *case-names ctxt*, *induct pred*: *ctxt.closure*]
declare *ctxt.closure.cases* [*consumes 1*, *case-names ctxt*, *cases pred*: *ctxt.closure*]

lemma *rstep-eq-closure*: *rstep* R = *ctxt.closure* (*subst.closure* R)
⟨*proof*⟩

lemma *ctxt-closed-rstep* [*intro*]: *ctxt.closed* (*rstep* R)
⟨*proof*⟩

lemma *ctxt-closed-one*:
ctxt.closed $r \Longrightarrow (s,\ t) \in r \Longrightarrow (Fun\ f\ (ss\ @\ s\ \#\ ts),\ Fun\ f\ (ss\ @\ t\ \#\ ts)) \in r$
⟨*proof*⟩

4.1 Well-formed TRSs

definition
wf-trs :: $('f,\ 'v)$ *trs* ⇒ *bool*
where
wf-trs $R = (\forall\ l\ r.\ (l,r) \in R \longrightarrow (\exists f\ ts.\ l = Fun\ f\ ts) \wedge vars\text{-}term\ r \subseteq vars\text{-}term\ l)$

lemma *wf-trs-imp-lhs-Fun*:
wf-trs $R \Longrightarrow (l,r) \in R \Longrightarrow \exists f\ ts.\ l = Fun\ f\ ts$
⟨*proof*⟩

lemma *rstep-imp-Fun*:
assumes *wf-trs* R
shows $(s,\ t) \in rstep\ R \Longrightarrow \exists f\ ss.\ s = Fun\ f\ ss$
⟨*proof*⟩

lemma *SN-Var*:
assumes *wf-trs* R **shows** *SN-on* (*rstep* R) $\{Var\ x\}$
⟨*proof*⟩

4.2 Function Symbols and Variables of Rules and TRSs

definition

vars-rule :: ('*f*, '*v*) *rule* ⇒ '*v set*
where
vars-rule r = *vars-term* (*fst r*) ∪ *vars-term* (*snd r*)

lemma *finite-vars-rule*:
finite (*vars-rule r*)
⟨*proof*⟩

definition *vars-trs* :: ('*f*, '*v*) *trs* ⇒ '*v set* **where**
vars-trs R = (⋃ *r*∈*R*. *vars-rule r*)

lemma *vars-trs-union*: *vars-trs* (*R* ∪ *S*) = *vars-trs R* ∪ *vars-trs S*
⟨*proof*⟩

lemma *finite-trs-has-finite-vars*:
assumes *finite R* **shows** *finite* (*vars-trs R*)
⟨*proof*⟩

lemmas *vars-defs* = *vars-trs-def vars-rule-def*

definition *funs-rule* :: ('*f*, '*v*) *rule* ⇒ '*f set* **where**
funs-rule r = *funs-term* (*fst r*) ∪ *funs-term* (*snd r*)

The same including arities.

definition *funas-rule* :: ('*f*, '*v*) *rule* ⇒ '*f sig* **where**
funas-rule r = *funas-term* (*fst r*) ∪ *funas-term* (*snd r*)

definition *funs-trs* :: ('*f*, '*v*) *trs* ⇒ '*f set* **where**
funs-trs R = (⋃ *r*∈*R*. *funs-rule r*)

definition *funas-trs* :: ('*f*, '*v*) *trs* ⇒ '*f sig* **where**
funas-trs R = (⋃ *r*∈*R*. *funas-rule r*)

lemma *funs-rule-funas-rule*: *funs-rule rl* = *fst* ‘ *funas-rule rl*
⟨*proof*⟩

lemma *funs-trs-funas-trs*:*funs-trs R* = *fst* ‘ *funas-trs R*
⟨*proof*⟩

lemma *finite-funas-rule*: *finite* (*funas-rule lr*)
⟨*proof*⟩

lemma *finite-funas-trs*:
assumes *finite R*
shows *finite* (*funas-trs R*)
⟨*proof*⟩

lemma *funas-empty*[*simp*]: *funas-trs* {} = {} ⟨*proof*⟩

lemma *funas-trs-union*[*simp*]: *funas-trs* $(R \cup S)$ = *funas-trs* $R \cup$ *funas-trs* S
⟨*proof*⟩

definition *funas-args-rule* :: $('f, 'v)$ *rule* ⇒ $'f$ *sig* **where**
funas-args-rule r = *funas-args-term* (*fst* r) ∪ *funas-args-term* (*snd* r)

definition *funas-args-trs* :: $('f, 'v)$ *trs* ⇒ $'f$ *sig* **where**
funas-args-trs $R = (\bigcup r{\in}R.$ *funas-args-rule* $r)$

lemmas *funas-args-defs* =
funas-args-trs-def funas-args-rule-def funas-args-term-def

definition *roots-rule* :: $('f, 'v)$ *rule* ⇒ $'f$ *sig*
where
roots-rule r = *set-option* (*root* (*fst* r)) ∪ *set-option* (*root* (*snd* r))

definition *roots-trs* :: $('f, 'v)$ *trs* ⇒ $'f$ *sig* **where**
roots-trs $R = (\bigcup r{\in}R.$ *roots-rule* $r)$

lemmas *roots-defs* =
roots-trs-def roots-rule-def

definition *funas-head* :: $('f, 'v)$ *trs* ⇒ $('f, 'v)$ *trs* ⇒ $'f$ *sig* **where**
funas-head P R = *funas-trs* P − (*funas-trs* R ∪ *funas-args-trs* P)

lemmas *funs-defs* = *funs-trs-def funs-rule-def*
lemmas *funas-defs* =
funas-trs-def funas-rule-def
funas-args-defs
funas-head-def
roots-defs

A function symbol is said to be *defined* (w.r.t. to a given TRS) if it occurs as root of some left-hand side.

definition
defined :: $('f, 'v)$ *trs* ⇒ $('f \times nat)$ ⇒ *bool*
where
defined R *fn* ⟷ $(\exists l\ r.\ (l, r) \in R \wedge$ *root* l = *Some fn*$)$

lemma *defined-funas-trs*: **assumes** *d*: *defined* R *fn* **shows** *fn* ∈ *funas-trs* R
⟨*proof*⟩

fun *root-list* :: $('f, 'v)$ *term* ⇒ $('f \times nat)$ *list*
where
root-list (*Var* x) = [] |
root-list (*Fun* f *ts*) = [(f, *length ts*)]

definition *vars-rule-list* :: $('f, 'v)$ *rule* ⇒ $'v$ *list*
where

vars-rule-list r = vars-term-list (fst r) @ vars-term-list (snd r)

definition *funs-rule-list* :: *('f, 'v) rule ⇒ 'f list*
where
funs-rule-list r = funs-term-list (fst r) @ funs-term-list (snd r)

definition *funas-rule-list* :: *('f, 'v) rule ⇒ ('f × nat) list*
where
funas-rule-list r = funas-term-list (fst r) @ funas-term-list (snd r)

definition *roots-rule-list* :: *('f, 'v) rule ⇒ ('f × nat) list*
where
roots-rule-list r = root-list (fst r) @ root-list (snd r)

definition *funas-args-rule-list* :: *('f, 'v) rule ⇒ ('f × nat) list*
where
funas-args-rule-list r = funas-args-term-list (fst r) @ funas-args-term-list (snd r)

lemma *set-vars-rule-list* [*simp*]:
set (vars-rule-list r) = vars-rule r
⟨*proof*⟩

lemma *set-funs-rule-list* [*simp*]:
set (funs-rule-list r) = funs-rule r
⟨*proof*⟩

lemma *set-funas-rule-list* [*simp*]:
set (funas-rule-list r) = funas-rule r
⟨*proof*⟩

lemma *set-roots-rule-list* [*simp*]:
set (roots-rule-list r) = roots-rule r
⟨*proof*⟩

lemma *set-funas-args-rule-list* [*simp*]:
set (funas-args-rule-list r) = funas-args-rule r
⟨*proof*⟩

definition *vars-trs-list* :: *('f, 'v) rule list ⇒ 'v list*
where
vars-trs-list trs = concat (map vars-rule-list trs)

definition *funs-trs-list* :: *('f, 'v) rule list ⇒ 'f list*
where
funs-trs-list trs = concat (map funs-rule-list trs)

definition *funas-trs-list* :: *('f, 'v) rule list ⇒ ('f × nat) list*
where

funas-trs-list trs = *concat* (*map funas-rule-list trs*)

definition *roots-trs-list* :: ('*f*, '*v*) *rule list* ⇒ ('*f* × *nat*) *list*
where
roots-trs-list trs = *remdups* (*concat* (*map roots-rule-list trs*))

definition *funas-args-trs-list* :: ('*f*, '*v*) *rule list* ⇒ ('*f* × *nat*) *list*
where
funas-args-trs-list trs = *concat* (*map funas-args-rule-list trs*)

lemma *set-vars-trs-list* [*simp*]:
set (*vars-trs-list trs*) = *vars-trs* (*set trs*)
⟨*proof*⟩

lemma *set-funs-trs-list* [*simp*]:
set (*funs-trs-list R*) = *funs-trs* (*set R*)
⟨*proof*⟩

lemma *set-funas-trs-list* [*simp*]:
set (*funas-trs-list R*) = *funas-trs* (*set R*)
⟨*proof*⟩

lemma *set-roots-trs-list* [*simp*]:
set (*roots-trs-list R*) = *roots-trs* (*set R*)
⟨*proof*⟩

lemma *set-funas-args-trs-list* [*simp*]:
set (*funas-args-trs-list R*) = *funas-args-trs* (*set R*)
⟨*proof*⟩

lemmas *vars-list-defs* = *vars-trs-list-def vars-rule-list-def*
lemmas *funs-list-defs* = *funs-trs-list-def funs-rule-list-def*
lemmas *funas-list-defs* = *funas-trs-list-def funas-rule-list-def*
lemmas *roots-list-defs* = *roots-trs-list-def roots-rule-list-def*
lemmas *funas-args-list-defs* = *funas-args-trs-list-def funas-args-rule-list-def*

lemma *vars-trs-list-Nil* [*simp*]:
vars-trs-list [] = [] ⟨*proof*⟩

context
fixes *R* :: ('*f*, '*v*) *trs*
assumes *wf-trs R*
begin

lemma *funas-term-subst-rhs*:
assumes *funas-trs R* ⊆ *F* **and** (*l*, *r*) ∈ *R* **and** *funas-term* (*l* · *σ*) ⊆ *F*
shows *funas-term* (*r* · *σ*) ⊆ *F*
⟨*proof*⟩

lemma *vars-rule-lhs*:
$r \in R \Longrightarrow$ *vars-rule* $r =$ *vars-term* (*fst* r)
⟨*proof*⟩

end

4.3 Closure Properties

lemma *ctxt-closed-R-imp-supt-R-distr*:
assumes *ctxt.closed* R **and** $s \rhd t$ **and** $(t, u) \in R$ **shows** $\exists t.\ (s, t) \in R \wedge t \rhd u$
⟨*proof*⟩

lemma *ctxt-closed-imp-qc-supt*: *ctxt.closed* $R \Longrightarrow \{\rhd\}\ O\ R \subseteq R\ O\ (R \cup \{\rhd\})^*$
⟨*proof*⟩

Let R be a relation on terms that is closed under contexts. If R is well-founded then $R \cup \rhd$ is well-founed.

lemma *SN-imp-SN-union-supt*:
assumes *SN* R **and** *ctxt.closed* R
shows *SN* $(R \cup \{\rhd\})$
⟨*proof*⟩

lemma *stable-loop-imp-not-SN*:
assumes *stable*: *subst.closed* r **and** *steps*: $(s, s \cdot \sigma) \in r^+$
shows $\neg$ *SN-on* r $\{s\}$
⟨*proof*⟩

lemma *subst-closed-supteq*: *subst.closed* $\{\trianglerighteq\}$ ⟨*proof*⟩

lemma *subst-closed-supt*: *subst.closed* $\{\rhd\}$ ⟨*proof*⟩

lemma *ctxt-closed-supt-subset*: *ctxt.closed* $R \Longrightarrow \{\rhd\}\ O\ R \subseteq R\ O\ \{\rhd\}$ ⟨*proof*⟩

4.4 Properties of Rewrite Steps

lemma *rstep-relcomp-idemp1* [*simp*]:
rstep (*rstep* R O *rstep* S) = *rstep* R O *rstep* S
⟨*proof*⟩

lemma *rstep-relcomp-idemp2* [*simp*]:
rstep (*rstep* R O *rstep* S O *rstep* T) = *rstep* R O *rstep* S O *rstep* T
⟨*proof*⟩

lemma *ctxt-closed-rsteps* [*intro*]: *ctxt.closed* ((*rstep* R)*) ⟨*proof*⟩

lemma *subset-rstep*: $R \subseteq$ *rstep* R ⟨*proof*⟩

lemma *subst-closure-rstep-subset*: *subst.closure* (*rstep* R) $\subseteq$ *rstep* R
⟨*proof*⟩

lemma *subst-closed-rstep* [*intro*]: *subst.closed* (*rstep R*) ⟨*proof*⟩

lemma *subst-closed-rsteps*: *subst.closed* ((*rstep R*)*) ⟨*proof*⟩

lemmas *supt-rsteps-subset* = *ctxt-closed-supt-subset* [*OF ctxt-closed-rsteps*]

lemma *supteq-rsteps-subset*:
  {⊵} *O* (*rstep R*)* ⊆ (*rstep R*)* *O* {⊵} (**is** *?S* ⊆ *?T*)
  ⟨*proof*⟩

lemma *quasi-commute-rsteps-supt*:
  quasi-commute ((*rstep R*)*) {▷}
  ⟨*proof*⟩

lemma *rstep-UN*:
  rstep (⋃ *i*∈*A*. *R i*) = (⋃ *i*∈*A*. *rstep* (*R i*))
  ⟨*proof*⟩

definition
  rstep-r-p-s :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *rule* ⇒ *pos* ⇒ ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *trs*
  where
    rstep-r-p-s R r p σ = {(*s*, *t*).
     let C = *ctxt-of-pos-term p s in p* ∈ *poss s* ∧ *r* ∈ *R* ∧ (*C*⟨*fst r* · *σ*⟩ = *s*) ∧ (*C*⟨*snd r* · *σ*⟩ = *t*)}

lemma *rstep-r-p-s-def'*:
  rstep-r-p-s R r p σ = {(*s*, *t*).
    p ∈ *poss s* ∧ *r* ∈ *R* ∧ *s* |- *p* = *fst r* · *σ* ∧ *t* = *replace-at s p* (*snd r* · *σ*)} (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *parallel-steps*:
  fixes p_1 :: *pos*
  assumes (*s*, *t*) ∈ *rstep-r-p-s* R_1 (l_1, r_1) p_1 σ_1
    and (*s*, *u*) ∈ *rstep-r-p-s* R_2 (l_2, r_2) p_2 σ_2
    and *par*: $p_1 \perp p_2$
  shows (*t*, (*ctxt-of-pos-term* p_1 *u*)⟨r_1 · σ_1⟩) ∈ *rstep-r-p-s* R_2 (l_2, r_2) p_2 σ_2 ∧
        (*u*, (*ctxt-of-pos-term* p_1 *u*)⟨r_1 · σ_1⟩) ∈ *rstep-r-p-s* R_1 (l_1, r_1) p_1 σ_1
⟨*proof*⟩

lemma *rstep-iff-rstep-r-p-s*:
  (*s*, *t*) ∈ *rstep R* ⟷ (∃ *l r p σ*. (*s*, *t*) ∈ *rstep-r-p-s R* (*l*, *r*) *p σ*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *rstep-r-p-s-imp-rstep*:
  assumes (*s*, *t*) ∈ *rstep-r-p-s R r p σ*
  shows (*s*, *t*) ∈ *rstep R*
  ⟨*proof*⟩

Rewriting steps below the root position.

definition
nrrstep :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *trs*
where
nrrstep R = {(*s*,*t*). ∃ *r i ps σ*. (*s*,*t*) ∈ *rstep-r-p-s R r* (*i*#*ps*) *σ*}

An alternative characterisation of non-root rewrite steps.

lemma *nrrstep-def'*:
nrrstep R = {(*s*, *t*). ∃ *l r C σ*. (*l*, *r*) ∈ *R* ∧ *C* ≠ □ ∧ *s* = *C*⟨*l*·*σ*⟩ ∧ *t* = *C*⟨*r*·*σ*⟩}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *nrrstepI*: (*l*,*r*) ∈ *R* ⟹ *s* = *C*⟨*l*·*σ*⟩ ⟹ *t* = *C*⟨*r* · *σ*⟩ ⟹ *C* ≠ □ ⟹ (*s*,*t*) ∈ *nrrstep R* ⟨*proof*⟩

lemma *nrrstep-union*: *nrrstep* (*R* ∪ *S*) = *nrrstep R* ∪ *nrrstep S*
⟨*proof*⟩

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

Rewriting step at the root position.

definition
rrstep :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *trs*
where
rrstep R = {(*s*,*t*). ∃ *r σ*. (*s*,*t*) ∈ *rstep-r-p-s R r* [] *σ*}

An alternative characterisation of root rewrite steps.

lemma *rrstep-def'*: *rrstep R* = {(*s*, *t*). ∃ *l r σ*. (*l*, *r*) ∈ *R* ∧ *s* = *l*·*σ* ∧ *t* = *r*·*σ*}
(**is** - = *?rhs*)
⟨*proof*⟩

lemma *rules-subset-rrstep* [*simp*]: *R* ⊆ *rrstep R*
⟨*proof*⟩

lemma *rrstep-union*: *rrstep* (*R* ∪ *S*) = *rrstep R* ∪ *rrstep S* ⟨*proof*⟩

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

lemma *subst-closed-rrstep*: *subst.closed* (*rrstep R*)
⟨*proof*⟩

lemma *rstep-iff-rrstep-or-nrrstep*: *rstep R* = (*rrstep R* ∪ *nrrstep R*)
⟨*proof*⟩

lemma *rstep-i-pos-imp-rstep-arg-i-pos*:
assumes *nrrstep*: (*Fun f ss*,*t*) ∈ *rstep-r-p-s R* (*l*,*r*) (*i*#*ps*) *σ*
shows (*ss*!*i*,*t*|-[*i*]) ∈ *rstep-r-p-s R* (*l*,*r*) *ps σ*

⟨*proof*⟩

lemma *ctxt-closure-rstep-eq* [*simp*]: *ctxt.closure* (*rstep R*) = *rstep R*
⟨*proof*⟩

lemma *subst-closure-rstep-eq* [*simp*]: *subst.closure* (*rstep R*) = *rstep R*
⟨*proof*⟩

lemma *supt-rstep-subset*:
{▷} *O rstep R* ⊆ *rstep R O* {▷}
⟨*proof*⟩

lemma *ne-rstep-seq-imp-list-of-terms*:
assumes (*s*,*t*) ∈ (*rstep R*)$^+$
shows ∃ *ts*. *length ts* > *1* ∧ *ts*!*0* = *s* ∧ *ts*![*length ts* − *1*) = *t* ∧
(∀ *i*<*length ts* − *1*. (*ts*!*i*,*ts*!(*Suc i*)) ∈ (*rstep R*)) (**is** ∃ *ts*. - ∧ - ∧ - ∧ *?P ts*)
⟨*proof*⟩

locale *E-compatible* =
fixes *R* :: (′*f*,′*v*)*trs* **and** *E* :: (′*f*,′*v*)*trs*
assumes *E*: *E O R* = *R Id* ⊆ *E*
begin

definition *restrict-SN-supt-E* :: (′*f*, ′*v*) *trs* **where**
restrict-SN-supt-E = *restrict-SN R R* ∪ *restrict-SN* (*E O* {▷} *O E*) *R*

lemma *ctxt-closed-R-imp-supt-restrict-SN-E-distr*:
assumes *ctxt.closed R*
and (*s*,*t*) ∈ (*restrict-SN* (*E O* {▷}) *R*)
and (*t*,*u*) ∈ *restrict-SN R R*
shows (∃ *t*. (*s*,*t*) ∈ *restrict-SN R R* ∧ (*t*,*u*) ∈ *restrict-SN* (*E O* {▷}) *R*) (**is** ∃ *t*.
- ∧ (*t*,*u*) ∈ *?snSub*)
⟨*proof*⟩

lemma *ctxt-closed-R-imp-restrict-SN-qc-E-supt*:
assumes *ctxt*: *ctxt.closed R*
shows *quasi-commute* (*restrict-SN R R*) (*restrict-SN* (*E O* {▷} *O E*) *R*) (**is**
quasi-commute ?r ?s)
⟨*proof*⟩

lemma *ctxt-closed-imp-SN-restrict-SN-E-supt*:
assumes *ctxt.closed R*
and *SN*: *SN* (*E O* {▷} *O E*)
shows *SN restrict-SN-supt-E*
⟨*proof*⟩
end

lemma *E-compatible-Id*: *E-compatible R Id*

⟨*proof*⟩

definition *restrict-SN-supt* :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *trs* **where**
restrict-SN-supt R = *restrict-SN R R* ∪ *restrict-SN* {▷} *R*

lemma *ctxt-closed-SN-on-subt*:
assumes *ctxt.closed R* **and** *SN-on R* {*s*} **and** *s* ⊵ *t*
shows *SN-on R* {*t*}
⟨*proof*⟩

lemma *ctxt-closed-R-imp-supt-restrict-SN-distr*:
assumes *R*: *ctxt.closed R*
and *st*: (*s*,*t*) ∈ (*restrict-SN* {▷} *R*)
and *tu*: (*t*,*u*) ∈ *restrict-SN R R*
shows (∃ *t*. (*s*,*t*) ∈ *restrict-SN R R* ∧ (*t*,*u*) ∈ *restrict-SN* {▷} *R*) (**is** ∃ *t*. - ∧ (*t*,*u*) ∈ *?snSub*)
⟨*proof*⟩

lemma *ctxt-closed-R-imp-restrict-SN-qc-supt*:
assumes *ctxt.closed R*
shows *quasi-commute* (*restrict-SN R R*) (*restrict-SN supt R*) (**is** *quasi-commute ?r ?s*)
⟨*proof*⟩

lemma *ctxt-closed-imp-SN-restrict-SN-supt*:
assumes *ctxt.closed R*
shows *SN* (*restrict-SN-supt R*)
⟨*proof*⟩

lemma *SN-restrict-SN-supt-rstep*:
shows *SN* (*restrict-SN-supt* (*rstep R*))
⟨*proof*⟩

lemma *nrrstep-imp-pos-term*:
(*Fun f ss*,*t*) ∈ *nrrstep R* ⟹
∃ *i s*. *t* = *Fun f* (*ss*[*i*:=*s*]) ∧ (*ss*!*i*,*s*) ∈ *rstep R* ∧ *i* < *length ss*
⟨*proof*⟩

lemma *rstep-cases*[*consumes 1*, *case-names root nonroot*]:
⟦(*s*,*t*) ∈ *rstep R*; (*s*,*t*) ∈ *rrstep R* ⟹ *P*; (*s*,*t*) ∈ *nrrstep R* ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

lemma *nrrstep-imp-rstep*: (*s*,*t*) ∈ *nrrstep R* ⟹ (*s*,*t*) ∈ *rstep R*
⟨*proof*⟩

lemma *nrrstep-imp-Fun*: (*s*,*t*) ∈ *nrrstep R* ⟹ ∃ *f ss*. *s* = *Fun f ss*
⟨*proof*⟩

lemma *nrrstep-imp-subt-rstep*:
assumes $(s,t) \in nrrstep\ R$
shows $\exists\, i.\ i < num\text{-}args\ s \wedge num\text{-}args\ s = num\text{-}args\ t \wedge (s|\text{-}[i],t|\text{-}[i]) \in rstep\ R \wedge (\forall\, j.\ i \neq j \longrightarrow s|\text{-}[j] = t|\text{-}[j])$
⟨*proof*⟩

lemma *nrrstep-subt*: **assumes** $(s,\ t) \in nrrstep\ R$ **shows** $\exists\, u \lhd s.\ \exists\, v \lhd t.\ (u,\ v) \in rstep\ R$
⟨*proof*⟩

lemma *nrrstep-args*:
assumes $(s,\ t) \in nrrstep\ R$
shows $\exists f\ ss\ ts.\ s = Fun\ f\ ss \wedge t = Fun\ f\ ts \wedge length\ ss = length\ ts$
$\wedge\ (\exists j{<}length\ ss.\ (ss!j,\ ts!j) \in rstep\ R \wedge (\forall\, i{<}length\ ss.\ i \neq j \longrightarrow ss!i = ts!i))$
⟨*proof*⟩

lemma *nrrstep-iff-arg-rstep*:
$(s,t) \in nrrstep\ R \longleftrightarrow$
$(\exists f\ ss\ i\ t'.\ s = Fun\ f\ ss \wedge i < length\ ss \wedge t = Fun\ f\ (ss[i{:=}t']) \wedge (ss!i,t') \in rstep\ R)$
(**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *subterms-NF-imp-SN-on-nrrstep*:
assumes $\forall\, s \lhd t.\ s \in NF\ (rstep\ R)$ **shows** $SN\text{-}on\ (nrrstep\ R)\ \{t\}$
⟨*proof*⟩

lemma *args-NF-imp-SN-on-nrrstep*:
assumes $\forall\, t{\in}set\ ts.\ t \in NF\ (rstep\ R)$ **shows** $SN\text{-}on\ (nrrstep\ R)\ \{Fun\ f\ ts\}$
⟨*proof*⟩

lemma *rrstep-imp-rule-subst*:
assumes $(s,t) \in rrstep\ R$
shows $\exists\, l\ r\ \sigma.\ (l,r) \in R \wedge (l{\cdot}\sigma) = s \wedge (r{\cdot}\sigma) = t$
⟨*proof*⟩

lemma *nrrstep-preserves-root*:
assumes $(Fun\ f\ ss,t) \in nrrstep\ R$ (**is** $(?s,t) \in nrrstep\ R$) **shows** $\exists\, ts.\ t = (Fun\ f\ ts)$
⟨*proof*⟩

lemma *nrrstep-equiv-root*: **assumes** $(s,t) \in nrrstep\ R$ **shows** $\exists f\ ss\ ts.\ s = Fun\ f\ ss \wedge t = Fun\ f\ ts$
⟨*proof*⟩

lemma *nrrstep-reflects-root*:
assumes $(s,Fun\ g\ ts) \in nrrstep\ R$ (**is** $(s,?t) \in nrrstep\ R$)
shows $\exists\, ss.\ s = (Fun\ g\ ss)$

⟨*proof*⟩

lemma *nrrsteps-preserve-root*:
assumes (*Fun f ss*,*t*) ∈ (*nrrstep R*)*
shows ∃ *ts*. *t* = (*Fun f ts*)
⟨*proof*⟩

lemma *nrrstep-Fun-imp-arg-rsteps*:
assumes (*Fun f ss*,*Fun f ts*) ∈ *nrrstep R* (**is** (*?s*,*?t*) ∈ *nrrstep R*) **and** *i* < *length ss*
shows (*ss*!*i*,*ts*!*i*) ∈ (*rstep R*)*
⟨*proof*⟩

lemma *nrrstep-imp-arg-rsteps*:
assumes (*s*,*t*) ∈ *nrrstep R* **and** *i* < *num-args s* **shows** (*args s*!*i*,*args t*!*i*) ∈ (*rstep R*)*
⟨*proof*⟩

lemma *nrrsteps-imp-rsteps*: (*s*,*t*) ∈ (*nrrstep R*)* ⟹ (*s*,*t*) ∈ (*rstep R*)*
⟨*proof*⟩

lemma *nrrstep-Fun-preserves-num-args*:
assumes (*Fun f ss*,*Fun f ts*) ∈ *nrrstep R* (**is** (*?s*,*?t*) ∈ *nrrstep R*)
shows *length ss* = *length ts*
⟨*proof*⟩

lemma *nrrstep-equiv-num-args*:
assumes (*s*,*t*) ∈ *nrrstep R* **shows** *num-args s* = *num-args t*
⟨*proof*⟩

lemma *nrrsteps-equiv-num-args*:
assumes (*s*,*t*) ∈ (*nrrstep R*)* **shows** *num-args s* = *num-args t*
⟨*proof*⟩

lemma *nrrstep-preserves-num-args*:
assumes (*s*,*t*) ∈ *nrrstep R* **and** *i* < *num-args s* **shows** *i* < *num-args t*
⟨*proof*⟩

lemma *nrrstep-reflects-num-args*:
assumes (*s*,*t*) ∈ *nrrstep R* **and** *i* < *num-args t* **shows** *i* < *num-args s*
⟨*proof*⟩

lemma *nrrsteps-imp-arg-rsteps*:
assumes (*s*,*t*) ∈ (*nrrstep R*)* **and** *i* < *num-args s*
shows (*args s*!*i*,*args t*!*i*) ∈ (*rstep R*)*
⟨*proof*⟩

lemma *nrrsteps-imp-eq-root-arg-rsteps*:
assumes *steps*: (*s*,*t*) ∈ (*nrrstep R*)*

shows *root* s = *root* $t \wedge (\forall\, i{<}$*num-args* $s.\ (s$ |- $[i],\ t$ |- $[i]\ ) \in ($*rstep* $R)^*)$
⟨*proof*⟩

lemma *SN-on-imp-SN-on-subt*:
assumes *SN-on* (*rstep* R) $\{t\}$ **shows** $\forall\, s{\trianglelefteq}t.$ *SN-on* (*rstep* R) $\{s\}$
⟨*proof*⟩

lemma *not-SN-on-subt-imp-not-SN-on*:
assumes $\neg$ *SN-on* (*rstep* R) $\{t\}$ **and** $s \trianglerighteq t$
shows $\neg$ *SN-on* (*rstep* R) $\{s\}$
⟨*proof*⟩

lemma *SN-on-instance-imp-SN-on-var*:
assumes *SN-on* (*rstep* R) $\{t \cdot \sigma\}$ **and** $x \in$ *vars-term* t
shows *SN-on* (*rstep* R) $\{$*Var* $x \cdot \sigma\}$
⟨*proof*⟩

lemma *var-imp-var-of-arg*:
assumes $x \in$ *vars-term* (*Fun* f ss) (**is** $x \in$ *vars-term* $?s$)
shows $\exists\, i <$ *num-args* (*Fun* f ss). $x \in$ *vars-term* ($ss!i$)
⟨*proof*⟩

lemma *subt-instance-and-not-subst-imp-subt*:
$s{\cdot}\sigma \trianglerighteq t \Longrightarrow \forall\, x \in$ *vars-term* $s.\ \neg((\,$*Var* $x){\cdot}\sigma \trianglerighteq t) \Longrightarrow \exists\, u.\ s \trianglerighteq u \wedge t = u{\cdot}\sigma$
⟨*proof*⟩

lemma *SN-imp-SN-subt*:
SN-on (*rstep* R) $\{s\} \Longrightarrow s \trianglerighteq t \Longrightarrow$ *SN-on* (*rstep* R) $\{t\}$
⟨*proof*⟩

lemma *subterm-preserves-SN-gen*:
assumes *ctxt*: *ctxt.closed* R
and *SN*: *SN-on* R $\{t\}$ **and** *supt*: $t \rhd s$
shows *SN-on* R $\{s\}$
⟨*proof*⟩

context *E-compatible*
begin

lemma *SN-on-step-E-imp-SN-on*: **assumes** *SN-on* R $\{s\}$
and $(s,t) \in E$
shows *SN-on* R $\{t\}$
⟨*proof*⟩

lemma *SN-on-step-REs-imp-SN-on*:
assumes R: *ctxt.closed* R
and *st*: $(s,t) \in (R \cup E\ O\ \{\rhd\}\ O\ E)$
and *SN*: *SN-on* R $\{s\}$
shows *SN-on* R $\{t\}$

⟨*proof*⟩

lemma *restrict-SN-supt-E-I*:
ctxt.closed R ⟹ *SN-on R* {*s*} ⟹ (*s*,*t*) ∈ *R* ∪ *E O* {▷} *O E* ⟹ (*s*,*t*) ∈ *restrict-SN-supt-E*
⟨*proof*⟩

lemma *ctxt-closed-imp-SN-on-E-supt*:
assumes *R*: *ctxt.closed R*
and *SN*: *SN* (*E O* {▷} *O E*)
shows *SN-on* (*R* ∪ *E O* {▷} *O E*) {*t*. *SN-on R* {*t*}}
⟨*proof*⟩
end

lemma *subterm-preserves-SN*:
SN-on (*rstep R*) {*t*} ⟹ *t* ▷ *s* ⟹ *SN-on* (*rstep R*) {*s*}
⟨*proof*⟩

lemma *SN-on-r-imp-SN-on-supt-union-r*:
assumes *ctxt*: *ctxt.closed R*
and *SN-on R T*
shows *SN-on* (*supt* ∪ *R*) *T* (**is** *SN-on ?S T*)
⟨*proof*⟩

lemma *SN-on-rstep-imp-SN-on-supt-union-rstep*:
SN-on (*rstep R*) *T* ⟹ *SN-on* (*supt* ∪ *rstep R*) *T*
⟨*proof*⟩

lemma *SN-supt-r-trancl*:
assumes *ctxt*: *ctxt.closed R*
and *a*: *SN R*
shows *SN* ((*supt* ∪ *R*)$^+$)
⟨*proof*⟩

lemma *SN-supt-rstep-trancl*:
SN (*rstep R*) ⟹ *SN* ((*supt* ∪ *rstep R*)$^+$)
⟨*proof*⟩

lemma *SN-imp-SN-arg-gen*:
assumes *ctxt*: *ctxt.closed R*
and *SN*: *SN-on R* {*Fun f ts*} **and** *arg*: *t* ∈ *set ts* **shows** *SN-on R* {*t*}
⟨*proof*⟩

lemma *SN-imp-SN-arg*:
SN-on (*rstep R*) {*Fun f ts*} ⟹ *t* ∈ *set ts* ⟹ *SN-on* (*rstep R*) {*t*}
⟨*proof*⟩

lemma *SNinstance-imp-SN*:
assumes *SN-on* (*rstep R*) {*t* · *σ*}

shows *SN-on* (*rstep R*) {*t*}
⟨*proof*⟩

lemma *rstep-imp-C-s-r*:
assumes (*s*,*t*) ∈ *rstep R*
shows ∃ *C σ l r*. (*l*,*r*) ∈ *R* ∧ *s* = *C*⟨*l*·*σ*⟩ ∧ *t* = *C*⟨*r*·*σ*⟩
⟨*proof*⟩

fun *map-funs-rule* :: ('*f* ⇒ '*g*) ⇒ ('*f*, '*v*) *rule* ⇒ ('*g*, '*v*) *rule*
where
map-funs-rule fg lr = (*map-funs-term fg* (*fst lr*), *map-funs-term fg* (*snd lr*))

fun *map-funs-trs* :: ('*f* ⇒ '*g*) ⇒ ('*f*, '*v*) *trs* ⇒ ('*g*, '*v*) *trs*
where
map-funs-trs fg R = *map-funs-rule fg* ‘ *R*

lemma *map-funs-trs-union*: *map-funs-trs fg* (*R* ∪ *S*) = *map-funs-trs fg R* ∪ *map-funs-trs fg S*
⟨*proof*⟩

lemma *rstep-map-funs-term*: **assumes** *R*: ⋀ *f*. *f* ∈ *funs-trs R* ⟹ *h f* = *f* **and** *step*: (*s*,*t*) ∈ *rstep R*
shows (*map-funs-term h s*, *map-funs-term h t*) ∈ *rstep R*
⟨*proof*⟩

lemma *wf-trs-map-funs-trs*[*simp*]: *wf-trs* (*map-funs-trs f R*) = *wf-trs R*
⟨*proof*⟩

lemma *map-funs-trs-comp*: *map-funs-trs fg* (*map-funs-trs gh R*) = *map-funs-trs* (*fg o gh*) *R*
⟨*proof*⟩

lemma *map-funs-trs-mono*: **assumes** *R* ⊆ *R*′ **shows** *map-funs-trs fg R* ⊆ *map-funs-trs fg R*′
⟨*proof*⟩

lemma *map-funs-trs-power-mono*:
fixes *R R*′ :: ('*f*,'*v*)*trs* **and** *fg* :: '*f* ⇒ '*f*
assumes *R* ⊆ *R*′ **shows** ((*map-funs-trs fg*)^^*n*) *R* ⊆ ((*map-funs-trs fg*)^^*n*) *R*′
⟨*proof*⟩

declare *map-funs-trs.simps*[*simp del*]

lemma *rstep-imp-map-rstep*:
assumes (*s*, *t*) ∈ *rstep R*
shows (*map-funs-term fg s*, *map-funs-term fg t*) ∈ *rstep* (*map-funs-trs fg R*)
⟨*proof*⟩

lemma *rsteps-imp-map-rsteps*: **assumes** (*s*,*t*) ∈ (*rstep R*)*

shows (*map-funs-term fg s*, *map-funs-term fg t*) ∈ (*rstep* (*map-funs-trs fg R*))*
⟨*proof*⟩

lemma *SN-map-imp-SN*:
assumes *SN*: *SN-on* (*rstep* (*map-funs-trs fg R*)) {*map-funs-term fg t*}
shows *SN-on* (*rstep R*) {*t*}
⟨*proof*⟩

lemma *rstep-iff-map-rstep*:
assumes *inj fg*
shows (*s*, *t*) ∈ *rstep R* ⟷ (*map-funs-term fg s*, *map-funs-term fg t*) ∈ *rstep* (*map-funs-trs fg R*)
⟨*proof*⟩

lemma *rstep-map-funs-trs-power-mono*:
fixes *R R'* :: ('*f*,'*v*)*trs* **and** *fg* :: '*f* ⇒ '*f*
assumes *subset*: *R* ⊆ *R'* **shows** *rstep* (((*map-funs-trs fg*)^^*n*) *R*) ⊆ *rstep* (((*map-funs-trs fg*)^^*n*) *R'*)
⟨*proof*⟩

lemma *subsetI3*: (⋀*x y z*. (*x*, *y*, *z*) ∈ *A* ⟹ (*x*, *y*, *z*) ∈ *B*) ⟹ *A* ⊆ *B* ⟨*proof*⟩

lemma *aux*: (⋃ *a*∈*P*. {(*x*,*y*,*z*). *x* = *fst a* ∧ *y* = *snd a* ∧ *Q a z*}) = {(*x*,*y*,*z*). (*x*,*y*) ∈ *P* ∧ *Q* (*x*,*y*) *z*} (**is** *?P* = *?Q*)
⟨*proof*⟩

lemma *finite-imp-finite-DP-on'*:
assumes *finite R*
shows *finite* {(*l*, *r*, *u*).
∃ *h us*. *u* = *Fun h us* ∧ (*l*, *r*) ∈ *R* ∧ *r* ⊵ *u* ∧ (*h*, *length us*) ∈ *F* ∧ ¬ (*l* ▷ *u*)}
⟨*proof*⟩

lemma *card-image-le'*:
assumes *finite S*
shows *card* (⋃ *y*∈*S*.{*x*. *x* = *f y*}) ≤ *card S*
⟨*proof*⟩

lemma *subteq-of-map-imp-map*: *map-funs-term g s* ⊵ *t* ⟹ ∃ *u*. *t* = *map-funs-term g u*
⟨*proof*⟩

lemma *map-funs-term-inj*:
assumes *inj* (*fg* :: ('*f* ⇒ '*g*))
shows *inj* (*map-funs-term fg*)
⟨*proof*⟩

lemma *rsteps-closed-ctxt*:
assumes (*s*, *t*) ∈ (*rstep R*)*
shows (*C*⟨*s*⟩, *C*⟨*t*⟩) ∈ (*rstep R*)*

⟨*proof*⟩

lemma *one-imp-ctxt-closed*: **assumes** *one*: $\bigwedge$ *f bef s t aft*. (*s*,*t*) ∈ *r* ⟹ (*Fun f* (*bef* @ *s* # *aft*), *Fun f* (*bef* @ *t* # *aft*)) ∈ *r*
shows *ctxt.closed r*
⟨*proof*⟩

lemma *ctxt-closed-nrrstep* [*intro*]: *ctxt.closed* (*nrrstep R*)
⟨*proof*⟩

definition *all-ctxt-closed* :: *'f sig* ⇒ (*'f*, *'v*) *trs* ⇒ *bool* **where**
all-ctxt-closed F r ⟷ (∀ *f ts ss*. (*f*, *length ss*) ∈ *F* ⟶ *length ts* = *length ss* ⟶
(∀ *i*. *i* < *length ts* ⟶ (*ts* ! *i*, *ss* ! *i*) ∈ *r*) ⟶ (∀ *i*. *i* < *length ts* ⟶ *funas-term* (*ts* ! *i*) ∪ *funas-term* (*ss* ! *i*) ⊆ *F*) ⟶ (*Fun f ts*, *Fun f ss*) ∈ *r*) ∧ (∀ *x*. (*Var x*, *Var x*) ∈ *r*)

lemma *all-ctxt-closedD*: *all-ctxt-closed F r* ⟹ (*f*,*length ss*) ∈ *F* ⟹ *length ts* = *length ss*
⟹ ⟦⋀ *i*. *i* < *length ts* ⟹ (*ts* ! *i*, *ss* ! *i*) ∈ *r* ⟧
⟹ ⟦⋀ *i*. *i* < *length ts* ⟹ *funas-term* (*ts* ! *i*) ⊆ *F* ⟧
⟹ ⟦⋀ *i*. *i* < *length ts* ⟹ *funas-term* (*ss* ! *i*) ⊆ *F* ⟧
⟹ (*Fun f ts*, *Fun f ss*) ∈ *r*
⟨*proof*⟩

lemma *all-ctxt-closed-sig-reflE*: **assumes** *all*: *all-ctxt-closed F r*
shows *funas-term t* ⊆ *F* ⟹ (*t*,*t*) ∈ *r*
⟨*proof*⟩

lemma *all-ctxt-closed-reflE*: **assumes** *all*: *all-ctxt-closed UNIV r*
shows (*t*,*t*) ∈ *r*
⟨*proof*⟩

lemma *all-ctxt-closed-relcomp*: **assumes** *all-ctxt-closed UNIV R all-ctxt-closed UNIV S*
shows *all-ctxt-closed UNIV* (*R O S*)
⟨*proof*⟩

lemma *all-ctxt-closed-relpow*:
assumes *acc*:*all-ctxt-closed UNIV Q*
shows *all-ctxt-closed UNIV* (*Q* ^^ *n*)
⟨*proof*⟩

lemma *all-ctxt-closed-subst-step-sig*:
fixes *r* :: (*'f*, *'v*) *trs* **and** *t* :: (*'f*, *'v*) *term*
assumes *all*: *all-ctxt-closed F r*
and *sig*: *funas-term t* ⊆ *F*
and *steps*: ⋀ *x*. *x* ∈ *vars-term t* ⟹ (*σ x*, *τ x*) ∈ *r*
and *sig-subst*: ⋀ *x*. *x* ∈ *vars-term t* ⟹ *funas-term* (*σ x*) ∪ *funas-term* (*τ x*)

$\subseteq F$
shows $(t \cdot \sigma,\ t \cdot \tau) \in r$
⟨*proof*⟩

lemma *all-ctxt-closed-subst-step*:
fixes $r :: ('f,\ 'v)\ trs$ **and** $t :: ('f,\ 'v)\ term$
assumes *all*: *all-ctxt-closed UNIV r*
and *steps*: $\bigwedge x.\ x \in vars\text{-}term\ t \Longrightarrow (\sigma\ x,\ \tau\ x) \in r$
shows $(t \cdot \sigma,\ t \cdot \tau) \in r$
⟨*proof*⟩

lemma *all-ctxt-closed-ctxtE*: **assumes** *all*: *all-ctxt-closed F R*
and *Fs*: *funas-term s* $\subseteq F$
and *Ft*: *funas-term t* $\subseteq F$
and *step*: $(s,t) \in R$
shows *funas-ctxt C* $\subseteq F \Longrightarrow (C\ \langle\ s\ \rangle,\ C\ \langle\ t\ \rangle) \in R$
⟨*proof*⟩

lemma *trans-ctxt-sig-imp-all-ctxt-closed*: **assumes** *tran*: *trans r*
and *refl*: $\bigwedge t.$ *funas-term t* $\subseteq F \Longrightarrow (t,t) \in r$
and *ctxt*: $\bigwedge C\ s\ t.$ *funas-ctxt C* $\subseteq F \Longrightarrow$ *funas-term s* $\subseteq F \Longrightarrow$ *funas-term t* $\subseteq F \Longrightarrow (s,t) \in r \Longrightarrow (C\ \langle\ s\ \rangle,\ C\ \langle\ t\ \rangle) \in r$
shows *all-ctxt-closed F r*
⟨*proof*⟩

lemma *trans-ctxt-imp-all-ctxt-closed*: **assumes** *tran*: *trans r*
and *refl*: *refl r*
and *ctxt*: *ctxt.closed r*
shows *all-ctxt-closed F r*
⟨*proof*⟩

lemma *all-ctxt-closed-rsteps*[*intro*]: *all-ctxt-closed F* $((rstep\ r)^*)$
⟨*proof*⟩

lemma *subst-rsteps-imp-rsteps*:
fixes $\sigma :: ('f,\ 'v)\ subst$
assumes $\bigwedge x.\ x \in vars\text{-}term\ t \Longrightarrow (\sigma\ x,\ \tau\ x) \in (rstep\ R)^*$
shows $(t \cdot \sigma,\ t \cdot \tau) \in (rstep\ R)^*$
⟨*proof*⟩

lemma *rtrancl-trancl-into-trancl*:
assumes *len*: *length ts* = *length ss*
and *steps*: $\forall\ i < length\ ts.\ (ts\ !\ i,\ ss\ !\ i) \in R^*$
and *i*: $i < length\ ts$
and *step*: $(ts\ !\ i,\ ss\ !\ i) \in R^+$
and *ctxt*: *ctxt.closed R*
shows $(Fun\ f\ ts,\ Fun\ f\ ss) \in R^+$
⟨*proof*⟩

lemma *SN-ctxt-apply-imp-SN-ctxt-to-term-list-gen*:
assumes *ctxt*: *ctxt.closed r*
assumes *SN*: *SN-on r* {*C*⟨*t*⟩}
shows *SN-on r* (*set* (*ctxt-to-term-list C*))
⟨*proof*⟩

lemma *rstep-subset*: *ctxt.closed R'* ⟹ *subst.closed R'* ⟹ *R* ⊆ *R'* ⟹ *rstep R* ⊆ *R'* ⟨*proof*⟩

lemma *trancl-rstep-ctxt*:
(*s*,*t*) ∈ (*rstep R*)$^+$ ⟹ (*C*⟨*s*⟩, *C*⟨*t*⟩) ∈ (*rstep R*)$^+$
⟨*proof*⟩

lemma *args-steps-imp-steps-gen*:
assumes *ctxt*: ⋀ *bef s t aft*. (*s*, *t*) ∈ *r* (*length bef*) ⟹
length ts = *Suc* (*length bef* + *length aft*) ⟹
(*Fun f* (*bef* @ (*s* :: (′*f*, ′*v*) *term*) # *aft*), *Fun f* (*bef* @ *t* # *aft*)) ∈ *R**
and *len*: *length ss* = *length ts*
and *args*: ⋀ *i*. *i* < *length ts* ⟹ (*ss* ! *i*, *ts* ! *i*) ∈ (*r i*)*
shows (*Fun f ss*, *Fun f ts*) ∈ *R**
⟨*proof*⟩

lemma *args-steps-imp-steps*:
assumes *ctxt*: *ctxt.closed R*
and *len*: *length ss* = *length ts* **and** *args*: ∀ *i*<*length ss*. (*ss*!*i*, *ts*!*i*) ∈ *R**
shows (*Fun f ss*, *Fun f ts*) ∈ *R**
⟨*proof*⟩

lemmas *args-rsteps-imp-rsteps* = *args-steps-imp-steps* [*OF ctxt-closed-rstep*]

lemma *replace-at-subst-steps*:
fixes *σ τ* :: (′*f*, ′*v*) *subst*
assumes *acc*: *all-ctxt-closed UNIV r*
and *refl*: *refl r*
and ∗: ⋀*x*. (*σ x*, *τ x*) ∈ *r*
and *p* ∈ *poss t*
and *t* |- *p* = *Var x*
shows (*replace-at* (*t* · *σ*) *p* (*τ x*), *t* · *τ*) ∈ *r*
⟨*proof*⟩

lemma *replace-at-subst-rsteps*:
fixes *σ τ* :: (′*f*, ′*v*) *subst*
assumes ∗: ⋀*x*. (*σ x*, *τ x*) ∈ (*rstep R*)*
and *p* ∈ *poss t*
and *t* |- *p* = *Var x*
shows (*replace-at* (*t* · *σ*) *p* (*τ x*), *t* · *τ*) ∈ (*rstep R*)*
⟨*proof*⟩

lemma *substs-rsteps*:

assumes $\bigwedge x.\ (\sigma\ x,\ \tau\ x) \in (rstep\ R)^*$
shows $(t \cdot \sigma,\ t \cdot \tau) \in (rstep\ R)^*$
⟨*proof*⟩

lemma *nrrstep-Fun-imp-arg-rstep*:
fixes *ss* :: (*'f*,*'v*)*term list*
assumes (*Fun f ss*,*Fun f ts*) ∈ *nrrstep R* (**is** (*?s*,*?t*) ∈ *nrrstep R*)
shows ∃ *C i*. *i* < *length ss* ∧ (*ss*!*i*,*ts*!*i*) ∈ *rstep R* ∧ *C*⟨*ss*!*i*⟩ = *Fun f ss* ∧ *C*⟨*ts*!*i*⟩ = *Fun f ts*
⟨*proof*⟩

lemma *pair-fun-eq*[*simp*]:
fixes *f* :: *'a* ⇒ *'b* **and** *g* :: *'b* ⇒ *'a*
shows ((λ(*x*,*y*). (*x*,*f y*)) ∘ (λ(*x*,*y*). (*x*,*g y*))) = (λ(*x*,*y*). (*x*,(*f* ∘ *g*) *y*)) (**is** *?f* = *?g*)
⟨*proof*⟩

lemma *restrict-singleton*:
assumes *x* ∈ *subst-domain* σ **shows** ∃ *t*. σ |*s* {*x*} = (λ*y*. *if y* = *x then t else Var y*)
⟨*proof*⟩

definition *rstep-r-c-s* :: (*'f*,*'v*)*rule* ⇒ (*'f*,*'v*)*ctxt* ⇒ (*'f*,*'v*)*subst* ⇒ (*'f*,*'v*)*term rel*
where *rstep-r-c-s r C* σ = {(*s*,*t*) | *s t*. *s* = *C*⟨*fst r* · σ⟩ ∧ *t* = *C*⟨*snd r* · σ⟩}

lemma *rstep-iff-rstep-r-c-s*: ((*s*,*t*) ∈ *rstep R*) = (∃ *l r C* σ. (*l*,*r*) ∈ *R* ∧ (*s*,*t*) ∈ *rstep-r-c-s* (*l*,*r*) *C* σ) (**is** *?left* = *?right*)
⟨*proof*⟩

lemma *rstep-subset-characterization*:
(*rstep R* ⊆ *rstep S*) = (∀ *l r*. (*l*,*r*) ∈ *R* ⟶ (∃ *l' r' C* σ . (*l'*,*r'*) ∈ *S* ∧ *l* = *C*⟨*l'* · σ⟩ ∧ *r* = *C*⟨*r'* · σ⟩)) (**is** *?left* = *?right*)
⟨*proof*⟩

lemma *rstep-preserves-funas-terms-var-cond*:
assumes *funas-trs R* ⊆ *F* **and** *funas-term s* ⊆ *F* **and** (*s*,*t*) ∈ *rstep R*
and *wf*: ⋀ *l r*. (*l*,*r*) ∈ *R* ⟹ *vars-term r* ⊆ *vars-term l*
shows *funas-term t* ⊆ *F*
⟨*proof*⟩

lemma *rstep-preserves-funas-terms*:
assumes *funas-trs R* ⊆ *F* **and** *funas-term s* ⊆ *F* **and** (*s*,*t*) ∈ *rstep R*
and *wf*: *wf-trs R*
shows *funas-term t* ⊆ *F*
⟨*proof*⟩

lemma *rsteps-preserve-funas-terms-var-cond*:
assumes *F*: *funas-trs R* ⊆ *F* **and** *s*: *funas-term s* ⊆ *F* **and** *steps*: (*s*,*t*) ∈ (*rstep R*)*

and *wf*: $\bigwedge l\ r.\ (l,r) \in R \Longrightarrow$ *vars-term* $r \subseteq$ *vars-term* l
shows *funas-term* $t \subseteq F$
⟨*proof*⟩

lemma *rsteps-preserve-funas-terms*:
assumes F: *funas-trs* $R \subseteq F$ **and** s: *funas-term* $s \subseteq F$
and *steps*: $(s,t) \in (rstep\ R)^*$ **and** *wf*: *wf-trs* R
shows *funas-term* $t \subseteq F$
⟨*proof*⟩

lemma *no-Var-rstep* [*simp*]:
assumes *wf-trs* R **and** $(Var\ x,\ t) \in rstep\ R$ **shows** *False*
⟨*proof*⟩

lemma *lhs-wf*:
assumes R: $(l,\ r) \in R$ **and** *funas-trs* $R \subseteq F$
shows *funas-term* $l \subseteq\ F$
⟨*proof*⟩

lemma *rhs-wf*:
assumes R: $(l,\ r) \in R$ **and** *funas-trs* $R \subseteq F$
shows *funas-term* $r \subseteq F$
⟨*proof*⟩

lemma *supt-map-funs-term* [*intro*]:
assumes $t \rhd s$
shows *map-funs-term fg* $t \rhd$ *map-funs-term fg* s
⟨*proof*⟩

lemma *nondef-root-imp-arg-step*:
assumes $(Fun\ f\ ss,\ t) \in rstep\ R$
and *wf*: $\forall (l,\ r) \in R.$ *is-Fun* l
and *ndef*: $\neg$ *defined* $R\ (f,$ *length ss*$)$
shows $\exists i <$ *length ss*. $(ss\ !\ i,\ t\ |\text{-}\ [i]) \in rstep\ R$
$\wedge\ t = Fun\ f\ (take\ i\ ss\ @\ (t\ |\text{-}\ [i])\ \#\ drop\ (Suc\ i)\ ss)$
⟨*proof*⟩

lemma *nondef-root-imp-arg-steps*:
assumes $(Fun\ f\ ss,t) \in (rstep\ R)^*$
and *wf*: $\forall (l,\ r) \in R.$ *is-Fun* l
and $\neg$ *defined* $R\ (f,$*length ss*$)$
shows $\exists ts.$ *length ts* $=$ *length ss* $\wedge\ t = Fun\ f\ ts \wedge (\forall i <$ *length ss*. $(ss!i,ts!i) \in (rstep\ R)^*)$
⟨*proof*⟩

lemma *rstep-imp-nrrstep*:
assumes *is-Fun s* **and** $\neg$ *defined* R (*the* (*root s*)) **and** $\forall (l,r) \in R.$ *is-Fun* l
and $(s,\ t) \in rstep\ R$
shows $(s,\ t) \in nrrstep\ R$

⟨*proof*⟩

lemma *rsteps-imp-nrrsteps*:
assumes *is-Fun s* **and** ¬ *defined R* (*the* (*root s*))
and *no-vars*: ∀ (*l*, *r*)∈*R*. *is-Fun l*
and (*s*, *t*) ∈ (*rstep R*)*
shows (*s*, *t*) ∈ (*nrrstep R*)*
⟨*proof*⟩

lemma *left-var-imp-not-SN*:
fixes *R* :: ('*f*,'*v*)*trs* **and** *t* :: ('*f*, '*v*) *term*
assumes (*Var y*, *r*) ∈ *R* (**is** (*?y*, -) ∈ -)
shows ¬ (*SN-on* (*rstep R*) {*t*})
⟨*proof*⟩

lemma *not-SN-subt-imp-not-SN*:
assumes *ctxt*: *ctxt.closed R* **and** *SN*: ¬ *SN-on R* {*t*} **and** *sub*: *s* ⊵ *t*
shows ¬ *SN-on R* {*s*}
⟨*proof*⟩

lemma *root-Some*:
assumes *root t* = *Some fn*
obtains *ss* **where** *length ss* = *snd fn* **and** *t* = *Fun* (*fst fn*) *ss*
⟨*proof*⟩

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

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

The set of minimally nonterminating terms with respect to a relation R.

definition *Tinf* :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *terms*
where
Tinf R = {*t*. ¬ *SN-on R* {*t*} ∧ (∀ *s* ◁ *t*. *SN-on R* {*s*})}

lemma *not-SN-imp-subt-Tinf*:
assumes ¬ *SN-on R* {*s*} **shows** ∃ *t*⊴*s*. *t* ∈ *Tinf R*
⟨*proof*⟩

lemma *not-SN-imp-Tinf*:
assumes ¬ *SN R* **shows** ∃ *t*. *t* ∈ *Tinf R*
⟨*proof*⟩

lemma *ctxt-of-pos-term-map-funs-term-conv* [*iff*]:

assumes $p \in poss\ s$
shows *map-funs-ctxt fg* (*ctxt-of-pos-term p s*) = (*ctxt-of-pos-term p* (*map-funs-term fg s*))
⟨*proof*⟩

lemma *var-rewrite-imp-not-SN*:
assumes *sn*: *SN-on* (*rstep R*) {*u*} **and** *step*: (*t*, *s*) ∈ *rstep R*
shows *is-Fun t*
⟨*proof*⟩

lemma *rstep-id*: *rstep Id* = *Id* ⟨*proof*⟩

lemma *map-funs-rule-id* [*simp*]: *map-funs-rule id* = *id*
⟨*proof*⟩

lemma *map-funs-trs-id* [*simp*]: *map-funs-trs id* = *id*
⟨*proof*⟩

definition *sig-step* :: *'f sig* ⇒ (*'f*, *'v*) *trs* ⇒ (*'f*, *'v*) *trs* **where**
sig-step F R = {(*a*, *b*). (*a*, *b*) ∈ *R* ∧ *funas-term a* ⊆ *F* ∧ *funas-term b* ⊆ *F*}

lemma *sig-step-union*: *sig-step F* (*R* ∪ *S*) = *sig-step F R* ∪ *sig-step F S*
⟨*proof*⟩

lemma *sig-step-UNIV*: *sig-step UNIV R* = *R* ⟨*proof*⟩

lemma *sig-stepI*[*intro*]: (*a*,*b*) ∈ *R* ⟹ *funas-term a* ⊆ *F* ⟹ *funas-term b* ⊆ *F* ⟹ (*a*,*b*) ∈ *sig-step F R* ⟨*proof*⟩

lemma *sig-stepE*[*elim*,*consumes 1*]: (*a*,*b*) ∈ *sig-step F R* ⟹ ⟦(*a*,*b*) ∈ *R* ⟹ *funas-term a* ⊆ *F* ⟹ *funas-term b* ⊆ *F* ⟹ *P*⟧ ⟹ *P* ⟨*proof*⟩

lemma *all-ctxt-closed-sig-rsteps* [*intro*]:
fixes *R* :: (*'f*, *'v*) *trs*
shows *all-ctxt-closed F* ((*sig-step F* (*rstep R*))*) (**is** *all-ctxt-closed - (?R**))
⟨*proof*⟩

lemma *wf-loop-imp-sig-ctxt-rel-not-SN*:
assumes *R*: (*l*,*C*⟨*l*⟩) ∈ *R* **and** *wf-l*: *funas-term l* ⊆ *F*
and *wf-C*: *funas-ctxt C* ⊆ *F*
and *ctxt*: *ctxt.closed R*
shows ¬ *SN-on* (*sig-step F R*) {*l*}
⟨*proof*⟩

lemma *lhs-var-imp-sig-step-not-SN-on*:
assumes *x*: (*Var x*, *r*) ∈ *R* **and** *F*: *funas-trs R* ⊆ *F*
shows ¬ *SN-on* (*sig-step F* (*rstep R*)) {*Var x*}
⟨*proof*⟩

lemma *rhs-free-vars-imp-sig-step-not-SN*:
assumes *R*: $(l,r) \in R$ **and** *free*: $\neg$ *vars-term r* $\subseteq$ *vars-term l*
and *F*: *funas-trs R* $\subseteq$ *F*
shows $\neg$ *SN-on* (*sig-step F* (*rstep R*)) $\{l\}$
⟨*proof*⟩

lemma *lhs-var-imp-rstep-not-SN*: **assumes** (*Var x*,*r*) $\in$ *R* **shows** $\neg$ *SN*(*rstep R*)
⟨*proof*⟩

lemma *rhs-free-vars-imp-rstep-not-SN*:
assumes $(l,r) \in R$ **and** $\neg$ *vars-term r* $\subseteq$ *vars-term l*
shows $\neg$ *SN-on* (*rstep R*) $\{l\}$
⟨*proof*⟩

lemma *free-right-rewrite-imp-not-SN*:
assumes *step*: $(t,s) \in$ *rstep-r-p-s R* (l,r) *p* σ
and *vars*: $\neg$ *vars-term l* $\supseteq$ *vars-term r*
shows $\neg$ *SN-on* (*rstep R*) $\{t\}$
⟨*proof*⟩

lemma *not-SN-on-rstep-subst-apply-term*[*intro*]:
assumes $\neg$ *SN-on* (*rstep R*) $\{t\}$ **shows** $\neg$ *SN-on* (*rstep R*) $\{t \cdot \sigma\}$
⟨*proof*⟩

lemma *SN-rstep-imp-wf-trs*: **assumes** *SN* (*rstep R*) **shows** *wf-trs R*
⟨*proof*⟩

lemma *SN-sig-step-imp-wf-trs*: **assumes** *SN*: *SN* (*sig-step F* (*rstep R*)) **and** *F*: *funas-trs R* $\subseteq$ *F* **shows** *wf-trs R*
⟨*proof*⟩

lemma *rhs-free-vars-imp-rstep-not-SN'*:
assumes $(l, r) \in R$ **and** $\neg$ *vars-term r* $\subseteq$ *vars-term l*
shows $\neg$ *SN* (*rstep R*)
⟨*proof*⟩

lemma *SN-imp-variable-condition*:
assumes *SN* (*rstep R*)
shows $\forall (l, r) \in R.$ *vars-term r* $\subseteq$ *vars-term l*
⟨*proof*⟩

lemma *rstep-cases'*[*consumes 1*, *case-names root nonroot*]:
assumes *rstep*: $(s, t) \in$ *rstep R*
and *root*: $\bigwedge l\ r\ \sigma.\ (l, r) \in R \Longrightarrow l \cdot \sigma = s \Longrightarrow r \cdot \sigma = t \Longrightarrow P$
and *nonroot*: $\bigwedge f$ *ss1 u ss2 v*. *s* = *Fun f* (*ss1* @ *u* # *ss2*) $\Longrightarrow$ *t* = *Fun f* (*ss1* @ *v* # *ss2*) $\Longrightarrow (u, v) \in$ *rstep R* $\Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *NF-Var*: **assumes** *wf*: *wf-trs R* **shows** (*Var x*, *t*) ∉ *rstep R*
⟨*proof*⟩

lemma *rstep-cases-Fun'*[*consumes 2*, *case-names root nonroot*]:
assumes *wf*: *wf-trs R*
and *rstep*: (*Fun f ss*,*t*) ∈ *rstep R*
and *root'*: ⋀*ls r σ*. (*Fun f ls*,*r*) ∈ *R* ⟹ *map* (λ*t*. *t*·*σ*) *ls* = *ss* ⟹ *r*·*σ* = *t* ⟹ *P*
and *nonroot'*: ⋀*i u*. *i* < *length ss* ⟹ *t* = *Fun f* (*take i ss*@*u*#*drop* (*Suc i*) *ss*) ⟹ (*ss*!*i*,*u*) ∈ *rstep R* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *rstep-preserves-undefined-root*:
assumes *wf-trs R* **and** ¬ *defined R* (*f*, *length ss*) **and** (*Fun f ss*, *t*) ∈ *rstep R*
shows ∃ *ts*. *length ts* = *length ss* ∧ *t* = *Fun f ts*
⟨*proof*⟩

lemma *rstep-ctxt-imp-nrrstep*: **assumes** *step*: (*s*,*t*) ∈ *rstep R* **and** *C*: *C* ≠ □ **shows** (*C*⟨*s*⟩,*C*⟨*t*⟩) ∈ *nrrstep R*
⟨*proof*⟩

lemma *rsteps-ctxt-imp-nrrsteps*: **assumes** *steps*: (*s*,*t*) ∈ (*rstep R*)* **and** *C*: *C* ≠ □ **shows** (*C*⟨*s*⟩,*C*⟨*t*⟩) ∈ (*nrrstep R*)*
⟨*proof*⟩

lemma *nrrstep-mono*:
assumes *R* ⊆ *R'*
shows *nrrstep R* ⊆ *nrrstep R'*
⟨*proof*⟩

lemma *rrstepE*:
assumes (*s*, *t*) ∈ *rrstep R*
obtains *l* **and** *r* **and** *σ* **where** (*l*, *r*) ∈ *R* **and** *s* = *l* · *σ* **and** *t* = *r* · *σ*
⟨*proof*⟩

lemma *nrrstepE*:
assumes (*s*, *t*) ∈ *nrrstep R*
obtains *C* **and** *l* **and** *r* **and** *σ* **where** *C* ≠ □ **and** (*l*, *r*) ∈ *R*
and *s* = *C*⟨*l* · *σ*⟩ **and** *t* = *C*⟨*r* · *σ*⟩
⟨*proof*⟩

lemma *singleton-subst-restrict* [*simp*]:
subst x s |*s* {*x*} = *subst x s*
⟨*proof*⟩

lemma *singleton-subst-map* [*simp*]:
f ∘ *subst x s* = (*f* ∘ *Var*)(*x* := *f s*) ⟨*proof*⟩

lemma *subst-restrict-vars* [*simp*]:
$(\lambda z.\ \textit{if}\ z \in V\ \textit{then}\ f\ z\ \textit{else}\ g\ z)\ |s\ V = f\ |s\ V$
⟨*proof*⟩

lemma *subst-restrict-restrict* [*simp*]:
assumes $V \cap W = \{\}$
shows $((\lambda z.\ \textit{if}\ z \in V\ \textit{then}\ f\ z\ \textit{else}\ g\ z)\ |s\ W) = g\ |s\ W$
⟨*proof*⟩

lemma *rstep-rstep*: $\textit{rstep}\ (\textit{rstep}\ R) = \textit{rstep}\ R$
⟨*proof*⟩

lemma *rstep-trancl-distrib*: $\textit{rstep}\ (R^+) \subseteq (\textit{rstep}\ R)^+$
⟨*proof*⟩

lemma *rsteps-closed-subst*:
assumes $(s,\ t) \in (\textit{rstep}\ R)^*$
shows $(s \cdot \sigma,\ t \cdot \sigma) \in (\textit{rstep}\ R)^*$
⟨*proof*⟩

lemma *join-subst*:
$\textit{subst.closed}\ r \Longrightarrow (s,\ t) \in r^{\downarrow} \Longrightarrow (s \cdot \sigma,\ t \cdot \sigma) \in r^{\downarrow}$
⟨*proof*⟩

lemma *join-subst-rstep* [*intro*]:
$(s,\ t) \in (\textit{rstep}\ R)^{\downarrow} \Longrightarrow (s \cdot \sigma,\ t \cdot \sigma) \in (\textit{rstep}\ R)^{\downarrow}$
⟨*proof*⟩

lemma *join-ctxt* [*intro*]:
assumes $(s,\ t) \in (\textit{rstep}\ R)^{\downarrow}$
shows $(C\langle s\rangle,\ C\langle t\rangle) \in (\textit{rstep}\ R)^{\downarrow}$
⟨*proof*⟩

lemma *rstep-simps*:
$\textit{rstep}\ (R^{=}) = (\textit{rstep}\ R)^{=}$
$\textit{rstep}\ (\textit{rstep}\ R) = \textit{rstep}\ R$
$\textit{rstep}\ (R \cup S) = \textit{rstep}\ R \cup \textit{rstep}\ S$
$\textit{rstep}\ \textit{Id} = \textit{Id}$
$\textit{rstep}\ (R^{\leftrightarrow}) = (\textit{rstep}\ R)^{\leftrightarrow}$
⟨*proof*⟩

lemma *rstep-rtrancl-idemp* [*simp*]:
$\textit{rstep}\ ((\textit{rstep}\ R)^*) = (\textit{rstep}\ R)^*$
⟨*proof*⟩

lemma *all-ctxt-closed-rstep-conversion*:
$\textit{all-ctxt-closed}\ \textit{UNIV}\ ((\textit{rstep}\ R)^{\leftrightarrow *})$

⟨*proof*⟩

definition *instance-rule* :: ('*f*, '*v*) *rule* ⇒ ('*f*, '*w*) *rule* ⇒ *bool* **where**
[*code del*]: *instance-rule lr st* ⟷ (∃ *σ*. *fst lr* = *fst st* · *σ* ∧ *snd lr* = *snd st* · *σ*)

definition *eq-rule-mod-vars* :: ('*f*, '*v*) *rule* ⇒ ('*f*, '*v*) *rule* ⇒ *bool* **where**
eq-rule-mod-vars lr st ⟷ *instance-rule lr st* ∧ *instance-rule st lr*

notation *eq-rule-mod-vars* ((-/ =$_v$ -) [*51*,*51*] *50*)

lemma *instance-rule-var-cond*: **assumes** *eq*: *instance-rule* (*s*,*t*) (*l*,*r*)
and *vars*: *vars-term r* ⊆ *vars-term l*
shows *vars-term t* ⊆ *vars-term s*
⟨*proof*⟩

lemma *instance-rule-rstep*: **assumes** *step*: (*s*,*t*) ∈ *rstep* {*lr*}
and *bex*: *Bex R* (*instance-rule lr*)
shows (*s*,*t*) ∈ *rstep R*
⟨*proof*⟩

lemma *eq-rule-mod-vars-var-cond*: **assumes** *eq*: (*l*,*r*) =$_v$ (*s*,*t*)
and *vars*: *vars-term r* ⊆ *vars-term l*
shows *vars-term t* ⊆ *vars-term s*
⟨*proof*⟩

lemma *eq-rule-mod-varsE*[*elim*]: **fixes** *l* :: ('*f*,'*v*)*term*
assumes (*l*,*r*) =$_v$ (*s*,*t*)
shows ∃ *σ τ*. *l* = *s* · *σ* ∧ *r* = *t* · *σ* ∧ *s* = *l* · *τ* ∧ *t* = *r* · *τ* ∧ *range σ* ⊆ *range Var* ∧ *range τ* ⊆ *range Var*
⟨*proof*⟩

4.5 Linear and Left-Linear TRSs

definition
linear-trs :: ('*f*, '*v*) *trs* ⇒ *bool*
where
linear-trs R ≡ ∀(*l*, *r*)∈*R*. *linear-term l* ∧ *linear-term r*

lemma *linear-trsE*[*elim*,*consumes 1*]: *linear-trs R* ⟹ (*l*,*r*) ∈ *R* ⟹ *linear-term l* ∧ *linear-term r*
⟨*proof*⟩

lemma *linear-trsI*[*intro*]: ⟦ ⋀ *l r*. (*l*,*r*) ∈ *R* ⟹ *linear-term l* ∧ *linear-term r*⟧ ⟹ *linear-trs R*
⟨*proof*⟩

definition

left-linear-trs :: $('f, 'v)$ *trs* $\Rightarrow$ *bool*
where
left-linear-trs R $\longleftrightarrow$ $(\forall (l, r) \in R.$ *linear-term l*$)$

lemma *left-linear-trs-union*: *left-linear-trs* $(R \cup S) =$ (*left-linear-trs R* $\wedge$ *left-linear-trs S*)
⟨*proof*⟩

lemma *left-linear-mono*: **assumes** *left-linear-trs S* **and** $R \subseteq S$ **shows** *left-linear-trs R*
⟨*proof*⟩

lemma *left-linear-map-funs-trs*[*simp*]: *left-linear-trs* (*map-funs-trs f R*) = *left-linear-trs R*
⟨*proof*⟩

lemma *left-linear-weak-match-rstep*:
assumes *rstep*: $(u, v) \in$ *rstep R*
and *weak-match*: *weak-match s u*
and *ll*: *left-linear-trs R*
shows $\exists t. (s, t) \in$ *rstep R* $\wedge$ *weak-match t v*
⟨*proof*⟩

context
begin

private fun *S* **where**
S R s t 0 = *s*
| *S R s t* (*Suc i*) = (*SOME u*. (*S R s t i*,*u*) ∈ *rstep R* ∧ *weak-match u* (*t*(*Suc i*)))

lemma *weak-match-SN*:
assumes *wm*: *weak-match s t*
and *ll*: *left-linear-trs R*
and *SN*: *SN-on* (*rstep R*) {*s*}
shows *SN-on* (*rstep R*) {*t*}
⟨*proof*⟩
end

lemma *lhs-notin-NF-rstep*: $(l, r) \in R \Longrightarrow l \notin$ *NF* (*rstep R*) ⟨*proof*⟩

lemma *NF-instance*:
assumes $(t \cdot \sigma) \in$ *NF* (*rstep R*) **shows** $t \in$ *NF* (*rstep R*)
⟨*proof*⟩

lemma *NF-subterm*:
assumes $t \in$ *NF* (*rstep R*) **and** $t \trianglerighteq s$
shows $s \in$ *NF* (*rstep R*)
⟨*proof*⟩

abbreviation
lhss :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *terms*
where
lhss R ≡ *fst* ‘ *R*

abbreviation
rhss :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *terms*
where
rhss R ≡ *snd* ‘ *R*

definition *map-funs-trs-wa* :: ('*f* × *nat* ⇒ '*g*) ⇒ ('*f*, '*v*) *trs* ⇒ ('*g*, '*v*) *trs* **where**
map-funs-trs-wa fg R = (λ(*l*, *r*). (*map-funs-term-wa fg l*, *map-funs-term-wa fg r*)) ‘ *R*

lemma *map-funs-trs-wa-union*: *map-funs-trs-wa fg* (*R* ∪ *S*) = *map-funs-trs-wa fg R* ∪ *map-funs-trs-wa fg S*
⟨*proof*⟩

lemma *map-funs-term-wa-compose*: *map-funs-term-wa gh* (*map-funs-term-wa fg t*) = *map-funs-term-wa* (λ (*f*,*n*). *gh* (*fg* (*f*,*n*), *n*)) *t*
⟨*proof*⟩

lemma *map-funs-trs-wa-compose*: *map-funs-trs-wa gh* (*map-funs-trs-wa fg R*) = *map-funs-trs-wa* (λ (*f*,*n*). *gh* (*fg* (*f*,*n*), *n*)) *R* (**is** *?L* = *map-funs-trs-wa ?fgh R*)
⟨*proof*⟩

lemma *map-funs-trs-wa-funas-trs-id*: **assumes** *R*: *funas-trs R* ⊆ *F*
and *id*: ⋀ *g n*. (*g*,*n*) ∈ *F* ⟹ *f* (*g*,*n*) = *g*
shows *map-funs-trs-wa f R* = *R*
⟨*proof*⟩

lemma *map-funs-trs-wa-rstep*: **assumes** *step*:(*s*,*t*) ∈ *rstep R*
shows (*map-funs-term-wa fg s*,*map-funs-term-wa fg t*) ∈ *rstep* (*map-funs-trs-wa fg R*)
⟨*proof*⟩

lemma *map-funs-trs-wa-rsteps*: **assumes** *step*:(*s*,*t*) ∈ (*rstep R*)*
shows (*map-funs-term-wa fg s*,*map-funs-term-wa fg t*) ∈ (*rstep* (*map-funs-trs-wa fg R*))*
⟨*proof*⟩

lemma *rstep-ground*:
assumes *wf-trs*: ⋀*l r*. (*l*, *r*) ∈ *R* ⟹ *vars-term r* ⊆ *vars-term l*
and *ground*: *ground s*
and *step*: (*s*, *t*) ∈ *rstep R*
shows *ground t*
⟨*proof*⟩

lemma *rsteps-ground*:
assumes *wf-trs*: $\bigwedge l\ r.\ (l,\ r) \in R \Longrightarrow \mathit{vars\text{-}term}\ r \subseteq \mathit{vars\text{-}term}\ l$
and *ground*: *ground s*
and *steps*: $(s,\ t) \in (\mathit{rstep}\ R)^*$
shows *ground t*
⟨*proof*⟩

definition *locally-terminating* :: $('f,'v)\mathit{trs} \Rightarrow \mathit{bool}$
where $\mathit{locally\text{-}terminating}\ R \equiv \forall\ F.\ \mathit{finite}\ F \longrightarrow \mathit{SN}\ (\mathit{sig\text{-}step}\ F\ (\mathit{rstep}\ R))$

definition $\mathit{non\text{-}collapsing}\ R \longleftrightarrow (\forall\ lr \in R.\ \mathit{is\text{-}Fun}\ (\mathit{snd}\ lr))$

lemma *supt-rstep-stable*:
assumes $(s,\ t) \in \{\rhd\} \cup \mathit{rstep}\ R$
shows $(s \cdot \sigma,\ t \cdot \sigma) \in \{\rhd\} \cup \mathit{rstep}\ R$
⟨*proof*⟩

lemma *supt-rstep-trancl-stable*:
assumes $(s,\ t) \in (\{\rhd\} \cup \mathit{rstep}\ R)^+$
shows $(s \cdot \sigma,\ t \cdot \sigma) \in (\{\rhd\} \cup \mathit{rstep}\ R)^+$
⟨*proof*⟩

lemma *supt-rsteps-stable*:
assumes $(s,\ t) \in (\{\rhd\} \cup \mathit{rstep}\ R)^*$
shows $(s \cdot \sigma,\ t \cdot \sigma) \in (\{\rhd\} \cup \mathit{rstep}\ R)^*$
⟨*proof*⟩

lemma *eq-rule-mod-vars-refl*[*simp*]: $r =_v r$
⟨*proof*⟩

lemma *instance-rule-refl*[*simp*]: *instance-rule r r*
⟨*proof*⟩

lemma *is-Fun-Fun-conv*: $\mathit{is\text{-}Fun}\ t = (\exists f\ ts.\ t = \mathit{Fun}\ f\ ts)$ ⟨*proof*⟩

lemma *wf-trs-def'*:
$\mathit{wf\text{-}trs}\ R = (\forall\,(l,\ r) \in R.\ \mathit{is\text{-}Fun}\ l \wedge \mathit{vars\text{-}term}\ r \subseteq \mathit{vars\text{-}term}\ l)$
⟨*proof*⟩

definition *wf-rule* :: $('f,\ 'v)\ \mathit{rule} \Rightarrow \mathit{bool}$ **where**
$\mathit{wf\text{-}rule}\ r \longleftrightarrow \mathit{is\text{-}Fun}\ (\mathit{fst}\ r) \wedge \mathit{vars\text{-}term}\ (\mathit{snd}\ r) \subseteq \mathit{vars\text{-}term}\ (\mathit{fst}\ r)$

definition *wf-rules* :: $('f,\ 'v)\ \mathit{trs} \Rightarrow ('f,\ 'v)\ \mathit{trs}$ **where**
$\mathit{wf\text{-}rules}\ R = \{r.\ r \in R \wedge \mathit{wf\text{-}rule}\ r\}$

lemma *wf-trs-wf-rules*[*simp*]: *wf-trs* (*wf-rules R*)
⟨*proof*⟩

lemma *wf-rules-subset*[*simp*]: $\mathit{wf\text{-}rules}\ R \subseteq R$

⟨*proof*⟩

fun *wf-reltrs* :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *trs* ⇒ *bool* **where**
wf-reltrs R S = (
wf-trs R ∧ (*R* ≠ {} ⟶ (∀ *l r*. (*l*, *r*) ∈ *S* ⟶ *vars-term r* ⊆ *vars-term l*)))

lemma *SN-rel-imp-wf-reltrs*:
assumes *SN-rel*: *SN-rel* (*rstep R*) (*rstep S*)
shows *wf-reltrs R S*
⟨*proof*⟩

lemmas *rstep-wf-rules-subset* = *rstep-mono*[*OF wf-rules-subset*]

definition *map-vars-trs* :: ('*v* ⇒ '*w*) ⇒ ('*f*, '*v*) *trs* ⇒ ('*f*, '*w*) *trs* **where**
map-vars-trs f R = (λ (*l*, *r*). (*map-vars-term f l*, *map-vars-term f r*)) ‘ *R*

lemma *map-vars-trs-rstep*:
assumes (*s*, *t*) ∈ *rstep* (*map-vars-trs f R*) (**is** - ∈ *rstep ?R*)
shows (*s* · τ, *t* · τ) ∈ *rstep R*
⟨*proof*⟩

lemma *map-vars-rsteps*:
assumes (*s*,*t*) ∈ (*rstep* (*map-vars-trs f R*))* (**is** - ∈ (*rstep ?R*)*)
shows (*s* · τ, *t* · τ) ∈ (*rstep R*)*
⟨*proof*⟩

lemma *rsteps-subst-closed*: (*s*,*t*) ∈ (*rstep R*)+ ⟹ (*s* · σ, *t* · σ) ∈ (*rstep R*)+
⟨*proof*⟩

lemma *supteq-rtrancl-supt*:
(*R*+ *O* {⊵}) ⊆ ({⊳} ∪ *R*)+ (**is** *?l* ⊆ *?r*)
⟨*proof*⟩

lemma *rrstepI*[*intro*]: (*l*, *r*) ∈ *R* ⟹ *s* = *l* · σ ⟹ *t* = *r* · σ ⟹ (*s*, *t*) ∈ *rrstep R*
⟨*proof*⟩

lemma *CS-rrstep-conv*: *subst.closure* = *rrstep*
⟨*proof*⟩

Rewrite steps at a fixed position

inductive-set *rstep-pos* :: ('*f*, '*v*) *trs* ⇒ *pos* ⇒ ('*f*, '*v*) *term rel* **for** *R* **and** *p*
where
rule [*intro*]:(*l*, *r*) ∈ *R* ⟹ *p* ∈ *poss s* ⟹ *s* |- *p* = *l* · σ ⟹
(*s*, *replace-at s p* (*r* · σ)) ∈ *rstep-pos R p*

lemma *rstep-pos-subst*:
assumes (*s*, *t*) ∈ *rstep-pos R p*
shows (*s* · σ, *t* · σ) ∈ *rstep-pos R p*
⟨*proof*⟩

lemma *rstep-pos-rule*:
assumes $(l, r) \in R$
shows $(l, r) \in$ *rstep-pos* R []
⟨*proof*⟩

lemma *rstep-pos-rstep-r-p-s-conv*:
rstep-pos R $p = \{(s, t) \mid s\ t\ r\ \sigma.\ (s, t) \in$ *rstep-r-p-s* R r p $\sigma\}$
⟨*proof*⟩

lemma *rstep-rstep-pos-conv*:
rstep $R = \{(s, t) \mid s\ t\ p.\ (s, t) \in$ *rstep-pos* R $p\}$
⟨*proof*⟩

lemma *rstep-pos-supt*:
assumes $(s, t) \in$ *rstep-pos* R p
and q: $q \in$ *poss* u **and** u: u |- $q = s$
shows $(u, (ctxt\text{-}of\text{-}pos\text{-}term\ q\ u)\langle t\rangle) \in$ *rstep-pos* R $(q$ @ $p)$
⟨*proof*⟩

lemma *rrstep-rstep-pos-conv*:
rrstep $R =$ *rstep-pos* R []
⟨*proof*⟩

lemma *rrstep-imp-rstep*:
assumes $(s, t) \in$ *rrstep* R
shows $(s, t) \in$ *rstep* R
⟨*proof*⟩

lemma *not-NF-rstep-imp-subteq-not-NF-rrstep*:
assumes $s \notin$ *NF* (*rstep* R)
shows $\exists\, t \trianglelefteq s.\ t \notin$ *NF* (*rrstep* R)
⟨*proof*⟩

lemma *all-subt-NF-rrstep-iff-all-subt-NF-rstep*:
$(\forall\, s \trianglelefteq t.\ s \in$ *NF* (*rrstep* R)) $\longleftrightarrow$ $(\forall\, s \trianglelefteq t.\ s \in$ *NF* (*rstep* R))
⟨*proof*⟩

lemma *not-in-poss-imp-NF-rstep-pos* [*simp*]:
assumes $p \notin$ *poss* s
shows $s \in$ *NF* (*rstep-pos* R p)
⟨*proof*⟩

lemma *Var-rstep-imp-rstep-pos-Empty*:
assumes (*Var* x, $t) \in$ *rstep* R
shows (*Var* x, $t) \in$ *rstep-pos* R []
⟨*proof*⟩

lemma *rstep-args-NF-imp-rrstep*:

assumes $(s,\ t) \in rstep\ R$
and $\forall\, u \lhd s.\ u \in NF\ (rstep\ R)$
shows $(s,\ t) \in rrstep\ R$
⟨*proof*⟩

lemma *rstep-pos-imp-rstep-pos-Empty*:
assumes $(s,\ t) \in rstep\text{-}pos\ R\ p$
shows $(s\ |\text{-}\ p,\ t\ |\text{-}\ p) \in rstep\text{-}pos\ R\ []$
⟨*proof*⟩

lemma *rstep-pos-arg*:
assumes $(s,\ t) \in rstep\text{-}pos\ R\ p$
and $i < length\ ss$ **and** $ss\ !\ i = s$
shows $(Fun\ f\ ss,\ (ctxt\text{-}of\text{-}pos\text{-}term\ [i]\ (Fun\ f\ ss))\langle t\rangle) \in rstep\text{-}pos\ R\ (i\ \#\ p)$
⟨*proof*⟩

lemma *rstep-imp-max-pos*:
assumes $(s,\ t) \in rstep\ R$
shows $\exists\, u.\ \exists\, p{\in}poss\ s.\ (s,\ u) \in rstep\text{-}pos\ R\ p \wedge (\forall\, v \lhd s\ |\text{-}\ p.\ v \in NF\ (rstep\ R))$
⟨*proof*⟩

4.6 Normal Forms

abbreviation *NF-trs* :: $('f,\ 'v)\ trs \Rightarrow ('f,\ 'v)\ terms$ **where**
$NF\text{-}trs\ R \equiv NF\ (rstep\ R)$

lemma *NF-trs-mono*: $r \subseteq s \Longrightarrow NF\text{-}trs\ s \subseteq NF\text{-}trs\ r$
⟨*proof*⟩

lemma *NF-trs-union*: $NF\text{-}trs\ (R \cup S) = NF\text{-}trs\ R \cap NF\text{-}trs\ S$
⟨*proof*⟩

abbreviation *NF-terms* :: $('f,\ 'v)\ terms \Rightarrow ('f,\ 'v)\ terms$ **where**
$NF\text{-}terms\ Q \equiv NF\ (rstep\ (Id\text{-}on\ Q))$

lemma *NF-terms-anti-mono*:
$Q \subseteq Q' \Longrightarrow NF\text{-}terms\ Q' \subseteq NF\text{-}terms\ Q$
⟨*proof*⟩

lemma *lhs-var-not-NF*:
assumes $l \in T$ **and** $is\text{-}Var\ l$ **shows** $t \notin NF\text{-}terms\ T$
⟨*proof*⟩

lemma *not-NF-termsE*[*elim*]:
assumes $s \notin NF\text{-}terms\ Q$
obtains $l\ C\ \sigma$ **where** $l \in Q$ **and** $s = C\langle l \cdot \sigma\rangle$
⟨*proof*⟩

lemma *notin-NF-E* [*elim*]:

fixes $R :: ('f, 'v)\ trs$
assumes $t \notin NF\text{-}trs\ R$
obtains $C\ l$ **and** $\sigma :: ('f, 'v)\ subst$ **where** $l \in lhss\ R$ **and** $t = C\langle l \cdot \sigma \rangle$
⟨*proof*⟩

lemma *NF-ctxt-subst*: $NF\text{-}terms\ Q = \{t.\ \neg\ (\exists\ C\ q\ \sigma.\ t = C\langle q \cdot \sigma \rangle \wedge q \in Q)\}$ (**is** $- = ?R$)
⟨*proof*⟩

lemma *some-NF-imp-no-Var*:
assumes $t \in NF\text{-}terms\ Q$
shows $Var\ x \notin Q$
⟨*proof*⟩

lemma *NF-map-vars-term-inj*:
assumes *inj*: $\bigwedge x.\ n\ (m\ x) = x$ **and** *NF*: $t \in NF\text{-}terms\ Q$
shows $(map\text{-}vars\text{-}term\ m\ t) \in NF\text{-}terms\ (map\text{-}vars\text{-}term\ m\ `\ Q)$
⟨*proof*⟩

lemma *notin-NF-terms*: $t \in Q \Longrightarrow t \notin NF\text{-}terms\ Q$
⟨*proof*⟩

lemma *NF-termsI* [*intro*]:
assumes *NF*: $\bigwedge C\ l\ \sigma.\ t = C\ \langle l \cdot \sigma \rangle \Longrightarrow l \in Q \Longrightarrow False$
shows $t \in NF\text{-}terms\ Q$
⟨*proof*⟩

lemma *NF-args-imp-NF*:
assumes *ss*: $\bigwedge s.\ s \in set\ ss \Longrightarrow s \in NF\text{-}terms\ Q$
and *someNF*: $t \in NF\text{-}terms\ Q$
and *root*: $Some\ (f,length\ ss) \notin root\ `\ Q$
shows $(Fun\ f\ ss) \in NF\text{-}terms\ Q$
⟨*proof*⟩

lemma *NF-Var-is-Fun*:
assumes *Q*: $Ball\ Q\ is\text{-}Fun$
shows $Var\ x \in NF\text{-}terms\ Q$
⟨*proof*⟩

lemma *NF-terms-lhss* [*simp*]: $NF\text{-}terms\ (lhss\ R) = NF\ (rstep\ R)$
⟨*proof*⟩

4.7 Relative Rewrite Steps

abbreviation $relstep\ R\ E \equiv relto\ (rstep\ R)\ (rstep\ E)$

lemma *args-SN-on-relstep-nrrstep-imp-args-SN-on*:
assumes *SN*: $\bigwedge u.\ s \rhd u \Longrightarrow SN\text{-}on\ (relstep\ R\ E)\ \{u\}$
and *st*: $(s,t) \in nrrstep\ (R \cup E)$

and *supt*: $t \rhd u$
shows *SN-on* (*relstep R E*) $\{u\}$
⟨*proof*⟩

lemma *Tinf-nrrstep*:
assumes *tinf*: $s \in$ *Tinf* (*relstep R E*) **and** *st*: $(s,t) \in$ *nrrstep* $(R \cup E)$
and *t*: $\neg$ *SN-on* (*relstep R E*) $\{t\}$
shows $t \in$ *Tinf* (*relstep R E*)
⟨*proof*⟩

lemma *subterm-preserves-SN-on-relstep*:
SN-on (*relstep R E*) $\{s\} \Longrightarrow s \trianglerighteq t \Longrightarrow$ *SN-on* (*relstep R E*) $\{t\}$
⟨*proof*⟩

inductive-set *rstep-rule* :: $('f, 'v)$ *rule* $\Rightarrow$ $('f, 'v)$ *term rel* **for** ϱ
where
rule: $s = C\langle fst\ \varrho \cdot \sigma\rangle \Longrightarrow t = C\langle snd\ \varrho \cdot \sigma\rangle \Longrightarrow (s, t) \in$ *rstep-rule* ϱ

lemma *rstep-ruleI* [*intro*]:
$s = C\langle l \cdot \sigma\rangle \Longrightarrow t = C\langle r \cdot \sigma\rangle \Longrightarrow (s, t) \in$ *rstep-rule* (l, r)
⟨*proof*⟩

lemma *rstep-rule-ctxt*:
$(s, t) \in$ *rstep-rule* $\varrho \Longrightarrow (C\langle s\rangle, C\langle t\rangle) \in$ *rstep-rule* ϱ
⟨*proof*⟩

lemma *rstep-rule-subst*:
assumes $(s, t) \in$ *rstep-rule* ϱ
shows $(s \cdot \sigma, t \cdot \sigma) \in$ *rstep-rule* ϱ
⟨*proof*⟩

lemma *rstep-rule-imp-rstep*:
$\varrho \in R \Longrightarrow (s, t) \in$ *rstep-rule* $\varrho \Longrightarrow (s, t) \in$ *rstep R*
⟨*proof*⟩

lemma *rstep-imp-rstep-rule*:
assumes $(s, t) \in$ *rstep R*
obtains *l r* **where** $(l, r) \in R$ **and** $(s, t) \in$ *rstep-rule* (l, r)
⟨*proof*⟩

lemma *term-subst-rstep*:
assumes $\bigwedge x.\ x \in$ *vars-term* $t \Longrightarrow (\sigma\ x, \tau\ x) \in$ *rstep R*
shows $(t \cdot \sigma, t \cdot \tau) \in$ *(rstep R)**
⟨*proof*⟩

lemma *term-subst-rsteps*:
assumes $\bigwedge x.\ x \in$ *vars-term* $t \Longrightarrow (\sigma\ x, \tau\ x) \in$ *(rstep R)**
shows $(t \cdot \sigma, t \cdot \tau) \in$ *(rstep R)**
⟨*proof*⟩

lemma *term-subst-rsteps-join*:
assumes $\bigwedge y.\ y \in \mathit{vars\text{-}term}\ u \Longrightarrow (\sigma_1\ y,\ \sigma_2\ y) \in (\mathit{rstep}\ R)^{\downarrow}$
shows $(u \cdot \sigma_1,\ u \cdot \sigma_2) \in (\mathit{rstep}\ R)^{\downarrow}$
⟨*proof*⟩

lemma *funas-trs-converse* [*simp*]: *funas-trs* (R^{-1}) = *funas-trs* R
⟨*proof*⟩

lemma *rstep-rev*: **assumes** $(s,\ t) \in \mathit{rstep\text{-}pos}\ \{(l,r)\}\ p$ **shows** $((t,\ s) \in \mathit{rstep\text{-}pos}\ \{(r,l)\}\ p)$
⟨*proof*⟩

lemma *conversion-ctxt-closed*: $(s,\ t) \in (\mathit{rstep}\ R)^{\leftrightarrow *} \Longrightarrow (C\langle s\rangle,\ C\langle t\rangle) \in (\mathit{rstep}\ R)^{\leftrightarrow *}$
⟨*proof*⟩

lemma *conversion-subst-closed*:
$(s,\ t) \in (\mathit{rstep}\ R)^{\leftrightarrow *} \Longrightarrow (s \cdot \sigma,\ \ t \cdot \sigma) \in (\mathit{rstep}\ R)^{\leftrightarrow *}$
⟨*proof*⟩

lemma *rstep-simulate-conv*:
assumes $\bigwedge\ l\ r.\ (l,\ r) \in S \Longrightarrow (l,\ r) \in (\mathit{rstep}\ R)^{\leftrightarrow *}$
shows $(\mathit{rstep}\ S) \subseteq (\mathit{rstep}\ R)^{\leftrightarrow *}$
⟨*proof*⟩

lemma *symcl-simulate-conv*:
assumes $\bigwedge\ l\ r.\ (l,\ r) \in S \Longrightarrow (l,\ r) \in (\mathit{rstep}\ R)^{\leftrightarrow *}$
shows $(\mathit{rstep}\ S)^{\leftrightarrow} \subseteq (\mathit{rstep}\ R)^{\leftrightarrow *}$
⟨*proof*⟩

lemma *conv-union-simulate*:
assumes $\bigwedge\ l\ r.\ (l,\ r) \in S \Longrightarrow (l,\ r) \in (\mathit{rstep}\ R)^{\leftrightarrow *}$
shows $(\mathit{rstep}\ (R \cup S))^{\leftrightarrow *} = (\mathit{rstep}\ R)^{\leftrightarrow *}$
⟨*proof*⟩

definition *suptrel* R = $(\mathit{relto}\ \{\triangleright\}\ (\mathit{rstep}\ R))^{+}$
end

5 Critical Pairs

theory *Critical-Pairs*
imports
Trs
First-Order-Terms.Unification
begin

We also consider overlaps between the same rule at top level, in this way we are not restricted to *wf-trs*.

context
fixes *ren* :: *'v* :: *infinite renaming2*
begin

definition
critical-Peaks :: (*'f*, *'v*) *trs* ⇒ (*'f*, *'v*) *trs* ⇒ (((*'f*, *'v*)*term* × (*'f*,*'v*)*term* × (*'f*,*'v*)*term*)) *set*
where
critical-Peaks P R = { ((*C* ·$_c$ *σ*)⟨*r'* · *τ*⟩, *l* · *σ*, *r* · *σ*) | *l r l' r' l'' C σ τ*.
(*l*, *r*) ∈ *P* ∧ (*l'*, *r'*) ∈ *R* ∧ *l* = *C*⟨*l''*⟩ ∧ *is-Fun l''* ∧
mgu-vd ren l'' l' = *Some* (*σ*, *τ*) }

definition
critical-pairs :: (*'f*, *'v*) *trs* ⇒ (*'f*, *'v*) *trs* ⇒ (*bool* × (*'f*, *'v*) *rule*) *set*
where
critical-pairs P R = { (*C* = □, (*C* ·$_c$ *σ*)⟨*r'* · *τ*⟩, *r* · *σ*) | *l r l' r' l'' C σ τ*.
(*l*, *r*) ∈ *P* ∧ (*l'*, *r'*) ∈ *R* ∧ *l* = *C*⟨*l''*⟩ ∧ *is-Fun l''* ∧
mgu-vd ren l'' l' = *Some* (*σ*, *τ*) }

lemma *critical-pairsI*:
assumes (*l*, *r*) ∈ *P* **and** (*l'*, *r'*) ∈ *R* **and** *l* = *C*⟨*l''*⟩
and *is-Fun l''* **and** *mgu-vd ren l'' l'* = *Some* (*σ*, *τ*) **and** *t* = *r* · *σ*
and *s* = (*C* ·$_c$ *σ*)⟨*r'* · *τ*⟩ **and** *b* = (*C* = □)
shows (*b*, *s*, *t*) ∈ *critical-pairs P R*
⟨*proof*⟩

lemma *critical-pairs-mono*:
assumes *S*$_1$ ⊆ *R*$_1$ **and** *S*$_2$ ⊆ *R*$_2$ **shows** *critical-pairs S*$_1$ *S*$_2$ ⊆ *critical-pairs R*$_1$ *R*$_2$
⟨*proof*⟩

lemma *critical-Peaks-main*:
fixes *P R* :: (*'f*, *'v*) *trs*
assumes *tu*: (*t*, *u*) ∈ *rrstep P* **and** *ts*: (*t*, *s*) ∈ *rstep R*
shows (*s*, *u*) ∈ (*rstep R*)^* *O rrstep P O* ((*rstep R*)^*)^−1 ∨
(∃ *C l m r σ*. *s* = *C*⟨*l* · *σ*⟩ ∧ *t* = *C*⟨*m* · *σ*⟩ ∧ *u* = *C*⟨*r* · *σ*⟩ ∧
(*l*, *m*, *r*) ∈ *critical-Peaks P R*)
⟨*proof*⟩

lemma *critical-Peaks-main-rrstep*:
fixes *R* :: (*'f*, *'v*) *trs*
assumes *tu*: (*t*, *u*) ∈ *rrstep R* **and** *ts*: (*t*, *s*) ∈ *rstep R*
shows (*s*, *u*) ∈ *join* (*rstep R*) ∨
(∃ *C l m r σ*. *s* = *C*⟨*l* · *σ*⟩ ∧ *t* = *C*⟨*m* · *σ*⟩ ∧ *u* = *C*⟨*r* · *σ*⟩ ∧
(*l*, *m*, *r*) ∈ *critical-Peaks R R*)
⟨*proof*⟩

lemma *parallel-rstep*:
fixes *p1* :: *pos*

assumes *p12*: $p1 \perp p2$
and *p1*: $p1 \in poss\ t$
and *p2*: $p2 \in poss\ t$
and *step2*: t |- $p2 = l2 \cdot \sigma 2\ (l2,r2) \in R$
shows $(replace\text{-}at\ t\ p1\ v,\ replace\text{-}at\ (replace\text{-}at\ t\ p1\ v)\ p2\ (r2 \cdot \sigma 2)) \in rstep\ R$
(**is** (*?one*,*?two*) ∈ -)
⟨*proof*⟩

lemma *critical-Peaks-main-rstep*:
fixes $R :: ('f, 'v)\ trs$
assumes *tu*: $(t, u) \in rstep\ R$ **and** *ts*: $(t, s) \in rstep\ R$
shows $(s, u) \in join\ (rstep\ R) \vee$
$(\exists\ C\ l\ m\ r\ \sigma.\ s = C\langle l \cdot \sigma\rangle \wedge t = C\langle m \cdot \sigma\rangle \wedge u = C\langle r \cdot \sigma\rangle \wedge$
$((l, m, r) \in critical\text{-}Peaks\ R\ R \vee (r, m, l) \in critical\text{-}Peaks\ R\ R))$
⟨*proof*⟩

lemma *critical-Peak-steps*:
fixes $R :: ('f, 'v)\ trs$ **and** S
assumes *cp*: $(l, m, r) \in critical\text{-}Peaks\ R\ S$
shows $(m, l) \in rstep\ S\ (m,r) \in rstep\ R\ (m,r) \in rrstep\ R$
⟨*proof*⟩

lemma *critical-Peak-to-pair*: **assumes** $(l, m, r) \in critical\text{-}Peaks\ R\ R$
shows $\exists\ b.\ (b, l, r) \in critical\text{-}pairs\ R\ R$
⟨*proof*⟩

lemma *critical-pairs-main*:
fixes $R :: ('f, 'v)\ trs$
assumes *st1*: $(s, t1) \in rstep\ R$ **and** *st2*: $(s, t2) \in rstep\ R$
shows $(t1, t2) \in join\ (rstep\ R) \vee$
$(\exists\ C\ b\ l\ r\ \sigma.\ t1 = C\langle l \cdot \sigma\rangle \wedge t2 = C\langle r \cdot \sigma\rangle \wedge$
$((b, l, r) \in critical\text{-}pairs\ R\ R \vee (b, r, l) \in critical\text{-}pairs\ R\ R))$
⟨*proof*⟩

lemma *critical-pairs*:
fixes $R :: ('f, 'v)\ trs$
assumes *cp*: $\bigwedge l\ r\ b.\ (b, l, r) \in critical\text{-}pairs\ R\ R \Longrightarrow l \neq r \Longrightarrow$
$\exists\ l'\ r'\ s.\ instance\text{-}rule\ (l, r)\ (l', r') \wedge (l', s) \in (rstep\ R)^* \wedge (r', s) \in (rstep\ R)^*$
shows $WCR\ (rstep\ R)$
⟨*proof*⟩

lemma *critical-pairs-fork*:
fixes $R :: ('f, 'v)\ trs$ **and** S
assumes *cp*: $(b, l, r) \in critical\text{-}pairs\ R\ S$
shows $(r, l) \in (rstep\ R)^{-1}\ O\ rstep\ S$
⟨*proof*⟩

lemma *critical-pairs-fork'*: **assumes** $(b,l,r) \in critical\text{-}pairs\ R\ S$

shows $(l,r) \in (rstep\ S)^{\frown -1}\ O\ rstep\ R$
⟨*proof*⟩

lemma *critical-pairs-complete*:
fixes $R :: ('f, 'v)\ trs$
assumes *cp*: $(b, l, r) \in critical\text{-}pairs\ R\ R$
and *no-join*: $(l, r) \notin (rstep\ R)^{\downarrow}$
shows $\neg\ WCR\ (rstep\ R)$
⟨*proof*⟩

lemma *critical-pair-lemma*:
fixes $R :: ('f, 'v)\ trs$
shows $WCR\ (rstep\ R) \longleftrightarrow$
$(\forall\ (b, s, t) \in critical\text{-}pairs\ R\ R.\ (s, t) \in (rstep\ R)^{\downarrow})$
(**is** $?l = ?r$)
⟨*proof*⟩

lemma *critical-pairs-exI*:
fixes $\sigma :: ('f, 'v)\ subst$
assumes P: $(l, r) \in P$ **and** R: $(l', r') \in R$ **and** l: $l = C\langle l''\rangle$
and l'': *is-Fun* l'' **and** *unif*: $l'' \cdot \sigma = l' \cdot \tau$
and b: $b = (C = \Box)$
shows $\exists\ s\ t.\ (b, s, t) \in critical\text{-}pairs\ P\ R$
⟨*proof*⟩

end
end

6 Parallel Rewriting

6.1 Multihole Contexts

theory *Multihole-Context*
imports
Trs
FOR-Preliminaries
SubList
begin

unbundle *lattice-syntax*

6.1.1 Partitioning lists into chunks of given length

fun *partition-by*
where
partition-by xs [] = [] |
partition-by xs (*y*#*ys*) = *take y xs* # *partition-by* (*drop y xs*) *ys*

lemma *partition-by-map0-append* [*simp*]:
partition-by xs (*map* (λ*x*. *0*) *ys* @ *zs*) = *replicate* (*length ys*) [] @ *partition-by xs zs*
⟨*proof*⟩

lemma *concat-partition-by* [*simp*]:
sum-list ys = *length xs* ⟹ *concat* (*partition-by xs ys*) = *xs*
⟨*proof*⟩

definition *partition-by-idx* **where**
partition-by-idx l ys i j = *partition-by* [*0*..<*l*] *ys* ! *i* ! *j*

lemma *partition-by-nth-nth-old*:
assumes *i* < *length* (*partition-by xs ys*)
and *j* < *length* (*partition-by xs ys* ! *i*)
and *sum-list ys* = *length xs*
shows *partition-by xs ys* ! *i* ! *j* = *xs* ! (*sum-list* (*map length* (*take i* (*partition-by xs ys*))) + *j*)
⟨*proof*⟩

lemma *map-map-partition-by*:
map (*map f*) (*partition-by xs ys*) = *partition-by* (*map f xs*) *ys*
⟨*proof*⟩

lemma *length-partition-by* [*simp*]:
length (*partition-by xs ys*) = *length ys*
⟨*proof*⟩

lemma *partition-by-Nil* [*simp*]:
partition-by [] *ys* = *replicate* (*length ys*) []
⟨*proof*⟩

lemma *partition-by-concat-id* [*simp*]:
assumes *length xss* = *length ys*
and ⋀*i*. *i* < *length ys* ⟹ *length* (*xss* ! *i*) = *ys* ! *i*
shows *partition-by* (*concat xss*) *ys* = *xss*
⟨*proof*⟩

lemma *partition-by-nth*:
i < *length ys* ⟹ *partition-by xs ys* ! *i* = *take* (*ys* ! *i*) (*drop* (*sum-list* (*take i ys*)) *xs*)
⟨*proof*⟩

lemma *partition-by-nth-less*:
assumes *k* < *i* **and** *i* < *length zs*
and *length xs* = *sum-list* (*take i zs*) + *j*
shows *partition-by* (*xs* @ *y* # *ys*) *zs* ! *k* = *take* (*zs* ! *k*) (*drop* (*sum-list* (*take k zs*)) *xs*)

⟨*proof*⟩

lemma *partition-by-nth-greater*:
assumes $i < k$ **and** $k <$ *length zs* **and** $j <$ *zs* ! i
and *length xs* = *sum-list* (*take i zs*) + j
shows *partition-by* (*xs* @ y # *ys*) *zs* ! k =
take (*zs* ! k) (*drop* (*sum-list* (*take k zs*) − *1*) (*xs* @ *ys*))
⟨*proof*⟩

lemma *length-partition-by-nth*:
sum-list ys = *length xs* ⟹ $i <$ *length ys* ⟹ *length* (*partition-by xs ys* ! i) = *ys* ! i
⟨*proof*⟩

lemma *partition-by-nth-nth-elem*:
assumes *sum-list ys* = *length xs* $i <$ *length ys* $j <$ *ys* ! i
shows *partition-by xs ys* ! i ! $j \in$ *set xs*
⟨*proof*⟩

lemma *partition-by-nth-nth*:
assumes *sum-list ys* = *length xs* $i <$ *length ys* $j <$ *ys* ! i
shows *partition-by xs ys* ! i ! j = *xs* ! *partition-by-idx* (*length xs*) *ys i j*
partition-by-idx (*length xs*) *ys i j* < *length xs*
⟨*proof*⟩

lemma *map-length-partition-by* [*simp*]:
sum-list ys = *length xs* ⟹ *map length* (*partition-by xs ys*) = *ys*
⟨*proof*⟩

lemma *map-partition-by-nth* [*simp*]:
$i <$ *length ys* ⟹ *map f* (*partition-by xs ys* ! i) = *partition-by* (*map f xs*) *ys* ! i
⟨*proof*⟩

lemma *sum-list-partition-by* [*simp*]:
sum-list ys = *length xs* ⟹
sum-list (*map* (λ*x*. *sum-list* (*map f x*)) (*partition-by xs ys*)) = *sum-list* (*map f xs*)
⟨*proof*⟩

lemma *partition-by-map-conv*:
partition-by xs ys = *map* (λ*i*. *take* (*ys* ! i) (*drop* (*sum-list* (*take i ys*)) *xs*)) [*0* ..< *length ys*]
⟨*proof*⟩

lemma *UN-set-partition-by-map*:
sum-list ys = *length xs* ⟹ ($\bigcup x \in$ *set* (*partition-by* (*map f xs*) *ys*). $\bigcup$ (*set x*)) = $\bigcup$(*set* (*map f xs*))
⟨*proof*⟩

lemma *UN-set-partition-by*:
sum-list ys = length xs ⟹ (⋃ zs ∈ set (partition-by xs ys). ⋃ x ∈ set zs. f x) = (⋃ x ∈ set xs. f x)
⟨*proof*⟩

lemma *Ball-atLeast0LessThan-partition-by-conv*:
(∀ i∈{0..<length ys}. ∀ x∈set (partition-by xs ys ! i). P x) =
(∀ x ∈ ⋃(set (map set (partition-by xs ys))). P x)
⟨*proof*⟩

lemma *Ball-set-partition-by*:
sum-list ys = length xs ⟹
(∀ x ∈ set (partition-by xs ys). ∀ y ∈ set x. P y) = (∀ x ∈ set xs. P x)
⟨*proof*⟩

lemma *partition-by-append2*:
partition-by xs (ys @ zs) = partition-by (take (sum-list ys) xs) ys @ partition-by (drop (sum-list ys) xs) zs
⟨*proof*⟩

lemma *partition-by-concat2*:
partition-by xs (concat ys) =
concat (map (λi . partition-by (partition-by xs (map sum-list ys) ! i) (ys ! i)) [0..<length ys])
⟨*proof*⟩

lemma *partition-by-partition-by*:
length xs = sum-list (map sum-list ys) ⟹
partition-by (partition-by xs (concat ys)) (map length ys) =
map (λi. partition-by (partition-by xs (map sum-list ys) ! i) (ys ! i)) [0..<length ys]
⟨*proof*⟩

datatype *('f, vars-mctxt : 'v) mctxt = MVar 'v | MHole | MFun 'f ('f, 'v) mctxt list*

6.1.2 Conversions from and to multihole contexts

primrec *mctxt-of-term* :: *('f, 'v) term ⇒ ('f, 'v) mctxt*
where
mctxt-of-term (Var x) = MVar x |
mctxt-of-term (Fun f ts) = MFun f (map mctxt-of-term ts)

primrec *term-of-mctxt* :: *('f, 'v) mctxt ⇒ ('f, 'v) term*
where
term-of-mctxt (MVar x) = Var x |
term-of-mctxt (MFun f Cs) = Fun f (map term-of-mctxt Cs)

lemma *term-of-mctxt-mctxt-of-term-id* [*simp*]:

term-of-mctxt (*mctxt-of-term t*) = *t*
⟨*proof*⟩

fun *num-holes* :: ('*f*, '*v*) *mctxt* ⇒ *nat*
where
num-holes (*MVar* -) = *0* |
num-holes MHole = *1* |
num-holes (*MFun* - *ctxts*) = *sum-list* (*map num-holes ctxts*)

lemma *num-holes-o-mctxt-of-term* [*simp*]:
num-holes ∘ *mctxt-of-term* = (λ*x*. *0*)
⟨*proof*⟩

lemma *mctxt-of-term-term-of-mctxt-id* [*simp*]:
num-holes C = *0* ⟹ *mctxt-of-term* (*term-of-mctxt C*) = *C*
⟨*proof*⟩

lemma *vars-mctxt-of-term*[*simp*]: *vars-mctxt* (*mctxt-of-term t*) = *vars-term t*
⟨*proof*⟩

lemma *num-holes-mctxt-of-term* [*simp*]:
num-holes (*mctxt-of-term t*) = *0*
⟨*proof*⟩

fun *funas-mctxt* :: ('*f*, '*v*) *mctxt* ⇒ '*f sig*
where
funas-mctxt (*MFun f Cs*) = {(*f*, *length Cs*)} ∪ ⋃(*funas-mctxt* ‘ *set Cs*) |
funas-mctxt - = {}

fun *funas-mctxt-list* :: ('*f*, '*v*) *mctxt* ⇒ ('*f* × *nat*) *list*
where
funas-mctxt-list (*MFun f Cs*) = (*f*, *length Cs*) # *concat* (*map funas-mctxt-list Cs*) |
funas-mctxt-list - = []

lemma *funas-mctxt-list* [*simp*]:
set (*funas-mctxt-list C*) = *funas-mctxt C*
⟨*proof*⟩

fun *split-term* :: (('*f*, '*v*) *term* ⇒ *bool*) ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *mctxt* × ('*f*, '*v*) *term list*
where
split-term P (*Var x*) = (*if P* (*Var x*) *then* (*MHole*, [*Var x*]) *else* (*MVar x*, [])) |
split-term P (*Fun f ts*) =
(*if P* (*Fun f ts*) *then* (*MHole*, [*Fun f ts*])
else let us = *map* (*split-term P*) *ts in* (*MFun f* (*map fst us*), *concat* (*map snd us*)))

fun *cap-till* :: $(('f, 'v)\ term \Rightarrow bool) \Rightarrow ('f, 'v)\ term \Rightarrow ('f, 'v)\ mctxt$
where
cap-till P (Var x) = (if P (Var x) then MHole else MVar x) |
cap-till P (Fun f ts) = (if P (Fun f ts) then MHole else MFun f (map (cap-till P) ts))

fun *uncap-till* :: $(('f, 'v)\ term \Rightarrow bool) \Rightarrow ('f, 'v)\ term \Rightarrow ('f, 'v)\ term\ list$
where
uncap-till P (Var x) = (if P (Var x) then [Var x] else []) |
uncap-till P (Fun f ts) = (if P (Fun f ts) then [Fun f ts] else concat (map (uncap-till P) ts))

lemma *split-term* [*simp*]:
split-term P t = (cap-till P t, uncap-till P t)
⟨*proof*⟩

definition *if-Fun-in-set F* = $(\lambda t.\ is\text{-}Var\ t \vee the\ (root\ t) \in F)$

lemma *if-Fun-in-set-simps* [*simp*]:
if-Fun-in-set F (Var x)
if-Fun-in-set F (Fun f ts) $\longleftrightarrow$ *(f, length ts)* $\in$ *F*
is-Var t $\Longrightarrow$ *if-Fun-in-set F t*
is-Fun t $\Longrightarrow$ *if-Fun-in-set F t* $\longleftrightarrow$ *the (root t)* $\in$ *F*
⟨*proof*⟩

lemma *if-Fun-in-set-mono*:
F $\subseteq$ *G* $\Longrightarrow$ *if-Fun-in-set F t* $\Longrightarrow$ *if-Fun-in-set G t*
⟨*proof*⟩

abbreviation *split-term-funas F* $\equiv$ *split-term (if-Fun-in-set F)*
abbreviation *cap-till-funas F* $\equiv$ *cap-till (if-Fun-in-set F)*
abbreviation *uncap-till-funas F* $\equiv$ *uncap-till (if-Fun-in-set F)*

lemma *if-Fun-in-set-uncap-till-funas*:
A $\subseteq$ *B* $\Longrightarrow$ *if-Fun-in-set A t* $\Longrightarrow$ *uncap-till-funas B t* = *[t]*
⟨*proof*⟩

lemma *cap-till-funasD* [*dest*]:
fn $\in$ *funas-mctxt (cap-till-funas F t)* $\Longrightarrow$ *fn* $\in$ *F* $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *cap-till-funas*:
$\forall$ *fn* $\in$ *funas-mctxt (cap-till-funas F t). fn* $\notin$ *F*
⟨*proof*⟩

lemma *uncap-till*:
$\forall$ *s* $\in$ *set (uncap-till P t). P s*
⟨*proof*⟩

lemma *uncap-till-singleton*:
assumes $s \in set\ (uncap\text{-}till\ P\ t)$
shows $uncap\text{-}till\ P\ s = [s]$
⟨*proof*⟩

lemma *uncap-till-idemp* [*simp*]:
$map\ (uncap\text{-}till\ P)\ (uncap\text{-}till\ P\ t) = map\ (\lambda s.\ [s])\ (uncap\text{-}till\ P\ t)$
⟨*proof*⟩

lemma *uncap-till-Fun* [*simp*]:
$P\ (Fun\ f\ ts) \Longrightarrow uncap\text{-}till\ P\ (Fun\ f\ ts) = [Fun\ f\ ts]$
⟨*proof*⟩

abbreviation *partition-holes xs Cs* ≡ *partition-by xs* (*map num-holes Cs*)
abbreviation *partition-holes-idx l Cs* ≡ *partition-by-idx l* (*map num-holes Cs*)

fun *fill-holes* :: ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *term list* ⇒ ('*f*, '*v*) *term*
where
fill-holes (*MVar x*) - = *Var x* |
fill-holes MHole [*t*] = *t* |
fill-holes (*MFun f cs*) *ts* = *Fun f* (*map* (λ *i*. *fill-holes* (*cs* ! *i*)
(*partition-holes ts cs* ! *i*)) [*0* ..< *length cs*])

The following induction scheme provides the *MFun* case with the list argument split according to the argument contexts. This feature is quite delicate: its benefit can be destroyed by premature simplification using the *sum-list ?ys = length ?xs* ⟹ *concat* (*partition-by ?xs ?ys*) = *?xs* simplification rule.

lemma *fill-holes-induct2*[*consumes 2*, *case-names MHole MVar MFun*]:
fixes *P* :: ('*f*,'*v*) *mctxt* ⇒ '*a list* ⇒ '*b list* ⇒ *bool*
assumes *len1*: *num-holes C* = *length xs* **and** *len2*: *num-holes C* = *length ys*
and *Hole*: ⋀*x y*. *P MHole* [*x*] [*y*]
and *Var*: ⋀*v*. *P* (*MVar v*) [] []
and *Fun*: ⋀*f Cs xs ys*. *sum-list* (*map num-holes Cs*) = *length xs* ⟹
sum-list (*map num-holes Cs*) = *length ys* ⟹
(⋀*i*. *i* < *length Cs* ⟹ *P* (*Cs* ! *i*) (*partition-holes xs Cs* ! *i*) (*partition-holes ys Cs* ! *i*)) ⟹
P (*MFun f Cs*) (*concat* (*partition-holes xs Cs*)) (*concat* (*partition-holes ys Cs*))
shows *P C xs ys*
⟨*proof*⟩

lemma *fill-holes-induct*[*consumes 1*, *case-names MHole MVar MFun*]:
fixes *P* :: ('*f*,'*v*) *mctxt* ⇒ '*a list* ⇒ *bool*
assumes *len*: *num-holes C* = *length xs*
and *Hole*: ⋀*x*. *P MHole* [*x*]
and *Var*: ⋀*v*. *P* (*MVar v*) []
and *Fun*: ⋀*f Cs xs*. *sum-list* (*map num-holes Cs*) = *length xs* ⟹
(⋀*i*. *i* < *length Cs* ⟹ *P* (*Cs* ! *i*) (*partition-holes xs Cs* ! *i*)) ⟹
P (*MFun f Cs*) (*concat* (*partition-holes xs Cs*))

shows *P C xs*
⟨*proof*⟩

lemma *funas-term-fill-holes-iff*: *num-holes C* = *length ts* ⟹
g ∈ *funas-term* (*fill-holes C ts*) ⟷ *g* ∈ *funas-mctxt C* ∨ (∃ *t* ∈ *set ts*. *g* ∈ *funas-term t*)
⟨*proof*⟩

lemma *fill-holes-MHole*:
length ts = *1* ⟹ *ts* ! *0* = *u* ⟹ *fill-holes MHole ts* = *u*
⟨*proof*⟩

lemmas
map-partition-holes-nth [*simp*] =
map-partition-by-nth [*of* - *map num-holes Cs* **for** *Cs*, *unfolded length-map*] **and**
length-partition-holes [*simp*] =
length-partition-by [*of* - *map num-holes Cs* **for** *Cs*, *unfolded length-map*]

lemma *length-partition-holes-nth* [*simp*]:
assumes *sum-list* (*map num-holes cs*) = *length ts*
and *i* < *length cs*
shows *length* (*partition-holes ts cs* ! *i*) = *num-holes* (*cs* ! *i*)
⟨*proof*⟩

lemma *concat-partition-holes-upt*:
assumes *i* ≤ *length cs*
shows *concat* [*partition-holes ts cs* ! *j*. *j* ← [*0* ..< *i*]] =
take (*sum-list* [*num-holes* (*cs* ! *j*). *j* ← [*0* ..< *i*]]) *ts*
⟨*proof*⟩

lemma *partition-holes-step*:
partition-holes ts (*C* # *Cs*) = *take* (*num-holes C*) *ts* # *partition-holes* (*drop* (*num-holes C*) *ts*) *Cs*
⟨*proof*⟩

lemma *partition-holes-map-ctxt*:
assumes *length cs* = *length ds*
and ⋀ *i*. *i* < *length cs* ⟹ *num-holes* (*cs* ! *i*) = *num-holes* (*ds* ! *i*)
shows *partition-holes ts cs* = *partition-holes ts ds*
⟨*proof*⟩

lemma *partition-holes-concat-id*:
assumes *length sss* = *length cs*

and $\bigwedge i.\ i < \mathit{length}\ cs \Longrightarrow \mathit{num\text{-}holes}\ (cs\ !\ i) = \mathit{length}\ (sss\ !\ i)$
shows *partition-holes* (*concat sss*) *cs* = *sss*
⟨*proof*⟩

lemma *partition-holes-fill-holes-conv*:
fill-holes (*MFun f cs*) *ts* =
Fun f [*fill-holes* (*cs* ! *i*) (*partition-holes ts cs* ! *i*). *i* ← [*0* ..< *length cs*]]
⟨*proof*⟩

lemma *fill-holes-arbitrary*:
assumes *lCs*: *length Cs* = *length ts*
and *lss*: *length ss* = *length ts*
and *rec*: $\bigwedge i.\ i < \mathit{length}\ ts \Longrightarrow \mathit{num\text{-}holes}\ (Cs\ !\ i) = \mathit{length}\ (ss\ !\ i) \wedge f\ (Cs\ !\ i)\ (ss\ !\ i) = ts\ !\ i$
shows *map* ($\lambda i.\ f\ (Cs\ !\ i)$ (*partition-holes* (*concat ss*) *Cs* ! *i*)) [*0* ..< *length Cs*] = *ts*
⟨*proof*⟩

lemma *fill-holes-MFun*:
assumes *lCs*: *length Cs* = *length ts*
and *lss*: *length ss* = *length ts*
and *rec*: $\bigwedge i.\ i < \mathit{length}\ ts \Longrightarrow \mathit{num\text{-}holes}\ (Cs\ !\ i) = \mathit{length}\ (ss\ !\ i) \wedge \mathit{fill\text{-}holes}\ (Cs\ !\ i)\ (ss\ !\ i) = ts\ !\ i$
shows *fill-holes* (*MFun f Cs*) (*concat ss*) = *Fun f ts*
⟨*proof*⟩

inductive
eq-fill ::
('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *mctxt* × ('*f*, '*v*) *term list* ⇒ *bool* ((-/ =$_f$ -) [*51*, *51*] *50*)
where
eqfI [*intro*]: *t* = *fill-holes D ss* ⟹ *num-holes D* = *length ss* ⟹ *t* =$_f$ (*D*, *ss*)

lemma *fill-holes-inj*:
assumes *num-holes C* = *length ss*
and *num-holes C* = *length ts*
and *fill-holes C ss* = *fill-holes C ts*
shows *ss* = *ts*
⟨*proof*⟩

lemma *eqf-refl* [*intro*]:
num-holes C = *length ts* ⟹ *fill-holes C ts* =$_f$ (*C*, *ts*)
⟨*proof*⟩

lemma *eqfE*:
assumes *t* =$_f$ (*D*, *ss*) **shows** *t* = *fill-holes D ss num-holes D* = *length ss*
⟨*proof*⟩

lemma *eqf-MFunI*:
assumes *length sss* = *length Cs*

and *length ts* = *length Cs*
and $\bigwedge$ *i*. *i* < *length Cs* $\Longrightarrow$ *ts* ! *i* =$_f$ (*Cs* ! *i*, *sss* ! *i*)
shows *Fun f ts* =$_f$ (*MFun f Cs*, *concat sss*)
⟨*proof*⟩

lemma *eqf-MFunE*:
assumes *s* =$_f$ (*MFun f Cs*,*ss*)
obtains *ts sss* **where** *s* = *Fun f ts length ts* = *length Cs length sss* = *length Cs*
$\bigwedge$ *i*. *i* < *length Cs* $\Longrightarrow$ *ts* ! *i* =$_f$ (*Cs* ! *i*, *sss* ! *i*)
ss = *concat sss*
⟨*proof*⟩

lemma *eqf-MHoleE*:
assumes *s* =$_f$ (*MHole*, *ss*)
shows *ss* = [*s*]
⟨*proof*⟩

fun *mctxt-of-ctxt* :: ('*f*, '*v*) *ctxt* ⇒ ('*f*, '*v*) *mctxt*
where
mctxt-of-ctxt Hole = *MHole* |
mctxt-of-ctxt (*More f ss*$_1$ *C ss*$_2$) =
MFun f (*map mctxt-of-term ss*$_1$ @ *mctxt-of-ctxt C* # *map mctxt-of-term ss*$_2$)

lemma *num-holes-mctxt-of-ctxt* [*simp*]:
num-holes (*mctxt-of-ctxt C*) = *1*
⟨*proof*⟩

lemma *mctxt-of-term*: *t* =$_f$ (*mctxt-of-term t*, [])
⟨*proof*⟩

lemma *mctxt-of-ctxt* [*simp*]:
C⟨*t*⟩ =$_f$ (*mctxt-of-ctxt C*, [*t*])
⟨*proof*⟩

lemma *fill-holes-ctxt-main*′:
assumes *num-holes C* = *Suc* (*length bef* + *length aft*)
shows ∃ *D*. (∀ *s*. *fill-holes C* (*bef* @ *s* # *aft*) = *D* ⟨*s*⟩) ∧ (*C* = *MFun f cs* ⟶ *D* ≠ □)
⟨*proof*⟩

lemma *fill-holes-ctxt-main*:
assumes *num-holes C* = *Suc* (*length bef* + *length aft*)
shows ∃ *D*. ∀ *s*. *fill-holes C* (*bef* @ *s* # *aft*) = *D* ⟨ *s* ⟩
⟨*proof*⟩

lemma *fill-holes-ctxt*:
assumes *nh*: *num-holes C* = *length ss*
and *i*: *i* < *length ss*
obtains *D* **where** $\bigwedge$ *s*. *fill-holes C* (*ss*[*i* := *s*]) = *D* ⟨ *s* ⟩

⟨*proof*⟩

fun *map-vars-mctxt* :: ('*v* ⇒ '*w*) ⇒ ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*w*) *mctxt*
where
map-vars-mctxt vw MHole = *MHole* |
map-vars-mctxt vw (*MVar v*) = (*MVar* (*vw v*)) |
map-vars-mctxt vw (*MFun f Cs*) = *MFun f* (*map* (*map-vars-mctxt vw*) *Cs*)

lemma *map-vars-mctxt-id* [*simp*]:
map-vars-mctxt (λ *x*. *x*) *C* = *C*
⟨*proof*⟩

lemma *num-holes-map-vars-mctxt* [*simp*]:
num-holes (*map-vars-mctxt vw C*) = *num-holes C*
⟨*proof*⟩

lemma *map-vars-term-eq-fill*:
t =$_f$ (*C*,*ss*) ⟹ *map-vars-term vw t* =$_f$ (*map-vars-mctxt vw C*, *map* (*map-vars-term vw*) *ss*)
⟨*proof*⟩

lemma *map-vars-term-fill-holes*:
assumes *nh*: *num-holes C* = *length ss*
shows *map-vars-term vw* (*fill-holes C ss*) =
fill-holes (*map-vars-mctxt vw C*) (*map* (*map-vars-term vw*) *ss*)
⟨*proof*⟩

lemma *split-term-eqf*:
t =$_f$ (*cap-till P t*, *uncap-till P t*)
⟨*proof*⟩

lemma *fill-holes-cap-till-uncap-till-id* [*simp*]:
fill-holes (*cap-till P t*) (*uncap-till P t*) = *t*
⟨*proof*⟩

lemma *num-holes-cap-till* [*simp*]:
num-holes (*cap-till P t*) = *length* (*uncap-till P t*)
⟨*proof*⟩

fun *split-vars* :: ('*f*, '*v*) *term* ⇒ (('*f*, '*v*) *mctxt* × '*v list*)
where
split-vars (*Var x*) = (*MHole*, [*x*]) |
split-vars (*Fun f ts*) = (*MFun f* (*map* (*fst* ∘ *split-vars*) *ts*), *concat* (*map* (*snd* ∘ *split-vars*) *ts*))

lemma *split-vars-num-holes*: *num-holes* (*fst* (*split-vars t*)) = *length* (*snd* (*split-vars t*))
⟨*proof*⟩

lemma *split-vars-vars-term-list*: *snd* (*split-vars t*) = *vars-term-list t*
⟨*proof*⟩

lemma *split-vars-vars-term*: *set* (*snd* (*split-vars t*)) = *vars-term t*
⟨*proof*⟩

lemma *split-vars-eqf-subst-map-vars-term*:
t · *σ* =$_f$ (*map-vars-mctxt vw* (*fst* (*split-vars t*)), *map σ* (*snd* (*split-vars t*)))
⟨*proof*⟩

lemma *split-vars-eqf-subst*: *t* · *σ* =$_f$ (*fst* (*split-vars t*), (*map σ* (*snd* (*split-vars t*))))
⟨*proof*⟩

lemma *split-vars-into-subst-map-vars-term*:
assumes *split*: *split-vars l* = (*C*,*xs*)
and *len*: *length ts* = *length xs*
and *id*: ⋀ *i*. *i* < *length xs* ⟹ *σ* (*xs* ! *i*) = *ts* ! *i*
shows *l* · *σ* =$_f$ (*map-vars-mctxt vw C*,*ts*)
⟨*proof*⟩

lemma *split-vars-into-subst*:
assumes *split*: *split-vars l* = (*C*,*xs*)
and *len*: *length ts* = *length xs*
and *id*: ⋀ *i*. *i* < *length xs* ⟹ *σ* (*xs* ! *i*) = *ts* ! *i*
shows *l* · *σ* =$_f$ (*C*,*ts*)
⟨*proof*⟩

lemma *eqf-funas-term*:
t =$_f$ (*C*,*ss*) ⟹ *funas-term t* = *funas-mctxt C* ∪ ⋃(*funas-term* ‘ *set ss*)
⟨*proof*⟩

lemma *eqf-all-ctxt-closed-step*:
assumes *ctxt*: *all-ctxt-closed F R*
and *ass*: *t* =$_f$ (*D*,*ss*) ⋀ *i*. *i* < *length ts* ⟹ (*ss* ! *i*, *ts* ! *i*) ∈ *R length ss* = *length ts funas-term t* ⊆ *F*
⋃ (*funas-term* ‘ *set ts*) ⊆ *F*
shows (*t*, *fill-holes D ts*) ∈ *R* ∧ *fill-holes D ts* =$_f$ (*D*, *ts*)
⟨*proof*⟩

fun *map-mctxt* :: (′*f* ⇒ ′*g*) ⇒ (′*f*, ′*v*) *mctxt* ⇒ (′*g*, ′*v*) *mctxt*
where
map-mctxt - (*MVar x*) = (*MVar x*) |
map-mctxt - (*MHole*) = *MHole* |
map-mctxt fg (*MFun f Cs*) = *MFun* (*fg f*) (*map* (*map-mctxt fg*) *Cs*)

fun *ground-mctxt* :: (′*f*, ′*v*) *mctxt* ⇒ *bool*
where
ground-mctxt (*MVar* -) = *False* |

ground-mctxt MHole = True |
ground-mctxt (MFun f Cs) = Ball (set Cs) ground-mctxt

lemma *ground-cap-till-funas* [*intro*]:
ground-mctxt (cap-till-funas F t)
⟨*proof*⟩

lemma *ground-eq-fill*: $t =_f (C,ss) \Longrightarrow$ *ground t = (ground-mctxt C ∧ (∀ s ∈ set ss. ground s))*
⟨*proof*⟩

lemma *ground-fill-holes*:
assumes *nh*: *num-holes C = length ss*
shows *ground (fill-holes C ss) = (ground-mctxt C ∧ (∀ s ∈ set ss. ground s))*
⟨*proof*⟩

lemma *split-vars-ground*: *split-vars t = (C,xs)* $\Longrightarrow$ *ground-mctxt C*
⟨*proof*⟩

lemma *split-vars-ground-vars*:
assumes *ground-mctxt C* **and** *num-holes C = length xs*
shows *split-vars (fill-holes C (map Var xs)) = (C, xs)*
⟨*proof*⟩

lemma *ground-map-mctxt*[*simp*]: *ground-mctxt (map-mctxt fg C) = ground-mctxt C*
⟨*proof*⟩

lemma *num-holes-map-mctxt*[*simp*]: *num-holes (map-mctxt fg C) = num-holes C*
⟨*proof*⟩

lemma *split-vars-map-mctxt*:
assumes *split*: *split-vars t = (map-mctxt fg C, xs)*
shows *split-vars (fill-holes C (map Var xs)) = (C, xs)*
⟨*proof*⟩

lemma *subst-eq-map-decomp*:
assumes $t \cdot \sigma =$ *map-funs-term fg s*
shows ∃ *C xs δs. s =*$_f$ *(C,δs) ∧ split-vars t = (map-mctxt fg C, xs) ∧ (∀ i < length xs.*
σ (xs ! i) = map-funs-term fg (δs ! i))
⟨*proof*⟩

lemma *map-funs-term-fill-holes*:
num-holes C = length ss $\Longrightarrow$
map-funs-term fg (fill-holes C ss) =$_f$ *(map-mctxt fg C, map (map-funs-term fg) ss)*
⟨*proof*⟩

lemma *eqf-MVarE*:
assumes $s =_f (MVar\ x, ss)$
shows $s = Var\ x\ ss = []$
⟨*proof*⟩

lemma *eqf-imp-subt*:
assumes *s*: $s =_f (C, ts)$
and *t*: $t \in set\ ts$
shows $s \trianglerighteq t$
⟨*proof*⟩

lemma *eqf-MFun-imp-strict-subt*:
assumes *s*:$s =_f (MFun\ f\ cs,\ ts)$
and *t*:$t \in set\ ts$
shows $s \triangleright t$
⟨*proof*⟩

fun *poss-mctxt* :: $('f, 'v)\ mctxt \Rightarrow pos\ set$
where
poss-mctxt $(MVar\ x) = \{[]\}$ |
poss-mctxt $MHole = \{\}$ |
poss-mctxt $(MFun\ f\ cs) = \{[]\} \cup \bigcup (set\ (map\ (\lambda\ i.\ (\lambda\ p.\ i\ \#\ p)\ `\ poss\text{-}mctxt\ (cs\ !\ i))\ [0\ ..<\ length\ cs]))$

lemma *poss-mctxt-simp* [*simp*]:
$poss\text{-}mctxt\ (MFun\ f\ cs) = \{[]\} \cup \{i\ \#\ p \mid i\ p.\ i < length\ cs \wedge p \in poss\text{-}mctxt\ (cs\ !\ i)\}$
⟨*proof*⟩
declare *poss-mctxt.simps*(*3*)[*simp del*]

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

fun *hole-poss* :: $('f, 'v)\ mctxt \Rightarrow pos\ set$
where
hole-poss $(MVar\ x) = \{\}$ |
hole-poss $MHole = \{[]\}$ |
hole-poss $(MFun\ f\ cs) = \bigcup (set\ (map\ (\lambda\ i.\ (\lambda\ p.\ i\ \#\ p)\ `\ hole\text{-}poss\ (cs\ !\ i))\ [0\ ..<\ length\ cs]))$

lemma *hole-poss-simp* [*simp*]:
$hole\text{-}poss\ (MFun\ f\ cs) = \{i\ \#\ p \mid i\ p.\ i < length\ cs \wedge p \in hole\text{-}poss\ (cs\ !\ i)\}$
⟨*proof*⟩
declare *hole-poss.simps*(*3*)[*simp del*]

lemma *hole-poss-empty-iff-num-holes-0*: $hole\text{-}poss\ C = \{\} \longleftrightarrow num\text{-}holes\ C = 0$
⟨*proof*⟩

lemma *mctxt-of-term-fill-holes* [*simp*]:
  fill-holes (*mctxt-of-term t*) [] = *t*
⟨*proof*⟩

lemma *hole-pos-not-in-poss-mctxt*:
  assumes *p* ∈ *hole-poss C*
  shows *p* ∉ *poss-mctxt C*
  ⟨*proof*⟩

lemma *hole-pos-in-filled-fun-poss*:
  assumes *is-Fun t*
  shows *hole-pos E* ∈ *fun-poss* ((*E* ·$_c$ *σ*)⟨*t* · *σ*⟩)
  ⟨*proof*⟩

fun
  subst-apply-mctxt :: ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*, '*w*) *gsubst* ⇒ ('*f*, '*w*) *mctxt* (**infixl** ·*mc* 67)
  where
    MHole ·*mc* - = *MHole* |
    (*MVar x*) ·*mc σ* = *mctxt-of-term* (*σ x*) |
    (*MFun f cs*) ·*mc σ* = *MFun f* [*c* ·*mc σ* . *c* ← *cs*]

lemma *subst-apply-mctxt-compose*: *C* ·*mc σ* ·*mc δ* = *C* ·*mc σ* ∘$_s$ *δ*
⟨*proof*⟩

lemma *subst-apply-mctxt-cong*: (⋀ *x*. *x* ∈ *vars-mctxt C* ⟹ *σ x* = *τ x*) ⟹ *C* ·*mc σ* = *C* ·*mc τ*
  ⟨*proof*⟩

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

lemma *subst-apply-mctxt-numholes*:
  shows *num-holes* (*c* ·*mc σ*) = *num-holes c*
⟨*proof*⟩

lemma *subst-apply-mctxt-fill-holes*:
  assumes *nh*: *num-holes c* = *length ts*
  shows (*fill-holes c ts*) · *σ* = *fill-holes* (*c* ·*mc σ*) [*ti* · *σ* . *ti* ← *ts*]
  ⟨*proof*⟩

lemma *subst-apply-mctxt-sound*:
  assumes *t* =$_f$ (*c*,*ts*)
  shows *t* · *σ* =$_f$ (*c* ·*mc σ*, [*ti* · *σ* . *ti* ← *ts*])
⟨*proof*⟩

fun *fill-holes-mctxt* :: ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *mctxt list* ⇒ ('*f*, '*v*) *mctxt*
where
fill-holes-mctxt (*MVar x*) - = *MVar x* |
fill-holes-mctxt MHole [] = *MHole* |
fill-holes-mctxt MHole [*t*] = *t* |
fill-holes-mctxt (*MFun f cs*) *ts* = (*MFun f* (*map* (λ *i*. *fill-holes-mctxt* (*cs* ! *i*) (*partition-holes ts cs* ! *i*)) [*0* ..< *length cs*]))

lemma *fill-holes-mctxt-Nil* [*simp*]:
fill-holes-mctxt C [] = *C*
⟨*proof*⟩

lemma *map-fill-holes-mctxt-zip* [*simp*]:
assumes *length ts* = *n*
shows *map* (λ(*x*, *y*). *fill-holes-mctxt x y*) (*zip* (*map mctxt-of-term ts*) (*replicate n* [])) =
map mctxt-of-term ts
⟨*proof*⟩

lemma *fill-holes-mctxt-MHole* [*simp*]:
length ts = *Suc 0* ⟹ *fill-holes-mctxt MHole ts* = *hd ts*
⟨*proof*⟩

lemma *partition-holes-fill-holes-mctxt-conv*:
fill-holes-mctxt (*MFun f Cs*) *ts* =
MFun f [*fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes ts Cs* ! *i*). *i* ← [*0* ..< *length Cs*]]
⟨*proof*⟩

lemma *partition-holes-fill-holes-mctxt-conv'*:
fill-holes-mctxt (*MFun f Cs*) *ts* =
MFun f (*map* (*case-prod fill-holes-mctxt*) (*zip Cs* (*partition-holes ts Cs*)))
⟨*proof*⟩

lemma *fill-holes-mctxt-mctxt-of-ctxt-mctxt-of-term* [*simp*]:
fill-holes-mctxt (*mctxt-of-ctxt C*) [*mctxt-of-term t*] = *mctxt-of-term* (*C*⟨*t*⟩)
⟨*proof*⟩

lemma *fill-holes-mctxt-mctxt-of-ctxt-MHole* [*simp*]:
fill-holes-mctxt (*mctxt-of-ctxt C*) [*MHole*] = *mctxt-of-ctxt C*
⟨*proof*⟩

lemma *partition-holes-fill-holes-conv'*:
fill-holes (*MFun f Cs*) *ts* =
Fun f (*map* (*case-prod fill-holes*) (*zip Cs* (*partition-holes ts Cs*)))
⟨*proof*⟩

lemma *fill-holes-mctxt-MFun-replicate-length* [*simp*]:
fill-holes-mctxt (*MFun c* (*replicate* (*length Cs*) *MHole*)) *Cs* = *MFun c Cs*

⟨*proof*⟩

lemma *fill-holes-MFun-replicate-length* [*simp*]:
fill-holes (*MFun c* (*replicate* (*length ts*) *MHole*)) *ts* = *Fun c ts*
⟨*proof*⟩

lemma *funas-mctxt-fill-holes-mctxt* [*simp*]:
assumes *num-holes C* = *length Ds*
shows *funas-mctxt* (*fill-holes-mctxt C Ds*) = *funas-mctxt C* ∪ ⋃(*set* (*map funas-mctxt Ds*))
(**is** *?f C Ds* = *?g C Ds*)
⟨*proof*⟩

lemma *fill-holes-mctxt-MFun*:
assumes *lCs*: *length Cs* = *length ts*
and *lss*: *length ss* = *length ts*
and *rec*: ⋀ *i*. *i* < *length ts* ⟹ *num-holes* (*Cs* ! *i*) = *length* (*ss* ! *i*) ∧ *fill-holes-mctxt* (*Cs* ! *i*) (*ss* ! *i*) = *ts* ! *i*
shows *fill-holes-mctxt* (*MFun f Cs*) (*concat ss*) = *MFun f ts*
⟨*proof*⟩

lemma *num-holes-fill-holes-mctxt*:
assumes *num-holes C* = *length Ds*
shows *num-holes* (*fill-holes-mctxt C Ds*) = *sum-list* (*map num-holes Ds*)
⟨*proof*⟩

lemma *fill-holes-mctxt-fill-holes*:
assumes *len-ds*: *length ds* = *num-holes c*
and *nh*: *num-holes* (*fill-holes-mctxt c ds*) = *length ss*
shows *fill-holes* (*fill-holes-mctxt c ds*) *ss* =
fill-holes c [*fill-holes* (*ds* ! *i*) (*partition-holes ss ds* ! *i*). *i* ← [*0* ..< *num-holes c*]]
⟨*proof*⟩

lemma *fill-holes-mctxt-sound*:
assumes *len-ds*: *length ds* = *num-holes c*
and *len-sss*: *length sss* = *num-holes c*
and *len-ts*: *length ts* = *num-holes c*
and *insts*: ⋀ *i*. *i* < *length ds* ⟹ *ts*!*i* =$_f$ (*ds*!*i*, *sss*!*i*)
shows *fill-holes c ts* =$_f$ (*fill-holes-mctxt c ds*, *concat sss*)
⟨*proof*⟩

lemma *poss-mctxt-fill-holes-mctxt*:
assumes *p* ∈ *poss-mctxt C*
shows *p* ∈ *poss-mctxt* (*fill-holes-mctxt C Cs*)
⟨*proof*⟩

fun *compose-mctxt* :: ('*f*, '*v*) *mctxt* ⇒ *nat* ⇒ ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *mctxt*
where
compose-mctxt C i Ci =

fill-holes-mctxt C [(*if i = j then Ci else MHole*). *j* ← [*0 ..< num-holes C*]]

lemma *funas-mctxt-compose-mctxt* [*simp*]:
assumes *i* < *num-holes C*
shows *funas-mctxt* (*compose-mctxt C i D*) = *funas-mctxt C* ∪ *funas-mctxt D*
⟨*proof*⟩

lemma *compose-mctxt-sound*:
assumes *s*: *s* =$_f$ (*C*, *bef* @ *si* # *aft*)
and *si*: *si* =$_f$ (*Ci*, *ts*)
and *i*: *i* = *length bef*
shows *s* =$_f$ (*compose-mctxt C i Ci*, *bef* @ *ts* @ *aft*)
⟨*proof*⟩

fun *mctxt-fill-partially-mctxts* :: ('*f*, '*v*) *term list* ⇒ ('*f*, '*v*) *term list* ⇒ ('*f*, '*v*) *mctxt list*
where
mctxt-fill-partially-mctxts [] *ts* = *map mctxt-of-term ts* |
mctxt-fill-partially-mctxts (*s* # *ss*) (*t* # *ts*) =
(*if s* = *t then* (*MHole* # *mctxt-fill-partially-mctxts ss ts*)
else (*mctxt-of-term t* # *mctxt-fill-partially-mctxts* (*s* # *ss*) *ts*))

fun
mctxt-fill-partially-fills ::
('*f*, '*v*) *term list* ⇒ ('*f*, '*v*) *term list* ⇒ ('*f*, '*v*) *term list list*
where
mctxt-fill-partially-fills [] *ts* = *map* (*const* []) *ts* |
mctxt-fill-partially-fills (*s* # *ss*) (*t* # *ts*) =
(*if s* = *t then* ([*s*] # *mctxt-fill-partially-fills ss ts*)
else ([] # *mctxt-fill-partially-fills* (*s* # *ss*) *ts*))

lemma *mctxt-fill-partially-mctxts-length* [*simp*]:
assumes *subseq ss ts*
shows *length* (*mctxt-fill-partially-mctxts ss ts*) = *length ts*
⟨*proof*⟩

lemma *mctxt-fill-partially-fills-length* [*simp*]:
assumes *subseq ss ts*
shows *length* (*mctxt-fill-partially-fills ss ts*) = *length ts*
⟨*proof*⟩

lemma *mctxt-fill-partially-numholes*:
assumes *subseq ss ts*
shows *sum-list* [*num-holes ci* . *ci* ← *mctxt-fill-partially-mctxts ss ts*] = *length ss*
⟨*proof*⟩

lemma *mctxt-fill-partially-sound*:
assumes *sl*: *subseq ss ts*
shows ⋀ *i*. *i* < *length ts* ⟹ *ts*!*i* =$_f$ (*mctxt-fill-partially-mctxts ss ts* ! *i*, *mc-*

txt-fill-partially-fills ss ts ! i)
⟨*proof*⟩

lemma *mctxt-fill-partially*:
assumes *ss*: *subseq ss ts*
and *t*: $t =_f (c,ts)$
shows $\exists\ d.\ t =_f (d,ss)$
⟨*proof*⟩

lemma *fill-holes-mctxt-map-mctxt-of-term-conv* [*simp*]:
assumes *num-holes C* = *length ts*
shows *fill-holes-mctxt C* (*map mctxt-of-term ts*) = *mctxt-of-term* (*fill-holes C ts*)
⟨*proof*⟩

lemma *fill-holes-mctxt-of-ctxt* [*simp*]:
fill-holes (*mctxt-of-ctxt C*) [*t*] = $C\langle t\rangle$
⟨*proof*⟩

definition
compose-cap-till P t i C =
fill-holes-mctxt (*cap-till P t*) (*map mctxt-of-term* (*take i* (*uncap-till P t*)) @
C # *map mctxt-of-term* (*drop* (*Suc i*) (*uncap-till P t*)))

abbreviation *compose-cap-till-funas F* ≡ *compose-cap-till* (*if-Fun-in-set F*)

lemma *fill-holes-compose-cap-till*:
assumes *i* < *num-holes* (*cap-till P s*) **and** *num-holes C* = *length ts*
shows *fill-holes* (*compose-cap-till P s i C*) *ts* =
fill-holes (*cap-till P s*) (*take i* (*uncap-till P s*) @ *fill-holes C ts* # *drop* (*Suc i*)
(*uncap-till P s*))
(**is** - = *fill-holes* - *?ss*)
⟨*proof*⟩

lemma *in-uncap-till-funas*:
assumes *root*: *root u* = *Some fn fn* ∈ *F*
and $t = C\langle u\rangle$
shows ∃ *i* < *length* (*uncap-till-funas F t*). ∃ *D*. *uncap-till-funas F t* ! *i* = $D\langle u\rangle$ ∧
mctxt-of-ctxt C = *compose-cap-till-funas F t i* (*mctxt-of-ctxt D*)
⟨*proof*⟩

lemma *uncap-till-funas-fill-holes-cancel* [*simp*]:
assumes *num-holes C* = *length ts* **and** *ground-mctxt C*
and *funas-mctxt C* ⊆ − *F*
shows *uncap-till-funas F* (*fill-holes C ts*) = *concat* (*map* (*uncap-till-funas F*) *ts*)
⟨*proof*⟩

lemma *uncap-till-funas-fill-holes-cap-till-funas* [*simp*]:
assumes *num-holes* (*cap-till-funas F s*) = *length ts*
shows *uncap-till-funas F* (*fill-holes* (*cap-till-funas F s*) *ts*) =

$concat\ (map\ (uncap\text{-}till\text{-}funas\ F)\ ts)$
⟨*proof*⟩

lemma *Ball-atLeast0LessThan-partition-holes-conv* [*simp*]:
$(\forall\, i \in \{0\ ..<\ length\ Cs\}.\ \forall\, x \in set\ (partition\text{-}holes\ xs\ Cs\ !\ i).\ P\ x) =$
$(\forall\, x \in \bigcup (set\ (map\ set\ (partition\text{-}holes\ xs\ Cs))).\ P\ x)$
⟨*proof*⟩

lemma *ground-fill-holes-mctxt* [*simp*]:
$num\text{-}holes\ C = length\ Ds \Longrightarrow$
$ground\text{-}mctxt\ (fill\text{-}holes\text{-}mctxt\ C\ Ds) \longleftrightarrow ground\text{-}mctxt\ C \wedge (\forall\, D \in set\ Ds.\ ground\text{-}mctxt\ D)$
⟨*proof*⟩

lemma *concat-map-uncap-till-funas-map-subst-apply-uncap-till-funas* [*simp*]:
$concat\ (map\ (uncap\text{-}till\text{-}funas\ F)\ (map\ (\lambda s.\ s \cdot \sigma)\ (uncap\text{-}till\text{-}funas\ F\ t))) = uncap\text{-}till\text{-}funas\ F\ (t \cdot \sigma)$
⟨*proof*⟩

lemma *concat-uncap-till-subst-conv*:
$concat\ (map\ (\lambda i.\ uncap\text{-}till\text{-}funas\ F\ ((uncap\text{-}till\text{-}funas\ F\ t\ !\ i) \cdot \sigma))\ [0\ ..<\ length\ (uncap\text{-}till\text{-}funas\ F\ t)]) =$
$uncap\text{-}till\text{-}funas\ F\ (t \cdot \sigma)$
⟨*proof*⟩

lemma *the-root-uncap-till-funas*:
$is\text{-}Fun\ t \Longrightarrow the\ (root\ t) \in F \Longrightarrow uncap\text{-}till\text{-}funas\ F\ t = [t]$
⟨*proof*⟩

lemma *funas-cap-till-subset*:
$funas\text{-}mctxt\ (cap\text{-}till\ P\ t) \subseteq funas\text{-}term\ t$
⟨*proof*⟩

lemma *funas-uncap-till-subset*:
$s \in set\ (uncap\text{-}till\ P\ t) \Longrightarrow funas\text{-}term\ s \subseteq funas\text{-}term\ t$
⟨*proof*⟩

lemma *ground-mctxt-subst-apply-context* [*simp*]:
$ground\text{-}mctxt\ C \Longrightarrow C \cdot mc\ \sigma = C$
⟨*proof*⟩

lemma *vars-term-fill-holes* [*simp*]:
$num\text{-}holes\ C = length\ ts \Longrightarrow ground\text{-}mctxt\ C \Longrightarrow$
$vars\text{-}term\ (fill\text{-}holes\ C\ ts) = \bigcup (vars\text{-}term\ `\ set\ ts)$
⟨*proof*⟩

6.1.3 Semilattice Structures

instantiation *mctxt* :: (*type*, *type*) *inf*

begin

fun *inf-mctxt* :: ('*a*, '*b*) *mctxt* ⇒ ('*a*, '*b*) *mctxt* ⇒ ('*a*, '*b*) *mctxt*
  where
    MHole ⊓ *D* = *MHole* |
    C ⊓ *MHole* = *MHole* |
    MVar x ⊓ *MVar y* = (*if x* = *y then MVar x else MHole*) |
    MFun f Cs ⊓ *MFun g Ds* =
    (*if f* = *g* ∧ *length Cs* = *length Ds then MFun f* (*map* (*case-prod* (⊓)) (*zip Cs Ds*))
    else MHole) |
    C ⊓ *D* = *MHole*

instance ⟨*proof*⟩

end

lemma *inf-mctxt-idem* [*simp*]:
  fixes *C* :: ('*f*, '*v*) *mctxt*
  shows *C* ⊓ *C* = *C*
  ⟨*proof*⟩

lemma *inf-mctxt-MHole2* [*simp*]:
  C ⊓ *MHole* = *MHole*
  ⟨*proof*⟩

lemma *inf-mctxt-comm* [*ac-simps*]:
  (*C* :: ('*f*, '*v*) *mctxt*) ⊓ *D* = *D* ⊓ *C*
  ⟨*proof*⟩

lemma *inf-mctxt-assoc* [*ac-simps*]:
  fixes *C* :: ('*f*, '*v*) *mctxt*
  shows *C* ⊓ *D* ⊓ *E* = *C* ⊓ (*D* ⊓ *E*)
⟨*proof*⟩

instantiation *mctxt* :: (*type*, *type*) *order*
begin

definition (*C* :: ('*a*, '*b*) *mctxt*) ≤ *D* ⟷ *C* ⊓ *D* = *C*
definition (*C* :: ('*a*, '*b*) *mctxt*) < *D* ⟷ *C* ≤ *D* ∧ ¬ *D* ≤ *C*

instance
  ⟨*proof*⟩

end

inductive *less-eq-mctxt'* :: ('*f*, '*v*) *mctxt* ⇒ ('*f*,'*v*) *mctxt* ⇒ *bool* **where**
  less-eq-mctxt' MHole u
| *less-eq-mctxt'* (*MVar v*) (*MVar v*)

| *length cs = length ds* $\Longrightarrow$ ($\bigwedge$*i. i < length cs* $\Longrightarrow$ *less-eq-mctxt′ (cs ! i) (ds ! i))*
$\Longrightarrow$ *less-eq-mctxt′ (MFun f cs) (MFun f ds)*

lemma *less-eq-mctxt-prime*: *C* $\leq$ *D* $\longleftrightarrow$ *less-eq-mctxt′ C D*
⟨*proof*⟩

lemmas *less-eq-mctxt-induct = less-eq-mctxt′.induct*[*folded less-eq-mctxt-prime*, *consumes 1*]
lemmas *less-eq-mctxt-intros = less-eq-mctxt′.intros*[*folded less-eq-mctxt-prime*]

lemma *less-eq-mctxtI2*:
C = MHole $\Longrightarrow$ *C* $\leq$ *MHole*
C = MHole $\vee$ *C = MVar v* $\Longrightarrow$ *C* $\leq$ *MVar v*
C = MHole $\vee$ *C = MFun f cs* $\wedge$ *length cs = length ds* $\wedge$ ($\forall$ *i. i < length cs* $\longrightarrow$
cs ! i $\leq$ *ds ! i*) $\Longrightarrow$ *C* $\leq$ *MFun f ds*
⟨*proof*⟩

lemma *less-eq-mctxt-MHoleE2*:
assumes *C* $\leq$ *MHole*
obtains (*MHole*) *C = MHole*
⟨*proof*⟩

lemma *less-eq-mctxt-MVarE2*:
assumes *C* $\leq$ *MVar v*
obtains (*MHole*) *C = MHole* | (*MVar*) *C = MVar v*
⟨*proof*⟩

lemma *less-eq-mctxt-MFunE2*:
assumes *C* $\leq$ *MFun f ds*
obtains (*MHole*) *C = MHole*
| (*MFun*) *cs* **where** *C = MFun f cs length cs = length ds* $\bigwedge$*i. i < length cs* $\Longrightarrow$
cs ! i $\leq$ *ds ! i*
⟨*proof*⟩

lemmas *less-eq-mctxtE2 = less-eq-mctxt-MHoleE2 less-eq-mctxt-MVarE2 less-eq-mctxt-MFunE2*

lemma *less-eq-mctxtI1*:
MHole $\leq$ *D*
D = MVar v $\Longrightarrow$ *MVar v* $\leq$ *D*
D = MFun f ds $\Longrightarrow$ *length cs = length ds* $\Longrightarrow$ ($\bigwedge$*i. i < length cs* $\Longrightarrow$ *cs ! i* $\leq$ *ds*
! i) $\Longrightarrow$ *MFun f cs* $\leq$ *D*
⟨*proof*⟩

lemma *less-eq-mctxt-MVarE1*:
assumes *MVar v* $\leq$ *D*
obtains (*MVar*) *D = MVar v*
⟨*proof*⟩

lemma *less-eq-mctxt-MFunE1*:

assumes *MFun f cs* ≤ *D*
obtains (*MFun*) *ds* **where** *D* = *MFun f ds length cs* = *length ds* ⋀*i*. *i* < *length cs* ⟹ *cs* ! *i* ≤ *ds* ! *i*
⟨*proof*⟩

lemmas *less-eq-mctxtE1* = *less-eq-mctxt-MVarE1 less-eq-mctxt-MFunE1*

instance *mctxt* :: (*type*, *type*) *semilattice-inf*
⟨*proof*⟩

fun *inf-mctxt-args* :: ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *mctxt list*
where
inf-mctxt-args MHole D = [*MHole*] |
inf-mctxt-args C MHole = [*C*] |
inf-mctxt-args (*MVar x*) (*MVar y*) = (*if x* = *y then* [] *else* [*MVar x*]) |
inf-mctxt-args (*MFun f Cs*) (*MFun g Ds*) =
(*if f* = *g* ∧ *length Cs* = *length Ds then concat* (*map* (*case-prod inf-mctxt-args*) (*zip Cs Ds*))
else [*MFun f Cs*]) |
inf-mctxt-args C D = [*C*]

lemma *inf-mctxt-args-MHole2* [*simp*]:
inf-mctxt-args C MHole = [*C*]
⟨*proof*⟩

lemma *fill-holes-mctxt-replicate-MHole* [*simp*]:
fill-holes-mctxt C (*replicate* (*num-holes C*) *MHole*) = *C*
⟨*proof*⟩

lemma *num-holes-inf-mctxt*:
num-holes (*C* ⊓ *D*) = *length* (*inf-mctxt-args C D*)
⟨*proof*⟩

lemma *length-inf-mctxt-args*:
length (*inf-mctxt-args D C*) = *length* (*inf-mctxt-args C D*)
⟨*proof*⟩

lemma *inf-mctxt-args-same* [*simp*]:
inf-mctxt-args C C = *replicate* (*num-holes C*) *MHole*
⟨*proof*⟩

lemma *inf-mctxt-inf-mctxt-args*:
fill-holes-mctxt (*C* ⊓ *D*) (*inf-mctxt-args C D*) = *C*
⟨*proof*⟩

lemma *inf-mctxt-inf-mctxt-args2*:
fill-holes-mctxt (*C* ⊓ *D*) (*inf-mctxt-args D C*) = *D*
⟨*proof*⟩

instantiation *mctxt* :: (*type*, *type*) *sup*
begin

fun *sup-mctxt* :: ('*a*, '*b*) *mctxt* ⇒ ('*a*, '*b*) *mctxt* ⇒ ('*a*, '*b*) *mctxt*
  where
    MHole ⊔ *D* = *D* |
    C ⊔ *MHole* = *C* |
    MVar x ⊔ *MVar y* = (*if x* = *y then MVar x else undefined*) |
    MFun f Cs ⊔ *MFun g Ds* =
    (*if f* = *g* ∧ *length Cs* = *length Ds then MFun f* (*map* (*case-prod* (⊔)) (*zip Cs Ds*))
    else undefined) |
    (*C* :: ('*a*, '*b*) *mctxt*) ⊔ *D* = *undefined*

instance ⟨*proof*⟩

end

lemma *sup-mctxt-idem* [*simp*]:
  fixes *C* :: ('*f*, '*v*) *mctxt*
  shows *C* ⊔ *C* = *C*
  ⟨*proof*⟩

lemma *sup-mctxt-MHole* [*simp*]: *C* ⊔ *MHole* = *C*
  ⟨*proof*⟩

lemma *sup-mctxt-comm* [*ac-simps*]:
  fixes *C* :: ('*f*, '*v*) *mctxt*
  shows *C* ⊔ *D* = *D* ⊔ *C*
  ⟨*proof*⟩

(⊔) is defined on compatible multihole-contexts. Note that compatibility is not transitive.

inductive-set *comp-mctxt* :: (('*a*, '*b*) *mctxt* × ('*a*, '*b*) *mctxt*) *set*
  where
    MHole1: (*MHole*, *D*) ∈ *comp-mctxt* |
    MHole2: (*C*, *MHole*) ∈ *comp-mctxt* |
    MVar: *x* = *y* ⟹ (*MVar x*, *MVar y*) ∈ *comp-mctxt* |
    MFun: *f* = *g* ⟹ *length Cs* = *length Ds* ⟹ ∀ *i* < *length Ds*. (*Cs* ! *i*, *Ds* ! *i*) ∈ *comp-mctxt* ⟹
    (*MFun f Cs*, *MFun g Ds*) ∈ *comp-mctxt*

lemma *comp-mctxt-refl*:
  (*C*, *C*) ∈ *comp-mctxt*
  ⟨*proof*⟩

lemma *comp-mctxt-sym*:
  assumes (*C*, *D*) ∈ *comp-mctxt*

shows $(D,\ C) \in$ *comp-mctxt*
⟨*proof*⟩

lemma *sup-mctxt-assoc* [*ac-simps*]:
assumes $(C,\ D) \in$ *comp-mctxt* **and** $(D,\ E) \in$ *comp-mctxt*
shows $C \sqcup D \sqcup E = C \sqcup (D \sqcup E)$
⟨*proof*⟩

No instantiation to *semilattice-sup* possible, since $(\sqcup)$ is only partially defined on terms (e.g., it is not associative in general).

interpretation *mctxt-order-bot*: *order-bot MHole* $(\leq)$ $(<)$
⟨*proof*⟩

lemma *sup-mctxt-ge1* [*simp*]:
assumes $(C,\ D) \in$ *comp-mctxt*
shows $C \leq C \sqcup D$
⟨*proof*⟩

lemma *sup-mctxt-ge2* [*simp*]:
assumes $(C,\ D) \in$ *comp-mctxt*
shows $D \leq C \sqcup D$
⟨*proof*⟩

lemma *sup-mctxt-least*:
assumes $(D,\ E) \in$ *comp-mctxt*
and $D \leq C$ **and** $E \leq C$
shows $D \sqcup E \leq C$
⟨*proof*⟩

lemma *inf-mctxt-args-MHole*:
assumes $(C,\ D) \in$ *comp-mctxt* **and** $i <$ *length* (*inf-mctxt-args C D*)
shows *inf-mctxt-args C D* ! $i =$ *MHole* $\vee$ *inf-mctxt-args D C* ! $i =$ *MHole*
⟨*proof*⟩

lemma *rsteps-mctxt*:
assumes $s =_f (C,\ ss)$ **and** $t =_f (C,\ ts)$
and $\forall i <$*length ss*. (*ss* ! i, *ts* ! i) $\in$ (*rstep R*)*
shows $(s,\ t) \in$ (*rstep R*)*
⟨*proof*⟩

fun *sup-mctxt-args* :: $('f,\ 'v)$ *mctxt* $\Rightarrow$ $('f,\ 'v)$ *mctxt* $\Rightarrow$ $('f,\ 'v)$ *mctxt list*
where
sup-mctxt-args MHole D = [*D*] |
sup-mctxt-args C MHole = *replicate* (*num-holes C*) *MHole* |
sup-mctxt-args (*MVar x*) (*MVar y*) = (*if x* = *y then* [] *else undefined*) |
sup-mctxt-args (*MFun f Cs*) (*MFun g Ds*) =
(*if f* = *g* $\wedge$ *length Cs* = *length Ds then concat* (*map* (*case-prod sup-mctxt-args*) (*zip Cs Ds*))

else undefined) |
sup-mctxt-args C D = *undefined*

lemma *sup-mctxt-args-MHole2* [*simp*]:
sup-mctxt-args C MHole = *replicate* (*num-holes C*) *MHole*
⟨*proof*⟩

lemma *num-holes-sup-mctxt-args*:
assumes (*C*, *D*) ∈ *comp-mctxt*
shows *num-holes C* = *length* (*sup-mctxt-args C D*)
⟨*proof*⟩

lemma *sup-mctxt-sup-mctxt-args*:
assumes (*C*, *D*) ∈ *comp-mctxt*
shows *fill-holes-mctxt C* (*sup-mctxt-args C D*) = *C* ⊔ *D*
⟨*proof*⟩

lemma *sup-mctxt-args*:
assumes (*C*, *D*) ∈ *comp-mctxt*
shows *sup-mctxt-args C D* = *inf-mctxt-args* (*C* ⊔ *D*) *C*
⟨*proof*⟩

lemma *term-for-mctxt*:
fixes *C* :: ('*f*, '*v*) *mctxt*
obtains *t* **and** *ts* **where** *t* =$_f$ (*C*, *ts*)
⟨*proof*⟩

lemma *comp-mctxt-eqfE*:
assumes (*C*, *D*) ∈ *comp-mctxt*
obtains *s* **and** *ss* **and** *ts* **where** *s* =$_f$ (*C*, *ss*) **and** *s* =$_f$ (*D*, *ts*)
⟨*proof*⟩

lemma *eqf-comp-mctxt*:
assumes *s* =$_f$ (*C*, *ss*) **and** *s* =$_f$ (*D*, *ts*)
shows (*C*, *D*) ∈ *comp-mctxt*
⟨*proof*⟩

lemma *comp-mctxt-iff*:
(*C*, *D*) ∈ *comp-mctxt* ⟷ (∃ *s ss ts*. *s* =$_f$ (*C*, *ss*) ∧ *s* =$_f$ (*D*, *ts*))
⟨*proof*⟩

lemma *hole-poss-parallel-pos* [*simp*]:
assumes *p* ∈ *hole-poss C* **and** *q* ∈ *hole-poss C* **and** *p* ≠ *q*
shows *parallel-pos p q*
⟨*proof*⟩

lemma *eq-fill-induct* [*consumes 1*, *case-names MHole MVar MFun*]:
assumes *t* =$_f$ (*C*, *ts*)
and ⋀*t*. *P t MHole* [*t*]

and $\bigwedge x.\ P\ (Var\ x)\ (MVar\ x)\ []$
and $\bigwedge f\ ss\ Cs\ ts.\ [\![length\ Cs = length\ ss;\ sum\text{-}list\ (map\ num\text{-}holes\ Cs) = length\ ts;$
$\forall i < length\ ss.\ ss\ !\ i =_f\ (Cs\ !\ i,\ partition\text{-}holes\ ts\ Cs\ !\ i) \wedge$
$P\ (ss\ !\ i)\ (Cs\ !\ i)\ (partition\text{-}holes\ ts\ Cs\ !\ i)]\!]$
$\Longrightarrow P\ (Fun\ f\ ss)\ (MFun\ f\ Cs)\ ts$
shows $P\ t\ C\ ts$
⟨*proof*⟩

lemma *hole-poss-subset-poss*:
assumes $s =_f\ (C,\ ss)$
shows $hole\text{-}poss\ C \subseteq poss\ s$
⟨*proof*⟩

fun *hole-num*
where
$hole\text{-}num\ []\ MHole = 0\ |$
$hole\text{-}num\ (i\ \#\ q)\ (MFun\ f\ Cs) = sum\text{-}list\ (map\ num\text{-}holes\ (take\ i\ Cs)) + hole\text{-}num\ q\ (Cs\ !\ i)$

lemma *hole-poss-nth-subt-at*:
assumes $t =_f\ (C,\ ts)$ **and** $p \in hole\text{-}poss\ C$
shows $hole\text{-}num\ p\ C < length\ ts \wedge t\ |\text{-}\ p = ts\ !\ hole\text{-}num\ p\ C$
⟨*proof*⟩

lemma *eqf-Fun-MFun*:
assumes $Fun\ f\ ss =_f\ (MFun\ g\ Cs,\ ts)$
shows $g = f \wedge length\ Cs = length\ ss \wedge sum\text{-}list\ (map\ num\text{-}holes\ Cs) = length\ ts \wedge$
$(\forall i < length\ ss.\ ss\ !\ i =_f\ (Cs\ !\ i,\ partition\text{-}holes\ ts\ Cs\ !\ i))$
⟨*proof*⟩

lemma *fill-holes-eq-Var-cases*:
assumes $num\text{-}holes\ C = length\ ts$
and $fill\text{-}holes\ C\ ts = Var\ x$
obtains $C = MHole \wedge ts = [Var\ x] \mid C = MVar\ x \wedge ts = []$
⟨*proof*⟩

lemma *num-holes-inf-mctxt-le*:
assumes $s =_f\ (C,\ ts)$ **and** $s =_f\ (D,\ us)$
shows $num\text{-}holes\ (C \sqcap D) \leq num\text{-}holes\ C + num\text{-}holes\ D$
⟨*proof*⟩

lemma *map-inf-mctxt-zip-mctxt-of-term* [*simp*]:
$map\ (\lambda(x,\ y).\ x \sqcap y)\ (zip\ (map\ mctxt\text{-}of\text{-}term\ ts)\ (map\ mctxt\text{-}of\text{-}term\ ts)) = map\ mctxt\text{-}of\text{-}term\ ts$
⟨*proof*⟩

lemma *inf-mctxt-ctxt-apply-term* [*simp*]:

mctxt-of-term $(C\langle t\rangle) \sqcap$ *mctxt-of-ctxt* $C =$ *mctxt-of-ctxt* C
mctxt-of-ctxt $C \sqcap$ *mctxt-of-term* $(C\langle t\rangle) =$ *mctxt-of-ctxt* C
⟨*proof*⟩

lemma *inf-fill-holes-mctxt-MHoles*:
num-holes $C =$ *length* $Cs \Longrightarrow$ *length* $Ds =$ *length* $Cs \Longrightarrow$
$\forall i<$*length* $Cs.$ Cs ! $i =$ *MHole* $\vee$ Ds ! $i =$ *MHole* $\Longrightarrow$
fill-holes-mctxt C $Cs \sqcap$ *fill-holes-mctxt* C $Ds = C$
⟨*proof*⟩

lemma *inf-fill-holes-mctxt-two-MHoles* [*simp*]: *num-holes* $C = 2 \Longrightarrow$
fill-holes-mctxt C [*MHole*, D] $\sqcap$ *fill-holes-mctxt* C [E, *MHole*] $= C$
⟨*proof*⟩

lemma *two-subterms-cases*:
assumes $s = C\langle t\rangle$ **and** $s = D\langle u\rangle$
obtains (*eq*) $C = D$ **and** $t = u$
| (*nested1*) C' **where** $C' \neq \Box$ **and** $C = D \circ_c C'$
| (*nested2*) D' **where** $D' \neq \Box$ **and** $D = C \circ_c D'$
| (*parallel1*) E **where** *num-holes* $E = 2$
and *mctxt-of-ctxt* $C =$ *fill-holes-mctxt* E [*MHole*, *mctxt-of-term* u]
and *mctxt-of-ctxt* $D =$ *fill-holes-mctxt* E [*mctxt-of-term* t, *MHole*]
| (*parallel2*) E **where** *num-holes* $E = 2$
and *mctxt-of-ctxt* $C =$ *fill-holes-mctxt* E [*mctxt-of-term* u, *MHole*]
and *mctxt-of-ctxt* $D =$ *fill-holes-mctxt* E [*MHole*, *mctxt-of-term* t]
⟨*proof*⟩

lemma *two-hole-ctxt-inf-conv*:
num-holes $E = 2 \Longrightarrow$
mctxt-of-ctxt $C =$ *fill-holes-mctxt* E [*MHole*, *mctxt-of-term* u] $\Longrightarrow$
mctxt-of-ctxt $D =$ *fill-holes-mctxt* E [*mctxt-of-term* t, *MHole*] $\Longrightarrow$
mctxt-of-ctxt $C \sqcap$ *mctxt-of-ctxt* $D = E$
⟨*proof*⟩

lemma *map-length-take-partition-by*:
$i <$ *length* $ys \Longrightarrow$ *sum-list* $ys =$ *length* $xs \Longrightarrow$
map length (*take* i (*partition-by* xs ys)) $=$ *take* i ys
⟨*proof*⟩

Closure under contexts can be lifted to multihole contexts.

lemma *ctxt-imp-mctxt*:
assumes $\forall s\ t\ C.\ (s, t) \in R \longrightarrow (C\langle s\rangle, C\langle t\rangle) \in R$
and $(t, u) \in R$
and *num-holes* $C =$ *length* $ss_1 +$ *length* $ss_2 + 1$
shows (*fill-holes* C (ss_1 @ t # ss_2), *fill-holes* C (ss_1 @ u # ss_2)) $\in R$
⟨*proof*⟩

lemma *mctxt-of-term-fill-holes′*:
num-holes $C =$ *length* $ts \Longrightarrow$ *mctxt-of-term* (*fill-holes* C ts) $=$ *fill-holes-mctxt* C

(*map mctxt-of-term ts*)
⟨*proof*⟩

lemma *vars-term-fill-holes'*:
num-holes C = *length ts* ⟹ *vars-term* (*fill-holes C ts*) = ⋃(*vars-term* ‘ *set ts*) ∪ *vars-mctxt C*
⟨*proof*⟩

lemma *vars-mctxt-linear*: **assumes** *t* =$_f$ (*C*, *ts*)
linear-term t
shows *vars-mctxt C* ∩ ⋃ (*vars-term* ‘ *set ts*) = {}
⟨*proof*⟩

lemma *mctxt-of-term-var-subst*:
mctxt-of-term (*t* · (*Var* ∘ *f*)) = *map-vars-mctxt f* (*mctxt-of-term t*)
⟨*proof*⟩

lemma *subst-apply-mctxt-map-vars-mctxt-conv*:
C ·*mc* (*Var* ∘ *f*) = *map-vars-mctxt f C*
⟨*proof*⟩

lemma *map-vars-mctxt-mono*:
C ≤ *D* ⟹ *map-vars-mctxt f C* ≤ *map-vars-mctxt f D*
⟨*proof*⟩

lemma *map-vars-mctxt-less-eq-decomp*:
assumes *C* ≤ *map-vars-mctxt f D*
obtains *C'* **where** *map-vars-mctxt f C'* = *C C'* ≤ *D*
⟨*proof*⟩

6.1.4 All positions of a multi-hole context

fun *all-poss-mctxt* :: ('*f*, '*v*) *mctxt* ⇒ *pos set*
where
all-poss-mctxt (*MVar x*) = {[]}
| *all-poss-mctxt MHole* = {[]}
| *all-poss-mctxt* (*MFun f cs*) = {[]} ∪ ⋃(*set* (*map* (λ *i*. (λ *p*. *i* # *p*) ‘ *all-poss-mctxt* (*cs* ! *i*)) [*0* ..< *length cs*]))

lemma *all-poss-mctxt-simp* [*simp*]:
all-poss-mctxt (*MFun f cs*) = {[]} ∪ {*i* # *p* | *i p*. *i* < *length cs* ∧ *p* ∈ *all-poss-mctxt* (*cs* ! *i*)}
⟨*proof*⟩

declare *all-poss-mctxt.simps*(*3*)[*simp del*]

lemma *all-poss-mctxt-conv*:
all-poss-mctxt C = *poss-mctxt C* ∪ *hole-poss C*
⟨*proof*⟩

lemma *root-in-all-poss-mctxt*[*simp*]:
[] ∈ *all-poss-mctxt C*
⟨*proof*⟩

lemma *hole-poss-mctxt-of-term*[*simp*]:
hole-poss (*mctxt-of-term t*) = {}
⟨*proof*⟩

lemma *poss-mctxt-mctxt-of-term*[*simp*]:
poss-mctxt (*mctxt-of-term t*) = *poss t*
⟨*proof*⟩

lemma *hole-poss-subst*: *hole-poss* (*C* ·*mc* σ) = *hole-poss C*
⟨*proof*⟩

lemma *all-poss-mctxt-mctxt-of-term*[*simp*]:
all-poss-mctxt (*mctxt-of-term t*) = *poss t*
⟨*proof*⟩

lemma *mctxt-of-term-leq-imp-eq*:
mctxt-of-term t ≤ *C* ⟷ *mctxt-of-term t* = *C*
⟨*proof*⟩

lemma *mctxt-of-term-inj*:
mctxt-of-term s = *mctxt-of-term t* ⟷ *s* = *t*
⟨*proof*⟩

lemma *all-poss-mctxt-map-vars-mctxt* [*simp*]:
all-poss-mctxt (*map-vars-mctxt f C*) = *all-poss-mctxt C*
⟨*proof*⟩

lemma *fill-holes-mctxt-extends-all-poss*:
assumes *length Ds* = *num-holes C* **shows** *all-poss-mctxt C* ⊆ *all-poss-mctxt* (*fill-holes-mctxt C Ds*)
⟨*proof*⟩

lemma *eqF-substD*:
assumes *t* · σ =$_f$ (*C*, *ss*)
hole-poss C ⊆ *poss t*
shows ∃ *D ts*. *t* =$_f$ (*D*, *ts*) ∧ *C* = *D* ·*mc* σ ∧ *ss* = *map* (λ *ti*. *ti* · σ) *ts*
⟨*proof*⟩

6.1.5 More operations on multi-hole contexts

fun *root-mctxt* :: ('*f*, '*v*) *mctxt* ⇒ ('*f* × *nat*) *option* **where**
root-mctxt MHole = *None*

| *root-mctxt* (*MVar x*) = *None*
| *root-mctxt* (*MFun f Cs*) = *Some* (*f*, *length Cs*)

fun *mreplace-at* :: ('*f*, '*v*) *mctxt* ⇒ *pos* ⇒ ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *mctxt* **where**
mreplace-at C [] *D* = *D*
| *mreplace-at* (*MFun f Cs*) (*i* # *p*) *D* = *MFun f* (*take i Cs* @ *mreplace-at* (*Cs* ! *i*) *p D* # *drop* (*i*+*1*) *Cs*)

fun *subm-at* :: ('*f*, '*v*) *mctxt* ⇒ *pos* ⇒ ('*f*, '*v*) *mctxt* **where**
subm-at C [] = *C*
| *subm-at* (*MFun f Cs*) (*i* # *p*) = *subm-at* (*Cs* ! *i*) *p*

lemma *subm-at-hole-poss*[*simp*]:
p ∈ *hole-poss C* ⟹ *subm-at C p* = *MHole*
⟨*proof*⟩

lemma *subm-at-mctxt-of-term*:
p ∈ *poss t* ⟹ *subm-at* (*mctxt-of-term t*) *p* = *mctxt-of-term* (*subt-at t p*)
⟨*proof*⟩

lemma *subm-at-mreplace-at*[*simp*]:
p ∈ *all-poss-mctxt C* ⟹ *subm-at* (*mreplace-at C p D*) *p* = *D*
⟨*proof*⟩

lemma *replace-at-subm-at*[*simp*]:
p ∈ *all-poss-mctxt C* ⟹ *mreplace-at C p* (*subm-at C p*) = *C*
⟨*proof*⟩

lemma *all-poss-mctxt-mreplace-atI1*:
p ∈ *all-poss-mctxt C* ⟹ *q* ∈ *all-poss-mctxt C* ⟹ ¬ (*p* <$_p$ *q*) ⟹ *q* ∈ *all-poss-mctxt* (*mreplace-at C p D*)
⟨*proof*⟩

lemma *funas-mctxt-sup-mctxt*:
(*C*, *D*) ∈ *comp-mctxt* ⟹ *funas-mctxt* (*C* ⊔ *D*) = *funas-mctxt C* ∪ *funas-mctxt D*
⟨*proof*⟩

lemma *mctxt-of-term-not-hole* [*simp*]:
mctxt-of-term t ≠ *MHole*
⟨*proof*⟩

lemma *funas-mctxt-mctxt-of-term* [*simp*]:
funas-mctxt (*mctxt-of-term t*) = *funas-term t*
⟨*proof*⟩

lemma *funas-mctxt-mreplace-at*:
assumes *p* ∈ *all-poss-mctxt C*

shows *funas-mctxt* (*mreplace-at C p D*) ⊆ *funas-mctxt C* ∪ *funas-mctxt D*
⟨*proof*⟩

lemma *funas-mctxt-mreplace-at-hole*:
assumes *p* ∈ *hole-poss C*
shows *funas-mctxt* (*mreplace-at C p D*) = *funas-mctxt C* ∪ *funas-mctxt D* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *map-vars-mctxt-fill-holes-mctxt*:
assumes *num-holes C* = *length Cs*
shows *map-vars-mctxt f* (*fill-holes-mctxt C Cs*) = *fill-holes-mctxt* (*map-vars-mctxt f C*) (*map* (*map-vars-mctxt f*) *Cs*)
⟨*proof*⟩

lemma *map-vars-mctxt-map-vars-mctxt*[*simp*]:
shows *map-vars-mctxt f* (*map-vars-mctxt g C*) = *map-vars-mctxt* (*f* ∘ *g*) *C*
⟨*proof*⟩

lemma *funas-mctxt-fill-holes*:
assumes *num-holes C* = *length ts*
shows *funas-term* (*fill-holes C ts*) = *funas-mctxt C* ∪ ⋃(*set* (*map funas-term ts*))
⟨*proof*⟩

lemma *mctxt-neq-mholeE*:
x ≠ *MHole* ⟹ (⋀*v*. *x* = *MVar v* ⟹ *P*) ⟹ (⋀*f Cs*. *x* = *MFun f Cs* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *prefix-comp-mctxt*:
C ≤ *E* ⟹ *D* ≤ *E* ⟹ (*C*, *D*) ∈ *comp-mctxt*
⟨*proof*⟩

lemma *less-eq-mctxt-sup-conv1*:
(*C*, *D*) ∈ *comp-mctxt* ⟹ *C* ≤ *D* ⟷ *C* ⊔ *D* = *D*
⟨*proof*⟩

lemma *less-eq-mctxt-sup-conv2*:
(*C*, *D*) ∈ *comp-mctxt* ⟹ *D* ≤ *C* ⟷ *C* ⊔ *D* = *C*
⟨*proof*⟩

lemma *comp-mctxt-mctxt-of-term1*[*dest*!]:
(*C*, *mctxt-of-term t*) ∈ *comp-mctxt* ⟹ *C* ≤ *mctxt-of-term t*
⟨*proof*⟩

lemmas *comp-mctxt-mctxt-of-term2*[*dest*!] = *comp-mctxt-mctxt-of-term1*[*OF comp-mctxt-sym*]

lemma *mfun-leq-mfunI*:

$f = g \Longrightarrow length\ Cs = length\ Ds \Longrightarrow (\bigwedge i.\ i < length\ Ds \Longrightarrow Cs\ !\ i \leq Ds\ !\ i) \Longrightarrow MFun\ f\ Cs \leq MFun\ g\ Ds$
⟨*proof*⟩

lemma *prefix-mctxt-sup*:
assumes $C \leq (E :: ('f, 'v)\ mctxt)\ D \leq E$ **shows** $C \sqcup D \leq E$
⟨*proof*⟩

lemma *mreplace-at-leqI*:
$p \in all\text{-}poss\text{-}mctxt\ C \Longrightarrow C \leq E \Longrightarrow D \leq subm\text{-}at\ E\ p \Longrightarrow mreplace\text{-}at\ C\ p\ D \leq E$
⟨*proof*⟩

lemma *prefix-and-fewer-holes-implies-equal-mctxt*:
$C \leq D \Longrightarrow hole\text{-}poss\ C \subseteq hole\text{-}poss\ D \Longrightarrow C = D$
⟨*proof*⟩

lemma *compare-mreplace-at*:
$p \in poss\text{-}mctxt\ C \Longrightarrow mreplace\text{-}at\ C\ p\ D \leq mreplace\text{-}at\ C\ p\ E \longleftrightarrow D \leq E$
⟨*proof*⟩

lemma *merge-mreplace-at*:
$p \in poss\text{-}mctxt\ C \Longrightarrow mreplace\text{-}at\ C\ p\ (D \sqcup E) = \ mreplace\text{-}at\ C\ p\ D \sqcup mreplace\text{-}at\ C\ p\ E$
⟨*proof*⟩

lemma *compare-mreplace-atI'*:
$C \leq D \Longrightarrow C' \leq D' \Longrightarrow p \in all\text{-}poss\text{-}mctxt\ C \Longrightarrow mreplace\text{-}at\ C\ p\ C' \leq mreplace\text{-}at\ D\ p\ D'$
⟨*proof*⟩

lemma *compare-mreplace-atI*:
$C \leq D \Longrightarrow C' \leq D' \Longrightarrow p \in poss\text{-}mctxt\ C \Longrightarrow mreplace\text{-}at\ C\ p\ C' \leq mreplace\text{-}at\ D\ p\ D'$
⟨*proof*⟩

lemma *all-poss-mctxt-mono*:
$C \leq D \Longrightarrow all\text{-}poss\text{-}mctxt\ C \subseteq all\text{-}poss\text{-}mctxt\ D$
⟨*proof*⟩

lemma *all-poss-mctxt-inf-mctxt*:
$(C, D) \in comp\text{-}mctxt \Longrightarrow all\text{-}poss\text{-}mctxt\ (C \sqcap D) = all\text{-}poss\text{-}mctxt\ C \cap all\text{-}poss\text{-}mctxt\ D$
⟨*proof*⟩

lemma *less-eq-subm-at*:
$p \in all\text{-}poss\text{-}mctxt\ C \Longrightarrow C \leq D \Longrightarrow subm\text{-}at\ C\ p \leq subm\text{-}at\ D\ p$

⟨*proof*⟩

lemma *inf-subm-at*:
p ∈ *all-poss-mctxt* (*C* ⊓ *D*) ⟹ *subm-at* (*C* ⊓ *D*) *p* = *subm-at C p* ⊓ *subm-at D p*
⟨*proof*⟩

lemma *less-eq-fill-holesI*:
assumes *length Ds* = *num-holes C length Es* = *num-holes C*
⋀*i*. *i* < *num-holes C* ⟹ *Ds* ! *i* ≤ *Es* ! *i*
shows *fill-holes-mctxt C Ds* ≤ *fill-holes-mctxt C Es*
⟨*proof*⟩

lemma *less-eq-fill-holesD*:
assumes *length Ds* = *num-holes C length Es* = *num-holes C*
fill-holes-mctxt C Ds ≤ *fill-holes-mctxt C Es i* < *num-holes C*
shows *Ds* ! *i* ≤ *Es* ! *i*
⟨*proof*⟩

lemma *less-eq-fill-holes-iff*:
assumes *length Ds* = *num-holes C length Es* = *num-holes C*
shows *fill-holes-mctxt C Ds* ≤ *fill-holes-mctxt C Es* ⟷ (∀ *i* < *num-holes C*. *Ds* ! *i* ≤ *Es* ! *i*)
⟨*proof*⟩

lemma *fill-holes-mctxt-suffix*[*simp*]:
assumes *length Ds* = *num-holes C* **shows** *C* ≤ *fill-holes-mctxt C Ds*
⟨*proof*⟩

lemma *fill-holes-mctxt-id*:
assumes *length Ds* = *num-holes C C* = *fill-holes-mctxt C Ds* **shows** *set Ds* ⊆ {*MHole*}
⟨*proof*⟩

lemma *fill-holes-suffix*[*simp*]:
num-holes C = *length ts* ⟹ *C* ≤ *mctxt-of-term* (*fill-holes C ts*)
⟨*proof*⟩

6.1.6 An inverse of *fill-holes*

fun *unfill-holes* :: ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term list* **where**
unfill-holes MHole t = [*t*]
| *unfill-holes* (*MVar w*) (*Var v*) = (*if v* = *w then* [] *else undefined*)
| *unfill-holes* (*MFun g Cs*) (*Fun f ts*) = (*if f* = *g* ∧ *length ts* = *length Cs then*
concat (*map* (λ*i*. *unfill-holes* (*Cs* ! *i*) (*ts* ! *i*)) [*0*..<*length ts*]) *else undefined*)

lemma *length-unfill-holes*[*simp*]:
assumes *C* ≤ *mctxt-of-term t*
shows *length* (*unfill-holes C t*) = *num-holes C*

⟨*proof*⟩

lemma *fill-unfill-holes*:
assumes $C \leq$ *mctxt-of-term* t
shows *fill-holes* C (*unfill-holes* C t) $= t$
⟨*proof*⟩

lemma *unfill-fill-holes*:
assumes *length* $ts =$ *num-holes* C
shows *unfill-holes* C (*fill-holes* C ts) $= ts$
⟨*proof*⟩

lemma *unfill-holes-subt*:
assumes $C \leq$ *mctxt-of-term* t **and** $t' \in$ *set* (*unfill-holes* C t)
shows $t' \trianglelefteq t$
⟨*proof*⟩

lemma *factor-hole-pos-by-prefix*:
assumes $C \leq D$ $p \in$ *hole-poss* D
obtains q **where** $q \leq_p p$ $q \in$ *hole-poss* C
⟨*proof*⟩

lemma *concat-map-zip-upt*: **assumes** $\bigwedge i.\ i < n \Longrightarrow$ *length* (*f* i) $=$ *length* (*g* i)
shows *concat* (*map* ($\lambda i.$ *zip* (*f* i) (*g* i)) [0..<*n*]) $=$ *zip* (*concat* (*map* *f* [0..<*n*])) (*concat* (*map* *g* [0..<*n*]))
⟨*proof*⟩

lemma *unfill-holes-by-prefix′*:
assumes *num-holes* $C =$ *length* *Ds* *fill-holes-mctxt* C *Ds* $\leq$ *mctxt-of-term* t
shows *unfill-holes* (*fill-holes-mctxt* C *Ds*) $t =$ *concat* (*map* (λ(*D*, *t*). *unfill-holes* *D* *t*) (*zip* *Ds* (*unfill-holes* C t)))
⟨*proof*⟩

lemma *unfill-holes-var-subst*:
$C \leq$ *mctxt-of-term* $t \Longrightarrow$ *unfill-holes* (*map-vars-mctxt* *f* C) ($t \cdot$ (*Var* $\circ$ *f*)) $=$ *map* ($\lambda t.\ t \cdot$ (*Var* $\circ$ *f*)) (*unfill-holes* C t)
⟨*proof*⟩

6.1.7 Ditto for *fill-holes-mctxt*

fun *unfill-holes-mctxt* :: ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *mctxt list* **where**
unfill-holes-mctxt MHole D = [*D*]
| *unfill-holes-mctxt* (*MVar w*) (*MVar v*) = (*if v* = *w then* [] *else undefined*)
| *unfill-holes-mctxt* (*MFun g Cs*) (*MFun f Ds*) = (*if f* = *g* ∧ *length Ds* = *length Cs then*
concat (*map* (λ*i*. *unfill-holes-mctxt* (*Cs* ! *i*) (*Ds* ! *i*)) [0..<*length Ds*]) *else undefined*)

lemma *length-unfill-holes-mctxt* [*simp*]:

assumes $C \leq D$
shows *length* (*unfill-holes-mctxt* C D) = *num-holes* C
⟨*proof*⟩

lemma *fill-unfill-holes-mctxt*:
assumes $C \leq D$
shows *fill-holes-mctxt* C (*unfill-holes-mctxt* C D) = D
⟨*proof*⟩

lemma *unfill-fill-holes-mctxt*:
assumes *length* Ds = *num-holes* C
shows *unfill-holes-mctxt* C (*fill-holes-mctxt* C Ds) = Ds
⟨*proof*⟩

lemma *unfill-holes-mctxt-mctxt-of-term*:
assumes $C \leq$ *mctxt-of-term* t
shows *unfill-holes-mctxt* C (*mctxt-of-term* t) = *map mctxt-of-term* (*unfill-holes* C t)
⟨*proof*⟩

6.1.8 Function symbols of prefixes

lemma *funas-prefix*[*simp*]:
$C \leq D \Longrightarrow fn \in$ *funas-mctxt* $C \Longrightarrow fn \in$ *funas-mctxt* D
⟨*proof*⟩

end

6.2 The Parallel Rewrite Relation

theory *Parallel-Rewriting*
imports
Trs
Multihole-Context
begin

The parallel rewrite relation as inductive definition

inductive-set *par-rstep* :: ('*f*,'*v*)*trs* ⇒ ('*f*,'*v*)*trs* **for** R :: ('*f*,'*v*)*trs*
where *root-step*[*intro*]: (s,t) ∈ $R \Longrightarrow$ ($s \cdot \sigma$,$t \cdot \sigma$) ∈ *par-rstep* R
| *par-step-fun*[*intro*]: ⟦⋀ i. $i <$ *length* $ts \Longrightarrow$ (ss ! i,ts ! i) ∈ *par-rstep* R⟧ ⟹ *length* ss = *length* ts
⟹ (*Fun* f ss, *Fun* f ts) ∈ *par-rstep* R
| *par-step-var*[*intro*]: (*Var* x, *Var* x) ∈ *par-rstep* R

lemma *par-rstep-refl*[*intro*]: (t,t) ∈ *par-rstep* R
⟨*proof*⟩

lemma *all-ctxt-closed-par-rstep*[*intro*]: *all-ctxt-closed* F (*par-rstep* R)
⟨*proof*⟩

lemma *args-par-rstep-pow-imp-par-rstep-pow*:
  length xs = *length ys* $\Longrightarrow \forall$ *i*<*length xs*. (*xs* ! *i*, *ys* ! *i*) $\in$ *par-rstep R* $\widehat{\ }\widehat{\ }$ *n* $\Longrightarrow$
  (*Fun f xs*, *Fun f ys*) $\in$ *par-rstep R* $\widehat{\ }\widehat{\ }$ *n*
⟨*proof*⟩

lemma *ctxt-closed-par-rstep*[*intro*]: *ctxt.closed* (*par-rstep R*)
⟨*proof*⟩

lemma *subst-closed-par-rstep*: (*s*,*t*) $\in$ *par-rstep R* $\Longrightarrow$ (*s* $\cdot$ σ, *t* $\cdot$ σ) $\in$ *par-rstep R*
⟨*proof*⟩

lemma *R-par-rstep*: *R* $\subseteq$ *par-rstep R*
  ⟨*proof*⟩

lemma *par-rstep-rsteps*: *par-rstep R* $\subseteq$ (*rstep R*)*
⟨*proof*⟩

lemma *rstep-par-rstep*: *rstep R* $\subseteq$ *par-rstep R*
  ⟨*proof*⟩

lemma *par-rsteps-rsteps*: (*par-rstep R*)* = (*rstep R*)* (**is** *?P* = *?R*)
⟨*proof*⟩

lemma *par-rsteps-union*: (*par-rstep A* $\cup$ *par-rstep B*)* =
    (*rstep* (*A* $\cup$ *B*))*
⟨*proof*⟩

lemma *par-rstep-inverse*: *par-rstep* (*R*$\widehat{\ }$−*1*) = (*par-rstep R*)$\widehat{\ }$−*1*
⟨*proof*⟩

lemma *par-rstep-conversion*: (*rstep R*)$^{\leftrightarrow *}$ = (*par-rstep R*)$^{\leftrightarrow *}$
  ⟨*proof*⟩

lemma *par-rstep-mono*: **assumes** *R* $\subseteq$ *S*
  shows *par-rstep R* $\subseteq$ *par-rstep S*
⟨*proof*⟩

lemma *wf-trs-par-rstep*: **assumes** *wf*: $\bigwedge$ *l r*. (*l*,*r*) $\in$ *R* $\Longrightarrow$ *is-Fun l*
  and *step*: (*Var x*, *t*) $\in$ *par-rstep R*
shows *t* = *Var x*
  ⟨*proof*⟩

main lemma which tells us, that either a parallel rewrite step of $l \cdot \sigma$ is inside l, or we can do the step completely inside σ

lemma *par-rstep-linear-subst*: **assumes** *lin*: *linear-term l*
  and *step*: (*l* $\cdot$ σ, *t*) $\in$ *par-rstep R*
shows ($\exists$ τ. *t* = *l* $\cdot$ τ $\wedge$ ($\forall$ *x* $\in$ *vars-term l*. (σ *x*, τ *x*) $\in$ *par-rstep R*) $\vee$
        ($\exists$ *C l*'' *l*' *r*'. *l* = *C*⟨*l*''⟩ $\wedge$ *is-Fun l*'' $\wedge$ (*l*',*r*') $\in$ *R* $\wedge$ (*l*'' $\cdot$ σ = *l*' $\cdot$ τ) $\wedge$

$((C \cdot_c \sigma) \langle\ r' \cdot \tau\ \rangle,\ t) \in$ *par-rstep R*))
⟨*proof*⟩

lemma *par-rstep-id*:
$(s,\ t) \in R \Longrightarrow (s,\ t) \in$ *par-rstep R*
⟨*proof*⟩

6.3 Parallel Rewriting using Multihole Contexts

datatype $('f,'v)$*par-info* = *Par-Info*
(*par-left*: $('f,'v)$*term*)
(*par-right*: $('f,'v)$*term*)
(*par-rule*: $('f,'v)$*rule*)

abbreviation *par-lefts* **where** *par-lefts* ≡ *map par-left*
abbreviation *par-rights* **where** *par-rights* ≡ *map par-right*
abbreviation *par-rules* **where** *par-rules* ≡ (λ *info*. *par-rule* ‘ *set info*)

definition *par-cond* :: $('f,'v)$*trs* ⇒ $('f,'v)$*par-info* ⇒ *bool* **where**
par-cond R info = (*par-rule info* ∈ *R* ∧ (*par-left info*, *par-right info*) ∈ *rrstep* {*par-rule info*})

abbreviation *par-conds* **where** *par-conds R* ≡ λ *infos*. *Ball* (*set infos*) (*par-cond R*)

lemma *par-cond-imp-rrstep*: **assumes** *par-cond R info*
shows (*par-left info*, *par-right info*) ∈ *rrstep R*
⟨*proof*⟩

lemma *par-conds-imp-rrstep*: **assumes** *par-conds R infos*
and *s* = *par-lefts infos* ! *i t* = *par-rights infos* ! *i*
and *i* < *length infos*
shows (*s*, *t*) ∈ *rrstep R*
⟨*proof*⟩

definition *par-rstep-mctxt* **where**
par-rstep-mctxt R C infos = {(*s*, *t*). *s* =$_f$ (*C*, *par-lefts infos*) ∧ *t* =$_f$ (*C*, *par-rights infos*) ∧ *par-conds R infos*}

lemma *par-rstep-mctxtI*: **assumes** *s* =$_f$ (*C*, *par-lefts infos*) *t* =$_f$ (*C*, *par-rights infos*) *par-conds R infos*
shows (*s*,*t*) ∈ *par-rstep-mctxt R C infos*
⟨*proof*⟩

lemma *par-rstep-mctxt-reflI*: (*s*,*s*) ∈ *par-rstep-mctxt R* (*mctxt-of-term s*) []
⟨*proof*⟩

lemma *par-rstep-mctxt-varI*: (*Var x*, *Var x*) ∈ *par-rstep-mctxt R* (*MVar x*) []
⟨*proof*⟩

lemma *par-rstep-mctxt-MHoleI*: $(l,r) \in R \Longrightarrow s = l \cdot \sigma \Longrightarrow t = r \cdot \sigma \Longrightarrow$ *infos* = [*Par-Info s t* (l,r)]
$\Longrightarrow (s,t) \in$ *par-rstep-mctxt R MHole infos*
⟨*proof*⟩

lemma *par-rstep-mctxt-funI*:
assumes *rec*: $\bigwedge$ *i*. *i* < *length ts* $\Longrightarrow$ (*ss* ! *i*, *ts* ! *i*) ∈ *par-rstep-mctxt R* (*Cs* ! *i*) (*infos* ! *i*)
and *len*: *length ss* = *length ts length Cs* = *length ts length infos* = *length ts*
shows (*Fun f ss*, *Fun f ts*) ∈ *par-rstep-mctxt R* (*MFun f Cs*) (*concat infos*)
⟨*proof*⟩

lemma *par-rstep-mctxt-funI-ex*:
assumes $\bigwedge$ *i*. *i* < *length ts* $\Longrightarrow \exists$ *C infos*. (*ss* ! *i*, *ts* ! *i*) ∈ *par-rstep-mctxt R C infos*
and *length ss* = *length ts*
shows $\exists$ *C infos*. (*Fun f ss*, *Fun f ts*) ∈ *par-rstep-mctxt R C infos* ∧ *C* ≠ *MHole*

⟨*proof*⟩

Parallel rewriting is closed under multihole-contexts.

lemma *par-rstep-mctxt*:
assumes $s =_f (C, ss)$ **and** $t =_f (C, ts)$
and ∀ *i*<*length ss*. (*ss* ! *i*, *ts* ! *i*) ∈ *par-rstep R*
shows (*s*, *t*) ∈ *par-rstep R*
⟨*proof*⟩

lemma *par-rstep-mctxt-rrstepI* :
assumes $s =_f (C, ss)$ **and** $t =_f (C, ts)$
and ∀ *i*<*length ss*. (*ss* ! *i*, *ts* ! *i*) ∈ *rrstep R*
shows (*s*, *t*) ∈ *par-rstep R*
⟨*proof*⟩

lemma *par-rstep-mctxtD*:
assumes (*s*, *t*) ∈ *par-rstep R*
shows $\exists$ *C ss ts*. $s =_f (C, ss) \wedge t =_f (C, ts) \wedge$ (∀ *i*<*length ss*. (*ss* ! *i*, *ts* ! *i*) ∈ *rrstep R*)
(**is** $\exists$ *C ss ts*. *?P s t C ss ts*)
⟨*proof*⟩

lemma *par-rstep-mctxt-mono*: **assumes** $R \subseteq S$
shows *par-rstep-mctxt R C infos* ⊆ *par-rstep-mctxt S C infos*
⟨*proof*⟩

lemma *par-rstep-mctxtE*:
assumes (*s*, *t*) ∈ *par-rstep R*

obtains *C infos* **where** $s =_f$ (*C*, *par-lefts infos*) **and** $t =_f$ (*C*, *par-rights infos*)
and *par-conds R infos*
⟨*proof*⟩

lemma *par-rstep-par-rstep-mctxt-conv*:
(*s*, *t*) ∈ *par-rstep R* ⟷ (∃ *C infos*. (*s*, *t*) ∈ *par-rstep-mctxt R C infos*)
⟨*proof*⟩

fun *subst-apply-par-info* :: ('*f*,'*v*)*par-info* ⇒ ('*f*,'*v*)*subst* ⇒ ('*f*,'*v*)*par-info* (**infixl** ·*pi* 67) **where**
Par-Info s t r ·*pi* σ = *Par-Info* (*s* · σ) (*t* · σ) *r*

lemma *subst-apply-par-info-simps*[*simp*]:
par-left (*info* ·*pi* σ) = *par-left info* · σ
par-right (*info* ·*pi* σ) = *par-right info* · σ
par-rule (*info* ·*pi* σ) = *par-rule info*
par-cond R info ⟹ *par-cond R* (*info* ·*pi* σ)
⟨*proof*⟩

lemma *par-rstep-mctxt-subst*: **assumes** (*s*,*t*) ∈ *par-rstep-mctxt R C infos*
shows (*s* · σ, *t* · σ) ∈ *par-rstep-mctxt R* (*C* ·*mc* σ) (*map* (λ *i*. *i* ·*pi* σ) *infos*)
⟨*proof*⟩

lemma *par-rstep-mctxt-MVarE*:
assumes (*s*,*t*) ∈ *par-rstep-mctxt R* (*MVar x*) *infos*
shows *s* = *Var x t* = *Var x infos* = []
⟨*proof*⟩

lemma *par-rstep-mctxt-MHoleE*:
assumes (*s*,*t*) ∈ *par-rstep-mctxt R MHole infos*
obtains *info* **where**
par-left info = *s*
par-right info = *t*
infos = [*info*]
(*s*, *t*) ∈ *rrstep R*
par-cond R info
⟨*proof*⟩

lemma *par-rstep-mctxt-MFunD*:
assumes (*s*,*t*) ∈ *par-rstep-mctxt R* (*MFun f Cs*) *infos*
shows ∃ *ss ts Infos*.
s = *Fun f ss* ∧
t = *Fun f ts* ∧
length ss = *length Cs* ∧
length ts = *length Cs* ∧
length Infos = *length Cs* ∧
infos = *concat Infos* ∧
(∀ *i* < *length Cs*. (*ss* ! *i*, *ts* ! *i*) ∈ *par-rstep-mctxt R* (*Cs* ! *i*) (*Infos* ! *i*))
⟨*proof*⟩

6.4 Variable Restricted Parallel Rewriting

fun *vars-below-hole* :: ('*f*,'*v*)*term* ⇒ ('*f*,'*v*)*mctxt* ⇒ '*v set* **where**
vars-below-hole t MHole = *vars-term t*
| *vars-below-hole t* (*MVar y*) = {}
| *vars-below-hole* (*Fun - ts*) (*MFun - Cs*) =
⋃ (*set* (*map* (λ (*t*,*C*). *vars-below-hole t C*) (*zip ts Cs*)))
| *vars-below-hole* (*Var* -) (*MFun - -*) = *Code.abort* (*STR* ''*assumption in vars-below-hole violated*'') (λ -. {})

lemma *vars-below-hole-no-hole*: *hole-poss C* = {} ⟹ *vars-below-hole t C* = {}
⟨*proof*⟩

lemma *vars-below-hole-mctxt-of-term*[*simp*]: *vars-below-hole t* (*mctxt-of-term u*) = {}
⟨*proof*⟩

lemma *vars-below-hole-vars-term*: *vars-below-hole t C* ⊆ *vars-term t*
⟨*proof*⟩

lemma *vars-below-hole-subst*[*simp*]: *vars-below-hole t* (*C* ·*mc* σ) = *vars-below-hole t C*
⟨*proof*⟩

lemma *vars-below-hole-Fun*: **assumes** *length ls* = *length Cs*
shows *vars-below-hole* (*Fun f ls*) (*MFun f Cs*) = ⋃ {*vars-below-hole* (*ls* ! *i*) (*Cs* ! *i*) | *i*. *i* < *length Cs*}
⟨*proof*⟩

lemma *vars-below-hole-term-subst*:
hole-poss D ⊆ *poss t* ⟹ *vars-below-hole* (*t* · σ) *D* = ⋃ (*vars-term* ‘ σ ‘ *vars-below-hole t D*)
⟨*proof*⟩

lemma *vars-below-hole-eqf*: **assumes** *t* =*f* (*C*, *ts*)
shows *vars-below-hole t C* = ⋃ (*vars-term* ‘ *set ts*)
⟨*proof*⟩

definition *par-rstep-var-restr R V* = {(*s*,*t*) | *s t C infos*.
(*s*, *t*) ∈ *par-rstep-mctxt R C infos* ∧ *vars-below-hole t C* ∩ *V* = {}}

lemma *par-rstep-var-restr-mono*: **assumes** *R* ⊆ *S W* ⊆ *V*
shows *par-rstep-var-restr R V* ⊆ *par-rstep-var-restr S W*
⟨*proof*⟩

lemma *par-rstep-var-restr-refl*[*simp*]: (*t*, *t*) ∈ *par-rstep-var-restr R V*
⟨*proof*⟩

the most important property: a substitution step and a parallel step can be merged into a single parallel step

lemma *merge-par-rstep-var-restr*:
  assumes *subst-R*: $\bigwedge x.\ (\delta\ x,\ \gamma\ x) \in$ *par-rstep R*
    and *st*: $(s,\ t) \in$ *par-rstep-var-restr R V*
    and *subst-eq*: $\bigwedge x.\ x \notin V \Longrightarrow \delta\ x = \gamma\ x$
  shows $(s \cdot \delta,\ t \cdot \gamma) \in$ *par-rstep R*
⟨*proof*⟩

the variable restricted parallel rewrite relation is closed under variable renamings, provided that the set of forbidden variables is also renamed (in the inverse way)

lemma *par-rstep-var-restr-subst*:
  assumes $(s,t) \in$ *par-rstep-var-restr R* $(\gamma\ `\ V)$
    and $\bigwedge x.\ \sigma\ x \cdot (Var\ o\ \gamma) = Var\ x$
  shows $(s \cdot \sigma,\ t \cdot \sigma) \in$ *par-rstep-var-restr R V*
⟨*proof*⟩

end

7 Orthogonality

theory *Orthogonality*
  imports
    Critical-Pairs
    Parallel-Rewriting
begin

This theory contains the result, that weak orthogonality implies confluence.

We prove the diamond property of *par-rstep* for weakly orthogonal systems.

context
  fixes *ren* :: $'v$ :: *infinite renaming2*
begin
lemma *weakly-orthogonal-main*: **fixes** *R* :: $('f,'v)trs$
  assumes *st1*: $(s,t1) \in$ *par-rstep R* **and** *st2*: $(s,t2) \in$ *par-rstep R* **and** *weak-ortho*:
    left-linear-trs R $\bigwedge b\ l\ r.\ (b,l,r) \in$ *critical-pairs ren R R* $\Longrightarrow l = r$
    and *wf*: $\bigwedge l\ r.\ (l,r) \in R \Longrightarrow$ *is-Fun l*
  shows $\exists\ u.\ (t1,u) \in$ *par-rstep R* $\wedge\ (t2,u) \in$ *par-rstep R*
⟨*proof*⟩

lemma *weakly-orthogonal-par-rstep-CR*:
  assumes *weak-ortho*: *left-linear-trs R* $\bigwedge b\ l\ r.\ (b,l,r) \in$ *critical-pairs ren R R* $\Longrightarrow l = r$
    and *wf*: $\bigwedge l\ r.\ (l,r) \in R \Longrightarrow$ *is-Fun l*
  shows *CR* (*par-rstep R*)
⟨*proof*⟩

lemma *weakly-orthogonal-rstep-CR*:
assumes *weak-ortho*: *left-linear-trs R* $\bigwedge$ *b l r*. (*b*,*l*,*r*) $\in$ *critical-pairs ren R R* $\Longrightarrow$ *l* = *r*
and *wf*: $\bigwedge$ *l r*. (*l*,*r*) $\in$ *R* $\Longrightarrow$ *is-Fun l*
shows *CR* (*rstep R*)
⟨*proof*⟩

end
end

8 Multi-Step Rewriting

theory *Multistep*
imports *Trs*
begin

Multi-step rewriting (without proof terms).

inductive-set
mstep :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *term rel*
for *R*
where
Var: (*Var x*, *Var x*) $\in$ *mstep R* |
args: $\bigwedge$*f n ss ts*. ⟦*length ss* = *n*; *length ts* = *n*; ∀ *i*<*n*. (*ss* ! *i*, *ts* ! *i*) $\in$ *mstep R*⟧ $\Longrightarrow$ (*Fun f ss*, *Fun f ts*) $\in$ *mstep R* |
rule: $\bigwedge$*l r* σ τ. ⟦(*l*, *r*) $\in$ *R*; ∀ *x*∈*vars-term l*. (σ *x*, τ *x*) $\in$ *mstep R*⟧ $\Longrightarrow$ (*l* · σ, *r* · τ) $\in$ *mstep R*

lemma *mstep-refl* [*simp*]:
(*t*, *t*) $\in$ *mstep R*
⟨*proof*⟩

lemma *mstep-ctxt*:
assumes (*s*, *t*) $\in$ *mstep R*
shows (*C*⟨*s*⟩, *C*⟨*t*⟩) $\in$ *mstep R*
⟨*proof*⟩

lemma *rstep-imp-mstep*:
assumes (*s*, *t*) $\in$ *rstep R*
shows (*s*, *t*) $\in$ *mstep R*
⟨*proof*⟩

lemma *rstep-mstep-subset*:
rstep R $\subseteq$ *mstep R*
⟨*proof*⟩

lemma *subst-rsteps-imp-rule-rsteps*:
assumes ∀ *x*∈*vars-term l*. (σ *x*, τ *x*) $\in$ (*rstep R*)*

and $(l, r) \in R$
shows $(l \cdot \sigma, r \cdot \tau) \in (rstep\ R)^*$
⟨*proof*⟩

lemma *mstep-imp-rsteps*:
assumes $(s, t) \in mstep\ R$
shows $(s, t) \in (rstep\ R)^*$
⟨*proof*⟩

lemma *mstep-rsteps-subset*:
shows $mstep\ R \subseteq (rstep\ R)^*$
⟨*proof*⟩

lemma *mstep-mono*: $R \subseteq S \Longrightarrow mstep\ R \subseteq mstep\ S$
⟨*proof*⟩

Thus if *mstep R* has the diamond property, then *rstep R* is confluent.

lemma *Var-mstep*:
assumes $*$: $\bigwedge l\ r.\ (l, r) \in R \Longrightarrow \neg\ is\text{-}Var\ l$
and $(Var\ x, t) \in mstep\ R$
shows $t = Var\ x$
⟨*proof*⟩

8.1 Maximal multi-step rewriting.

inductive-set
mmstep :: $('f, 'v)\ trs \Rightarrow ('f, 'v)\ term\ rel$
for R
where
Var: $(Var\ x, Var\ x) \in mmstep\ R$ |
args: $\bigwedge f\ n\ ss\ ts.\ [\![length\ ss = n; length\ ts = n;$
$\neg\ (\exists\,(l, r) \in R.\ \exists\,\sigma.\ Fun\ f\ ss = l \cdot \sigma);$
$\forall\, i < n.\ (ss\ !\ i, ts\ !\ i) \in mmstep\ R]\!] \Longrightarrow$
$(Fun\ f\ ss, Fun\ f\ ts) \in mmstep\ R$ |
rule: $\bigwedge l\ r\ \sigma\ \tau.\ [\![(l, r) \in R; \forall\, x \in vars\text{-}term\ l.\ (\sigma\ x, \tau\ x) \in mmstep\ R]\!] \Longrightarrow$
$(l \cdot \sigma, r \cdot \tau) \in mmstep\ R$

lemma *mmstep-imp-mstep*:
assumes $(s, t) \in mmstep\ R$
shows $(s, t) \in mstep\ R$
⟨*proof*⟩

lemma *mmstep-mstep-subset*:
$mmstep\ R \subseteq mstep\ R$
⟨*proof*⟩

end

9 Implementation of First Order Rewriting

theory *Trs-Impl*
imports
Trs
First-Order-Rewriting.Term-Impl
First-Order-Terms.Matching
First-Order-Rewriting.Abstract-Rewriting-Impl
Option-Util
Transitive−Closure.RBT-Map-Set-Extension
begin

9.1 Implementation of the Rewrite Relation

9.1.1 Generate All Rewrites

type-synonym $('f, 'v)$ *rules* = $('f, 'v)$ *rule list*

context fixes $R :: ('f,'v)$*rules*
begin

definition *rrewrite* :: $('f, 'v)$ *term* ⇒ $('f, 'v)$ *term list*
where
rrewrite s = *List.maps* (λ (*l*, *r*) . *case match s l of*
None ⇒ []
| *Some* σ ⇒ [*r* · σ]) *R*

lemma *rrewrite-sound*: *t* ∈ *set* (*rrewrite s*) ⟹ (*s*,*t*) ∈ *rrstep* (*set R*)
⟨*proof*⟩

lemma *rrewrite-complete*: **assumes** (*s*,*t*) ∈ *rrstep* (*set R*)
shows ∃ *u*. *u* ∈ *set* (*rrewrite s*)
⟨*proof*⟩

lemma *rrewrite*: **assumes** ⋀ *l r*. (*l*,*r*) ∈ *set R* ⟹ *vars-term l* ⊇ *vars-term r*
shows *set* (*rrewrite s*) = {*t*. (*s*,*t*) ∈ *rrstep* (*set R*)}
⟨*proof*⟩

fun *rewrite* :: $('f, 'v)$ *term* ⇒ $('f, 'v)$ *term list* **where**
rewrite s = (*rrewrite s* @ (*case s of Var* - ⇒ [] | *Fun f ss* ⇒
concat (*map* (λ (*i*, *si*). *map* (λ *ti*. *Fun f* (*ss*[*i* := *ti*])) (*rewrite si*))
(*zip* [*0*..< *length ss*] *ss*))))

declare *rewrite.simps*[*simp del*]

lemma *rewrite-sound*: *t* ∈ *set* (*rewrite s*) ⟹ (*s*,*t*) ∈ *rstep* (*set R*)
⟨*proof*⟩

lemma *rewrite*: **assumes** ⋀ *l r*. (*l*,*r*) ∈ *set R* ⟹ *vars-term l* ⊇ *vars-term r*

shows *set* (*rewrite s*) = {*t*. (*s*,*t*) ∈ *rstep* (*set R*)}
⟨*proof*⟩

lemma *rewrite-complete*: **assumes** (*s*,*t*) ∈ *rstep* (*set R*)
shows ∃ *w*. *w* ∈ *set* (*rewrite s*)
⟨*proof*⟩
end

lemma *rrewrite-mono*: *set R* ⊆ *set S* ⟹ *set* (*rrewrite R s*) ⊆ *set* (*rrewrite S s*)
⟨*proof*⟩

lemma *Union-image-mono*: (⋀ *x*. *x* ∈ *A* ⟹ *f x* ⊆ *g x*) ⟹ ⋃ (*f* ‘ *A*) ⊆ ⋃ (*g* ‘ *A*)
⟨*proof*⟩

lemma *rewrite-mono*: **assumes** *set R* ⊆ *set S*
shows *set* (*rewrite R s*) ⊆ *set* (*rewrite S s*)
⟨*proof*⟩

definition *first-rewrite* :: ('*f*,'*v*)*rules* ⇒ ('*f*,'*v*)*term* ⇒ ('*f*,'*v*)*term option*
where *first-rewrite R s* ≡ *case rewrite R s of Nil* ⇒ *None* | *Cons t* - ⇒ *Some t*

9.1.2 Checking a Single Rewrite Step

definition *is-root-step* :: ('*f*, '*v*)*trs* ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term* ⇒ *bool*
where
is-root-step R s t = (∃ (*l*, *r*) ∈ *R*. *case match-list Var* [(*l*,*s*),(*r*,*t*)] *of*
None ⇒ *False*
| *Some* - ⇒ *True*)

lemma *rrstep-code*[*code-unfold*]: (*s*,*t*) ∈ *rrstep R* ⟷ *is-root-step R s t*
⟨*proof*⟩

lemma *is-root-step*: *is-root-step R s t* ⟹ (*s*, *t*) ∈ *rrstep R*
⟨*proof*⟩

fun *is-rstep* :: ('*f*,'*v*)*trs* ⇒ ('*f*,'*v*)*term* ⇒ ('*f*,'*v*)*term* ⇒ *bool* **where**
is-rstep R (*Fun f ts*) (*Fun g ss*) = (
f = *g* ∧ *length ts* = *length ss* ∧ (∃ *i* ∈ *set* [*0*..<*length ss*].
ss = *ts*[*i* := *ss* ! *i*] ∧ *is-rstep R* (*ts* ! *i*) (*ss* ! *i*))
∨ (*Fun f ts*, *Fun g ss*) ∈ *rrstep R*)
| *is-rstep R s t* = ((*s*,*t*) ∈ *rrstep R*)

lemma *is-rstep-sound*: *is-rstep R s t* ⟹ (*s*,*t*) ∈ *rstep R*
⟨*proof*⟩

lemma *is-rstep-complete*: **assumes** (*s*,*t*) ∈ *rstep R*

shows *is-rstep R s t*
⟨*proof*⟩

lemma *is-rstep*[*simp*]: *is-rstep R s t* ⟷ (*s*,*t*) ∈ *rstep R*
⟨*proof*⟩

lemma *in-rstep-code*[*code-unfold*]:
st ∈ *rstep R* ⟷ (*case st of* (*s*,*t*) ⇒ *is-rstep R s t*)
⟨*proof*⟩

9.2 Computation of a Normal Form

definition *compute-rstep-NF* :: ('*f*,'*v*)*rules* ⇒ ('*f*,'*v*)*term* ⇒ ('*f*,'*v*)*term option*
where *compute-rstep-NF R s* ≡ *compute-NF* (*first-rewrite R*) *s*

lemma *compute-rstep-NF-sound*:
assumes *res*: *compute-rstep-NF R s* = *Some t*
shows (*s*, *t*) ∈ (*rstep* (*set R*))^* ⟨*proof*⟩

lemma *compute-rstep-NF-complete*: **assumes** *res*: *compute-rstep-NF R s* = *Some t*
shows *t* ∈ *NF* (*rstep* (*set R*)) ⟨*proof*⟩

lemma *compute-rstep-NF-SN*: **assumes** *SN*: *SN* (*rstep* (*set R*))
shows ∃ *t*. *compute-rstep-NF R s* = *Some t*
⟨*proof*⟩

9.2.1 Computing Reachable Terms with Limit on Derivation Length

fun *reachable-terms* ::
('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *term* ⇒ *nat* ⇒ ('*f*, '*v*) *term list*
where
reachable-terms R s 0 = [*s*]
| *reachable-terms R s* (*Suc n*) = (
let ts = (*reachable-terms R s n*) *in*
remdups (*ts*@(*concat* (*map* (λ *t*. *rewrite R t*) *ts*)))
)

lemma *reachable-terms-nat*:
assumes *t* ∈ *set* (*reachable-terms R s i*)
shows ∃ *j*. *j* ≤ *i* ∧ (*s*,*t*) ∈ (*rstep* (*set R*))^^*j*
⟨*proof*⟩

lemma *reachable-terms*:
assumes *t* ∈ *set* (*reachable-terms R s i*)
shows (*s*,*t*) ∈ (*rstep* (*set R*))^*
⟨*proof*⟩

lemma *reachable-terms-one*:
assumes *t* ∈ *set* (*reachable-terms R s* (*Suc 0*))

shows $(s,t) \in (rstep\ (set\ R))\hat{}=$
⟨*proof*⟩

9.2.2 Algorithms to Ensure Joinability

definition
check-join-NF ::
$('f :: showl, 'v :: showl)\ rules \Rightarrow$
$('f, 'v)\ term \Rightarrow ('f, 'v)\ term \Rightarrow showsl\ check$
where
check-join-NF R s t ≡ *case* (*compute-rstep-NF R s*, *compute-rstep-NF R t*) *of*
(*Some s'*, *Some t'*) ⇒
check (*s'* = *t'*) (
showsl (*STR ''the normal form ''*) ∘ *showsl s'* ∘ *showsl* (*STR '' of ''*) ∘ *showsl s*
∘ *showsl* (*STR '' differs from*↩*the normal form ''*) ∘ *showsl t'* ∘ *showsl* (*STR '' of ''*) ∘ *showsl t*)
| - ⇒ *error* (*showsl* (*STR ''strange error in normal form computation''*))

lemma *check-join-NF-sound*:
assumes *ok*: *isOK* (*check-join-NF R s t*)
shows (*s*, *t*) ∈ *join* (*rstep* (*set R*))
⟨*proof*⟩

function *iterative-join-search-main* ::
$('f,'v)\ rules \Rightarrow ('f,'v)\ term \Rightarrow ('f,'v)\ term \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where
iterative-join-search-main R s t i n = (*if i* ≤ *n then*
(((*list-inter* (*reachable-terms R s i*) (*reachable-terms R t i*)) ≠ []) ∨ (*iterative-join-search-main R s t* (*Suc i*) *n*)) *else False*)
⟨*proof*⟩

termination ⟨*proof*⟩

lemma *iterative-join-search-main*:
iterative-join-search-main R s t i n ⟹ (*s*,*t*) ∈ *join* (*rstep* (*set R*))
⟨*proof*⟩

definition *iterative-join-search* **where**
iterative-join-search R s t n ≡ *iterative-join-search-main R s t 0 n*

lemma *iterative-join-search*: *iterative-join-search R s t n* ⟹ (*s*,*t*) ∈ *join* (*rstep* (*set R*))
⟨*proof*⟩

definition
check-join-BFS-limit ::
nat ⇒ $('f :: showl, 'v :: showl)\ rules \Rightarrow$

(′f, ′v) term ⇒ (′f, ′v) term ⇒ showsl check
where
check-join-BFS-limit n R s t ≡ check (iterative-join-search R s t n)
(showsl (STR ″could not find a joining sequence of length at most ″) ∘
showsl n ∘ showsl (STR ″ for the terms ″) ∘ showsl s ∘
showsl (STR ″ and ″) ∘ showsl t ∘ showsl-nl)

lemma *check-join-BFS-limit-sound*:
assumes *ok*: *isOK* (*check-join-BFS-limit n R s t*)
shows (*s*, *t*) ∈ *join* (*rstep* (*set R*))
⟨*proof*⟩

definition *map-funs-rules* :: (′f ⇒ ′g) ⇒ (′f, ′v) *rules* ⇒ (′g, ′v) *rules* **where**
map-funs-rules fg R = *map* (*map-funs-rule fg*) *R*

lemma *map-funs-rules-sound*[*simp*]:
set (*map-funs-rules fg R*) = *map-funs-trs fg* (*set R*)
⟨*proof*⟩

9.2.3 Displaying TRSs as Strings

fun *showsl-rule′* :: (′f ⇒ *showsl*) ⇒ (′v ⇒ *showsl*) ⇒ *String.literal* ⇒ (′f, ′v) *rule* ⇒ *showsl*
where
showsl-rule′ fun var arr (*l*, *r*) =
showsl-term′ fun var l ∘ *showsl arr* ∘ *showsl-term′ fun var r*

definition *showsl-rule* ≡ *showsl-rule′ showsl showsl* (*STR* ″ −> ″)
definition *showsl-weak-rule* ≡ *showsl-rule′ showsl showsl* (*STR* ″ −>= ″)

definition
showsl-rules′ :: (′f ⇒ *showsl*) ⇒ (′v ⇒ *showsl*) ⇒ *String.literal* ⇒ (′f, ′v) *rules* ⇒ *showsl*
where
showsl-rules′ fun var arr trs =
showsl-list-gen (*showsl-rule′ fun var arr*) (*STR* ″″′) (*STR* ″″″) (*STR* ″↩″) (*STR* ″″″) *trs* ∘ *showsl-nl*

definition *showsl-rules* ≡ *showsl-rules′ showsl showsl* (*STR* ″ −> ″)
definition *showsl-weak-rules* ≡ *showsl-rules′ showsl showsl* (*STR* ″ −>= ″)

definition
showsl-trs′ :: (′f ⇒ *showsl*) ⇒ (′v ⇒ *showsl*) ⇒ *String.literal* ⇒ *String.literal* ⇒ (′f, ′v) *rules* ⇒ *showsl*
where
showsl-trs′ fun var name arr R = *showsl name* ∘ *showsl* (*STR* ″↩|↩″) ∘ *showsl-rules′ fun var arr R*

definition *showsl-trs* ≡ *showsl-trs′ showsl showsl* (*STR* ″rewrite system:″) (*STR*

'' –> '')

9.2.4 Computing Syntactic Properties of TRSs

definition *add-vars-rule* :: ('*f*, '*v*) *rule* ⇒ '*v list* ⇒ '*v list*
where
add-vars-rule r xs = *add-vars-term* (*fst r*) (*add-vars-term* (*snd r*) *xs*)

definition *add-funs-rule* :: ('*f*, '*v*) *rule* ⇒ '*f list* ⇒ '*f list*
where
add-funs-rule r fs = *add-funs-term* (*fst r*) (*add-funs-term* (*snd r*) *fs*)

definition *add-funas-rule* :: ('*f*, '*v*) *rule* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
add-funas-rule r fs = *add-funas-term* (*fst r*) (*add-funas-term* (*snd r*) *fs*)

definition *add-roots-rule* :: ('*f*, '*v*) *rule* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
add-roots-rule r fs = *root-list* (*fst r*) @ *root-list* (*snd r*) @ *fs*

definition *add-funas-args-rule* :: ('*f*, '*v*) *rule* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
add-funas-args-rule r fs = *add-funas-args-term* (*fst r*) (*add-funas-args-term* (*snd r*) *fs*)

lemma *add-vars-rule-vars-rule-list-conv* [*simp*]:
add-vars-rule r xs = *vars-rule-list r* @ *xs*
⟨*proof*⟩

lemma *add-funs-rule-funs-rule-list-conv* [*simp*]:
add-funs-rule r fs = *funs-rule-list r* @ *fs*
⟨*proof*⟩

lemma *add-funas-rule-funas-rule-list-conv* [*simp*]:
add-funas-rule r fs = *funas-rule-list r* @ *fs*
⟨*proof*⟩

lemma *add-roots-rule-roots-rule-list-conv* [*simp*]:
add-roots-rule r fs = *roots-rule-list r* @ *fs*
⟨*proof*⟩

lemma *add-funas-args-rule-funas-args-rule-list-conv* [*simp*]:
add-funas-args-rule r fs = *funas-args-rule-list r* @ *fs*
⟨*proof*⟩

definition *insert-vars-rule* :: ('*f*, '*v*) *rule* ⇒ '*v list* ⇒ '*v list*
where
insert-vars-rule r xs = *insert-vars-term* (*fst r*) (*insert-vars-term* (*snd r*) *xs*)

definition *insert-funs-rule* :: $('f, 'v)$ *rule* $\Rightarrow$ $'f$ *list* $\Rightarrow$ $'f$ *list*
where
insert-funs-rule r fs = *insert-funs-term* (*fst r*) (*insert-funs-term* (*snd r*) *fs*)

definition *insert-funas-rule* :: $('f, 'v)$ *rule* $\Rightarrow$ $('f \times nat)$ *list* $\Rightarrow$ $('f \times nat)$ *list*
where
insert-funas-rule r fs = *insert-funas-term* (*fst r*) (*insert-funas-term* (*snd r*) *fs*)

definition *insert-roots-rule* :: $('f, 'v)$ *rule* $\Rightarrow$ $('f \times nat)$ *list* $\Rightarrow$ $('f \times nat)$ *list*
where
insert-roots-rule r fs =
foldr List.insert (*option-to-list* (*root* (*fst r*)) @ *option-to-list* (*root* (*snd r*))) *fs*

definition *insert-funas-args-rule* :: $('f, 'v)$ *rule* $\Rightarrow$ $('f \times nat)$ *list* $\Rightarrow$ $('f \times nat)$ *list*
where
insert-funas-args-rule r fs = *insert-funas-args-term* (*fst r*) (*insert-funas-args-term* (*snd r*) *fs*)

lemma *set-insert-vars-rule* [*simp*]:
set (*insert-vars-rule r xs*) = *vars-term* (*fst r*) ∪ *vars-term* (*snd r*) ∪ *set xs*
⟨*proof*⟩

lemma *set-insert-funs-rule* [*simp*]:
set (*insert-funs-rule r xs*) = *funs-term* (*fst r*) ∪ *funs-term* (*snd r*) ∪ *set xs*
⟨*proof*⟩

lemma *set-insert-funas-rule* [*simp*]:
set (*insert-funas-rule r xs*) = *funas-term* (*fst r*) ∪ *funas-term* (*snd r*) ∪ *set xs*
⟨*proof*⟩

lemma *set-insert-roots-rule* [*simp*]:
set (*insert-roots-rule r xs*) = *root-set* (*fst r*) ∪ *root-set* (*snd r*) ∪ *set xs*
⟨*proof*⟩

lemma *set-insert-funas-args-rule* [*simp*]:
set (*insert-funas-args-rule r xs*) = *funas-args-term* (*fst r*) ∪ *funas-args-term* (*snd r*) ∪ *set xs*
⟨*proof*⟩

abbreviation *vars-rule-impl r* ≡ *insert-vars-rule r* []
abbreviation *funs-rule-impl r* ≡ *insert-funs-rule r* []
abbreviation *funas-rule-impl r* ≡ *insert-funas-rule r* []
abbreviation *roots-rule-impl r* ≡ *insert-roots-rule r* []
abbreviation *funas-args-rule-impl r* ≡ *insert-funas-args-rule r* []

lemma *set-vars-rule-impl*:
set (*vars-rule-impl r*) = *vars-rule r*
⟨*proof*⟩

lemma *xxx-rule-list-code*[*code*]:
vars-rule-list r = *add-vars-rule r* []
funs-rule-list r = *add-funs-rule r* []
funas-rule-list r = *add-funas-rule r* []
roots-rule-list r = *add-roots-rule r* []
funas-args-rule-list r = *add-funas-args-rule r* []
⟨*proof*⟩

lemma *xxx-trs-list-code*[*code*]:
vars-trs-list trs = *foldr add-vars-rule trs* []
funs-trs-list trs = *foldr add-funs-rule trs* []
funas-trs-list trs = *foldr add-funas-rule trs* []
funas-args-trs-list trs = *foldr add-funas-args-rule trs* []
⟨*proof*⟩

definition *insert-vars-trs* :: ('*f*, '*v*) *rule list* ⇒ '*v list* ⇒ '*v list*
where
insert-vars-trs trs = *foldr insert-vars-rule trs*

definition *insert-funs-trs* :: ('*f*, '*v*) *rule list* ⇒ '*f list* ⇒ '*f list*
where
insert-funs-trs trs = *foldr insert-funs-rule trs*

definition *insert-funas-trs* :: ('*f*, '*v*) *rule list* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
insert-funas-trs trs = *foldr insert-funas-rule trs*

definition *insert-roots-trs* :: ('*f*, '*v*) *rule list* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
insert-roots-trs trs = *foldr insert-roots-rule trs*

definition *insert-funas-args-trs* :: ('*f*, '*v*) *rule list* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *nat*) *list*
where
insert-funas-args-trs trs = *foldr insert-funas-args-rule trs*

lemma *set-insert-vars-trs* [*simp*]:
set (*insert-vars-trs trs xs*) = (⋃ *r* ∈ *set trs*. *vars-rule r*) ∪ *set xs*
⟨*proof*⟩

lemma *set-insert-funs-trs* [*simp*]:
set (*insert-funs-trs trs fs*) = (⋃ *r* ∈ *set trs*. *funs-rule r*) ∪ *set fs*
⟨*proof*⟩

lemma *set-insert-funas-trs* [*simp*]:
set (*insert-funas-trs trs fs*) = (⋃ *r* ∈ *set trs*. *funas-rule r*) ∪ *set fs*
⟨*proof*⟩

lemma *set-insert-roots-trs* [*simp*]:

set (*insert-roots-trs trs fs*) = (⋃ *r* ∈ *set trs*. *roots-rule r*) ∪ *set fs*
⟨*proof*⟩

lemma *set-insert-funas-args-trs* [*simp*]:
set (*insert-funas-args-trs trs fs*) = (⋃ *r* ∈ *set trs*. *funas-args-rule r*) ∪ *set fs*
⟨*proof*⟩

abbreviation *vars-trs-impl trs* ≡ *insert-vars-trs trs* []
abbreviation *funs-trs-impl trs* ≡ *insert-funs-trs trs* []
abbreviation *funas-trs-impl trs* ≡ *insert-funas-trs trs* []
abbreviation *roots-trs-impl trs* ≡ *insert-roots-trs trs* []
abbreviation *funas-args-trs-impl trs* ≡ *insert-funas-args-trs trs* []

definition *defined-list* :: ('*f*, '*v*) *rule list* ⇒ ('*f* × *nat*) *list*
where
defined-list R = [*the* (*root l*). (*l*, *r*) ← *R*, *is-Fun l*]

lemma *set-defined-list* [*simp*]:
set (*defined-list R*) = {*fn*. *defined* (*set R*) *fn*}
⟨*proof*⟩

definition *check-left-linear-trs* :: ('*f* :: *showl*, '*v* :: *showl*) *rules* ⇒ *showsl check*
where
check-left-linear-trs trs =
check-all (λ*r*. *linear-term* (*fst r*)) *trs*
<+? (λ -. *showsl-trs trs* ∘ *showsl* (*STR* ''↩*is not left−linear*↩''))

lemma *check-left-linear-trs* [*simp*]:
isOK (*check-left-linear-trs R*) = *left-linear-trs* (*set R*)
⟨*proof*⟩

definition *check-varcond-subset* :: (-,-) *rules* ⇒ *showsl check*
where
check-varcond-subset R =
check-allm (λ*rule*.
check-subseteq (*vars-term-impl* (*snd rule*)) (*vars-term-impl* (*fst rule*))
<+? (λ*x*. *showsl* (*STR* ''*free variable* '') ∘ *showsl x*
∘ *showsl* (*STR* '' *in right−hand side of rule* '') ∘ *showsl-rule rule* ∘ *showsl-nl*)
) *R*

lemma *check-varcond-subset* [*simp*]:
isOK (*check-varcond-subset R*) = (∀ (*l*, *r*) ∈ *set R*. *vars-term r* ⊆ *vars-term l*)
⟨*proof*⟩

definition *check-varcond-no-Var-lhs* =
check-allm (λ*rule*.
check (*is-Fun* (*fst rule*))
(*showsl* (*STR* ''*variable left−hand side in rule* '') ∘ *showsl-rule rule* ∘ *showsl-nl*))

lemma *check-varcond-no-Var-lhs* [*simp*]:
isOK (*check-varcond-no-Var-lhs R*) ⟷ (∀ (*l*, *r*) ∈ *set R*. *is-Fun l*)
⟨*proof*⟩

definition *check-wf-trs* :: (-,-) *rules* ⇒ *showsl check*
where
check-wf-trs R = *do* {
check-varcond-no-Var-lhs R;
check-varcond-subset R
} <+? (λ*e*. *showsl* (*STR* ''*the TRS is not well−formed*↩''') ∘ *e*)

lemma *check-wf-trs-conf* [*simp*]:
isOK (*check-wf-trs R*) = *wf-trs* (*set R*)
⟨*proof*⟩

definition *check-not-wf-trs* :: (-,-) *rules* ⇒ *showsl check* **where**
check-not-wf-trs R = *check* (¬ *isOK* (*check-wf-trs R*)) (*showsl* (*STR* ''*The TRS is well formed*↩''))

lemma *check-not-wf-trs*:
assumes *isOK*(*check-not-wf-trs R*)
shows ¬ *SN* (*rstep* (*set R*))
⟨*proof*⟩

lemma *instance-rule-code*[*code*]:
instance-rule lr st ⟷ *match-list* (λ -. *fst lr*) [(*fst st*, *fst lr*), (*snd st*, *snd lr*)] ≠ *None*
(**is** *?l* = (*match-list ?d ?list* ≠ *None*))
⟨*proof*⟩

definition
check-CS-subseteq :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *rule check*
where
check-CS-subseteq R S ≡ *check-allm* (λ (*l*,*r*). *check* (*Bex* (*set S*) (*instance-rule* (*l*,*r*))) (*l*,*r*)) *R*

lemma *check-CS-subseteq* [*simp*]:
isOK (*check-CS-subseteq R S*) ⟷ *subst.closure* (*set R*) ⊆ *subst.closure* (*set S*)
(**is** *?l* = *?r*)
⟨*proof*⟩

definition *reverse-rules* :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *rules* **where**
reverse-rules rs ≡ *map prod.swap rs*

lemma *reverse-rules*[*simp*]: *set* (*reverse-rules R*) = (*set R*)^−1 ⟨*proof*⟩

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

where
map-funs-rules-wa fg R = map (λ(*l*, *r*). (*map-funs-term-wa fg l*, *map-funs-term-wa fg r*)) *R*

lemma *map-funs-rules-wa*: *set* (*map-funs-rules-wa fg R*) = *map-funs-trs-wa fg* (*set R*)
⟨*proof*⟩

lemma *wf-rule* [*code*]:
wf-rule r ⟷
is-Fun (*fst r*) ∧ (∀ *x* ∈ *set* (*vars-term-impl* (*snd r*)). *x* ∈ *set* (*vars-term-impl* (*fst r*)))
⟨*proof*⟩

definition *wf-rules-impl* :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *rules*
where
wf-rules-impl R = filter wf-rule R

lemma *wf-rules-impl* [*simp*]:
set (*wf-rules-impl R*) = *wf-rules* (*set R*)
⟨*proof*⟩

fun *check-wf-reltrs* :: (-,-) *rules* × (-,-) *rules* ⇒ *showsl check* **where**
check-wf-reltrs (*R*, *S*) = (*do* {
check-wf-trs R;
if R = [] *then succeed*
else check-varcond-subset S
})

lemma *check-wf-reltrs*[*simp*]:
isOK (*check-wf-reltrs* (*R*, *S*)) = *wf-reltrs* (*set R*) (*set S*)
⟨*proof*⟩

declare *check-wf-reltrs.simps*[*simp del*]

definition *check-linear-trs* :: (-,-) *rules* ⇒ *showsl check* **where**
check-linear-trs R ≡
check-all (λ (*l*,*r*). (*linear-term l*) ∧ (*linear-term r*)) *R*
<+? (λ -. *showsl-trs R* ∘ *showsl* (*STR* ''↩*is not linear*↩'')))

lemma *check-linear-trs* [*simp*]:
isOK (*check-linear-trs R*) ⟷ *linear-trs* (*set R*)
⟨*proof*⟩

definition *non-collapsing-impl R* = *list-all* (*is-Fun o snd*) *R*

lemma *non-collapsing-impl*[*simp*]: *non-collapsing-impl R* = *non-collapsing* (*set R*)
⟨*proof*⟩

type-synonym $('f, 'v)$ *term-map* $= 'f \times nat \Rightarrow ('f, 'v)$ *term list*

definition *term-map* :: $('f$::*compare-order*, $'v)$ *term list* $\Rightarrow ('f, 'v)$ *term-map* **where**
term-map ts = *fun-of-map* (*rm.*α (*elem-list-to-rm* (*the* $\circ$ *root*) *ts*)) []

definition
is-NF-main :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ $('f$::*compare-order*, $'v)$ *term-map* $\Rightarrow ('f, 'v)$ *term* $\Rightarrow$ *bool*
where
is-NF-main var-cond R-empty m = (*if var-cond then* (λ-. *False*)
else if R-empty then (λ-. *True*)
else (λt. $\forall$ *u*$\in$*set* (*supteq-list t*).
if is-Fun u then
$\forall$ *l*$\in$*set* (*m* (*the* (*root u*))). $\neg$ *matches u l*
else True))

lemma *neq-root-no-match*:
assumes *is-Fun l* **and** *the* (*root l*) $\neq$ *the* (*root t*)
shows $\neg$ *matches t l*
⟨*proof*⟩

lemma *all-not-conv*: $(\forall x \in A.\ \neg\ P\ x) = (\neg\ (\exists x \in A.\ P\ x))$ ⟨*proof*⟩

lemma *efficient-supteq-list-do-not-match*:
assumes $\forall$ *l*$\in$*set ls*. $\forall$ *u*$\in$*set* (*supteq-list t*). *the* (*root l*) $\neq$ *the* (*root u*) $\longrightarrow \neg$ *matches u l*
shows
($\forall$ *l*$\in$*set ls*. $\forall$ *u*$\in$*set* (*supteq-list t*). $\neg$ *matches u l*) $\longleftrightarrow$
($\forall$ *u*$\in$*set* (*supteq-list t*). $\forall$ *l*$\in$*set* (*term-map ls* (*the* (*root u*))).
$\neg$ *matches u l*)
(**is** *?lhs* $\longleftrightarrow$ *?rhs* **is** - $\longleftrightarrow$ ($\forall$ *u*$\in$*set ?subs*. $\forall$ *l*$\in$*set* (*?ls u*). $\neg$ *matches u l*))
⟨*proof*⟩

lemma *supteq-list-ex*:
($\exists$ *u*$\in$*set* (*supteq-list l*). $\exists \sigma.\ t \cdot \sigma = u$) $\longleftrightarrow$ ($\exists \sigma.\ l \trianglerighteq t \cdot \sigma$)
⟨*proof*⟩

definition *is-NF-trs R* = *is-NF-main* ($\exists$ *r*$\in$*set R*. *is-Var* (*fst r*)) (*R* = []) (*term-map* (*map fst R*))
definition *is-NF-terms Q* = *is-NF-main* ($\exists$ *q*$\in$*set Q*. *is-Var q*) (*Q* = []) (*term-map Q*)

lemma *is-NF-main-NF-trs-conv*:
is-NF-main ($\exists$ *r*$\in$*set R*. *is-Var* (*fst r*)) (*R* = []) (*term-map* (*map fst R*)) *t* $\longleftrightarrow$
t $\in$ *NF-trs* (*set R*)
(**is** *is-NF-main ?var ?R ?map t* $\longleftrightarrow$ -)
⟨*proof*⟩

lemma *is-NF-trs* [*simp*]:
is-NF-trs R = (λ *t*. *t* ∈ *NF-trs* (*set R*))
⟨*proof*⟩

lemma *is-NF-terms* [*simp*]:
is-NF-terms Q = (λ *t*. *t* ∈ *NF-terms* (*set Q*))
⟨*proof*⟩

9.2.5 Grouping TRS-Rules by Function Symbols

type-synonym ('*f*,'*v*)*rule-map* = (('*f* × *nat*) ⇒ ('*f*,'*v*)*rules*)*option*

fun *computeRuleMapH* :: ('*f*,'*v*)*rules* ⇒ (('*f* × *nat*) × ('*f*,'*v*)*rules*)*list option*
where *computeRuleMapH* [] = *Some* []
| *computeRuleMapH* ((*Fun f ts*,*r*) # *rules*) = (*let n* = *length ts in case computeRuleMapH rules of None* ⇒ *None* | *Some rm* ⇒
(*case List.extract* (λ (*fa*,*rls*). *fa* = (*f*,*n*)) *rm of*
None ⇒ *Some* (((*f*,*n*), [(*Fun f ts*,*r*)]) # *rm*)
| *Some* (*bef*,(*fa*,*rls*),*aft*) ⇒ *Some* ((*fa*,(*Fun f ts*,*r*) # *rls*) # *bef* @ *aft*)))
| *computeRuleMapH* ((*Var* -, -) # *rules*) = *None*

definition *computeRuleMap* :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *rule-map* **where**
computeRuleMap rls ≡
(*case computeRuleMapH rls of*
None ⇒ *None*
| *Some rm* ⇒ *Some* (λ*f*.
(*case map-of rm f of*
None ⇒ []
| *Some rls* ⇒ *rls*)))

lemma *computeRuleMapHSound2*: (*computeRuleMapH R* = *None*) = (∃ (*l*, *r*) ∈ *set R*. *root l* = *None*)
⟨*proof*⟩

lemma *computeRuleMapSound2*: (*computeRuleMap R* = *None*) = (∃ (*l*, *r*) ∈ *set R*. *root l* = *None*)
⟨*proof*⟩

lemma *computeRuleMapHSound*: **assumes** *computeRuleMapH R* = *Some rm*
shows (λ (*f*,*rls*). (*f*,*set rls*)) ' *set rm* = {((*f*,*n*),{(*l*,*r*) | *l r*. (*l*,*r*) ∈ *set R* ∧ *root l* = *Some* (*f*, *n*)}) | *f n*. {(*l*,*r*) | *l r*. (*l*,*r*) ∈ *set R* ∧ *root l* = *Some* (*f*, *n*)} ≠ {}} ∧ *distinct-eq* (λ (*f*,*rls*) (*g*,*rls*'). *f* = *g*) *rm*
⟨*proof*⟩

lemma *computeRuleMapSound*:
assumes *computeRuleMap R* = *Some rm*

shows (*set* (*rm* (*f*,*n*))) = {(*l*,*r*) | *l r*. (*l*,*r*) ∈ *set R* ∧ *root l* = *Some* (*f*, *n*)}
⟨*proof*⟩

lemma *computeRuleMap-left-vars*:
shows (*computeRuleMap R* ≠ *None*) = (∀ *lr* ∈ *set R*. ∀ *x*. *fst lr* ≠ *Var x*)
⟨*proof*⟩

lemma *computeRuleMap-defined*: **fixes** *R* :: ('*f*,'*v*)*rules*
assumes *computeRuleMap R* = *Some rm*
shows (*rm* (*f*,*n*) = []) = (¬ *defined* (*set R*) (*f*,*n*))
⟨*proof*⟩

lemma *computeRuleMap-None-not-SN*:
assumes *computeRuleMap R* = *None*
shows ¬ *SN-on* (*rstep* (*set R*)) {*t*}
⟨*proof*⟩

end

9.3 Implementation of Parallel Rewriting With Variable Restriction

theory *Rewrite-Relations-Impl*
imports
Trs-Impl
Parallel-Rewriting
Multistep
begin

9.3.1 Checking a Single Parallel Rewrite Step with Variable Restriction

context
fixes *R* :: ('*f*,'*v*)*rules* **and** *V* :: '*v set*
begin
fun *is-par-rstep-var-restr* :: ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term* ⇒ *bool*
where
is-par-rstep-var-restr (*Fun f ss*) (*Fun g ts*) =
(*Fun f ss* = *Fun g ts* ∨
vars-term (*Fun g ts*) ∩ *V* = {} ∧ (*Fun f ss*, *Fun g ts*) ∈ *rrstep* (*set R*) ∨
(*f* = *g* ∧ *length ss* = *length ts* ∧ *list-all2 is-par-rstep-var-restr ss ts*))
| *is-par-rstep-var-restr s t* = (*s* = *t* ∨ *vars-term t* ∩ *V* = {} ∧ (*s*,*t*) ∈ *rrstep* (*set R*))

lemma *is-par-rstep-code-helper*: *vars-term t* ∩ *V* = {} ⟷
(∀ *x* ∈ *set* (*vars-term-list t*). *x* ∉ *V*)

⟨*proof*⟩

lemmas *is-par-rstep-var-restr-code*[*code*] = *is-par-rstep-var-restr.simps*[*unfolded is-par-rstep-code-helper*]

lemma *is-par-rstep-var-restr*[*simp*]:
is-par-rstep-var-restr s t ⟷ (*s*, *t*) ∈ *par-rstep-var-restr* (*set R*) *V*
⟨*proof*⟩
end

lemma *par-rstep-var-restr-code*[*code-unfold*]:
(*s*, *t*) ∈ *par-rstep-var-restr* (*set R*) *V* ⟷ *is-par-rstep-var-restr R V s t*
⟨*proof*⟩

9.4 Implementation of Parallel Rewriting

9.4.1 Checking a Single Parallel Rewrite Step

fun *is-par-rstep* :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term* ⇒ *bool*
where
is-par-rstep R (*Fun f ss*) (*Fun g ts*) =
(*Fun f ss* = *Fun g ts* ∨ (*Fun f ss*, *Fun g ts*) ∈ *rrstep* (*set R*) ∨
(*f* = *g* ∧ *length ss* = *length ts* ∧ *list-all2* (*is-par-rstep R*) *ss ts*))
| *is-par-rstep R s t* = (*s* = *t* ∨ (*s*,*t*) ∈ *rrstep* (*set R*))

lemma *is-par-rstep*[*simp*]:
is-par-rstep R s t ⟷ (*s*, *t*) ∈ *par-rstep* (*set R*)
⟨*proof*⟩

lemma *par-rstep-code*[*code-unfold*]: (*s*, *t*) ∈ *par-rstep* (*set R*) ⟷ *is-par-rstep R s t* ⟨*proof*⟩

9.4.2 Generate All Parallel Rewrite Steps

fun *root-rewrite* :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term list*
where
root-rewrite R s = *concat* (*map* (λ (*l*, *r*).
(*case match s l of*
None ⇒ []
| *Some σ* ⇒ [(*r* · *σ*)])) *R*)

lemma *root-rewrite-sound*:
assumes *t* ∈ *set* (*root-rewrite R s*)
shows (*s*, *t*) ∈ *rrstep* (*set R*)
⟨*proof*⟩

Generate all possible parallel rewrite steps for a given term, assuming that the underlying TRS is well-formed.

fun *parallel-rewrite* :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term list*
where
parallel-rewrite R (*Var x*) = [*Var x*]

| *parallel-rewrite R (Fun f ss) = remdups*
(root-rewrite R (Fun f ss) @ map (λss. Fun f ss) (product-lists (map (parallel-rewrite R) ss)))

lemma *parallel-rewrite-par-step*:
assumes *t ∈ set (parallel-rewrite R s)*
shows *(s, t) ∈ par-rstep (set R)*
⟨*proof*⟩

9.5 Implementation of Multi-Step Rewriting

9.5.1 Checking a Single Multi-Step Rewrite

fun *root-steps-substs* :: *('f, 'v) rules ⇒ ('f, 'v) term ⇒ ('f, 'v) term ⇒ (('f, 'v) term list × ('f, 'v) term list) list*
where
root-steps-substs R s t = concat (map (λ (l, r).
(case match s l of
None ⇒ []
| Some σ ⇒ (case match t r of
None ⇒ []
| Some τ ⇒ (let var-list = filter (λx. x ∈ vars-term r) (vars-distinct l) in [(map σ var-list, map τ var-list)])))) R)

lemma *root-steps-substs-exists*:
assumes *(ss, ts) ∈ set (root-steps-substs R s t)*
shows *∃ l r σ τ vl. (l, r) ∈ set R ∧ vl = filter (λx. x ∈ vars-term r) (vars-distinct l) ∧*
l · σ = s ∧ r · τ = t ∧ (ss, ts) = (map σ vl, map τ vl)
⟨*proof*⟩

lemma *size-match-subst-Fun*:
assumes *is-Fun l* **and** *x ∈ vars-term l*
and *match:match s l = Some τ*
shows *size (τ x) < size s*
⟨*proof*⟩

abbreviation *remove-trivial-rules R ≡ filter (λ (l, r). ¬ (is-Var l) ∨ ¬ (is-Var r)) R*

lemma *trivial-rrstep*:
assumes *∃ x y. (Var x, Var y) ∈ R ∧ x ≠ y*
shows *(s, t) ∈ rrstep R*
⟨*proof*⟩

lemma *size-root-steps-substs*:
assumes *(ss, ts) ∈ set (root-steps-substs (remove-trivial-rules R) s t)*
and *s' ∈ set ss t' ∈ set ts*
shows *size s' + size t' < size s + size t*

⟨*proof*⟩

function (*sequential*) *is-mstep* :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term* ⇒ *bool*
where
is-mstep R (*Fun f ss*) (*Fun g ts*) =
(*Fun f ss* = *Fun g ts* ∨ (*Fun f ss*, *Fun g ts*) ∈ *rrstep* (*set R*) ∨
list-ex (λ (*ss*, *ts*). *list-all2* (*is-mstep R*) *ss ts*) (*root-steps-substs* (*remove-trivial-rules R*) (*Fun f ss*) (*Fun g ts*)) ∨
(*f* = *g* ∧ *length ss* = *length ts* ∧ *list-all2* (*is-mstep R*) *ss ts*))
| *is-mstep R s t* = (*s* = *t* ∨ (*s*, *t*) ∈ *rrstep* (*set R*) ∨
list-ex (λ (*ss*, *ts*). *list-all2* (*is-mstep R*) *ss ts*) (*root-steps-substs* (*remove-trivial-rules R*) *s t*))
⟨*proof*⟩

termination ⟨*proof*⟩

Show that all multi-steps are covered by the definition above.

lemma *mstep-is-mstep*:
assumes (*s*, *t*) ∈ *mstep* (*set R*)
shows *is-mstep R s t*
⟨*proof*⟩

lemma *mstep-root-helper*:
assumes *list-ex* (λ (*ss*, *ts*). *list-all2* (*is-mstep R*) *ss ts*) (*root-steps-substs* (*remove-trivial-rules R*) *s t*)
and ⋀ *ss ts s' t'*. (*ss*, *ts*) ∈ *set* (*root-steps-substs* (*remove-trivial-rules R*) *s t*) ⟹ *s'* ∈ *set ss* ⟹ *t'* ∈ *set ts* ⟹ *is-mstep R s' t'* ⟹ (*s'*, *t'*) ∈ *mstep* (*set R*)
shows (*s*, *t*) ∈ *mstep* (*set R*)
⟨*proof*⟩

lemma *is-mstep-mstep*:
assumes *is-mstep R s t*
shows (*s*, *t*) ∈ *mstep* (*set R*)
⟨*proof*⟩

lemma *is-mstep*[*simp*]:
is-mstep R s t ⟷ (*s*, *t*) ∈ *mstep* (*set R*)
⟨*proof*⟩

lemma *mstep-code*[*code-unfold*]: (*s*, *t*) ∈ *mstep* (*set R*) ⟷ *is-mstep R s t* ⟨*proof*⟩

9.5.2 Generate All Multi-Step Rewrites

fun *root-subst-with-rhs* :: ('*f*, '*v*) *rules* ⇒ ('*f*, '*v*) *term* ⇒ (('*f*, '*v*) *term* × ('*f*, '*v*) *term list*) *list*
where
root-subst-with-rhs R s = *concat* (*map* (λ (*l*, *r*).
(*case match s l of*

None ⇒ []
| *Some* σ ⇒ [(*r*, *map* σ (*vars-distinct r*))]))
R)

lemma *root-steps-subst-rhs-exists*:
assumes (*r*, *ss*) ∈ *set* (*root-subst-with-rhs R s*)
shows ∃ *l* σ. (*l*, *r*) ∈ *set R* ∧ *l* · σ = *s* ∧ *ss* = *map* σ (*vars-distinct r*)
⟨*proof*⟩

context
fixes *R* :: ('*f*, '*v*) *rules*
assumes *wf-trs* (*set R*)
begin

private lemma ∗: *list-all* (λ(*l*, *r*). *is-Fun l* ∧ (*vars-term r* ⊆ *vars-term l*)) *R*
⟨*proof*⟩

lemma *varcond*:
⋀*l r*. (*l*, *r*) ∈ *set R* ⟹ *is-Fun l* ∧ *vars-term r* ⊆ *vars-term l*
⟨*proof*⟩

lemma [*termination-simp*]:
assumes (*l*, *r*) ∈ *set R*
and *Some* σ = *match* (*Fun g ts*) *l*
and *x* ∈ *vars-term r*
shows *size* (σ *x*) < *Suc* (*size-list size ts*)
⟨*proof*⟩

Compute the list of terms reachable in multi-step from a given term.

fun *mstep-rewrite-main* :: ('*f*, '*v*) *term* ⇒ ('*f*, '*v*) *term list*
where
mstep-rewrite-main (*Var x*) = [*Var x*]
| *mstep-rewrite-main* (*Fun f ss*) = *remdups* (
(*concat* (*map* (λ(*r*, *ts*).
(*map* (λ*args*. *r* · (*mk-subst Var* (*zip* (*vars-distinct r*) *args*))) (*product-lists* (*map mstep-rewrite-main ts*))))
(*root-subst-with-rhs R* (*Fun f ss*))))
@(*map* (λ*ss*. *Fun f ss*) (*product-lists* (*map mstep-rewrite-main ss*))))

lemma *mstep-rewrite-main-mstep*:
assumes *t* ∈ *set* (*mstep-rewrite-main s*)
shows (*s*, *t*) ∈ *mstep* (*set R*)
⟨*proof*⟩

end

We need to be able to export code for *mstep-rewrite-main*, hence the following definitions.

typedef ('*f*, '*v*) *wfTRS* = {*R* :: ('*f*, '*v*) *rules*. *wf-trs* (*set R*)}

⟨*proof*⟩

setup-lifting *type-definition-wfTRS*

lift-definition *get-TRS* :: ('*f*, '*v*) *wfTRS* ⇒ ('*f*, '*v*) *rules* **is** λ *R*. *R* ⟨*proof*⟩

lemma *is-wf-get-TRS*: *wf-trs* (*set* (*get-TRS R'*))
⟨*proof*⟩

definition *mstep-rewrite-wf R* = *mstep-rewrite-main* (*get-TRS R*)

lemmas *mstep-rewrite-wf-simps* = *mstep-rewrite-main.simps*[*OF is-wf-get-TRS*, *folded mstep-rewrite-wf-def*]
declare *mstep-rewrite-wf-simps*[*code*]

lift-definition (*code-dt*) *get-wfTRS* :: ('*f* :: *showl*, '*v* :: *showl*) *rules* ⇒ ('*f*, '*v*) *wfTRS option* **is**
λ *R*. *if isOK* (*check-wf-trs R*) *then Some R else None*
⟨*proof*⟩

definition *err-wf* **where** *err-wf* = *STR ''TRS is not well−formed''*

definition *mstep-dummy-impl R s t* = ((*s*,*t*) ∈ *mstep* (*set R*))
lemma *mstep-dummy-impl*[*code*]: *mstep-dummy-impl R* = *Code.abort* (*STR ''mstep-dummy''*) (λ -. *mstep-dummy-impl R*)
⟨*proof*⟩

lift-definition (*code-dt*) *get-wfTRS-sub* :: ('*f* :: *showl*, '*v* :: *showl*) *rules* ⇒ ('*f*, '*v*) *wfTRS* **is**
λ *R*. *if isOK* (*check-wf-trs R*) *then R else Code.abort err-wf* (λ -. [])
⟨*proof*⟩

definition *mstep-rewrite R* = *mstep-rewrite-wf* (*get-wfTRS-sub R*)

lemma *mstep-rewrite-mstep*:
assumes *t* ∈ *set* (*mstep-rewrite R s*)
shows (*s*, *t*) ∈ *mstep* (*set R*)
⟨*proof*⟩

end

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] TeReSe, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[3] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher-Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009.